POLITECNICO DI TORINO

Master Course in Mechatronic Engineering

Master of Science Thesis

Control of 2 DoF Robotic Arm Using Matlab and Simulink
Design, Simulation and Control of 3 DoF Robotic Arm Using ROS

Advisor:
Prof. Marina Indri

Author:
Ali Al Zouzou

External advisor at Universidad Carlos III de Madrid:
Prof. Ramon Barber Castaño

December 2018

Contents

List of Figures

List of Tables

Acknowledgement

Before starting my Introduction, i would like to thank my supervisors **Prof. Ramon Barber Castaño** at Universidad Carlos III de Madrid and **Prof. Marina Indri** at Politecnico di Torino for introducing me to this project and for their guidance during this graduation research.

Prof. Ramon Barber Castaño i want to thank for his work as my daily supervisor. Also i would like to thank **Prof. Dorin-sabin copaci** for his useful comments on the project.

Abstract

This work presents a methodology of design and control in real time using low cost hardware and working with a real robot and its simulated model in a coordinated way.
Since the robot was already built, the methodology starts in the control phase using *Matlab*® and *Simulink*® as integration platfroms, which are widely used in engineering design and control tools.
As robotic prototype, the robot **SIDEMAR** of 2 degrees of freedom is used, which has been designed as a service robot prototype.
In addition, the second step of the project will present a 3 DoF robotic arm designed and build totally from the beginning. The methodology of the second robot starts with the design phase using 3D CAD tool Solidworks which is mostly used as a design interface for mechanical projects. However, as a second phase, the simulation and control of the 3 DoF robotic arm have been done using ROS (Robotic Operating System), which is the most powerful tool for simulating and analysing the behaviour of a designed robot even before building the real prototype.

Chapter 1

Introduction

1.1 Background

A robotic arm is a mechanical structure designed in a similar way as human arm, it is usually programmable to replace the functionality of human arms in many different fields such as services or industrial fields. The robot arm might be a full mechanism itself or a part of a more complex robots (ex. Humanoid Robots). The manipulator are made up by links, the links of the manipulator are connected by joints (Motors), the joints can be revolute joints (type R joints) that perform rotational motions or linear joints (type L joints) that perform linear displacement motions. Usually each robotic arm has an end-effector at its end, the end-effector has a similar functionality to the human hand that is ready to do several tasks in the service fields such as gripping, placing, cleaning, and other tasks, and in the industrial fields such as welding, assembling, painting. For example, robot arms in automotive assembly lines perform a variety of tasks such as welding and placement during assembly.
Nowadays, robots have gained an increasing of interests from the research areas and industries due to the benefits and facilities they provide to the workspace. Such robot are programmed to continuously perform changing their tasks in unstructured human environments. In coexistence way, when the robot and the human share the same workspace, and in cooperation way, when the robot and the human share the same tasks[16].
As robotic arms are able to work autonomously, they require wide acknowledgement in many different engineering fields such as *mechanical engineering*,

electronics engineering, *computer science* and many others.
Based on these technologies, the robotic arms in this project have been built for study cases of actuators, sensors and control. The arms have been designed to be service robots for the welfare benefits, excluding manufacturing operations, so it is a system that cannot be used in the industries or product manufacturing.
The following chapters will explain the reason behind working on these projects and will also reveal the objectives and the stages of control work and development process followed during the working time. Finally, the parts of this document will comment how the works carried out are organized.

1.2 Motivation

This research is based upon the work on control of 2 degrees of freedom robotic arm already designed and assembled using different sensors and different controllers that potentially can be used as a service robotic arm for multi-applications and tasks.
For the same purpose a new robot will be designed and controlled with 3DOF to perform a more complex tasks and operations. This work was developed within the Department of System Engineering and Automation at *Universidad Carlos III de Madrid - Spain* and with the collaboration of *Politecnico di Torino - Italy*.
The main idea arises from the need to measure the variations of the angle of the terminal part of the robotic arm constituted by two metal(Aluminium) links, each driven by a DC servomotor.
For this purpose, it is necessary to use the appropriate electronics able to perform these measurements easily and reliably.
The sensors that were used to this project were *9DOF Razor IMU* sensor equipped with gyroscope that allows measuring the variation of the angles of the system. Moreover, the sensor must be connected to a microcontroller that will be the interface between the process and the monitoring software.
The first available choice was the use of the low-cost STM32F4 Discovery microcontroller that supports up to 140 I/O ports and however its price is more or less 23 euros. The second choice is the use of Arduino Mega 2560 that has 54 digital I/O pins and up to 16 Analog inputs and its price is more or less 50 euros. both controllers offer integration with the chosen software tool (*Matlab*® **and** *Simulink*®).
Matlab-Simulink were chosen as the software to interact with the hardware, since they offer a high level language and an interactive environment for numerical calculation, visualization and programming, the language, tools

available in this software allow us to find different solutions for various cases. Moreover, this software can be used in a wide variety of application, such as signal processing and communications, image and video processing, control systems, tests and measurements. However, the use of *Simulink*® tool, an environment of block diagrams are available for simulations and model-based design. it supports design and level simulation systems, automatic code generation and continuous testing and verifications of embedded-systems. In addition, it offers graphical representations of the results simulation to carry out more analysis.
In addition, ROS (Robot operating system) has been used to simulate and control the 3 DoF robotic arm. This software is a powerful tool that allows analysing the behaviour of the designed robot in real simulations. This way we can ensure safety for the human since we can totally know how the behaviour of the robot will be in real-time. However, ROS supports graphical interpreting of the results and it provides the user with all the required details regarding poses, velocities and actions of the robot.

1.3 Objectives

The main objectives are listed below:

1. Study of hardware possibilities for the control of 2 DoF robotic arm using Matlab *Simulink*®
    - Study the possibility of using Arduino Mega 2560 with *Simulink*® to control the 2 DoF robotic arm.
2. Study of control strategies for the control of 2 DoF robotic arm.
3. Design and construction of a 3 DoF robotic arm using Solidworks.
4. Simulations and Control of the 3 DoF robotic arm using ROS (Robot Operating System) and Arduino.

1.4 Steps of the Work

The thesis work is composed of two main parts, the first part is about the control of a 2 DoF robotic arm called SIDEMAR robot already existing in the Robotic Lab at Universidad Carlos III de Madrid. The first step of work was to control the robot arm using a Simulink model done by a previous student and STM32f4 Discovery microcontroller with some changes regarding the hardware connections and the data type of the system's input/output. The first step has been done to make sure that the robot hardware is working properly; during the first step we found out that the driver of the lower motor was not working, so we had to do the work considering the upper motor because eventually the control work should be done for educational purposes. The second step of work is about the use of an Arduino Mega 2560 and Simulink Matlab in order to control the SIDEMAR robot with different control strategies (PI, PID, Fuzzy Logic Control and Adaptive controller). The first point in the second step is to read the sensors; two sensors have been used in the system, the first one is 12-bit programmable magnetic position sensor responsible for measuring the angular position of the robot's motor, and the second sensor is 9DOF Razor IMU to measure the angular velocity of robot's links. Then after reading the sensors, the first control strategy has been done using Simulink standard PID controller. However, to check the effect of the Derivative term in practice, a designed PI controller using Simulink blocks has been done, and finally the results of both controllers have been recorded and compared. The third control strategy has been done using Fuzzy Logic Control; the same control loop of PID controller has been used but the PID controller was replaced by Fuzzy Logic Control Simulink block, the Fuzzy Logic Control is programmed logically according to the error signal e(t) of the system. The last control strategy has considered the Adaptive Control. At the beginning, the problem of Adaptive control in Simulink was storing samples of the input u(t) and the output y(t) periodically in an array, because the stored samples have to be used in the least square code (MATLAB, C, C + + or FORTRAN) written within a Simulink block called S-function. The solution was to use Recursive Least Square Estimator Simulink block that receives the input and output of the plant and produce the values that define the control parameters. The duration of the first part of the work was almost three Months (March 2018 to May 2018).
In the second part of the work, the idea was to design a 3 DoF robotic arm. Firstly, a quick study of the robot structure and dimensions has been done on the papers. Then, 3D design of the robot has been done using Solidworks. Solidworks allows us to see the movement of the robot's links when the complete assembly of the robot is ready. Moreover, when the design of the

robot was satisfying, the robot parts have been sent to the Laboratory of Materials at Universidad Carlos III de Madrid for execution. Meanwhile, a study for integrating and simulating the robot arm in ROS (Robot Operating System) has been done. To do the simulation, the real simulator Gazebo has been used, and this work has been done by connecting Gazebo with ROS to control the robot. For the control of the robot on Gazebo, a standard PID controller has been used. Furthermore, Moveit! has been used in order to do the motion planning of the robotic arm. After that, the robot hardware was totally available in the Robotic Lab, so the step was to build up the real robot and to connect the Arduino Mega 2560 microcontroller with the motors and drivers. The last step of the work was to control the real robot using ROS and Arduino IDE; since we were out of time, a fast open loop control has been done to make sure that the robot's connections are correctly done and the control works. The duration of the second part of the work was almost 3 months (June 2018 to August 2018).

Chapter 2

General Description of SIDEMAR Robot

This chapter will briefly discuss the elements that make up the system and the characteristic of the software used.

2.1 Hardware Description

2.1.1 Description of SIDEMAR Robot

The SIDEMAR robot exists in the Robotic Lab at Universidad Carlos III de Madrid has been designed and implemented in a way that allows its usage as a service robot. However, the robot's material has been carefully chosen in order to keep the robot's weight light but at the same time it should have a high resistance to external forces that the robot could expose to. Moreover, the experimental platform was designed to carry out direct, inverse and dynamic studies.
To satisfy the characteristics, the used material that build up the links of the robot was Aluminium Alloy, because it has a reduced weight compared to other materials in the market. However, its stiffness allows it to handle external forces and the assembly of the robot can be implemented and operated easily since this material has a sweet metallic structure which makes it a good material to deal with.
The robot is divided into three main parts, and these parts assembled together to perform the final platform [1].

1. **Base Link:** The base of the robot is fixed on a table. It is intended to hold the entire robot (Lower Link and Upper Link) with their motors. However, the base is divided into two parts, the part that is fixed on the table, this part has been used to carry and resist all the external forces that the robot could be subjected to, and a part that holds the Lower Link's motor of the robot [1].

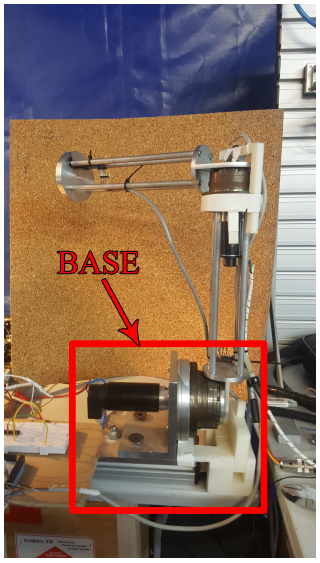


Figure 2.1: Base of SIDEMAR Robot

2. **Lower Link:** The Lower Link is made by two circular Aluminium plates connected by 4 hollow Aluminium tubes. The DC motor of the Lower Link is found on its lower end and it is docked to the base of the robot; and on its higher end, the DC motor of the Higher Link is fixed. The lower link has a restriction in its functionality which it can't rotate 360°, this makes the control of the lower arm even more difficult. The robot's lower link is shown in Figure 2.2 [1].

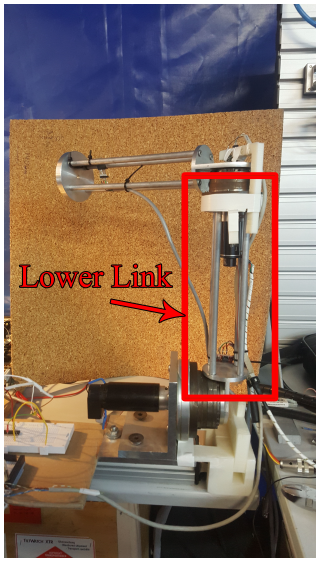


Figure 2.2: Lower Link of SIDEMAR Robot

3. **Higher Link:** The higher link is also made up by two circular Aluminium plates connected by 4 hollow Aluminium to reduce the robot's weight. The DC motor that controls the rotation of the higher link is found on its first end and docked on the higher end of the Lower Link, and the last end is set to be free. The higher link is shown in Figure 2.3 [1].

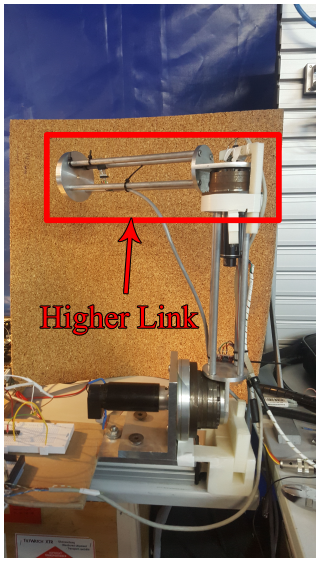


Figure 2.3: Higher Link of SIDEMAR Robot

2.1.2 STM32f4 Discovery Micro-controller

STM32f4 Discovery belongs to the family of low-cost micro-controllers, based on ARM Cortex -M4 32-bit RISC high performance core operates at frequency up to 168 MHz. This micro-controller also features Floating Point Unit (FPU) single precision which supports all ARM single precision data-processing instructions and data types [2].

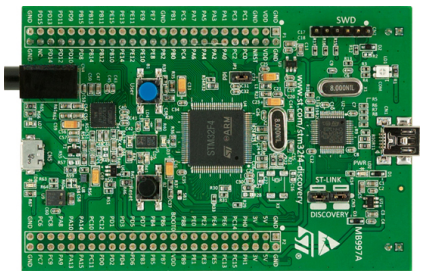


Figure 2.4: STM32f4 Discovery micro-controller

However, this controller incorporates high-speed embedded memories (Flash memory up to 1 Mbyte, up to 192 Kbytes of SRAM), up to 4 Kbytes of backup SRAM, which is very applicable for our application to run the model that contains the **SimMechanics Model** that requires a large hardware memory when the system runs in *External Mode*. Moreover, it offers an extensive

range of enhanced I/Os. [2]

Peripheral	Features
Operating Voltage	1.8V to 3.6V
Flash Memory	1 Mbyte
Communication interfaces	USB, I2C, I2S, CAN, SPI
Cristal Oscillator	32 KHz + 4 to 26 MHz

Table 2.1: Main characteristics of STM32F4 Discovery micro-controller

2.1.3 Arduino Mega 2560 microcontroller

Arduino Mega 2560 also belongs to the family of low-cost micro-controllers. Based on the ATmega2560, it features a communications with other micro-controllers and with computers. Arduino Mega 2560 offers high number of facilities regarding the large number of digital and PWM (Pulse With Modulation) I/Os available in it. However, it differs from all preceding micro-controllers in that it doesn't use the FTDI USB-to-Serial driver chip. instead, it features the ATmega8U2 programmed as USB-to-Serial converter (Arduino Mega Manual [3]).

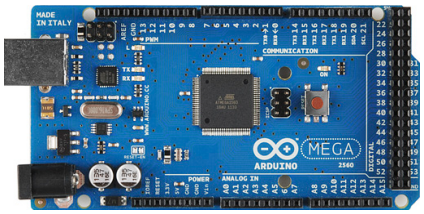


Figure 2.5: Adruino Mega 2560 micro-controller

Moreover, Arduino has enough information available and offers integration with chosen software **Matlab-Simulink**. On contrary, the amount of memory available in this controller (256 Kbyte) makes some restrictions regarding the *simMechanics* model available in the complete model of the system since it requires a large amount of memory (Arduino Mega Manual [3]).

Peripheral	Features
Operating Voltage	5V
Input Voltage(Limits)	6-20 V
Digital I/O pins	54(of which 14 provide PWM output)
Analog Input Pins	16
Flash Memory	256 Kbyte of which 8 Kbyte used by Bootloader
SRAM	8 Kbyte
EEPROM	4 Kbyte
Clock Speed	16 MHz

Table 2.2: Main Characteristics of Arduino Mega 2560 microcontroller

2.1.4 Driver

The driver used is MCDC2805, it was designed for FAULHABER DC micromotors. In combination with the reliable IE2-512 encoder, positioning resolution is up to 0.18 degrees can be achieved even at very low speeds. The motion controller is based on a powerful 16-bit micro-controller with excellent filtering quality (MCDC 2805 Datasheets).

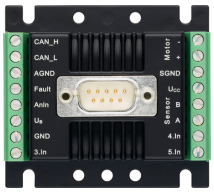


Figure 2.6: FAULHABER MCDC2805 motor driver

This intelligent motion controller performs the following tasks:

1. Velocity Control
2. Velocity Profiles
    - Ramping, triangle, trapezoidal and more complicated velocity profiles
3. Positioning Mode
4. Torque controlling
5. Saving and running program sequences

2.1.5 Sensors

12-bit Programmable Magnetic Position Sensor AS5045B

As a general description of this sensor, this sensor measures the angular position through a contactless magnetic position sensor. It is able to measure angular positions for a full revolution of 360° [4].
The measurements of the absolute angles provide a current indications of the angular positions of magnets with 4096 positions per resolution. The digital data is available as a PWM signal that can be read directly by the used micro-controller [4].
This sensor is compatible for usage in several applications such as elevators, robotics, automation, motor control and optical encoder replacement.

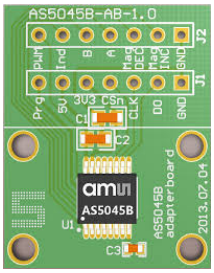


Figure 2.7: 12-bit programmable Magnetic Position Sensor AS5045B

9DOF Razor IMU

To achieve the correct measurements of the angles of rotation, the free end of the Upper link and the base docked end of lower link have two Razor IMU sensors connected.

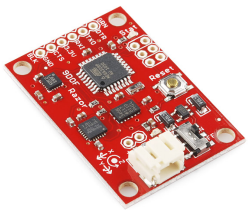


Figure 2.8: 9DOF Razor IMU sensor

The 9DOF Razor IMU incorporates three sensors - an ITG-3200 (system 3-axis electromechanical micro gyroscope), ADXL345(3-axis accelerometer) and HMC5883L(Triple-axis magnetometer) to give nine measurements degrees inertial. The output of all the sensors are processed by an ATmega328 of a board and output through a serial interface (9 Degrees of Freedom - Razor IMU Datasheet).

2.2 Software Description

2.2.1 Matlab

Matlab® *figure 2.9* is a high performance technical calculation environment for numerical calculation and visualization. It integrates numerical analysis, matrix calculation, signal processing and graphics in an easy-to-use environment, where the problems and the solutions are expressed as they are written mathematically, without the traditional programming[5].
With this software we are able to communicate with different hardware and to analyse, calculate and find different solutions to different problems numerically and in a very easy and interesting way [5].

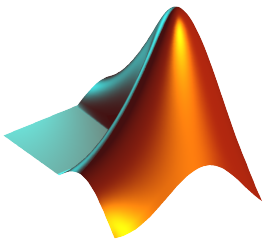


Figure 2.9: *Matlab*® Logo

2.2.2 Simulink

In this project, *Simulink*® is the working interface and the main programming software used. It allows us to simulate, generate automatic codes, continuous testing and verify our embedded systems. Also it is a data flow graphical programming tool for modelling and simulations of multi-domain dynamics systems[6].

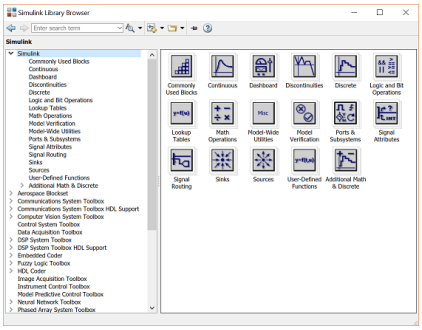


Figure 2.10: Simulink Library Browser

However, it provides graphical and numerical representations of the embedded systems. This software is integrated with *Matlab*® which allows the user to export and import data from and to *Matlab*® algorithms. The blocks library supported by *Simulink*® are customizable and can be edited by the user to fit their applications[6].
Moreover, *Simulink*® offers supports packages for different kinds of microcontrollers (Arduino Family, ARM Cortex) which allow us to interact, read, write, and to generate automatic code and then to compile the entire system into the used hardware.

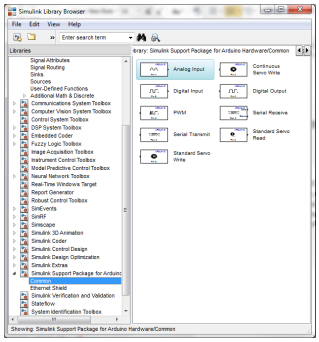


Figure 2.11: *Simulink*® Support Packages for Arduino Hardware

2.2.3 Faulhaber Motion Manager

Faulhaber Motion Manager is a software intended to configure and operate FAULHABER motors. The control of the system motors can be connected directly into the computer via different interfaces. The software provides a user interface with the connected motors and it allows changing, reading and reloading parameters (Faulhaber Motion Manager Datasheet[7]).

However, with this software we are able to change and modify the parameters of the driver, and to change the mode of the motor in such a way to keep all the parameter compatible with the hardware and motors used in the system.

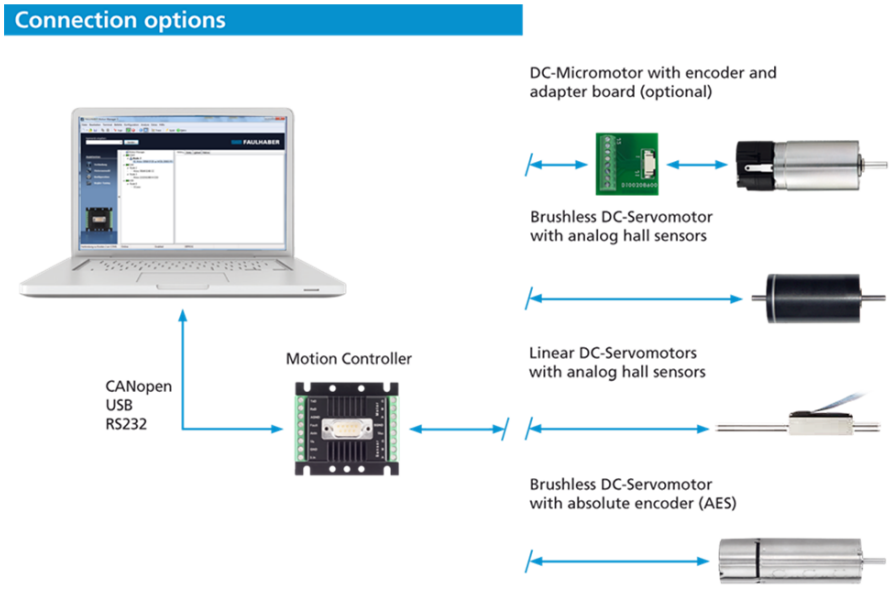


Figure 2.12: Faulhaber Motion Manager connections with the hardware

To summarize, this software is designed for the following tasks[7]:

- The configurations of device functionalities and parameters.
- Devices operations via various interfaces (USB, CAN, etc..)
- Input command for motor control.
- Creating programs for motor configurations.

- Providing dynamic setting for controlling the parameters of the controllers.
- Analysing graphically the performance of the drivers.

Chapter 3

Hardware Connections and Control Tests with STM32f4 Discovery Microcontroller

3.1 General Overview

This chapter explains the Hardware connections and control tests of the real robot with the STM32f4 discovery micro-controller, the main objective of this work is to let the robot's arm follow a reference signal required (in our case we used a square waveform).
This work has been done in two stages:

- Hardware Connections: In this stage all the hardware required (driver, 9DOF razor IMU, 12-bit magnetic position sensor, power supply) to perform the control test have been connected to the micro-controller. Moreover, the driver required an external circuit to perform a voltage conversion from 3 to 5 Volts, since the driver's voltage that controls the speed is 5 Volts. Figure 3.1 shows the entire system connections with the STM32f4 micro-controller.

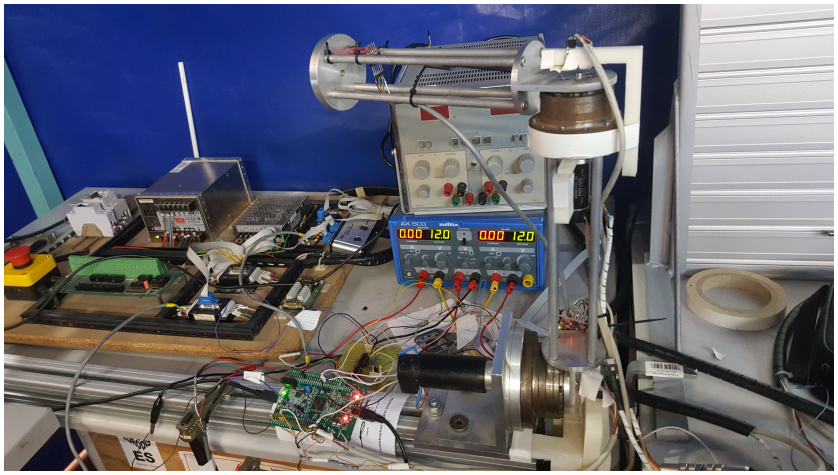


Figure 3.1: Hardware connections with STM32F4 micro-controller

- Control Models: The *Simulink*® control model for this micro-controller is done by a previous student, some changes have been required regarding the received data types of the sensors and the frequency of the entire system. The complete control model is divided into two sub-models:

  – The first sub-model is embedded in the micro-controller and it is responsible for receiving data from the sensors and transmitting data to the micro-controller through USB VCP STM32F4 *Simulink*® blocks.

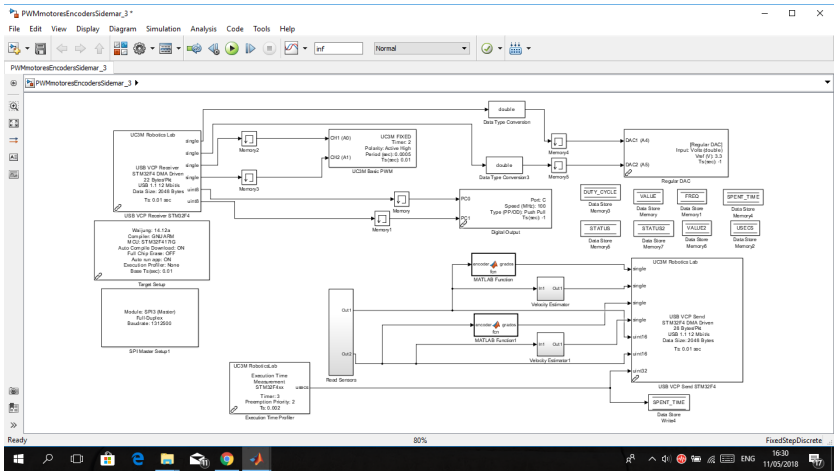


Figure 3.2: *Simulink*® First Sub-model

– The second sub-model is in charge of controlling the robot; this sub-model connects the simulation with the real robot, the data transmitted by the control blocks are received in this sub-model to close the loop, then the error signal (which is the difference between the reference signal and the data received by the sensors) is the input to the standard PID controller. The output of the PID controller is received by a zero cross comparator Simulink block that produces a digital signal 0 or 1 to control the direction of the motor and an absolute value Simulink block that transmit data to control the speed of the motors.

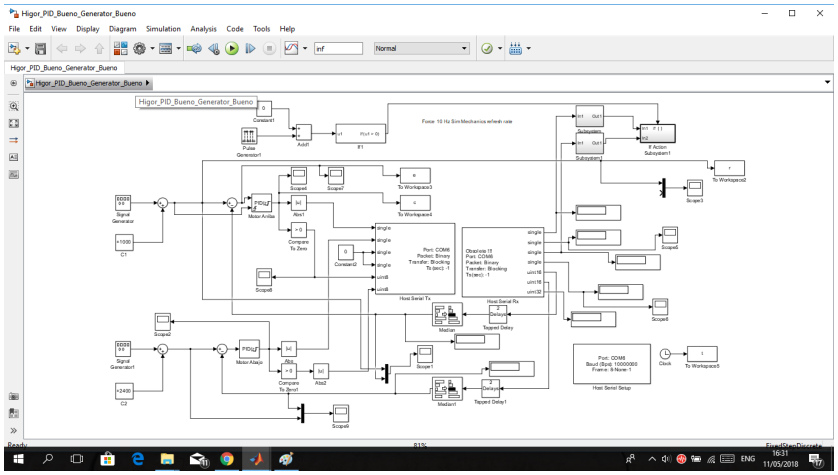


Figure 3.3: *Simulink*® Second Sub-model

3.2 Hardware Connections

3.2.1 Connections with the Driver

Faulhaber MCDC2805 driver has three main connections with the micro-controller in order to control the motor's speed and direction.
The three connections are the following:

- Analog Input signal(AnIn): This signal is connected to the power pin (5 V) of the micro-controller, the functionality of this pin is to control the motor's speed.

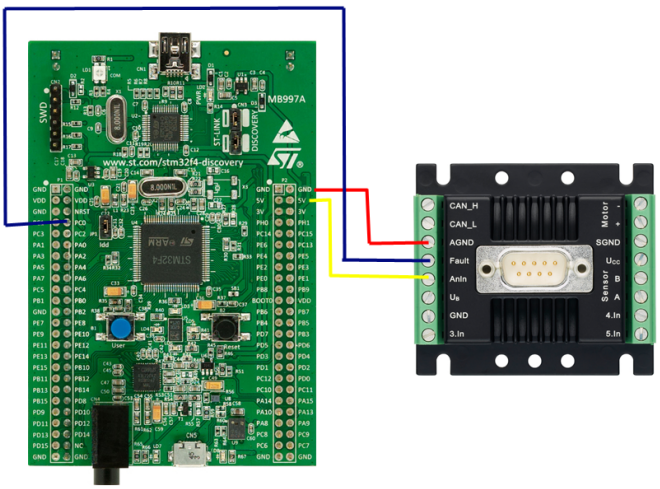


Figure 3.4: Driver connection with STM32f4 micro-controller

- Fault Output: This signal is Digital signal input to the driver that reads values 0 or 1, when signal reads the value 0, the motor rotates clockwise, otherwise, the motor rotates counter clockwise.

- AGND: is the Analog ground signal and it is simply connected to the ground pin of the micro-controller.

3.2.2 Connections with the Sensors

The sensors are connected due to their **Datasheets**. Each pin of the sensor is intended to receive and transmit data to the micro-controller.
The sensors are powered up by the power pins of the micro-controller (3.3 V for the 9DOF razor IMU and 5 V for the 12-bit magnetic position sensor). Moreover, the magnetic position sensor requires the following more 3 signals to be connected[4]:

- SCK: Clock frequency signal provided by the micro-controller to the sensor.

- Chip Select (CSn; active low) selects a device within a network of AS5045Bs and initiates serial data transfer when the logic signal is low. When CSn is set to high the serial data transfer terminates.

- DO: Digital output, is the Data output of synchronous serial interface, this signal is a Pulse With Modulated output(PWM), whose duty cycle is proportional to the measured angle. For angle position 0 to 4094.

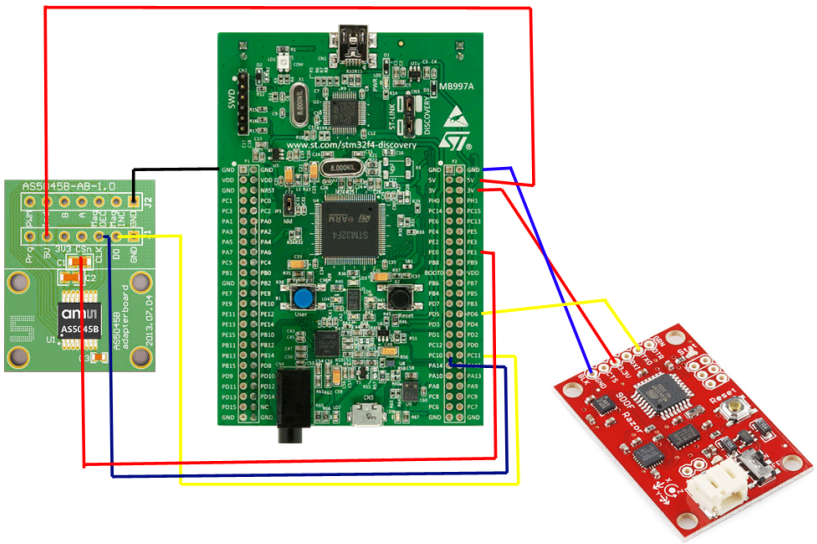


Figure 3.5: Sensors connections with the micro-controller

3.3 Control Tests and Results

Once the hardware connections are done, the *Simulink*® blocks for sensors and actuators that allow to perform movements and sense the position of the model are used.
These blocks allow us to receive data from the sensors and to transmit them into another model that is intended to do the control of the real robot.
Control Blocks related to the hardware that connects the simulation with the real robot are used to control the real robot.
In our case, a standard PID controller as shown in Figure 3.6 has been used to control the robotic arm.

