POLITECNICO DI TORINO

Master's Degree course in Mechatronics Engineering

Master's Degree Thesis

Preliminary Implementation of a system layout for the control of an inflatable robotic arm prototype

Supervisors

Prof. Stefano Mauro

Matteo Gaidano

Pierpaolo Palmeri

Candidate

Francesco Contran

Academic Year 2022-2023

Abstract

The robotics for space applications is a sector of the space industry in which new technologies are being explored. In particular, Politecnico di Torino aims to launch in the space its 2-link robotic manipulator arm, with the characteristic of having the links inflatable. The so called POPUP robot is in a experimental phase in the laboratories of Politecnico di Torino, aiming to reach the space in the next years. Thanks to an inflatable structure, the occupied space and the weight to be launched are less than a typical rigid robot. However, the complexity in modelling and controlling the robotic arm increases. In this project there is the goal to establish a control method for the robot. In particular, three methods of control have been developed. The first one is the control in the joint space, the second one the control in the operational space, the third one the target reaching control. The main work was to realize and to optimize the algorithms of these three kind of control.

While the first two controls were already present as a draft, and therefore the goal was to validate and improve them, the third one is a new one. Fixing the control in the joint space was the first task of the project. Thanks to a telecontrol operated through a PS4 controller, the user could see in real time the progress of its codes, checking the progress in the laboratory on the field.

After having fixed the control in the joint space, an analysis of the whole structure for the control in the operational space was done. The problem was identified and corrected, however some optimizations are still required, in terms of speed of comunication.

The structure of the target reaching control, instead, was built from scratch, and this control still needs some adjustments.

After some test done to check the motors, and an Optitrack final test to validate the controls, the control in the joint space was validated. The control in the operational space needs still some optimization, but is better than before and has improved. For the control of the target

reaching optimizations are still needed, and the next steo will be to put a visual servoing control.

Contents

List of Figures

Chapter 1

Introduction

The space exploration began in the 60s, with USSR and USA competing for the space exploration. The collaboration between the nations interested in the space exploration then increased. In fact, in 1998, the International Space Station, orbiting around planet Earth, has been launched, as one of the biggest scientific international project of all time.

The robots became crucial for space application, replacing humans in dangerous situation. Humans can't operate in lack of oxygen, and is safer to manipulate, repair and construct on the space with robots. Therefore, aerospace robotic application increased through the years, and not only for the exploration of the planets and satellites in the solar system. [1]
The Canadarm2 is an example of this. It can move all along the ISS, performing all the operations needed in that field, thanks to a 17.6 m length when fully extended

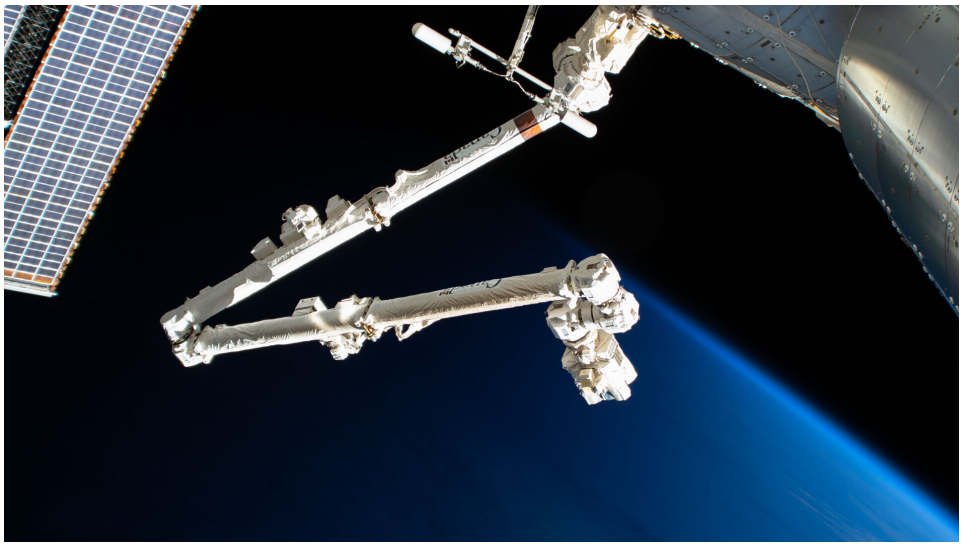

Figure 1.1. Canadarm2 robotic arm mounted on International Space Station

This robotic arm helps to maintenance of the spatial station, is able to move stuff, astronaut and equipment perform catches in the space by grappling visiting vehicles and sending them to the International Space Station (ISS) [2].

However, developing a robot for space application is a difficult challenge, starting from the design and arriving to the control. The environment in the space is totally different than the one on planet Earth. One of the problem is the rigid structure of the manipulator, which is heavy and therefore forces the manipulator itself to have an high payload. Soft robotics can face this challenge. In fact, robots made of soft materials are lighter and some applications are already present on planet Earth. With their low mass, they open the chance to launch them in the space thanks to a spacecraft and the possibility to transport them with less cost, since they are lighter. The inflatable links of soft robots are of course easier to launch in the space, since their mass is lower than the one of a traditional robot. With the possibility of inflating the robot on the field of application, the space occupied at the moment of the launch is a fraction of the case with a rigid structure. [3]. Having these advantages leads to some disadvantages. In particular, in

performance, given the fact that inflatable links are present their stiffness and damping factor have to be taken into account. Positioning the end effector and controlling it in the 3 directions of the cartesian space is not properly accurate because of the oscillations of the inflatable link during the motions. [4]. Controlling the robot is therefore not easy
Usually, the control algorithms for rigid body robot manipulators are based on precise mathematical models. [5].
With these models, the relations between the robot joint angles and the end effector pose and orientation are obtained. With dynamics, the relation expands to the movement and velocity of the robot. A soft robot has not a rigid structure, therefore those algorithms cannot be applied in those points when deformation can occur, due to the stiffness and damping of the inflatable part.
Therefore, these control algorithms may not be as reliant on precise mathematical models, but instead use sensory feedback to adjust the robot's motion in real-time [6].

Two of the main methods for the control of the robot are in the joint space and the control in the operational space. [5]
The goal of this thesis project is to develop the control in the joint space and the control in the operational space for the POPUP Robot, and to test them in order to see if they pass the validation and they are reliable.
The POPUP robot prototype, developed in the DIMEAS laboratories of Politecnico di Torino, is a two-link soft robotic arm with inflatable links, three T-MOTOR AK-8080 electrical motors and rigid joints.

The first prototype presents inflatable links with cylindrical shape and made out PVC fixed to a 3D printed support which connect the link to the actuator. The links to be fixed through screws to the other joints, allowing the possibility to add elements, e.g. sensors, inside the links during development stage [7]. A pneumatic line is responsible to control the inflation and deflation stage. It allows the links to be inflated and deflated, providing the necessary pressure which is in the range of 10-60 kPa.

For future applications, a wrist will also be integrated, extending the degrees of freedom from 3 to 6. In figure 1.2, the POPUP robot from Politecnico di Torino is shown. It is made of a two inflatable links and three motors. Motor 1, on the base of the robot, is used to move the arm in the XY plane. With motor 2, the robot can move in the 3D space an go upwards, while motor 3 give the possibility to move the links and extend or retract the robot. The cartesian space of the robot is also shown in the figure 1.2

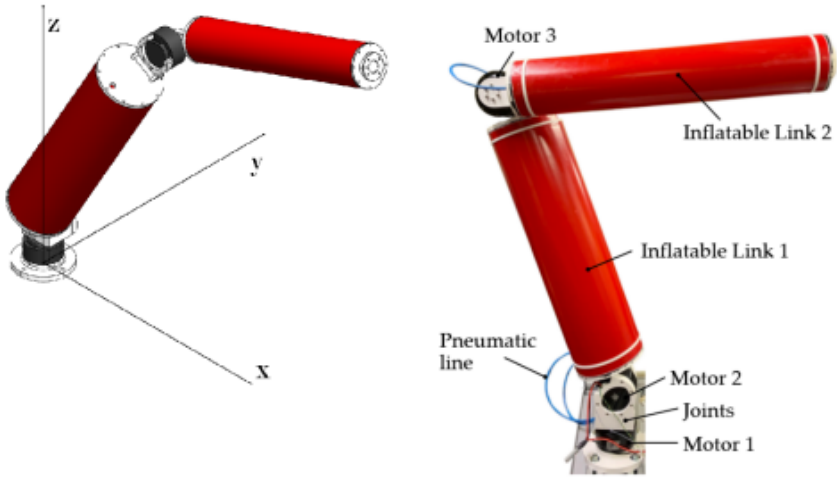

Figure 1.2. POPUP robot

The aim of the project is to implement and test a control for the robotic arm on the field. The robot will be controlled thanks to a PS4 controller. Before the beginning of this project, the communication system and the robot structure were already present. However, the control was present only in a draft way, in the sense that the control mode were defined, but not acceptable on the field. The aim of the project is to start from these drafts and to implement a wokrking control in the joint space, in the operational space and to try to implement a new function, the target reaching control.

Chapter 2

Methods

2.1 Robot telecontrol

In order to control the movement of the robot, the user uses a PS4 controller, where every single button gives a command to the robot. In particular, in the beginning phases of this project, in which the telecontrol was inherited, the robot had 2 modes to be controlled. Now the modes are 3.The first two modes were already drafted, although not implemented nor validated. The third mode of control, instead, was brand new. In figure 2.1, it's shown the old framework for the commands of the PS4 controller. It has to be noticed that this was a draft, a more in depth analysis and improvement of this framework was one of objectives of this project

Some of advantages of using a PS4 controller to move the motors of the robot are:

- direct controllability of the robot arms;
- the possibility to have a direct feedback on the robot arm movement;
- the chance to check in real time and to have a preliminary feedback on the tests without having to wait for the results at the end of the running program;
- the rapid and immediate understanding of how to move the manipulator.

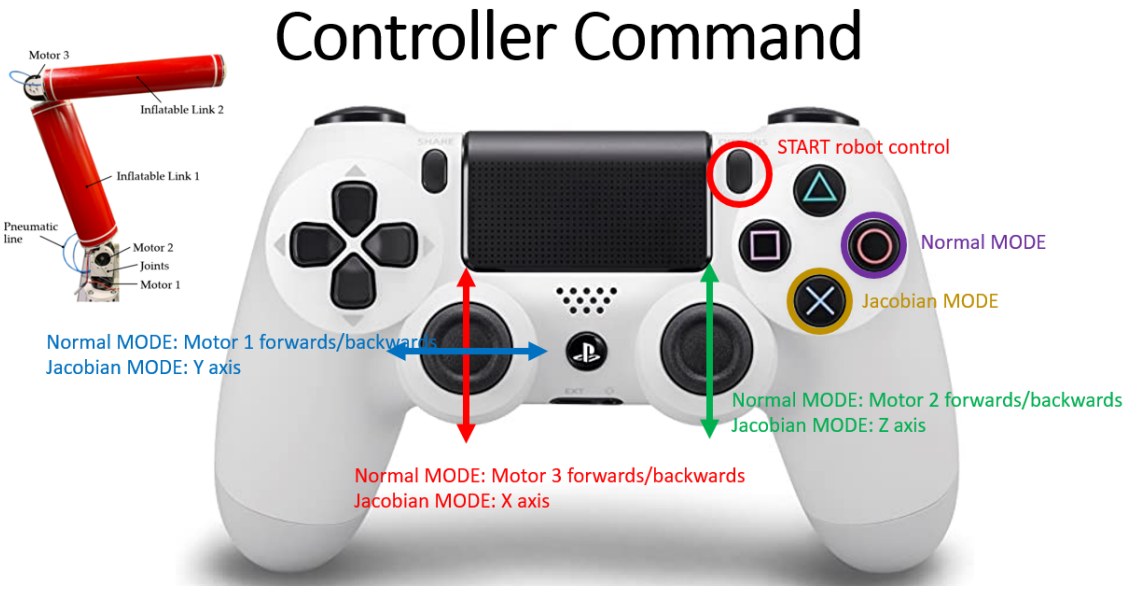

Figure 2.1. The Joypad and its commands

The user has 3 modes to control the inflatable robot:

- Control in the joint space;
- Control in the operational space via inverse Jacobian control;
- Target reaching mode;

In figure 2.1 is shown how every button of the controller can be used in order to control the movement of the robot, given the control mode. In the small image of the robotic arm, the three motors with each respective number are stated. Motor 1 is on the base, motor 2 allows the movement on the vetical plane, motor 3 is between link 1 and link 2.
A more in depth description of all the 3 control modes is stated below

2.1.1 Control in the joint space

The control in the joint space is the easiest control method to implement on a robot like the POPUP one. Basically the user, by pushing the sticks of the controller, assigns a velocity to each motor at the time, controlling the motor speed directly.

The user can move each motor clockwise, counterclockwise and he can even move the motors together, by simply using the correct combination of the buttons of the controller. In the figure 2.2 we can see a preliminary block scheme of the control in the joint space. The user controls the joypad, giving the command to the communication system which actuates the command sending to the motor the desired velocities. A more in depth look on how the communication system actually works will be explored in the next chapter

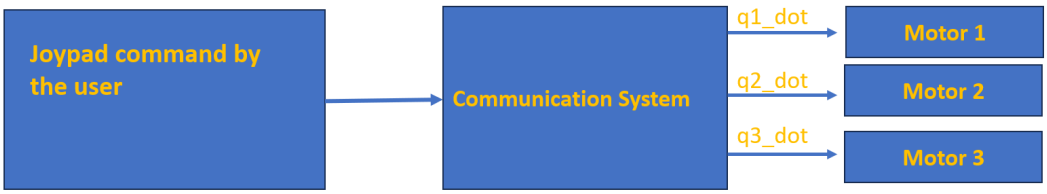

Figure 2.2. The control in the joint space

2.1.2 Control in the operational space

The control in the operational space is more complex than the control in the joint space. In fact, while in the first case the user was directly controlling the robot motors, here we have some mathematical and matricial computation which comes in place.

A reference block scheme for the jacobian inverse control is stated below. The idea for this control comes from the differential inverse kinematics. It has to be noted that this is a speed control, so the component of the position of the motors that may be present in the original version of the control are here not considered .

But first, it's necessary to explain what the Jacobian transformation matrix is. According to the Denavit Hartenberg convention, it's possible to build the transmission matrix which describes every possible translation and rotation around the joint and the links of which the robot is made, just starting from the geometry of the robot.

Thanks to the Denavit Hartenberg convention and to the transmission matrices, it's possible to build the Jacobian of the robot, which is the key to both forward and inverse kinematics. In forward kinematics, given the angle of the motors in the joint space, we can translate those angles in the position and orientation of the end effector.
$v_e = J(q)\dot{q}$, which means that the velocity of the ned effector can be obtained by multiplying the jacobian matrix with the velocity in the joints. This is the reason why this kind of control is called control in the operational space, because the goal is to control the position and velocity of the end effector, which is used exactly to do operations and manipulations. The inverse kinematics, starting from the position and orientation of the end effector, extracts the position angle of the motors. $\dot{q} = J^{-1} * v_e$ It's a powerful tool, but mathematically speaking has a cost, since it's necessary to invert a matrix. [5].

In the block scheme of figure 2.3 the difference between the desired and the effective positions passes through a function that computes the desired velocity. This desired velocity, along with the actual position measured from the motors, is the input of the jacobian inverse matrix.

In output there is the desired velocity in the motor space, and subtracting to it the actual motor velocity and applying a gain K, the control loop closes itself.

Now that the basic concepts are explained, it's possible to understand this kind of control. In this case, the user wants to move the end effector in the cartesian space, along x,y,z, and to move the end effector in the cartesian space with a specific velocity. So the input from the user are vector velocities, with a module, a direction and a sense.

The jacobian inverse function receives as an input the desired input from the user and the actual feedback position from the motors. From the motor position it's possible the calculate the inverse jacobian. Multiplying the desired velocity in the cartesian space by the Jacobian inverse matrices, these gives to motors the velocities to achieve the desired behaviour the user put as an input. The end effector will then move with the velocities and the directions given by the user, actuating its commands by sending the right speed to the three motors

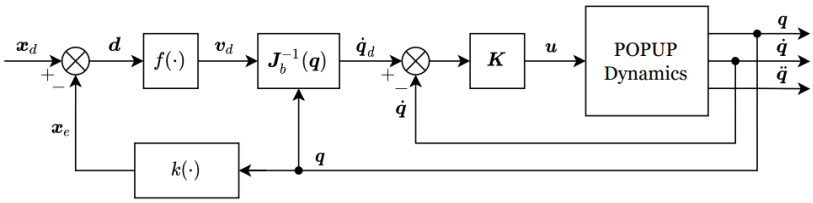

Figure 2.3. The reference block scheme for Jacobian inverse control

2.1.3 The target reaching control mode

The objective of the target reaching mode is to make the robot reach a desired target point in the 3D space. In particular, given the point to reach, the end effector has to move in order to touch it

Once the point is well established, what happens is that a direction of movement has to be given to the end effector. The setup is then pretty simple. The user press square on the joypad and the robot actually moves to reach the target, if the stick L3 points upwards.

