



POLITECNICO
MILANO 1863

**SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE**

EXECUTIVE SUMMARY OF THE THESIS

Realization and control of a 3D printed anthropomorphic Robotic Arm

LAUREA MAGISTRALE IN MECHANICAL ENGINEERING - INGEGNERIA MECCANICA

Authors: LORENZO FERRARI, LUCA BERTON

Advisor: PROF. HERMES GIBERTI

Co-advisors: PROF. MARCO CARNEVALE, ING. FEDERICO INSERO

Academic year: 2020-2021

1. Introduction

In this thesis work an anthropomorphic 3D printed robotic arm actuated by 3D printed strain wave gears (also known as harmonic drives) is designed. A software is generated for providing the virtual model of the robot, tools for motion planning are used in order to design a pick and place task and a dynamic simulation environment is prepared. This product is addressed to non-industrial applications where limited payload and low precision are sufficient. We identified the need of such a product in educational, recreative and small business fields. Hence an affordable solution is designed. Moreover, this product can be possibly brought into the “collaborative” robotics field, unlocking a deeper interaction between man and machine remaining in the non-industrial field. In fact, real Cobots have very high price tags, and this preclude their use in didactic field. As presented in [4], in Italy there is a misalignment between the competencies requested by the Industry and the ones given from University, especially concerning soft skills. It is acknowledged that acquired hard skills of STEM disciplines are of very high level but there is a lack in adaptability and creativity in Engineering and Computer Science. Robotics can be the link connecting STEM com-

petencies to Industry4.0 in general. Hence, there is the need of a revolution in pedagogy to give substance to what is learned in school, starting from children. It is needed to train children minds to computational thinking. Introducing Robotics to the young allows to develop abilities in problem solving and in orientation in complex problems. For older students Robotics can lead to a deeper comprehension of mathematics and programming, since a practical feedback is given synchronously with the learning process.

The structure of this thesis is briefly reported in the following.

Some characteristics for the robot have been enumerated and a review of existing commercial products and open source projects was done.

Pre-sizing of the robot was done considering former tentative and related work. The starting point was the aesthetics of a previous design of a robotic arm [5], a 3D printable harmonic drive speed reducer [3] (and before [1]), and already provided stepper motors from the Italian firm R.T.A. Robot links length, maximum speed and acceleration were set in this phase. A simplified 2D model of the arm was considered.

Afterwards the actual design of the robot was done considering a 3D printing-oriented mindset. This allowed to design features that cannot be realized with traditional manufacturing

processes. Some considerations on the available tools and 3D printing were done. A preliminary theoretical assessment of the robot limits was done to have a starting point for the actual testing phase.

To actuate the robot some electronic devices were studied. A micro-controller was required to move the robot motors and hence some coding was done. Arduino MEGA was used as micro-controller and Arduino IDE coding development environment was used to write software. An initial low level control of the robot arm was assessed implementing a mock PID controller running on the Arduino MEGA itself. For the high level control with ROS the Arduino firmware was developed.

Then, an overview of different programming approaches was performed so the reader is able to understand why an offline approach, using ROS as framework to generate the software, can be a good choice to perform the high level control. Consequently, the focus was on the generation of the virtual model of the robot. Two models was realized: one which represents the real robot and the other with the addition of a gripper. Since an end effector was not developed in the actual robot, an OnRobot gripper has been added to the virtual model to program with more completeness. Motion planning was done with the MoveIt! package and, in particular, a task of pick and place was planned to move a box using the model with the gripper. Furthermore, it was also described how to generate the files to launch Gazebo (dynamic simulator) and how to control the multibody model using ROS control. Also an integration of Gazebo into MoveIt! was performed which allow to plan a movement in MoveIt! and directly make the multibody model execute it. So, a task can be tested in the dynamic simulator to verify that it is feasible before making the real robot execute it.

Finally, a critical evaluation of the proposed solution was done, together with the identification of some possible future related work.

2. Already existing products

Before starting with the review of existing products we identified some required characteristics for our robot that are reported hereafter.

- be affordable; the robot must be made of cheap materials, must have easy-to-sort

components and electronics (boards, motors, drivers, sensors).

- be safe; the robot must be made of a non hazardous material, have completely enclosed electronics, tidy cable management, smooth edges.
- be general purpose; the robot must be modular and allow the user to choose the tool to mount on the end effector, and it has to be easy to change at any moment.
- be open source; the robot control logic must rely on open source software, thus free to download, install and eventually modify, to be adapted to specific needs.
- allow young students to program simple movements of the robot exploiting a graphical interface. Instead, students with some coding experience can program more complex tasks.
- have a simulation environment. Students can program a movement and test it on the virtual model, in this way they can understand if the task is feasible for the robot and if there are some changes that can improve the quality of the movement. A simulation environment is also useful because testing a task in virtual reality before actually making the real robot execute it improve the safety.
- be aesthetically pleasant; since the robot would be a mainstream product, it would compete with other commercial solutions, and customers also consider the look of the product, especially for display applications open to public.
- have a universal grounding fixture system; the robot must be adaptable to any workbench.
- be reliable and long lasting; nothing it's more tedious than having a tool that is often unavailable due to failures.
- be cheap and easy to repair; the robot must be made of functionally independent parts, such that if one component fails, no perfectly working part has also to be changed.

These characteristics were confirmed as the review of products was proceeding.

Some similar, cheap and low precision robotic arms (3D printed and not) were analysed to gather information to design a competitive product. The closest 3D printed robotic arm is the

Niryo One by NiryoRobotics¹. It is a relatively inexpensive robot (1599€² + VAT for the kit version) platform considering the already developed coding environment. This robot has a magnetic position feedback system that exploits a magnet attached to the back end of the stepper motors and a custom PCB with the actual sensor. This solution quite refined but very expensive. Anyways, the outreach and the maximum payload are very limited, respectively 410mm and 300g.

Hence, to be competitive our robot must be bigger and/or capable of carrying a higher payload. A position feedback system is required to catch up with the NiryoRobotics manipulator. The advantage of our robot is that we propose ourselves to use harmonic drives to achieve torque multiplication. In this way we can have higher torques to the joints than a belt and pulley transmission system.

3. Robot design

3.1. Preliminary sizing

A simplified mathematical model was developed to obtain a first tentative idea of the robot outreach and of the lengths of each link. This model only considered the robot behavior in a plane. Hence, only joints J2, J3 and J5 are of interest. By the way, only a 2R kinematic configuration was considered for simplicity and so only dof associated to J2 and J3 are considered in the model, also because these are the most stressed joints, having to lift the most of the weight of the arm with considerable lever arm. Only links L1 and L2 are considered, while L3 is neglected. Link L3 anyway adds more lever arm to weight and inertial forces, and this must be considered. Therefore transport of moment was performed, placing the masses on link L3 at the end of link L2 and adding a concentrated bending moment. Since in the very first stages of the design geometry of the links was not known, masses, moments of inertia and positions of the centers of gravity of the links needed to be estimated. Links were simplified as cylinders and the moment of inertia of thick cylinders was used, placing the center of gravity in the middle of links themselves. Since the final 3D printed pieces are hollow, the den-

sity of the material used (ABS filament) was corrected with an empirical coefficient smaller than one to not overestimate too much the masses of the links and hence obtain a very small outreach. Anyway, in this first stage masses of the links were very overestimated. Usually, this is not a good engineering practice but, given the nature of the project and the numerous uncertainties on previous existing work, we reckon that being more conservative than usual is wiser.

Limit maximum dimensions were computed performing a dynamic assessment of the torques required to joints J2 and J3 to lift the arm in some very stressful movements. To do so, maximum values of speed and acceleration of the robot arm needed to be set first. To set them it is required to establish the maximum allowable speed and acceleration that the robot could have, considering the dynamic behavior of the motors. The torque vs. speed curves on the manufacturer data-sheets were used considering 24V supply. Since only a qualitative representation was given (no polynomial function provided), the curves were approximated by straight lines.

The same simplified 2R model was used to establish theoretical limits to range of motion after completion of the mechanical design phase (when the final dimensions were set) and after simulating the 3D printed parts manufacturing with the slicer software to obtain an estimation of the actual mass of the parts.

3.2. Mechanical design

The mechanical design phase started with a review of what already available. We kept the characteristic aesthetic sign of the robot arm developed in [5] having to change slightly some dimensions for a feasible manufacturing. The harmonic drive design of [3] was given as a packet of CAD files. After studying how the characteristic components of a harmonic drive were realized, some modifications were done for what concerns hardware (for the sake of a lean BOM and easier assembling), and geometries (NEMA23 motors used required new ways to assemble plastic parts). Implementation of the position feedback system was done from ground up. While the prototype was assembled, some further modifications to the geometry of the flexible spline and the circular spline were needed to mitigate a mechanical problem that arose. We want to state

¹<https://niryo.com/product/niryo-one/>

²at the time of writing this document

here that the harmonic drive design used was not tested under load before the beginning of this thesis activity. Indeed, we experienced a failing meshing when we actuated the joint J2 with only the link L1 when using the original design of the flexible spline. We noticed that the loss of the meshing was happening when the joint was overcoming the weight force while when in favor of gravity it did not happen. This suggests that when the motor lifts the link the wave generator starts to turn freely on the flexible spline causing the link to fall instead of rising. We reckon that in this situation the resistant applied torque to the speed reducer exceeded the maximum bearable meshing torque between flexible spline and circular spline using the original rubber belt as flexible spline. Therefore the aim was to increase the meshing resistance. The easiest possible solution was to increase the orthogonal force on the rigid spline by increasing the wheelbase between the bearing on the wave generator. Anyway, we planned to use this solution only if necessary since it introduces more stress on the parts. Then, finding inspiration on the layout of commercial metal harmonic drives, the flexible spine was made taller, radially more compliant and with the teeth only on the lip to further increase compliance. Moreover, we designed a circular spline having a complementary teeth profile to the flexible spline.

The position feedback system was designed using high precision 5-turn potentiometers. This was the best compromise between the need of the feedback system and a cheap solution. Potentiometers become position transducers thanks to a gearing system. A gear is mounted on the potentiometer shaft and it is actuated by a conjugate teeth profile on the link for the axial joints (J1 and J4, and therefore the solution is external) or a gear ring mounted on the harmonic drive cover for the other joints (and in this case the solution is completely internal).

Then, the links were designed considering the harmonic drives dimensions, the aesthetic requirement and the position feedback system. We needed to exceed the maximum lengths of the links obtained in the preliminary sizing phase because some geometries otherwise would have been unfeasible. That is why the assessment of the robot range of motion was needed.

The material chosen for the construction of all the structural parts of the robot is ABS. This choice was taken due to ongoing agreements between the University and the Supplier. ABS has quite good mechanical properties that can be exploited to build strong and stiff components. Anyway, when 3D printed it suffers layer delamination (cracking) and limited layer adhesion strength. For some other parts PETG was used because it has superior layer adhesion. To guarantee the highest possible mechanical resistance, direction of deposition of the layers is of paramount importance. The main load on the robot arm is bending moment, thus printing direction was set not to be the same as the axis of the arm itself. Anyway, to have the greatest strength one has to sacrifice printing quality.

3.3. Low level control

The micro-controller for actuation of the robot was selected. An Arduino MEGA board was chosen because it is the simplest, cheapest micro-controller having enough pins to control all motors and to read all signals coming from the position feedback system. Strategies for actuating the stepper motors using the provided electronics (stepper drivers boards) were analysed. An already available Arduino library was used allowing to synchronize all six stepper motors. Some tests were done to establish compatibility and accuracy of the library with the R.T.A. stepper driver board. Small errors were proven but they are not a problem thanks to the position feedback system. Sizing of the Analog to Digital Conversion (ADC) system to read position by means of the potentiometers was done. The original aim was to have an ADC that allowed to indirectly detect half a motor step. Motors are built with 200step/rev but the biggest micro-stepping that can be set on the board is 2, thus the aim is to "see" 0.9deg. For this purpose external 14 bit ADC are required, since the Arduino MEGA embedded ADC has only 10bit resolution.

A mock PID controller was developed to run directly on the Arduino board for a simple, low level interaction with the robot. It was not possible to implement a proper discrete time PID controller running on the Arduino (that is a digital device) because the library used to synchronize all the motors uses a blocking strategy: as

the motors are moving it's not possible to ask to Arduino to do anything (also reading signals from potentiometers) and therefore the clock for the controller can't work properly. "Mock" PID means that a continuous time approach was used to compute the action for the motors but approximating the time derivative of the error with a finite difference and the integral of the error with the area of a trapezoid.

In the thesis work the digital model of this robot arm was developed. Arduino code to be deployed on the micro-controller itself to let it interface with ROS was implemented. Arduino is an embedded piece of hardware of a ROS Control system, receiving control inputs from and sending Mechanism States to the ROS hardware interface [2]. The Arduino MEGA acts as a ROS node that receives the joints angles and the maximum desired speed, and sends the measured positions of each link. Communication is done through Serial communication protocol.

3.4. High level control

An overview of different control strategies was performed highlighting the pros and cons of online and offline programming approaches which are the more diffuse in industries. Hybrid programming was also briefly explained, it relies on augmented reality and can be a promising technology for the future of robotics. Benefits of using an offline programming strategy to control the robot and the choice of using ROS which is an open source framework are presented.

A software was generated with ROS to provide two virtual models: one which represents the real robot and the other with the addition of a gripper. A virtual model is obtained by generating the xacro file of the robot. In this file the meshes of each link were imported from the CAD files, each link was defined considering its relative position and the collision geometries were also determined. The latter are much simpler with respect to the actual shape of the links and, in this way, there is a reduction of computational cost in detecting collisions during the execution of a task. In the xacro file each joint was also defined: type, position, axis of rotation, limits on effort and velocity. Consequently, the xacro file is really important because it provides the kinematic chain of the robot. Furthermore, in the model with the gripper has been picked a

gripper of OnRobot. The choice of the gripper is not permanent, in the future could be modified and also substituted with a custom designed gripper. The gripper was added to address the programming phase with more completeness.

Once the virtual models were realized the MoveIt! motion planning package was generated to plan some movements. In particular, it allows to generate the motion law that the end effector should follow to perform a desired task considering just the kinematics of the robot. To configure the virtual model of our robot in MoveIt! its setup assistant interface was used. The user can choose algorithms for path planning given from the OMPL (Open Motion Planning Library) and also algorithms to solve the inverse kinematics problem which is required to obtain the joints configurations in executing a specific motion law of the tool center point. Hence, there was a description of how to plan simple tasks with a GUI and also how to program more complex task with programming languages. In fact, a pick a place task, programmed in Python language, was planned to move a box using the model with the gripper.

The virtual model of the robot was also launched in a dynamic simulation environment which is called Gazebo. Here, the multibody model reproduces the real robot in a realistic way since also its dynamic properties are considered. In fact mass and inertia properties of each link were added to the xacro file with also the transmissions related to each joint and a plugin was added to allow the interaction between ROS and Gazebo. There is also an explanation of how to control the multibody model with ROS control in which the type of controllers and the respective gains were defined and how to send joints target positions by means of ROS topics. Finally, an integration of Gazebo in MoveIt! was performed. In this way the user can plan a task in MoveIt! and directly make the multibody model execute it. Then, it is explained how to get the torques which actuate each joint during the execution of a planned movement and so it is verified that these values respect the limit effort defined in the xacro file of the robot model.

4. Conclusions

Looking to the list of requirements given in the beginning, several points have been satisfied.

A cost estimation considering self-sourcing components was done to compare the designed robot arm to the identified competitors on the market. Our robot is cheaper than the the closest competitor, still having a position feedback system, bigger and stronger . Anyway at this stage our robot lacks of a development environment that would make easier and at high level the use of the product. The end-effector and tools were not designed so it's not clear at this stage a specific task for the robot.

The robot has smooth and rounded edges and speed reducers are not exposed. Cable management has been cured as much as possible considering that the speed reducers are not hollow shaft. Cables are routed externally and pass through the arm links to go to a main harness. An enclosure for all the electronics was not addressed since the robot is at a very early prototyping phase. Anyway, a 3D printed base for electronics was designed to tidy and separated high from low voltages.

Being fully 3D printable, speed reducers included, potentially makes this robot accessible to a very big community of Universities and Makers because the BOM has very few parts and production can be fully done locally. No proprietary hardware, electronic board or software was used to control the robot.

The original design of the harmonic drive was not capable of resisting to very high resistant torque. We tested several combinations of parts and for the most stressed joints J2 and J3 the best one is using the circular spline with complementary HTD3M teeth profile and flexible spline made of hard plastic and in particular PETG plastic was used for its superior layer adhesion strength compared to ABS. The circular spline with complementary teeth profile also reduces backlash; moreover, due to the shear stresses a plastic having high inter-layer adhesion strength such as PETG is needed. The wave generator wheelbase was increased of 0.80mm to increase the meshing resistance. The obtained modified harmonic drive has big backlash. This caused to abandon the idea of detecting half motor step. Moreover, the quality of the 3D printed gearing system was not high enough and a play between gears mounted on the potentiometer shaft and the gears on the links or harmonic drive covers exists, nullifying the desired resolution. For

these reasons, the embedded Arduino ADC was used.

A low level control of the robot was achieved: the user can directly send target joints positions on Arduino MEGA on the Arduino IDE. The high level control was also addressed. The Arduino firmware for interaction with ROS is ready. The virtual model of the robot and tools for motion planning are completed. The MoveIt! GUI allows to plan simple motion and also more complex tasks can be programmed by means of coding languages. In particular, an example task of pick and place to move a box is available.

For the moment an Hardware Interface which allow the interaction between ROS control and the physical hardware is missing. Once available, it will be possible to plan a task in MoveIt! but also to make the real robot execute it.

The files to launch the multibody model in the dynamic simulation environment (Gazebo) and the packages to control it are prepared. Also the integration of Gazebo into MoveIt! is performed and allow to plan a task in MoveIt! and test it in Gazebo before making the robot execute it. In this way the user can understand if the robot is able to perform the task and if some refinements must be carried out.

A original look was given to the robot. Thanks to 3D printing, customization of the robot aesthetic is very simple to adapt the arm to tastes and blend its design to the one of other products. The robot base can be mounted on any table top. It can be possible to mount the arm on a Automated Guided Vehicle for a mobile robot application.

The flexible spline of the harmonic drive realized with hard PETG plastic showed poor reliability, being prone to fracture due to the high shear stresses, especially when mounted in the most stressed joint, the robot shoulder.

The modular design proven to be effective in easing disassembling and assembling for maintenance and repair.

References

- [1] Ilgaz Askin. Design of a 3d-printed harmonic drive for low-cost modular robot. Master's thesis, Politecnico di Milano Scuola di Ingegneria Industriale e dell'Informazione, 2018/2019.

- [2] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtké, and Enrique Fernández Perdomo. `ros_control`: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2017.
- [3] Marco Rotulo. Ridimensionamento di un riduttore armonico per un robot stampato in 3d. 2020.
- [4] UniversalRobots. Carta delle idee della robotica collaborativa 2021. robotica collaborativa, quali sfide per l'education, September 2021.
- [5] Siyu Wu. Euclid design and optimization of a 3d printed robotic arm. Master's thesis, Politecnico di Milano School of Design, 2019.



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Realization and control of a 3D printed anthropomorphic Robotic Arm

TESI DI LAUREA MAGISTRALE IN
MECHANICAL ENGINEERING - INGEGNERIA MECCANICA

Authors: **Lorenzo Ferrari, Luca Berton**

Student IDs: 945716, 945661

Advisor: Prof. Hermes Giberti

Co-advisors: Prof. Marco Carnevale, Ing. Federico Insero

Academic Year: 2020-21

Ringraziamenti

Ringrazio il Professor Hermes Giberti, il Professor Marco Carnevale e in particolare l'Ingegnere Federico Inero per la loro disponibilità e per avermi aiutato durante questo percorso.

Ringrazio i miei colleghi Damiano Bargna, Federico Celeri, Davide Bonsignore con i quali ho condiviso questi anni universitari e che hanno reso più leggeri i momenti difficili.

Desidero inoltre ringraziare la mia famiglia e i miei amici per avermi sempre supportato nel corso degli anni e per avermi accompagnato a questo importante traguardo.