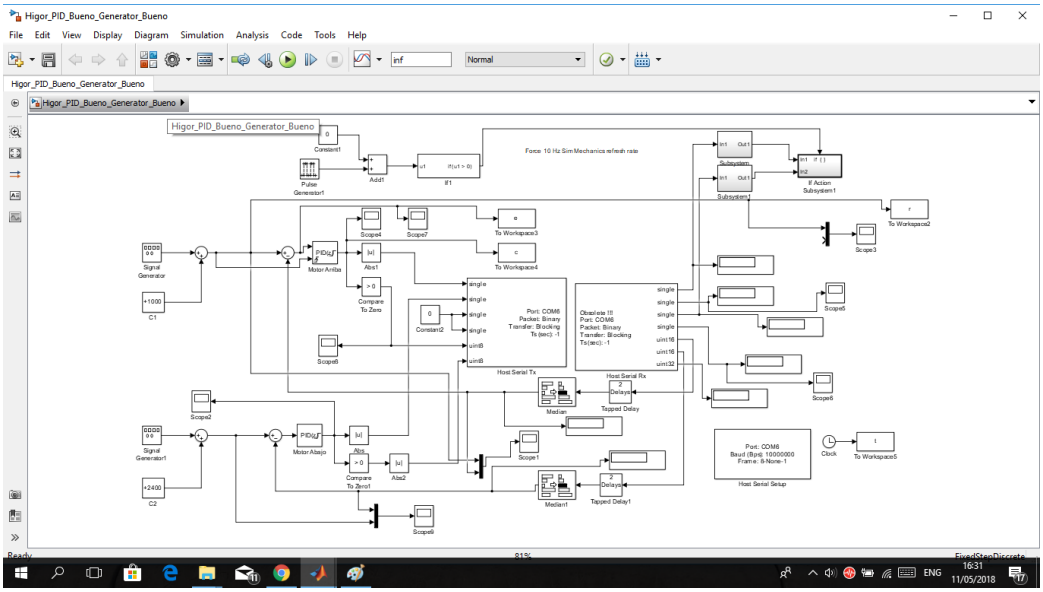


Figure 3.6: Simulink full model with Hardware Blocks

The model of the robot is included in the sub-system1 block.

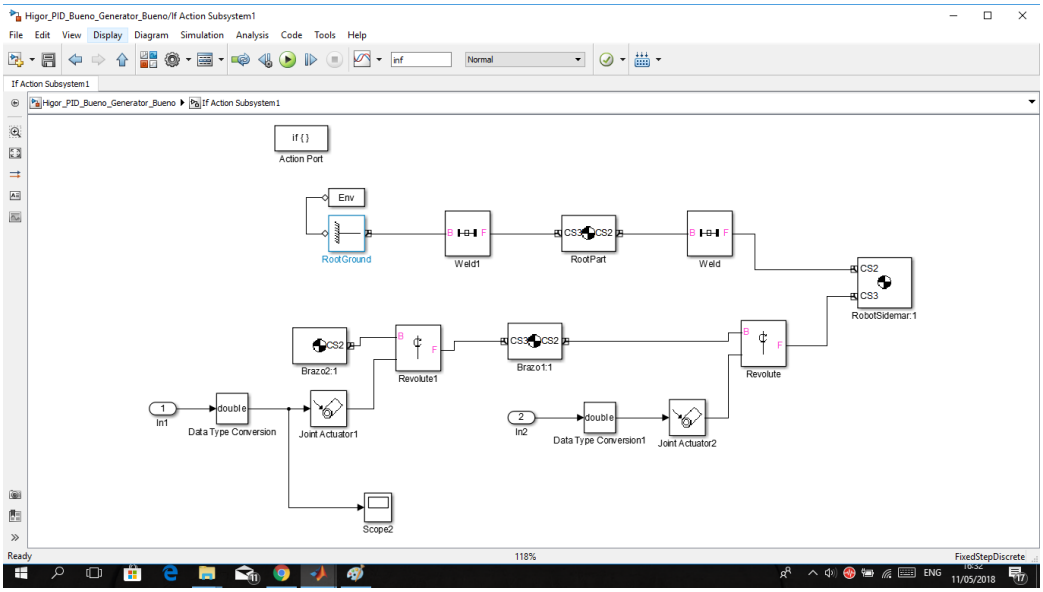
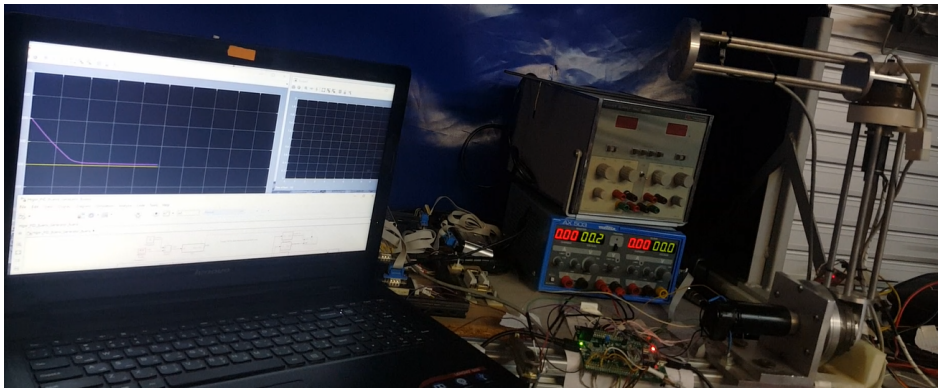


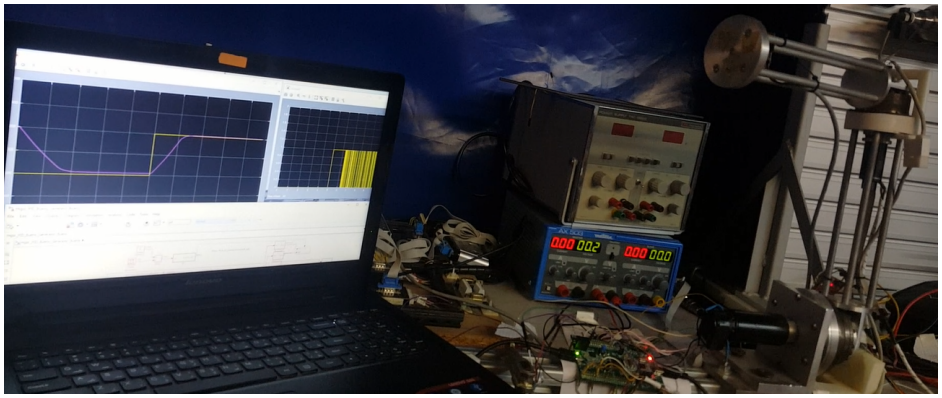
Figure 3.7: SIDEMAR model in Sim-Mechanics^{TM}

The outputs of the micro-controller are received by the Host Serial Rx comes from the USB connection in order to close the control loop.

However, the outputs of the PID block is connected to DC motors through an Analog signal ranging from 0-3.3 Volts, which is the working range of the I/O signals of the micro-controller. Moreover, a zero crossing comparator produces a digital signal 0 or 1 to activate the switch to change the motor rotation direction, which is connected to the fault signal of the driver.
To test the performance of the system, the control test of the real robot has to be done.
Figures 3.8a and 3.8b show the response of the arm to the reference signal. When the reference signal is low level the upper arm of the robot follows it until the error is almost zero as shown in Figure 3.8a, and the same response happens when the reference signal is high level, the upper link of the robot changes its direction to follow the reference signal.



(a) Reference signal is low



(b) Reference signal is high

Figure 3.8: Robotic Arm Response

Chapter 4

Hardware Connections, Simulink Blocks and Control Strategies with Arduino Mega 2560

4.1 General Overview

This chapter explains in details the connections of the Hardware SIDEMAR with Arduino mega 2560, also the *Simulink*® blocks used in order to read the data coming out from the magnetic position sensor using the SPI library. Moreover, the different control strategies (Standard PID controller, PID control design using *Simulink*® blocks, Fuzzy Logic Controller, Adaptive Controller) used in order to control the robotic arm.

4.2 Hardware Connections with Arduino Mega 2560

The connections of Arduino mega 2560 with the Driver and the Magnetic Position Sensor have been done with a way to keep the hardware working in the correct mode also in real-time.
As Arduino mega 2560 doesn't support Analog output, the characteristic of the driver that has the Analog Input (AnIn) has to be configured in order to

be able to control the speed of the motor. This characteristic is provided by the software FAULHABER MOTION MANAGER that allows us to configure the characteristics of the driver, whose able to work under several conditions and modes.
Figure 4.1 shows the characteristics of the Command Source(SOR) changed in order to provide the driver with PWM signal supported by Arduino Mega 2560

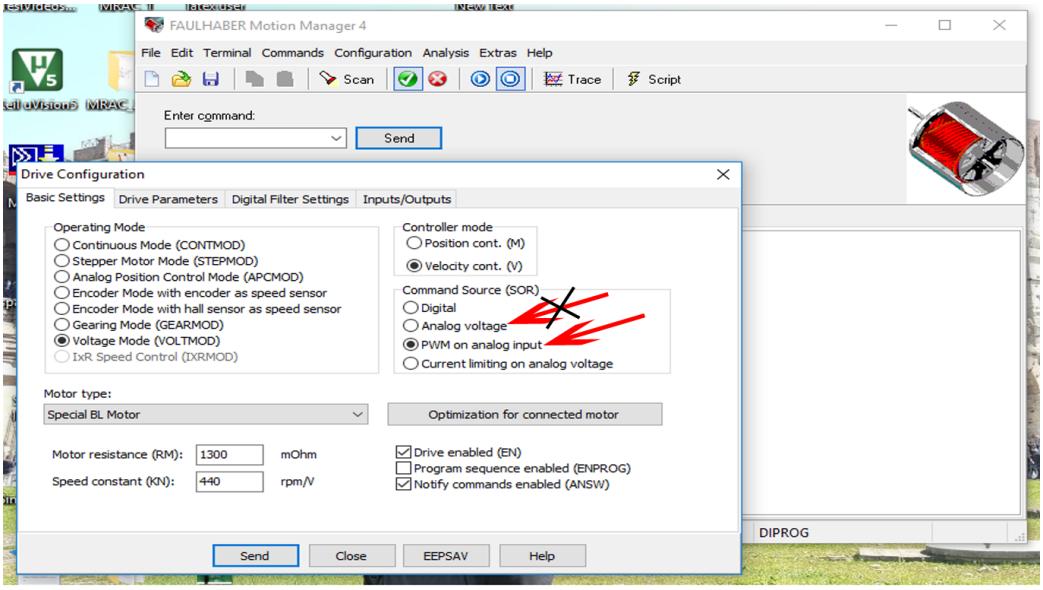


Figure 4.1: MCDC 2805 driver's Configuration

On the other hand, as the Magnetic Position Sensor provides two digital output signals A and B; these signals can be used in order to read the data from the sensor, the logic states of the signals depend on the CSn (Chip select signal). If the chip select signal CSn is at logic high when the system is powered up, the two incremental outputs A and B will remain at logic high until the chip select signal CSn goes into logic low state. When the two incremental outputs A and B are at high logic, the control requests data from the sensors. The direction of the motor depends on the state of the two signals, when signal A leads signal B, the direction is clock-wise; otherwise, the direction of the motor is counter clock-wise[4]. Figure 4.2 shows the Incremental Modes of A and B.

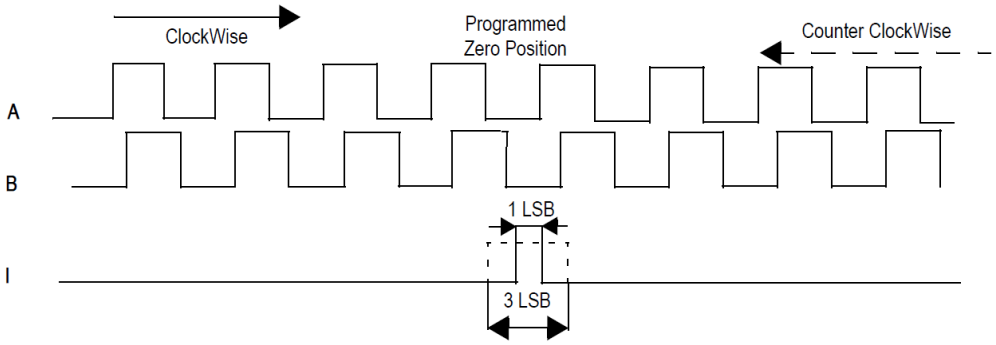


Figure 4.2: Incremental Outputs A and B modes[4]

After testing the *Simulink*® blocks shown in Figure 4.3 (where pin8 is CSn signal, pin 32 and 34 are signals A and B respectively), this method has been avoided due to the problem of zero-position, because when the robot arm moves and then the system is stopped, we have to manually calibrate the position of the arm into the programmed zero-position shown in Figure 4.2.

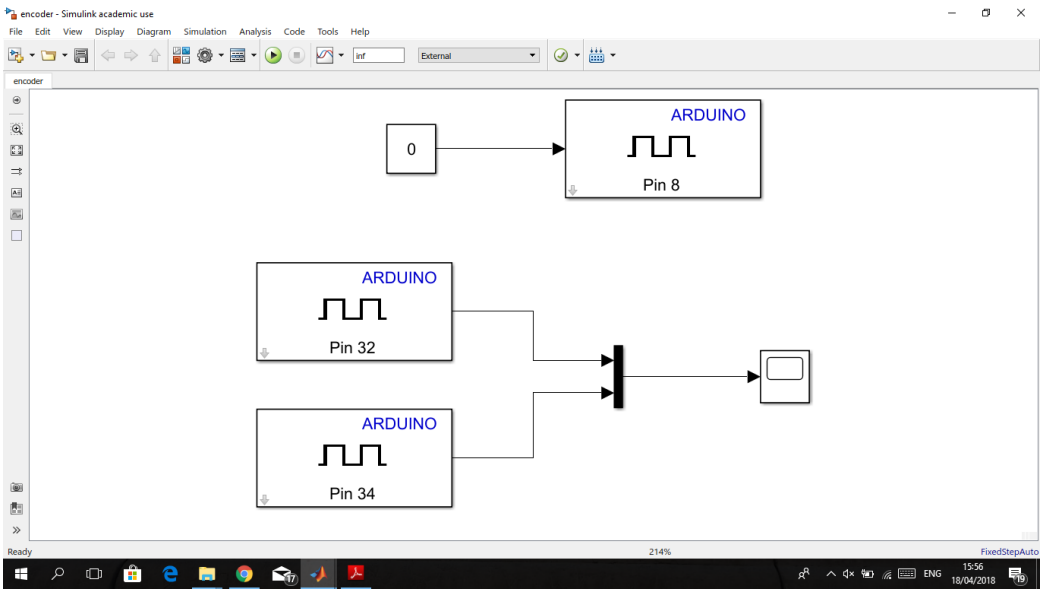


Figure 4.3: Simulink Blocks for Incremental Outputs A and B

In order to avoid manual calibration, the solution was to read the data from the sensors using the SPI library, since Arduino mega 2560 supports SPI (Serial Peripheral Interface) blocks that allow reading and writing to SPI

devices connected to *Arduino*® hardware, this method has been followed.

Typically, the Arduino microcontroller (Called Master device) initiates the communication and supplies the clock to the sensor (Called Slave device) which controls the data transfer rate[8].
To clarify, in the SPI system there are 4 signal lines[8].

- Master Out, Slave In(MOSI) - which is the data going from the master to the slave(From micro-controller to the sensors).
- Master In, Slave Out(MISO) - which is the data going from the slave to the master(data received by micro-controller from the sensors).
- Serial Clock(SCK) - when this toggles both the master and the slave sample the next bit.
- Slave Select(SS) - this tells a particular slave to go "active".

Figure 4.4 shows the way the data is exchanged as one byte is sent.

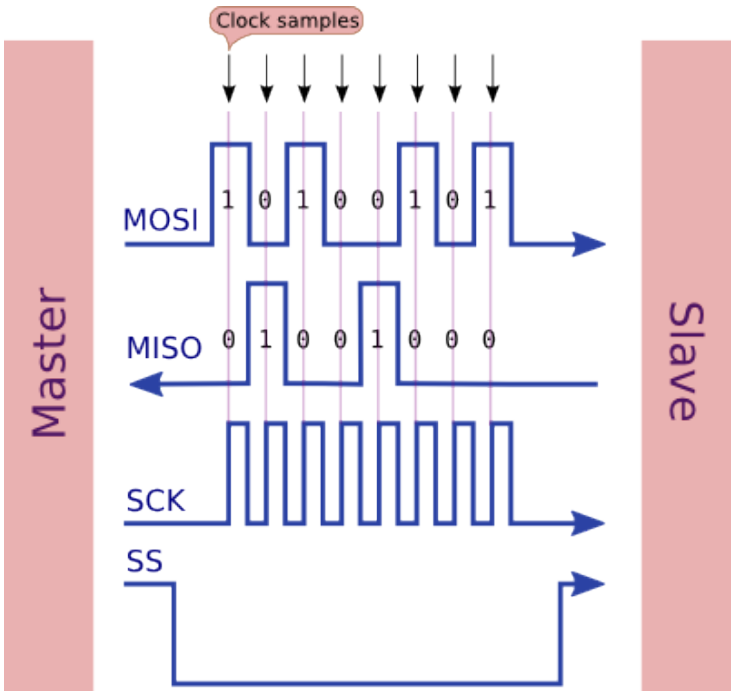


Figure 4.4: SPI data exchange

However, these pins are multi-provided by the Arduino mega 2560 as shown in Figure 4.5. ICSP header can be also used (the one shown inside the red box in Figure 4.5).

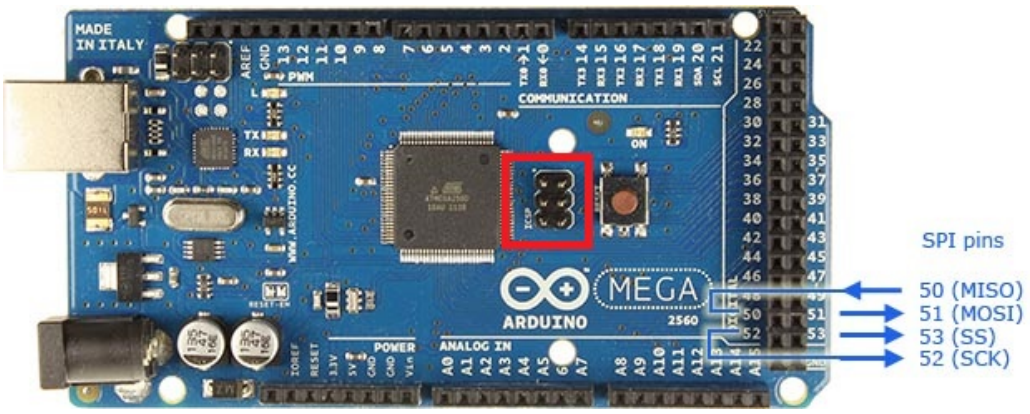


Figure 4.5: Arduino Mega 2560 SPI pins

According to the magnetic position sensor Datasheet, the 4 signals to be connected with the SPI pins are:

- Vss: Ground or Negative supply Pin connected to the ground pin of the micro-controller.
- VDD5V: Positive supply pin (5V) connected to the 5Volts pin of the micro-controller.
- DO: Digital Output of the sensor is connected to the MISO(Master In Slave Out) pin of the micro-controller.
- CLK: Clock Input to the sensor is connected to the SCK of the micro-controller.
- CSn: Chip Select is connected to the SS(Slave select) pin of the micro-controller.

Figure 4.6 shows the connections of the magnetic position sensor, driver and the power supply with the micro-controller.

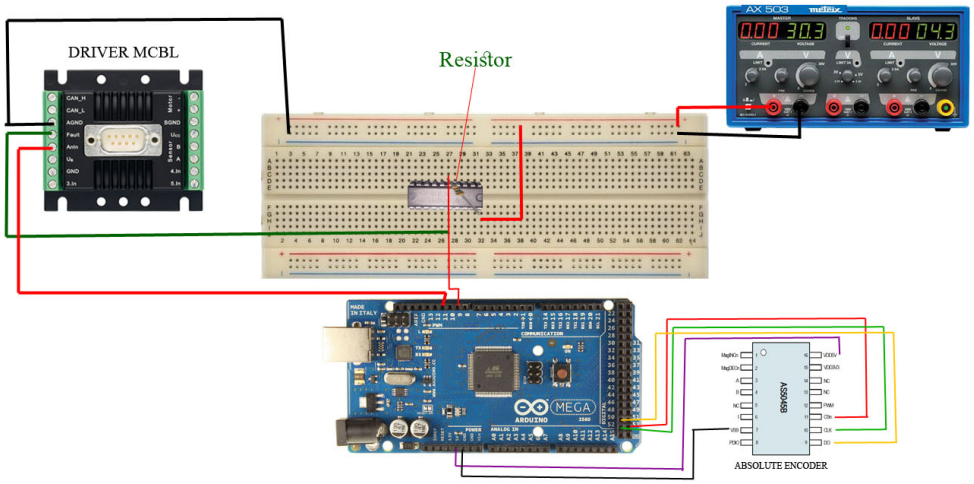


Figure 4.6: Arduino Mega 2560 connections with the Hardware

However, to make sure that the position sensor is correctly working. A fast manual test has been done by moving the upper robotic arm manually and see how the position sensor is responding due to the rotation of the arm. Figure 4.7 shows the response of the sensor to the manual random movement of the upper arm, we can see that range of working angle is between 0 and 4094 (0° to 360°)

Figure 4.7: Magnetic position sensor response to manual random test

4.3 Control Strategies with Arduino Mega 2560 and *Simulink*® Matlab

For controlling the 2 DoF robotic arm, a *Simulink*® model has to be designed. For this, *Simulink*® offers a support packages for *Arduino*® hardware, these packages contain I/O blocks, SPI library, Wireless blocks and Ethernet blocks that allow to work on different projects with various purposes.
The designed model contains blocks related to I/Os of the system, the communication interfaces with the control model of the robot and other hardware issues. The micro-controller is connected to the control model of the robot via USB.
The input block includes reading data from the encoder (Slave device) using the SPI block as shown in Figure 4.8.

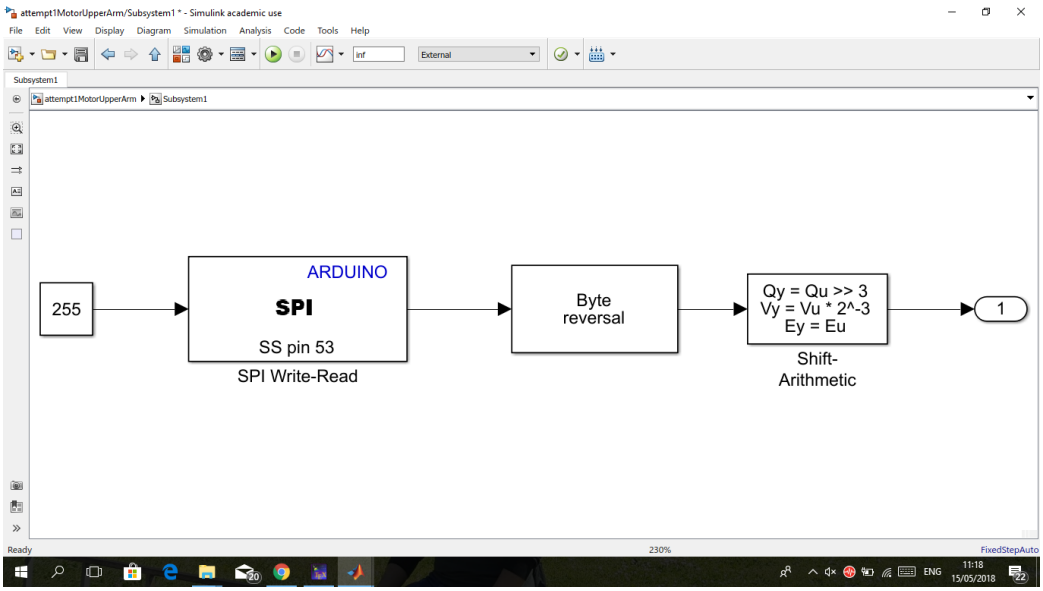


Figure 4.8: Read data via SPI block from Magnetic position sensor

The output blocks include a PWM signal and Digital Output signal to the speed and direction control of the robot's DC motor as shown in Figure 4.9

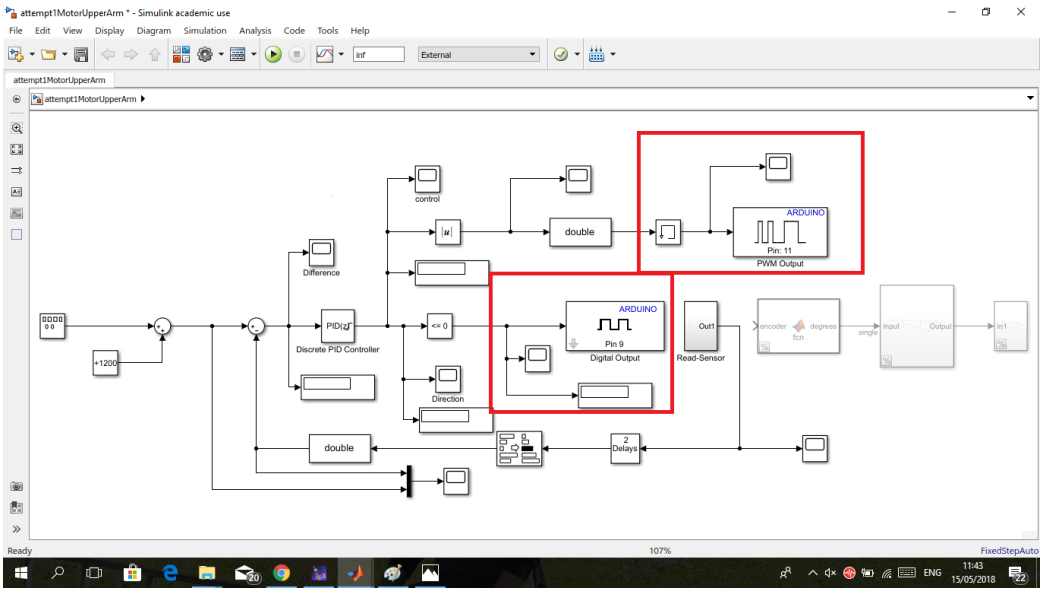


Figure 4.9: Output Blocks to Arduino Mega 2560 micro-controller

Once the inputs and the outputs of the system are defined, the data type of the system have to be similar. So, data type conversion blocks have been added to the input of the model to convert a uint16 data type comes out from the sensor into a double data type to be the inputs of the controller. Moreover, to test the system, a standard PID controller in Simulink has been used as shown in Figure 4.10

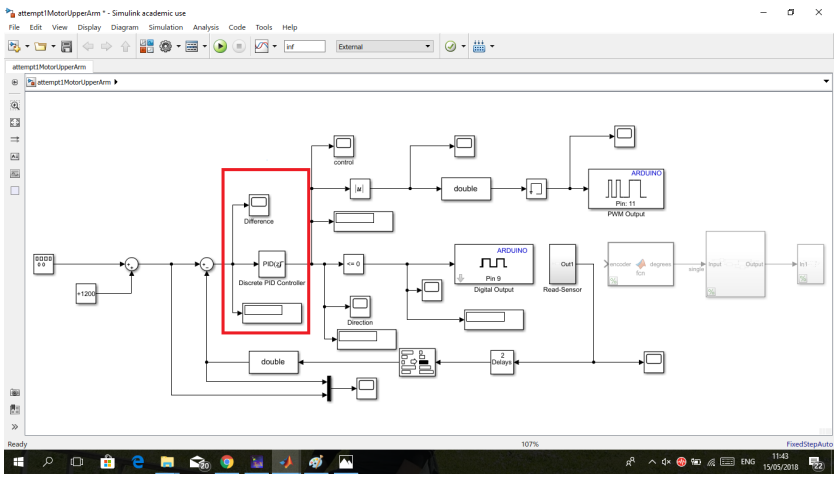


Figure 4.10: Control Model with Standard PID Controller

The aim of this test was to check if the arm is able to follow the square-wave reference signal used in the Simulink Model in Figure 4.10.

4.3.1 Experimental Results with Standard PID Controller

The result of the Simulink standard PID controller is shown in Figure 4.11.

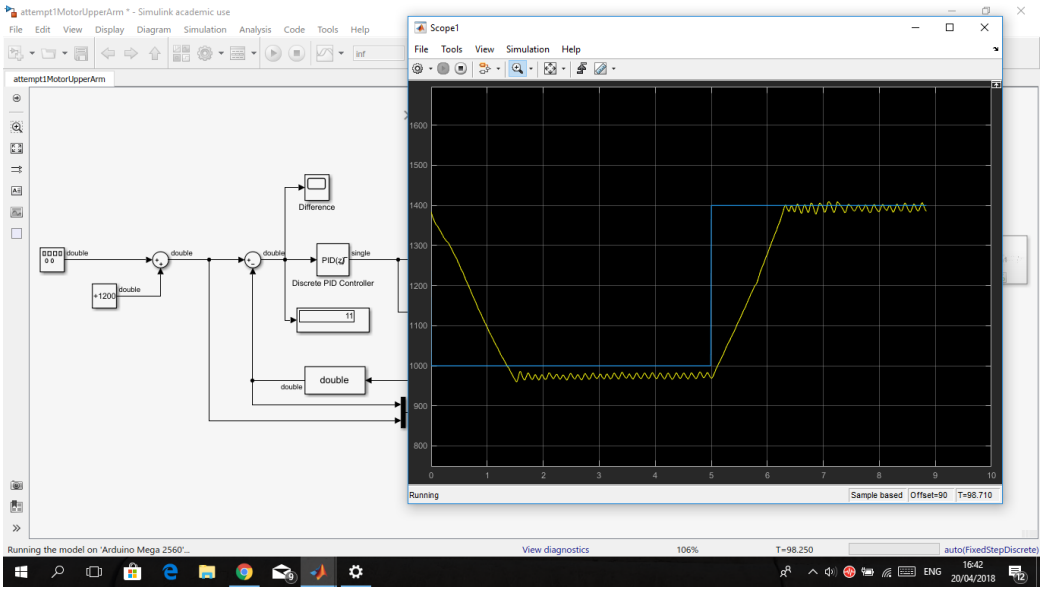


Figure 4.11: Responce of the Upper Arm to the Standard PID controller

The Simulink standard PID controller uses the compensator formula $P + I\frac{1}{s} + D\frac{N}{1+N}$ (where N is the filter coefficient), the compensator formula has a low-pass filter on its derivative term. The low-pass filter used in the standard PID controller is to smooth the controller output signal (CO). In the next section, a designed PID controller is used. The Derivative term of the PID controller is set to zero, so the controller has been working as PI controller. The results of both tests are compared in the next section.
The designed PID controller follows the formula:

$$K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

Figure 4.12 shows the *Simulink*® blocks for the designed PID controller with standard discrete *Simulink*® blocks.

As K_p, K_i and K_d are constants, a standard gain block have been used in *Simulink*®, and a discrete integrator and derivative blocks are used to the integration and derivative of the error respectively.

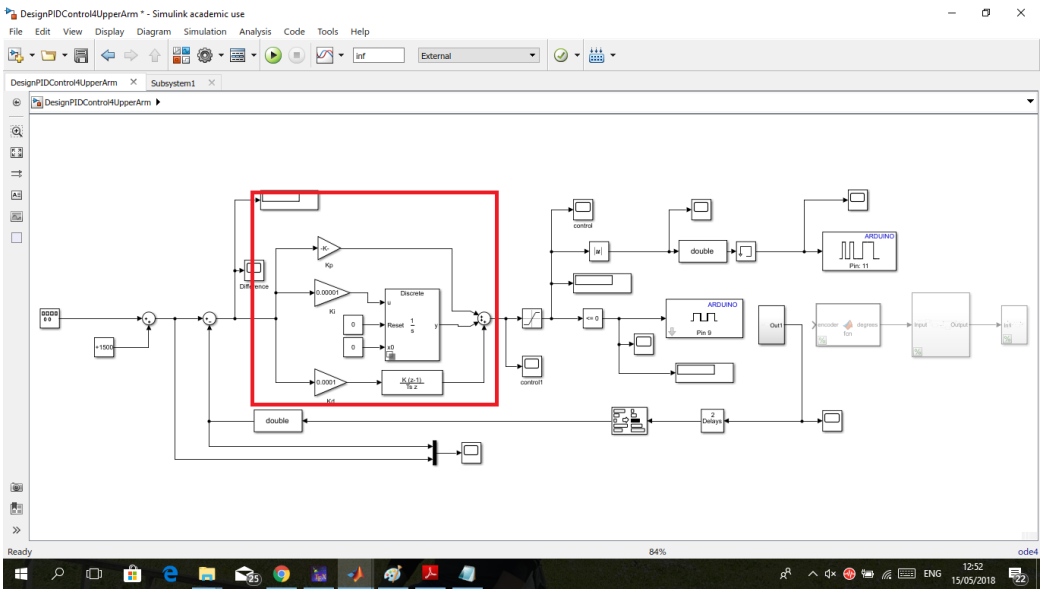


Figure 4.12: Control system with designed PID controller

4.3.2 Experimental Results with Designed PID Controller

Figure 4.13 shows the results of the control system with designed PID controller when the Derivative term is set to zero.