To make this algorithm effective, it is necessary to compute the position of the end effector. This is done thanks to the direct kinematics of the robotic arm, computed using the Denavit-Hartenberg convention. [5]

Once the estimated feedback position of the end effector of the robot is computed, there a now two points in the space with their 3 coordinates. What happpens is that a simple subtraction between the two vectors gives the direction of movement to the end effector.
$directionofmovement = targetpoint - endeffector$ By multiplying the direction of movement with the desired velocity to reach the target, a vector velocity comes out of the equation. And with this vector velocity, and the position of the motors, it's sufficient to repeat the jacobian inverse control to give to the motors the velocity necessary to reach the target point in the space with the desired velocity.

A couple of things has to be noticed. For this algorithm to work, it's crucial to pick a point in the space with a textbook precision. Small variations in the order of cm or mm can generate a result different from what is asked. And also has to be considered the fact that a stop button is necessary in this case. From the computation, in fact, the robot will oscillate around the target position, once he is very close to it. Therefore, it is necessary to implement a sort of good range position to stop the robot, or simply to do that manually.

In the figure 2.4, a simple block scheme of what has been described above can be seen. The inputs are the robot motor positions and the target point to be reach. From the positions of the motors, thanks to the direct kinematics, the position of the end effector is derived. Once the end effector estimated position and the target point are established, subtracting the two vectors gives a direction of movement, along with a velocity. With these data passing through the jacobian inverse matrix, the velocity to be sent to the three motors is then computed and sent But before going into the details of the code and the test implemented to running those 3 possible control modes for the POPUP robot, it is necessary to describe the actual communication framework, which is a key part of the robot control

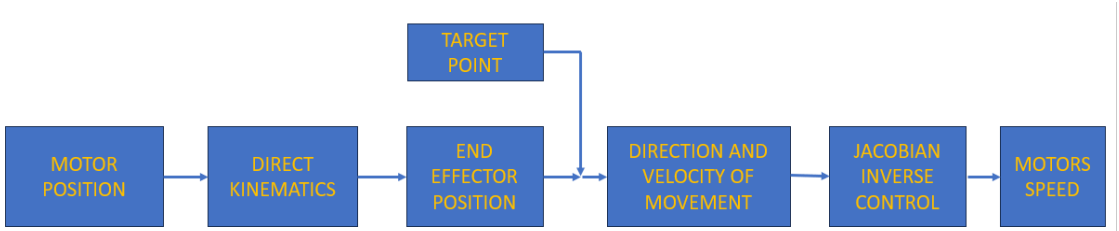

Figure 2.4. Target reaching mode block scheme

Chapter 3

The POPUP Robot communication system

The POPUP Robot has a communication system made by three principal actors

- **Laptop**:
  The laptop is used to communicate with the user, displaying the real-time commands sent to the motor. On the laptop there are the codes used to calculate all the equations needed to control the robot. The laptop has Ubuntu 20.04 installed on, necessary to utilize ROS Noetic. ROS is used with his nodes to control the POPUP robot. The graphical interface provide immediate data and feedback to the user

- **STM32H7 Board**:
  This is an electronic board developed by STM. This board is fundamental as a bridge between the PC and the actuator. Linked to the PC by a USB cable, the board receives the commands by the laptop, and it sends them to the VESC drivers used to directly give commands to the motor. The communication between the STM-32 and the Vescs hardware is done by a CAN protocol, while the messages between the board and the PC are coded under a USART protocol. From the board, three pair of wires are linked to the VESC drivers, each one for one motor. Therefore, a single VESC

actuates the control on motor 1, another one on motor 2, and a third one on motor 3.

- **VESC**:
  The VESC drivers are the bridge between STM-32 and the actual motors, converting the commands in rpm sent out by the PC and STM-32 directly in erpm, the unity of measure required by the motors, accounting for pole pair and gear reduction of the AK-8080 motors, used to move the links of the robot. The setup of the VESC hardware is done by a specific app developed to this purpose, called VESC tool.

In the figure 3.1, the main actors of the communication are shown, taking into account that the VESC hardware is directly mounted on the POPUP Robot.

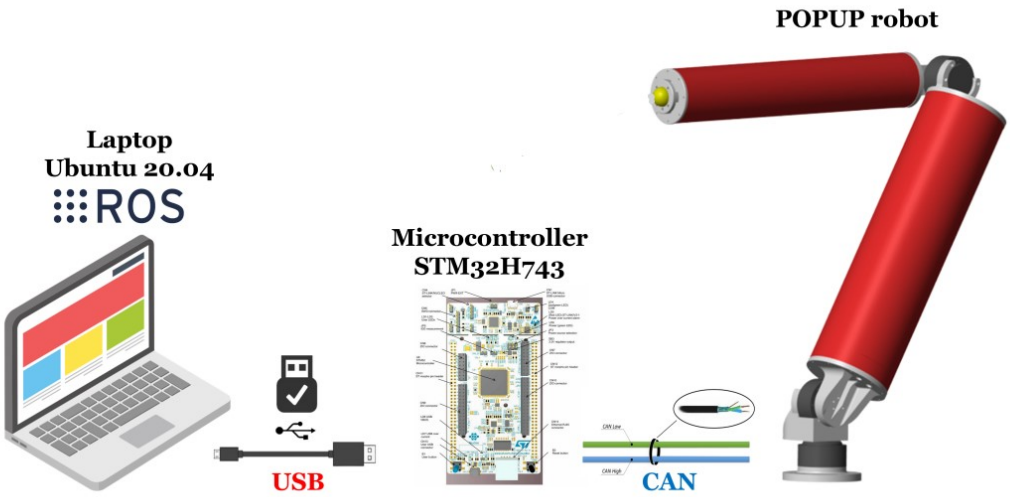

Figure 3.1. The POPUP Robot communication actors

Although the communication system was already present at the beginning of this project, it is necessary to give a deeper description of ROS and STM32 codes. The draft codes for moving the robot with joint

space control and operational space control were not accurate nor effective. Part of the problems were in the communication system, therefore is necessary to give more information about this topic

3.1 ROS

ROS stands for Robot Operating System, with a given structure used exaclty for the purpose to communicate with the robots. ROS is made by nodes, the actors of this communicaton. All the nodes refers to a main node called master, the main node between all. And ROS is also made of packages, the basic units to develop its applications, like the folders of a PC. A simple example of communication is stated in the figure 3.2 [8]

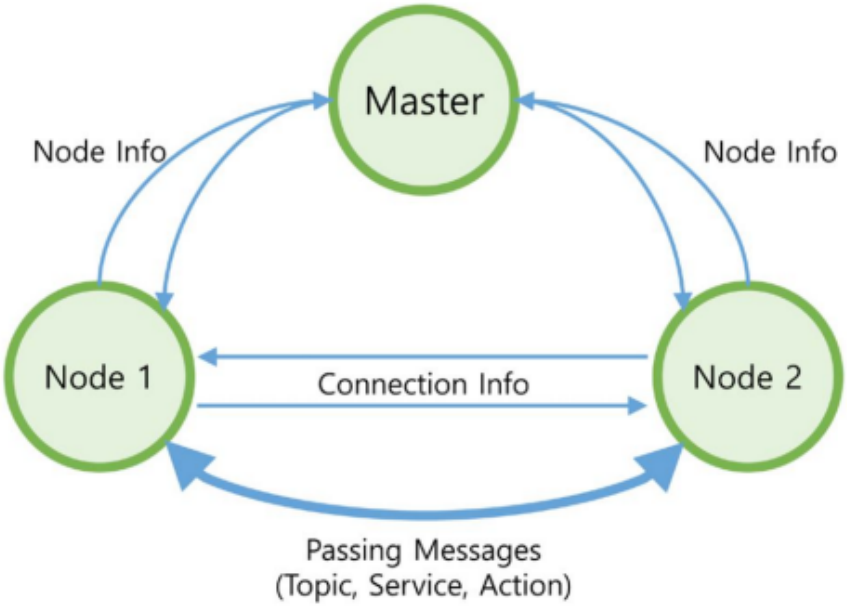

Figure 3.2. A simple message communication in ROS

The communication between nodes, that is the key to understand how the inflatable POPUP Robot sends message, is made by a sort of protocol. ROS is made by node, and each node can be publisher and subscribers. Of course a publisher has to publish a message, called

topic, while a subscriber has to subscribe to receive the message. A deeper description follows

- **Topic**
  A topic is an argument of conversation, basically. Therefore, when a publisher sends out a message, the object of the message is the topic. When the subscriber wants to read a message has to ask the name of the topic
- **Message**
  If the topic is like the title of a message, like the object of an e-mail, the message is the actual data sent out. It can be a boolean, an integer, a string.
- **Subscriber**
  Subscribe means receiving a message. Therefore a subscriber is a node with the puropose of receiving a message, corresponding to the topic. It receives the information given by the publisher on the desired topic to which it's subscribed
- **Publisher**
  Transmitts a message corresponding to the topic to all the subscriber subscribed to that specific topic node

This kind of communication is asynchronous, but the topic constantly transmitts the message. Therefore, when the data of the message are constant, and we need periodically transmitted data, this is the protocol to use. In this project, only publisher and subscriber were used to communicate.

3.1.1 Rosserial package

In ROS packages are very useful as folder and to be transferred from one system to the other. For the communication with the serial interface of the PC, the Rosserial package was necessary.

It has to be noticed that ROS communicates via publishers and subscribers. However, that is not the case for the USART communication

protocol. Given the fact that the messages from the PC to the board have to pass through a USB cable, a message translator is needed. The Rosserial package is used exactly to complete this task of translation of the messages. This pakcages were made by a server, the PC, and a client, the board, as it is synthethized in figure 3.3. The rosserial server was a python one, the client an STM32 one.

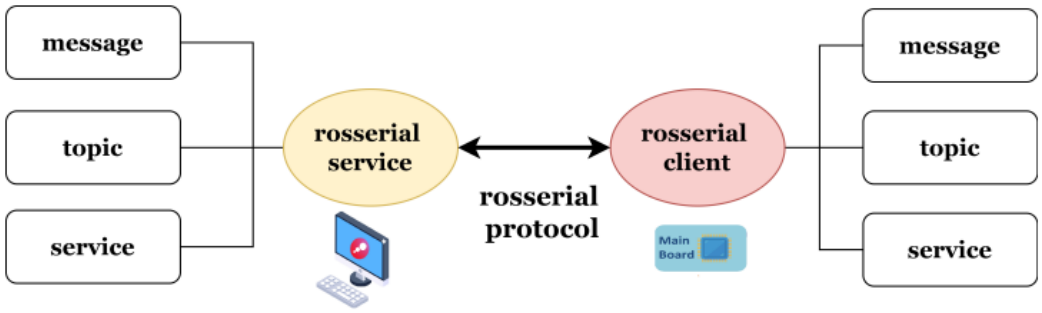

Figure 3.3. The rosserial protocol

3.2 STM-32

The STM-32 board is an electronic device made by a 32 bit microcontroller developed by STM

"The STM32 family of 32-bit microcontrollers based on the Arm Cortex®-M processor is designed to offer new degrees of freedom to MCU users. It offers products combining very high performance, real-time capabilities, digital signal processing, low-power / low-voltage operation, and connectivity, while maintaining full integration and ease of development.

The unparalleled range of STM32 microcontrollers, based on an industry-standard core, comes with a vast choice of tools and software to support project development, making this family of products ideal for both small projects and end-to-end platforms"

An high quality performance microcontroller was needed in order to control the three motors at the same time. In the PC there is the process

of the commands, the recption of the feedback, the matrices calculus, but the bridge to send all this information is the STM-32 board. The board also communicates with the VESC drivers and the POPUP robot in a bidirectional manner, sending out commands and receiving back the feedback position to be given to the PC. All this information are regulated by a protocol. In particular, the communication between the motors and the board is codified by the FD-CAN protocol. The communication with the PC is under the USART protocol. But, first of all, the STM-32 has some elemets to be configured in order to work. Not only the USART and FDCAN configuration were used, but also a couple of LEDs, and a timer to synchronize the operations.

- **USART**

  The USART protocol is used to transmitt data from the PC to the board thanks to a USB cable. The message is codified with a start bit, a data field up to nine bits long, and a stop bit. In these specific case 8 bits were used for the data field. The baud rate was 115200. This data will be important for further development.

- **Interrupt and DMA** The STM-32 interrupt and DMA function were implemented to have a faster communication. Whenever a new data is sent to the USART, the PC stops what he is doing int he main code to save the data, to return running the main code. With the direct memory access, the speed of communication can become even faster. The microcontroller receives a bit, and can directly access the memory part in which to store the data, saving a lot of time

- **FDCAN**

  "CAN FD (Controller Area Network Flexible Data-Rate) is a data-communication protocol used for broadcasting sensor data and control information on 2 wire interconnections between different parts of electronic instrumentation and control system. This protocol is used in modern high performance vehicles.

  CAN FD is an extension to the original CAN bus protocol that

was specified in ISO 11898-1.The basic idea to overclock part of the frame and to oversize the payload dates back to 1999. Developed in 2011 and released in 2012 by Bosch, CAN FD was developed to meet the need to increase the data transfer rate up to 5 times faster and with larger frame/message sizes for use in modern automotive Electronic Control Units (ECUs).

As in the classic CAN, CAN FD protocol is designed to reliably transmit and receive sensor data, control commands and to detect data errors between electronic sensor devices, controllers and microcontrollers. Although CAN FD was primarily designed for use in high performance vehicle ECUs, the pervasiveness of classic CAN in the different industries will lead into inclusion of this improved data-communication protocol in a variety of other applications as well, such as in electronic systems used in robotics, defense, industrial automation, underwater vehicles, medical equipment, avionics, down-hole drilling sensors, etc." [9]

The FD-CAN is essencially a communication protocol faster than the common CAN, the controller area network. It's a robust communication protocol, which has to be configured by the STM-32 with the same baud rate of the USART, otherwise a desyncronization of the data will happen

- **Timer**
  The timer was configured to be fast, but also to control at a constant frequency the data exchanged and transmitted between the actors of the communication. In particular, the timer TIM2 is controlled by an HAL in the main loop

- **main.c and main.cpp code**
  The code written in STM-32 main files are the key of the communication. In particular, in the main file it's present a function who sets the rpm to be sent to the VESC and therefore to the three motors. This velocity is converted into erpm, taking into account the pole pairs of the motors and also the gear ratio. In particular, $erpm = rpm * 21 * 80$. In the main are also present a couple of

function which collect the messages that come out from the vesc. In particular, this messages can be in the form of one group of data, or four group of data. The main.cpp file has the objective to send to the motors the input received from ROS, and also to send to ROS the input received from the motors

Chapter 4

Fixing the problems in the robot control

The first thing that was noticed in the control in the robot in the joint space, was that the robot was moving, but was doing so not in a smooth manner, but lagging a bit. The velocity of the motor was not constant and the movement was not completely smooth. Sometimes the robot even stopped, even with the stick of the controller ordering a move of the motors. In the jacobian transpose operational space control the problems were way bigger. The robot was suppose to move along x,y,z with a assigned velocity by the user with the sticks of the controller. In fact what the robot was doing was totally wrong, like a random movement. This of course cannot be acceptable, and had to be heavily fixed.

The first thing to be done in order to fixing the problem was to run some tests and, thanks to direct visible feedback from the robot, to understand what was wrong.

4.1 Fixing the joint space robot control

To start, the problem of fixing the robot movement in the joint space control was the first thing to be addressed. Therefore, in order to have a clear feedback of what was happening, a log of the data visible to the user while running the code and moving the robotic arm was implemented. This log was then useful also for the operational space control.

```
def send_data_motor(id,pos,speed,kp,kd,curr):
    id_trans = float(random.getrandbits(8))
    msg.data =[id,speed,id_trans]

    pub.publish(msg)

    currenttime=time.time()
    #fileopen=open('loaddata.txt','w')
    #fileopen.write('\n Funge?')
    #fileopen.close()
    f = open('loaddata.txt', 'a')  # creiamo il file obje
    with f:  # usiamo il file object come context manager nel with
        testo=str(msg.data)
        tempo=str(currenttime)
        f.write('\n')  # scriviamo il file
        f.write(testo)
        f.write(tempo)
        # still open
```

As it can be seen in the code above, the function **send_data_motor()**, was already implemented in order to have a visual feedback on the terminal of the Visual Studio Code of the data send to the motors. By just saving the current time in which the function was called, and the text string displayed on the terminal, on a file called loaddata.txt, it was possible to save the log of the file. In order to have a log, it was necessary to stop the program. Therefore, every single test was firstly done on the field, then, after stopping the program, the log of the data was present.

Once the log were present, it was necessary to analyze them. What it was noticed when looking at the log of the data while remembering the action done on the controller, it was that the velocity sent to the motors was not constant. This has a simple explanation and was expected. The control of the speed in the joint space control is done by moving the stick L3, R3 of the PS4 controller.