Luca Berton

Contents

Ringraziamenti	i
Contents	iii
List of Figures	vii
List of Tables	xi
List of Listings	xiii
Nomenclature	xv
Abstract	xvii
Sommario	xix
1 Introduction	1
2 State of the Art	5
2.1 Arduino Tinkerkit Braccio Robot	5
2.2 Niryo One	7
2.3 Niryo Ned	10
2.4 BCN3D Moveo	14
2.5 Identified upgrades for our Robot	16
3 Robot design	19
3.1 Preliminary Sizing	19
3.1.1 Schematization and Nomenclature	19
3.1.2 Model of the robot arm	20
3.1.3 Definition of links lengths	21
3.1.4 Estimation of Inertia	22

3.1.5	Dynamic assessment	23
3.2	Mechanical design	30
3.2.1	Harmonic Drives	31
3.2.2	Selection of rotation transducers	40
3.2.3	HD modifications after testing	42
3.2.4	Robot arm links	48
3.2.5	Wiring management	55
3.2.6	Modularity and Photo Gallery	56
3.3	Manufacturing with 3D printing	62
3.3.1	Machines and Slicer Software	62
3.3.2	Material selection	64
3.3.3	Printing orientation	66
3.3.4	3D printing the flexible spline	67
3.3.5	Sacrificial features	67
3.3.6	Solutions to tools limits	67
3.4	Theoretical limits of the robot arm	69
4	Electronics and Control	73
4.1	Electronics	73
4.1.1	Stepper driver description and characteristics	73
4.1.2	Other electronics	74
4.1.3	Selection of the microcontroller	74
4.1.4	Analog to digital converter (ADC) sizing	78
4.1.5	Potentiometer wiring	79
4.2	Control	79
4.2.1	Strategies for actuating the stepper motors	79
4.2.2	Preliminary testings	80
4.2.3	Rotation sensing	87
4.2.4	PID control	89
4.2.5	ROS interface	94
5	Programming approaches	101
5.1	Online programming	102
5.1.1	Pros and Cons	104
5.2	Offline programming	105
5.2.1	General overview	105
5.2.2	Pros and Cons	109
5.3	Hybrid programming (Augmented Reality)	110

5.4	ROS-based offline programming	112
6	ROS-based offline programming	115
6.1	Robot model with gripper	116
6.1.1	Visualization in RViz	116
6.1.2	Manipulator model	119
6.1.3	OnRobot gripper model	121
6.2	Robot model of real robot	123
6.3	MoveIt! motion planning	124
6.3.1	MoveIt! Setup Assistant	125
6.3.2	MoveIt! GUI	129
6.3.3	Pick and place application	130
6.4	Gazebo modelling	141
6.4.1	Robot model with gripper	141
6.4.2	Robot model of real robot	146
6.5	Dynamic simulation integrating Gazebo into MoveIt!	149
7	Results and Conclusions	153
7.1	Results	153
7.1.1	Cost estimation	153
7.1.2	Limits to range of motion	157
7.1.3	Arduino library concerns	162
7.1.4	Structural concerns	163
7.2	Conclusions	163
	Bibliography	167
A	Codes	171
A.1	Rotation assessment	171
A.2	Rotation speed assessment	173
A.3	Mock Arduino PID controller	174
A.4	Arduino Firmware for ROS	182
A.5	Codes for the digital model of the robot with gripper	189
A.6	MoveIt! configuration files for robot model with gripper	200
A.7	Code to launch the virtual model with gripper in MoveIt!	204
A.8	Codes to launch the virtual model with gripper in Gazebo	206

A.9	Codes to control the virtual model with gripper in Gazebo	207
A.10	Code to plan the pick and place task in MoveIt!	209
A.11	Codes for the digital model of the real robot	218
A.12	MoveIt! configuration files for robot model of the real robot	225
A.13	Code to launch the virtual model of the real robot in MoveIt!	228
A.14	Codes to launch the virtual model of the real robot in Gazebo	230
A.15	Codes to launch the virtual model of the real robot in both MoveIt! and Gazebo	232

B Drawings 235

B.1	Harmonic Drive Parts	235
B.2	Robot Parts	236

List of Figures

1.1	Prusa MK3S 3D printer	3
2.1	Arduino Tinkerkit Braccio Robot	6
2.2	Niryo One	7
2.3	Assembling detail of the Niryo One shoulder	9
2.4	Niryo One sensored stepper motor	10
2.5	Niryo Ned internal structure	12
2.6	Niryo Ned Workspace	13
2.7	NEMA17 stepper motor with planetary gear box	14
2.8	BCN3D Moveo	15
2.9	BNC3D Moveo end effector close up	15
3.1	Simplification of links geometry	23
3.2	Torque vs. Speed characteristic of J2 motor	24
3.3	Torque vs. Speed characteristic of J3 motor	24
3.4	Maximum acceleration test	25
3.5	Maximum speed test for J2	27
3.6	Maximum speed test for J3	27
3.7	Motion laws	28
3.8	Torque profiles	29
3.9	Traditional HD layout. 1: input shaft. 2: wave generator. 3: flexible spline. 4: circular spline. 5: output shaft. 6: housing	32
3.10	Captive nut orientations	33
3.11	NEMA17 and NEMA23 base flanges	35
3.12	NEMA17 and NEMA23 wave generators	36
3.13	Flexible spline	37
3.14	Modified output flange	37
3.15	Gears for non axial joints	38
3.16	HD cover for axial joints	39
3.17	Exploded view of a NEMA23 harmonic drive	39

3.18	Rotary magnetic encoders	40
3.19	Bourns 3548 potentiometer	42
3.20	Commercial harmonic drive parts	44
3.21	Modified flexible spline3.20	45
3.22	Wave generator with 4 bearings	46
3.23	3D printable elliptical bearing	47
3.24	Robot base	49
3.25	Axial coupler	50
3.26	Link 0	51
3.27	Split Link 0	52
3.28	J2 potentiometer mounting system	53
3.29	Assembled link 1	54
3.30	Module 1	57
3.31	Module 2	58
3.32	Module 3	59
3.33	Module 4	60
3.34	Real robot assembled	61
3.35	Warping and delamination mechanism	65
3.36	Load direction and layers	66
3.37	Base with CAD modeled support	68
3.38	90deg swing starting from 0deg torque curves	71
3.39	Symmetric swing up form 4th to 1st quadrant	71
3.40	Limit β angle	72
4.1	BSD 02.V connections	74
4.2	ADS1115	79
4.3	Potentiometer pinout	79
4.4	Rotation angle assessment trends	85
4.5	Error vs. Cumulated error	85
4.6	ROS Control overview	95
5.1	Teach pendant	103
5.2	Lead through method	104
5.3	Simulating model of a 5 dof robot	106
5.4	Microsoft Hololens	111
5.5	AR: miniature airplane washing robot	111
6.1	Block diagram to control a robot	116

6.2	Robot model	117
6.3	rqt graph in Rviz	117
6.4	Robot model - name of components and frames	118
6.5	Collisions geometry	120
6.6	OnRobot gripper	122
6.7	Real robot model	123
6.8	MoveIt! system architecture	125
6.9	manipulator group	127
6.10	RRT example	128
6.11	RRTstar example	129
6.12	MoveIt! GUI	130
6.13	Pick and place	138
6.14	Pick	139
6.15	Homing	140
6.16	Place	140
6.17	Return home	141
6.18	ROS Control + Gazebo	143
6.19	Gazebo simulation	145
6.20	rqt	146
6.21	joint trajectory controller	146
6.22	Real robot model in Gazebo	149
6.23	Initial pose	150
6.24	Final pose	151
6.25	Torque plot	151
7.1	Broken flexible spline	160

List of Tables

2.1	Arduino Tinkerkit Braccio Robot specification	7
2.2	Niryo One specifications	8
2.3	Niryo One joints limits	8
2.4	Niryo Ned specifications	11
2.5	Niryo Ned speed specifications	11
3.1	Gearing for non axial joints	41
3.2	Gearing for axial joints	41
3.3	3ntr A4v4 characteristics	62
3.4	Prusa MK3S+ characteristics	63
3.5	ABS properties	64
3.6	PET characteristics	65
3.7	Masses estimated with Slicer software	69
4.1	BSD 02.V characteristics	73
4.2	Arduino UNO characteristics	75
4.3	Arduino MEGA characteristics	76
4.4	Raspberry Pi Pico characteristics	77
4.5	Teensy 3.6 characteristics	77
4.6	Potentiometer wiring	79
4.7	Speed assessment results	86
6.1	Joints name - Frame position of each joint - Axis of rotation	118
6.2	Lower and upper bounds of each joint	123
6.3	Limit effort of each joint	124
6.4	Initial pose information of a simple movement	150
6.5	Final pose information of a simple movement	151
7.1	Cost per category	153
7.2	Motor prices	154
7.3	Cost for bearings	156

7.4	Results of tests on Joint 2 using original wave generator wheelbase. Comparison between different combinations of flexible spline and circular spline.	158
7.5	Results of tests on Joint 2 using the PETG modified flexible spline with HTD3M teeth. Comparison between original and modified circular splines increasing wave generator wheelbase.	159
7.6	Results of tests on Joint 2 using the circular spline with HTD3M complementary teeth. Comparison between different combinations of flexible splines and increased wave generator wheelbase.	161

List of Listings

4.1	Interrupting square wave	80
4.2	Non interrupting square wave	80
4.3	Definition of a stepper driver object	81
4.4	Definition of two synchronized stepper drivers	82
4.5	Definition of speed profile	83
4.6	Potentiometer setup procedure	88
4.7	Example of PID control of a square wave period	90
4.8	Initial position measure	93
4.9	Control action computation	93
4.10	Updating position measure and variables	94
4.11	Definition of callback function for actuation	96
4.12	Definition of the Subscriber	97
4.13	Definition of callback function for setting speed	97
4.14	Definition of Publisher for position feedback	97
4.15	Firmware setup	98
4.16	Homing function	99
4.17	Firmware loop	100
A.1	Rotation assessment	171
A.2	Speed assessment	173
A.3	Mock Arduino PID controller	174
A.4	Arduino Firmware for ROS	182
A.5	lucabot.xacro	189
A.6	2fgt_urdf_simplified_collision.xacro	195
A.7	joint_limits.yaml	200
A.8	kinematics.yaml	201
A.9	lucabot.srdf	201
A.10	demo.launch	204
A.11	lucabot.launch	206
A.12	lucabot.world	207
A.13	robot_controllers.yaml	207

A.14 gripper_gains.yaml	208
A.15 pick_and_place.py	209
A.16 lucabot_moveit_gazebo.xacro	218
A.17 lucabot.gazebo.xacro	224
A.18 joint_limits_real_robot.yaml	225
A.19 lucabot_real_robot.srdf	226
A.20 ros_controllers.yaml	227
A.21 demo_real_robot.launch	228
A.22 gazebo.launch	231
A.23 ros_controllers.launch	231
A.24 demo_gazebo.launch	232

Nomenclature

ABS Acrylonitrile Butadiene Styrene

ADC Analog to Digital Converter

AR Augmented Reality

ASA Acrylonitrile Styrene Acrylate

BOM Bill Of Materials

CAD Computer Aided Design

DOF Degrees Of Freedom

DXF Drawing Exchange Format

GUI Graphical User Interface

HD Harmonic Drive

HMD Head Mounted Display

IK Inverse Kinematics

OMPL Open Motion Planning Library

OS Operating System

PCB Printed Circuit Board

PETG PolyEthylene Terephthalate Glycol-modified

PID Proportional Integrative Derivative

PLA Polylactic Acid

RAMPS RepRap Arduino Mega Pololu Shield

ROS Robot Operating System

RRT Rapidly-Exploring Random Tree

RViz ROS Visualization

SCARA Selective Compliance Assembly Robot Arm

SEA Series Elastic Actuator

SRDF Semantic Robot Description Format

STEM Science Technology Engineering Mathematics

STEP STandard for Exchange of Product model data

STL STereo Lithography

TPU Thermoplastic Poly Urethane

URDF Unified Robot Description Format

VAT Value Added Tax

XML Extensible Markup Language

Abstract

In this thesis work an anthropomorphic 3D printed robotic arm is designed, a software is generated for providing the virtual model of the robot, tools for motion planning are used in order to design a pick and place task and a dynamic simulation environment is prepared.

This product is addressed to non-industrial applications where limited payload and low precision are sufficient. Some field of application can be the educational one for learning robotics on the field, interactive display applications such as in museums and in shop windows. Hence an affordable solution was designed. Moreover, this product can be possibly brought into the "collaborative" robotics field, unlocking a deeper interaction between man and machine remaining in the non-industrial field. In fact, real Cobots have very high price tags, and this preclude their use in didactic field.

Some characteristics for the robot have been enumerated and a review of existing commercial products and open source projects is done.

Pre-sizing of the robot is done considering former tentative and related work. The starting point is the aesthetics of a previous design of a robotic arm, a strain wave gear (also known as harmonic drive) speed reducer and already provided stepper motors from the Italian firm R.T.A. Robot links length, maximum speed and acceleration are set in this phase. A simplified 2D model of the arm was considered.

Afterwards the actual design of the robot is done considering a 3D printing-oriented mindset. This allowed to design features that cannot be realized with traditional manufacturing processes. Some considerations on the available tools and 3D printing are done.

A preliminary theoretical assessment of the robot limits is done to have a starting point for the actual testing phase.

To actuate the robot some electronic devices are studied. A micro-controller is required to move the robot motors and hence programming of the micro-controller is needed. Arduino MEGA is used as micro-controller and Arduino IDE coding development environment is used. An initial low level control of the arm is assessed implementing a mock PID

controller running on the Arduino MEGA itself. For the high level control with ROS the Arduino firmware is developed.

Then, an overview of different programming approaches is performed so the reader is able to understand why an offline approach, using ROS as framework to generate the software, can be a good choice to perform the high level control. Consequently, the focus is on the generation of the virtual model of the robot. Two models have been realized: one which represents the real robot and the other with the addition of a gripper. Since an end effector was not developed in the actual robot, an OnRobot gripper is added to the virtual model to program with more completeness. Motion planning is done with the MoveIt! package and, in particular, a task of pick and place is planned to move a box using the model with the gripper. Furthermore, it is also described how to generate the files to launch Gazebo (dynamic simulator) and how to control the multibody model using ROS control. Also an integration of Gazebo into MoveIt! is performed which allow to plan a movement in MoveIt! and directly make the multibody model execute it. So, a task can be tested in the dynamic simulator to verify that it is feasible before making the real robot execute it.

Finally, a critical evaluation of the proposed solution is done, together with the identification of some possible future related work.

Keywords: 6 axes manipulator; harmonic drive; strain wave gear; additive manufacturing; ROS; MoveIt!; ROS control; Gazebo

Sommario

In questa tesi è stato sviluppato un braccio robotico a sei assi che viene costruito mediante stampa 3D a filamento. Inoltre, si espone come generare un software per fornire il modello virtuale del manipolatore, come pianificare i movimenti in modo tale da progettare una task di pick and place e come preparare un ambiente di simulazione dinamica.

Questo prodotto è destinato ad applicazioni non industriali dove sono sufficienti payload limitati e bassa precisione. Alcuni contesti in cui questo robot può trovare utilizzo sono l'istruzione, per poter insegnare ed apprendere la robotica sul campo, oppure in musei per mostre interattive e nei negozi per esporre dinamicamente in vetrina dei prodotti. Perciò, è stato necessario sviluppare una soluzione relativamente economica. Questo braccio robotico può essere con alcuni accorgimenti convertito in un robot "collaborativo", per una più profonda interazione tra l'operatore e la macchina, sempre rimanendo nel campo non industriale, dove esistono i veri Cobot che però, dato l'elevato costo, precludono in molti casi il loro utilizzo in ambiente didattico.

Le caratteristiche desiderabili per questo robot sono state delineate e una panoramica su prodotti commerciali e progetti open source già esistenti è stata fatta per inquadrare meglio il robot.

Il predimensionamento del robot è stato condotto tenendo in considerazione le precedenti esperienze e progetti correlati. Il punto di partenza è l'estetica di un progetto precedente di un robot a sei assi, un riduttore armonico e i motori passo-passo gentilmente forniti dall'impresa italiana R.T.A. Le lunghezze dei bracci del robot, velocità e accelerazione massime sono state definite. In questa fase è stato utilizzato un modello bidimensionale semplificato del braccio.

Successivamente le parti del robot sono state modellate al CAD tenendo un approccio orientato alla stampa 3D. In questo modo sono state realizzate geometrie che le convenzionali tecniche di produzione non sono in grado di realizzare. Sono state fatte alcune considerazioni sugli strumenti a disposizione e sulla stampa 3D in generale.

Una verifica teorica dei limiti del braccio robotico è stata fatta come punto di partenza

per la fase sperimentale di test del robot.

L'azionamento del robot richiede l'utilizzo di alcuni componenti elettronici, tra cui un microcontrollore che richiede di essere programmato; dunque, del codice è stato scritto per permettere al microcontrollore di azionare le schede dei motori passo-passo. Il microcontrollore scelto è un Arduino MEGA che è stato programmato usando l'ambiente di sviluppo Arduino IDE. Inizialmente, un controllo di basso livello è stato sviluppato implementando una sorta di controllo di tipo PID. Successivamente, è stato sviluppato il firmware per l'interazione con ROS e da caricare su Arduino.

In seguito, è presente una panoramica dei vari approcci di programmazione. Il lettore è così in grado di capire perchè un approccio offline, utilizzando ROS come struttura per generare il software, può essere una buona scelta per eseguire il controllo ad alto livello. Di conseguenza, viene generato il modello virtuale del robot. Sono stati realizzati due modelli virtuali: uno rappresenta il robot fisico mentre l'altro ha in aggiunta un gripper. Siccome un gripper non è stato sviluppato sul robot fisico, un gripper di OnRobot è stato aggiunto al modello virtuale per programmare in modo più completo. La pianificazione dei movimenti è realizzata con il pacchetto MoveIt!, in particolare una task di pick and place viene pianificata per muovere una scatola utilizzando il modello dotato di gripper. Inoltre, viene descritto come generare i file per eseguire Gazebo (simulatore dinamico) e come controllare il modello multibody usando ROS control. Viene effettuata anche l'integrazione di Gazebo in MoveIt! che permette di pianificare movimenti in MoveIt! e farli eseguire direttamente dal modello multibody. Così, una task può essere testata nel simulatore dinamico per verificare che sia realizzabile prima di farla eseguire dal robot fisico.

In conclusione, sono state fatte alcune considerazioni critiche sulla soluzione ottenuta. Alcuni possibili aspetti da sviluppare in futuro sono stati individuati.

Parole chiave: braccio robotico a 6 assi; riduttore armonico; additive manufacturing; ROS; MoveIt!; ROS control; Gazebo

1 | Introduction

Learning Robotics in school can be beneficial for innovation. Industry can develop faster if the new generations start working with a new mindset. As presented in UniversalRobots [20], in Italy there is a misalignment between the competencies requested by the Industry and the ones given from University, especially concerning soft skills. It is acknowledged that acquired hard skills of STEM disciplines are of very high level but there is a lack in adaptability and creativity in Engineering and Computer Science. Robotics can be the link connecting STEM competencies to Industry4.0 in general.

Hence, there is the need of a revolution in pedagogy to give substance to what is learned in school, starting from children. It is needed to train children minds to computational thinking. Introducing Robotics to the young allows to develop abilities in problem solving and in orientation in complex problems. For older students Robotics can lead to a deeper comprehension of mathematics and programming, since a practical feedback is given synchronously with the learning process.

Among all robotic platforms, 6 dof robotic arms are very versatile since they mimic a human arm.

Industrial solutions are very application-oriented and the very high capital cost required to buy them, the knowledge and skills one must have prevent the use of these products in non-industrial applications. Indeed, industry-grade products are overkilled for education and entertainment purposes. Hence, the need of a cheap, simple, and low precision robot appears in the education field. One possible application of such a robotic arm is introduction to robotics for high school and universities students with practical sessions on programming movements and trajectories in both joints and work spaces. Moreover, it can be used to practically explain the actual construction of a motor and speed reducer apparatus. In fact, showing how the everyday things are made helps learning. Moreover, challenging young minds to face problems that apparently are simple, but then show an higher level of complexity then expected, helps to develop new solutions.

Another possible application is interactive display of goods in shops. The customer can ask the robot to manipulate an item to show all sides without the risk of damaging it. The

same concept can be applied to museums. It would be possible to interact at a distance with items in their cases. In all these applications the payload is limited and there's no need of high precision and repeatability of the movements.

The aim of this work is to develop a low-cost 6 dof robotic arm that can be addressed to teachers, researchers, students and whoever finds a spot for such a product. Such solution must:

- be affordable; the robot must be made of cheap materials, must have easy-to-sort components and electronics (boards, motors, drivers, sensors).
- be safe; the robot must be made of a non hazardous material, have completely enclosed electronics, tidy cable management, smooth edges.
- be general purpose; the robot must be modular and allow the user to choose the tool to mount on the end effector, and it has to be easy to change at any moment.
- be open source; the robot control logic must rely on open source software, thus free to download, install and eventually modify, to be adapted to specific needs.
- allow young students to program simple movements of the robot exploiting a graphical interface. Instead, students with some coding experience can program more complex tasks.
- have a simulation environment. Students can program a movement and test it on the virtual model, in this way they can understand if the task is feasible for the robot and if there are some changes that can improve the quality of the movement. A simulation environment is also useful because testing a task in virtual reality before actually making the real robot execute it improve the safety.
- be aesthetically pleasant; since the robot would be a mainstream product, it would compete with other commercial solutions, and customers also consider the look of the product, especially for display applications open to public.
- have a universal grounding fixture system; the robot must be adaptable to any workbench.
- be reliable and long lasting; nothing it's more tedious than having a tool that is often unavailable due to failures.
- be cheap and easy to repair; the robot must be made of functionally independent parts, such that if one component fails, no perfectly working part has also to be changed.