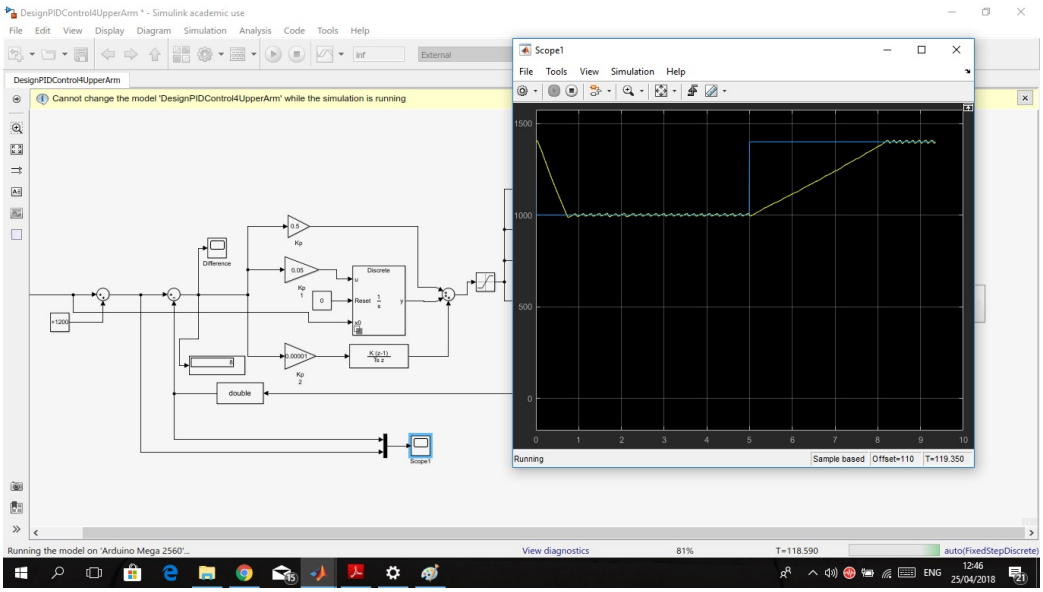
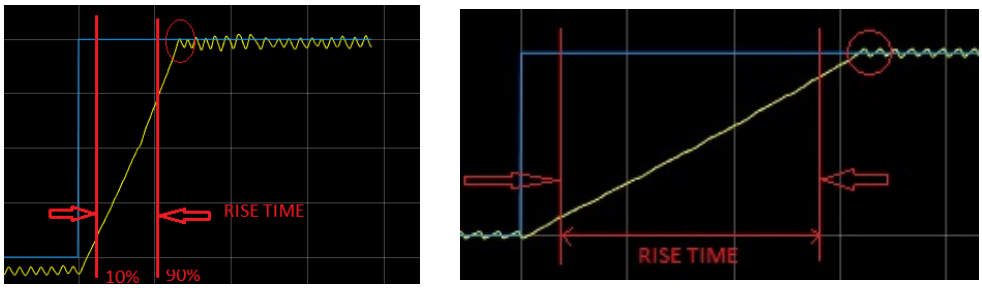


Figure 4.13: Upper Arm response to the designed PID controller

Comparing the results of both tests, we can see that the PID controller has somewhat shorter rise time and smaller overshoot. The comparison is shown in Figures 4.14a and 4.14b



(a) PID **(b)** PI

Figure 4.14: Response Characteristics to PID and PI Controllers

4.4 Fuzzy Logic Control

4.4.1 General Overview

The purpose of this work is to build a fuzzy logic control model for a dynamic system on a robotic arm example. The software used in this work was *Matlab*® 2018a with *Simulink*® environment that provides a Toolbox for the Fuzzy Logic Control.
Fuzzy controllers are used in many different fields to control products such as autonomous robots, washing machines, video cameras and many other applications. Fuzzy Control is based on fuzzy logic that is simply described as controlling with words rather than controlling with numericals values. It is generally defined as "control with sentences rather than equations"; However, it is based on defining logic inputs with conditions "if" and producing outputs according to the pre-defined inputs[9]. For example, we can take a typical fuzzy controller as the following:

- If error e(t) is Negative then the output is A.
- If error e(t) is Positive then the output is B.
  ...

These logic conditions are called *rule base*. These rules are defined with the conditions if-then format, The if-side refers to *condition* and then-side refers to *Conclusion*[9]. The purpose here is to define different conditions and situations for the input error and produce the output that controls the motor's directions and speed according to the pre-defined input.
As Fuzzy controllers are very simple conceptually. They consist of 3 stages. An input stage, a processing stage and an output stage as shown in Figure 4.15.

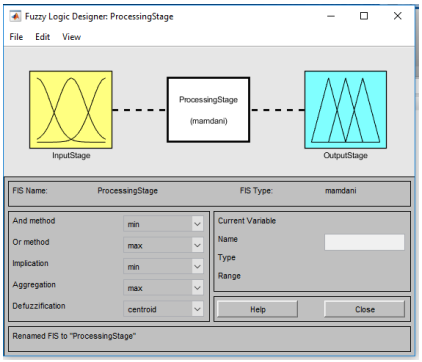


Figure 4.15: Stages of Fuzzy Logic Controller

The input stage maps sensors or other inputs, to the appropriate membership functions and truth values. The processing stage invokes each appropriate rule and generates a result for each, then it combines the results of the rules. Finally, the output stage converts the combined results back into a specific control output value.

4.4.2 Rules, Conditions and Simulink Blocks

Our Fuzzy control takes the error signal $e(t) = Y_r - Y_p$ as input and produces the voltage and direction as output.

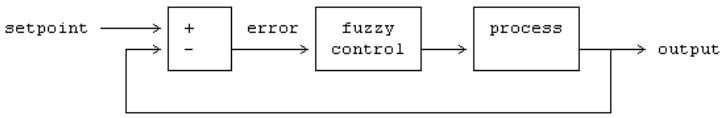


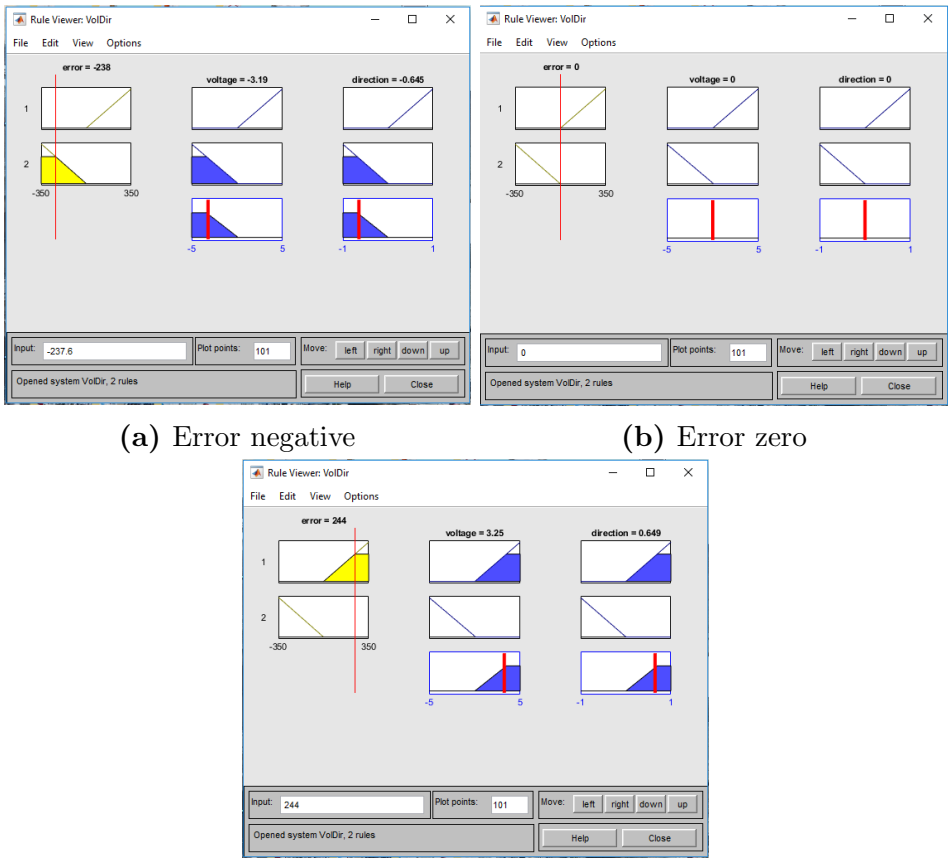
Figure 4.16: Control Scheme of a Fuzzy Logic Controller

The proposed system has 3 conditions:

Error	Voltage	Direction
Error $= [-350, 0[$	Voltage $= [-3.3, 0[$	Direction $[-0.6, 0[$
Error $= 0$	Voltage $= 0$	Direction $= 0$
Error $=]0, 350]$	Voltage $=]0, 3.3]$	Direction $]0, 0.6]$

Table 4.1: conditions for inputs and outputs of Fuzzy Controller

However, Fuzzy control provides rule viewer that confirms the characteristic and the rules of your system, these viewer is shown in figures 4.17a, 4.17b and 4.17c



(a) Error negative **(b)** Error zero

(c) Error positive

Figure 4.17: Conditions For Error Signal

Finally, after defining all the conditions and the rules for this controller, the only change that has to be done for the *Simulink*® model is to replace the PID controller with *Fuzzy Logic Controller Simulink*® block and to define the name of the fuzzy controller in the "FIS name" in the Block parameters box.

Figure 4.18 shows the Fuzzy Logic Controller Simulink model.

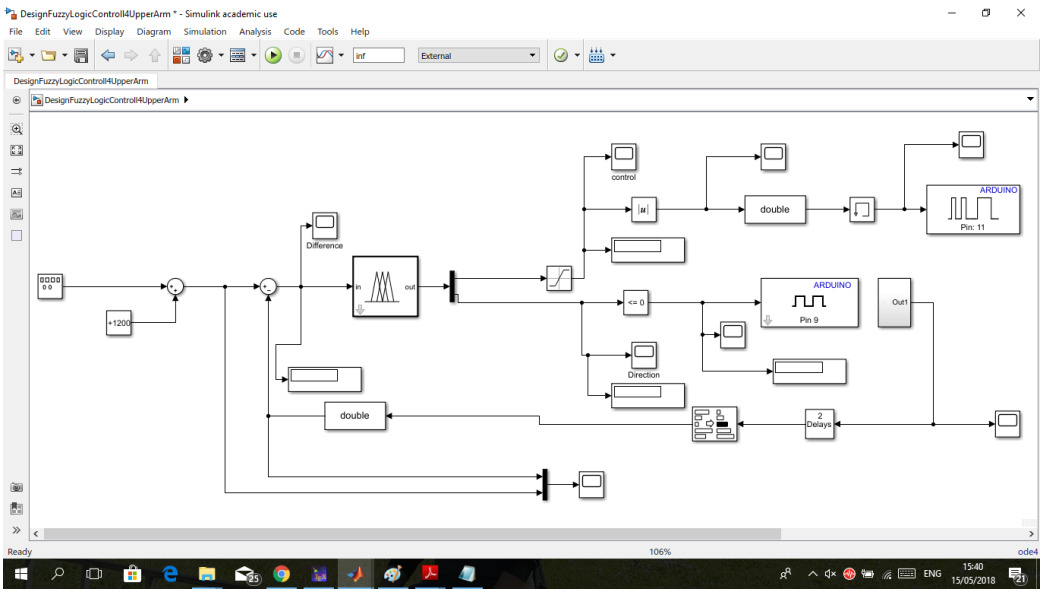
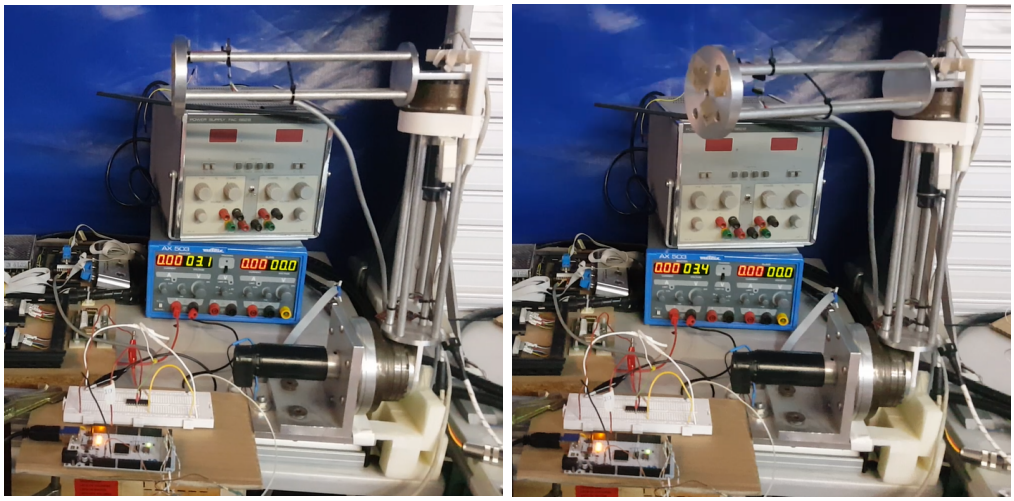


Figure 4.18: Simulink Model with Fuzzy Logic Controller

4.4.3 Experimental Results and Conclusion

Once the control model is ready, the robot arm should follow the reference signal. Figures 4.19a, 4.19b, 4.20 show the response of the robotic arm with respect to the reference signal.



(a) Situation 1 **(b)** Situation 2

Figure 4.19: Response of Robotic Arm to Fuzzy Logic Controller

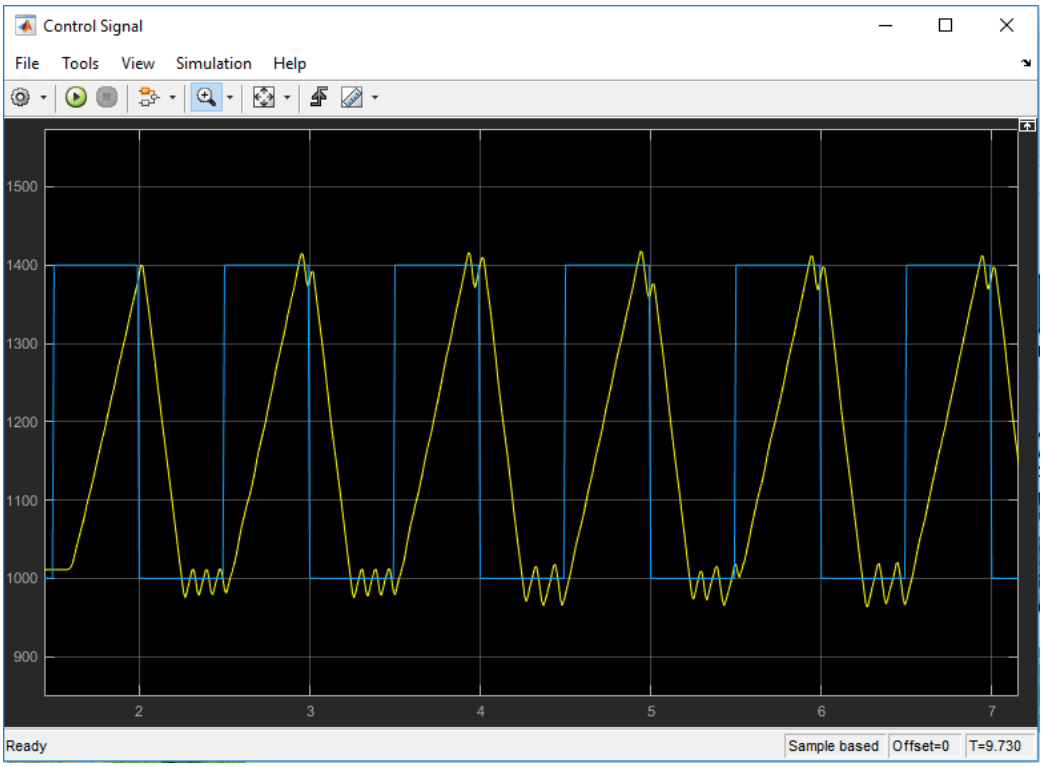


Figure 4.20: Control response of robotic arm

However, the voltage, Direction and Error signal have been recorded to show the movement of the arm depending on these values.

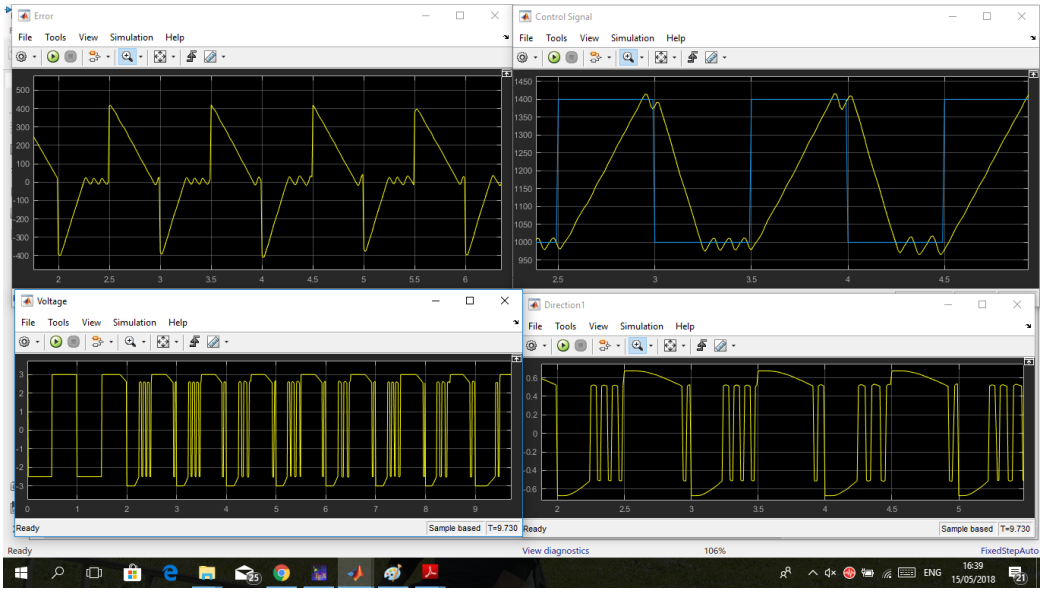


Figure 4.21: Voltage, Direction, Error signal of the Fuzzy Logic Controller

As conclusion, we find from the previous figures that when the error signal is negative the robot arm regulates its positions until the error signal is almost equal to zero with some vibrations. However, when the reference signal changes its level from low to high, the difference between the feedback (sensor's data) and the reference signal increases, this way the robot arm changes its direction to follow the reference signal until the error signal is also equal to zero. As we discussed during the tests, the vibrations might be due to the old hardware used in the system.

4.5 Adaptive Controller

4.5.1 General Overview

Adaptive Control is a set of different techniques that automatically adjust the control parameters of the systems in real-time; the adaptive controllers keep changing the control parameters in order to maintain the desired performance when the plant dynamic model's parameters are changing and/or unknown in time[10].

Consider the first case when the plant dynamic model's parameters are changing in time; this case may occur either because the conditions of the surrounding environment change (the dynamical characteristic of a robot arm) or because considering a simplified linear model for non-linear systems[10].

In the second case when the plant dynamic model's parameters are unknown in time but constant, the controller structure does not depend on the plant model parameters; however, it requires a knowledge in the control parameters in order to tune them. the adaptive controller can automatically tune the control parameters in a closed loop but the effect of adaptation may vanish when the time increases and the procedure of the adaptation may require a restart when the operation conditions change[10].

However, achieving a good performance in the control system for both cases requires the consideration of adaptive controller approach.
Moreover, the case considered for our system requires a control scheme as shown in Figure 4.22, where a typical closed loop is shown. In addition, a second loop is added to identify the system parameters and to calculate the adaptive controller parameters.

Identifying the plant model requires measuring the input/output of the plant model. However, the data collected from the system allow tuning and designing the controller. Figure 4.22 shows the implementation procedure of an adaptive controller in real-time, where the data collected from the plant's input and output are input to the system identification process and tuning of the controller parameters is done according to the data collected.

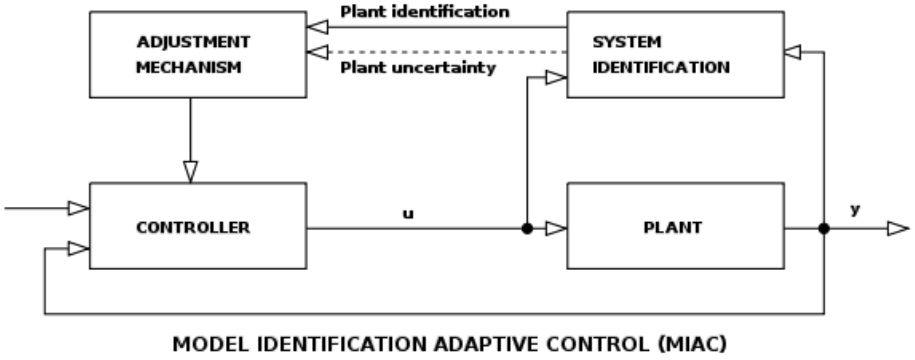


Figure 4.22: General Scheme Overview of The Adaptive Control

4.5.2 Equations

The controller used in the model identification adaptive control is PID controller with adaptive parameters, the reason behind this is that PID controller is much used in control loops. Its parameters are need to be adjusted in function of the control process and remain unchanged during its regular activities.
On contrary, this kind of adaptive controllers has a disadvantage or a difficulty in the controller start-up service, since it is necessary an observation period to survey with greater certainty the controller performance, or is more difficult to adjust the regulators without actually knowing the exact system dynamic.
The robot model is a damped second order system with input and output linearities to count for different response times at different arm positions.
The regulator parameters are calculated using the pole placement technique (R. Barber, 1997). The pole placement technique's equations are[11]:

$$u(t) = \frac{r(t).(g_0 + g_1 + g_2) - y(t).(g_0 + g_1.z^{-1} + g_2.z^{-2})}{1 - z^{-1}} \tag{4.1}$$

However, the plant is assumed to have the following expression:

$$y(t) = \frac{b_0.z^{-1}}{1 + a_1.z^{-1} + a_2.z^{-2}} u(t) \tag{4.2}$$

The closed loop equation considering equations (4.1) and (4.2) is:

$$y(t) = \frac{b_0.z^{-1}.(g_0 + g_1 + g_2)}{(1 - z^{-1}).(1 + a_1.z^{-1} + a_2.z^{-2}) + b_0.z^{-1}.(g_0 + g_1.z^{-1} + g_2.z^{-2})} r(t) \tag{4.3}$$

The values g_0, g_1 and g_2 are found using the settling time and damped natural frequency equations.
Therefore, the characteristic polynomial equation for a system behaves as a second order is the following[11]:

$$P = 1 + t_1 z^{-1} + t_2 z^{-2} \tag{4.4}$$

where t_1 and t_2 are:

$$t_1 = -2e^{-\zeta.\omega_n.T.cos(T.\omega_n\sqrt{(1-\zeta^2)})}$$
$$t_2 = e^{-2.\zeta.\omega_n.T}$$

where ζ is the damping factor, ω_n is the natural frequency and T is the sampling period of the system.
Equations (4.2) and (4.4) should be equal since they have the same roots.

$$(1 - z^{-1}).(1 + a_1.z^{-1} + a_2.z^{-2}) + b_0.z^{-1}.(g_0 + g_1.z^{-1} + g_2.z^{-2}) = 1 + t_1 z^{-1} + t_2 z^{-2} \tag{4.5}$$

So, the gains of the controller g_0, g_1 and g_2 are:

$$g_0 = \frac{t_1 + (1 - a_1)}{b_0}$$
$$g_1 = \frac{t_2 + (a_1 - a_2)}{b_0}$$
$$g_2 = \frac{a_2}{b_0}$$

Finally, the transfer function of the PID controller is defined by the expression (4.6):

$$u(k) = u(k-1) + r(k).(g_0 + g_1 + g_2) - (g_0.y(k) + g_1.y(k-1) + g_2.y(k-2)) \tag{4.6}$$

4.5.3 Implementation in *Simulink*® and *Matlab*®

At the beginning, the implementation of the Adaptive controller has to be done using the S-function (System-Function) in *Simulink*®. This is a descriptive language writing within a block of *Simulink*® in *MATLAB*®, C, C++ or FORTRAN, becoming a powerful mechanism for extending the capabilities of *Simulink*®.
The main problem with this type of User-defined functions is the creation of the .tlc files, which is a complex work that has to be done manually. Moreover, this kind of Simulink blocks cannot work directly in External Mode on a target hardware as required for our work.
So, an idea of using standard Simulink blocks has to be followed. Figure 4.23 shows the implementation of the standard Simulink blocks for expression (4.6).

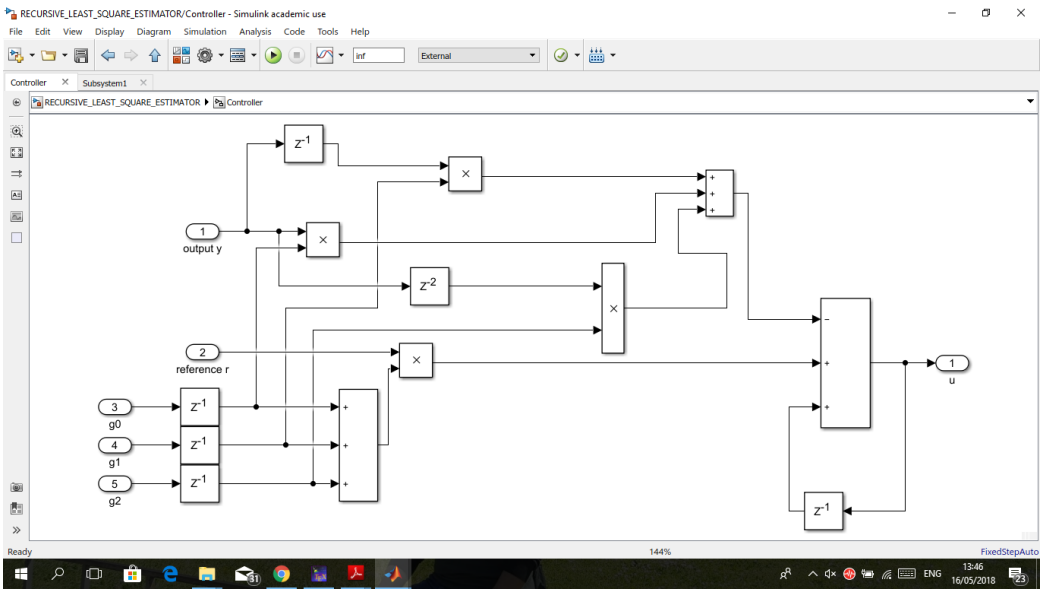


Figure 4.23: PID controller Simulink Blocks

Furthermore, the equation of the discrete system that models our robot is shown in expression (4.7):

$$y_n = b_0.u_{n-1} - a_1.y_{n-1} - a_2.y_{n-2} \tag{4.7}$$

For this model, the values b_0, a_1 and a_2 will change when the inertia of the robot changes.

The terms u_n, u_{n-1} and y_{n-1} are the inputs and outputs of the plant, these data are collected in real-time. However, the values b_0, a_1 and a_2 are the values to be found in order to identify the system. The method used to find these values is recursive least square method.
The terms u_n, u_{n-1} and y_{n-1} are the *model regressor* and inputs to the recursive least square block that estimates the b_0, a_1 and a_2 values. The implementation of the model regressor in Simulink is shown in Figure 4.24 [12].

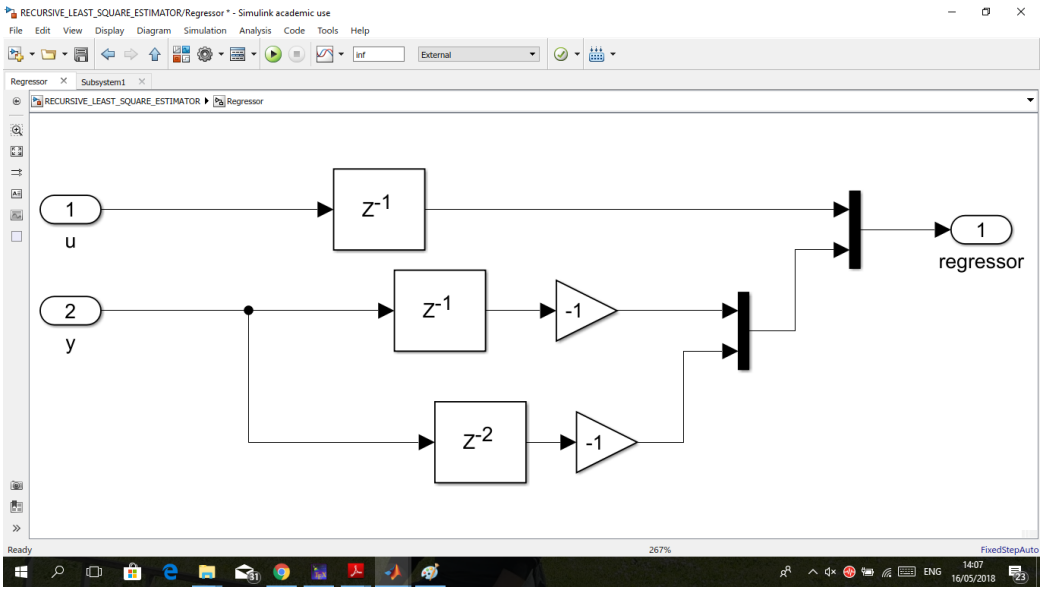


Figure 4.24: Model regressors for recursive least square estimator

The regressor's ouput is the input to the Recursive Least Square Estimator block in addition to the output signal of the system y_n, the output of the Recursive Least Estimator block are the estimated values b_0, a_1 and a_2 of the system when the inertia changes. The Recursive Least Square Estimator block is configured as follow[12]:

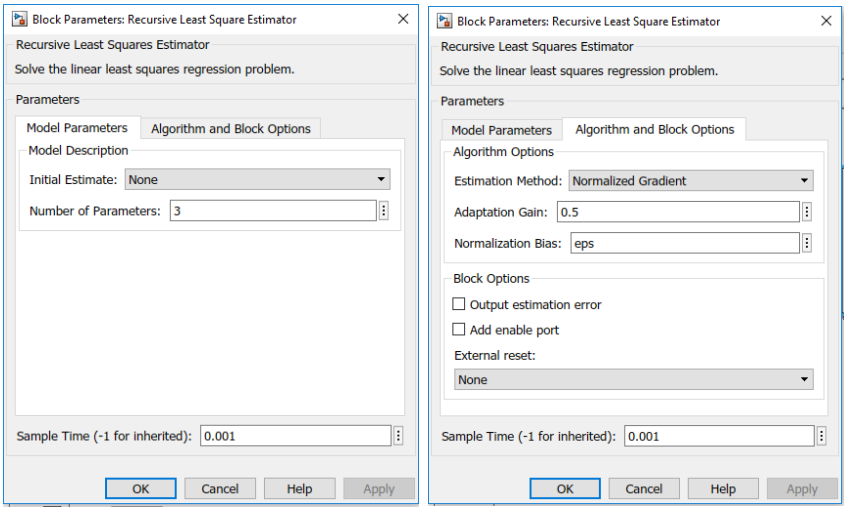
- **Initial Estimate:** None. By default, the software uses the value 1
- **Adaptation Gain:** 3. b_0, a_1 and a_2
- **Sample Time:** 0.001

To set the estimation options[12]:

- **Estimation Method:** Normalized Gradient

- **Number of parameters:** Adaptation gain, it is specified as a real positive scalar. it is directly proportional to the relative information content in the measurements

- **Normalization Bias:** Bias in adaptation gain scaling, Bias, specified as a real non-negative scalar. The normalized gradient algorithm scales the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. Bias is the term introduced in the denominator to prevent these jumps.