Thanks to the class **MyController** present online, the movement of the stick corresponds directly to a certain value number. For example, if you push the stick upwards, you may get as value input from the controller something like 20000. This value has to be sent directly to the motor, and the previous code was dividing this value by 32, in order to

send to the motors a speed in erpm compatible with the working ranges of the motors. An example of this code example is below

```
def on_L3_right(self,value):
        global y_L3_right_left
        if(mode_switch_flag==0):
            speed_motor[1]=-value/32
        elif(mode_switch_flag==1):

            y_L3_right_left = value/(640000)
```

What doesn't work in this approach is the fact that the user has to have a completely controlled hand while pushing the sticker upwards. By drifting just a little bit, the velocity of the motor changes. And also, sometimes the stick of the controller locks, sending out a value equal to 0 when the value should be the maximum. Another issue was that on the left stick both the control of motor 1 and motor 3 were addressed. This means that a minor, unwanted shift of the stick out of the 2 axis passing for the center of the L3 stick will move both motors simultaneously, even if the user doesn't want that. Another small issue is that for very low velocity the motors don't move. This is a small issue in this case, a much bigger issue for other kind of controls.
To fix this issue, the solution was simple, however in two steps. The first try was to increase to a low value the velocity under a certain threshold. However, this solution didn't manage to take out the sort of lagging in the operation. Therefore, the best solution and final one was to bring the commands for the three motors out of the stick L3, R3. Now the commands are on the arrows of the controller and on the L1, R1 buttons. In this way, by simply pushing an arrow or a button the robot moves with an assigned velocity in a completely smooth manner. To stop the movement, is necessary to push the triangle button on the controller. The trade off on this approach is that the assigned velocity to the motors is constant, and has to be chosen before running the code.

In the code below there is an example on this approach of controlling the motor number 2. By pressing the up arrow, the motor moves with a velocity of 1000 erpm. This number may of course be changed, but

not while running the code, and this is the trade off at the moment to have a very good performance. In the future probably an evolution of this implementation can be done. **mode_switch_flag** is a variable which changes on the basis of the mode the user wants to control the robot. By pressing circle, the control is in the joint space, by pressing X is in the operational space, by pressing square, the control is in target reaching mode.

```
def on_up_arrow_press(self):
        if (mode_switch_flag==0):
            speed_motor[2]=1000
        if(mode_switch_flag==1):
            global z_R3_up_down
            z_R3_up_down=-0.05
```

4.2 Fixing the operational space control

As stated before, fixing the operational space jacobian inverse control was way more difficult than fixing thee joint space one. In fact, the structure and complexity with this kind of control, and the implication each single component of the code and of the communication structure has on it made the whole work quite challenging.

The first thing to be done was to analyze the log of the data, and therefore the code implementing the jacobian inverse control.

4.2.1 The analysis of the data

By analyzing the data and looking at the visual feedback in the lab environment, what was noticed was that the first big problem: in the log of the command the command was the same for some successive instruction. At the beggining the thought process was that this was normal. The robot has in fact to move to another position before receiving another instruction. Once the robot has moved to another position, then the feedback from the motors shall change, and therefore the command given to three motors. While in theory all of this is correct, in practice what happened was that the command stayed constant for some time istant while the robot was moving. In particular, each command was sent out every 0.11 seconds, so maybe the robot had the same command for 0.4 seconds, which is a lot and explained why nothing was working. The robot was trying to fix his position but doing so it was applying the algorithm to a past position. In this way, nothing can work.
But before analyzing how to fix this issue, there were other elements of the structure to be analyze, the first being the code itself.

4.2.2 The analysis of the code

```
def jacobian_norm_function():
    [speed_motor[1],speed_motor[2],speed_motor[3]] =
    jacobian_fnc(fb_m_pos[1],fb_m_pos[2],fb_m_pos[3],
    x_L3_up_down,y_L3_right_left,z_R3_up_down)

    #x_L3_up_down
    a = [[speed_motor[1],speed_motor[2],speed_motor[3]]]
    normalized = 1500*preprocessing.normalize(a)
    print(type(normalized))
    for j in range (1,4,1):
        if(normalized[0][j-1]<5000):
            speed_motor[j] = normalized[0][j-1]

        else:
            speed_motor[j]=0
```

So, the first thing to be notice in this function implementing the jacobian inverse control is that the output are the speed to be given to the three motors, while the input to the function computing those three speed is vector with the three feedback motor position from the three motors and the VESC, and also the input from the user, who decides whether the robot has to move along x, y or z or a combination of the three.

Before analyzing in depth the code of this function, it's interesting to notice that the speed of the motors are not sent to the motors as they come out from the function. Instead, they are normalized between 0 and 1500. Why is this happening? In order to understand that, what was tried was to remove this normalization. Overall, it didn't make much sense, and had to be understood. In the only experiment done without this normalization the robot behave like crazy, rotating at a crazy high speed, completely losing control and cutting the pipe act to pump the air in the inflatable links. A similar behaviour was also dangerous, since the user risked to take the link of the robot directly in the face.
Once it was understood why the robot had this limitation between 0

and 1500 erpm, it was to be determined why to normalize the data out of the function. To maintain the sort of proportion between the speed of the motors this was necessary, and was useful also for the lower speed, that the motors just can't achieve. However, a more robust further control was introduced. Even though from the motors the maximum absolute value coming out shall be 1500, sometimes higher value are achieved. A maximum absolute limit of 5000 is capped, otherwise, for the security of the user, the robot will be stopped.

Now the code of the actual function computing the jacobian can be analyzed

```
J = np.transpose(cross0)
    f = open('loaddata.txt', 'a')  # creiamo il file obje
    with f:  # usiamo il file object come context manager nel with
        jacobiano=str(J)

        f.write('\n')  # scriviamo il file
        f.write('J=')
        f.write(jacobiano)


    print('J= ', J)



    J0_T = np.linalg.inv(J)
    joystick_in = [j1_in_x, j2_in_y, j3_in_z]
    conversion_factor=60/(2*math.pi)
    k= (conversion_factor*80*21)
    [q1dot, q2dot, q3dot]= np.matmul(J0_T,joystick_in)

    return [q1dot*k, q2dot*k, q3dot*k]
```

In the first part of the code, which here is not reported, the Jacobian function is computed. Of course a check of all the values to be put in the matrices was done, but unfortunately, the error was not there. Once the jacobian inverse function is computed, a good idea was to implement a storage of it in the file, and also to put it in the terminal for the user to be seen. Now, a necessary element in this process was the conversion factor. Remebering that the function calculates the speed in rpm, and the motors need in erpm, the conversion factor multiplication of 21*80

is necessary. However, the input of the matrices calculating sine and cosine were in radiants per second, so it was necessary to convert them back in degrees. By multiplying the Jacobian inverse function with the input from the joystick, and multiplying that by the conversion factor gives the three speeds.

The conversion factor between degree and radiants was not so accurate because instead of 2pi it was 6.28 the number written in the draft original code. This minor change can help the code to be more precise and accurate

4.2.3 Analyzing the motor feedback behaviour

As stated in the previous sections, one of the issues was that the command in the jacobian function was the same for consecutive messages. The reason for this has to be in the feedback from the motors. If the feedback from the motors doesn't change, then it can't change the calculus of the speed from the jacobian, since one of the inputs is the feedback from the motors itself. Therefore, it was decided to run some tests on the motors, at different velocities, following a specific pattern, to understand if there were problems in the transmission of the feedback. It turned out that the issue was important.

The three motors were each tested at a speed of 300 erpm, 500 erpm and 1000 erpm. A first thing to be put on evidence is that motor 2, which has to bear the greatest load, was not able to execute a movement between 0 and 90 degrees when the speed was too low. And even in the case of 500 erpm, the movement was not smooth. This was similar for motor 1 and 3. Motor 3 was able to move at each assigned velocity, however, at a velocity of 300 erpm the movement was not smooth and full of lag. For motor 1, the same characteristic of motor 2 were observed by the user, however, at 500 erpm the movement was smoother, although with some lagging. Having said that, what was done was collecting the data of the feedback of motor position in degree received by the motors, and this is what came out.

Motor 1 Test

As it is possible to see in the figure 4.1 , in which on evidence there is a section of the multi run of the motor 1, the feedback from the motor is not smooth. In particular what happens is that in theory the motor should go from 5 degrees to 6 degrees to 7 degrees, for example. Here, instead, the motor stays at a constant 8 degrees for 5 to 6 cycle clocks, then it jumps directly to 12 degrees, then to 15, 16, 17, before coming back and then having a huge jump. This is a major concern, but the thought process was that maybe the speed of the motor was too low. Watching the robot in the lab, at a low speed of the motor,

the movement was not smooth and sometimes was even coming back a bit. So, a test at 1000 erpm, a speed in which the motor behaviour was instead totally right, was necessary.

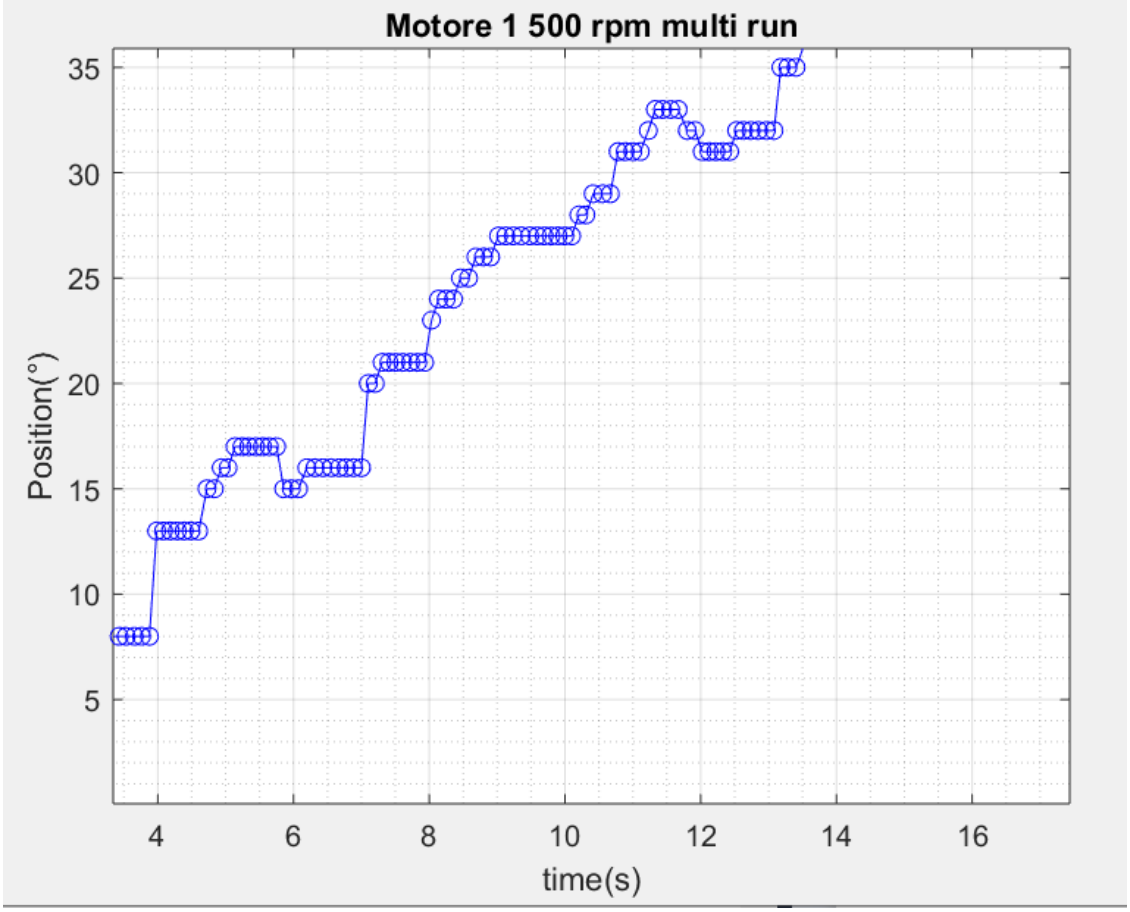

Figure 4.1. The motor 1 test at 500 erpm

The first thing to notice in the motor 1 test at 1000 erpm is that the situation is better. Here, no unexpected coming back in degrees are observed, in figure 4.2. However, the problem of the feedback degrees staying constant and then jumping is an issue still not solved. For example, we have a stall at 31 degrees and then a jump to 37 degrees. It seems like the feedback from the motor, which is fast and arrives every 0.11 seconds, can't follow what motor 1 was doing. Of course, a test also on motor 2 and on motor 3 was absolutely necessary to

determine if the problem was only on one motor, or on all the 3.

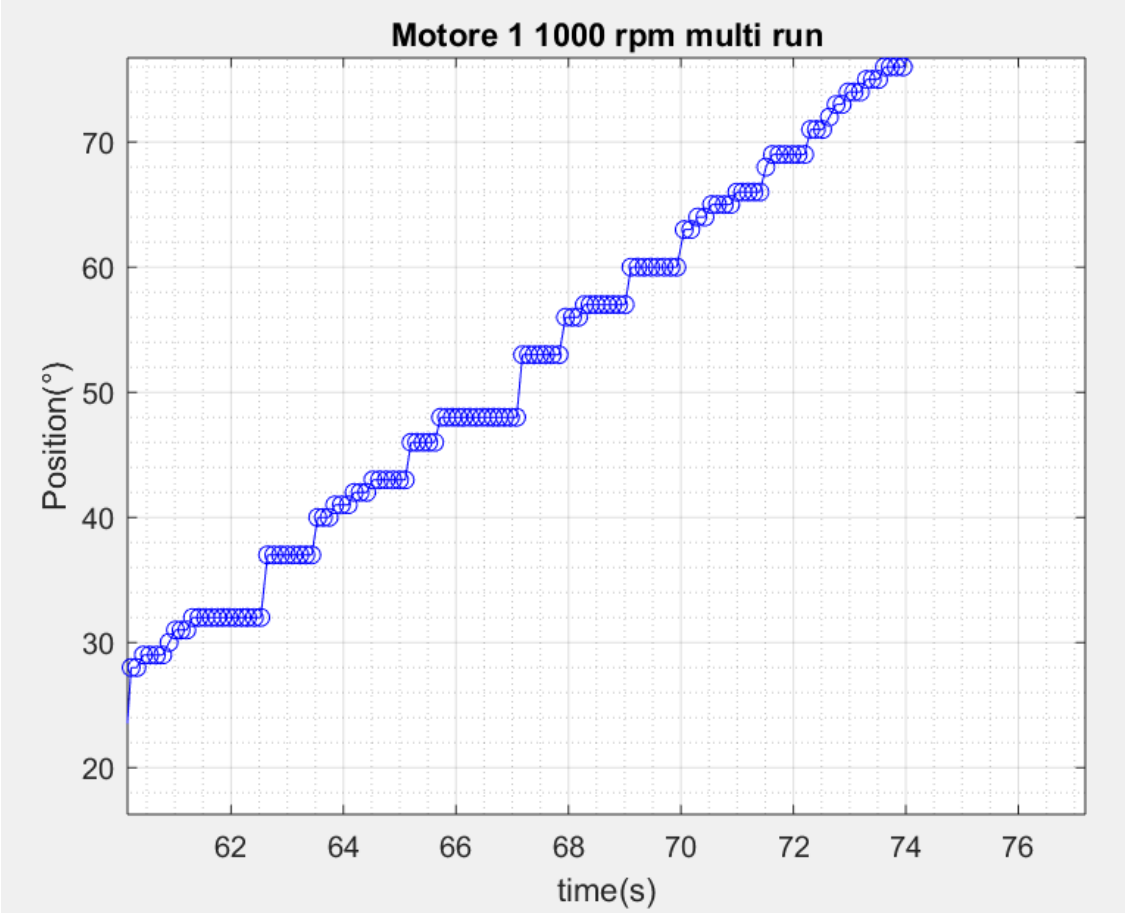

Figure 4.2. The motor 1 test at 1000 erpm

Motor 2 Test

The test on motor 2 at 500 erpm displays a similar behaviour to the one on motor 1 at 500 erpm. In particular, these jumps in degrees are present, although more sparse in the space, as shown in figure 4.3. In this specific 14 seconds section, for example, a jump between 29 and 27 was observed, as well as from 24 to 22, 13 to 10, and 7 to 5. This are not huge jumps, so the situation might seem a little better. But this can be just a case of good behaviour in a limited space of environment. A rigorous analysis of the whole path will evidence in fact that the jumps

in degrees are present almost everywhere.