Considering what is above, 3D printing is a feasible way to realize structural components of a low-cost 6 dof robot. Consumer 3D printers are affordable and guarantee good level of manufacturing quality.

Two aspects should be analyzed more in detail: safety and the open source world.

The world of “open source” requires as mentioned open access to software, but also free access to schematics, drawings, bills of materials etc. This means that the user can source components and build the structure on his/her own.

This option can live aside the conventional commercial strategy of manufacturing and selling a finished product or “kits” to customers. There are many firms selling 3D printers that have chosen the open source way. For example, PrusaPrinters¹ sells 3D printers in the form of already assembled products (Figure 1.1b) or as kits (Figure 1.1a), giving to customers everything they will need to assemble it, while keeping the product open source.



Figure 1.1: Prusa MK3S 3D printer

Thus, for those who own a 3D printer would be possible to print their own structural parts and then procure by themselves electronics and motors. By the way, since the bill of materials can be long, the manufacturer of the robotic arm can sell kits containing all parts needed but the 3D printed ones. For those who don't have a 3D printer at disposal and don't want to have one, kits containing everything can be the solution.

For what concerns safety, considering that no heavy-duty tasks would be ever performed on such a product, collaborative tasks become very interesting. It would be possible to interact with the robot while it is performing some task, for example during a university practical robotics laboratory. Therefore, the robot will be design considering that humans

¹<https://www.prusaprinters.org/>

may interact with it, and so important aspects that need to be thought carefully are impact detection and backdrivability.

2 | State of the Art

A brief description of the products named hereafter is done, mainly for taking inspiration for the Robot that has been developed in this thesis work. Main characteristics are collected to have an idea of the market requirements and to try to design a competitive product.

Among “DIY” products there are:

- Arduino Tinkerkit Braccio Robot.
- Niryo One.
- Niryo Ned.
- BNC3D Moveo.

The closest competitor is the Niryo One, a 3D printed robotic arm with 6 dof.

2.1. Arduino Tinkerkit Braccio Robot

This product that can be purchased directly from Arduino eStore¹, is very affordable for both academic institutions and students, costing² only 199€+VAT for the kit without an Arduino micro-controller board and 219€+VAT including also an original Arduino board, which is the control unit of the device. It's made of injection molding plastic parts and small and cheap servomotors. It is really oriented to a didactic use to make students approach to the robotics world, and it cannot be considered a product intended for business purposes. Website overview description tells that some possible tasks may be moving objects or attach a camera to the end effector.

¹<https://store.arduino.cc/tinkerkit-braccio-robot>

²at the time of writing this document

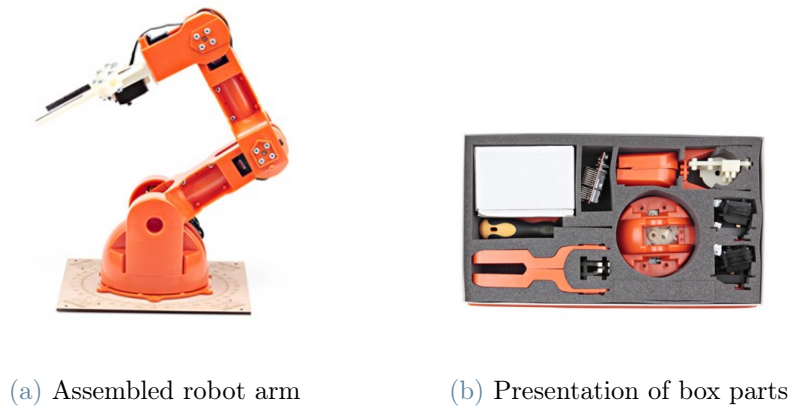


Figure 2.1: Arduino Tinkerkit Braccio Robot

This product has a short BOM composed of structural parts, drives and electronics, and general purpose hardware. Reporting the list on the website:

- Plastic Parts x 21
- Screws x 63
- Flat Washer x 16
- Hexagon Nut x 7
- Springs x 2
- Servo Motors: 2 x SR 311, 4 x SR 431
- Arduino compatible Shield x 1
- Power Supply 5V, 4A x 1
- Phillips Screwdriver x 1
- Spiral Cable Protection Wrap x 1

The Arduino compatible shield is open-source: schematics can be downloaded for free in the Documentation section of the product page on the website. This board can be stacked on an Arduino UNO board. The problem with this product is that electronics is not contained inside the robot base and wires are neither routed inside the links nor close to it. The power-supply (present on the shield board) is rated for 5V, so it is not particularly hazardous, but wires are still exposed to damage.

The technical specifications are reported in the Table 2.1.

Parameter	Value
Weight	792g
Maximum Height	52cm
Maximum Extension	80cm
Load Capacity	150g at 32cm or 400g at minimum span

Table 2.1: Arduino Tinkerkit Braccio Robot specification

Speed specification are reported in a misleading notation, thus are not reported here, but no comparison would be ever done with this very funny product!

2.2. Niryo One

This product can be purchased from the NiryoRobotics website³ and costs⁴ 1599€+VAT for the kit version and 1799€+VAT for an already assembled product. This price tag becomes quite prohibitive for most students, but universities or small business enterprises can still afford this device. This robotic arm is capable of more demanding tasks than the Arduino Braccio. It is actuated by NEMA17 servo motors through belt and pulley speed reducers. The control board is a Raspberry Pi3, thus a more powerful computing device than an Arduino board, since it can host an Operating System (OS).



(a) Front view



(b) Back panel and rear view

Figure 2.2: Niryo One

The structural parts are 3D printed in PLA plastic and there are also aluminum tubular parts that compose the links. This solution is not very ideal, as said in the introduc-

³<https://niryo.com/product/niryo-one/>

⁴at the time of writing this document

tion of this work, because makes harder for the customer to source by him/herself these components.

Technical specifications are reported in the Table 2.2.

Parameter	Value
Weight	3,2kg
Max Reach	440mm
Maximum payload	300g

Table 2.2: Niryo One specifications

No data about speed or acceleration is available on the specification sheet on the documentation section of the product page. The workspace is reported in the specification sheet as a table with joints limits (Table 2.3).

	Min	Max
J1	-175deg	+175deg
J2	-90deg	+36.7deg
J3	-80deg	+90deg
J4	-175deg	+175deg
J5	-100deg	+110deg
J6	-147.5deg	+147.5deg

Table 2.3: Niryo One joints limits

CAD (SolidWorks), STEP and STL files of structural parts and the BOM are accessible for free on the github repository⁵ of NiryoRobotics. The BOM includes also some proprietary (not commercial) boards of which schematics are not provided.

The base is clamped to the workbench by a set of suction cups: this solution is indeed simple, but not universal since a simple plywood tabletop won't hold the vacuum of the suction cups. The designers have also embedded a set of four slots for M5 screws for intense use, providing more accuracy and repeatability since the base is more steadily fixed on the workbench. Electronics is contained in the base and connectors are accessible from the back or the product. Wires routing is well managed, with very short visible sections.

Some parts of the transmission systems are exposed: in the base the pulley and the belt controlling the second joint of the arm are exposed, and this can be dangerous for an operator working very close to the robot.

⁵https://github.com/NiryoRobotics/niryo_one

An interesting solution that is possible to find on this structure is a torsional spring placed on the second joint of the arm that helps the motor to bear the weight of the links (Figure 2.3⁶).

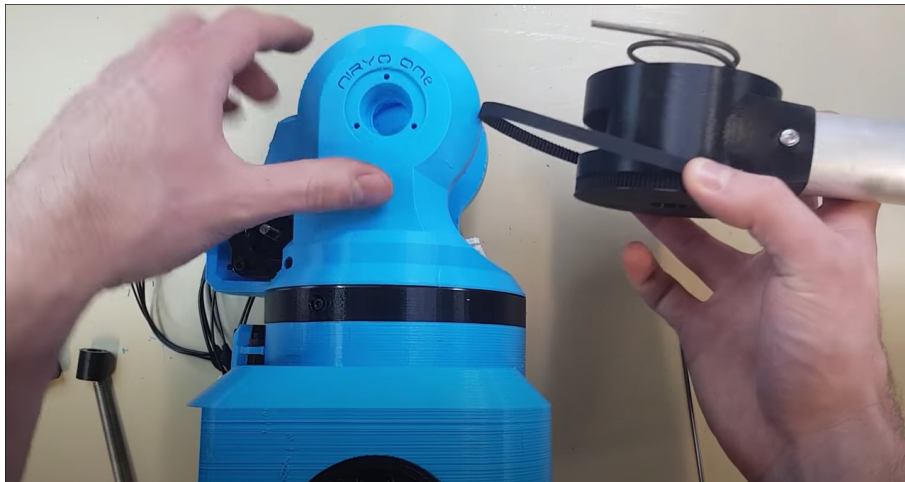


Figure 2.3: Assembling detail of the Niryo One shoulder

It's important to notice that this spring is not placed in series with the actuator, thus this solution doesn't constitute a SEA joint. The NEMA17 stepper motor actuates the shoulder link through the timing belt on a pulley providing the required reduction ratio, therefore the rotation of the arm is directly related to the motor one. Thus, this solution is not capable of detecting if an external action is blocking the link.

Looking to specifications and BOM, no trace of a position feedback system can be found. Encoders or other methods for measuring position of the link are advertised. Anyway, we reckon that the stepper motors used by NiryoRobotics are sensed using a magnetic encoder. In Figure 2.4 it is possible to see a PCB board inserted in a 3D printed housing and a magnet attached to the back end of the motor shaft.

⁶Source: Assembling video tutorial <https://www.youtube.com/watch?v=3njgLE8F2XY&list=PLZ9-LjwJrSVMAmexZBYtVxNB9Hmwtuq7H&index=4> time stamp 5:15

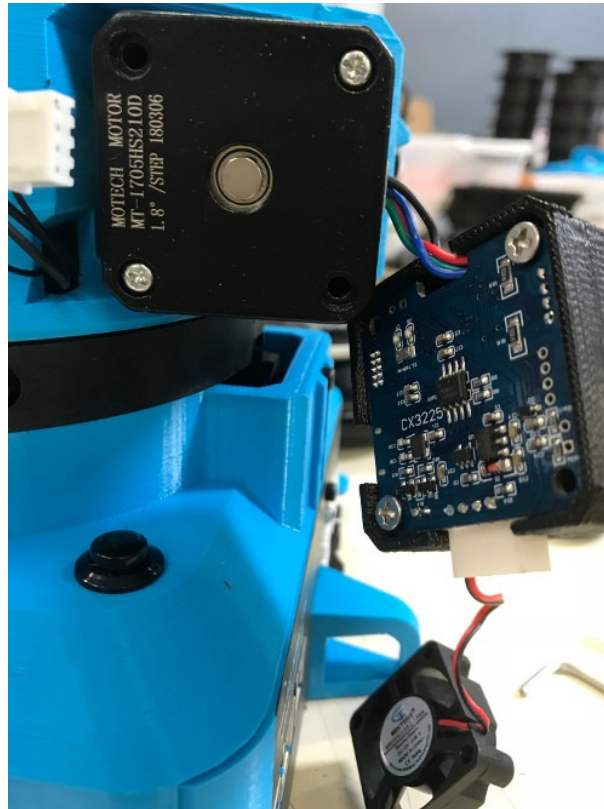


Figure 2.4: Niryo One sensed stepper motor

2.3. Niryo Ned

This robotic arm from NiryoRobotics is presented as the successor of the Niryo One. As the latter, Ned is a open-source 6 dof robotic arm that is addressed to Educational, Vocational Training and Research fields. It is presented as new collaborative robot ready for Industry 4.0 and given its open-source nature, the aim of the French firm is to democratize robotics. Indeed, the price tag of 2999€+VAT is quite competitive with respect to other industrial solutions. The increased cost compared with the One is due to many upgrades, starting from the construction.

Niryo Ned structure is made of aluminum parts machined with CNC and the tubular elements also found in the Niryo One. This time the aluminum tubes are fastened to the the aluminum machined parts, providing indeed a more rigid structure. NiryoRobotics states that the Ned is capable of fluid and smooth movements and repeatability of 0.5mm.

Looking carefully Figure 2.5, it is possible to find 3D printed elements realized with FDM technology and PLA plastic. Some 3D printed parts are cover panels to hide the internal metal structure, like the one close to the robot wrist or the one covering the pulley of the

shoulder joint. Other plastic parts seem structural like the base flange and the housing of the joints.

Actuation is achieved with NEMA stepper motors and reduction with belts and pulleys. Specifications are reported in a very detailed way. In Table 2.4 are reported the most relevant ones.

Parameter	Value
Weight	6.5kg
Reach	440mm
Payload	300g
TCP max speed	1144 ^{mm} /s
Repeatability	0.5mm
Supply voltage	12V

Table 2.4: Niryo Ned specifications

For the Ned speed limits for each joint are given and reported in Table 2.5.

Joint	Limit
J1	150 ^{deg} /s
J2	115 ^{deg} /s
J3	140 ^{deg} /s
J4	180 ^{deg} /s
J5	180 ^{deg} /s
J6	180 ^{deg} /s

Table 2.5: Niryo Ned speed specifications

A graphical representation of the robot workspace is reported in Figure 2.6.

According to the product website, this robot is equipped with a magnetic collision detection sensor on motors (not specified on which axes). This feature brings this product real close to the industrial collaborative solutions.

Programming of the robot can be done in several ways, all relying on a Raspberry Pi 4 board as computing unit. It's supported the programming environment Niryo Studio and PyNiryo developed by NiryoRobotics, but there is also the compatibility with ROS. For a total control it is also possible to program it through straight C++ or Python code.



Figure 2.5: Niryo Ned internal structure

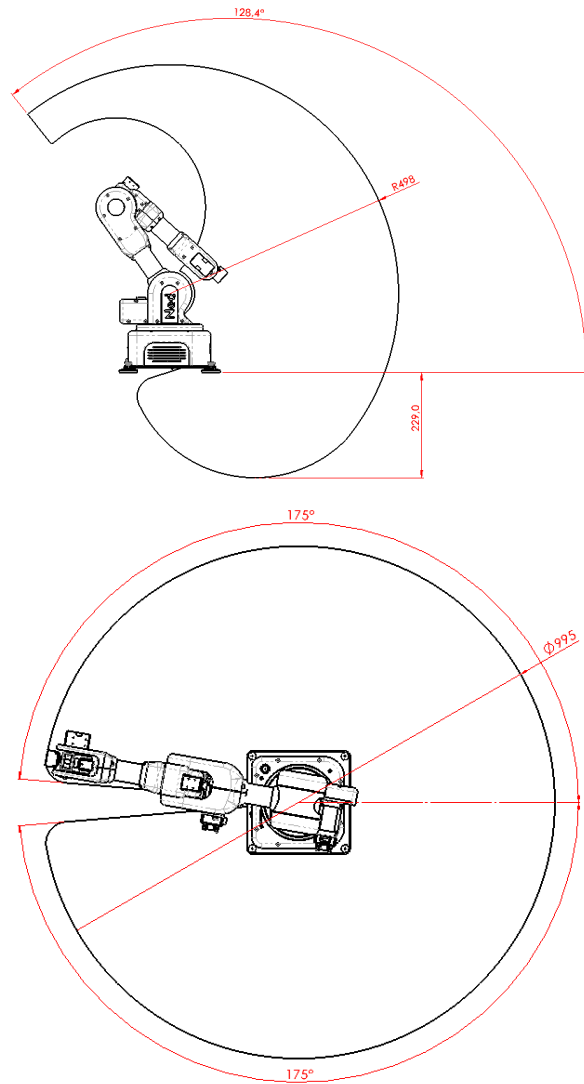


Figure 2.6: Niryo Ned Workspace

2.4. BCN3D Moveo

This robotic arm has been developed by BCN3D Technologies Inc. in collaboration with the Department d'Ensenyament from the Generalitat of Catalunya to achieve a low cost robotic arm to be used in the educational field, to overcome the high price barrier of industrial equipment.

This product cannot be purchased as BCN3D decided not to commercialize it, but being fully open source it's possible to have access to everything one may need to source components and realize the robot structure through additive manufacture. Everything from BOM to CAD files and even an assembly manual is available on the GitHub repository of BCN3D⁷.

This robot is driven by NEMA17 or NEMA23 stepper motors and speed reduction is achieved through belts and 3D printed pulleys, sometimes embedded in the robot structure. There is also listed one NEMA17 stepper motors including a planetary gearbox with reduction ratio of 5:1, like the one in Figure 2.7.



Figure 2.7: NEMA17 stepper motor with planetary gear box

The control logic is based on a Marlin firmware (available on the GitHub repository) run on an Arduino MEGA 2560 board. Moreover, a RAMPS V1.4 board is used to control all the stepper motors. Thus, electronics is shared with the world of consumer 3D printers, making those parts widely available and cheap. The control interface is the free software Pronterface, again also used with 3D printers. An official support for ROS environment is missing.

Some hardware components are also sourced from the 3D printers world, like for example smooth rods, belts and pulleys. Fasteners are general purpose metric bolts and nuts.

⁷<https://github.com/BCN3D/BCN3D-Moveo>

The gripper is very well built and BCN3D states that the design has been thought so that it's very easy to adapt the end effector to specific need.



Figure 2.8: BCN3D Moveo



Figure 2.9: BNC3D Moveo end effector close up

2.5. Identified upgrades for our Robot

As it is possible to see in the before presented products, existing 3D printed robot arms have a price tag in the range of 1500 ÷ 3000€. To be competitive and more accessible to a wider public, our robot should fall in that price range and be more capable, or be cheaper.

The described existing products have limited outreach of less than 500mm and very low payload, less than 0.5kg even for the Niryo Ned with aluminum links. The ambition for our robot is to bring the payload (to be intended as mass of gripper and carried object) up to 1kg or above if possible. We reckon that this is an issue of actuators (motors and speed reducer combination) rather than of structural resistance of the 3D printed links, and thus more torque capable motors need to be selected. Brushless motors have higher power density and so for the same output torque a more compact motor can be used. For this reason and for control possibilities (Field Oriented Control FOC) these drives would be very appealing but they are very expensive (more than 200€ for small and good quality motors), breaking the constraint of keeping low the cost of the robot. NEMA17 stepper, even geared, are not powerful enough and thus bigger steppers should be selected, like NEMA23 motors. It's true that these motors are bigger and heavier but are needed only for the most stressed joints and so as going up on the arm, small NEMA17 can still be used. Since we propose ourselves to use harmonic drives for speed reduction and torque multiplication, we can theoretically achieve higher torques at the joint than all the described products that rely on belt and pulley transmissions. Another hardware feature that is missing on all the reviewed products is a safety braking device to lock the most stressed joints when the robot is not powered or when a power outage occurs and the robot was performing a task. In this way damage to the robot itself and to the surroundings can be avoided. Selected motors from R.T.A. for shoulder and elbow joints are equipped with a braking device.

Arduino Thinkerkit Braccio is a very simple product that lacks of a position feedback system to control in position the robot joints. In fact this is evident from its price tag. We think that this feature is of paramount importance for learning control theory and implement reliable pick and place applications, also just for displaying objects. The closest competitor, the Niryo One has a magnetic position feedback system mounted on the rear of each stepper motor. For the BNC3D Moveo a position feedback system was not identified instead and therefore it can be controlled only in openloop. Therefore, a position feedback system needs to be developed keeping in mind that the products must be affordable, and hence a cheap but reliable solution is needed.

To make a truly collaborative robot force sensors on the joints or only on the wrist are needed. Anyways these sensors are very expensive and are precluded to keep the cost limited. The Niryo Ned has collision sensors on its motors, but this product is in the upper side of price range and it is not fully 3D printed, so it would not be correct to try to catch with it.

We recognize that ROS has become a standard nowadays and so also our product need an high level interaction system, that makes easier the usage by not skilled people.

3 | Robot design

3.1. Preliminary Sizing

The usual way of sizing a mechanical device is first define the required characteristics, namely loads acting on the structure, size the structure and then find the most appropriate drives to actuate it.

This thesis work is part of several previous works. Hence, a route was already sketched. The "order of magnitude" of the dimensions of the robot arm links were already known, and motors characteristics were already clear.

Moreover, we had a collaboration with the Italian firm R.T.A. that provided some samples of the motors on their catalog, thus we tried to use the motors already available to size the robot arm.

The aim was to find the maximum lengths of the robot arm links that the motors could operate. Links' sizing was carried out statically and dynamically.