(a) Model Parameters **(b)** Algorithm and Block options

Figure 4.25: Block Parameters: Recursive Least Square Estimator

So, after we get the values b_0, a_1 and a_2, the controller gain has to be calculated following the expressions of g_0, g_1 and g_2. Figure 4.26 shows the implementation in $Simulink^{®}$ and $Matlab^{®}$.

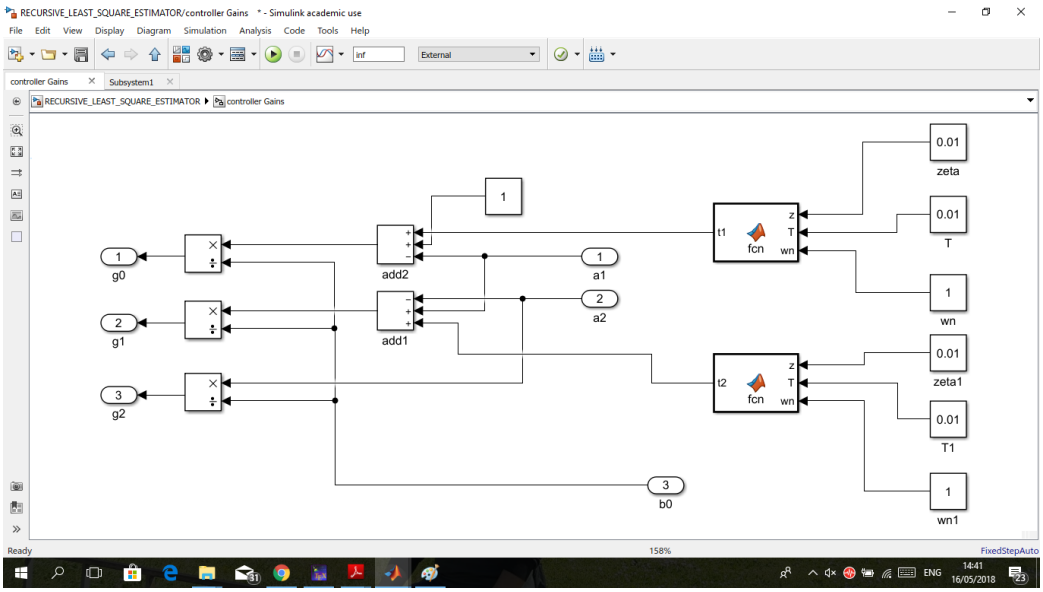


Figure 4.26: Controller Gain Calculation

Finally, the full model of the *Adapitve controller* is the chain connecting all the previous sub-systems with additional to the simulink blocks related to the input and output of the system. The full Adaptive controller model is shown in Figure 4.27.

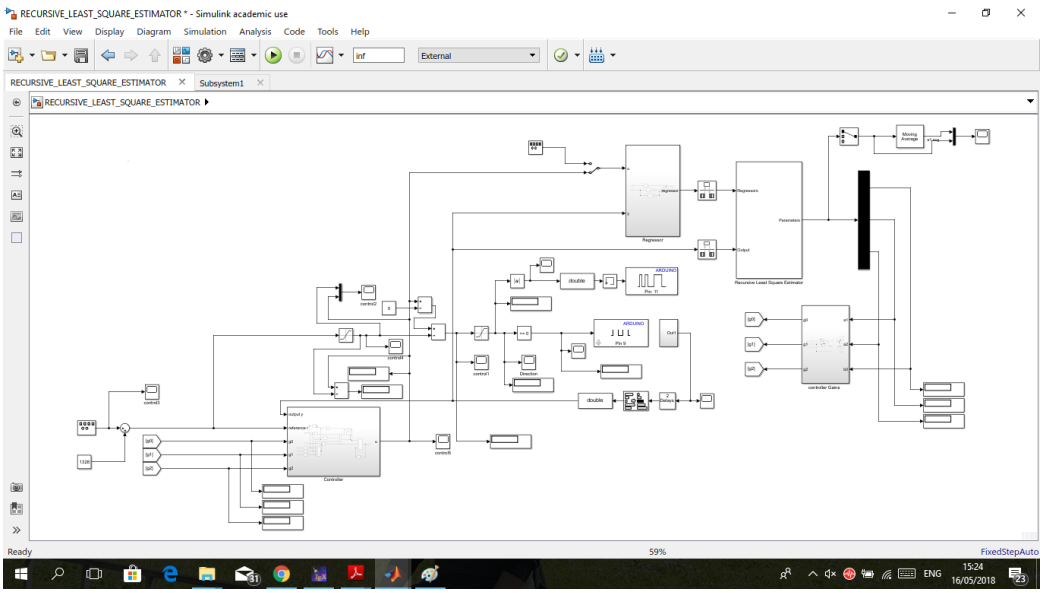
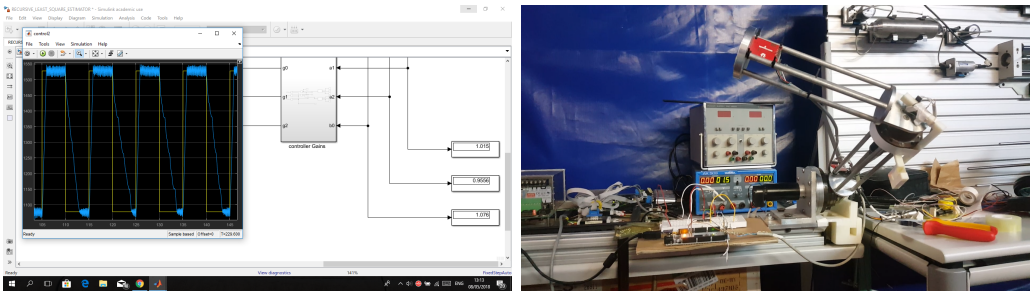


Figure 4.27: Full Adaptive Controller Control Model

4.5.4 Experimental Results

The first test carried out by checking the response of the upper robotic arm by changing the inertia of the robot when the lower arm is moving down. Figures 4.28a and 4.28b show the first response.



(a) First response to adaptive controller **(b)** Real robot response

Figure 4.28: Adaptive Control First Test

It is obvious that we have a vibration in the arm when the control signal achieves the set point. So, it was important to take into account the sampling frequency of the system to be compatible with the PWM signal frequency (268 Hz) of the magnetic position sensor, so the sample period has changed from 0.01 to 0.001s. The result is shown in Figure 4.29.

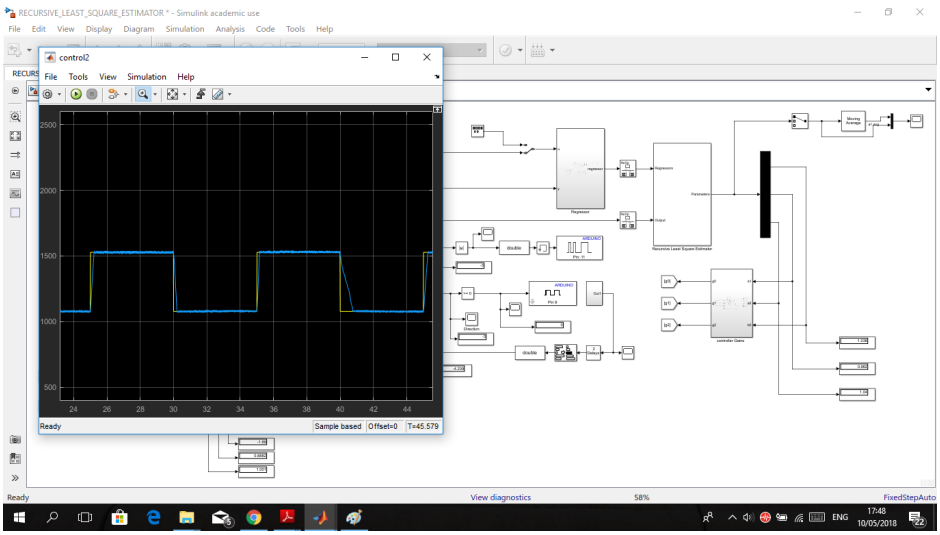


Figure 4.29: Second Response with changing sampling time

Moreover, it is important to detect the inertia changes by checking the output values b_0, a_1 and a_2. The following test has been done and the arm positions and changes of system values are shown Figures 4.30 and 4.31 respectively.

Figure 4.30: Response of robotic arm to the adaptive controller

The graphical representation of values b_0, a_1 and a_2:

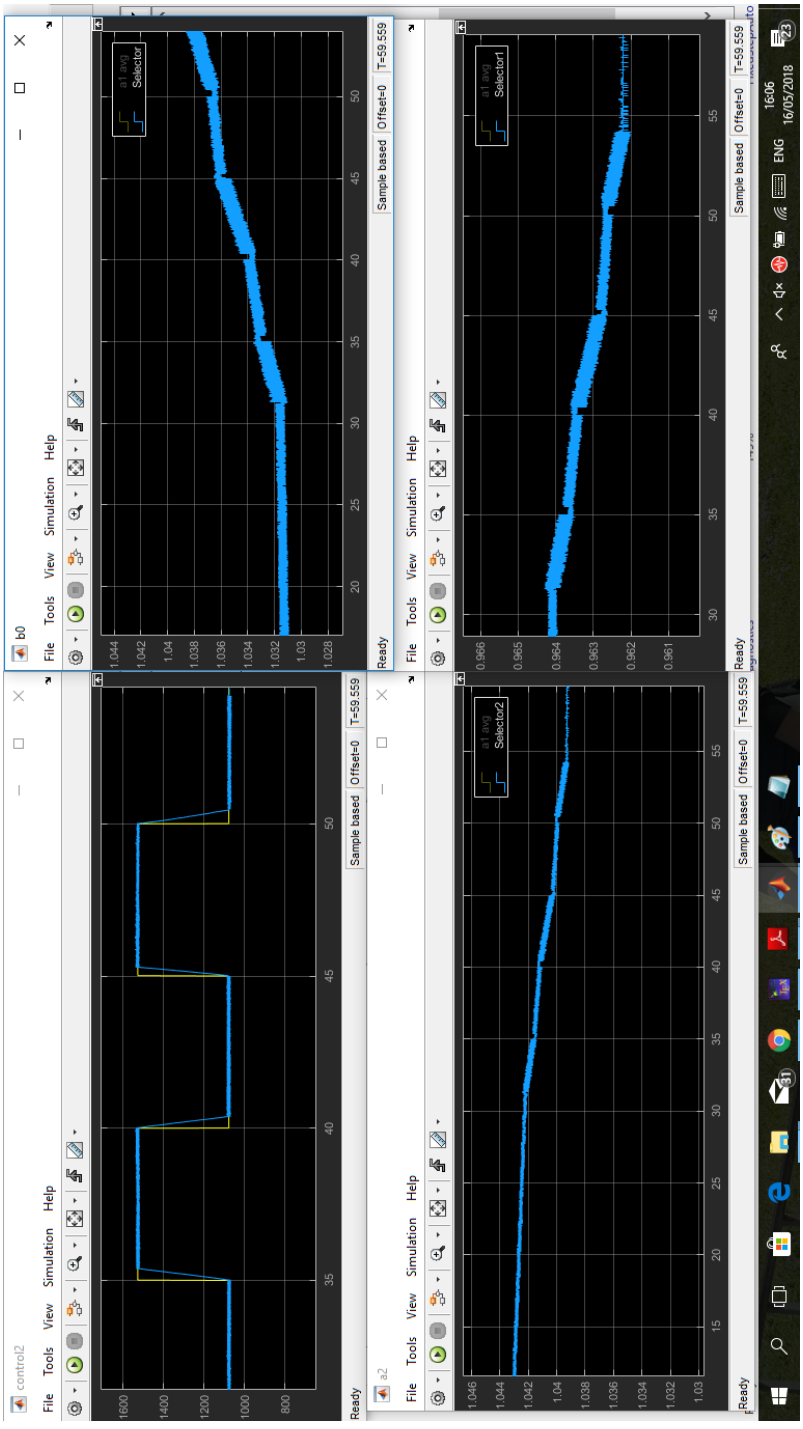


Figure 4.31: b_0, a_1, a_2 and system response to the inertia changes

Chapter 5

Study, Design and Simulation of 3 DoF Robotic Arm for Mobile Robot Usage

5.1 Introduction

Robotics represent a complex area of applications for research and education purposes[13]. The word robot was coined by a Czech novelist Karel Capek in 1920 play titled Rassum's Universal Robot (RUR), and it represents a word for worker or servant. As a general definition of robot, it is a reprogrammable, multifunctional manipulator designed to move materials, parts,

tools or specialized devices through variable programmed motions for the performance of a variety of tasks (Robot Institute of America, 1979).
The main purpose of mobile robots is to build a full autonomous and intelligent mobile robots[13] ready to replace the human's work in dangerous areas and in factories and even to help human in many different field like offices, house work, kitchen, etc...
Nowadays, mobile manipulation robot are a major subject in research and development environment. The mobile manipulation robots are composed of two different systems, manipulation robot arm and mobile robots. This system offers a wide range of workspace for the robotic arm to work in. However, the environment that the robot arm works in is unstructured, which makes it a very difficult task to do.
In the faculty of System Engineering and Automation at Universidad Carlos III de Madrid, engineers are educating with automations specializations in the area of robotics. It is based on the research results from the area of Mechatronics systems and mobile robotic systems to be specific. The research area are focused on automation systems, sensors system, control systems, communications systems, mobile manipulator control.
In Robotics Lab of the Department of System Engineering and Automations, there was a TurtleBot mobile robot, a student Ximena de Diego was developing a voice control for the mobile robot to work inside the robotics lab. The idea was to design a 3 DoF robotic arm to be mounted over the TurtleBot mobile robot and integrate the control of both systems to cooperate together. The systems have to deal with cameras, sensors and actuators to work on both systems synchronously.

5.2 Study of 3 DoF Robotic Arm Parameters and Design

A 3 DoF robotic arm has been chosen to be the target of the study in our project. Figure 5.2 shows the 2D kinematic model representation of the robotic arm that is going to be designed.

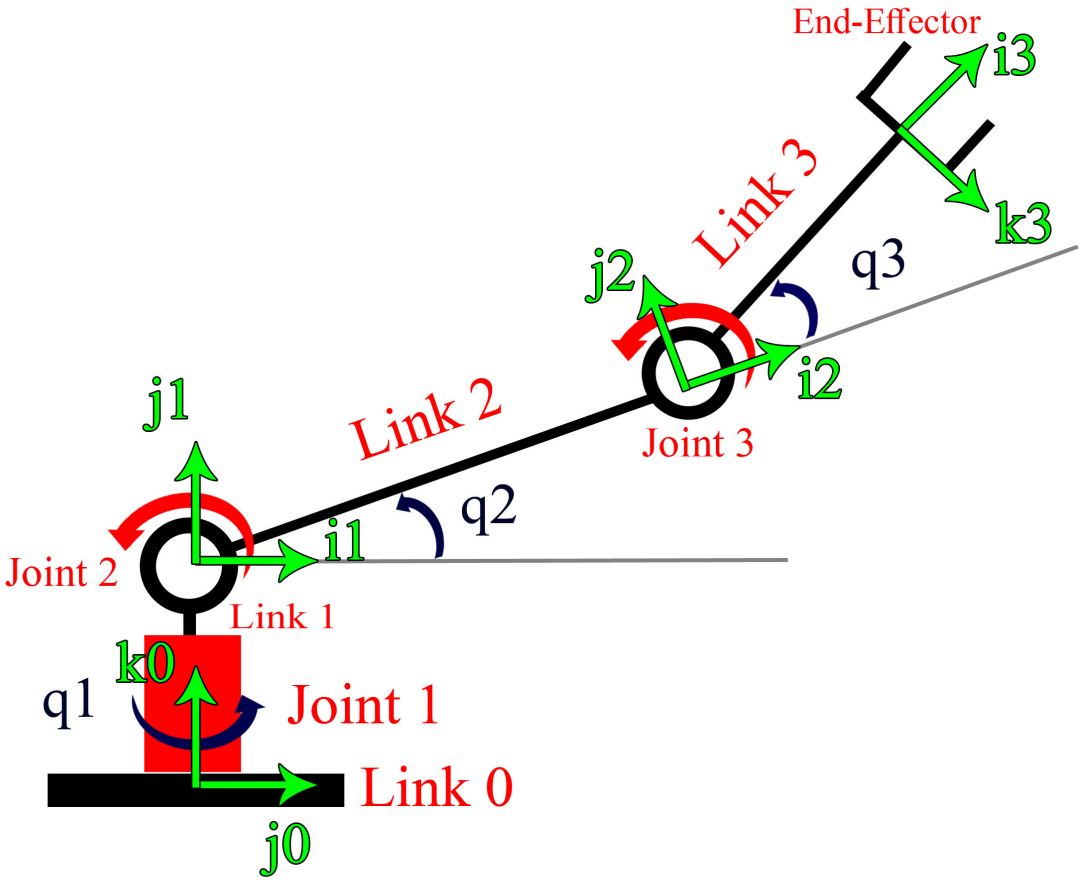


Figure 5.2: Kinematics Model of 3 DoF Robotic Arm

The serial manipulator shown in Figure 5.2 has three joints (n=3), each joint is a revolute joint with 1 DoF. However, each joint connects 2 Links (so number of Links = n+1 = 4 including Link0). As a general form, the number of DoF for any serial manipulator can be found using the following formula:

$$DOF = 6(NumberOfMovableLinks) - 5(NumberOf1DoFJoints) \quad (5.1)$$

Where in our case we have 3 movable links and 3 (1 DoF) joints.
Moreover, defining the parameters of the robotic arm is an important task to know the kinematics of the robot. For that, Denavit-Hartenberg convention has been applied, this convention defines only 4 parameters (2 translations

and 2 rotations) and these parameters are defined as follow (where Ri is the reference frame for the joint i):

1. **Parameter d_i:** it defines the translation along the motion axis k_{i-1}, between the origin of R_{i-1} and the intersection of the axis defined by k_{i-1} and i_i.

2. **Parameter θ_i:** it defines the rotation angle around axis k_{i-1} such that i_{i-1} overlaps i_i. Sign follows the RHR.

3. **Parameter a_i:** it defines the minimum signed distance between axis k_{i-1} and k_i along the common normal, measured along i_i.

4. **Parameter α_i:** it defines the rotation angle arount motion axis i_i such that k_{i-1} overlaps k_i. Sign follows the RHR.

Referring to the robotic arm in Figure 5.2. Table 5.1 shows the parameters that define the kinematic model of the robotic arm.

Link	d_i	θ_i	a_i	α_i
1	0	q1	0	90°
2	0	q2	Link2=30 cm	0°
3	0	q3	Link3=30 cm	90°

Table 5.1: Denavit-Hartenberg parameters for 3 DoF Robotic Arm

Where q_i are the angles of rotations around the axis of motion of each joint.

5.3 Design of 3 Dof Robotic Arm using Solidworks

Figure 5.3: SolidWorks

Solidworks is a 3D CAD software that enables the user to bring their imagination to life. However, this software allows you to create your own design, edit, visualize and make-up any type of CAD drawing. Moreover, it provides the conversion of your designed project into many different formats. One of the main reason of using Solidworks for designing the robotic arm is that solidworks provides an exporter from solidworks to URDF format, this exporter is an add-in that allows the convenient export of Solidworks parts and assemblies into URDF files. The exporter will create a ROS-like package that contains directory for textures, meshes and URDF file that describe the robot parts, dimensions and joints. The exported package is not directly able to be used in ROS (Robotic Operating System). However, some changes that we will talk about in the next section have to be done.
The design of the 3 DoF robotic arm has different parts, these parts are connected with joints (motors) and at the second end of the arm is the gripper.

1. **Base Link:** Base Link is two circular plastic plates connected by Aluminium hollow tubes, this link is intended to hold the entire system. Also, The first motor that controls the rotation of the first link (Link_1) is attached to its upper plate. The design of the base link allows it to carry some weights in order to keep the arm stationary in case of arm centre of mass is far away from the centre of base link.

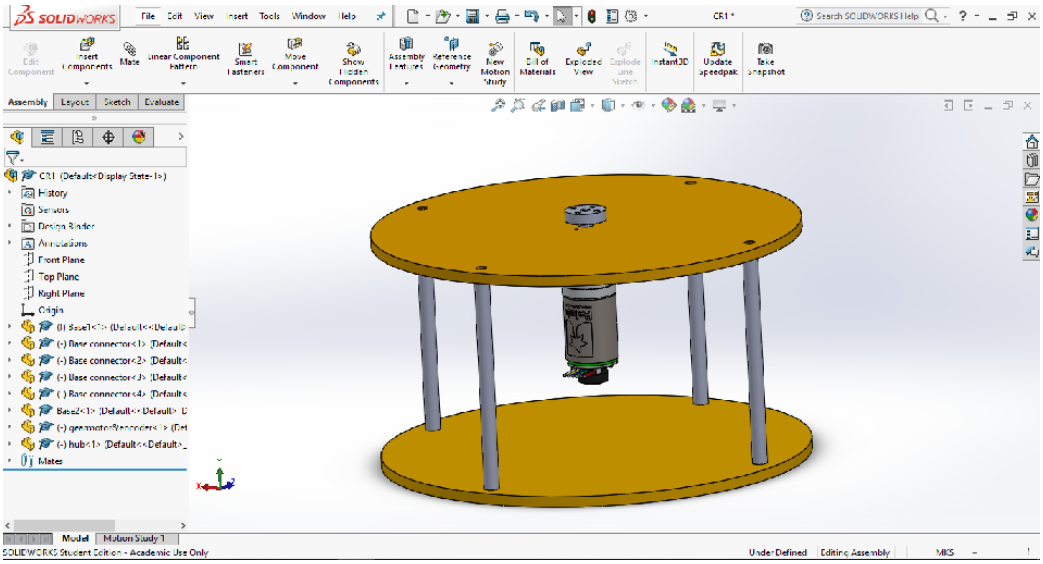


Figure 5.4: Base Link

2. **Link1:** This link connects base_link and link_2, it holds the second motor that is in charge of controlling the rotation of link_2. However, this link is simply a plastic circular plate that rotates the arm in the xy-plane.

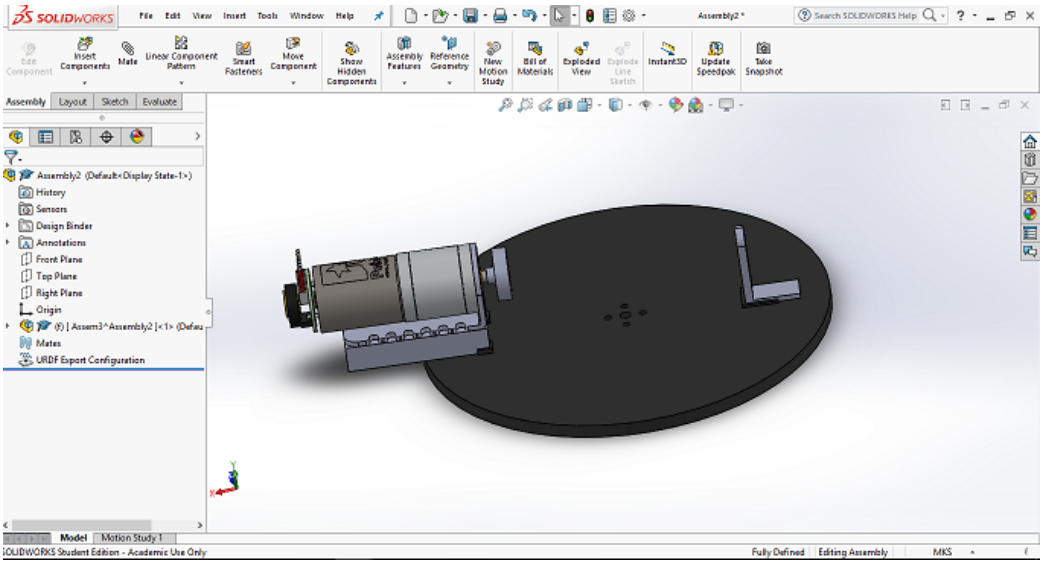


Figure 5.5: Link1

3. **Link2:** Link_2 connects Link_3 with Link_1, it carries the third motor that is in charge of controlling the movement of Link_3. Link_2 is two Aluminium rods connected by 8 Aluminium hollow tubes and 2 ball bearings to allow the rotation of the other sides of Link_2 and Link_3 as shown in Figure 5.6.

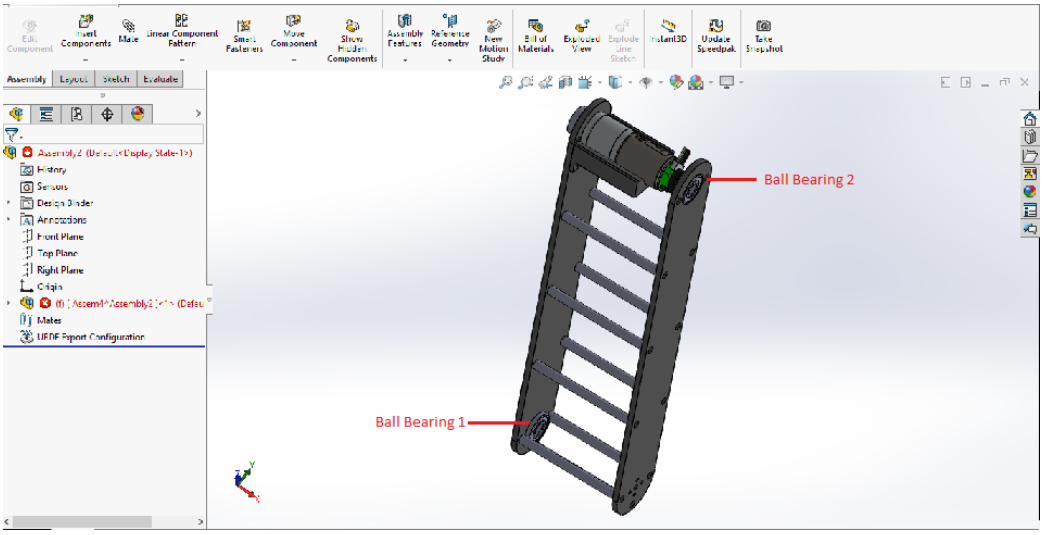


Figure 5.6: Link2

4. **Link3:** Link_3 connects Link_2 with the Gripper, it is made up by 2 Aluminium rods connected by 6 Aluminium hollow tubes as shown in Figure 5.7.

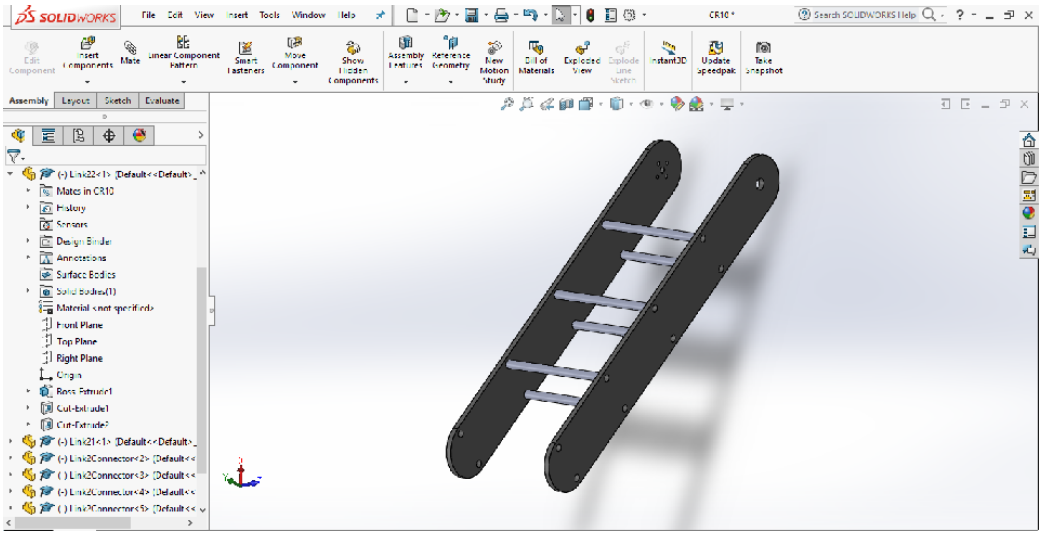
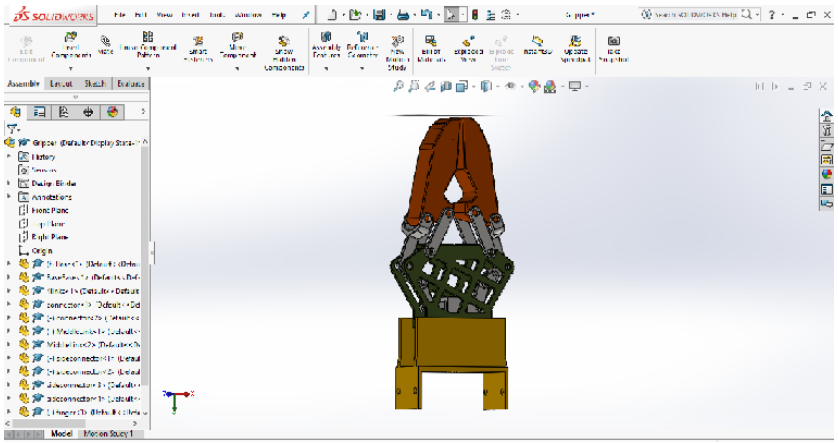


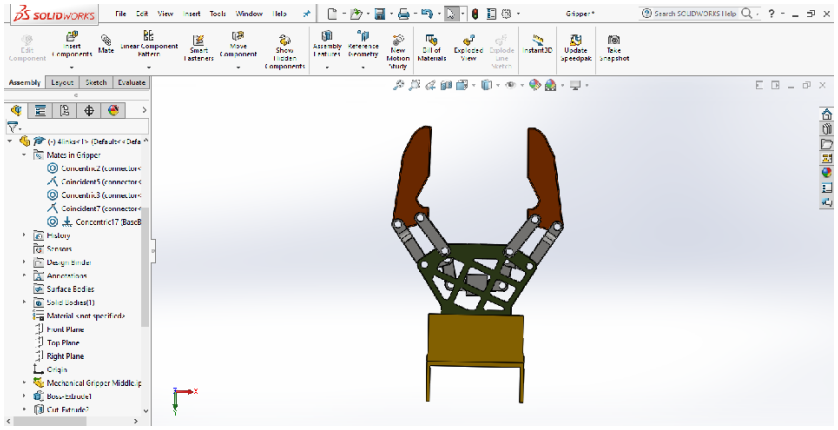
Figure 5.7: Link3

5. **Gripper:** The gripper is the wrist of the robot; it is in charge of grasping objects. The gripper is totally made up by plastic material

and printed out using the 3D printer machine. The design of the gripper contains one brushed DC motors that controls a prismatic joint to open and close the gripper. Figures 5.8a and 5.8b show the design of the gripper when it is opened and closed.



(a) Gripper Close



(b) Gripper Open

Figure 5.8: Gripper

Finally, the design of each part of the robotic arm has been finished and the complete robot assembly is shown in Figure 5.9.