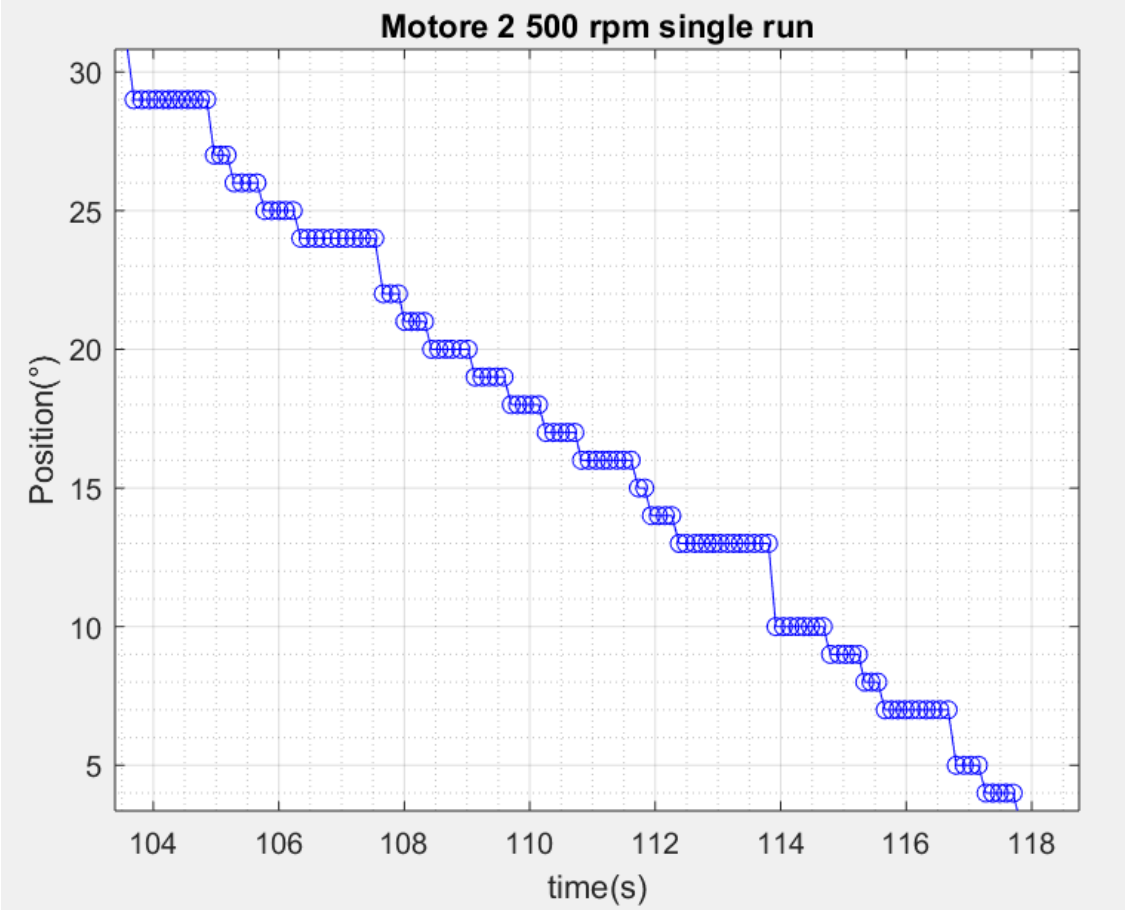

Figure 4.3. The motor 2 test at 500 erpm

In the figure 4.4 is evident that at 1000 erpm, in another section with respect to the one chosen before, almost everywhere a jump of 2 to degree is present, for the given time interval. So the behaviour of motor 2 feedback is similar to the one in motor 1. It has still to be analyzed if motor 3 will confirm this trend, as it seems probable, or it reverts it

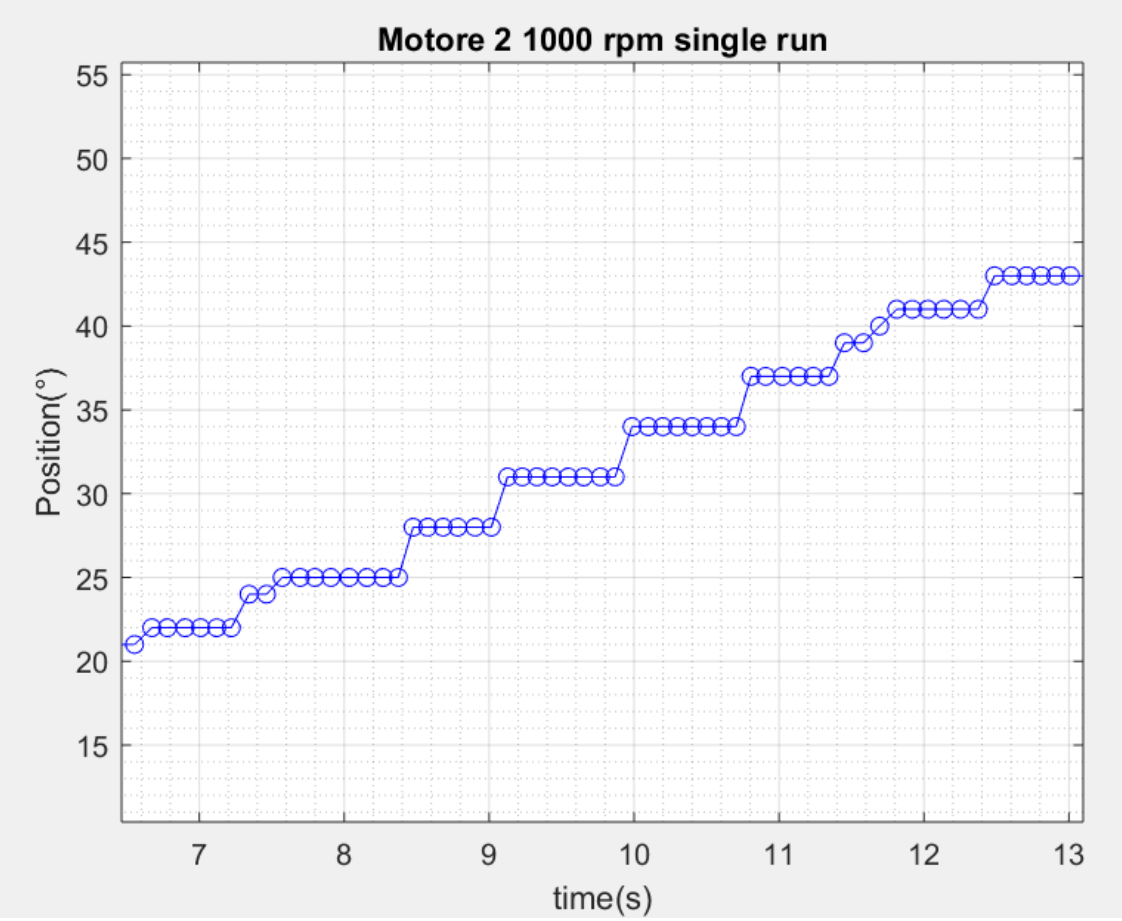

Figure 4.4. The motor 2 test at 1000 erpm

4.2.4 Motor 3 test

Motor 3 was tested at 3 different speed, considering the fact that it responded to the 300 erpm low velocity. So three test were present here. A first thing to notice, is that at 300 erpm the feedback from the motors seems to respond pretty well, as shown in figure 4.5. And by the way, this was observed at the test of motor 2 at 500 erpm, too. The reason to explain this fact has to be in the fact that both the sections were picked in the descendent part of the movement. In fact, figure 4.6 has not a very good behaviour. It is important to say that the motors have a PID controller implemented in the VESC drivers, however, no

gravity compensation is present.

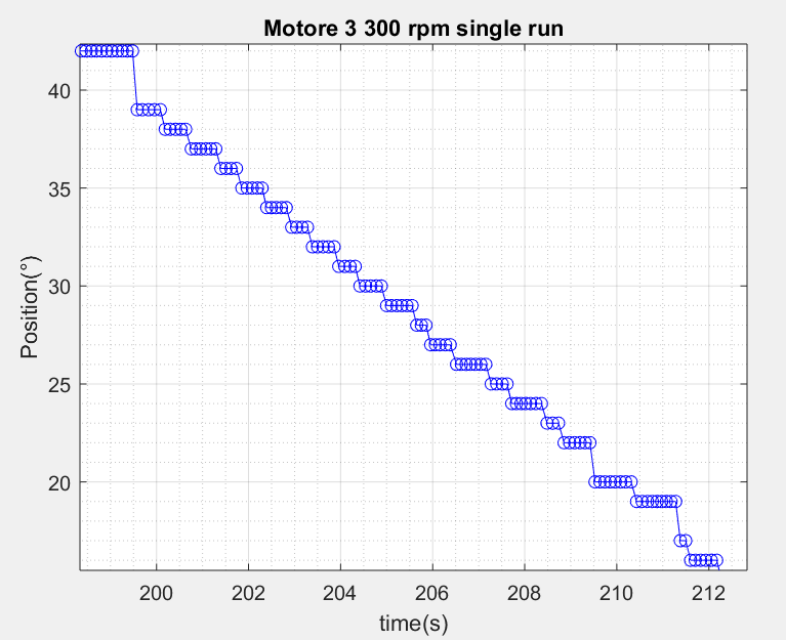

Figure 4.5. The motor 3 test at 300 erpm

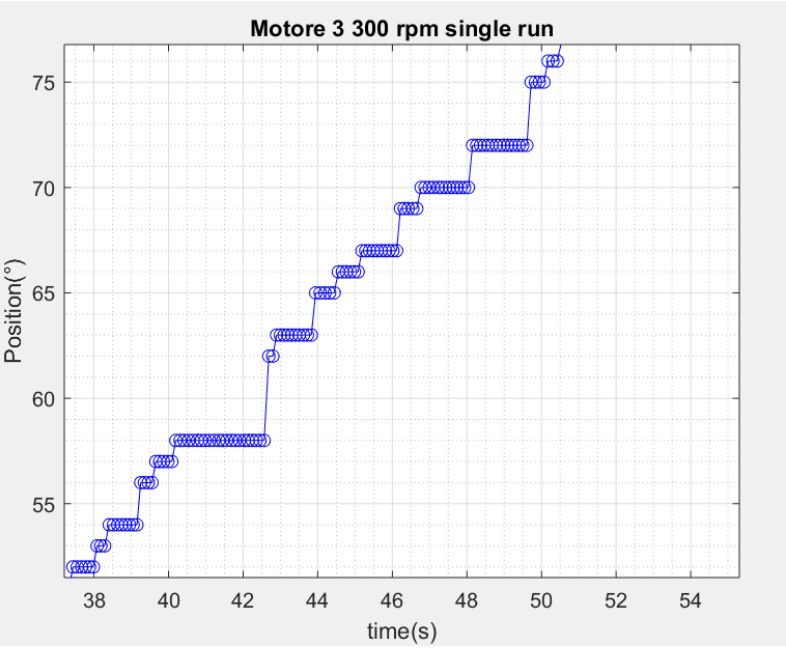

Figure 4.6. Another section of motor 3 test at 300 erpm

Probably, the fact that there is no weight to carry to come down, although the movement is smooth and not a free fall, might help a smoother transmission of the data. In fact, a different section in the upper part presents way more lagging and jumping the steps in degrees. So, the problem is still present.

For what concerns the situation at 500 erpm, there is no surprise in observing that the behaviour is still the one with jumps between angle positions, as shown in figure 4.7

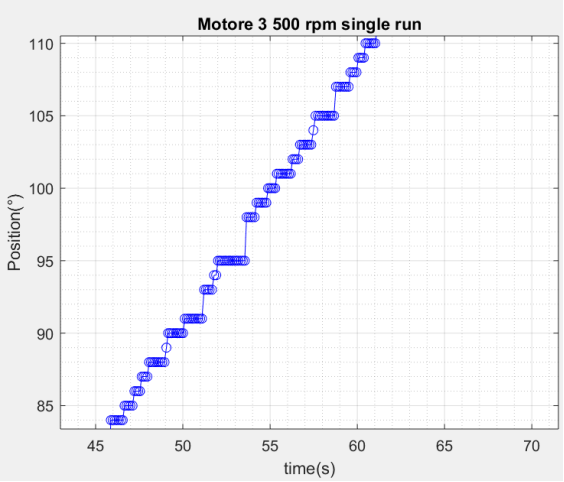

Figure 4.7. A section of motor 3 test at 500 erpm

And even at 1000 erpm, the situation doesn't change. The jumping are still observed, and this is a problem evident in figure 4.8.

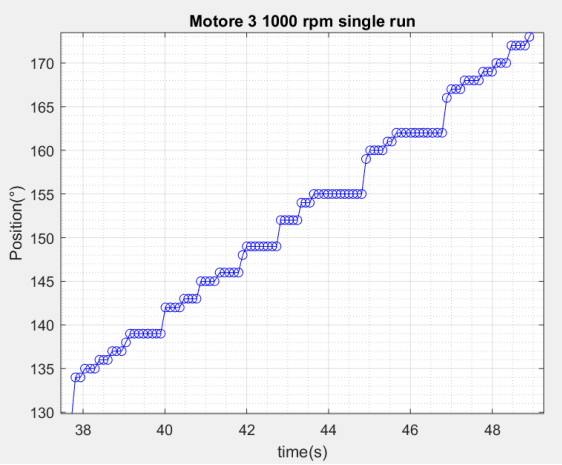

Figure 4.8. A section of motor 3 test at 1000 erpm

Now that all the test are finalized, the problem has finally emerged. The commands in the jacobian inverse operational space control were staying constant while the robot was changing position exactly because the input of the controller jacobian, being the motor feedback velocity, had those problems. Apparently, the motors are responding too slow to the whole system, causing a domino effect that puts KO the control system, leading to a weird and unexpected behaviour. To solve the problem, the transmission speed has to be increased. But how much faster is it possible to go? Can the issue be solved in a decisive way?

4.3 Fixing the feedback motor problems

Once the feedback from the motors has determined to be the issue on the jacobian inverse operational space control of the robot, the problem had to be solved. In the previous chapter was stated that in the communication system, 3 were the main actors, so each of them can contribute to make the communication faster. The first thing to be noticed is that the FDCAN baud rate can go up to Mbit/s, as we may see in the image below. However, the USART is fixed to 115200 bit/s, which is way lower than the limit allowed on the FDCAN itself. The idea is that increasing both shall be the key to improve the communication. But there is a problem. A third actor involved in the communication system are the VESC drivers. The VESC drivers are in fact regulating the communication between the motors and the STM-32 board. This means that it's possible to improve for sure the velocity of the communication on the board, but this will be completely useless, unless also the rate of communication from the motors to the drivers and to the STM-32 board increase.
In the figure 4.9 it's possible to analyze the structure of the program loaded on each of the VESC drivers. In yellow are put on evidence a couple of key parameters: the CAN baud rate and the CAN Status message mode. It's important also the parameter between the two, the CAN Status rate, which has a frequency. Changing this frequency will be important in order to fix the feedback problem.

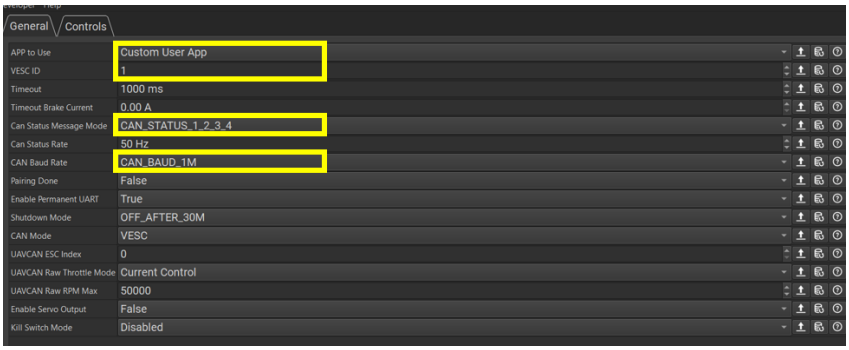

Figure 4.9. The VESC tool interface

First, the CAN Baud Rate represents a limit to the velocity we can assign to the FDCAN and therefore to the entire transmission system. Luckily this limit is 1 MBit/s, the FDCAN can go faster, and so the bottleneck seems to be the USART interface. By speeding up the velocity of the USART transmission to 1 Mbit/s, the velocity of communication should be higher. However, by doing that and conversely changing the corresponding baud rate on the ROS node act to be the bridge between the two platforms, a communication error occurs, and therefore a transmission KO happens, leading to a desynchronization of the code. Why is that possible?
In order to understand that, a step back is needed. In fact the CAN transmission mode from the VESC is set to transmitt message 1,2,3 and 4. In the file containing the instruction to program this stuff, in the status message 1, the data of the feedback rpm velocity from the VESC is present. In the status message 4, the data of the angular position of the motor is present. The VESC tool doesn't allow to send only message 1 and 4, 2 and 3 have to be sent too. Each message accounts for 8 byte, which are 64 bits. We have 3 motors, so the bits are in fact 192. So, basically 192 bits come out from the VESC to go to the board. Each of those information is codified with the CAN protocol. Accounting for that, each message weights in fact 116 bits, so with three motors there are 348 bits. However, the message sent out are 4. This leads to something like 1392 bits per message. With this data passing through the USART interface, even with the unpacking of the unuseful data, the frequency of the transmission rate from the VESCs is capped at 50 Hz, more or less.
Now, ROS has to process those datas in order to use them. And with this amount of data arriving, ROS needs a buffer, or it will lose data. At the start of this project he buffer was 1024 byte long, for transmission and reception. In figure 4.10 the standard statement to handle the nodes is reported.

```
namespace ros
{
typedef NodeHandle_<STM32Hardware, 25, 25, 1024, 1024>  NodeHandle; // default
25, 25, 512, 512 This set our NodeHandle to have 25 Publishers, 25 Subscriber,
1024 bytes for input buffer and 1024 bytes for output buffer.
}

#endif
```

Figure 4.10. The ROS buffer configuration

What is interesting is that by diving by 2 that value, it was possible to increase the CAN status rate up to 100 Hz, and by doubling the USART baud rate and the corresponding baud rate on the ROS Node, the transmission was twice as fast. So, the next logical step was to see by how much it was possible to increase those value. And therefore a try on going with a baud rate of 460800 to go with a frequency of 200 Hz was done. Unfortunately, this led to a desync. The reason for this is that the buffer is indeed needed. With blocks of information running at high speed, ROS need a buffer to recover the information it can't immediately process. And so at 230400 bit/s and with a frequency of 100 Hz the situation was a lot better, with a buffer of 521 bytes. But in the end, rare transmission KOs events occurred.
This events were even more frequent when implementing the target reaching control, so the best trade off was to increase the CAN Status rate up to 75 Hz, with a baud rate of the USART of 172800. This is a small increase, but improve greatly the whole code. The robot is now moving along x,y,z, although not completely precisely, and the situation has improved greatly. Which led to the idea to start implementing a draft of a target reaching command in the space