3.1.1. Schematization and Nomenclature

The robot that has been developed is a 6 dof (six rotational joints or 6R) robotic arm. Joints have been uniquely named in J1, J2, J3, J4, J5, J6. Numeration starts at the base of the robot and end at its wrist. Each joint is associated with a concentrated mass due to the stepper motor and its speed reducer. Moments of inertia of the stepper motor and speed reducer have been considered too.

Also links connecting the joints have been named. In particular, there are three links plus one for the base column. From base to wrist links are named L0, L1, L2, L3. Each link is associated with a mass and a moment of inertia.

For evaluation of masses and moments of inertia, the reader can refer to Section 3.1.4.

3.1.2. Model of the robot arm

A simplified mathematical model was developed to obtain a first tentative idea of the robot outreach and of the lengths of each link. This model only considered the robot behavior in a plane. Hence, only joints J2, J3 and J5 are of interest. By the way, only a 2R kinematic configuration (like a SCARA robot) was considered for simplicity and so only dof associated to J2 and J3 are considered in the model, also because these are the most stressed joints, having to lift the most of the weight of the arm with considerable lever arm. The major difference with a SCARA robot is the direction of gravity. Only links L1 and L2 are considered, while L3 is neglected. Link L3 anyway adds more lever arm to weight and inertial forces, and this must be considered. Therefore transport of moment was performed, placing the masses on link L3 at the end of link L2 and adding a concentrated bending moment.

Transported bending moment has two components: bending moment of weight force and bending moment due to inertial forces.

Weight force has direction of negative z and its lever arm changes as J2 and J3 rotates as in Eq3.1.

$$M_{trans,weight} = \left(m_{payload} + m_{J6} + \frac{1}{2}m_{L3} \right) L_3 g \cos(\beta + \gamma) \quad (3.1)$$

Inertial forces have been decomposed in x and z directions. The points of interest are the TCP P and the center of gravity G_3 of link L3. To compute components of acceleration of these points Eq.3.2 and Eq.3.3 are used.

$$\begin{bmatrix} \ddot{x}_P \\ \ddot{z}_P \end{bmatrix} = J_P \begin{bmatrix} \ddot{\beta} \\ \ddot{\gamma} \end{bmatrix} + \dot{J}_P \begin{bmatrix} \dot{\beta} \\ \dot{\gamma} \end{bmatrix} \quad (3.2)$$

$$\begin{bmatrix} \ddot{x}_{G_3} \\ \ddot{z}_{G_3} \end{bmatrix} = J_{G_3} \begin{bmatrix} \ddot{\beta} \\ \ddot{\gamma} \end{bmatrix} + \dot{J}_{G_3} \begin{bmatrix} \dot{\beta} \\ \dot{\gamma} \end{bmatrix} \quad (3.3)$$

J and \dot{J} are respectively the Jacobian matrix and its derivative. These matrices have the same construction as the ones for the kinematics of the SCARA robot, the difference is the lengths that are used. Hence, link L3 is always considered aligned with the link L2.

Transport moment of inertia forces is computed as in Eq.3.4.

$$\begin{aligned}
M_{trans,inertia} = & L_3 (m_{payload} + m_{J6}) \begin{bmatrix} \ddot{x}_P \\ \ddot{z}_P \end{bmatrix}^T \begin{bmatrix} \sin(\beta + \gamma) \\ \cos(\beta + \gamma) \end{bmatrix} + \\
& + \frac{L_3}{2} m_{L3} \begin{bmatrix} \ddot{x}_{G_3} \\ \ddot{z}_{G_3} \end{bmatrix}^T \begin{bmatrix} \sin(\beta + \gamma) \\ \cos(\beta + \gamma) \end{bmatrix}
\end{aligned} \tag{3.4}$$

The model was developed with MATLAB programming language.

3.1.3. Definition of links lengths

Knowing the maximum static holding torque of the motors (data available on the datasheet of the motors), a static assessment was performed to compute the lengths L_1 , L_2 and L_3 of links, considering the most critical robot pose, clearly the one with fully extended horizontal position. Iteratively, some (L_1, L_2, L_3) sets were evaluated and the required holding torque was computed. These sets are built using three discrete vectors for the lengths L_1 , L_2 , L_3 . Initial and final value of the vectors were chosen considering former works and modified with common sense if no result was found. Discretization step was chosen to limit the computational effort. Having discrete values to be evaluated leads to non optimal results in many computational applications, since one has no clue on what happens between two values. By the way, as will be presented hereafter, this stage is just pre-sizing of the links lengths and therefore we were not interested in finding an optimal solution.

With a post processing operation on the obtained results, only combinations that had a required torque smaller than the value provided by the manufacturer were filtered. Moreover, the maximum possible outreach was computed using the previously described filtered sets. All solutions presenting cumulated lengths of the links equal the maximum outreach were extracted.

Among these possible solutions, the one with a most balanced set of lengths was chosen in prevision of the geometry modeling phase with CAD software. Indeed, having links with almost equal length facilitate standardization of geometries and features.

The lengths found were approximated down for the dynamic assessment, since adding also inertial forces decreases the maximum links' length.

3.1.4. Estimation of Inertia

To perform a dynamical assessment is mandatory to know inertia. Since the robot was not manufactured yet, an estimation of mass and moment of inertia was done considering the material that will be used (ABS plastic density $\rho_{ABS} = 1.07 \cdot 10^3 \text{kg/m}^3$). One characteristic of 3D printing is having hollow structures with some infill material. Thus, the volume of the links is not fully related to their mass, but a multiplying coefficient c smaller than one was used to correct the mass (Eq.3.5).

$$m = c \cdot \rho V \quad (3.5)$$

Since the actual geometry of the links was still unknown, the position of the center of gravity was unknown too, thus it was placed in the middle of the link.

Links were simplified as simple cylinders and the moment of inertia of thick cylinders were used.

The coefficient c was set empirically (considering as said that the final structure will be partially hollow) but it cannot be set equal to the infill percentage that one plans to use. c has to be higher than the infill percentage to consider that the geometry will be more complicated than a simple cylinder. Anyway, c should not be set too high otherwise a small arm is obtained, since inertia would be overestimated. A reasonable value is $c = 0.45$ so to have masses of the link in the correct range.

A similar reasoning can be done for the speed reducer. Also speed reducers were simplified as thick cylinders and will be partially hollow. This time there will be metallic parts (bearings, screws and nuts) that are significantly heavier than ABS plastic. Hence, for estimation of their mass the corrective coefficient is set to $c = 1$.

The mathematical model requires the moment of inertia with respect to an axis passing through the center of gravity of the object (since momentum balance is done considering this point). Moment of inertia of links L1 and L2 is computed with Eq.3.6 since they are rotating around an axis parallel with x-axis in Figure 3.1.

$$J_{xx} = \frac{1}{12} m (h^2 + 3 \cdot r^2) \quad (3.6)$$

Moment of inertia of speed reducers need to be estimated too. As said, rotation of joint J5 is neglected for simplicity but its contribution of inertia must be considered. Joint J2 has no influence since it is placed in the center of rotation (shoulder) of the whole arm.

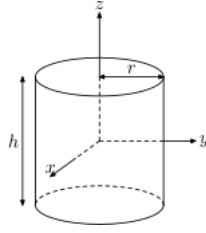


Figure 3.1: Simplification of links geometry

For joints J3 and J5 the axis of rotation is parallel to z-axis in Figure 3.1, thus Eq.3.7 must be used.

$$J_{zz} = \frac{1}{2} m r^2 \quad (3.7)$$

Joints J4 and J6 are instead axial and thus its moment of inertia can still be computed with Eq.3.6.

Moment of inertia of the speed reducers must be referred to the center of gravity of the correct link. Thus, transport of moment of inertia is performed as in Eq.3.8. Motor controlling J4 rotation was considered in the middle of link L2 and so no transport of moment of inertia is needed.

$$\left\{ \begin{array}{ll} J_{J3} = J_{zz} + m_{J3} \left(\frac{L_1}{2}\right)^2 & J3 \text{ referred to link } L1 \\ J_{J4} = J_{xx} & J4 \text{ placed in the middle of } L2 \\ J_{J5} = J_{zz} + m_{J5} \left(\frac{L_2}{2}\right)^2 & J5 \text{ referred to link } L2 \\ J_{L3} = J_{xx} + m_{L3} \left(\frac{L_2}{2} + \frac{L_3}{2}\right)^2 & L3 \text{ referred to link } L2 \\ J_{J6} = J_{xx} + m_{J6} \left(\frac{L_2}{2} + L_3\right)^2 & J6 \text{ referred to link } L2 \end{array} \right. \quad (3.8)$$

Since the actual application for the robot was not known, an indicative payload mass and inertia were used. The payload mass represents the mass of the actual object or tool to be carried and also the mass of the functional apparatus to perform the task (grippers, adapter flanges etc.). A desirable payload for a robotic arm is around 1kg, therefore this value was used for the rest of the assessment.

3.1.5. Dynamic assessment

The dynamic assessment was meant to verify the links lengths set after performing a motion simulation as a stress test. To do so, maximum values of speed and acceleration

of the robot arm needed to be set first. To set them it is required to establish the maximum allowable speed and acceleration that the robot could have, considering the dynamic behavior of the motors. The torque vs. speed curves on the manufacturer data-sheets were used considering 24Vsupply. Since only a qualitative representation was given (no polynomial function provided), the curves were approximated by straight lines (see Figure 3.2 and Figure 3.3). This approximation is acceptable since in this phase of the design we were interested only in indicative results.

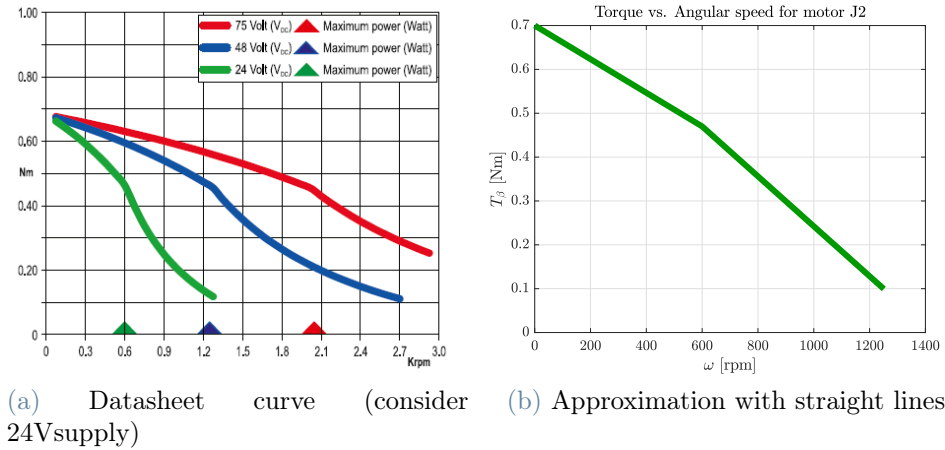


Figure 3.2: Torque vs. Speed characteristic of J2 motor

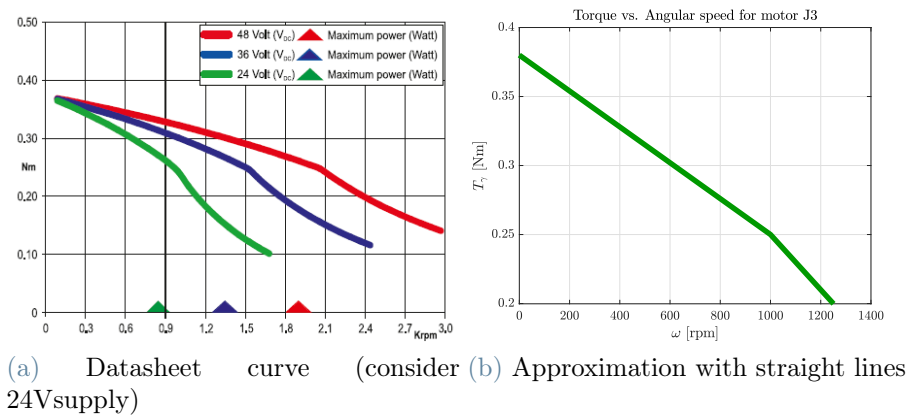


Figure 3.3: Torque vs. Speed characteristic of J3 motor

The final set of lengths is:

$$\begin{cases} L_1 = 200\text{mm} \\ L_2 = 200\text{mm} \\ L_3 = 180\text{mm} \end{cases}$$

The following numerical results are obtained with this set.

Indeed, these lengths are only indicative since they must comply with geometrical limitation for a feasible and proper manufacturing. Therefore, was not necessary to exactly match the constraints.

In fact, during the CAD modeling activity those dimensions were changed because of physical constraints. In particular, the length of the link 2 suitable for housing the speed reducer exceeded the limit computed. Therefore, since this limit length is no more satisfied, assessment of possible limits of the motor of joint J2 was required. This preliminary analysis is discussed in Section 3.4.

Setting maximum angular acceleration

Firstly, the maximum acceleration admissible was iteratively computed by computing the torques required by motors on joints J2 and J3 increasing in discrete steps the acceleration value. The maximum allowable acceleration is the value that makes these torque values matching the torque predicted by the torque-speed curve, imposing null speed. It is possible to notice that the static holding torque provided on the datasheet is higher than the torque at null speed on the torque curves. In Figure 3.4 it's possible to see the result.

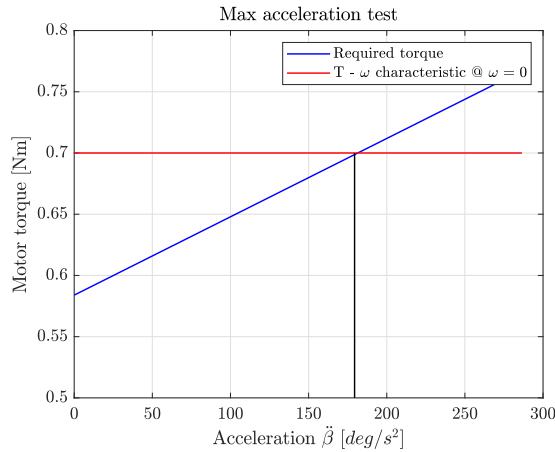


Figure 3.4: Maximum acceleration test

The meaning of this test is to establish how fast each joint can accelerate considering a sudden departure from static condition, so with an infinitesimal speed, and thus it's better to use the torque with zero speed. The maximum value of acceleration found depends on the set of the links lengths, but (L_1, L_2, L_3) is fixed. Indeed, the true aim is to verify that the allowable maximum acceleration for the robot arm is not bigger than the limit one for the motor. This eventuality is indeed remote since stepper motors have extremely high

maximum acceleration limit (practically infinite).

The robot arm is in the fully outstretched horizontal pose and accelerate upwards so that the inertial forces are summed with the weight forces. Since the position of the arm is constant in the time instant considered, the transport bending moment can be evaluated with a simple formula, as the lever arm is the outreach of the robot and the direction of acceleration is known (negative z -axis).

Once the maximum allowable value of acceleration for the arm is found, this characteristic value must be set to be smaller for safety, but still guaranteeing a good level of performance. The final value was set to $22.5^{\text{deg/s}^2}$. This value is needed for the motion simulation test described in the following.

Setting maximum angular speed

A limiting maximum value of angular speed was given by the control board of the motors kindly provided by R.T.A. On the datasheet a maximum stepping frequency of 60kHz is given but it looks unrealistic because would mean a motor speed of $54000^{\text{deg/s}}$. Moreover, on the torque vs speed curves in Figure 3.5 and Figure 3.6 there are limit speeds for 24V supply. The aim is to verify if the maximum allowable speed for the joints exceeded lower between the limit of the driver board and the one of motors, in this case the NEMA23 motor limit. Considering the speed reducer transmission ratio, the limit value of the link's speed is $223^{\text{deg/s}}$, that is still too big. We considered as limit $90^{\text{deg/s}}$ that is close to the maximum values of some of the existing products analyzed in Chapter 2.

The test performed consists in computing the torque required by motors on joints J2 and J3 when both links are rotating with constant speed, increasing at each iteration. Inertial forces generated in this condition are only centrifugal, since acceleration is imposed null for both joints. To convert a force aligned with the link's axis into torque the arm must be "L-shaped", so with links L1 and L2 orthogonal one with the other. Maximum allowable speed must be computed for both J2 and J3 and thus two separate tests are required.

To establish the maximum speed of for J2 the most stressful pose for the arm is with link L1 horizontal and link L2 vertical in downward position so that the centrifugal force sums with the weight force (see Figure 3.5). In this pose the motor on J2 actively bears the centrifugal load.

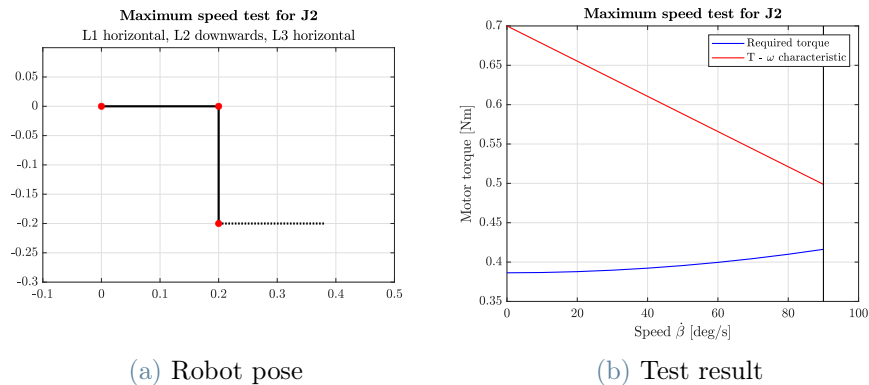


Figure 3.5: Maximum speed test for J2

For J3 instead the worst case is when it is rotating and it is bearing weight force. Therefore the required torque is constant for any speed, but it's still needed to verify if a speed value (lower than the board limit) is limiting the torque deliverable. Since rotation introduced by joint J5 rotation was not considered, it's not possible to assess centrifugal force produced when links L2 and L3 are orthogonal. In this configuration J3 has to resist the centrifugal force of link L2 but since it is vertical for sure it can.

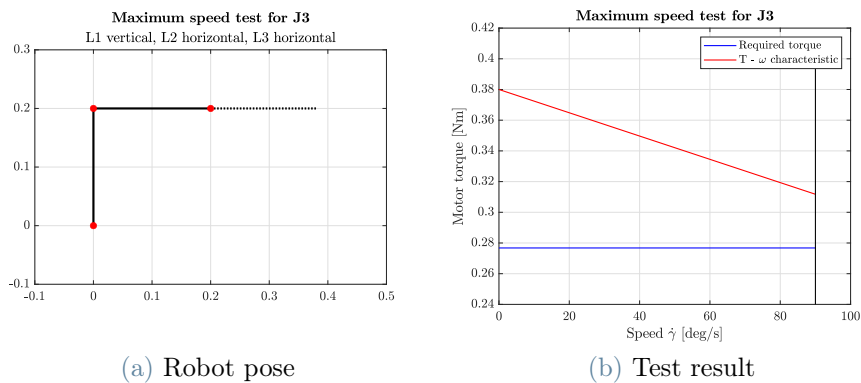


Figure 3.6: Maximum speed test for J3

Anyway, for safety the link L3 was considered horizontal to also add the transport moment of the weight force, in both tests for J2 and J3.

As it's possible to see in Figure 3.5 and in Figure 3.6, the board limit speed can be theoretically reached. For safety it is reduced to $60^{\text{deg/s}}$. This value is needed for the motion simulation test described in the following.

Motion simulation as stress test

Finally, a trajectory simulation was done to see the contributions of speed and acceleration together. The trajectory used is a trapezoidal speed profile (piece-wise constant acceleration) with synchronized motion between joints J2 and J3, as presented in Figure 3.7.