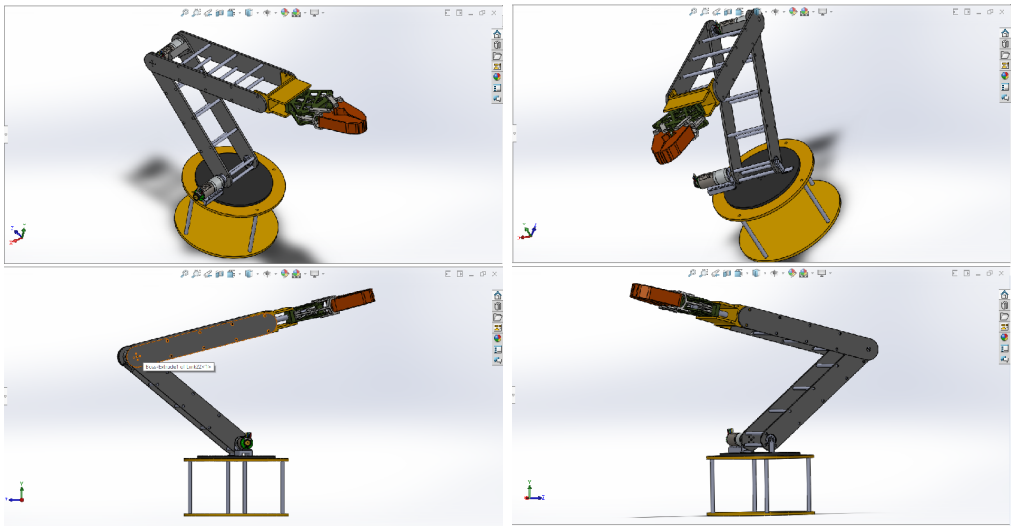


Figure 5.9: Complete Robotic Arm assembly

5.3.1 Robot 3D dimensions

1. **Base_link:**

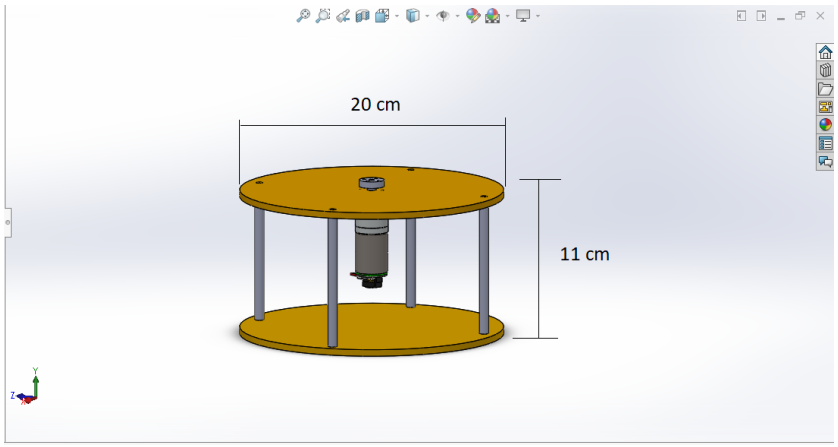


Figure 5.10: Dimensions Base_link

2. **Link_1:**

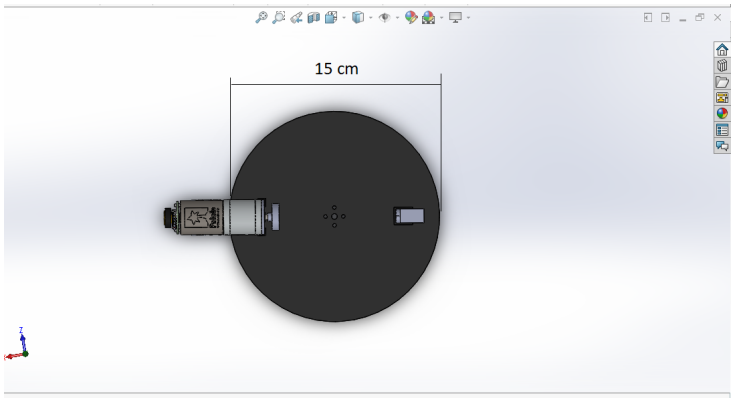


Figure 5.11: Dimensions Link_1

3. **Link_2:**

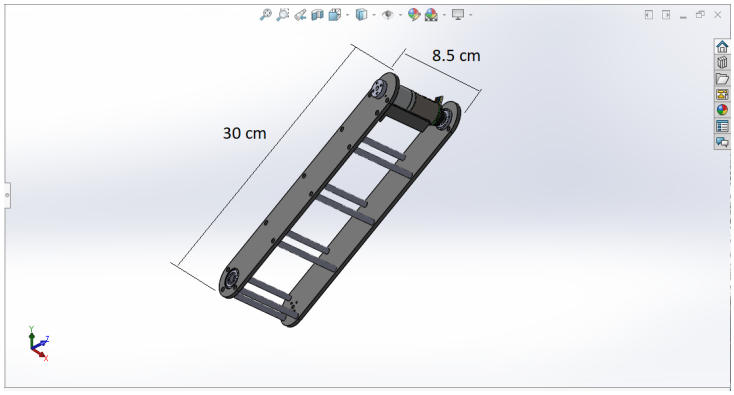


Figure 5.12: Dimensions Link_2

4. **Link_3:**

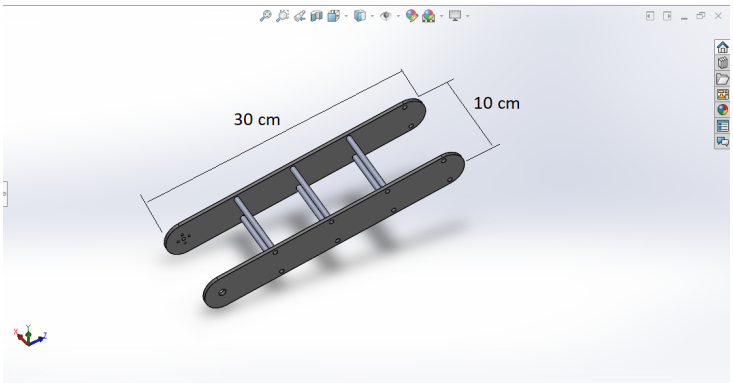


Figure 5.13: Dimensions Link_3

5. **Gripper:**

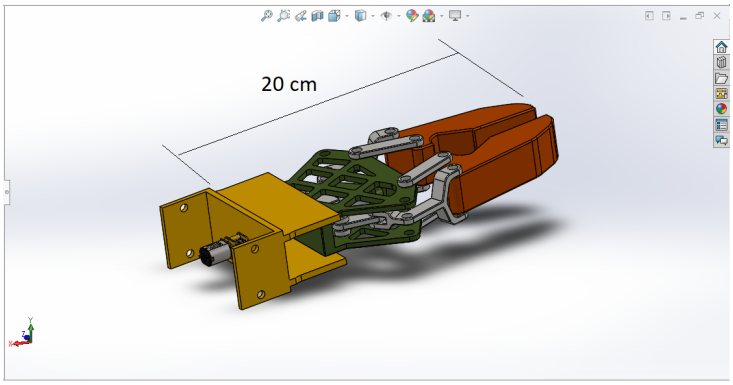


Figure 5.14: Dimensions Gripper

5.4 Exporting SolidWorks project to URDF Packages

Now that the complete robot assembly is done, the robot is ready to be exported as URDF (Unified Robot Description Format) supported by SolidWorks extensions, thanks to SolidWorks to URDF exporter that allows to specify and name the links that make up the robot and to specify the joints that connect two links and choose the type of each joint (Revolute, Prismatic, Continuous)[14].

However, exporting your robot into URDF files will create texture, launch, config, meshes and urdf directories as shown in Figure 5.15. The URDF directory contains a file with .urdf format that characterizes the entire structure and connections of the robot. However, it calls the meshes directory that contains .STL format files that include the 3D models of each part of the robot. Note that naming the entire package should contain only small letters.

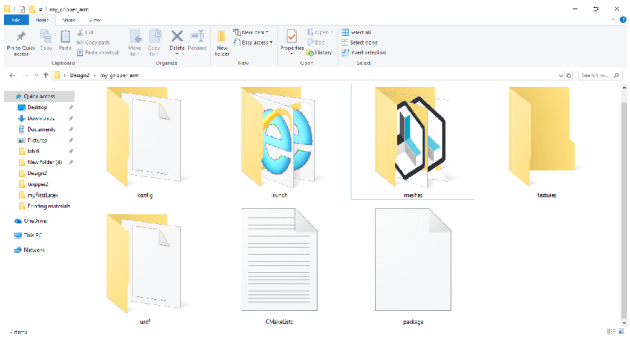


Figure 5.15: SolidWorks to URDF exporter package

On the other hand, the .urdf file created by the exporter is not ready to be used with ROS (Roboting Operating System), because one of the problems of the exporter is that it doesn't give appropriate inertias of the robotic parts and it produces numbers with values 10E-17 and less. Figure 5.16 shows a part of the .urdf file created by the exporter.

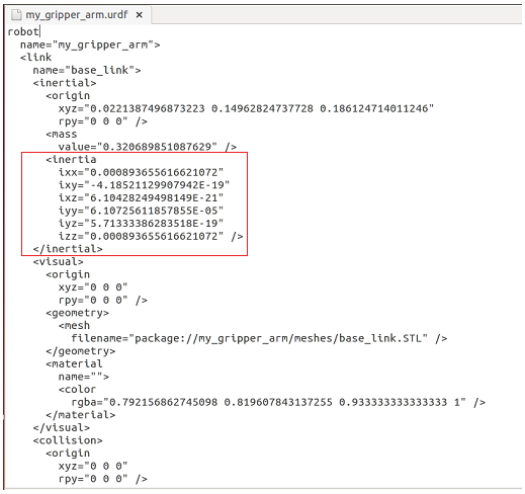


Figure 5.16: Inertias in the exported .urdf file

This problem arises when using the URDF files in Gazebo Simulator, the robot will collapse. However, changing the inertias requires a software able to read .STL model files included in the mesh directory and also able to measure and compute the interias of these files. The software used for this target is **MeshLab**, this software is an open source which provides several tools for meshes. However, the tool we used to find the inertias of the models is "Compute Geometric Measures" which calculates the *Principal axes matrix* as shown in Figure 5.17.

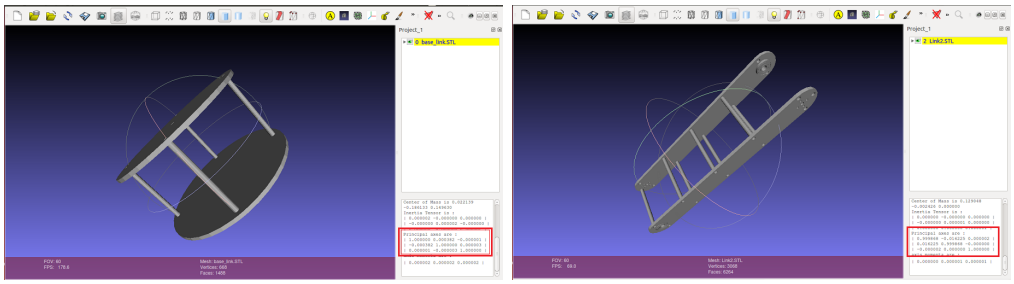


Figure 5.17: Pricipal Axes Inertia Matric Created By MeshLab

The Principal axes matrix form is the following:

$$\begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

The inertia values are taken from the principal axis matrix and replaced in the .urdf file directory.

In addition, it is also important to pay attention for adding joint limits (maximum and minimum rotation for the joint) in the urdf file in order to make your robot movable. The urdf file comes with zero values for the joint limits which means the joint can not rotate. Since we have chosen Revolute joints, they should have limits for rotation (Upper limit and Lower limit) unlike the continuous joint that rotates 360° continuously. The .urdf file automatically comes with joint limits tag that includes *effort:* which means the maximum force supported by the tagged joint, *lower and upper:* to assign for the lower and upper limit for joint rotation respectively and they are measured in radian/revolute joints, and *velocity:* which enforces the joint to a maximum velocity.

5.4.1 Visualizing the Robot 3D Model on Rviz

Rviz is a 3D visualization environment, it lets us view what the robot is doing, seeing and thinking. Rviz lets you look at the world through the robot size through cameras, lasers and coordinate frames. It is a powerful tool for debugging robot applications[14].

Before visualizing the 3D model on Rviz, we can check graphically whether we have any error or any syntax mistyping by using the *urdf_to_graphiz* which generates a pdf file with the structure of robot as shown in Figure 5.19.

Now that we made sure that there are no errors in the urdf files, we can visualize the robot 3D model on Rviz by running display.launch file created by Solidworks_to_URDF exporter; this file contains joint_state_publisher GUI (Graphical User Interface) that will open an Rviz interface window with sliders as shown in Figure 5.18, each slider controls one joint movement according to the limits defined in the urdf file[14].

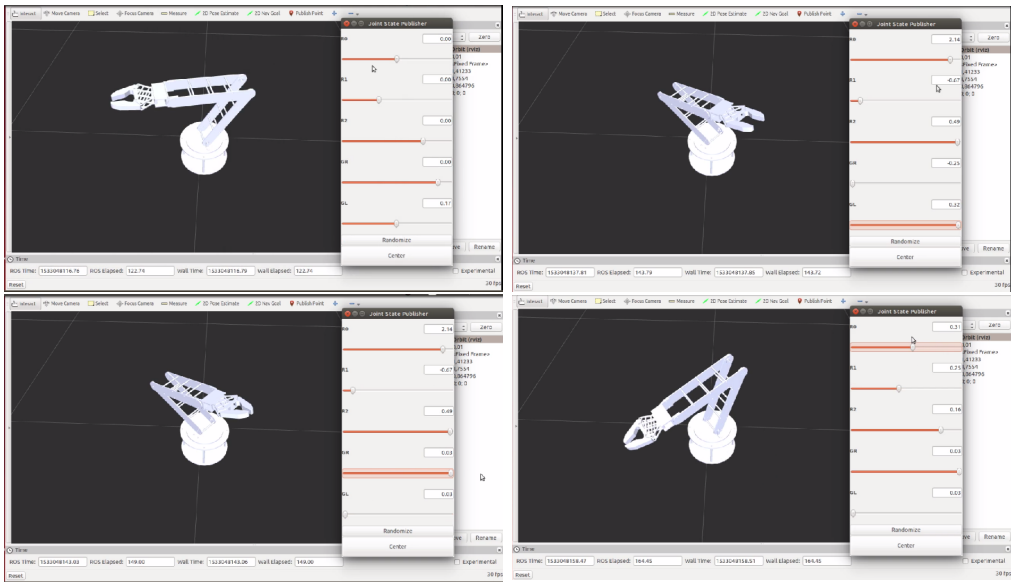


Figure 5.18: Visualization of Robot 3D model on Rviz

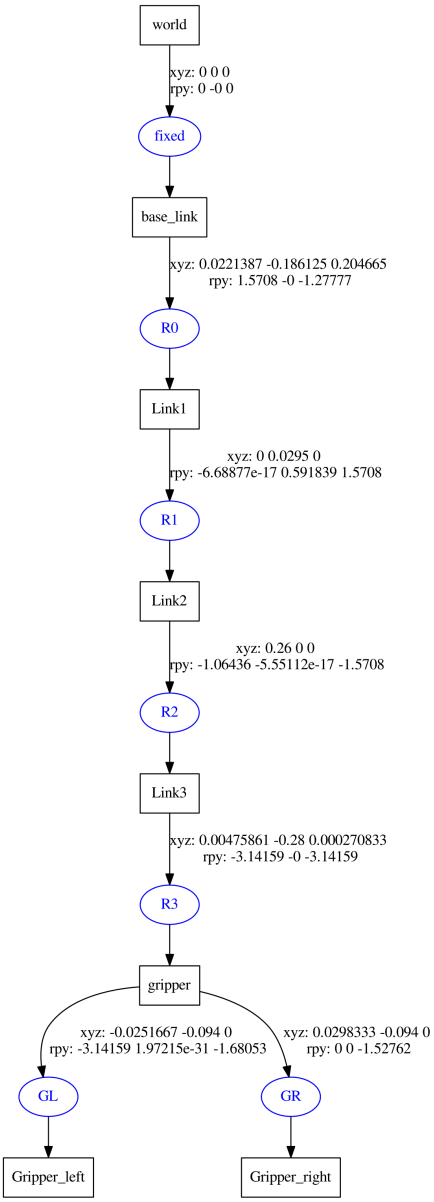


Figure 5.19: Graphical representation of robotic arm

5.5 Simulation of Robotic Arm in ROS

5.5.1 Integration of ROS with Gazebo Simulator

The simulation of robotic arm in ROS has been done using GAZEBO.

Figure 5.20: GAZEBO LOGO

Gazebo is a 3D simulation software that allows robotic developers to simulate their robots in a realistic way. Moreover, simulating the model movement and interaction requires the integration of Gazebo with ROS, in this way ROS is used as an interface for the robot[14].

In addition, Gazebo allows using multi-robots simulations, adding sensors, cameras and objects in 3D world. However, the data generated by the sensors are realistic[14].

As Gazebo is now an independent simulator from ROS, the integration of ROS with Gazebo requires a set of ROS packages called gazebo_ros_pkgs, these packages provide necessary interfaces for simulating the robot in Gazebo using ROS services, messages, plugins and dynamic reconfigures. Figure 5.21 shows the entire packages supported by gazebo_ros_pkgs[14].

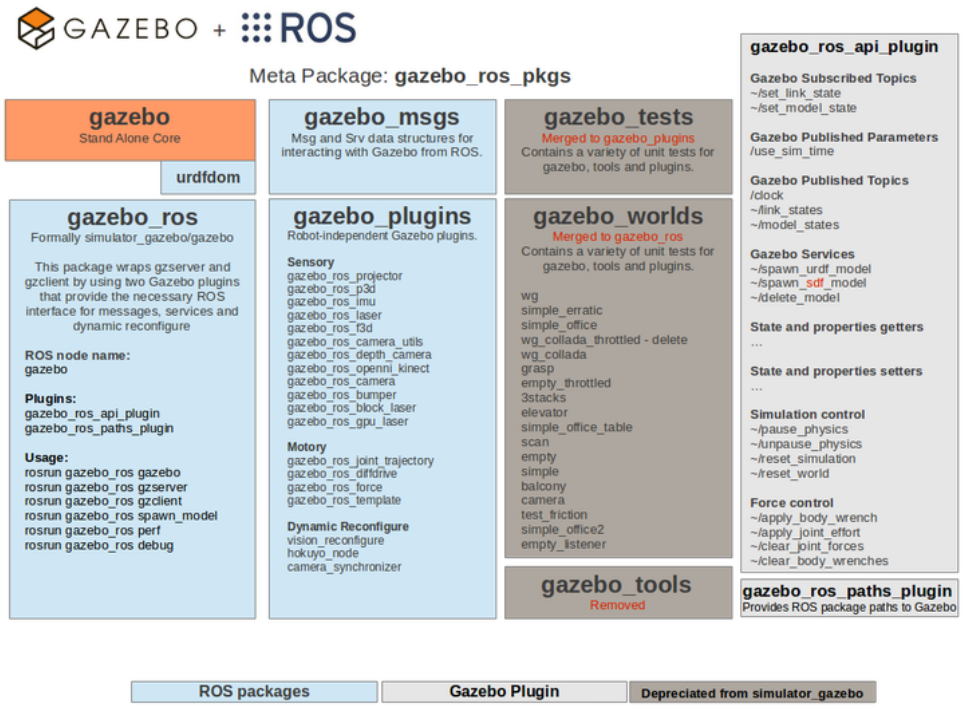


Figure 5.21: Gazebo ROS Interfacing Packages

5.5.2 GAZEBO-ROS Control

After interfacing ROS with Gazebo, ROS requires a control packages that will set-up different controllers to actuate the joints of the robot; for that, ROS-control package provides controllers for simulating the robot in Gazebo with simple gazebo plugin adapters. Figure 5.22 shows the relationship between the simulations, controllers, transmission and hardware.

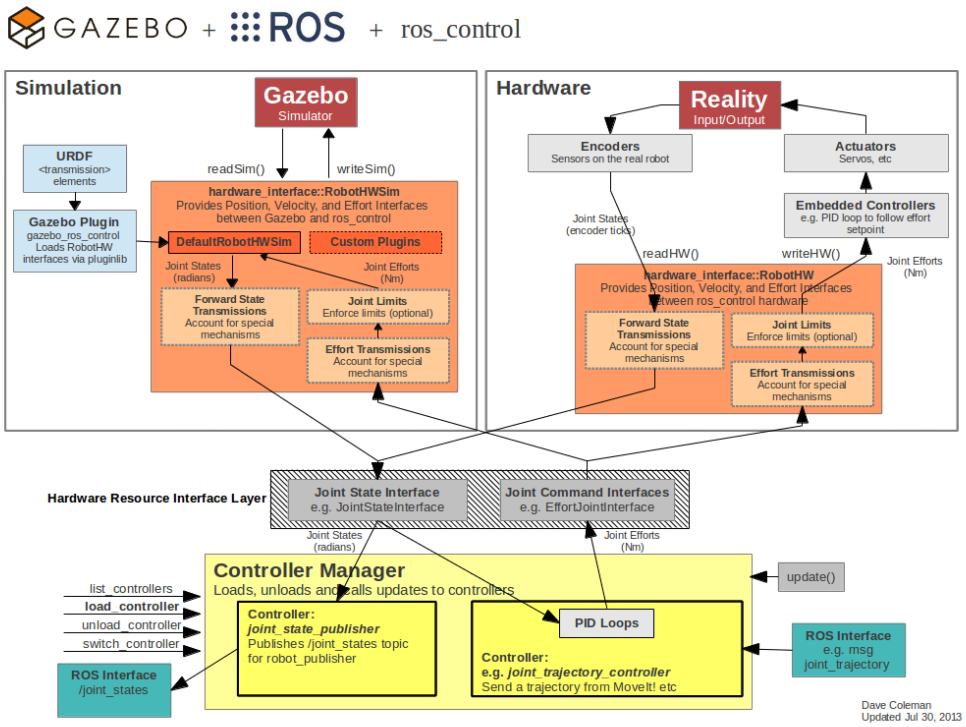


Figure 5.22: Relationship between Gazebo, ROS and ros-control

5.5.3 Usage of ROS-Control with URDF File

In order to be able to use the ros-control in your robot, several elements have to be added to the URDF file of your robot[14].

1. **Adding Transmission Elements To The URDF File:** The transmission element is responsible for adding actuator for each joint by defining the joint name, type of transmission and hardware interface (position, velocity or effort interfaces). Effort interface are used for our robot as shown in Figure 5.23.

```
<transmission name="tran3">
<type>transmission_interface/SimpleTransmission</type>
<joint name="R2">
<hardwareInterface>EffortJointInterface</hardwareInterface>
<joint/>
<actuator name="motor3">
<hardwareInterface>EffortJointInterface</hardwareInterface>
<mechanicalReduction>1</mechanicalReduction>
<actuator/>
<transmission/>
```

Figure 5.23: Transmission element in URDF file

2. **Adding gazebo-ros-control Plugin:** A Gazebo plugin has to be also added to the URDF file, the plugin parses the transmission tags and loads the appropriate hardware interface and controller managers. The default plugin added to the URDF file is shown in Figure 5.24.

```
<gazebo>
<plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
<robotparam>/robot_description</robotparam>
<!--<robotNamespace>/ROBOT</robotNamespace>-->
<!--<updateRate>100.000</updateRate>-->
<plugin/>
<gazebo/>
```

Figure 5.24: Gazebo Plugin

3. **Creating a .yaml Control File:** A .yaml file has to be created in the config directory in order to define the PID controller gains and the control settings for the joints. This file has to be called by a launch file that will be created in the next step. The .yaml file is shown in Figure 5.25.

```
#Publish all joint states
joint_state_controller:
type: joint_state_controller/JointStateController
publish_rate: 50

#Position Controllers
R0_position_controller:
type: effort_controllers/JointPositionController
joint: R0
pid: {p: 100.0, i: 0.1, d: 10.0}

R1_position_controller:
type: effort_controllers/JointPositionController
joint: R1
pid: {p: 100.0, i: 0.1, d: 10.0}

R2_position_controller:
type: effort_controllers/JointPositionController
joint: R2
pid: {p: 100.0, i: 0.1, d: 10.0}

GL_position_controller:
type: effort_controllers/JointPositionController
joint: GL
pid: {p: 10.0, i: 0.01, d: 1.0}

GR_position_controller:
type: effort_controllers/JointPositionController
joint: GR
pid: {p: 10.0, i: 0.01, d: 1.0}
```

Figure 5.25: Joint position PID control of robot joints

4. **Creating a Launch File:** Now that the controller settings and the PID gains have been defined, a launch file has to be created in the Launch directory. The launch file calls the .yaml control file and spawn the urdf model to open in Gazebo. The launch file is shown in Figure 5.26.

```
<?xml version="1.0" encoding="UTF-8"?>

<launch>

<param name="robot_description" command="cat '$(find
my_gripper_arm)/urdf/my_gripper_arm.urdf'" />

<arg name="x" value="0.0" />

<arg name="y" value="0.0" />

<arg name="z" value="0.0" />

<!-- load joint controller confgurations from YAML file to parameter server -->

<rosparam  file="$(find my_gripper_arm)/config/my_gripper_arm_control.yaml"
command="load"/>

<node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false"

 output="screen" args="arm_controller GL_position_controller GR_position_controller
joint_state_controller"/>

<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
output="screen" args="-urdf -model my_gripper_arm -param /robot_description -y -0.1"/>

<!-- now convert joint_states to TF transforms for Rviz, etc -->

<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen">

<remap from="/joint_states" to="/joint_states" />

</node>

</launch>
```

Figure 5.26: Arm Control Launch File

The PID gains chosen in the .yaml control file are random values. Consequently, rqt_reconfigure has to be used to test if the chosen PID gains are good for the robotic arm behaviour. The rqt_reconfigure allows us to calibrate the PID gains on runtime until we find the correct behaviour of the robot. With the PID gains chosen initially in the .yaml control file, the robotic arm showed a bad behaviour; it was unstable and the rotations of the links were affected by vibrations. After tuning the PID gains, we found out that p=1000, i=10 and d=100 are good gains for the robotic arm behaviour. The rqt_reconfigure command window is shown in Figure 5.27.

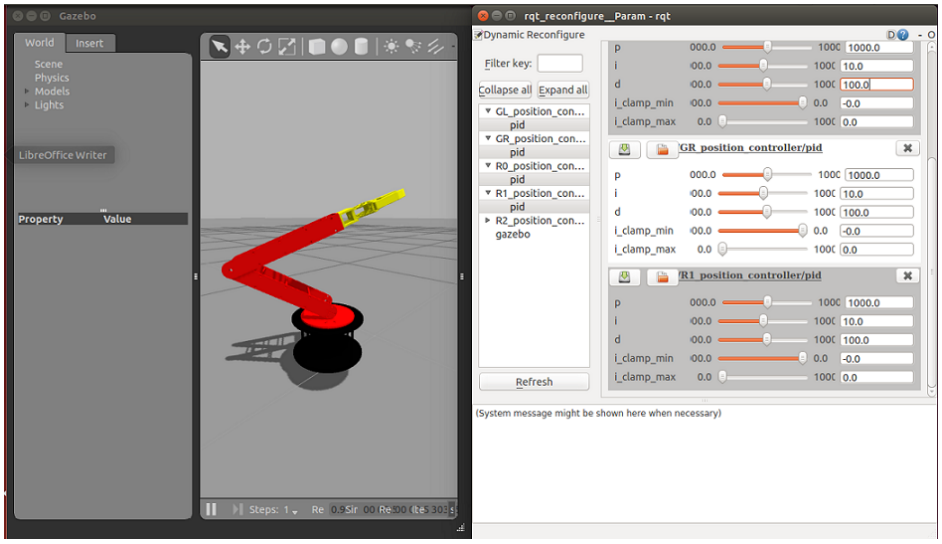


Figure 5.27: Tuning PID gains

Controlling Joints Position Manually

Now that the PID gains have been tuned, we can control each joint position manually by publishing a *joint command* on the joint topic. The joints topics are available after running the robot control launch file, the topic list is given in Figure 5.28.

```
^Cwrg@wrg-laptop:~$ rostopic list
/GL_position_controller/command
/GL_position_controller/pid/parameter_descriptions
/GL_position_controller/pid/parameter_updates
/GL_position_controller/state
/GR_position_controller/command
/GR_position_controller/pid/parameter_descriptions
/GR_position_controller/pid/parameter_updates
/GR_position_controller/state
/R0_position_controller/command
/R0_position_controller/pid/parameter_descriptions
/R0_position_controller/pid/parameter_updates
/R0_position_controller/state
/R1_position_controller/command
/R1_position_controller/pid/parameter_descriptions
/R1_position_controller/pid/parameter_updates
/R1_position_controller/state
/R2_position_controller/command
/R2_position_controller/pid/parameter_descriptions
/R2_position_controller/pid/parameter_updates
/R2_position_controller/state
/calibrated
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/rosout
/rosout_agg
/tf
/tf_static
```

Figure 5.28: Joint Position Controllers

However, the data (joint commands) will be published on the pre-defined joint topics with general form /"joint_name"_position_controller/command, where "joint_name" is replaced by the joint name that we want to publish data on. Figure 5.29 shows the response of the robotic arm.

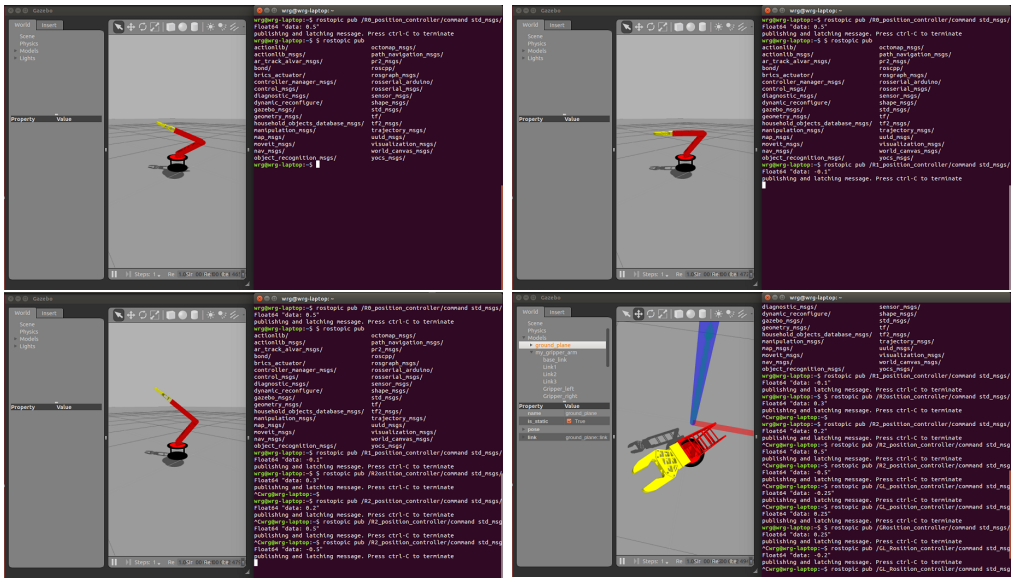


Figure 5.29: Manually controlling robot joint position

Publishing Commands Through GUI Window

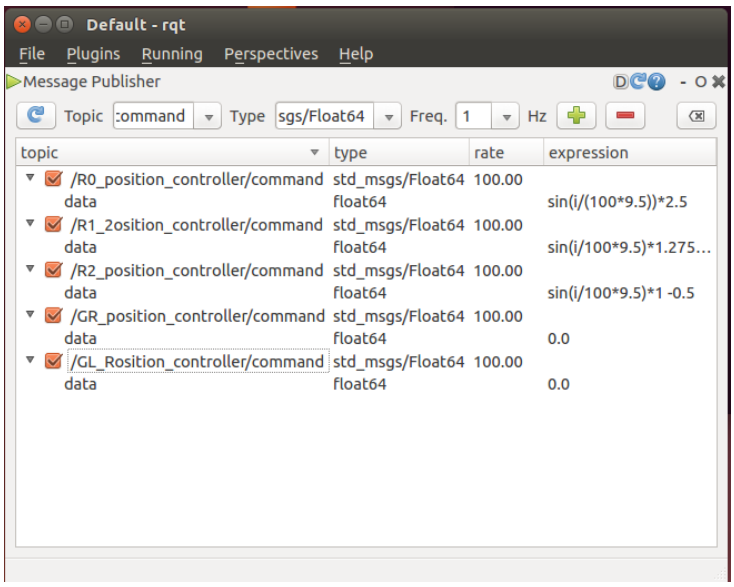


Figure 5.30: GUI message publisher

The GUI (Graphical User Interface) window allows us to send joint commands very fast. Moreover, it provides a way to move all the robot joints at the same time (either by publishing constant commands or by waves), also it is possible to publish a sine wave command at the robot's exact joint limits by publishing expression (5.2) as a command.

$$sin(i/rate * speed) * diff + offset \tag{5.2}$$

where:

- **i:** The RQT time variable
- **rate:** the frequency that the sin wave is evaluated. usually 100 is used.
- **speed:** speed of the joint actuation.
- **diff:** upper_limit - lower_limit /2.
- **offset:** upper_limit - diff.

The results of the sine wave commands to the robot joints through the GUI window message publisher are shown in Figure 5.31.