4.4 Target reaching control implementation

To reach a target in the space with its end effector, the robot has to know the position of its target. This is done by the user, which sets a point in the space, and has to measure its position with respect to the 0,0,0 point of the inflatable robot. To do so, the faster way was to use the meter, accepting for the measurements error, at least in a preliminary phase to check if the implementation qualitative works.
Having said that, once the point in the space is determined and set, it is necessary to determine the position of the end effector of the robot. According to the code below, this is done by implementing a function which has to compute the forward kinematics. As an input of the function are the feedback from the motors, which give the angular position of the three motors.

```
def target_reach_function():
    target_point=np.array([0.25,-0.88,0.43])

    direction_of_movement=np.array([0,0,0])
    end_effector=forward_kinematics(fb_m_pos[1],fb_m_pos[2],fb_m_pos[3]
```

After defining the parameter of the Denavit-Hartenberg convention for the robot, which are of course the same defined also for the jacobian inverse operational space control, the transformation matrices from link 0 to link 1, from link 1 to link 2, and from link 2 to link 3 has to be computed. Then, each of these matrices has to be multiplied in the precise order in which they were computed, obtaining in this way the transformation matrix from link 0 up to link 3.

```
def forward_kinematics(q1,q2,q3):
    a = [0, 0, 0.728, 0.684]          #m
    alpha = [0, math.pi/2, math.pi, 0]   #rad
    d = [0, 0.15270, 0.03935, 0.15780]   #m


...Matrix calculus


    p = R03[0:3, 3]

    [x_end,y_end,z_end]=p

    return[x_end,y_end,z_end]
```

The whole transfromation matrices is then ready, having the position and orientation of the end effector inside its columns. In particular, according to the theory, the cartesian position of the end effector is in the first three elements of the last column of the matrix. This position is saved in a vector and returned to the main script, in order to be used to compute a direction of orientation to be given to the whole code.

```
direction_of_movement2=
np.subtract(target_point,end_effector)

    direction_of_movement=[[direction_of_movement2[0],direction_of_movement2[1],
    direction_of_movement2[2]]]



    direction_of_movement_normalized=
    0.05*preprocessing.normalize(direction_of_movement)
    xupdown=[direction_of_movement_normalized[0][0]]
    yupdown=[direction_of_movement_normalized[0][1]]
    zupdown=[direction_of_movement_normalized[0][2]]
```

What is done next is subtracting from the target point the end effector position. In this way a new vector, with the orientation the end effector shall have, emerges. And by putting iit in a vector and multiplying it by 0.05, an assigned velocity to reach the target is now given. However, here a normalization is present, too, in order to move for each

direction between 0 and 5 cm/s. Once xupdown, the direction with velocity along x, yupdown and zupdown are determined, then it remains to use the exact same jacobian inverse control to give to the motors the correct velocity to rech their targets.

In theory all of this shall work, however this is not completely true in practice, and this is the reason why this control is still in a draft stage at the end of this project.

To control the behaviour of the control and whether it was working or not working, a log of the data was implemented. In particular, in the code below is possible to see that the end effector position, along with the vector direction of movement and feedback motor position are stored in the file. In this way, when the robot theoretically reaches the target, starting to tremble around a point, there is a chance to see whether in its end effector space that is the point it has to reach, or if simply it stops because it can't achieve the requested configuration. It has to be stated that although some degrees of liberty are present in the robot, the arm is fixed on a table and in a lab, and it can't achieve the whole 3D space configurations, even because of the wires and of the pipes that are there to link all the system. In some configurations, the robot may cut those wires and pipes, so it has to be avoided.
This is one of the reason why this kind of control doesn't work properly, and the resolution of this is not easy, at least in a preliminary phase of the research project in which the robot is still anchored at the table

```
f=open('direzionemov.txt','a')
    tempoistant=time.time()
    with f:
        f.write('\n')
        f.write(str(xupdown))
        f.write('  ')
        f.write(str(yupdown))
        f.write('  ')
        f.write(str(zupdown))
        f.write('  ')
        f.write(str(tempoistant))
        f.write('  ')
        f.write(str(end_effector))
        f.write(str(fb_m_pos[1]))
        f.write('  ')
        f.write(str(fb_m_pos[2]))
```

```
f.write('  ')
f.write(str(fb_m_pos[3]))
```

4.4.1 The problems in the target reaching and the solution attempted

As stated before the target reaching control is still in a preliminary phase and therefore not finalized when this project is being written. A first thing to notice is that the log of the data was very useful for understanding well the data to be chosen from the forward kinematics configuration. Having the end effector position at the end of the code helps realizing if the robot is doing what he shall, i.e. reaching the target, or if its behaviour is wrong.
In this sense, a thing that was observed during the many tests is that the initial position of the robot, with the starting configuration with the robot fully extended on the ground, gave to the end effector a different position with respect to the one measured with the meter. And this is a issue to be solved, because an error in terms of some cm in the starting configuration may lead to a far greater error in the final result. The reason for this error has to be right at the start of the whole process of controlling the robot.
What the user shall do to control the robot is open the pump who inflates the area in the links, turn out the current that is going to feed the motors, and attach to the USB port of the PC the STM-32 board and the PS4 controller. What happens is that the STM-32 board starts acting as soon as it is feed by the current of the USB port. So, in the space of time from the linking of STM-32 to the actual start of the program in python, the STM-32 has a sort of memory of its action. This results in a starting configuration where the robot angles are not 0,0,0 as they should, but instead they are something like 359,0,357. And this leads to a starting position of the end effector which is not the correct one and as a domino effect to many other problems.
To fix this issue the approach was to try to remove this offset from the motor feedback position. So, the idea is to store the first measure of the angular position of the motor, knowing that is not correct, but has an offset of some degree to each of the motors. Then, this value will be subtracted to the next measure the robot will receive, once the user has pressed the start bottom. In this way the offset shall be removed,

and the user should be able to see, once it plays start on the controller, the feedback from the motors being 0,0,0.

```
def callback_status(data):
    global str_status,flag_offset
    #rospy.loginfo('I heard %s', data.data)
    str_status = data.data
    print(str_status)
    #condizione if "Press START to begin..." posizionare flag offset a 1
    if(str_status=="Press START to begin..."):
        flag_offset=1
        #print(type(str_status))


#read message on topic /motorFeedback
def callback(data):
    #rospy.loginfo('I heard %s', data.data)
    global fb_motor, flag_offset
    fb_motor = data.data
    #if flag==true: inizializza offset motori e posiziona flag a 0
    if (flag_offset==1):

        initial_feedback()
        print("Initial feedback")
        flag_offset=0
```

After implementing this code, which is done by simply taking the position of the motor feedback once a flag is turned on, what happened is that the end effector position at the starting configuration, as observed from the log of data, was exactly the one measured with the meter. So, a minor source of error has been eliminated.
The target reaching control is implemented by pushing the square button on the PS4 controller. However, this command has been implemented on the stick L3 at a cost of having a little shaking in the movement of the robot. The user has to keep pushing the L3 stick upwards in order to move the robotic arm, and once the user doesn't push anymore the stick, the robot immediately stops. This was done because, especially in this kind of control, having another bottom to stop the robot was somewhat dangerous. Once the robot is very close to target,

and it only has some centimeters as a difference between its end effector position and the target point one, it starts to shake from one position to another one, because it accounts only for minor adjustments, that are anyway present and it has to fulfill. Once the robot start to shake, the user shall be able to stop it just by not giving any other order.

For what concerns the results of the target reaching point control, there are some promising signs, however they are not satisfying the accuracy and the repeatability enough. In short terms, what happens is that the robot does the correct movements to reach the target in some case, in other cases it picks a counter intuitive trajectory. And in general, the accuracy at the ending position is still a work in progress. Sometimes the robot misses of 3-4 centimeters, other times of 20 centimeters. The problems are usually along the z axis, where major offsets were observed. A reason for this may be in the fact that the robot does not have a gravity compensation strategy implemented, and therefore the PID integrated on the VESC can't adapt fast enough when the robot arm is passing trough the vertical. Anyway, the first steps in order to achieve this control are done.

Chapter 5

Final tests and graphs

Once all the implementation of the codes and the updates were finalized, a necessary step to be done is to validate the code and the methods written. This cannot simply be done by showing the robot moving, but it has to be rigorous. But before jumping into these arguments, it is appropriate to elaborate on which were the major concerns observed in the laboratory session tests.
For what concerns the joint space control, the good news is that the control works properly, and no issue of any kind were found. However, when sending to the motors speed too low, in the order of 400-500 erpm or less, the motors shake a lot or simply can't bear the load, as in case of motor 1 and 2. Apart from this minor concern, for what regards the joint space control, no other concerns were observed.
For what concerns the operational space jacobian inverse control and the target reaching control, here is a sort of list of the major point of concerns.

- **Motor 3**
  What was observed in the jacobian operational space control is that motor 3 doesn't work properly for some specific configurations. In particular, when the speed of the motor is low, the motor can't achieve that. And another issue is that while moving the motor along x,y,z, a move from motor 3 is needed sometimes in order to achieve a straight line, but motor 3 is slow to react and sometimes just doesn't do what is ordered from the console.

- **Singularities**
  This problem was omitted before, but what happens if the user starts the program on python and then decides to immediately go using the operational space jacobian inverse control is that the robot goes into a singularity. In particular, the jacobian matrix cannot be inverted since it is rank deficient, and we have an error in the transmission, since the robot just can't have a proper command. It's adviced to press immediately the emergency stop button for this issue, in order to avoid damages. The robot will move anyway when the singularity happens, so being careful is necessary. However, this problems is of a minor concerns. The user has just to be careful and to avoid to start using the jacobian inverse operational space control when the robot is in a singularity and nothing wrong will happen.

- **Some configurations cannot be achieved**
  This point is directly connected to the previous one: it's not possible to put a robot in a singularity, and this limits the potential of the movement along x,y,z. If the trajectory along x,y,z will lead to a singularity, what the robot will do is to start trying to force itself to not go in that configuration, resulting in a confusing behaviour for the beginner user who may not understand why the robot is stopping and not going ahead. Other than this, it is recommended to avoid impossible situations for the robot. If the robot is straight up, or straight along the table, is not recommended to give the order to move along x or along z the end effector, because the robot cannot achieve that configuration, but it will try anyway, with the risk of breaking down

- **The feedback of motor 2 angle is not precise in the vertical position**
  One of the issues that may be difficult to solve is the fact that the feedback of the robot angle position is nowhere near precise when the robot motor 2 forms an angle of 90 degrees with the starting position. In fact, the reading from the motors is of about 70 to 75

degrees, which is a problem still to be understood

- **The target reaching control is not always repeatable**
  This is a problem noticed while running many tests and one of the reason because the target reaching control is still not validated but just in a draft phase. The fact is that the control is not repeatable in the sense if that the robot reaches the target, and is moved far away from it, it finds very difficult to reach the target again, which shouldn't make sense. A further investigation on what happens on the motor feedback position when the robot is controlled for several minutes is probably needed

Having said that, to validate the motors and their velocity that was acceptable in order to have a correct behaviour of the three methods, two test were run. The first one took some of the velocities from 200 erpm to 1500 erpm in order to understand where was the point of malfunctioning of the motors at low velocity. The second test was done with OptiTrack, an equipment certified in order to watch how the trajectories of the controls were precise.

5.1 Motor velocity tests

In the motor velocity test, all the three motors were tested for the constant speed of 200,400,600,800,1000,1200,1500 erpm. The data taken were the timing of the test, the motor feedback position, and the motor feedback velocity. It is important to understand if the three motors behave in the same manner, and with the same timing. Does motor 1 took the exact time of motor 2 and 3 to go from 10 to 70 degrees at 600 erpm? The test were done to answer this question and also to see what was the sort of breaking point, where the velocities were not achieved by the motors.
So, for the tests two configuration were used in order to understand whether there was a change for motor 1 and motor 2. Of course motor 3 doesn't change. The first configuration was done with the arm in the

starting configuration, completely extended.

5.1.1 Motor 1 test: starting configuration

First of all, at 200 erpm and 400 erpm motor 1 was not working. Therefore, reliable data are collected only from 600 erpm up to 1500 erpm. As it is possible to see in the figure 5.1 , what was plotted are the motor feedback velocity over time, with the motor moving from 0 up to 90 degrees and coming back. The reference line at 600 and -600 erpm are plotted as well as a reference point. With this velocity, the robot motor behaviour is totally wrong, with a noise unacceptable and a variance sky high. Therefore, 600 erpm is not a value that motor 1 can achieve properly, and this is evident. That also explains the tremble observed by the user

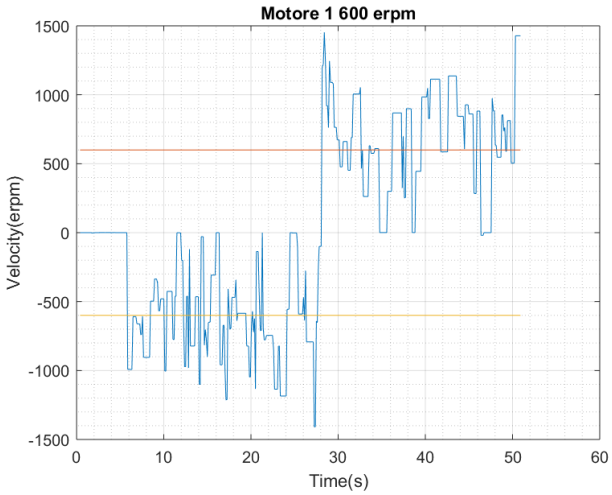

Figure 5.1. Motor 1 at 600 erpm

For what concerns 800 erpm, the situation is better, as shown in figure 5.2, at least for what concerns the coming back part, that in fact, as observed by the user, was smooth. However, it's not possible to accept the noise and the variance in the upgoing part of motor 1 tests, since the noise is too high and the variance unacceptable. Therefore, probably the safe threshold is even higher, probably at about 900 erpm

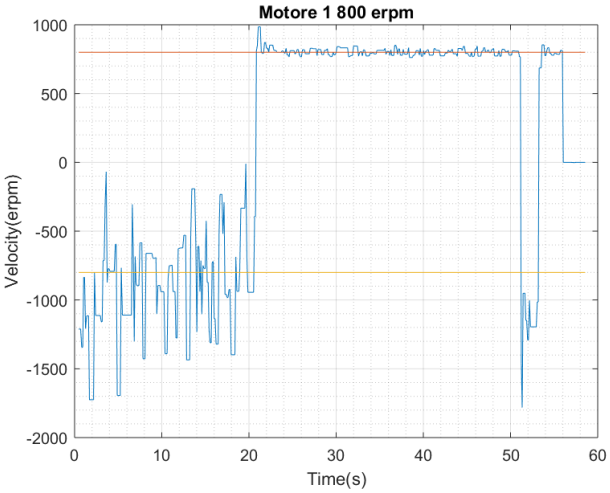

Figure 5.2. Motor 1 at 800 erpm

When the velocity speeds up to 1000 erpm, no major issues are observed. The noise falls down, and the error is very slow, while the user can see that the commands and the movement are way smoother. However, some little noise is still present, see figure 5.3. In the steady state, an error up to 40 erpm is observed, in both of the sense of the test. While this error is just of 4%, it is still a notable error. The user can't notice major drift and trembles, therefore 1000 erpm is a good velocity, but the threshold between the motor working well and the motor not working well, at least in this position, is about 900 erpm.

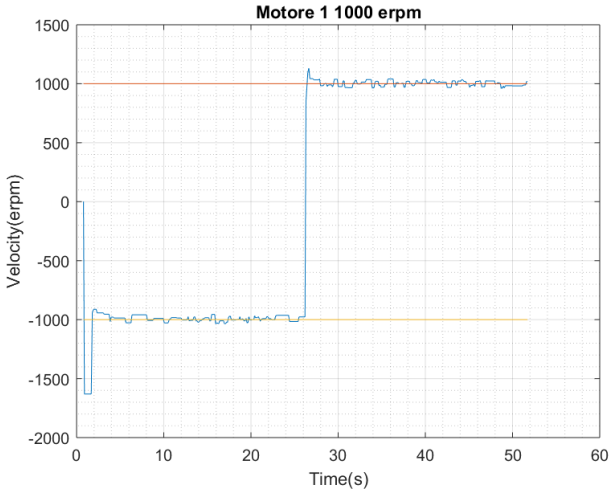

Figure 5.3. Motor 1 at 1000 erpm

In the next figure 5.4, there is the test at 1200 erpm. Here the noise seems to be even lower, and the movement is observed as totally smooth. The maximum distance form the reference of 1200 is of 25 erpm, which accounts for an error of about 2%, which is very good for this configuration.