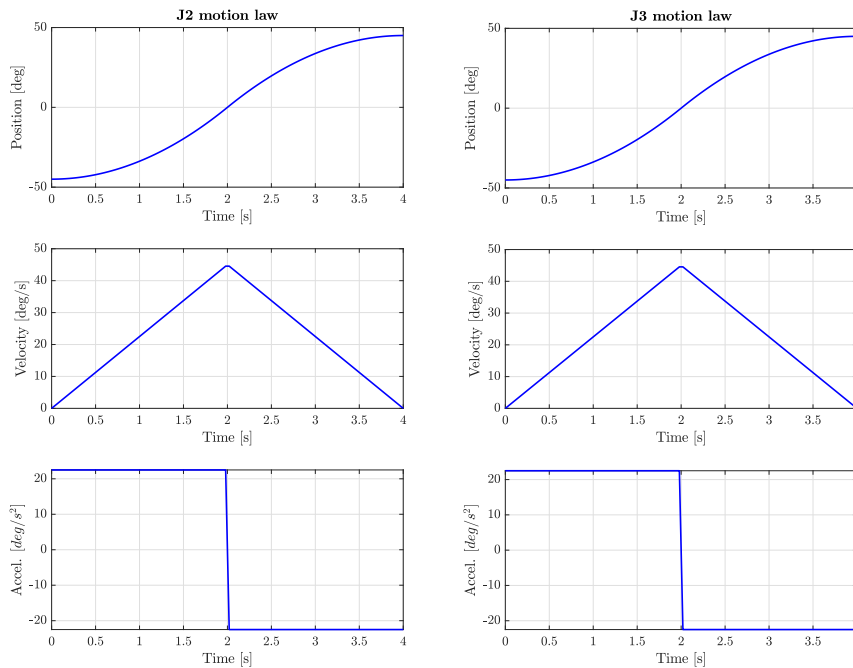


Figure 3.7: Motion laws

The range of the simulated motion considered was an indicative stress test representative of the most stressful movement for the robot arm. The trajectory is a symmetric swing from the fourth to the first quadrant because the maximum acceleration is reached with the arm fully outstretched in horizontal position (maximum lever arm) and it is summed with the weight force, exactly when the speed is maximum and so the torque deliverable by the motors is minimum, being the torque vs. speed curves monotonically decreasing.

The simulation aims to find the maximum allowable links lengths comparing the required torque curves of motors of joints J2 and J3 with the deliverable torque curves (torque characteristic of the motor evaluated at the speed of the trapezoidal profile and scaled with the transmission ratio). If the required torque is lower than the deliverable torque (at almost any speed) the test is passed. By trial-and-error procedure the maximum lengths of the links were tweaked. It is clear that the final feasible links lengths depends on the

maximum speed and acceleration that are set.

The trajectory simulation was the strictest test to be satisfied. Results are shown in Figure 3.8.

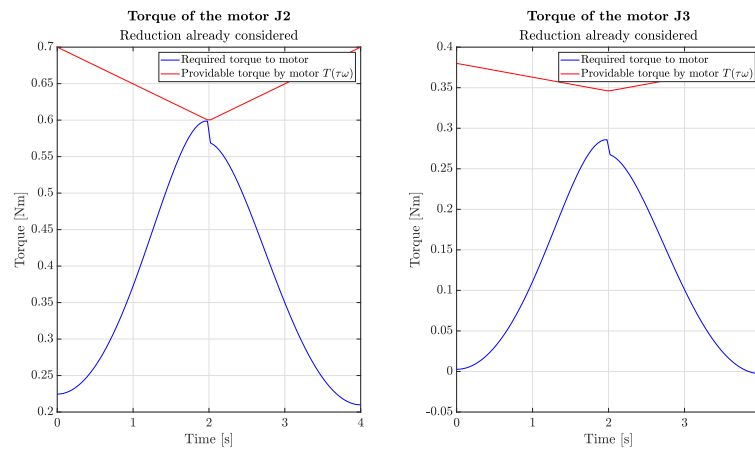


Figure 3.8: Torque profiles

3.2. Mechanical design

Modeling of parts was performed with the CAD software Autodesk Inventor Pro.

Advantages and drawbacks of the manufacturing technology used were clear, thus modeling was performed consequently.

In particular, modeling for 3D printing needs to:

- consider the direction of the main load that the component has to withstand and hence geometrical features must be realized considering the most appropriate printing orientation.
- avoid too small features that can be unfeasible after slicing.
- avoid too steep overhangs in places where support material can be difficult to remove or can cause quality issues.
- avoid supporting functional geometrical features as holes for screws and gear teeth.
- account for geometric deviations due to building direction (for example, circular geometries printed perpendicularly to the build plate).
- account for higher tolerance than conventional subtractive manufacturing processes

The aesthetics of the robot was taken from a previous thesis work [25]. In particular, the section where two links are joined resembling the Opera House Theater of Sidney was kept since it constitutes the characteristic sign of the robot arm.

Eventually, some modifications to the typical geometry were needed since different constraints arose. Among the important we want to highlight:

- The characteristic section was tweaked because the links lengths changed. The overall look was maintained but the angle of the side cuts was increased from 30deg to 45deg otherwise the central section of the link 1 would have been too weak.
- The housing of the stepper motors was enlarged to contain the new and bigger drives. In fact, the motors used for joints J2 and J3 have a braking device embedded that makes them longer. The links were modeled with some "bulges" that were fillet to make a seamless transition. Instead, the housing for the cover of the HD are smaller and almost flush.

3.2.1. Harmonic Drives

Speed reducers used in this robotic arm are a further re-design of some other HDs already developed in previous thesis works ([18] and before [2]).

This kind of transmission is very used in robotics applications because they can achieve high reduction ratio in compact space, allowing to use smaller and lighter motors in the robot joints. Commercial (metal) HD have practically zero backlash and they can transmit high torque with high reliability.

HDs are of a peculiar type because they use the deformation of a flexible element to achieve speed reduction. This element is called flexible spline and it's a sort of cylindrical shallow cup (having thin walls) and with teeth on a lip on the outer surface. Deformation is achieved with the wave generator that is an elliptical component (wave generator plug) endowed with a thin-walled deformable ball bearing. The wave generator is placed inside the flexible spline and the outer ring of the deformable ball bearing is everywhere in contact with the flexible spline, so that the latter deforms elastically in an elliptical shape. The ball bearing allows decoupling between the wave generator plug and the flexible spline, thus no sliding motion can occur between parts, causing wear. The flexible spline teeth mesh over the last major component that is called circular spline. The circular spline has circular shape and it's not deformable. The pitch diameter of the circular spline and the major axis of the ellipse that is obtained deforming the pitch diameter of the flexible spline are ideally equal. In this way the flexible spline meshes on the circular spline only on two regions on opposite sides. Moreover, the circular spline has a different number of teeth than the flexible spline. It is common to find two more teeth on the circular spline. To achieve speed reduction, the input of the motion is the wave generator (connected to the motor) and the output is the flexible spline. The base of the latter is usually rigid enough to connect a shaft to extract the motion from the drive. The kinematic principle exploits the difference in the number of teeth to achieve speed reduction. The formula describing the transmission ratio is given in Eq.3.9 ([19], [21]).

$$reduction\ ratio = \frac{flex\ spline\ teeth - circular\ spline\ teeth}{flex\ spline\ teeth} \quad (3.9)$$

As said, the number of teeth on the circular spline is bigger than the one of the flexible spline, and hence the reduction ratio is computed with negative sign. For every full rotation of the flexible spline (using two more teeth on the circular spline) the flexible spline moves of only two teeth in the opposite direction (explanation of the negative sign of the transmission in Eq.3.9).

Straightforward implementations of HD have quite significant axial encumbrance (Figure 3.9 [21]) and in [18] the aim was to minimize this dimension.

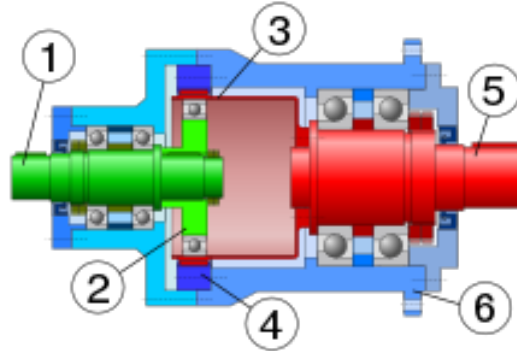


Figure 3.9: Traditional HD layout. 1: input shaft. 2: wave generator. 3: flexible spline. 4: circular spline. 5: output shaft. 6: housing

In the cited work some changes to the typical elements of an harmonic drive were done. The wave generator has two eccentric ball bearings that deform the flexible spline in the elliptical shape. The flexible spline itself is not a single piece but the flexible element is a closed synchronous rubber belt that is inserted in a rigid holder. The circular spline is instead quite conventional. We were given the CAD files and starting from those part we started studying new features to improve them. We changed type of hardware (screws and nuts) used and modified how the hardware is used in the plastic parts. We modified geometries to adapt the parts to the new type of motors used. We modified the manufacturing strategy to achieve the flexible spline due to an initial shortage of components. We implemented the mechanical components for the position feedback system.

Hardware

The first thing was to change the hardware used so to make the BOM leaner, in particular using only one type of screw but with different lengths and thread diameter. In this way sourcing of parts is easier and assembling more straightforward.

The characteristics identified for the screws were:

- Having a small head. In particular, a cylindrical head was identified as the best option.
- Being easily accessible. Since some components have small dimensions, the most suitable tool for tightening the screws was identified as an Allen key. Thus screws with an hexagonal socket head cap were chosen. In particular, ISO4762 (DIN912) screws were chosen. Depending on the specific need, M3 and M4 screws were used.

For the M3 screws the length ranges from 10mm to 25mm, while only M4x16 screws were used. These screws are widely available on the mainstream commercial channels and on e-Stores. Moreover, fully compatible screws are sold without being compliant with the before mentioned standards, thus lowering the cost for fasteners.

Other hardware needed was essentially nuts. Two types of nuts were used: square nuts and self-locking hexagonal nuts with nylon inserts (nyloc nuts). CAD models were assembled using ISO7040 M3 and M4 self-locking nuts (but again on the market are available compatible nuts with no specified standard) and DIN562 M3 square nuts. These nuts are widely used in 3D printers and thus can be easily sorted. Two types of nuts were required because as much fastening techniques were used, hence the most appropriate nut was selected.

Nuts need to be inserted in the 3D printed parts to fasten parts together. To make assembling easier we studied what type of nuts to use and how to insert them in the parts. In Figure 3.10¹ it's possible to see that there are two options to realize a captive nut. From left to right, the first three nuts are inserted with the same orientations, while the last is different.

The first method uses a cavity in the part and completely insert the nut in the plastic. For this technique square nuts are more suitable since they are smaller with respect to hexagonal ones, and therefore the bridge² length is shorter, facilitating manufacturing through 3D printing.

The other way uses a pocket of the same shape of the nut in the part and then press fit the nut into it. This method can be used with both hexagonal and square nuts. With this method, the nut is almost invisible.

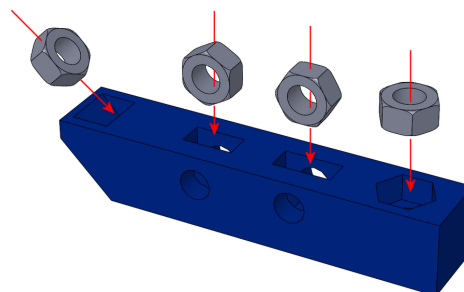


Figure 3.10: Captive nut orientations

¹Source: <https://www.instructables.com/CAPTIVE-NUTS-AND-MORE-IN-3D-PRINTING/>

²In 3D printing a bridge is an extrusion of plastic performed in mid air.

Another way to mate 3D printed parts is to use threaded inserts similar to the ones used for injection molding plastic parts. These inserts are quite convenient but are challenging to be precisely placed. In fact, they require a heat source as a soldering iron to let the plastic around them melt and then by pressing the insert it's possible to embed it in the part. Ultrasound can be also used. Hence, this technique is not as straightforward as pressing a nut and can be dangerous and damage the plastic part.

The selected screws and nuts were used in all other parts of the robot arm.

Base flange

The flange that is used to connect the stepper motor to the HD was changed too. The flange was adapted to use square nuts for the holes securing the HD cover. Moreover, since the robot developed in this thesis work uses also NEMA23 stepper motors, a flange with the correct hole spacing was required. In Figure 3.11 are reported the flanges for both NEMA17 and NEMA23 stepper motors.

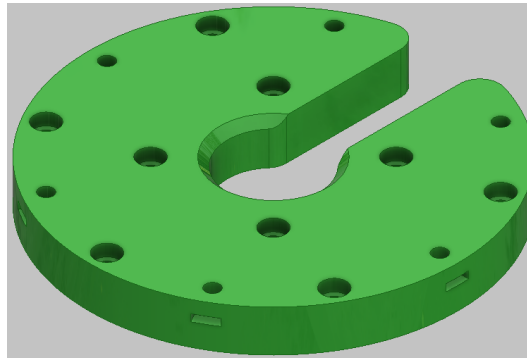
Two possibilities were analyzed for the connection of the NEMA23 drives with the flange. Those motors have through holes instead of threaded holes of NEMA17 ones. If for NEMA17 motors is mandatory to assemble them as first part to the flange, NEMA23 motors can be mounted afterward. The first option was to use the same geometry of NEMA17 flanges so inserting the screws in it, insert the motor and secure it with nuts. The second option is to trap the nyloc nuts in the flange so that the motor can be secured afterward when the HD is completely assembled. This second strategy was preferred since it's more practical because during assembly it's not necessary to continuously moving the heavy motors with the risk of damaging the brake.

Wave generator

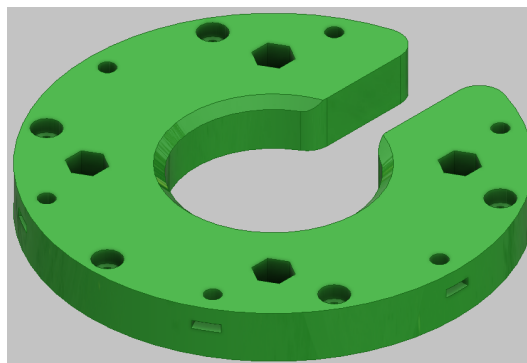
A new wave generator for NEMA23 motors was needed. The differences with respect to the NEMA17 version are:

- The diameter of the shaft. For NEMA17 it is 5mm while for NEMA23 is 6.35mm that corresponds to 0.25inch. Therefore we had to modify the diameter of the central through hole on the wave generator.
- The required mounting hub. The Pololu mounting hub used for NEMA23 motors has six holes instead of four in the NEMA17 version, and the hole spacing is also different.

To mount the small bearings on the extremities of the wave generator M3 screws were



(a) NEMA17 base flange



(b) NEMA23 base flange

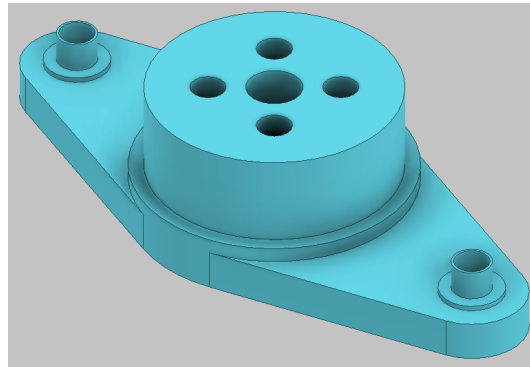
Figure 3.11: NEMA17 and NEMA23 base flanges

used. Anyway, the internal diameter of the bearings is 4mm. Thus, a sort of bushing to bring from 4mm to 3mm the internal hole was modeled on the part. This feature is very thin and so it will be just one layer thick once printed. Another solution can be use countersunk M3 screws so that the conical portion of the screw head engages the inner track of the bearing assuring axial alignment. To avoid adding another item on the BOM, it was preferred to use a 3D printed bushing on the wave generator itself.

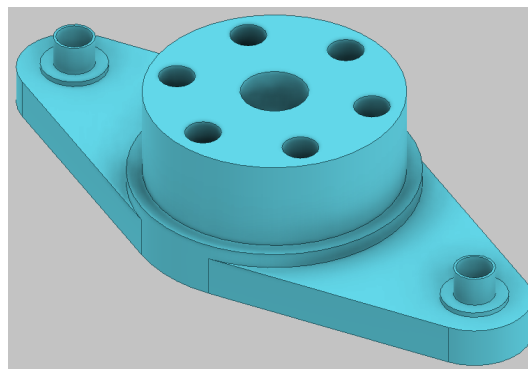
In Figure 3.12 the two wave generators are reported.

Flexible spline

In the original design the flexible spline was a closed synchronous rubber belt turned inside out. In particular was chosen a belt with HTD 3M tooth profile with 70 teeth. The circular spline has two more teeth and therefore the reduction ration is $i = -\frac{1}{35}$. This strategy has several advantages, the main of which is that flexibility is together with very high resistance due to the internal steel wires or polymer fiber reinforcements that are oriented as the main load that is tangential stress induced by torque. Anyway, this particular model was hard to sort on the available suppliers and expensive.



(a) NEMA17 wave generator



(b) NEMA23 wave generator

Figure 3.12: NEMA17 and NEMA23 wave generators

Thus, we tried to purchase an open belt and turn it into a closed one by several attempts that are reported in the following list.

- We simply cut at the correct length the belt and fitted it in the groove modeled in the rigid part of the flexible spline. We tried to actuate the speed reducer but it didn't work. The main cause is that the junction point of the two extremities of the belt made a cusp. We ascribed this to the steel wire inside the belt that induce a spring behavior that makes it open in the weakest point, so where there is a discontinuity in the reinforcement wire itself. This issue cannot be solved simply using a tight fit between belt and groove because the non supported length must be big to allow deformability, and so the cusp shows up anyway.
- We tried to close the piece of belt by melting the two extremities with a soldering iron and fitted it in the rigid part but the cusp happened again and the reducer didn't work again.
- We tried to glue on the rubber a thin strip of spring steel to restore the continuity of the steel wire behavior. Unfortunately, no glue was found to stick on the rubber.

The only option left was to 3D print the flexible spline. One section must be rigid and flexible in another. For what CAD modeling concerns, the required changes are:

- model the HTD tooth profile and the new flexible element in the CAD software.
- modify the interface between the output flange and rigid part of the flexible spline, because a different printing orientation was required. The shaft on which the upper 61805 bearing is brought from the rigid part of the flexible spline to the output flange (Figure 3.14).

In Section 3.3.4 further details are explained on the actual printing strategy. For now it's important to say that the model was split in two bodies as it's possible to see in Figure 3.13.

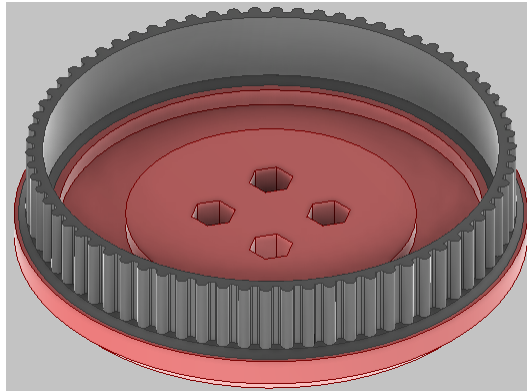


Figure 3.13: Flexible spline

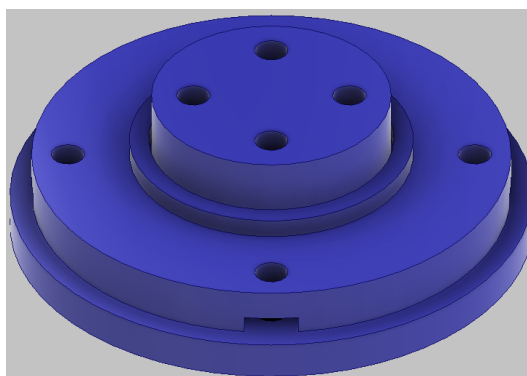


Figure 3.14: Modified output flange

Gears for rotation sensing and HD cover

Another modification to the original design was to add gear teeth on the cover of the harmonic drive. Indeed, two version of the cover are needed.

For non axial joints rotation sensing is performed with two gears one of which is a ring fitted with interference on the HD cover. To avoid slip three grooves were modeled on the gear ring engaging with three spines on the cover. To provide support to the ring a ridge was modeled on the cover. Since it would interfere with the screws mating the cover to the base flange, some cutouts were modeled to facilitate the tightening operation. In Figure 3.15 the described solution is presented.

Several options for blocking axial movement were analyzed, but given the small thickness of the HD cover no satisfying solution was found. Among the main solutions we thought about:

- using three set screws to be screwed radially in gear ring and in the spines on the cover. No space was available for a metal nut, thus the set screws would cut threads in the plastic and this fact is not ideal and made us discard this solution.
- using some thin plastic part to be fitted on top the ring and in a groove on the cover, but the thickness of the cover was too small and we discarded also this option.

For axial joints (namely J1 and J4) rotation sensing is performed with two gears one of which is modeled as a feature on the arm's links. Hence, the original cover of the HD can be used.

Anyway, during the prototyping phase we noticed that the edge on the three lips to fasten the cover on the base flange were prone to rupture. The cause is mainly not so strong layer adhesion of ABS when 3D printed (see Section 3.3.2). Therefore a chamfer was added to thicken the resistant section and proved a transition from the lips to the thin section of the cover (see Figure 3.15a and Figure 3.16).