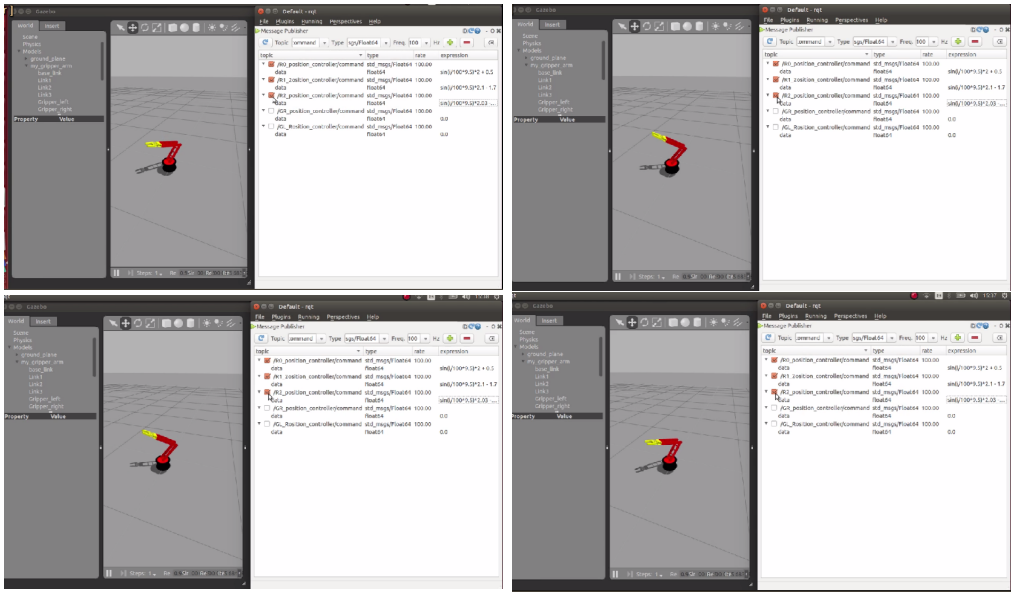


Figure 5.31: GUI Joints Message Publisher

Connecting Gazebo with Rviz

After using ROS-control to send commands to the robot in real simulation Gazebo, it is also important to see what our robot is doing in visualizing tool Rviz. With ROS_control Joint State Controller, Rviz can read the robot's state from the real simulation gazebo and broadcast the state in Rviz. This step is important for future work when we want to connect the real robot hardware with ROS, then we should be able to visualize the robot movements and actions in real life in the same way we are using it in the real simulation Gazebo. Figure 5.32 shows the movements in Gazebo and at the same time visualizing the same movements on Rviz.

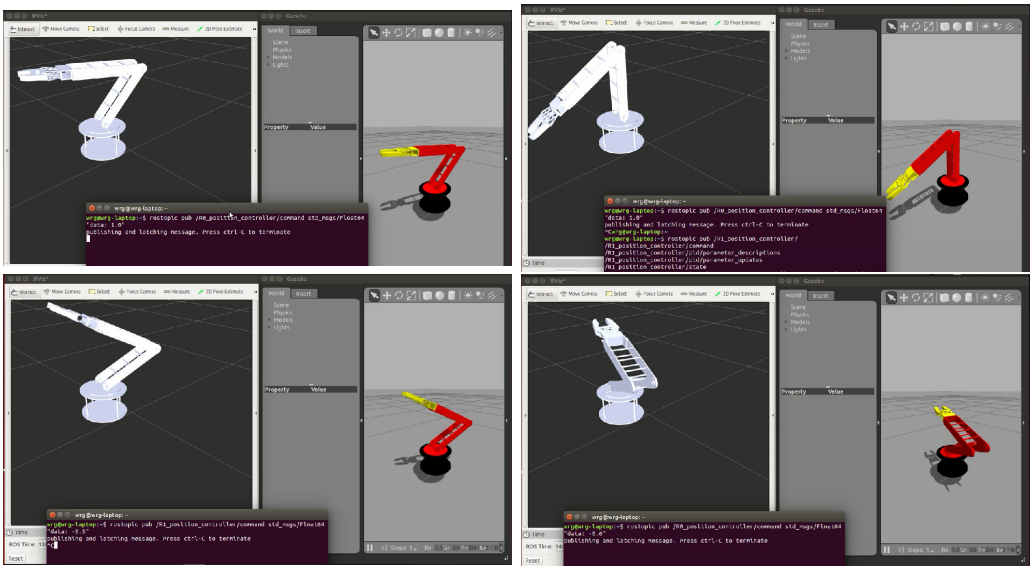


Figure 5.32: Robot Control with Gazebo and Rviz

Chapter 6

Robot Arm Motion Planning with Moveit! and Gazebo

6.1 General Overview

Figure 6.1: Moveit! Software Logo[14]

Moveit! is a software for building robot applications, integrating the state of art and motion planning, kinematics, collision checking and dynamic 3D environment representation[14].
The Moveit! setup assistant allows you to easily configure Moveit! for any robot using GUI. The generated founds by setup assistant allow you to start visualizing and interacting with the Robot in Rviz quickly[14].
In this chapter, we are going to see how to configure Moveit! for our own robot, which is a very simple task provided that the URDF file that describes our robot is available. However, we will also use the Moveit! configuration tools to define motion planning and we will see how to plan and execute motion for the robot joints whether by using Moveit! planning group or programmatically using C++ or Python code.

Moreover, what we call motion planning, is to be able to move the end_effector from its current position into a desired one, which is a non trivial task, since every joint in the robot should follow a sequence of values in coordination with other joints[14].
As it is simpler for robots with high degrees of freedom to be able to reach a specific pose for its end_effector, and as Moveit! deals with high degrees of freedom robots, some restrictions regarding planning of the end-effector to a specific poses and grasping actions has occurred, and this is due to the low degrees of freedom we have in our robot. Figure 6.2 shows the Moveit! architecture.

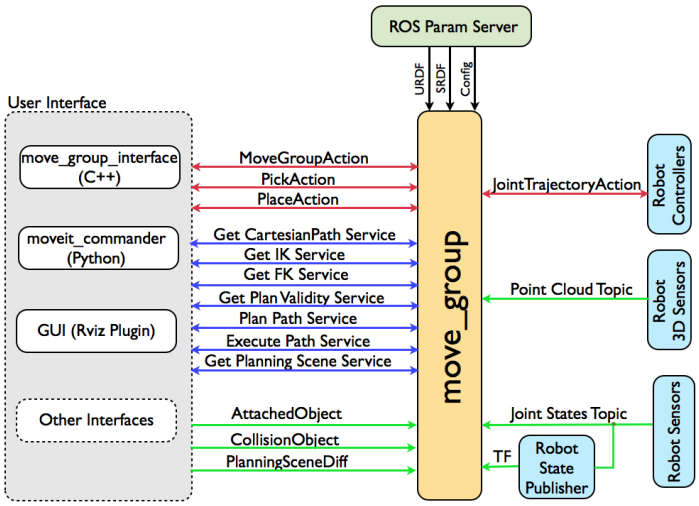


Figure 6.2: Moveit! Architecture[14]

We can see at the centre of Figure 6.2 the move_group element, the idea is to define joints groups to implement moving actions using motion planning algorithms.
Using definition languages (URDF, SDF or YAML) and with Standard ROS tools, a group can be defined. Two joints groups have to be defined, the first one is the group that define the joints of the arm chain with the joint limits, and in the same way, the joints for the end_effector group has to be defined. And then in order to get the output of the motion planning on the robot simulator (Gazebo in our case) or even a real hardware robot, the controller to be used is *JointTrajectoryAction*. Moreover, the /joint_states tool is also needed to monitor the execution of the plans by means of the robot state publisher.

6.2 Steps to Integrate the Robotic Arm In Moveit!

In this section we are going through different steps in order to get our robotic arm working with Moveit!. As mentioned in the previous section, the URDF file that describes the robot or a Collada file should be available for the robot. Moreover, to get the robot working also with gazebo, some additional files have to be added to the created package by Moveit!.

1. **Opening Up Moveit! Setup Assistant Window:** After Moveit! Setup Assistant is launched, it will request the URDF or Collada file that describes you robot as shown in Figure 6.3. Then once the robot model is successfully launched we will go through different steps to define robot groups and positions as we will see in the next steps.

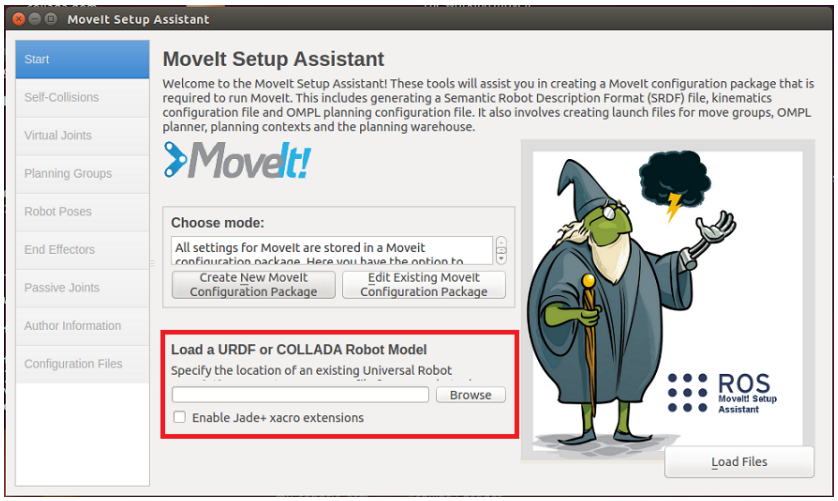


Figure 6.3: Moveit! setup assistance

2. **Self Collision:** Fortunately, Moveit! Setup Assistant provides a collision checking between the robot links by generating a *Collision Matrix*; this matrix contains information about when the links of the robot collide to improve the performance of the motion planning.

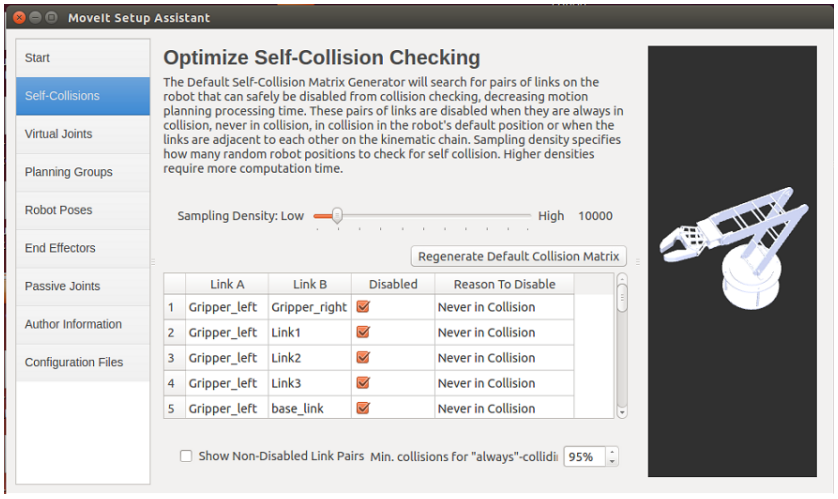


Figure 6.4: Moveit! Collision Matrix

3. **Adding Virtual Joint:** Adding virtual joint is not important in our case, because this joint represents the motion of the robot base in a plane, but we have a fixed-base robotic arm. This joint is usually used for mobile robots.

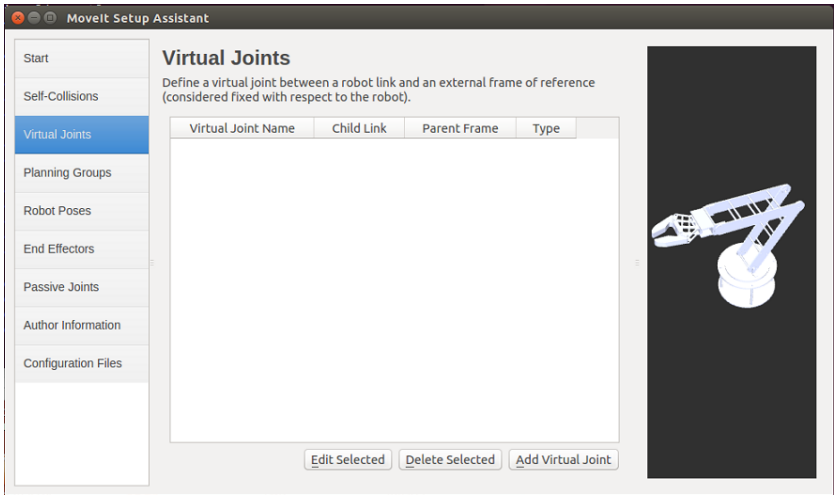


Figure 6.5: Virtual Joint tab

4. **Planning groups:** In this step we are going to define two planning groups (arm and gripper) for the robotic arm (Figure 6.6) by defining the joints that build up each group, also we are going to define the Inverse Kinematic solver provided by the setup assistant. This solver

is in charge of finding Inverse Kinematic solution when we plan to move the gripper from one position to another.

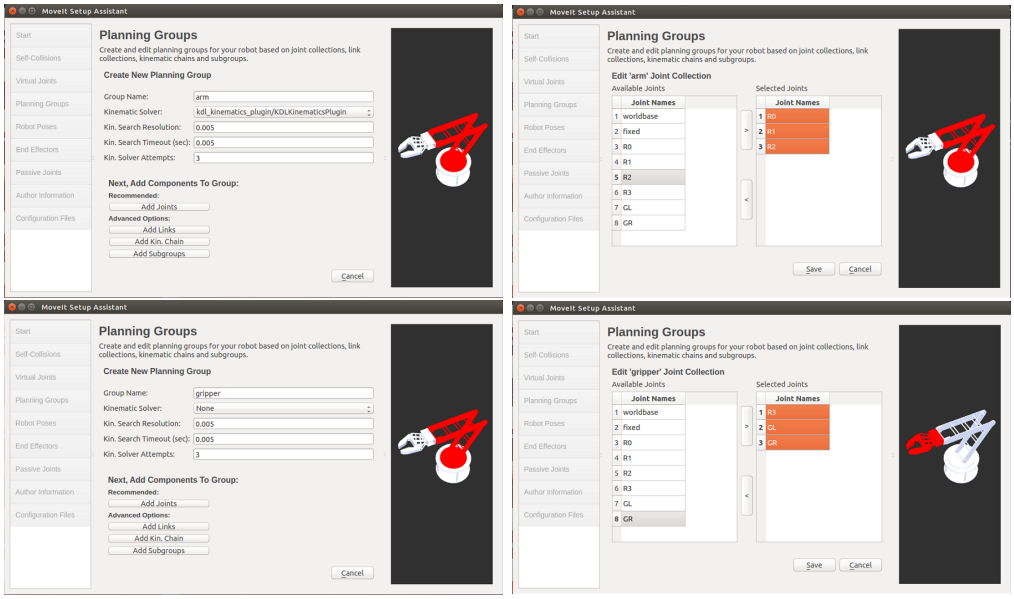


Figure 6.6: Planning Groups for Robotic Arm

5. **Robot Poses:** This step allows us to define known poses of the robot arm and gripper in order to be able to call them in later work. These poses are quite important especially when working with real robots because it is crucial to define a start or home position for the robot where it is safe to store it. However, we can define poses for the gripper in open and close mode as we can see in Figure 6.7.

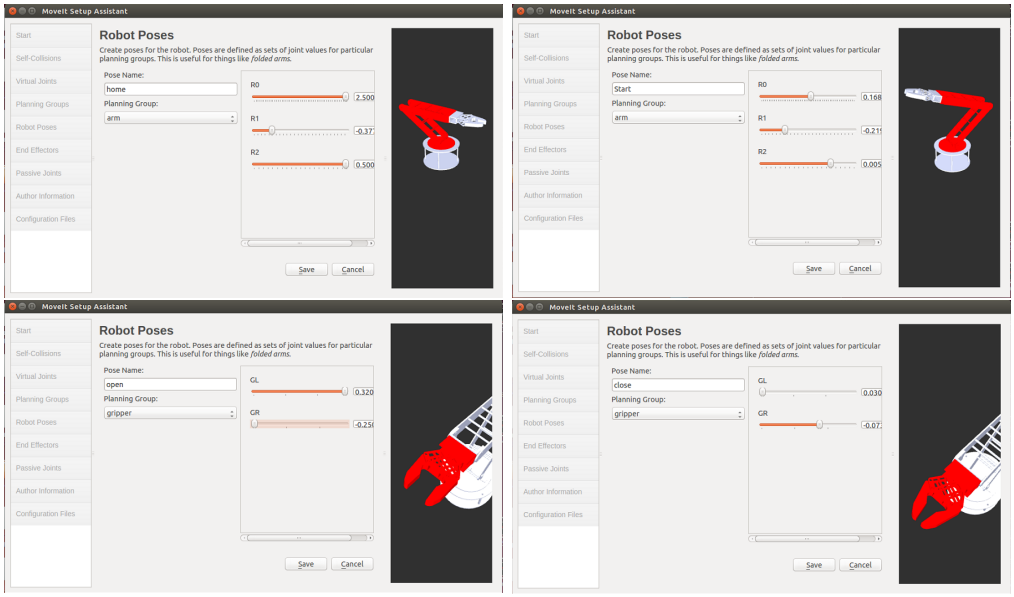


Figure 6.7: Robotic Arm Pre-defined Poses

6. **Defining Robotic Arm End-effector:** So as it is known, every robotic arm has an end-effector in charge of doing different tasks. In this step we are going to define the end-effector to our robot by defining its name, related group and the parent link connected to it.

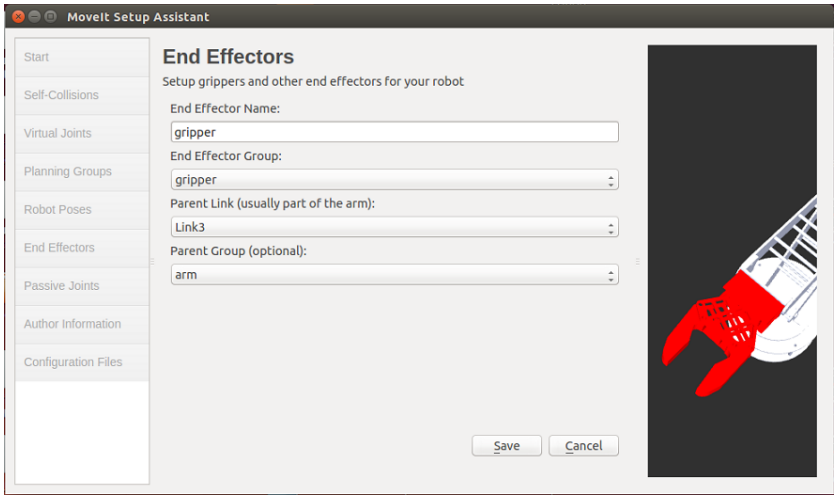


Figure 6.8: Gripper End-effector

7. **Specify Author Information:** The information specified in this step are not necessarily true. We can specify different invalid names and email address, but without this step the package is not going to be generated, because Moveit! requests author details for its package.xml file. However, the information we defined are correct.

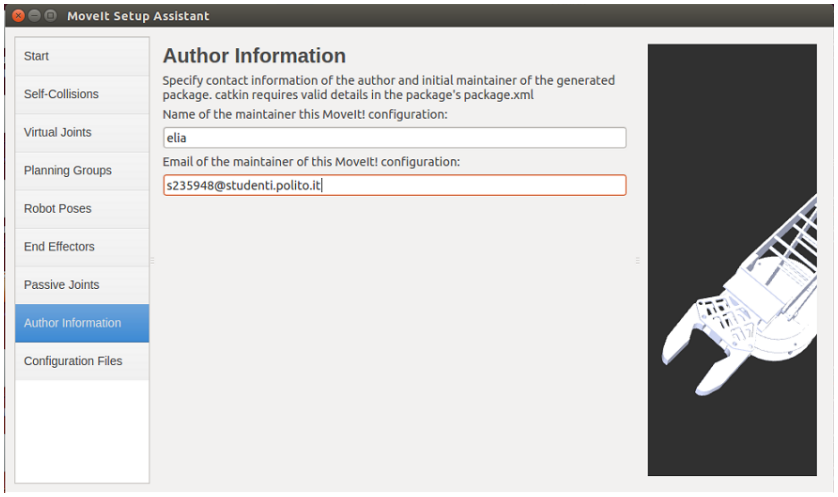


Figure 6.9: Author Information

8. **Generating The Package:** The package to be generated should be named in the following form *"Robot_Name"_moveit_config*, so our robot is named my_gripper_arm and the generated package has the name my_gripper_arm_moveit_config.

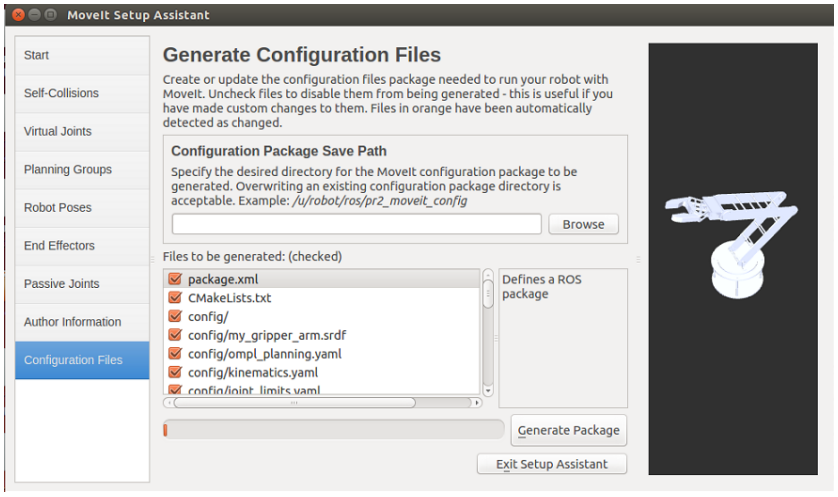


Figure 6.10: Moveit Setup Package

6.3 Integrating the Robotic Arm Moveit! Package Into Rviz

Rviz plugin provided by Moveit! allows the user to perform various actions, such as planning different goals, adding and removing objects from the scene and so on.
The setup assistant generates different launch files, and the one in charge of launching Rviz plugin, fake controllers and Moveit! as well is called demo.launch.
However, when Rviz is successfully launched, a "planning" and "scene objects" tabs are going to appear in the Rviz interface, the Planning tab is in charge of planning different goals, also executing the planned goals if Moveit! Rviz is connected with the real simulator Gazebo or with the real robot. Consequently, we know that Moveit! provides two different functions, the first one is to create the plan from current position into goal position and see how the arm is going to move (See Figure 6.11), and the second one is to send the plan to the robot in simulation (or in the real robot). In this section we are going to deal with the first function.

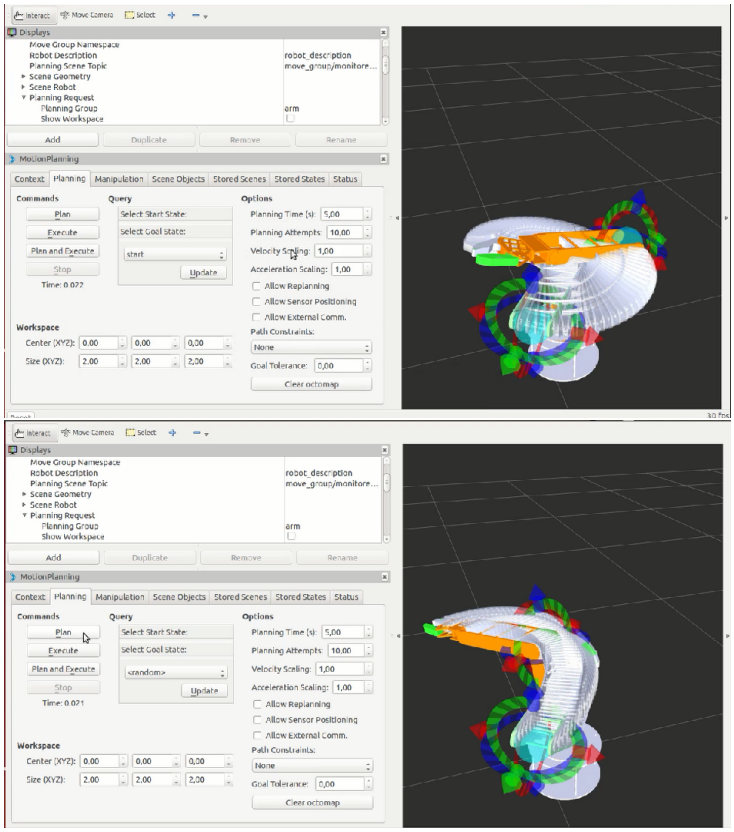


Figure 6.11: Planning into desired goal in Moveit! Rviz

6.4 Integrating the Robotic Arm Moveit! Package Into Gazebo

After we have performed the plan for the robot and we visualize the plan in Moveit! Rviz, it is time to send the plan to the robot in real execution; this step requires defining the controller to be used for the manipulator joints (arm and gripper) in .yaml file inside Moveit! config directory. In this case, the configured controller is *JointTrajectoryController* provided by ROS_control package, because the output messages provided by the motion plans are of that type of controllers. Consequently, we need to define two controllers: arm and gripper as shown in Figure 6.12.

```
Controller_list:
  - Name: arm_controller
    action_ns: "follow_joint_trajectory"
    type: FollowJointTrajectory
    default: true
    joints: [R0, R1, R2]
    allowed_execution_duration_scaling: 1.2
    allowed_goal_duration_margin: 0.5

  - Name: gripper_controller
    action_ns: "follow_joint_trajectory"
    type: FollowJointTrajectory
    default: true
    joints: [GL, GR]
    allowed_execution_duration_scaling: 1.2
    allowed_goal_duration_margin: 0.5
```

Figure 6.12: Manipulator Joint_Trajectory_Controller

Moreover, it is also necessary to define the joint names of the two controllers in a .yaml file shown in Figure 6.13.

```
controller_joint_names: [R0, R1, R2, GR, GL]
```

Figure 6.13: Controller Joint Names

Moveit! setup assistant creates an empty launch file with name "robot_name"_moveit_controller_manager.launch.xml, this file should call the controllers defined for the manipulator and also should contain the parameters to run the Moveit! controller manager as shown in Figure 6.14.

```
<launch>
<rosparam file="$(find my_gripper_arm_moveit_config)/config/controllers.yaml"/>
<param name="use_controller_manager" value="false"/>
<param name="trajectory_execution/execution_duration_monitoring" value="false"/>
<arg name="moveit_controller_manager"
default="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
<param name="moveit_controller_manager" value="$(arg
moveit_controller_manager)"/>
</launch>
```

Figure 6.14: Robot Moveit! Controller Manager

And finally, a launch file that will set up the entire system for controlling the robot has to be created. This file should launch the move_group that is in charge of communicating with the real simulator Gazebo, moveit_rviz, joint_names and the joint state publisher that reads the state of every joint periodically. The file is shown in Figure 6.15.

```
<launch>
<include file="$(find my_gripper_arm_moveit_config)/launch/move_group.launch">
<arg name="publish_monitored_planning_scene" value="true" />
</include>
<include file="$(find my_gripper_arm_moveit_config)/launch/moveit_rviz.launch">
<arg name="config" value="true"/>
</include>
<rosparam command="load" file="$(find
my_gripper_arm_moveit_config)/config/joint_names.yaml"/>
<include file="$(find
my_gripper_arm_moveit_config)/launch/planning_context.launch">
<arg name="load_robot_description" value="true" />
</include>
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
<param name="/use_gui" value="false"/>
<rosparam param="/source_list">[/joint_states]</rosparam>
</node>
</launch>
```

Figure 6.15: Planning Execution System Launch File

6.5 Motion Planning with Moveit! Rviz and Gazebo Simulator

After Moveit! Rviz is connected with the real simulator Gazebo, a very simple test using the planning tab provided by Moveit! Rviz has to be done in order to make sure that the robot arm is working in visualization and in simulation as well.
This test is simply done by selecting start state and goal state in Moveit! Rviz and then plan and execute the selected states. The state could be one of the predefined poses (home or start) in the robot poses tab in Moveit! setup assistant. In this simple test we are going to set home pose as a start state and start pose as a goal state. Figure 6.16 shows the movement of the robotic arm in Moveit! Rviz and in Gazebo simulator as well.

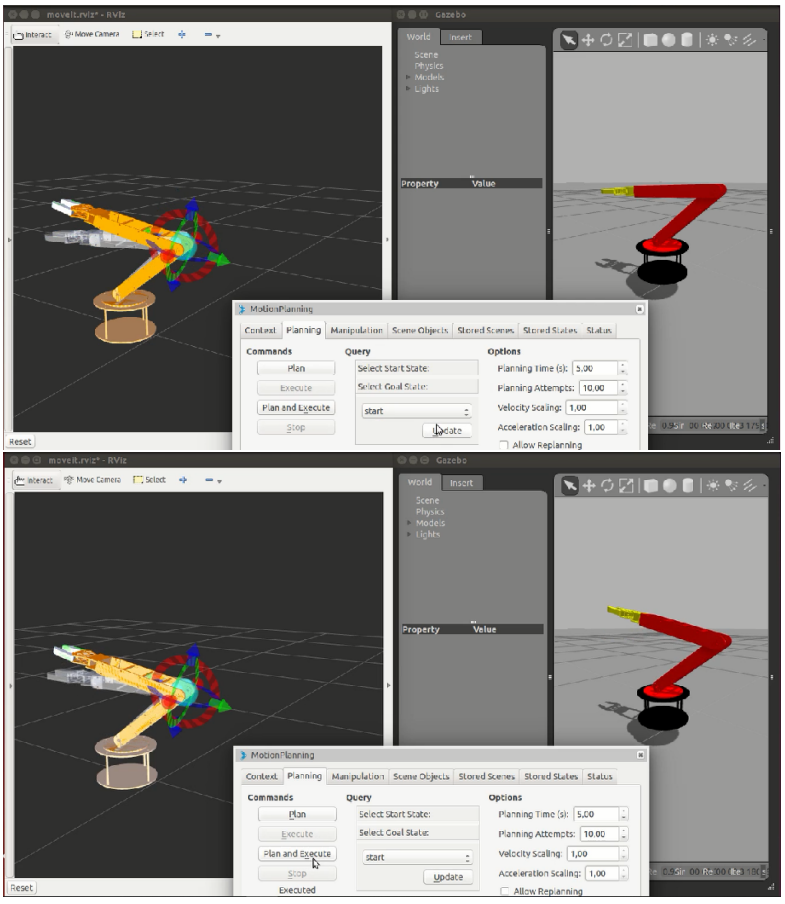


Figure 6.16: Simple Motion Planning With Moveit! Rviz and Gazebo

6.5.1 Motion Planning Programmatically Using C++ and Python Codes

Planning a Random Target Using Simple C++ Code

Before the movement of the robotic arm randomly using C++ code, it is important to talk about the Move Group Interface that should be defined in any C++ code in order to do specific operations. The Move Group is the user interface in Moveit! that provides the user with a wide range of functionalities for most operations. For example, setting joint values and pose target, getting states of the robot, moving, creating random plans, adding objects into the environment, etc...
However, the Move Group requires defining the group in charge of doing the motion planning. This group should be defined as a chain in the Moveit! setup assistant, which in our case is the group "arm" defined as three successive joints connected by links.
In the C++ code Figure 6.17, a random target created by the Move Group is set to the robot arm.

```
#include <ros/ros.h>
#include <moveit/move_group_interface/move_group.h>
int main(int argc, char **argv)
{
 ros::init(argc, argv,  "move_robot_randomly");
 // start a background "spinner", so our node can process ROS messages
 //  - this lets us know when the move is completed
 ros::AsyncSpinner spinner(1);
 spinner.start();
 moveit::planning_interface::MoveGroup group("arm");
 group.setRandomTarget();
 group.move();
}
```

Figure 6.17: Planning a Random Pose with C++ Code

Figure 6.18 shows the response of the robotic arm in Moveit! Rviz and Gazebo into the Random Planning Code.