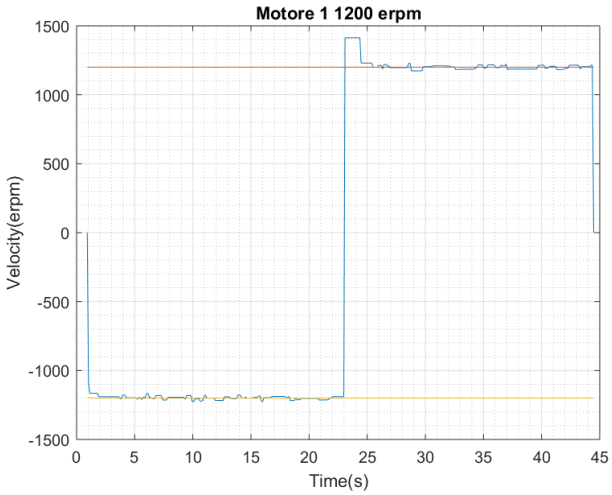

Figure 5.4. Motor 1 at 1200 erpm

For what concerns 1500 erpm, the situation is even better, see figure 5.5 , with a maximum error of about 25 erpm, which is less than 2%. Therefore, it is possible to affirm that after 900 erpm the bahaviour of the robot motor 1 is totally fine and expected

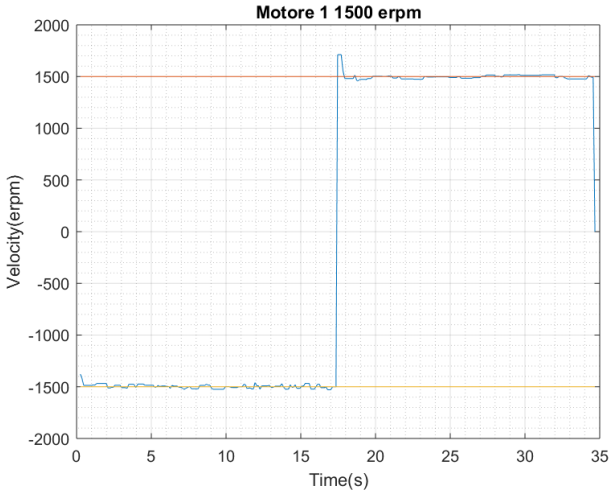

Figure 5.5. Motor 1 at 1500 erpm

So, to resume the whole experiment, with a velocity of 800 erpm or less motor 1 doesn't act in a properly way, while from 900 upwards it does what it should.

5.1.2 Motor 2 test: starting configuration

For what concerns motor 2 in the starting configuration, at 200 erpm, as well as at 400 and 600, the robot just did not move. Probably the weight to carry was too high for that slow velocity, accounting for the fact that a gravity compensation is not present. Therefore, the test data collected are from 800 erpm up to 1500 erpm only.
At 800 erpm, the robot moves, however the rise is very difficult, as it is possible to see in the first 10 seconds of the test in figure 5.6. At steady state the situation is better, but still an error of about 150 erpm is observed. This is about 18%, therefore is not acceptable. It has to be state dthat at 800 erpm motor 2 doesn't work well

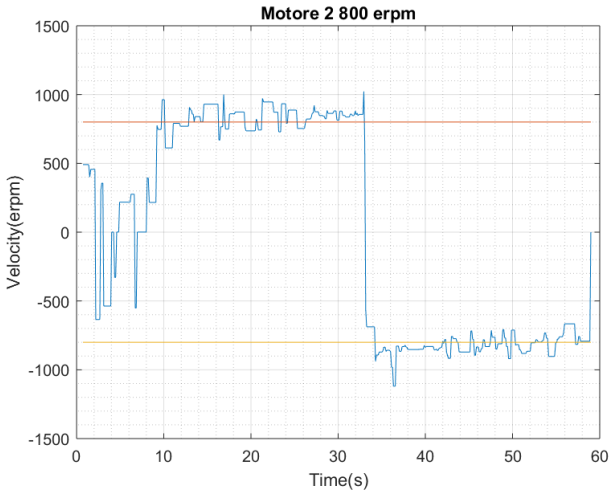

Figure 5.6. Motor 2 at 800 erpm

For what concerns the 1000 erpm test, the situation is better, see figure 5.7. No tremble is observed, and the data suggest a maximum error of about 100 erpm, or about 10%. While the user does not notice this drifts, in fact they are present. The decision is to say that this velocity is the threshold, however for further development a gravity compensation will be necessary, because this is an high value

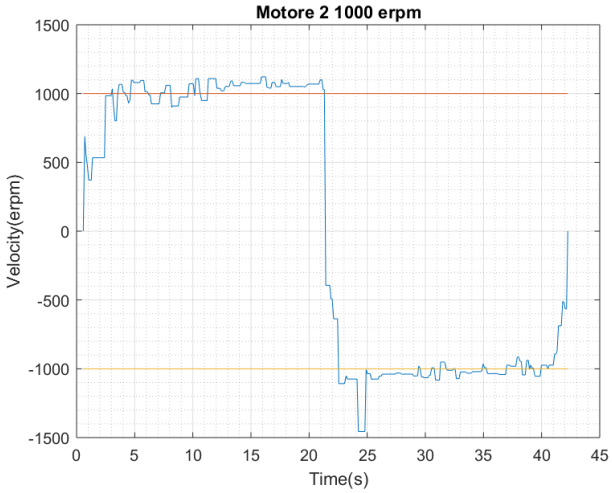

Figure 5.7. Motor 2 at 1000 erpm

At 1200 erpm, the situation improves, but still an error of 8% is present, while no drifting or tremble is observed by the user, as shown in figure 5.8

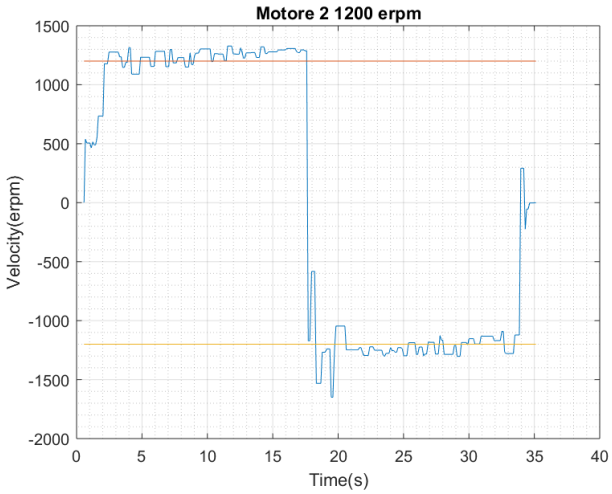

Figure 5.8. Motor 2 at 1200 erpm

Finally, at 1500 erpm, an error of about 6-7% is observed from the reference velocity. This is still an issue, probably due to the gravity, but no strange tremble is observed in figure 5.8

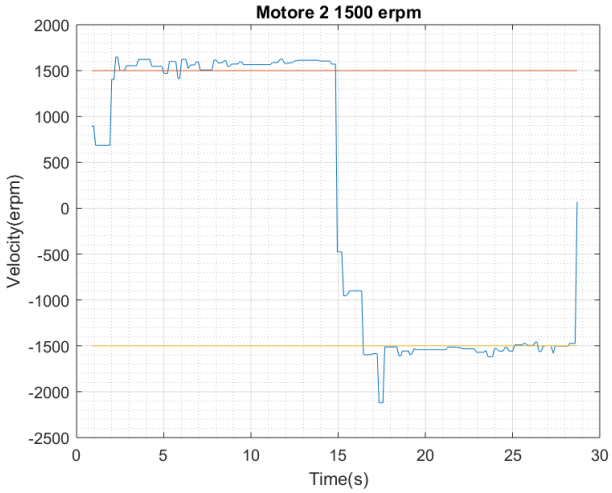

Figure 5.9. Motor 2 at 1500 erpm

To resume, motor 2 as a threshold of correct behaviour at about 950-1000 erpm. However, the errors are greater than the ones at motor 1. The explanation is in the effect of the gravity, which is not compensated with any kind of control and affects the transmission and the PID implemented on the VESC.

5.1.3 Motor 3 test: starting configuration

Motor 3 is in the lucky case where the motor moved for all the speed of the test. Therefore, the data and graphs collected are more.
Starting from 200 erpm, where the user observes a strong tremble, the error is unacceptable and the noise far too great. In fact, it seems that at low velocity the motor 3, to which the constant command of going at 200 erpm is sent, behaves in a random way around 200 erpm, but with an enormous variance, see figure 5.10

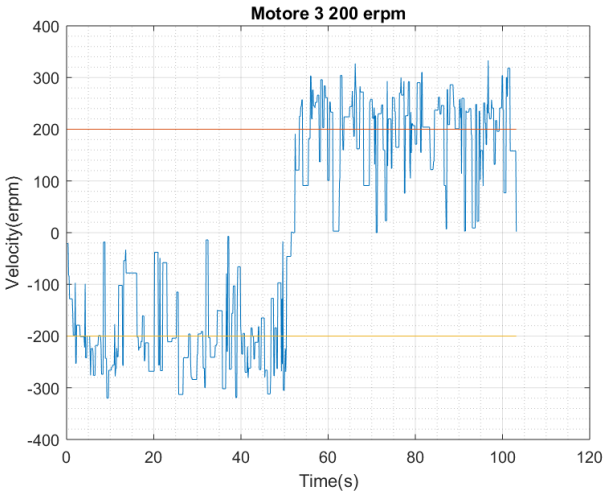

Figure 5.10. Motor 3 at 200 erpm

At 400 erpm, the situation improves, but not by much. An error of about 50% is still observed in figure 5.11 , with a lot of noise. The user also sees a strong tremble, too.

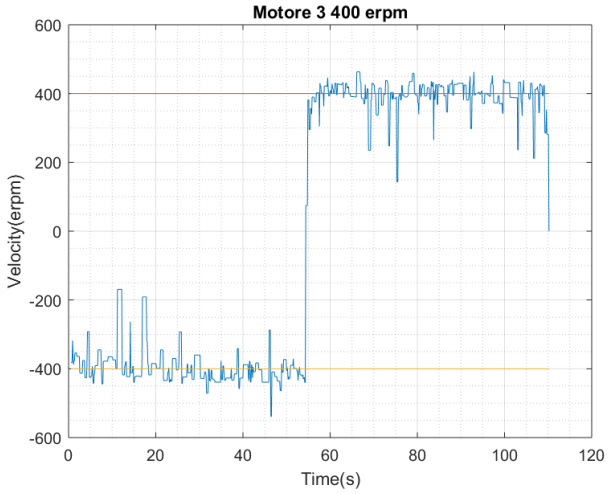

Figure 5.11. Motor 3 at 400 erpm

At 600 erpm, the situation defintely improved, with a tremble far less intense, and a maximum error of about 20 erpm, which is around 3.3%, an acceptable value. So the threshold seems to be around 600 erpm for motor 3, see figure 5.12

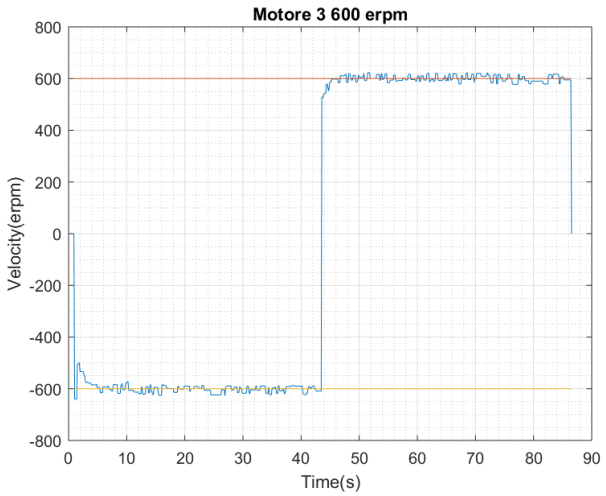

Figure 5.12. Motor 3 at 600 erpm

At 800 erpm, when the user doesn't observe anymore tremble, the maximum error falls down to 2.5%, confirming the trend that the higher the velocity, the better the motor behaves, as shown in figure 5.13

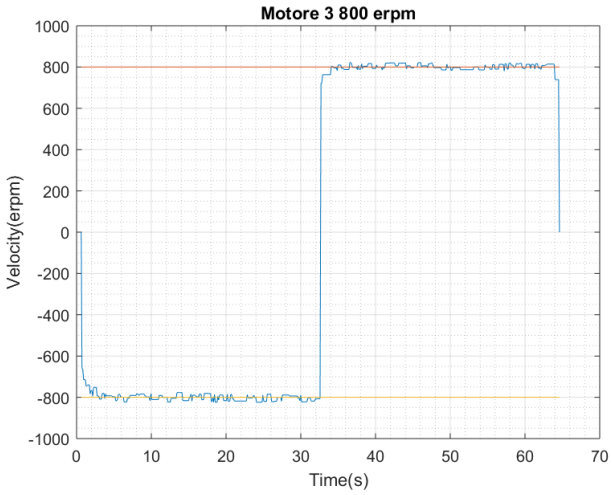

Figure 5.13. Motor 3 at 800 erpm

At 1000 erpm, the error falls to 2%, with a perfect behaviour observed by the user, and reflected in figure **??**

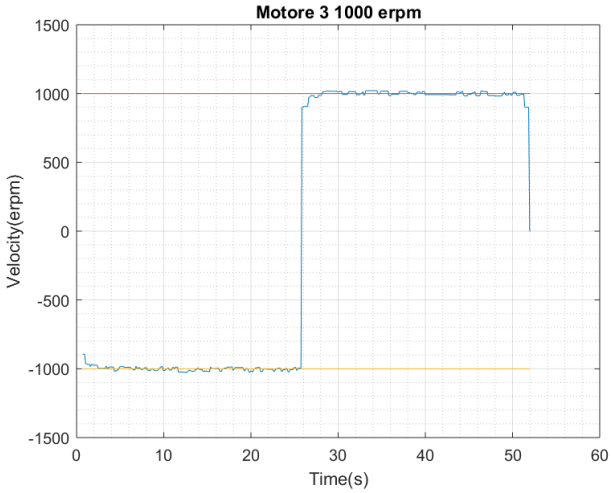

Figure 5.14. Motor 3 at 1000 erpm

So, to resume, motor 3 has a threshold of working at about 600 erpm, which is lower than the one of motor 1 and motor 2. However, using simply the starting configuration was not enough. Therefore, other tests were done with motor 1 and motor 2 in the bent configuration, with motor 3 at 180 degrees. In this test, done of course only for motor 1 and motor 2, some interesting results were observed.

5.1.4 Motor 1 test: bent configuration

The test for motor 1 in the bent configuration is interesting, because if in the starting configuration the motor never start to move until 600 erpm, here at 400 erpm the first complete test loop was observed. However, as it is possible to see below in figure 5.15, the tremble was very high and the velocity definitely not accurate nor acceptable for validation.

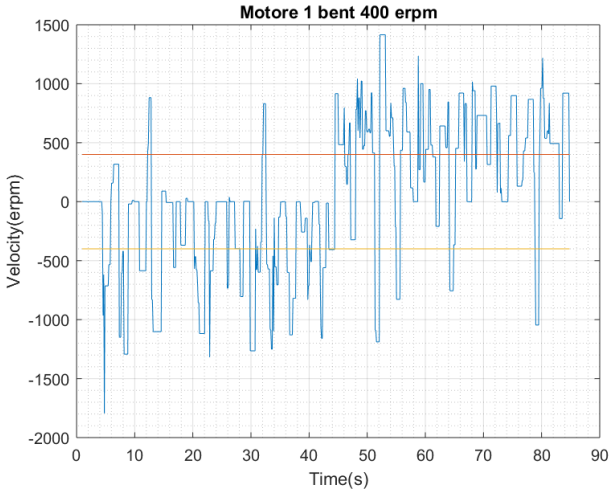

Figure 5.15. Motor 1 at 400 erpm

The situation slightly improved at 600 erpm, but still was not acceptable, with error greater than 100%, see figure 5.16. A strong tremble was observed, too, even in this configuration. This is not shocking, since the threshold for motor 1 was in the starting configuration of about 800-900 erpm. In this case the robot starts to moving before 600 erpm, at 400 erpm. However, the situation is not better, at least for the first two velocities observed.

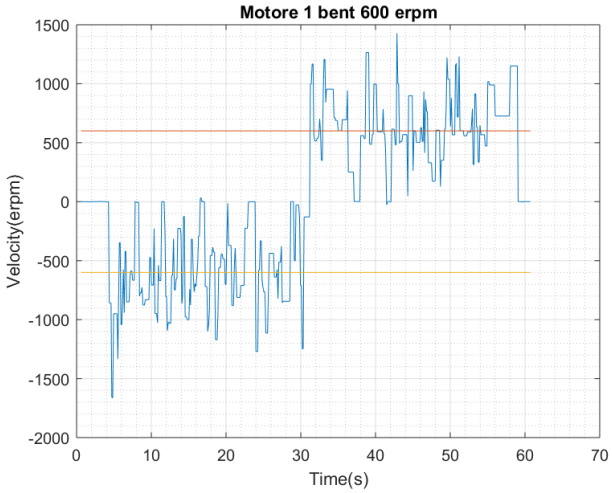

Figure 5.16. Motor 1 at 600 erpm

The situation at 800 erpm is curious. At the same exact velocity in the previous test the robot was behaving well only in one direction. Here the same happens, but the direction is reverted, as shown in figure 5.17. A strong tremble is observed for a part of the movement only.