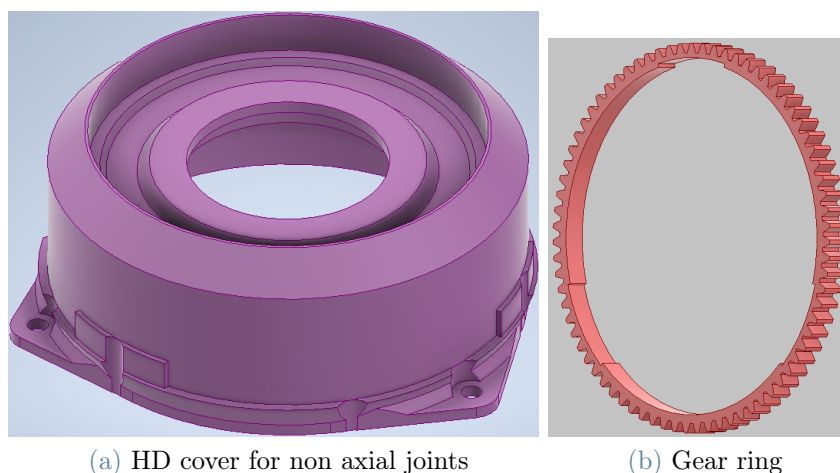


Figure 3.15: Gears for non axial joints

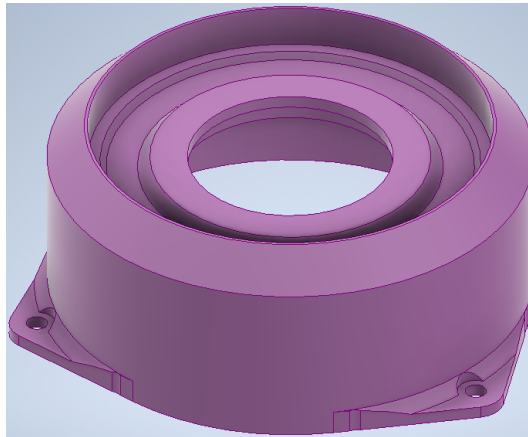


Figure 3.16: HD cover for axial joints

Finally, a representation of the assembly of the harmonic drive for joint J1 (NEMA23 stepper motor) is reported in Figure 3.17. See Attachments for technical drawings.

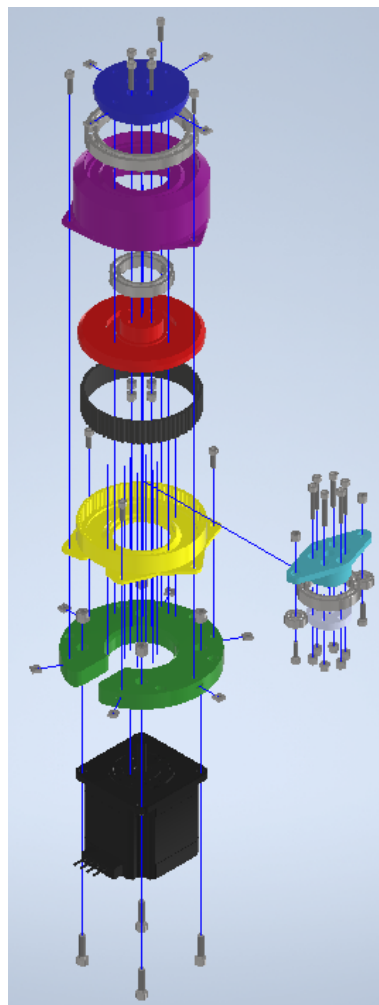


Figure 3.17: Exploded view of a NEMA23 harmonic drive

3.2.2. Selection of rotation transducers

For control of the robot assessment of the rotation sensing system was required.

Several possibility were found and discussed, among which rotary optical, magnetic encoders and drive encoders. The problems with this types of encoder are the high cost (>100€ each) and that is was difficult to find constructive solutions due to the available sizes of the magnetic discs and required positioning of the parts. A total redesign of the HD would have been necessary.



Figure 3.18: Rotary magnetic encoders

High precision potentiometers were the final choice because are the ones that most comply with the low cost constraint, being relatively inexpensive. In particular, multi-turn potentiometers were used. They are actuated by means of gearwheels. As seen, there are two types of joint in the arm that require different approaches:

- joints J1 and J4 are axial and use an external solution. The gear on the potentiometer meshes on gear teeth realized on the link.
- joints J2, J3, J5 use an internal solution. The gear on the potentiometer meshes on gear teeth realized on the cover of the harmonic drive.

The gears were sized to have a transmission ratio $\tau < 5$. The reason of this limit value of τ is that the gear on the transducer should be small for reasons of encumbrance but big enough for feasible manufacturing through 3D printing. Gear sizing was performed with the Spur Gear Design Accelerator tool of Autodesk Inventor.

Moreover, constraints on diameters of gearwheels are identified:

- the external diameter of the gearing embedded on the links must smaller than the external diameter of the links, so that teeth are flush with the external surface of the link.

- the inner diameter of the gear ring to be mounted on the HD cover must be bigger than the external diameter of the HD cover itself. Clearly, this is because otherwise it would not be possible to fit the ring gear on the cover.

To satisfy these constraints we acted on the wheelbase value of the gear set. In Table 3.1 and Table 3.2 the main characteristics of the gears are reported.

Parameters	Value
Gear Ratio	4.4118
Wheelbase	52mm
Module	1.125mm
Pressure angle	20deg
Pinion gear	
Pitch diameter	19.125mm
Number of teeth	17
Ring gear	
Pitch diameter	84.375mm
Number of teeth	75

Table 3.1: Gearing for non axial joints

Parameters	Value
Gear Ratio	4.8824
Wheelbase	59mm
Module	1.125mm
Pressure angle	20deg
Pinion gear	
Pitch diameter	19.125mm
Number of teeth	17
Embedded gear	
Pitch diameter	93.375mm
Number of teeth	83

Table 3.2: Gearing for axial joints

The obtained gears have been modified adding features to allow mounting on the potentiometer shaft. A through hole in the center of the gear and a collar were modeled. The collar height depends on the the space available for mounting. The gear is planned to be fastened on the potentiometer shaft with one M2 set screw, but friction fit is enough.

Consequently five-turns potentiometers where chosen because $\tau < 5$ and each joint won't perform a complete rotation. Therefore there is a good safety margin to not engage the mechanical endstop of the potetiometer. The chosen potentiometer is the model Bourns 3548.



Figure 3.19: Bourns 3548 potentiometer

This solution requires Analog to Digital Converters (ADC) for which the number of bits of the converter has to be sized. In Section 4.1.4 the sizing of the ADC is carried out.

3.2.3. HD modifications after testing

During the very early testing phases further minor modification but of paramount importance for the proper working of the robot arm were needed. We experienced a quite pronounced backlash and failing meshing of the flexible spline over the circular spline inside the joint J2 with only the link L1 attached to it. In Section 7.1.2 further details will be given.

A quite pronounced backlash in the HD showed up when testing the robot. A big contribution to backlash was caused by the screws connecting the output flange to the flexible spline that were loosening when the joints were actuated. Using some thread locker fluid is necessary. Excluding loosen screws, we reckon that the play is caused by at least 2 factors that are hereafter reported in order of importance:

1. A non-perfect meshing between circular spline and flexible spline. If there is a circumferential gap between the teeth on the flexible spline and the ones on the circular spline, the output flange can rock back and forth. The rotation can be transmitted to the 624 ball bearings on the wave generator even if the motor is locked (with brakes or just by its holding torque). This driver is therefore responsible for the backlash without any resistant torque applied to the HD.
2. Flexibility of the rubber material of the synchronous belt and of TPU material. Even if the meshing between the teeth on the flexible spline and the ones on the circular spline is ideal, rubber teeth can significantly deform when loaded. Thus, this driver is activated when there is a resistant torque applied to the HD.

We tried to further reduce the amount of backlash by acting on the identified drivers. In

the original design the circular spline has a teeth profile realized with a simple extruded semicircular feature. This fact from one side is beneficial when the HD is not loaded since guarantees enough play to allow the flexible element to deform reliably. On the other hand, when the speed reducer is loaded the non-exact teeth profile on the circular spline can facilitate the loss of the meshing when a critical threshold of resistant torque is applied. And that is exactly what happened during testing.

If backlash can be neglected (at least in the very early stages of prototyping), a reliable HD that allows for a decent range of motion to the most stressed joints is mandatory. We tried some simple teaks to the original design to solve the issue.

The easiest possible solution is to increase the orthogonal force on the rigid spline by increasing the wheelbase between the bearing on the wave generator. We modeled several wave generators increasing progressively this distance. The principal concern on this modification is that if the orthogonal force is too big the HD can completely block because the friction force becomes too high. Moreover, by pressing too hard the teeth profile on the flexible spline deform significantly being made of soft rubber, compromising meshing. Therefore we used this modification only when necessary.

Another minor modification is to change the teeth profile on the circular spline. We modeled a circular spline having a complementary HTD3M teeth profile to achieve a more accurate meshing.

Finally, it is possible to slightly change the shape and proportions of the flexible spline. We modified the shape of the 3D printable flexible spline and wave generator finding inspiration on designs of commercial products. Looking to Figure 3.20 we can see that:

- Commercial flexible gears have a diameter to height ratio almost equal to 1 and the toothed section is only where the wave generator is.
- Commercial wave generators are elliptical and touch the flexible gear all around its perimeter.

Therefore, the 3D printable flexible spline was modified but changes were constrained by the dimensions of the original design. The rigid section was reduced to have a taller flexible section and teeth were modeled only in the last 6mm of it. The main motivation for this is that we noticed that the deformation of the 3D printable flexible spline (made as the belt was glued to it) was not covering the bearing of the wave generator. This was caused by both a too circumferentially rigid and too short flexible section. When the wave generator was inserted the section in TPU had the shape of a cone rather than a cup. Clearly the deformation during meshing is more important to evaluate but

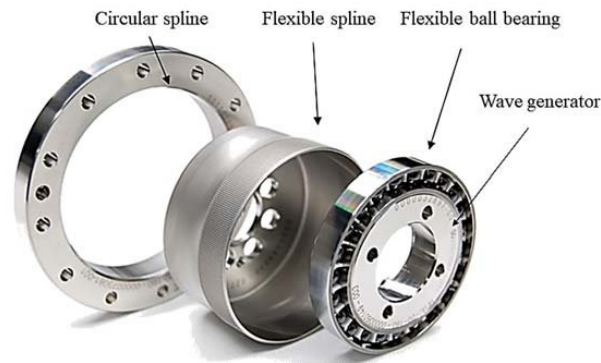


Figure 3.20: Commercial harmonic drive parts

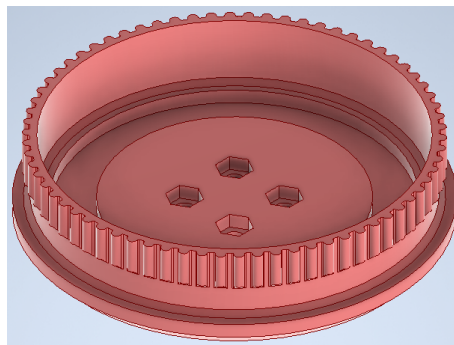
unless a transparent cover and circular spline are made, it's not possible to see the actual deformation. A FEM simulation can be done in future to assess this aspect. As presented in [5] a mathematical model of the HD can be developed considering the actual teeth profile of flexible spline and circular spline. Stresses and displacements can be simulated to assess quantitatively the stress state of the flexible spline and understand the actual resistance of the 3D printed parts.

As said previously, backlash is also due to the intrinsic characteristic of how the flexible spline is made, that is with a rubber belt or with TPU if 3D printed. Backlash is therefore amplified under load by the deformation of the soft polymer itself. Commercial flexible gears are made of thin-walled metallic cylinders that are circumferentially elastic and highly deformable but torsionally stiff. Our 3D printed flexible spline instead is also torsionally deformable being made of TPU. Three possibilities arise considering what discussed about the backlash.

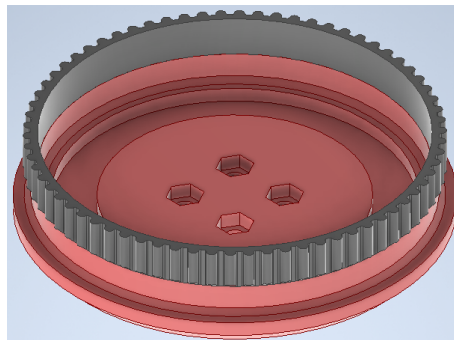
1. Make the entire length of the thin section of the flexible spline in hard plastic to mimic commercial products. This solution has the advantage to reduce backlash, but the drawback is that deformability is reduced and therefore accurate tolerance between parts must be guaranteed. Fatigue concerns arise because periodic shear stress along the layers can cause premature failure of the part. In this regard ABS plastic must not be used because has not the best layer adhesion among polymers and thin parts tend to delaminate easily. Moreover, the drive will be noisier.
2. Make the entire length of the thin section of the flexible spline in TPU. This possibility is dual with the first one: tolerances can be bigger, longevity and silence of the drive are higher but backlash under load is intrinsically bigger and therefore the robot would be more unprecise.

3. Make a compromise realizing the teeth in TPU and the left section of the flexible spline in hard plastic. Again, ABS is not recommended as in point 1.

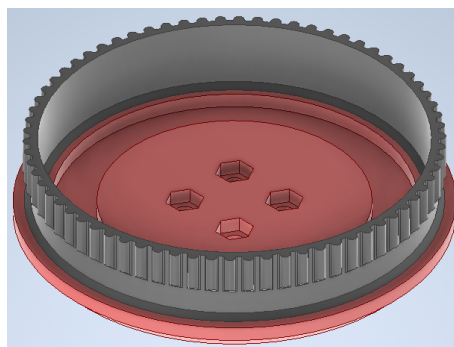
The main concern about this strategy is the layer adhesion strength. In Figure 3.21 the proposed solutions are reported.



(a) All hard plastic



(b) Half rigid plastic, half TPU



(c) All TPU

Figure 3.21: Modified flexible spline3.20

We thought also about modifying the wave generator to better constrain the flexible spline to assume an elliptical shape. The idea for this tweak is that if the flexible element is less free when it's not meshing, then the meshing will be more accurate. The wave generator was modified adding other two 624 ball bearing perpendicularly to the original ones that are in contact with the flexible spline but don't causing meshing (Figure 3.22). This solution is the simplest modification to the wave generator design that goes in the direction of the actual designs. Anyway, due to lack of 624 ball bearing was not fully tested and anyway it did not prove to be beneficial.

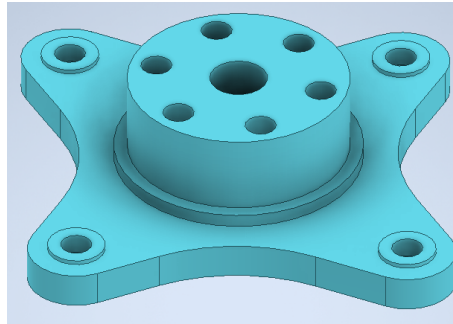


Figure 3.22: Wave generator with 4 bearings

The ultimate solution would be to design a 3D printable elliptical ball bearing like many DIY solutions using for instance Air Soft balls as rotating elements. The principal critical point is to realize a flexible outer ring like the one on actual elliptical bearings (Figure 3.23). Moreover, to avoid excessively fast wearing of the 3D printed bearing, plastic balls should be used rather than metal or ceramic ones.

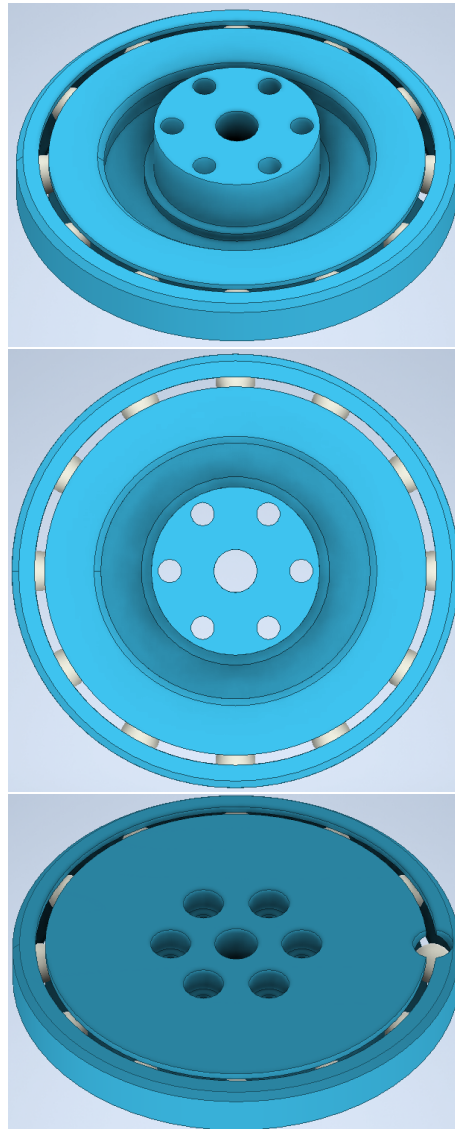


Figure 3.23: 3D printable elliptical bearing

3.2.4. Robot arm links

Base

The main requirement for the robot base is that it must be easily fastened on any workbench. A 3D printable structure has been modeled with a total of six through holes to be used to mate the base, for instance, to a plywood sheet using M4 screws. For this reason a circular base was chosen. To better distribute the pressure pockets for washers have been modeled too. For a more reliable joint threaded inserts for wood were chosen. Another possibility is to mount the robot on a base made of aluminum extrusions. To make the base more rigid a set of six ribs were placed. Sizing of the base was performed with simple static equilibrium. The base diameter was chosen to be printable on a common desktop 3D printer, and was set to 180mm. To fit the part on the build plate of the 3D printer used the base was cut. With the robot in the fully horizontal outstretched position the maximum bending moment was computed, using the simplified model described in Section 3.1.2. The screw on the opposite side of the arm was verified to yielding, as if only one screw was used to bear the load at a distance from the base axis. The resistance class of the screw was known and so the yielding load could be derived. The core diameter was derived knowing the thread diameter. The result of this simple assessment is that one M4 screw is more than enough to hold the base when the maximum bending moment is applied.

$$\begin{cases} F_{ext} = \frac{M_{bending}}{r_{screw}} \\ F_{yield} = YS \pi \frac{d_{core}^2}{4} \end{cases} \rightarrow F_{ext} \ll F_{yield}$$

The most critical part of this assessment is that the load to be applied on the screw is too much for the plastic material. Only an experimental evaluation can be done to verify the structural resistance of the holes. As said before, using wide washers for sure help to better distribute the axial force on the material. To account for this uncertainty many perimeters should be used for printing the part.

To connect the HD of link J1 to the base 5 square nuts and M3 screws were used. Pockets to insert the nuts have been modeled along the internal cavity of the part. To make insertion of the nuts easier the base of the pocket has an horizontal surface to rest the nut on and then push it in with a flat screw driver.

The motor for joint J1 is a NEMA23 stepper so a square cutout was modeled to fit the motor flange.

On the base the housing of the potentiometer used to measure the rotation of the joint J1 was placed. To make assembling easier and maintenance more straightforward, the transducer is inserted in a flange realized on the base and then tightened in place with the hex nut provided. To assure angular positioning a mating structure was modeled for the corresponding feature on the potentiometer. Underneath, a cavity has been modeled to pass outside the motor and potentiometer cables.

In Figure 3.24 the part is presented.

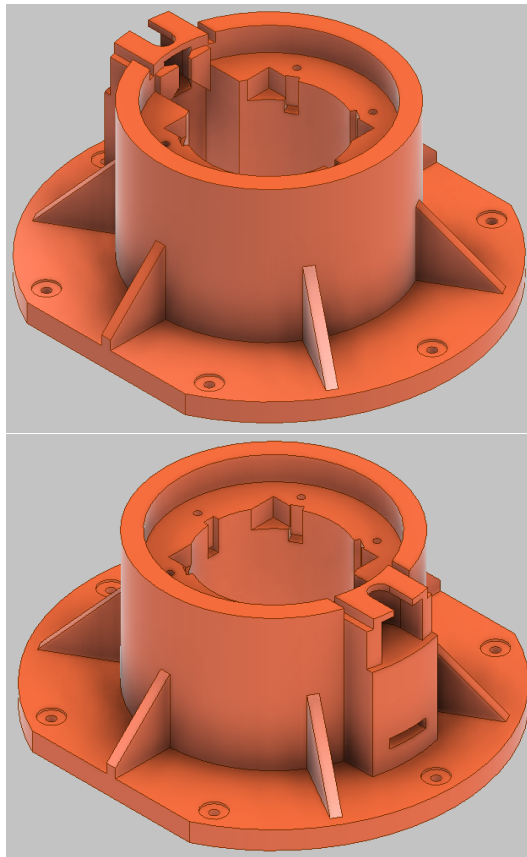


Figure 3.24: Robot base

Link 0

Link 0 is the column that has the shoulder of the robot arm (joint J2).