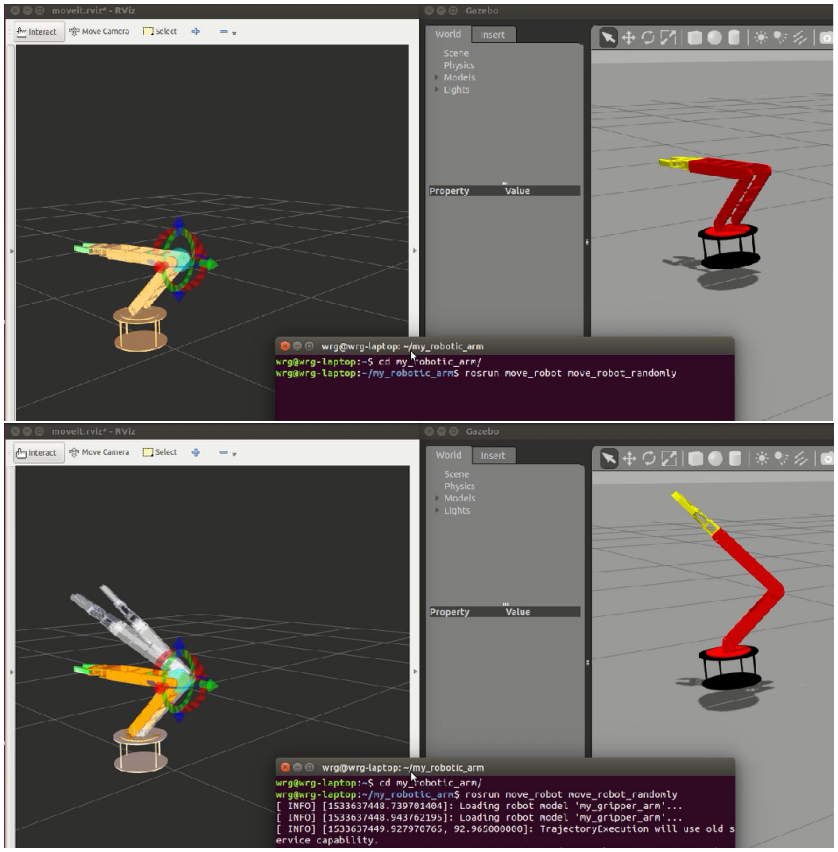


Figure 6.18: Random Planning Response in Moveit! Rviz and Gazebo

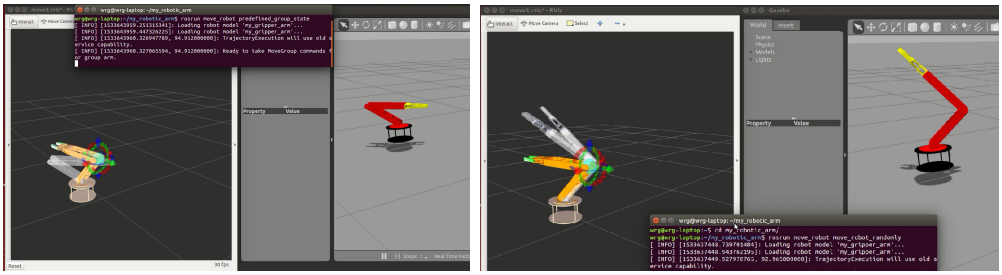
Planning a Predefined Group States Using C++ Code

As mentioned before, the Moveit! setup assistant provides a predefined group states for the robot. These states are crucial when working with the real robot hardware. For safety, we can position the robot in a home position every time we operate it. In this code we ask the robot to move into its home position and then take the start position to be prepared for operation.
In order to do so with code, it is important to read the robot state and then update it according to the pre-defined group as shown in Figure 6.19.

```
#include <ros/ros.h>
#include <moveit/move_group_interface/move_group.h>
int main(int argc, char **argv)
{
 ros::init(argc, argv,  "move_robot_randomly");
 // start a background "spinner", so our node can process ROS messages
 //  - this lets us know when the move is completed
 ros::AsyncSpinner spinner(1);
 spinner.start();
 moveit::planning_interface::MoveGroup group("arm");
 robot_state::RobotState current_state  = *group.getCurrentState();
 current_state.setToDefaultValues(current_state.getJointModelGroup("arm"),
"home");
 group.setJointValueTarget(current_state);
 group.move();
 sleep(2.0);
 robot_state::RobotState new_current_state  = *group.getCurrentState();
new_current_state.setToDefaultValues(new_current_state.getJointModelGroup("arm"
), "start");
 group.setJointValueTarget(new_current_state);
 group.move();
}
```

Figure 6.19: Planning a predefined group states for robotic arm

Figures 6.20a and 6.20b show the response of robotic arm to the predefined group states.



(a) Home State **(b)** Start State

Figure 6.20: Robot States

Planning Joint Position Using Python Code

One more interesting planning is setting joint values (Forward Kinematic) to move the robot arm into a specific positions. In this code, we set up a complete sequence for the robot joints in order to perform a process for grasping. The process of grasping consists of four main robot positions (pre-grasp, gripper open, post-grasp and gripper close). In pre-grasp position, the end-effector of the robot moves toward the object to be grasped but keeps a distance from it, then the gripper opens. After the grippers opens, the robot's end-effector moves so close to the object to be grasped (post-grasp position) and then the gripper closes.
Figure 6.21 shows a part of the python code, where we have to define a groups for the arm and for the gripper, then read the joints positions of the robot's current state and update them according to the desired sequence of joints positions.

```
/ !#usr/bin/env python
import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg

 #Setup node
print "\nStarting setup"
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('move_group_python', anonymous=True)

 #setup move_group
robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
display_trajectory_publisher =
rospy.Publisher('/move_group/display_planned_path ,'
      moveit_msgs.msg.DisplayTrajectory(

 #Setup arm group
arm_group = moveit_commander.MoveGroupCommander("arm")
arm_group.set_goal_tolerance(0.02)
arm_group.set_planning_time(5)
arm_group.set_num_planning_attempts(10)
end_effector_link =  arm_group.get_end_effector_link()

 #Setup end effector group
gripper_group = moveit_commander.MoveGroupCommander("gripper")
#gripper_group.set_end_effector_link("gripper")
gripper_group.set_goal_tolerance(0.02)
gripper_group.set_planning_time(1)
gripper_group.set_num_planning_attempts(5)

 #get joint values
first_joint_values = arm_group.get_current_joint_values()
print "\n First arm joint values:", first_joint_values
second_joint_values = gripper_group.get_current_joint_values()
print "\n First gripper joint values:", second_joint_values

ARM_HOME = arm_group.get_current_joint_values()
print "\nARM_HOME Joint Values:", ARM_HOME
ARM_HOME[0] = 0
ARM_HOME[1] = 0
ARM_HOME[2] = 0.2
arm_group.set_joint_value_target(ARM_HOME)
plan1 = arm_group.plan()
rospy.sleep(2)
arm_group.go(wait=True)
rospy.sleep(2)
#moveit_commander.roscpp_shutdown()

ARM_START = arm_group.get_current_joint_values()
print "\nARM_START Joint Values:", ARM_START
ARM_START[0] = 0
ARM_START[1] = 0.5
ARM_START[2] = -0.5
arm_group.set_joint_value_target(ARM_START)
```

Figure 6.21: Planning a Joint Positions of Robotic Arm

The response of the robotic arm is shown in Figure 6.22.

Figure 6.22: Complete Grasping Action Process

6.6 Motion Planning Failure in Moveit!

A common problem due to inverse kinematics occurs when working with low degrees of freedom robotic arm. When planning a goal position to the

robot's end-effector, the inverse kinematic solver is unable to plan any motion to reach the target goal position.

6.6.1 Fail Planning A Goal Position

In Figure 6.23 a python code has been created in order to move the robot end-effector into a desired position, this code simply includes the positions (x, y and z) and quaternion orientations (x, y, z and w) for the end-effector.

```
/ !#usr/bin/env python
import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg

 #Setup node
print "\nStarting setup"
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('move_group_python_interface_tutorial', anonymous=True)

 #setup move_group
robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
display_trajectory_publisher =
rospy.Publisher('/move_group/display_planned_path ,'
      moveit_msgs.msg.DisplayTrajectory(


 #setup arm group
arm = moveit_commander.MoveGroupCommander("arm")
arm.set_goal_tolerance(0.02)
arm.set_planning_time(5)
arm.set_num_planning_attempts(10)

end_effector_link = arm.get_end_effector_link
print "end effector link", end_effector_link
#arm.set_Planner_id("RRTConnectkConfigDefault")
print "current pose", arm.get_current_pose()
target = geometry_msgs.msg.Pose()
 #set position
target.position.x = -0.05
target.position.y = 0.14
target.position.z = 0.17

 #set orientation
target.orientation.x = 0.87
target.orientation.y = 0.09
target.orientation.z = 0.05
target.orientation.w = 0.47

arm.set_pose_target(target)
rospy.sleep(2)
arm.go(wait=True)
rospy.sleep(2)
moveit_commander.os._exit(0)
moveit_commander.roscpp_shutdown()
```

Figure 6.23: Planning a goal position

However, we know that the specified goal position is reachable by the arm, and this is done by moving the robot arm to a random position and then getting its current pose (see the pose in Figure 6.24), then copying the current pose into the python code, moving the arm back into a random target and finally running the python code.

```
current pose header:
  seq: 0
  stamp:
    secs: 599
    nsecs: 481000000
  frame_id: /world
pose:
  position:
    x: -0.235226341516
    y: -0.124144293715
    z: 0.550780678725
  orientation:
    x: -0.786983971533
    y: -0.608573896227
    z: 0.0620659178784
    w: 0.0802612186484
```

Figure 6.24: End-effector pose

The message we get from Moveit! when running the code is that the Inverse kinematic is unable to solve the planning problem, and then the code is unable to receive joint sequence values from Moveit! in order to move the robot to the desired pose. The messages are shown in Figure 6.25.

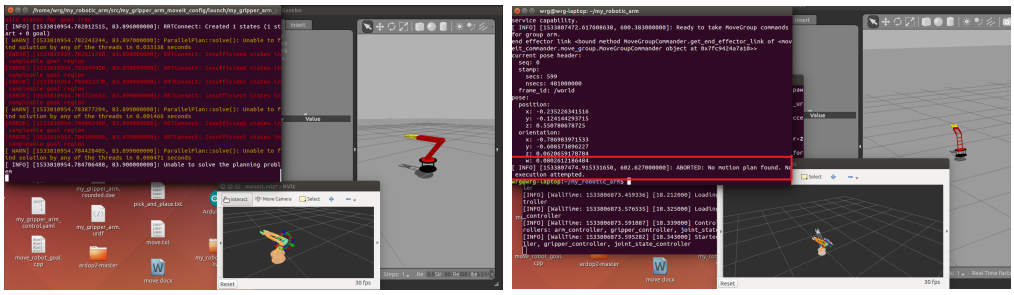


Figure 6.25: Planning a goal Position error messages

The inverse kinematics used in this test is *KDLKinematicsPlugin* supported by Moveit! setup assistant. This inverse kinematic is good for 6 and 7 DoF robotic arms where it is easy to find as many solutions as possible in order to move the end-effector into a desired position. But using this Inverse kinematic with a 3 DoF robotic arm creates a problem in finding solutions for planning a goal position to the end-effector.
For that reason, another inverse kinematic solution called IKFast has been used to see if this could solve the problem. According to OpenRave Wikipedia the IKfast inverse kinematics is defined as "IKfast is a powerful inverse kinematic solver that analytically solve the kinematic equations of any complex kinematics chain".
Using this solver with Moveit! requires using Openrave motion planning software that generates a C++ code to be used later. However, Moveit! IKFast tool generates IKFast kinematic plugin to be replaced in the kinematic.yaml file generated by Moveit! setup assistant under the config directory.
Unfortunately, installing this inverse kinematic doesn't solve the problem in simulation, and the same message was given by Moveit!.

6.6.2 Fail Planning A Pick_Place Action

Planning with Moveit! also provides the action of picking and placing objects, this planning also requires defining a specific position for the object to be grasped in the simulated world. However, this process contains different movements of the robotic arm in order to achieve the grasping action. The first movement is to get the robot arm ready to start working (set into start position), then moving into the pre-grasp position, post-grasp position and finally perform the grasping of the object inserted in the simulated world and move it into another place. Figure 6.26 shows the python code for grasping action.

```
!# /usr/bin/env python
import sys
import rospy
from moveit_commander import RobotCommander, MoveGroupCommander
from moveit_commander import PlanningSceneInterface, roscpp_initialize,
roscpp_shutdown
from geometry_msgs.msg import PoseStamped
from moveit_msgs.msg import Grasp, GripperTranslation, PlaceLocation
from trajectory_msgs.msg import JointTrajectoryPoint

if __name__=='__main__':

  roscpp_initialize(sys.argv)
  rospy.init_node('moveit_py_demo', anonymous=True)

  scene = PlanningSceneInterface()
  robot = RobotCommander()
  arm = MoveGroupCommander("arm")
  gripper = MoveGroupCommander("gripper")
  rospy.sleep(2)

  scene.remove_attached_object("gripper", "part")
 #clean the scene
  scene.remove_world_object("table")
  scene.remove_world_object("part")

 #publish a demo scene
p = PoseStamped()
p.header.frame_id = robot.get_planning_frame()
print "frame.id:", robot.get_planning_frame()
 #add a table
p.pose.position.x = 0.7
p.pose.position.y = -0.2
p.pose.position.z = 0.3
scene.add_box("table", p, (0.5, 1.5, 0.3))

 #add an object to be grasped
p.pose.position.x = 0.5
p.pose.position.y = -0.209705615117
p.pose.position.z = 0.493073164482
scene.add_box("part", p, (0.07, 0.01, 0.2))
rospy.sleep(2)

arm.set_named_target("home")
arm.go(wait=True)
rospy.sleep(2)

arm.set_named_target("start")
arm.go(wait=True)
rospy.sleep(2)

gripper.set_named_target("open")
gripper.go(wait=True)
rospy.sleep(2)

grasps[] =
g = Grasp()
g.id = "test"
```

```
print "\n Current pose:", arm.get_current_pose()
rospy.sleep(2)
grasp_pose = PoseStamped()
grasp_pose.header.frame_id = "base_footprint"
grasp_pose.pose.position.x = p.pose.position.x - 0.2
grasp_pose.pose.position.y = p.pose.position.y
grasp_pose.pose.position.z = p.pose.position.z
grasp_pose.pose.orientation.x = 0.0
grasp_pose.pose.orientation.y = 0.0
grasp_pose.pose.orientation.z = 0.0
grasp_pose.pose.orientation.w = 1.0
arm.set_pose_target(grasp_pose)
arm.go   ()
rospy.sleep(2)
g.grasp_pose = grasp_pose

 #define the pre_grasp pose approach
g.pre_grasp_approach.direction.header.frame_id = "base_footprint"
g.pre_grasp_approach.direction.vector.x = 1.0
g.pre_grasp_approach.direction.vector.y = 0.0
g.pre_grasp_approach.direction.vector.z = 0.0
g.pre_grasp_approach.min_distance = 0.05
g.pre_grasp_approach.desired_distance = 0.1

g.pre_grasp_posture.header.frame_id = "gripper"
g.pre_grasp_posture.joint_names = ["GL", "GR"]

pos = JointTrajectoryPoint()
pos.positions.append(0.0)
g.pre_grasp_posture.points.append(pos)

#set the grasp posture
g.grasp_posture.header.frame_id = "gripper"
g.grasp_posture.joint_names = ["GL", "GR"]

pos = JointTrajectoryPoint()
pos.positions.append(0.2)
pos.effort.append(0.0)
g.grasp_posture.points.append(pos)

 #set the post_grasp retreat
g.post_grasp_retreat.direction.header.frame_id = "base_footprint"
g.post_grasp_retreat.direction.vector.x = 0.0
g.post_grasp_retreat.direction.vector.y = 0.0
g.post_grasp_retreat.direction.vector.z = 1.0
g.post_grasp_retreat.desired_distance = 0.25
g.post_grasp_retreat.min_distance = 0.01

g.allowed_touch_objects = ["part"]
g.max_contact_force = 0

 #append the grasp to the list of grasps
grasps.append(g)
rospy.sleep(2)
 #pick the object
robot.arm.pick("part", grasps)
rospy.spin()
roscpp_shutdown()
```

Figure 6.26: Pick_Place Python Code

However, when running the code, we can see two objects are added (table and box) to the scene as shown in Figure 6.27.

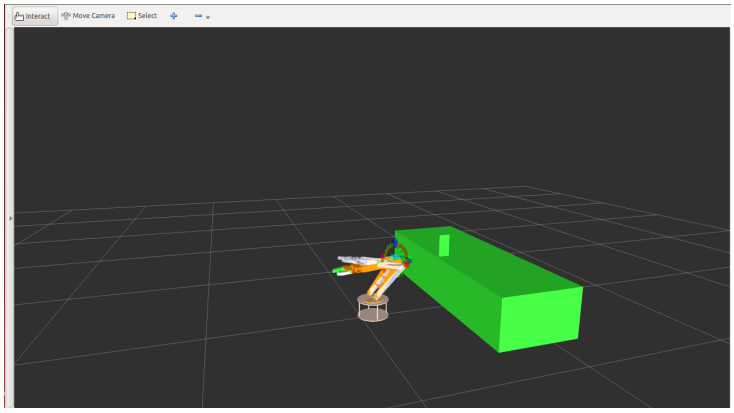


Figure 6.27: Adding Objects to the Simulated World

Up until the algorithm reaches the section of setting the grasp pose target specified according to the pose of the inserted box, then the Inverse Kinematic is unable to set any plan in order to move the robot's end-effector into the pre-defined position and the given error message is shown in Figure 6.28.

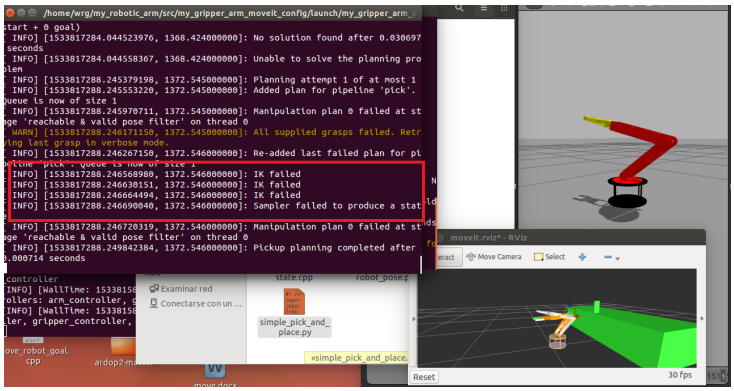


Figure 6.28: Adding Objects to the Simulated World

This is obvious because the robot was able to do the first two pre-defined movements (home and start) as shown in Figure 6.29.

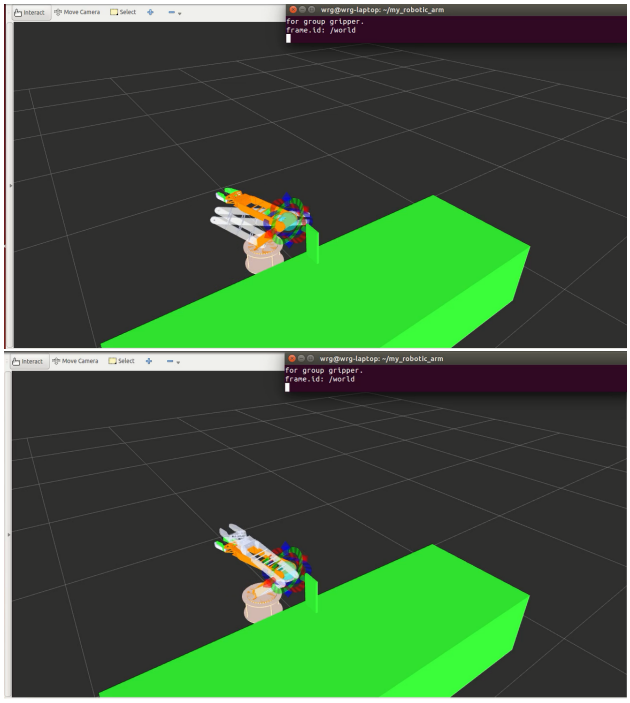


Figure 6.29: Home and Start movement with objects in the scene

6.6.3 Conclusion

The simulation of low degrees of freedom robotic arms in Moveit! have some restrictions due to the motion planning of the arm's end-effector. These restrictions are related to the Inverse Kinematic that in most of the cases is unable to find solutions for the end-effector to move from current position into a desired one. As lots of open sources for 6 and 7 DoF robotic arms are available to directly work with in ROS, the 6 DoF robotic arms don't have any problems when dealing with the motion planning for their end-effector. Consequently, when the number of degrees of freedom of a robotic arm increases, it is simpler to plan motions and it is easier to find inverse kinematics for the end-effector.
However, as IKFast solution has been used and did not work, one more possible solution is to replace the C++ code generated by the IKfast by a C++ code written by the user that defines the inverse kinematics equations of the robotic arm. This work can also be done in future.

Chapter 7

Building Up The Real Robot

7.1 Hardware Description

7.1.1 Motors

The robotic arm has 4 motors, 3 for the links and 1 for the gripper.

Link's Motor

The calculations for every link joint motor are the same, here we will consider the calculation of motor joint 2 because it is affected by the maximum torque and the joint motor for link_1 has an axis of rotation parallel to the gravity, so the torque has no effect on it. Here the calculation considers the worst way for the joint (The joint is shown as a red point in the diagram of Figure 7.1) to lift an object, this case is when the robot arm is fully extended as shown in the diagram of Figure 7.1[15].

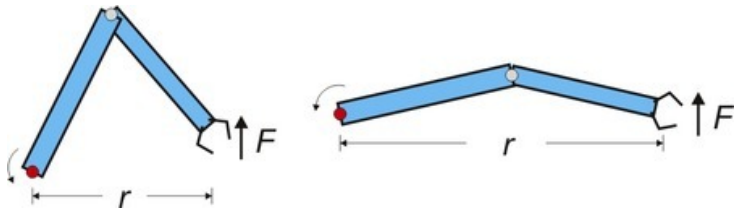


Figure 7.1: Robot arm lifting geometry

The distance r is 80 cm and the object to be lift assumed to have a maximum of 350 grams, multiplying 80cm by 0.35 kg we get 28 kg.cm torque required by the motor in order to lift the object.

Figure 7.2: 6V Low-Power (LP) 25D mm Gearmotors

According to the calculations done before, the gearmotor chosen consists of a low-power, 6V brushed DC motor combined with 499:1 metal gearbox, and it has an integrated 48 CPR quadrature encoder on the motor's shaft, which provides 23,945.84 counts per revolution of the gearbox output shaft. The general specifications of the motor are in the Table 7.1.

Gear ratio	499:1
No-load speed @6V	11 rpm
No-load current @6V	250 mA
stall current @6V	2400 mA
stall torque @6V	30 kg.cm
Encoders	Yes
Price	34.95 USD

Table 7.1: 25D mm Metal Gearmotors General Specifications

Gripper Motor

Figure 7.3: 12V High-Power Carbon Brush (HPCB) Micro Metal Gearmotors

This gearmotor is a high-power(hp), 12 V brushed DC motor with long-life carbon brushes and 986.41:1 metal gearbox. It has a cross-section of 10x12 mm, and the D-shaped output shaft is 9 mm long and 3 mm in diameter. This motor can also work at 6 V, the general specifications of the motor is shown in Table 7.2.

Gear ratio	986.41:1
No-load speed @12V	35 rpm
No-load current @12V	0.06 A
stall current @12V	0.75 A
stall torque @12V	30 kg.cm
No-load speed @6V	18 rpm
No-load current @6V	0.03 A
stall current @6V	0.38 A
stall torque @6V	5 kg.cm
Encoders	NO
Price	24.95 USD

Table 7.2: Micro Metal Gearmotors General Specifications

7.1.2 Driver

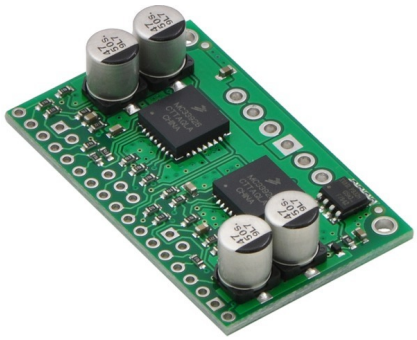


Figure 7.4: Brushed DC Motor Driver

This dual brushed DC motor driver, based on Freescale's MC33926 full H-bridge, has a wide operating range of 5-28 V and can deliver almost 3 A continuously (5A peak) to each of its motor channels. The MC39926 works with 3-5 V logic levels, supports ultrasonic (up to 20 kHz) PWM, and features current feedback, under-voltage protection, over-current protection, and over-temperature protection. Two drivers are needed for our system because each driver has two motor channels and the system has 4 motors. The general specifications of the driver are shown Table 7.3.

Motor Driver	MC39926
Motor Channels	2
Minimum operating voltage	5 V
Maximum operating voltage	28 V
Continuous output current per channel	2.5 A
current sense	0.525 V/A
Maximum PWM frequency	20 kHz
Minimum Logic Voltage	2.5 V
Maximum Logic Voltage	5.5 V

Table 7.3: Brushed DC Motor Driver General Specifications

7.1.3 Micro-controller

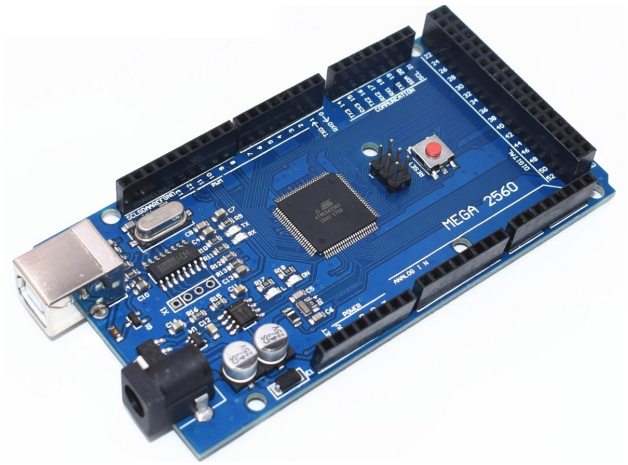


Figure 7.5: Arduino Mega 2560

The micro-controller used in this project is Arduino Mega 2560, one of the reasons behind using this micro-controller is that it provides 6 interrupt pins (2, 3, 18, 19, 20 and 21) which used for reading the encoder of the motors. These interrupt pins will halt what the processor is currently doing and execute the other functions, this is known as interrupt service routine or ISR. The interrupts can be set to trigger on rising, falling and change signal edges. Moreover, it is a low-cost micro-controller and it offers a large number of digital I/Os pins (54 of which 14 provide PWM output) and it uses a flash memory of 256 Kbyte which is good space for compiling our codes into it.

7.1.4 Hardware Prices

Hardware	Quantity	Price(USD)
25D mm Metal Gear motor	3	34.95 (Total = 104.85)
Micro Metal Gear motor	1	24.95
Brushed DC Motor Driver	1	29.95
Arduino Mega 2560	1	34.00

Table 7.4: Robot Arm Hardware Prices

7.2 Building The Real Robot

As mentioned in the design part, the robot consists of 5 parts, base link that handles the entire arm, Link_1 rotates in the xy-plane, Link_2 and Link_3 rotate in xz-plane and yz-plane and finally, the gripper that made up of 2 fingers.
The gripper is 3D printed with plastic types ABS, this type of plastic has high impact strength and it is recommended for end-use parts. However, Base_link and Link_1 are made up of PVC plastic with circular shapes and 4 mm thickness and finally, Link_2 and Link_3 are made up of flat shaped Aluminium bars with length 30 cm and 4 mm thickness connected with hollow Aluminium tubes with 6 mm outer diameter and 5 mm inner diameter. The entire arm (Link_1, Link_2, Link_3 and the Gripper) weighs 1.3 kg including the motors of each link.
However, with these dimensions the robot workspace is a cylinder with 80 cm diameter and 90 cm height as shown in Figure 7.6.

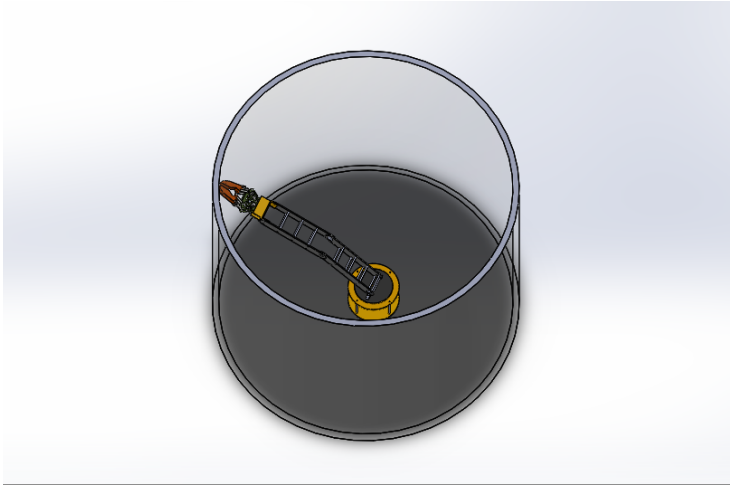


Figure 7.6: Robot 3D Workspace

1. **GRIPPER:** Let's start with the real gripper and how it works in reality. The gripper motor connects an 8 mm extruder screw through a joint, a nut is connected with the middle link in the gripper which is able to move up and down according to the rotation of the gripper's motor. Figure 7.7 shows the real connections.

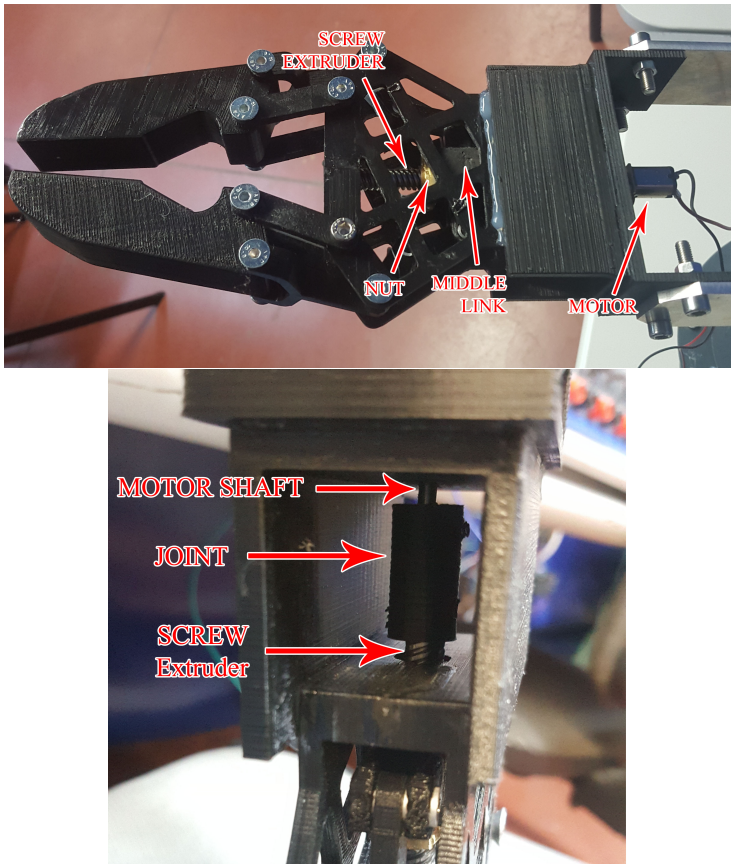
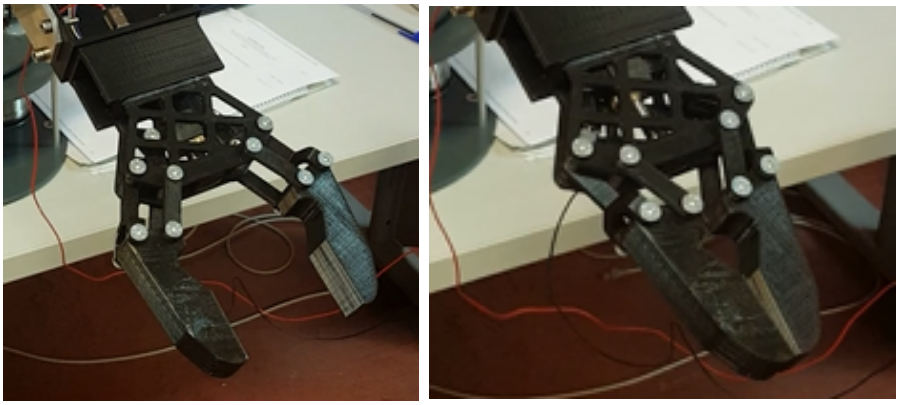


Figure 7.7: Real Gripper Connections

However, rotating the gripper motor clock-wise opens the gripper, and close if the motor is rotating counter clock-wise. Figure 7.8a and 7.8b show the gripper open and gripper close.



(a) Real Gripper Open **(b)** Real Gripper Close

Figure 7.8: Real Gripper

2. **Base_link:** Figure 7.9 shows the base link with the first motor that holds the Link_1 of the robot arm.

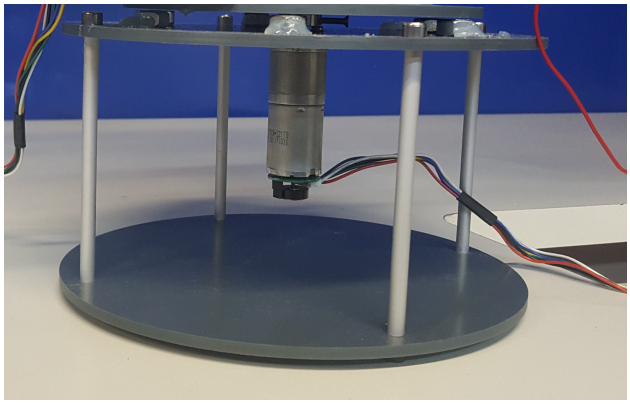


Figure 7.9: Real Robot Base Link

3. **Link_1:** Link_1 of the real robot holds the motor of Link_2 from one side and a support to Link_2 on the other side. However, we put several nuts on base_link in order to balance the first link when it is rotating.