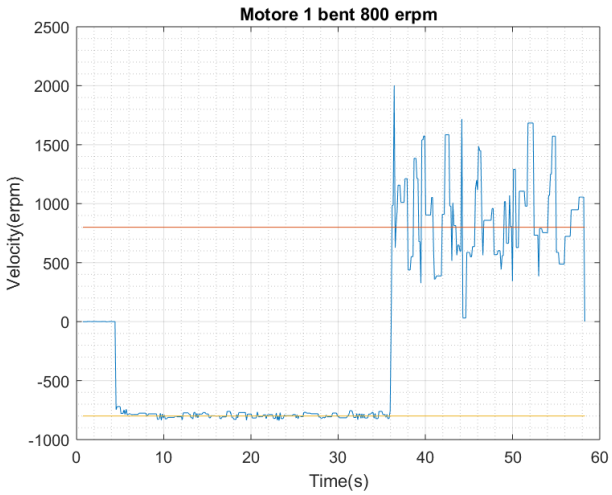

Figure 5.17. Motor 1 at 800 erpm

Therefore, it is possible to say that the situation for motor 1 definitely didn't improve with the change of position, but stayed the same.

The threshold is probably around 900 erpm, like in the first configuration test. This is proved when testing at 1000 erpm, where a smooth movement is observed, and the error in the data is of about 3%, a value similar to the previous test, see figure 5.18

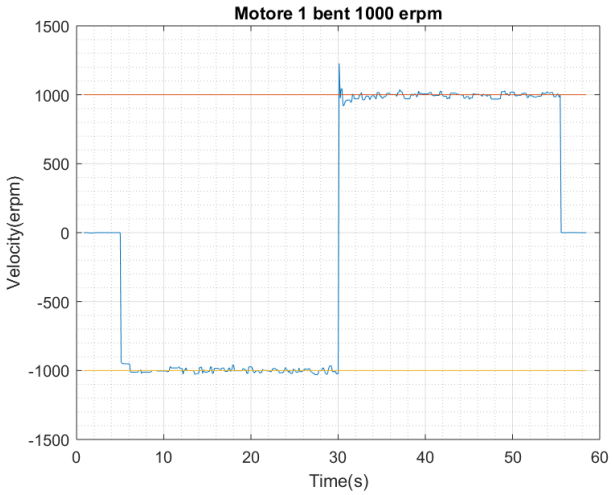

Figure 5.18. Motor 1 at 1000 erpm

At 1200 erpm, with the surprise of no one, the movement is smooth and linear, and the error falls to about 2.5%, confirming the trend of a better quality movement the faster the robot is, as shown in 5.19.

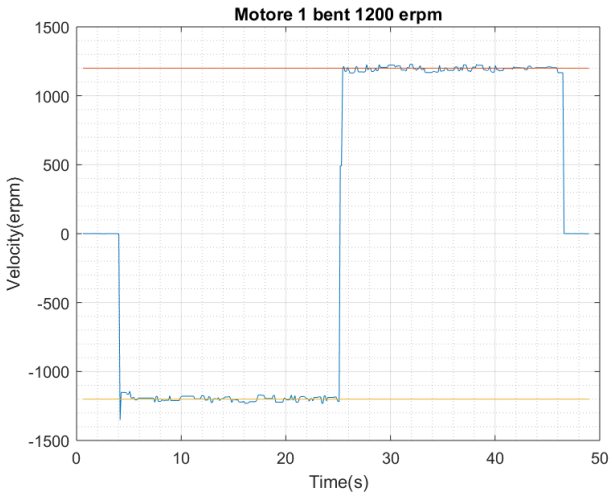

Figure 5.19. Motor 1 at 1200 erpm

Finally, at 1500 erpm the trend continues, with a very smooth movement and an error from transmitted velocity to received one of about 1%, definitely acceptable, see 5.20.

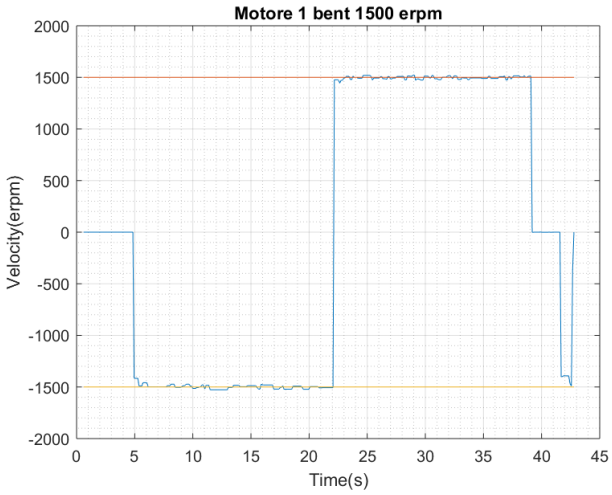

Figure 5.20. Motor 1 at 1500 erpm

To resume, motor 1 didn't change much its characteristics when the position was different.

5.1.5 Motor 2 test: bent configuration

For what concerns motor 2, the first test done at the starting configuration effectively work from 800 erpm upwards. Before 800 erpm, the robot simply didn't move. However, in this position, the first complete test loop was observed for 400 erpm. However, at 400 erpm, as observed in the figure 5.21, the robot was going up and down and drifting a lot. The observed errors are enormous.

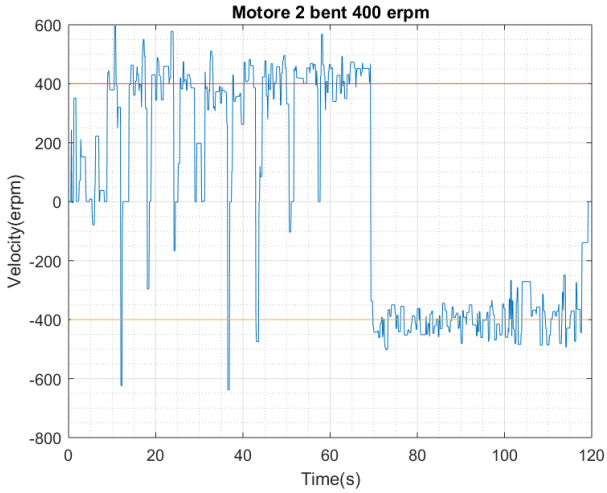

Figure 5.21. Motor 2 at 400 erpm

The situation at 600 erpm was better, but still not acceptable, with a strong tremble and errors over 25%. However, the situation is way better than the one presented in the starting configuration, see figure 5.22

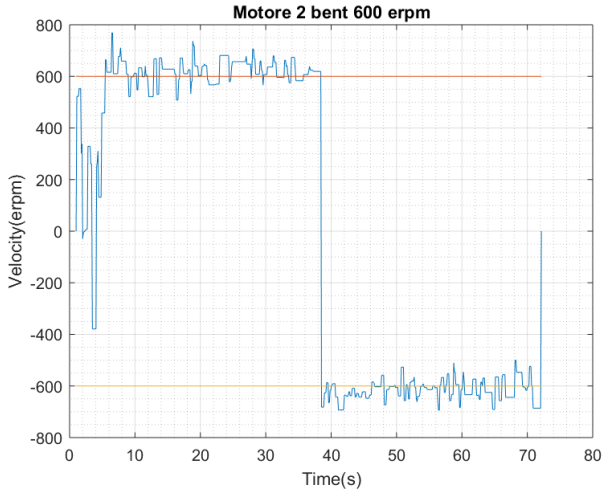

Figure 5.22. Motor 2 at 600 erpm

At 800 erpm, the tremble is little, and the error falls to about 10%, still a great value, but better than the previous test, as shown in 5.23. Therefore, here the threshold is probably at about 850 erpm.

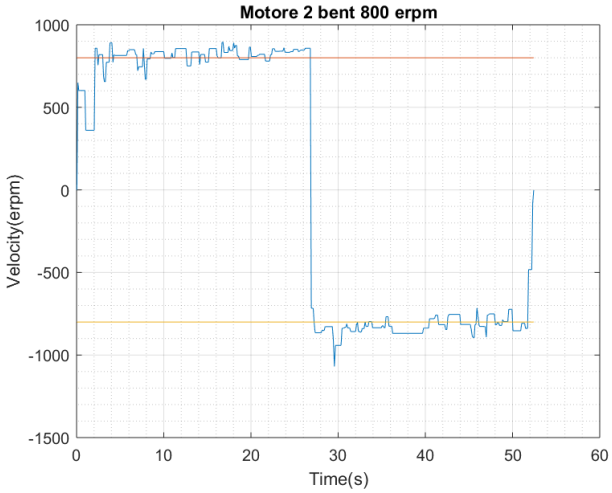

Figure 5.23. Motor 2 at 800 erpm

At 1000 erpm, no tremble is observed, the movement is smooth, but errors of the order of 6-7% are still present in the figure 5.24, although not observable in real time by the user

At 1200 erpm, the situation doesn't change much, with errors still

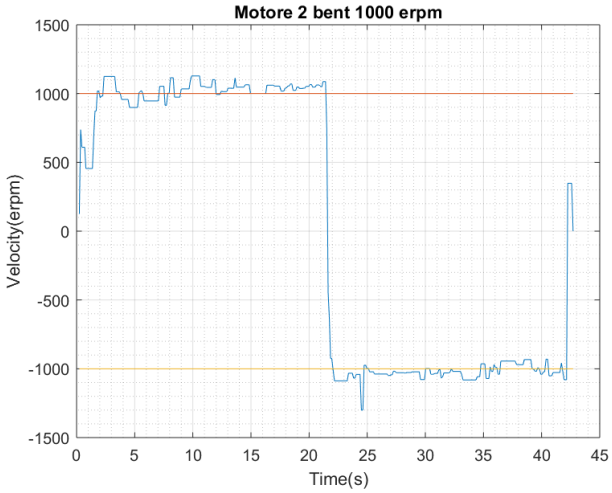

Figure 5.24. Motor 2 at 1000 erpm

close to the same order of percentage, and a movement smooth, see figure 5.25. The errors are a little greater than in the case with the motor in starting configuration, but the user can't notice that.

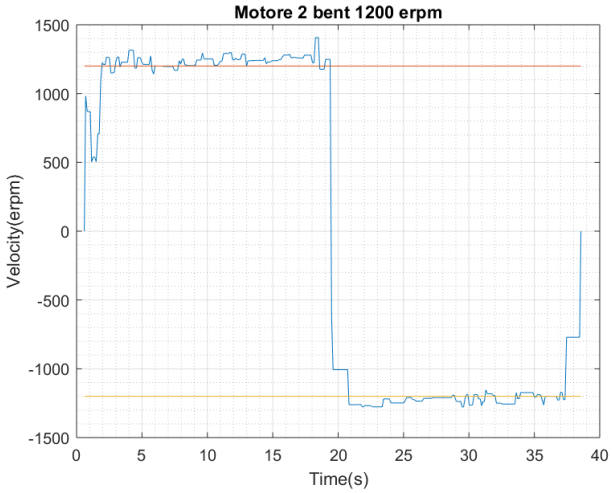

Figure 5.25. Motor 2 at 1200 erpm

The same can be told also for the test at 1500 erpm, see figure 5.26. So to resume, the threshold is a little lower, and the motor starts moving before 800 erpm this time, but the situation has not greatly improved in this case.

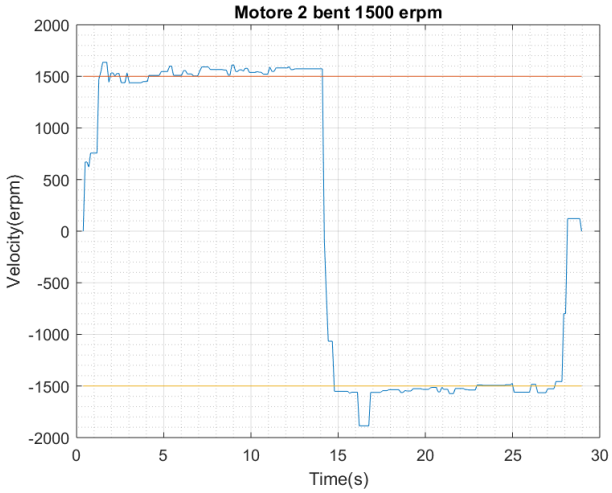

Figure 5.26. Motor 2 at 1500 erpm

5.2 Motor timing tests

Along with the velocity test, in the data collected from the motors there was also the timing, chosen from 10 to 70 degrees, in order to avoid some errors in the initial and final positions. The goal was to understand if the 3 motors behave the same way, and also in which region the timing became linear. On the x-axis, the velocity of the motor, on the y axis, the time in seconds to execute the route from 10 to 70 degrees at each of these velocities.

The first test on motor 1 presents the following graph in figure 5.27 and 5.28, with an highly nonlinear region. The motors starts to behave in a linear-like way only after the data collected at 1000 erpm, in line with what was observed in the previous tests.

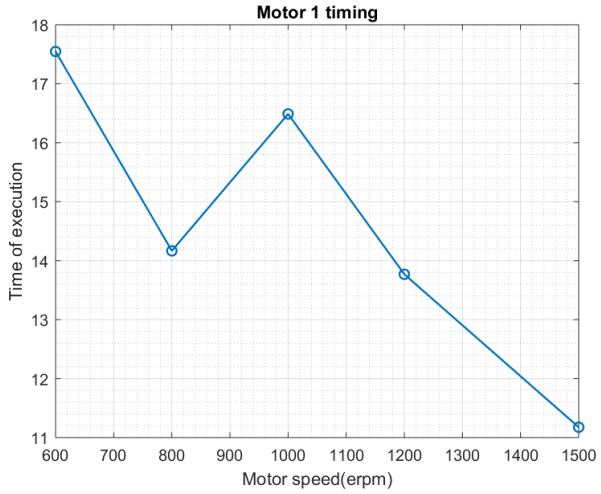

Figure 5.27. Motor 1 timing

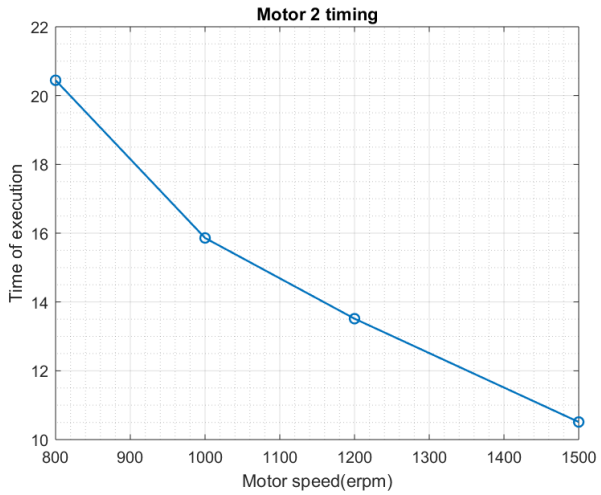

Figure 5.28. Motor 2 timing

Motor 3 instead has more points, therefore it is possible to observe three region. From 200 to 600 erpm there is high non linearity, from 1000 to 1500 the curve is linear, and in the middle the curve is somewhat linear. The fact that 600 erpm timing is the double of 1200 erpm one is perfectly right, see figure 5.29

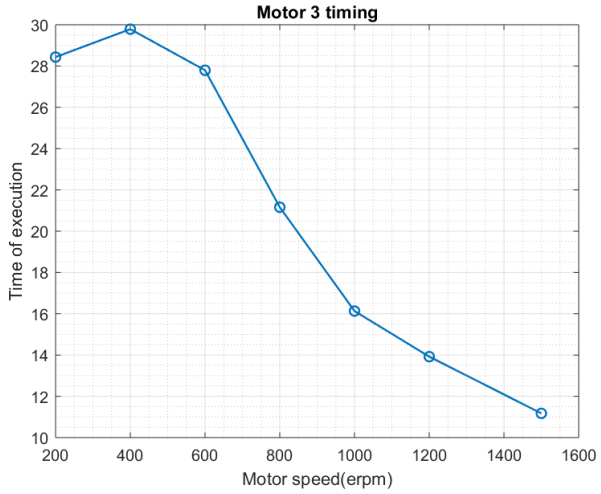

Figure 5.29. Motor 3 timing

After having observed the three motors timing in detail, the decision was to plot against each other, to see if major differences were observed. Comparing the three timing in figure 5.30, the behaviour is quite similar in the region where all the data are present, however motor 2 is a little faster than motor 1 and 3.

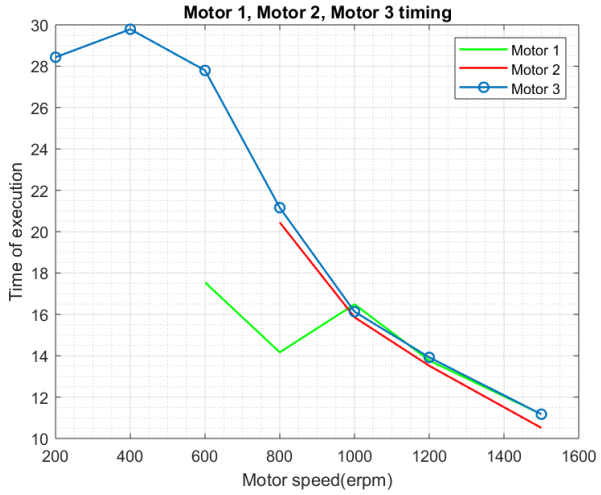

Figure 5.30. Motor 1,2,3 timing comparison

The next step was to understand if a bent configuration had a more

linear region or if the behaviour was similar to the starting one. Therefore, the graph of the timing of motor 1 in the new position follows in figure 5.31. Even with more data, the linear region doesn't start until 800 erpm. This is easy to see because at 600 erpm the time is of about 18 seconds, at 1200 of about 12.5 seconds, way higher than a half.