Inside Link 0 the joint J1 is housed, allowing rotation around the vertical axis. For measuring the rotation of J1 an external potentiometer was used, since no space was found to put it inside the link. For this reason Link 0 has gear teeth directly realized on the structure itself and flush with the external surface.

Joint J1 is axial and coupling with the link is performed with a 3D printed coupler (Figure 3.25) screwed axially on the output flange of the HD and radially to the link. It contains 6 pockets for square nuts. To guarantee radial alignment a spine was modeled in the cavity for J1 joint that slides in the corresponding groove in the coupler. To set the correct height of the coupler a 4mm hole is used to temporarily insert an Allen key allowing to tighten the radial screws.

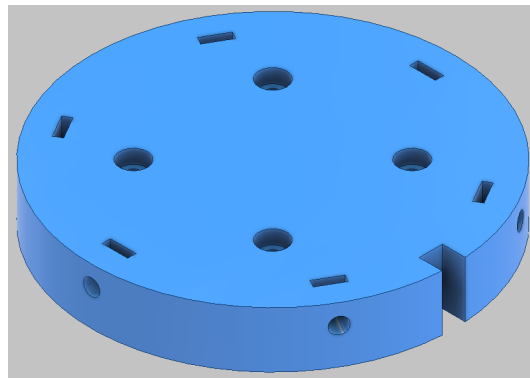


Figure 3.25: Axial coupler

To connect the HD of joint J2 the same as exposed for the base was done. To guide outside the motor's wires a channel was modeled in the part.

The part is presented in Figure 3.26.

The most appropriate building direction (layering parallel to its axis) would require support material to be generated on the teeth. This fact is not ideal since the gear quality could be compromised. Thus, Link 0 can be split in two parts (Figure 3.27) to make manufacturing easier. Moreover, this allows the part to be printed on smaller machines. The part with the J2 housing can be printed with the best layering direction to assure strength, while the part with the gear embedded can be printed with layer perpendicular to the link axis for better quality of the gear. The two parts are assembled using six M3 screws and square nuts.

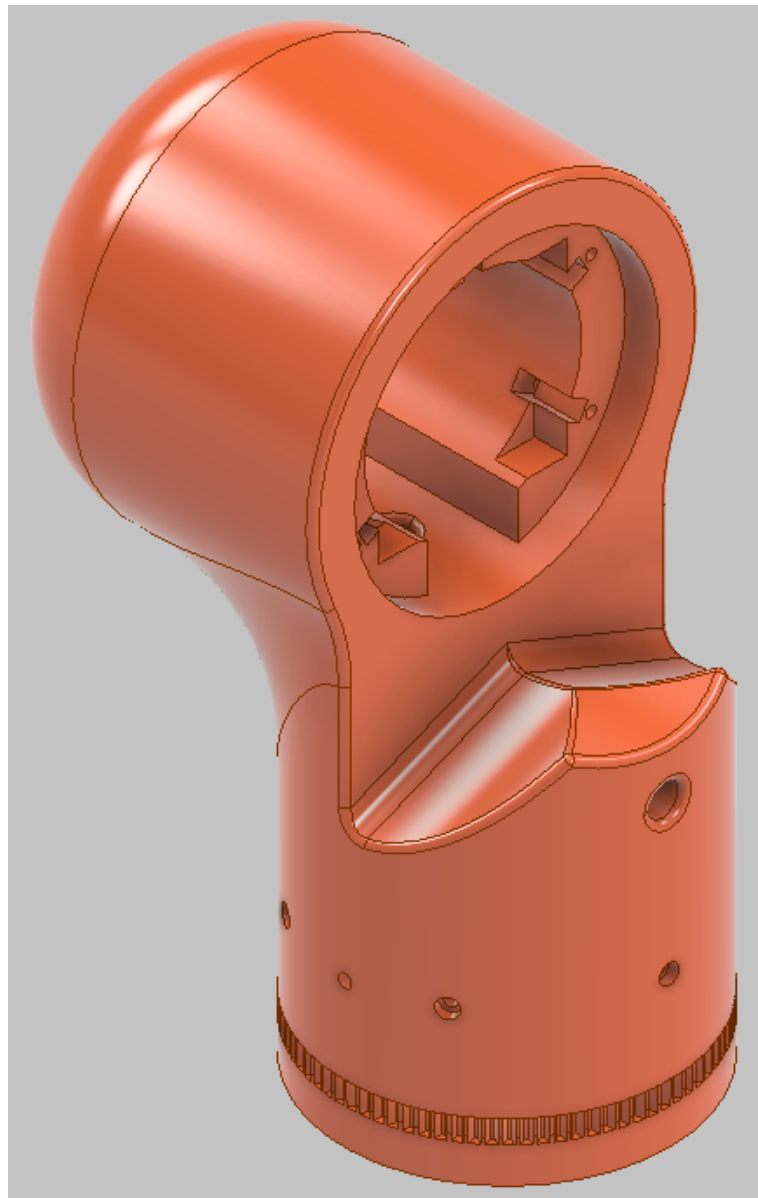


Figure 3.26: Link 0

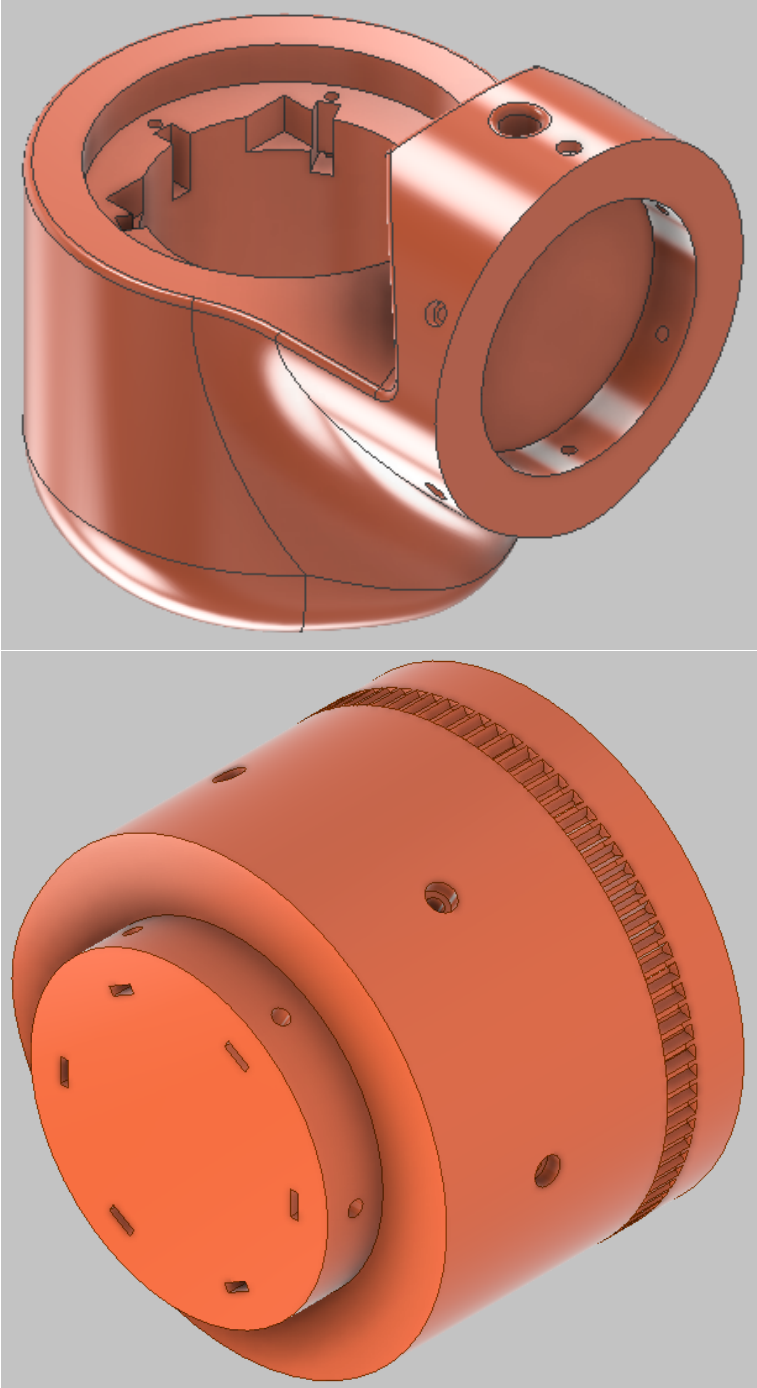


Figure 3.27: Split Link 0

Link 1

Link 1 is the arm of the robot. It extends from the shoulder (joint J2) and the elbow (joint J3) of the robot arm.

It is the biggest part of the robot. It can be realized as one piece only with a 3D printer having large build volume. Due to limitations on the building volume the part was split in two that are screwed together radially using six square nuts and M3 screws, similarly to Figure 3.27.

To make the robot more aesthetically pleasant the two bulges to house the stepper motor were placed on the same side so that if the robot is seen in the fully vertical outstretched position there is alternation of big and small bulges. This configuration is beneficial also for inertia distribution.

In this part there is the first non axial joint for which it's possible to see how the potentiometer mounting was modeled (Figure 3.28). A "U"-shaped flange is used to secure the potentiometer in the modeled slot. To block rotation of the transducer a groove is cut on the floor of the HD cover housing. To bring outside the potentiometer's wires there is a channel that exits from the part. The output flange of the HD for joint J2 is fastened to the link with four M3 screws.



Figure 3.28: J2 potentiometer mounting system

To mount joint J3 there are again five slots for square nuts. The part is presented in Figure 3.29.

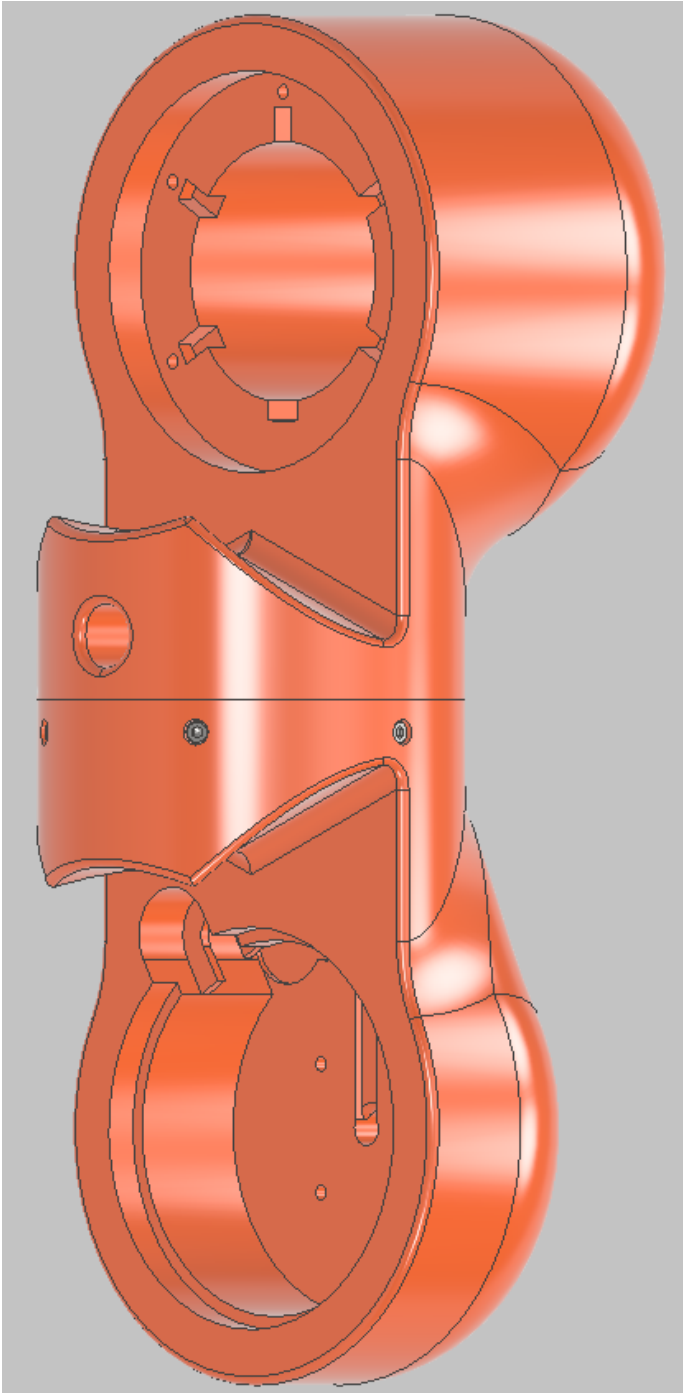


Figure 3.29: Assembled link 1

Link 2

Link 2 is the forearm of the robot. It is divided in two parts. The lower one is connected to the elbow (joint J3) of the robot arm and the upper part is free to rotate axially thanks the joint J4.

Inside this link is contained the joint J4, therefore it should be long enough to house the HD attached to the stepper motor. Another radial-to-axial coupler is used for joint J4. Two possibilities for the geometrical shape of link L2 were analyzed. It could be straight or "L"-shaped.

We were interested in having a spherical wrist, so the joint J4 must lay on the axis connecting joints J2 and J5, hence it must be straight. Using an "L"-shaped link permits to have a shorter link and hence to obtain a less limited range of motion but the spherical wrist is lost. Hence, the indicative maximum length of link L2 of 200mm (computed in the preliminary sizing phase) was increased to 270mm.

On the external surfaces of the two parts constituting the Link 2 there are feature to house the external potentiometer for rotation sensing of joint J4. The upper part has teeth embedded and flush with the surface. The lower part has a protrusion that is used to insert the potentiometer and fasten it in place with the provided nut. The mounting operation requires to insert the transducer in the railing structure on the lower part, screwing the nut to secure it but leaving it able to move radially. Then the small gear is mounted on potentiometer shaft and both can be moved radially to achieve the meshing of the two gears. Finally the nut can be fully tightened and a spacer can be inserted to assure positioning of the sensor. To assure that the potentiometer can't rotate the spacer and the support structure have a geometric feature that is mated with the corresponding one on the transducer.

Link 3

Link 3 was not fully designed because a specific task for the arm was not defined. Functionally it is identical to the lower part of link L2 and therefore it was used.

3.2.5. Wiring management

Wires exiting the links need to be grouped and routed towards the base. This task is accomplished externally guiding the cables from one link to the previous one with arcs.

The wire cordon must not be pinched or obstacle the arm. Hence, together with the wires a stiff element must be used to sustain the arc. At this scope a piece of filament for 3D

printers can be used. This piece of filament is held in place by one 3D printed piece at each end screwed on the links.

Finally, to protect the cable a nylon sleeve is wrapped around them, giving also a more pleasant looking to the robot.

Since the prototype realized was a very early stage one, cable management was not fully completed.

3.2.6. Modularity and Photo Gallery

The robot arm can be divided into modules that are independent one with the respect to the other. In total, five modules can be identified:

1. Module n°1 is composed by the robot base and the HD+motor of joint J1. See Attachments for technical drawing.
2. Module n°2 is composed by the Link 0 and the HD+motor of joint J2. It attaches to the HD of J1 by means of the axial-to-radial coupler. See Attachments for technical drawing.
3. Module n°3 is composed by the Link 1 and the HD+motor of joint J3. It attaches to the HD of J2. See Attachments for technical drawing.
4. Module n°4 is composed by the two halves of Link 2, HD+motor of joint J4 and HD+motor of joint J5. See Attachments for technical drawing.
5. Module n°5 is composed by the Link 3 and HD+motor of joint J6. Moreover, the tool can be considered in this module.