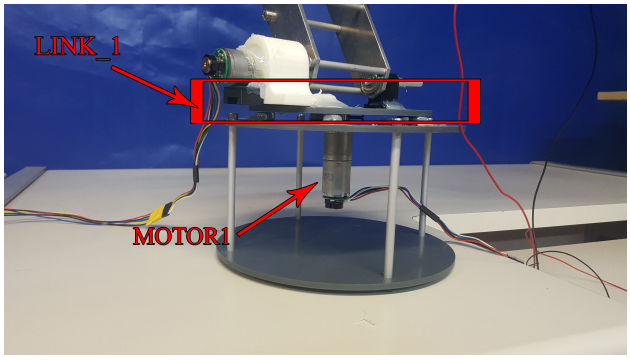


Figure 7.10: Real Robot First Link

4. **Link_2:** Link_2 is 2 flat shaped Aluminium bars connected by hollow Aluminium tubes, also it holds the motor that controls the movement of Link_3 as shown in Figure 7.11.

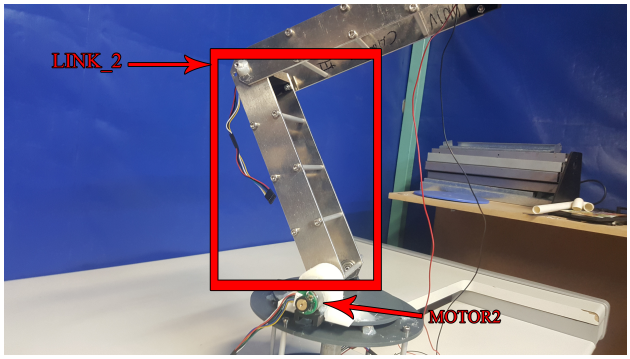


Figure 7.11: Real Robot Second Link

5. **Link_3:** Link_3 has been built in exactly the same way as Link_2. However, its last end connects the gripper which is the last part of the robot as shown in Figure 7.12.

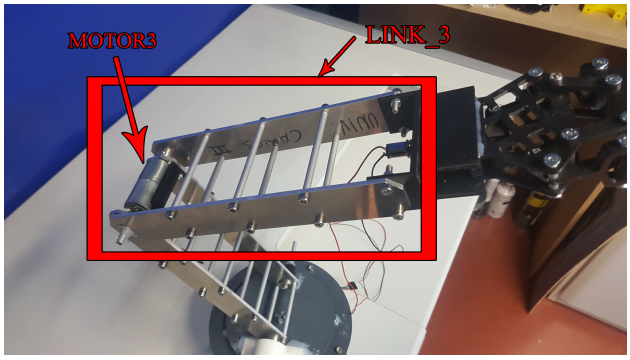


Figure 7.12: Real Robot Third Link

6. **Complete Robot:** Figure 7.13 shows the complete real robot with its all parts connected together.

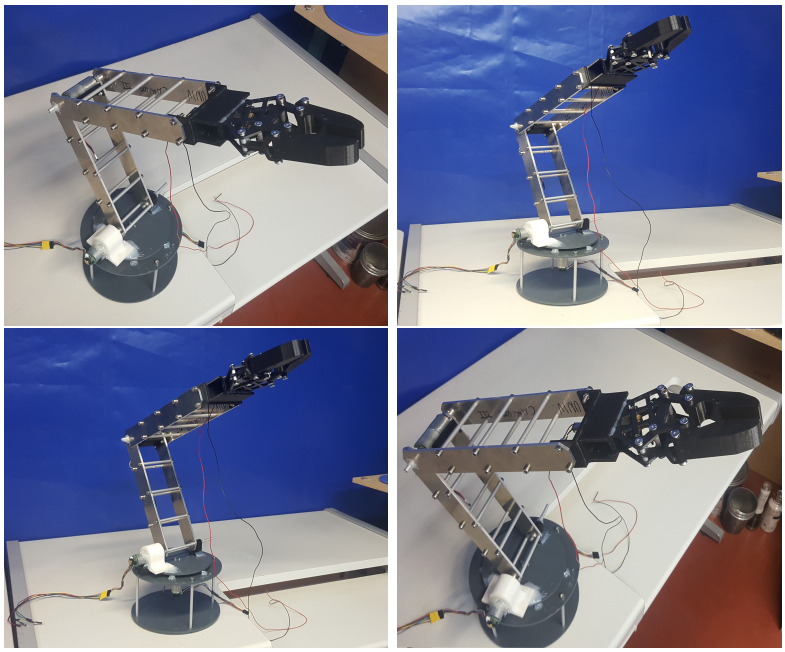


Figure 7.13: Complete Real Robot

7.3 Hardware Connections

The motor/encoder has 6 wires as shown in the previous Figure 7.2. Table 7.5 describes the function of each wire.

COLOUR	FUNCTION
Red	motor power (connects to one motor terminal)
Black	motor power (connects to the other motor terminal)
Green	encoder GND
Blue	encoder Vcc (3.5 - 20 V)
Yellow	encoder A output
White	encoder B output

Table 7.5: Motor Wires Function

Encoder A output and encoder B output are square waves from 0V to Vcc with approximately 90° out of phase. However, the order of transition (signal A leads signal B or signal B leads signal A) indicates the direction of the motor, while the frequency of transition indicates the speed of the motor. The following oscilloscope capture in Figure 7.14 shows the A and B (Yellow and White) encoder outputs of the motor.

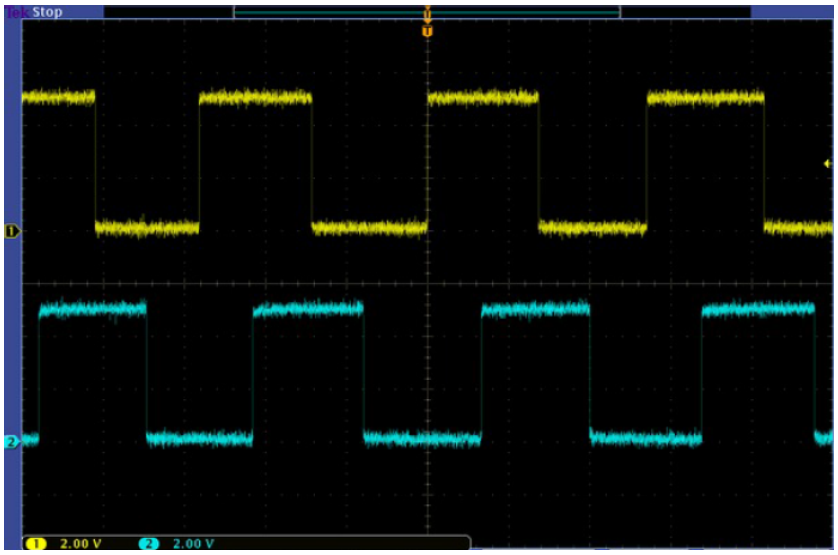


Figure 7.14: Encoder A and B outputs for 25D LP Gearmotor

Moreover, the motor is controlled using the MC39926 dual driver.

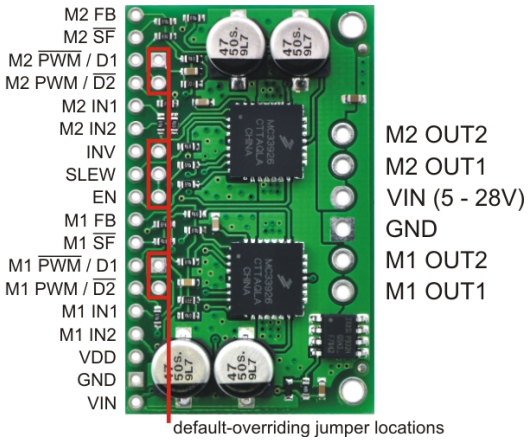


Figure 7.15: MC39926 dual motor driver

For typical applications, two signals of the driver IN1 and IN2 that control the motor output pins OUT1 and OUT2 should be connected to control the motor direction, and one of the PWM disable lines D1 or $\bar{D}2$ to control the speed of the motor, but we have to note that if one of the two disable lines is connected to control the speed, the second one should be set into an appropriate mode in order to enable the function of the connected one, and finally the enable signal EN is set to LOW in default, so this signal must be set to HIGH in order to run the board.
Table 7.6 shows the description of the driver control signals.

PIN	Default State	Description
OUT2	HIGH	The motor output pin controlled by IN2
OUT1	HIGH	The motor output pin controlled by IN1
IN2	HIGH	The logic input control of OUT2
IN1	HIGH	The logic input control of OUT1
PWM / $\bar{D}2$	LOW	Inverted disable input, when $\bar{D}2$ is LOW, OUT1 and OUT2 are set to high impedance
$\bar{PWM}$ / D1	HIGH	Disable input, when D1 is high, OUT1 and OUT2 are set to high impedance
EN	LOW	Enable input, when EN is LOW, the both motor driver ICs are in a low-current sleep mode.

Table 7.6: MC39926 Dual Driver pins specifications

As we already know that each driver is able to control 2 motors. The diagram of Figure 7.16 shows the total connections of the 4 motors and 2 driver with the Arduino Mega 2560 micro-controller.

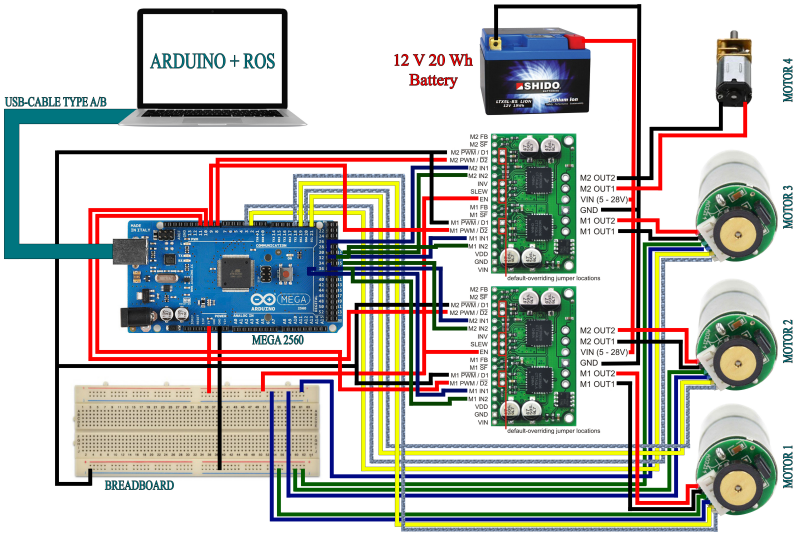


Figure 7.16: Robot Hardware Connections

However, as shown in Figure 7.16, Arduino and ROS are going to be used in order to control the robot arm. This interface allows us to control the directions and the speed of the motors by publishing data from ROS publisher into Arduino program. We will talk about this in details in the next section.
In addition, Figure 7.17 shows the real connections of the robot hardware in the same way shown in the diagram of Figure 7.16.

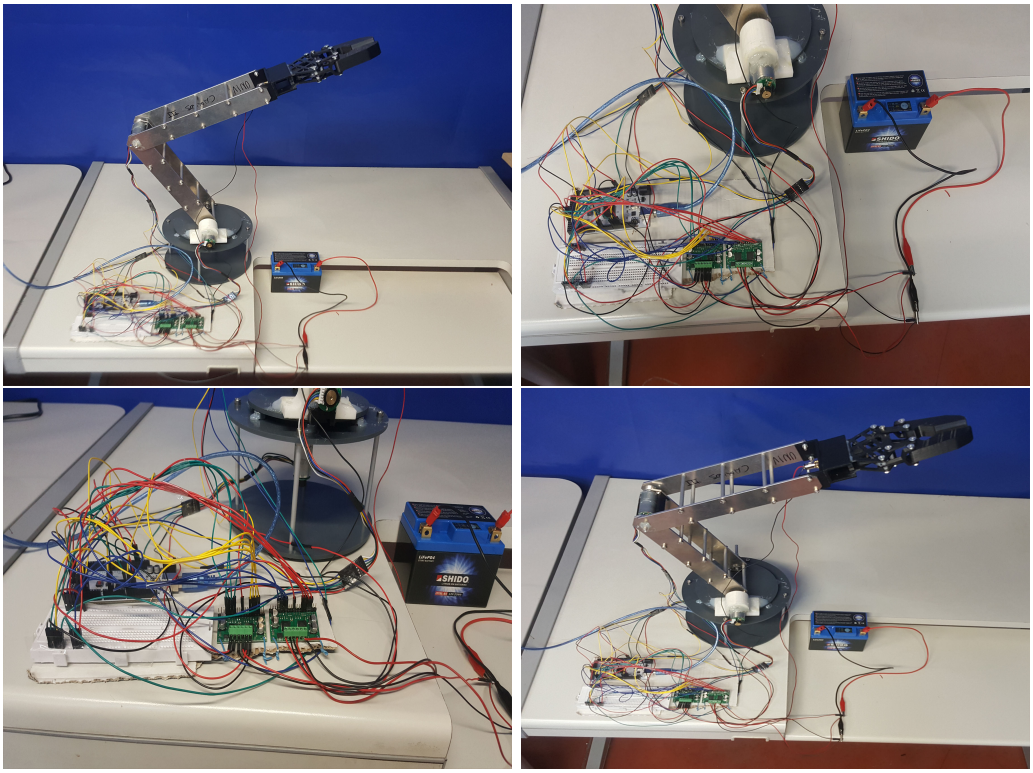


Figure 7.17: Robot Hardware Real Connections

7.4 Open Loop Control of Robotic Arm with Arduino + ROS

7.4.1 General Overview

In this section we are not going to discover what Arduino is. Thanks to this low-cost hardware, you can easily perform large projects. However, as any micro-controller, Arduino has a limited capacity, especially in its cheaper versions. So, it is obvious that for large projects it is necessary to connect Arduino with more powerful CPU, this CPU will be in charge of the most complex tasks, which leaves the Arduino as an element of communication with actuators and sensors.
Consequently, ROS will be used as the central CPU and it will be connected to Arduino in order to send and obtain data from the actuators and sensors. This communication can be made through the rosserial-arduino package. Through this package, we can use ROS with Arduino IDE. ROS_serial provides a communication protocol in ROS that allows working with Arduino. It allows Arduino to create a node, which can publish or subscribe ROS messages, TF transformations and obtain ROS system parameters.
In addition, to be able to use ROS with Arduino, it is necessary to copy the ros_lib library, which allows the Arduino IDE programs to interact with ROS. The ros_lib library is installed with the rosserial_arduino package mentioned before, so the procedure is simple, just locate the rosserial_package and copy the ros_lib into the Arduino build environment to enable arduino programs to interact with ROS.

7.4.2 Arduino IDE Code

The following diagram shown in Figure 7.18 shows the open loop control of robotic arm DC motors.

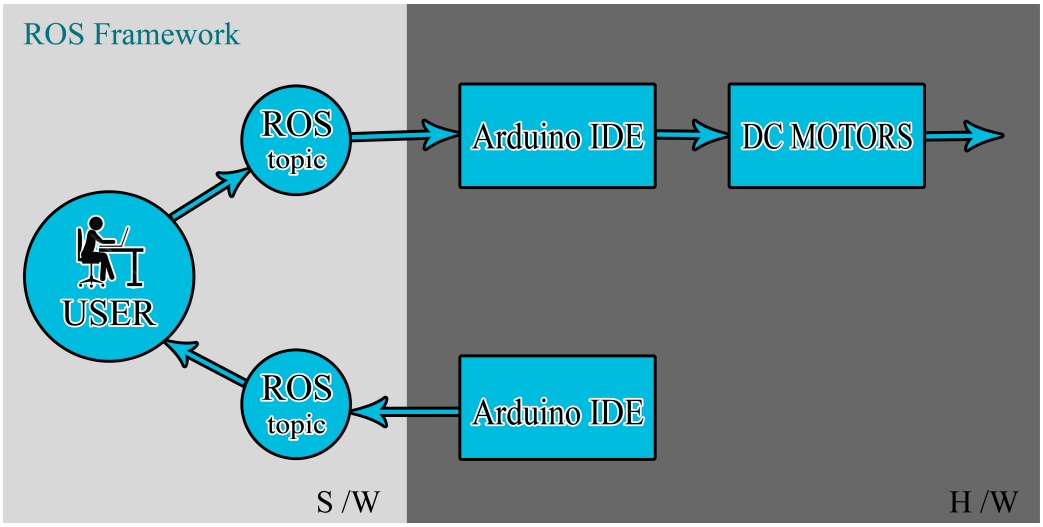


Figure 7.18: Open Loop DC motor Control

Firstly, in the Arduino code we are going to define the dependencies and declare the pins that control the motors directions and speed as shown in Figure 7.19.

```
#include <ros.h>
#include <std_msgs/Int16.h>

ros::NodeHandle nh;
//int i=0;
#define PWMM1 8
#define PWMM2 9
#define PWMM3 10
#define PWMM4 11
#define dir1M1 30
#define dir2M1 31
#define dir1M2 32
#define dir2M2 33
#define dir1M3 34
#define dir2M3 35
#define dir1M4 36
#define dir2M4 37
```

Figure 7.19: Ardiono IDE dependencies and pins

Then in a void setup() loop, the pins mode are going to be defined that runs once when the arduino code is compiled. However, subscription to the topics are defined and nh node is initialized. See Figure 7.20.

```
void setup() {
 // put your setup code here, to run once:
 pinMode(PWMM1, OUTPUT);
 pinMode(PWMM2, OUTPUT);
 pinMode(PWMM3, OUTPUT);
 pinMode(dir1M1, OUTPUT);
 pinMode(dir2M1, OUTPUT);
 pinMode(dir1M2, OUTPUT);
 pinMode(dir2M2, OUTPUT);
 pinMode(dir1M3, OUTPUT);
 pinMode(dir2M3, OUTPUT);
 pinMode(dir1M4, OUTPUT);
 pinMode(dir2M4, OUTPUT);
 analogWrite(PWMM1, 0);
 analogWrite(PWMM2, 0);
 analogWrite(PWMM3, 0);
 analogWrite(PWMM4, 0);
 digitalWrite(dir1M1, HIGH);
 digitalWrite(dir2M1, HIGH);
 digitalWrite(dir1M2, HIGH);
 digitalWrite(dir2M2, HIGH);
 digitalWrite(dir1M3, HIGH);
 digitalWrite(dir2M3, HIGH);
 digitalWrite(dir1M4, HIGH);
 digitalWrite(dir2M4, HIGH);

 nh.initNode();
 nh.subscribe(subcmdM1);
 nh.subscribe(subcmdM2);
 nh.subscribe(subcmdM3);
 nh.subscribe(subcmdM4);

}
```

Figure 7.20: Arduino pins mode and subscribers

Moreover, four callback functions are going to be declared in order to activate the motors. These functions control the motor's directions according to the sign of the data published by ROS. However, the data published are used to regulate the motor's speed which works in a range 0-255 PWM and then the data is automatically converted into 0-5 Volts for the motor speed. Figure 7.21 shows the first motor callback function, the same callback functions are used for the rest of the motors.

```
void cmdMotor1CB( const std_msgs::Int16& msgM1)
{
 if(msgM1.data >= 0) {
  analogWrite(PWMM1, msgM1.data);
  digitalWrite(dir1M1, HIGH);
  digitalWrite(dir2M1, LOW);
  delay(50);
  analogWrite(PWMM1, 0);
  digitalWrite(dir1M1, HIGH);
  digitalWrite(dir2M1, HIGH);
 } else {
  analogWrite(PWMM1, -msgM1.data);
  digitalWrite(dir1M1, LOW);
  digitalWrite(dir2M1, HIGH);
  delay(50);
  analogWrite(PWMM1, 0);
  digitalWrite(dir1M1, HIGH);
  digitalWrite(dir2M1, HIGH);
 }
}
```

Figure 7.21: First Motor Callback function

Finally, four subscribers are going to be declared, the topics are called *cmd_MotorX*, where X could be *1, 2, 3 or 4* (See Figure 7.22) depending on the motor that we are going to publish data on. Note that *std_msgs::Int16* topic type is used.

```
ros::Subscriber<std_msgs::Int16> subcmdM1("cmd_Motor1", cmdMotor1CB);
ros::Subscriber<std_msgs::Int16> subcmdM2("cmd_Motor2", cmdMotor2CB);
ros::Subscriber<std_msgs::Int16> subcmdM3("cmd_Motor3", cmdMotor3CB);
ros::Subscriber<std_msgs::Int16> subcmdM4("cmd_Motor4", cmdMotor4CB);
```

Figure 7.22: ROS Subscribers

7.4.3 Experimental Results

Now, we can upload the code to the board and run the Arduino node, we can see in Figure 7.23 that ROS is subscribing on the cmd_MotorX topic mentioned in Figure 7.22.

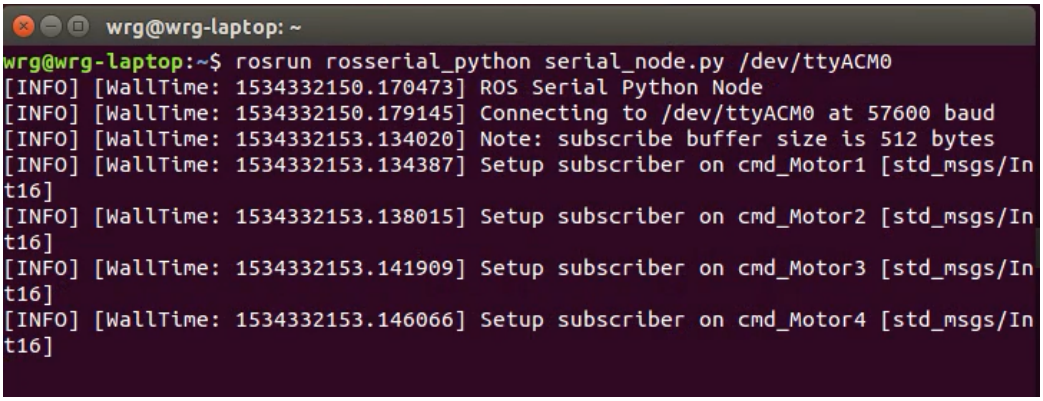


Figure 7.23: Setup Subscribers on ROS Topics

And then, we manually publish data on ROS topics as shown in the terminal of Figure 7.24.

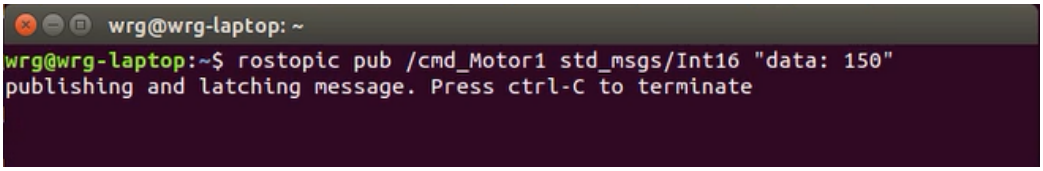


Figure 7.24: rostopic data publisher

This code has been used in order to grasp an object manually, and the results are shown in Figure 7.25.

Figure 7.25: Experimental Result Open Loop Control

Chapter 8

Conclusion and Future Work

8.1 Conclusion

As a conclusion, the major purposes of this project have been accomplished. The response of the PID controller showed somewhat shorter rise time to reach the set point and smaller overshoot compared to the PI controller. However, the Fuzzy Logic Control showed a slow response for the control signal (CO) to reach the set point, large overshoot and the robot's link when the control output reaches the set point was affected by vibrations. The Adaptive control showed good results, it has been tested by changing the inertia of the system, the control parameters were successfully calibrated every time the inertia changed and the system showed fast response to the changing of the control parameters.
Furthermore, the 3 DoF robotic arm has been integrated and simulated on ROS and Gazebo, the robot responses to the ROS commands using GUI (Graphical User Interface) and ROS topics were successfully achieved. However, some motion planning (random position, pre-defined position and forward kinematic) in Moveit! were performed successfully by the robotic arm. On the contrary, the failure of some motion planning (Planning a goal position or pick place actions) in Moveit! indicates the restrictions of the low DoF robotic arms compared to 6 or 7 DoF. Moreover, with the open loop control to the real robot, we were able to grasp an object and move it to another place.

8.2 Future Work

8.2.1 Hardware Improvement

In the section of Hardware description, we have done the calculations of the motor's torque in order to lift an object. The calculations have only considered the torque needed by the motor in order to lift an object with almost 300g.

However, this simple result doesn't take into account the weight of the arm and the torque needed by the motor to only lift the arm itself. So, the new calculations have to factor the torque that the robot arm imposes on our joint. However, the torque from each piece of the arm has to be added. Fortunately, we don't have to measure and weigh the location of each part. instead, we can weigh the whole assembly and find the point at which we can balance it; this point is the centre of mass of the arm, then we can consider the arm as a weightless rod with a mass corresponding to the whole arm's weight at the distance from the joint to the centre of mass as shown in Figure 8.1.

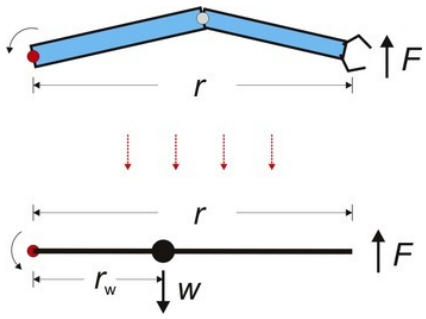


Figure 8.1: Robotic Arm Center of Mass

The entire assembly of our project weighs 1.3 kg, and the centre of mass is 40cm from the joint. The arm would then put 52 kg-cm of torque on our motor. This is the net torque required by the motor in order to only lift the arm.

8.2.2 Adding Force-Sensing Resistor

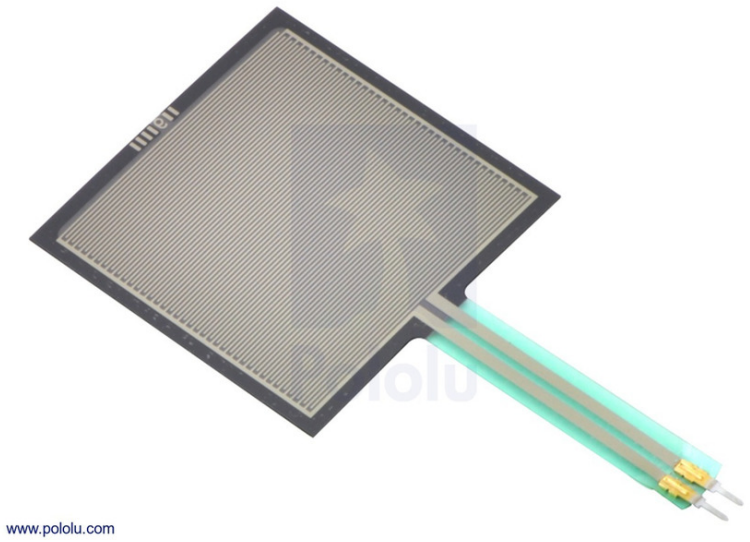


Figure 8.2: Force-Sensing Resistor

This Force-Sensing Resistor (FSR) works as a variable resistor. The resistance of the active area decreases when a force is applied to it. The sensitive area may sense a force varying from few grams into kilograms, which make it a very useful sensor for our project.
The sensor is going to be mounted on one of the gripper's fingers. When the gripper is going to grasp an object, the sensor senses the force applied by the object on the finger (See Figure 8.3); this force is going to be converted into a high signal sent to the Arduino board, which will stop the gripper's motor. This allows us to deal with objects of different width and materials.

Moreover, adding vision sensors (ultrasonic, infrared, cameras, image sensor, light or colour sensors) to the robotic arm allows us to guide the robot to work autonomously.

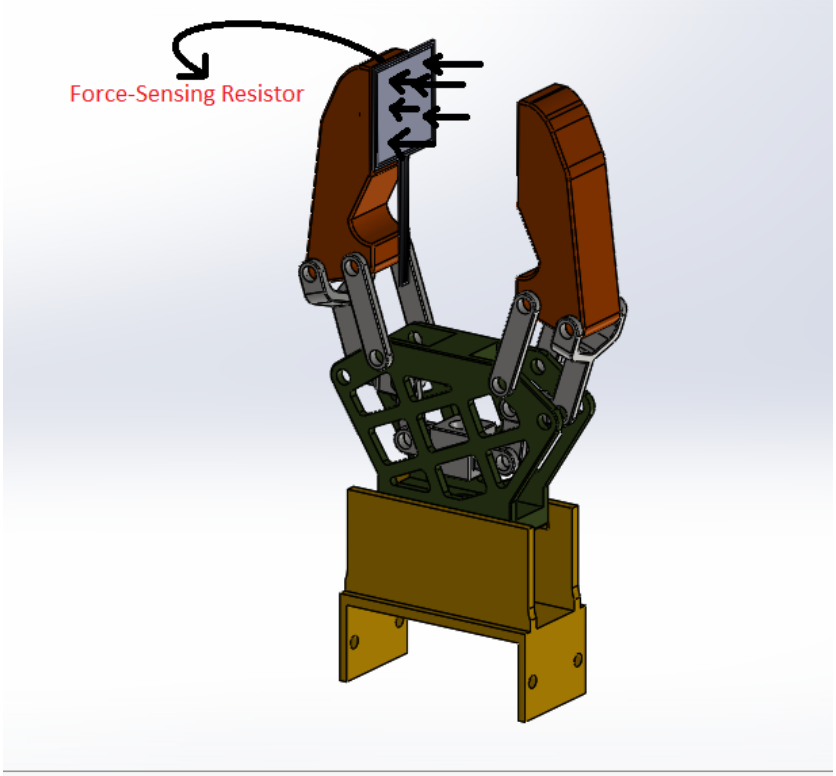


Figure 8.3: FSR sensor with the gripper

8.2.3 Control Work

In section 5.2, we show the robot kinematics model and Denavit-Hartenberg parameters. In the next step of robot control, we will use these parameters to obtain the inverse kinematics.
To do so, it is important to find the arm's position in order to place the gripper correctly. The position of the arm depends on the position angle of the motors (Direct Kinematics). However, Direct kinematics allows us to find the position angle of the motors in order to achieve the pre-known goal position (Inverse Kinematics). Moreover, Inverse kinematics equations can be used in C++ code with ROS and Arduino, the program can receive the inverse kinematics equations and set the required angle position to reach the goal point.

Bibliography

[1] Ramon Barber, Cristina Castejon, Antonio Flores, Higinio Rubio, A Hardware-Software Approach for Design and Control of Mechatronics Devices, *International Journal of Research in Engineering and Science (IJRES)*, http://www.ijres.org, Volume 4 Issue 10, October. 2016, PP. 68-80, ISSN (Online): 2320-9364, ISSN (Print): 2320-9356

[2] https://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f4-series.html?querycriteria=productId=SS1577

[3] http://www.mantech.co.za/datasheets/products/a000047.pdf

[4] https://www.mouser.com/ds/2/588/AS5045B_Datasheet-1101416.pdf

[5] https://en.wikipedia.org/wiki/MATLAB

[6] https://es.mathworks.com/help/simulink/

[7] https://www.faulhaber.com/OnlineUpdates/Moman5/EN_7000_00043_Rev5.pdf

[8] https://www.arduino.cc/en/Reference/SPI

[9] Technical University of Denmark, Department of Automation, bldg 326, DK-2800 Lyngby, DENMARK. Tech. report no 98-E 864 (design), September 30, 1999.

[10] Landau I.D, Lozano R, M'Saad M, Karimi A. Adaptive control Algorithms, Analysis and applications. 2011, XXII, 590 p. with Online file/Updates. Hardcover, ISBN: 978-0-85729-663-4.

[11] R. Barber, D. R. Rosa, S. Garrido. Adaptive control of a DC motor for educational purposes. *International Federation of Automatic Control(IFAC)*, August, 2013.

[12] https://es.mathworks.com/help/ident/ug/online-recursive-least-squares-estimation.html?s_tid=srchtitle

[13] Ladislav Jurišica, Roman Murár. Mobile robots and their subsystems. AT&P journal Plus1. 2008

[14] Anil Mahtani, Luis Sanchez, Enrique Fernández, Aaron Martinez. Effective Robotics Programming with ROS. Birmingham B3 2PB, UK. Packt Publishing Ltd, December 2016, ISBN 978-1-78646-365-4.

[15] https://www.pololu.com/blog/10/force-and-torque

[16] Dzmitry Tsetserukou, Naoki Kawakami, Sasumu Tachi. iSoRA: Humanoid Robot Arm for Intelligent Haptic Interaction with the Environment. University of Tokyo, February 2009. https://www.tandfonline.com/doi/pdf/10.1163/156855309X462619