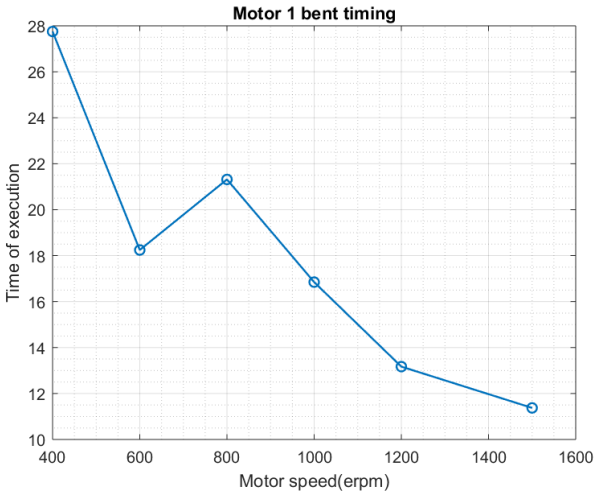

Figure 5.31. Motor 1 bent configuration timing

By comparing the starting configuration with the normal one, the next graph shown in figure 5.32 tells that the linear region for motor 1 in the bent configuration starts before the linear region in the starting configuration.

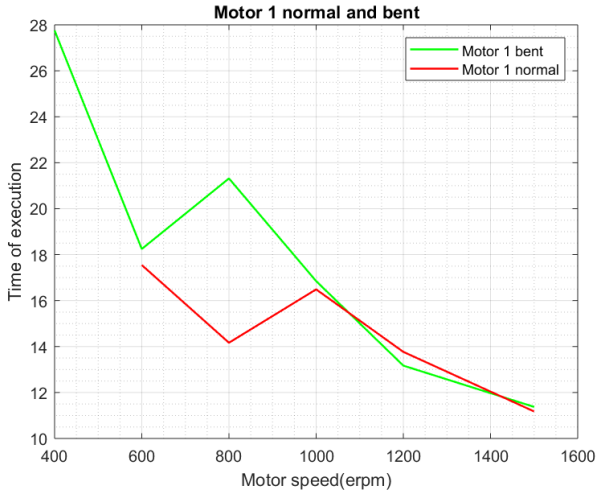

Figure 5.32. Motor 1 bent and normal configuration comparison

The same was done for motor 2, in which the bent configuration led to more data points. Here the linear region in figure 5.33 seems to start from 600 erpm, with a time of about 27 seconds, while at 1200 erpm the time is of about 13.5 seconds. However it has to be noticed that the motor 2 was shaking a bit at 600 erpm.

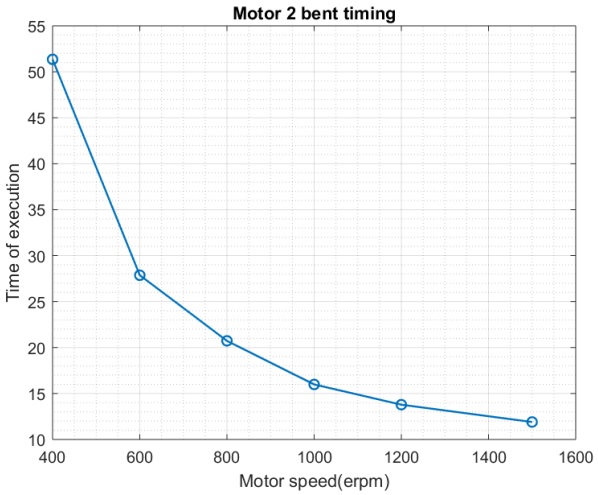

Figure 5.33. Motor 2 bent configuration timing

The comparison of the two motor 2 configurations gave almost an identical graph, with the normal configuration just a little faster than the bent one, see figure 5.34. This behaviour was also observed when the three motors were compared all together. Maybe in that measure at 1500 erpm of motor 2 a little noise was present, or maybe the motor behaves just a little faster

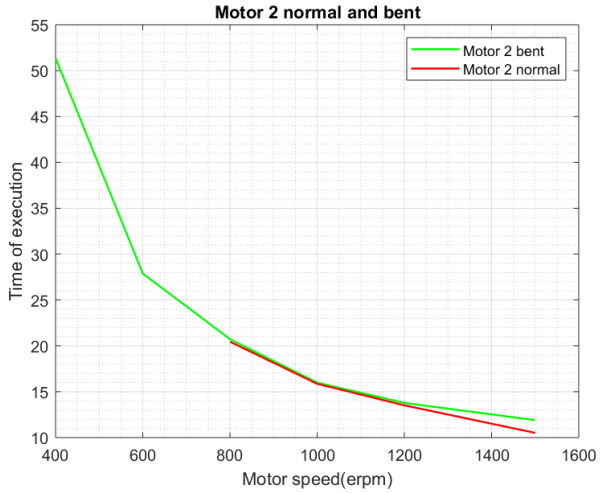

Figure 5.34. Motor 2 bent and normal configuration comparison

To conclude the experiment, the two motors in the bent configuration were compared and plotted in 5.35. The linear region of motor 2 is larger, with motor 1 that has a minor change in the slope after 1200 erpm. Therefore, motor 2 behaves better than motor 1.

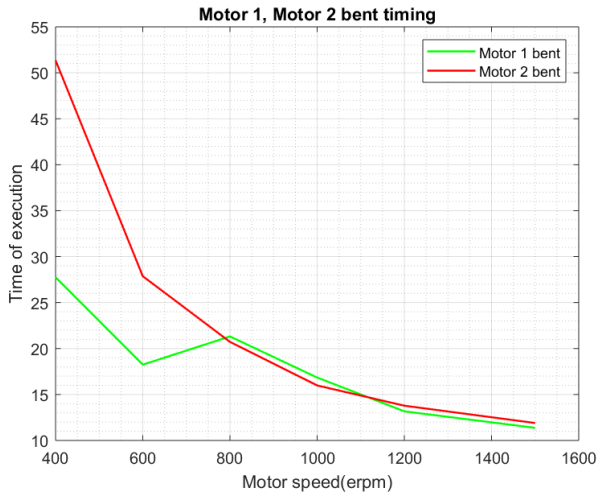

Figure 5.35. Motor 1,2 bent configuration comparison

5.3 Optitrack tests

The final part of the project was validating the motion of the robot according to an accurate measurement tool. Optitrack is a tool, that thanks to a camera and to some markers, can determine the position of a body in the space, and from that evaluate position and rotation of a another body with respect to the reference one. Every body is determined through the markers, seen by the camera, and all the visual data are elaborated thanks to a workstation. In this case, the three motors were marked, as well as the end effector, with the base of the robot marked as well as a reference. Test were tried for the control in the joint space, for the control in the operational space, and even for the target reaching control.

5.3.1 The control in the joint space

The control in the joint space was the first to be evaluated, with test for 500,750,1000 and 1500 erpm, going clockwise for almost 70 degree before coming back. The result of the test are visible in figure 5.36. In this graph, in yellow there is the command velocity sent to each motor, in rpm. In red there is the actual velocity measured from the motors. As it is possible to see, the discrete behaviour doesn't follow the command of the velocity sent until 0.59 rpm, which is roughly the command sent at 1000 erpm. Therefore, the nonlinearity region observed in figure 5.27 is confirmed also by Optitrack

Similar test were also ran for motor 2 and motor 3. However, the results from Optitrack were not reliable, giving a movement of just 15 and 25 degrees, respectively. The reason for the error is still to be clarified, but the pattern in the graphs was similar to the motor 1 pattern, therefore it is safe to say that this kind of control works as soon as the velocity is above a certain threshold.

Since the control in the joint space cannot actuate properly the velocity sent to the motors when this velocities are low, it follows as a cascade that the control in the operational space won't work at low velocities, leading to unexpected behaviours

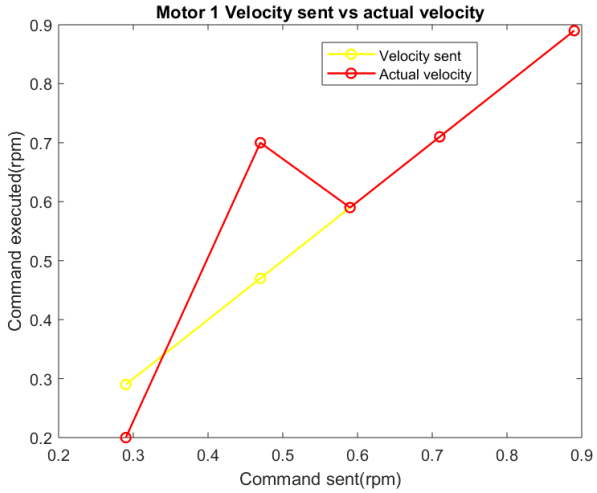

Figure 5.36. Motor 1 Optitrack test

5.3.2 The control in the operational space

For what concerns the control in the operational space, test were tried for the movement in x,y,z, with three test for each of the motors. In figure 5.37 one of the movement test along Y is displayed. A similar movement along x is observed, although this was not so evident from the visual feedback.

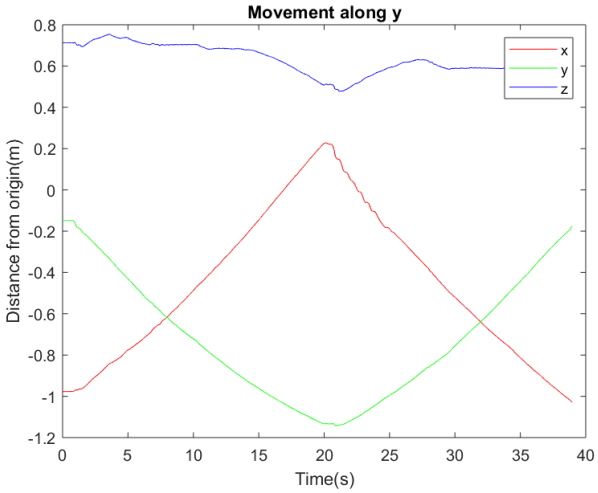

Figure 5.37. Movement along Y test

Similar results were also observed from test along X and Z, however, the accuracy of Optitrack might be discussed, since the double movement along two of the three axis was not always observed by the user. The robot is not just moving along x,y,z, but is not so evident the movement along the other cartesian coordinates as displayed in figure 5.37

A couple of the test had also decent results, as in figure 5.38. This is a test movement along the z axis, and no significant drifts are observed along the other two axis. However, in most of the tests, the scenario was the one reported in figure 5.37

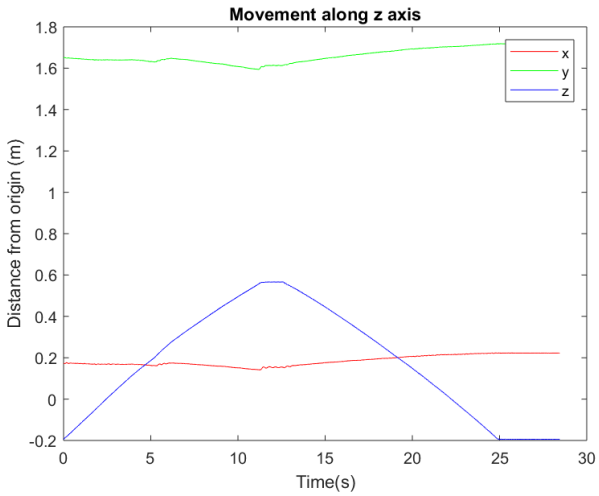

Figure 5.38. Movement along Z test

5.3.3 Target reaching control

For what concerns target reachong control, several test were tried, but the position and the point to reach, along with the route of the robot often were sometime out of the camera angle. Anyway, the robot never reached the target properly. A reliable test was done, with the results in figure 5.39.

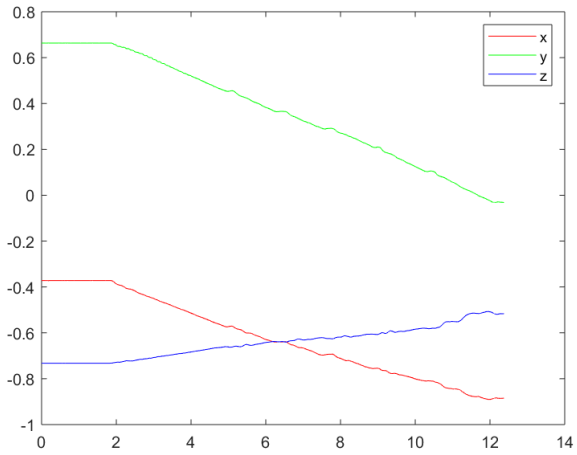

Figure 5.39. Target reaching the point (-1,0.2,0.5)

Here the target point to be reached in the space was the point with coordinates (-1,0.2,-0.5). At the end of the process the resulting point reached by the end effector has an error in x of 12 cms, in y of 17 cms, in z of 1 cm only. Therefore this kind of control is still not reliable and needs further development and tuning. The reason for the lack of reliability is on the control structure. This kind of control uses the control in the operational space in the loop. Therefore, being the control in the operational space not completely accurate, that reflects in the target reaching control. Once the control in the operational space will be reliable, for sure the target reaching control will improve

Chapter 6

Further developments and conclusion

The POPUP inflatable robotic arm can now be controlled in three different modes, instead of the two present at the beginning. The first mode is the joint space control, now fully working and validated. The tuning work of moving the command to the arrows was a success, avoiding the tremble in the movement and giving a steady parameter to the the motors. Now the only issue is at very lowe velocities, that are at the beginning of the range of the motors and don't act very well due to communication problems. And in fact communication problems also affect, in this case in a significant way, the second control, the control in the operational space thanks to the jacobian inverse kinematics. This kind of control is working well when the three motors have each a velocity higher than 800 erpm, otherwise is only qualitatively correct, in the sense that the actuation of the low velocity is incorrect. Therefore, the effect is that the robot moves along the desired axis, but also drifting along another one. The analysis of the whole communication structure lead to a drastical improvement. However, this is still not enough to fully validate this control. For the third type of control, the target reaching control further developments are needed in order to have a better behaviour. But being the target reaching control based on the operational space control, in the sense that the operational space control is in the loop of the target reaching control, for sure an improvement of the latter will lead to an improvement of the former.
The goal of this thesis project was to improve and validate the two controls already present and drafted, which were not working. However,

fixing the operational space control was by far a bigger challenge than expected. Every single component of the structure has been analyzed in order to understand where the problem actually was. And after the identification of the problem the control performance improved significantly, but it still needs further validation.
Meanwhile, a preliminary step in the introduction of the target reaching control has been done. This will lead in the future to a new step in the process of development of the POPUP robot.
The next steps are trying to validate the jacobian operational space control. To do that, it will be necessary to full understand the correct size of the buffer in the ROS node handler, and probably the communication with the PC needs to be done through an Ethernet cable, instead of a USB one.
It has to be discussed if the motors, which don't work at low velocity, should be in a range of movement starting from an higher value with respect to zero. But then all the proportion in the control is at risk of being lost, leading to a wrong behaviour once again. Therefore, a remodulation analysis of the command will be a challenge necessary to face in order to arrive at a more reliable result.
Finally, when the control in the target reaching will be done, further developments will include the presence of a visual servoing control as a real time feedback to reach the target

Bibliography

[1] Yongchang Zhang, Pengchun Li, Jiale Quan, Longqiu Li, Guangyu Zhang, and Dekai Zhou. Progress, challenges, and prospects of soft robotics for space applications. *Advanced Intelligent Systems*, 2022.

[2] Garcia Mark. Remote manipulator system (canadarm2). www.canadarm2.int [Online; Page Last Updated: Oct 24, 2018].

[3] Pierpaolo Palmieri, Matteo Gaidano, Andrea Ruggeri, Laura Salamina, Mario Troise, and Stefano Mauro. An inflatable robotic assistant for onboard applications, 2021.

[4] João Oliveira, Afonso Ferreira, and João C.P. Reis. Design and experiments on an inflatable link robot with a built-in vision sensor. *Mechatronics*, 2020.

[5] Luigi Villani Giuseppe Oriolo Bruno Siciliano, Lorenzo Sciavicco. *Robotic Modelling, Planning and Control*. 2009.

[6] Josie Hughes, Utku Culha, Fabio Giardina, Fabian Guenther, Andre Rosendo, and Fumiya Iida. Soft manipulators and grippers: A review. *Frontiers in Robotics and AI*, 2016.

[7] Laura Salamina Mario Troise Pierpaolo Palmieri, Matteo Gaidano and Stefano Mauro. Design of a lightweight and deployable soft robotic arm for aerospace applications. *International Conference on Intelligent Robots and Systems (IROS*, February 2022.

[8] Jonathan Cacace Joseph Lentini. *Mastering ROS for robotics programming*. Packt Publishing, 2021.

[9] CAN FD. Can fd — Wikipedia, the free encyclopedia, 2010. [Online; accessed 29-August-2023].