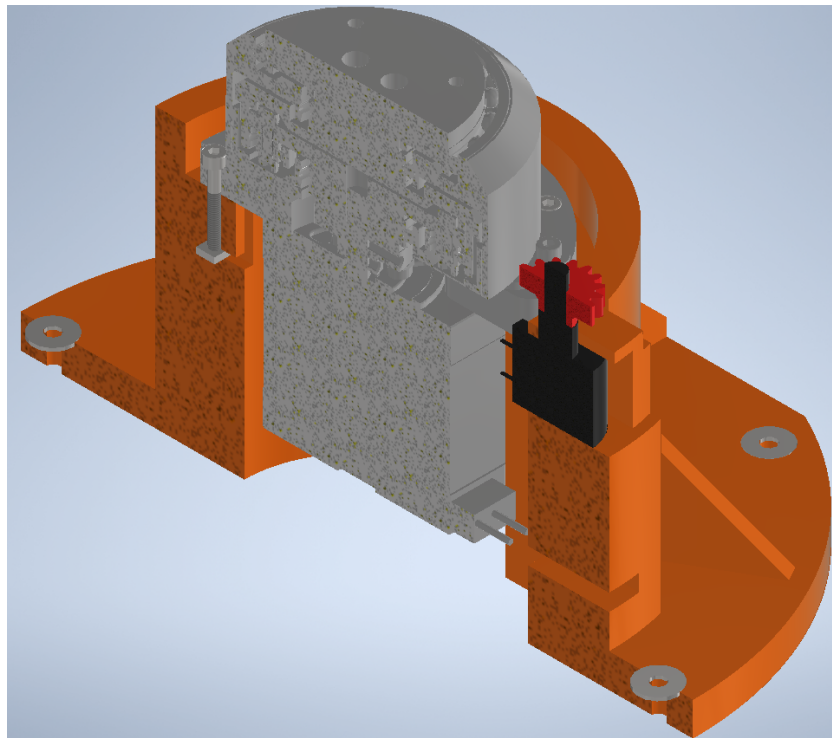


Figure 3.30: Module 1

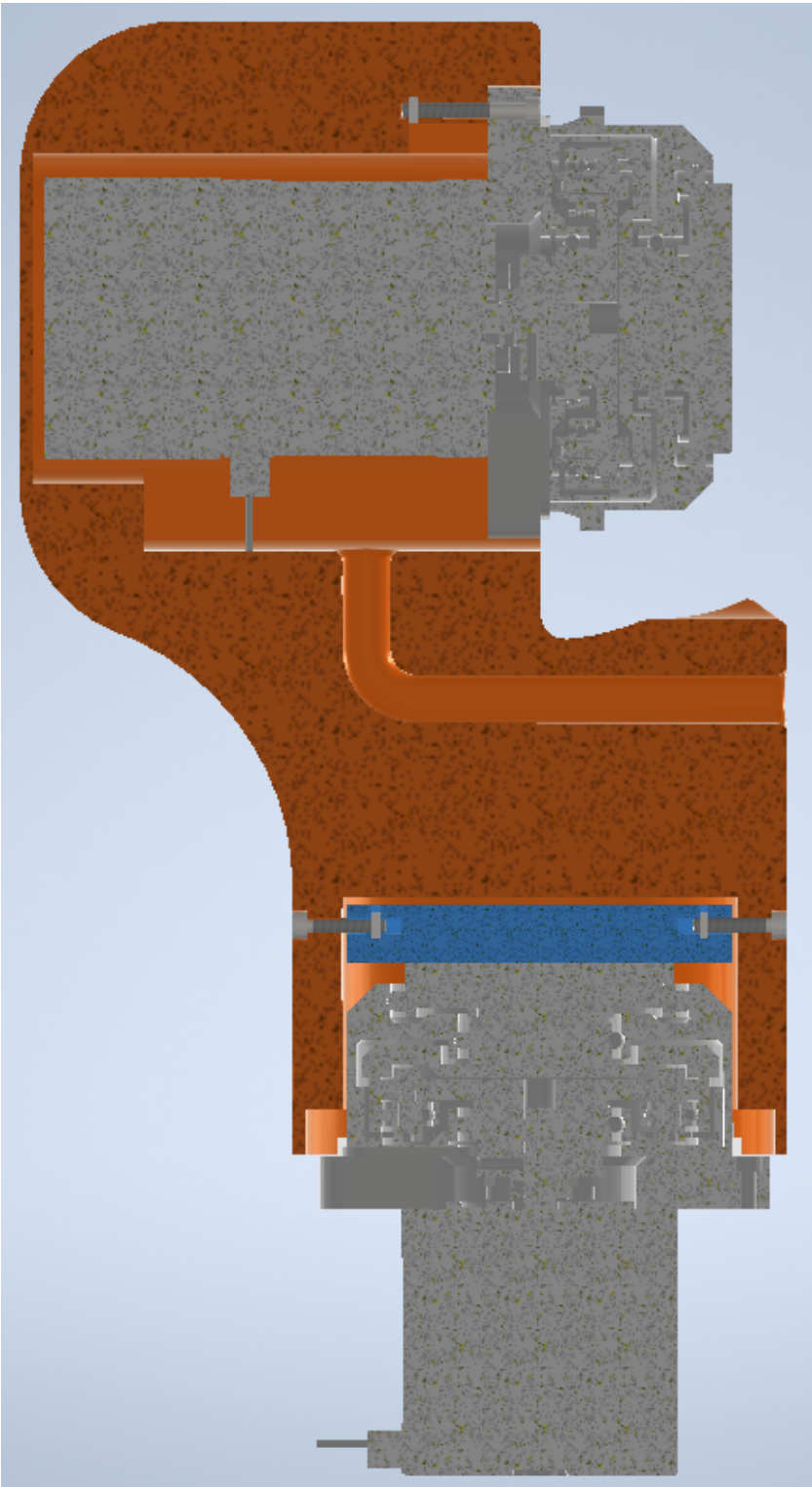


Figure 3.31: Module 2

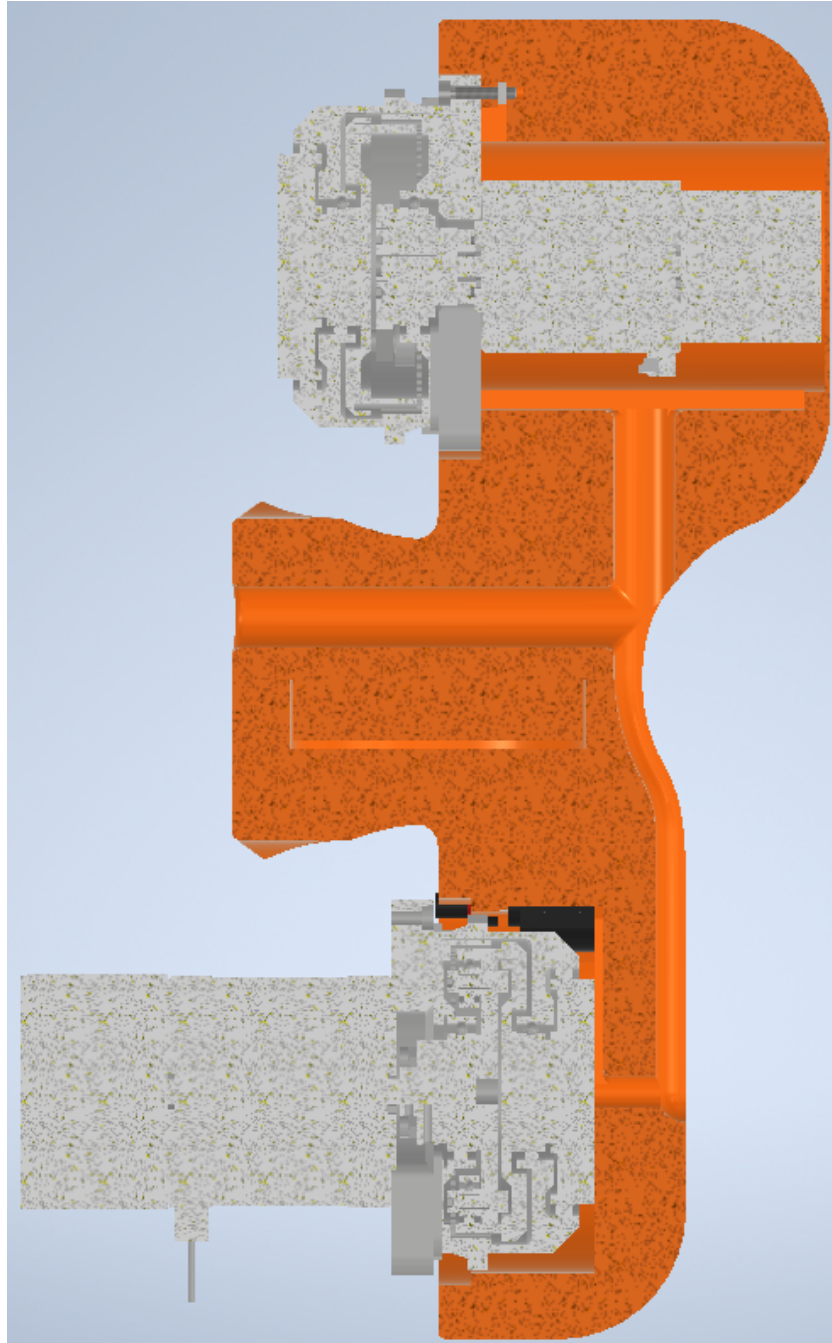


Figure 3.32: Module 3

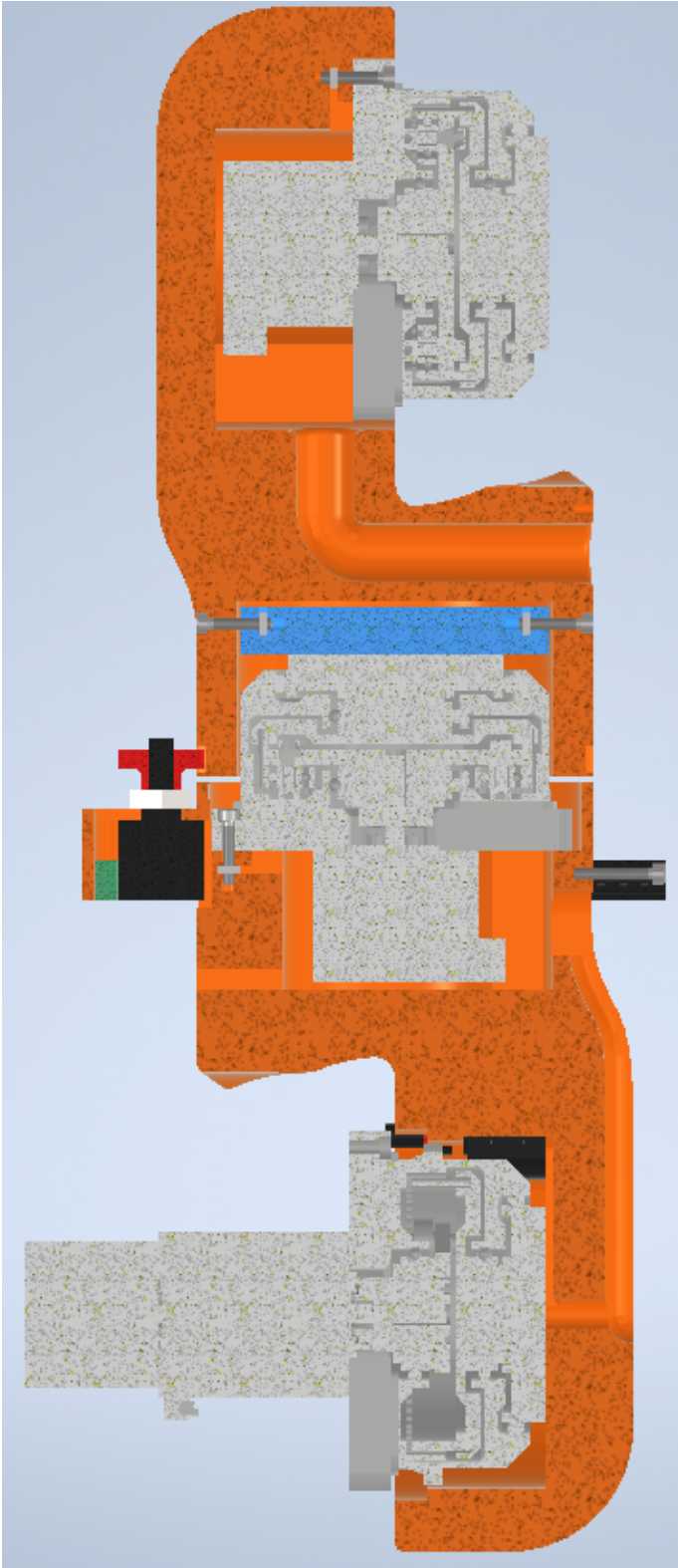


Figure 3.33: Module 4

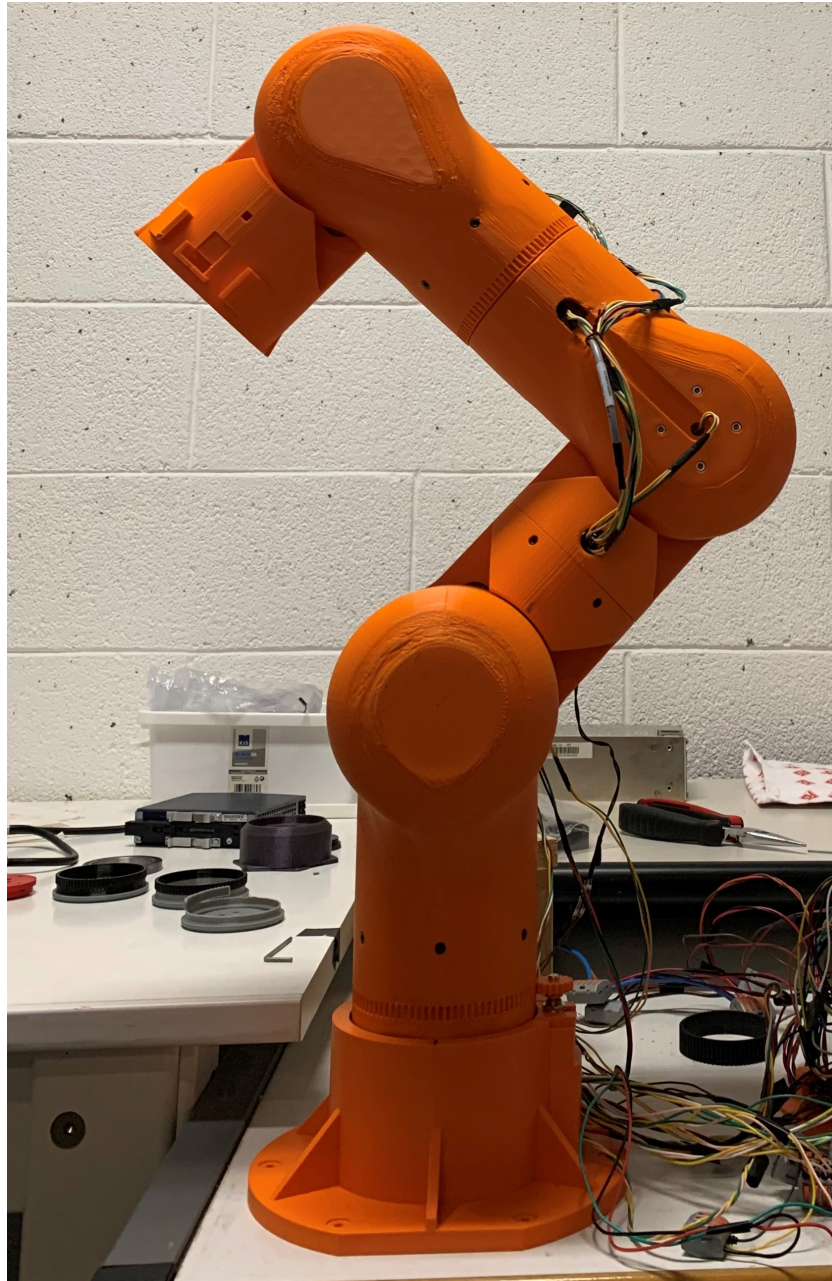


Figure 3.34: Real robot assembled

3.3. Manufacturing with 3D printing

3.3.1. Machines and Slicer Software

The machine used for the construction of the prototype was the 3ntr A4v4³. Actually, there were two available machines, and this allowed to reduce the total printing time. In particular, one machine was used to build the links that were very time consuming, while the other was used for the HD part and other small pieces. A summary of the printer characteristics is given in Table 3.3.

Parameter	Value
Actual build volume	$295 \times 200 \times 200\text{mm}^3$
Number of nozzles	3
Max nozzle temperature	450°C
Max chamber temperature	90°C
Mechanical precision	0.011mm
Nozzle diameter	0.4mm
Layer height (min/max)	0.1/0.6mm
Filament diameter	2.85mm

Table 3.3: 3ntr A4v4 characteristics

The multiple extruders functionality was exploited to print support material with a different polymer than the main part (namely SSU0), so to ease its removal. Given the presented characteristics, this machine is capable of printing in practice any polymer. In the manufacturer website are reported the following supported materials:

- ABS
- PETG
- ASA
- Elastomers
- Nylon
- Polycarbonate blends

The slicer software used was KISS (Keep It Simple Slicer) version 1.5⁴ in the PRO version that allows to manage multiple extruders. According to the website, this version of the slicer was deprecated in August 4, 2016. Indeed, if compared with other slicer software

³<https://3ntr.net/it/a4v4-stampante-3d>

⁴<https://www.kisslicer.com/>

(also addressed to consumers) functionalities of this software are quite basic and limited, complying with the "philosophy of simplicity" that gives the name to the software, but still showing that was developed many years ago. Among missing modern features there are:

- Custom supports: there is no possibility of placing support blockers neither support enforcers. The slicer chooses automatically where to place support material and the only free parameters left to the user are support density and distance from the object.
- Multi-material parts. It's not possible to manufacture a single part made of different materials, even if the machine has multiple extruders.

Due to agreements between the University and the seller, we could not choose another software for slicing and we could only use the materials provided by the latter. We needed to manufacture multi material parts and therefore another 3D printer was required. We had a Prusa MK3S+ by PrusaPrinters for personal use that was used for printing small parts and in particular the flexible spline of HD. Some characteristics are reported in Table 3.4.

Parameter	Value
Actual build volume	$250 \times 210 \times 210\text{mm}^3$
Number of nozzles	1
Max nozzle temperature	300°C
Heated chamber	NO
Max bed temperature	120°C
Nozzle diameter	0.4mm
Layer height (min/max)	0.05/0.35mm
Filament diameter	1.75mm

Table 3.4: Prusa MK3S+ characteristics

The slicer software used is PrusaSlicer. This software is a custom version of Slic3r developed by the manufacturer of the printer. It is a modern slicer with the most important function for the scope of this thesis work that is multi-material parts. Even if the machine has just one extruder (being an affordable prosumer 3D printer) it can simulate multiple extruders (virtual multi-extruder). The Prusa MK3S+ supports the G-code command M600 to change filament at a certain height of the model but, as it is, it performs just a color change considering the same material parameters. To use materials with different printing parameters (mainly nozzle temperature and printing speed) virtual multi-extruder must be set and use the M600 command as trigger for the virtual tool

change at a certain height.

PrusaPrinters also sell a device that allows the change filament type automatically called MMU2S⁵ that stands for Multi Material Unit, but was not at disposal. Actually, it wasn't needed since the material change occurs from a certain layer. The MMU is actually useful for multicolor parts that have different colors in the same layer.

3.3.2. Material selection

The material chosen for the construction of all the structural parts of the robot is ABS. This choice was taken due to the before mentioned agreements. Some typical characteristics of ABS material are reported in Table 3.5 [22].

Parameter	Value
Density	1.04 – 1.12g/cm ³
Young's modulus	2.28GPa
Tensile strength	43MPa
Glass transition temperature	105°C
Working temperature range	from –20°C to 80°C

Table 3.5: ABS properties

As it is possible to see, ABS has quite good mechanical properties that can be exploited to build strong and stiff components.

Indeed, to successfully print ABS some precautions must be taken, among which:

- A heated building plate is required to ensure that the printed part adhere to printing surface and so to limit warping. One of the main challenges when printing with ABS is to make the first layer adhere to the build plate because at is cools it tends to contract and edges start to peel from the the build surface. See Figure 3.35⁶
- ABS suffers layer delamination (cracking) caused by layer contraction during cooling, thus a heated chamber is required. Moreover, blowing cold air over the layer worsen this phenomenon.
- At high temperatures ABS starts decomposing into its constituents; butadiene is a carcinogenic substance; the others are possible carcinogenic substances. Therefore, an enclosed chamber with filtered air vents is mandatory for safety reasons.

⁵<https://www.prusa3d.it/original-prusa-i3-multi-material-2-0/>

⁶<http://3dp-engineering.com/why-3d-prints-warp/>

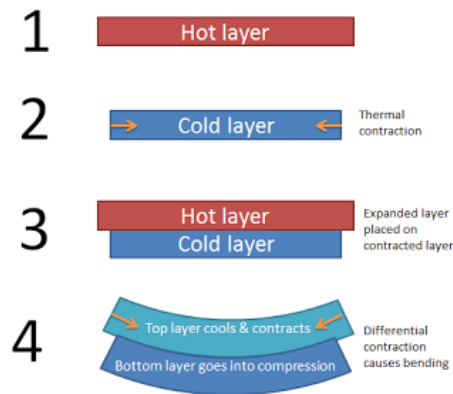


Figure 3.35: Warping and delamination mechanism

The choice of ABS is very valid for the available machine in the laboratory because it has all the characteristics to handle this material.

The robot arm has been designed to be printable also with small consumer 3D printers that in general lack of a sealed enclosure. So, ABS can become difficult to be used and hence it's useful to do a brief review of some other materials. The most used material is PLA (Poly Lactic Acid) because it is very easy to print (adheres well even on a non-heated build plate and doesn't suffer of layer delamination, at least for small pieces). However, there are drawbacks that prevent the use of PLA as a structural material. Even if it has high tensile strength and Young's modulus (so rigidity) it has a very brittle behavior and it is very hygroscopic, worsening its brittleness over time.

The real alternative to ABS for consumer 3D printers is PETG. Characteristics of this material are reported in Table 3.6 (data for PET material [23]⁷).

Parameter	Value
Density	$\sim 1.27\text{g}/\text{cm}^3$
Young's modulus	2.8 – 3.1GPa
Tensile strength	55 – 75MPa
Elastic limit	50 – 150%
Impact strength (notched)	$3.6\text{kJ}/\text{m}^2$

Table 3.6: PET characteristics

Looking to some manufacturer of filaments for 3D printing (for example the Prusa Prusament PETG filament⁸) is possible to see that PETG is very ductile (in some Charpy test the specimens printed flat on the build plate didn't even break) and has a higher tensile

⁷<http://www.matweb.com/search/DataSheet.aspx?MatGUID=4de1c85bb946406a86c52b688e3810d0>

⁸https://prusament.com/media/2020/01/PETG_TechSheet_ENG.pdf

strength than ABS and ASA⁹.

PETG is almost as easy to print as PLA but it is very ductile. The major drawback for printing is that this material is very sticky and a coated nozzle is recommended over the common bronze ones. PETG has higher density than ABS, so this means that the final parts will be heavier if the same settings are used.

3.3.3. Printing orientation

To guarantee the highest possible mechanical resistance, direction of deposition of the layers is of paramount importance. Any 3D printed object is the weakest where two layers touch, because layer adhesion is not perfect since molten material is deposited over already solidified plastic. Therefore, along the layer deposition direction (z -axis for all planar 3D printers) it's expected the lowest associated tensile strength. Hence the main load acting on the piece should not cause components of stress perpendicular to the layers (Figure 3.36¹⁰). For many materials also shear stresses along the layer are critical.

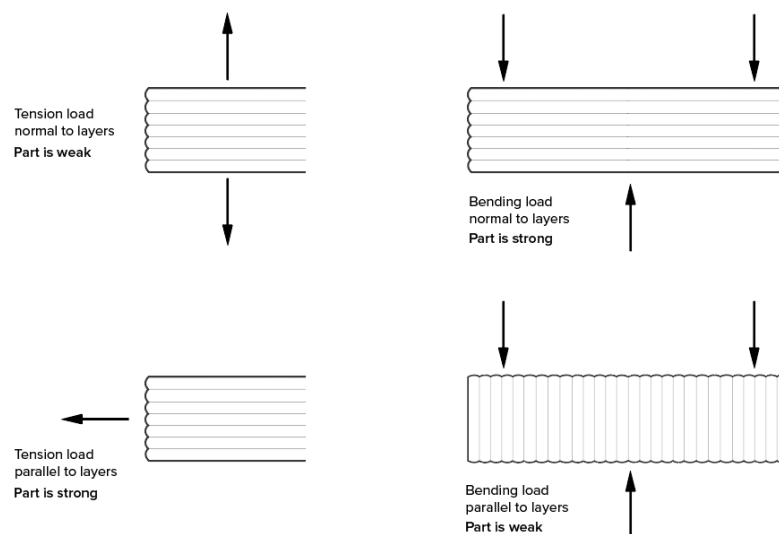


Figure 3.36: Load direction and layers

The main load on the robot arm is bending moment, thus printing direction should not be the same as the axis of the arm itself. The words "should not" are used instead of "must not" because sometimes it is necessary to overcome this general indication for a feasible manufacturing: the simplest example is to fit the part in the building volume. Another occasion for which the general indication is not followed is if there are critical features for which quality is more important.

⁹https://prusament.com/media/2018/09/ASA_DataSheet_ENG.pdf

¹⁰<https://www.hubs.com/knowledge-base/how-design-parts-fdm-3d-printing/>

3.3.4. 3D printing the flexible spline

As said in Section 3.2.1 the flexible spline of the HD was modified. The part is divided in two bodies to be printed with different material. The base of the flexible spline must be rigid to be connected with the output flange, hence a rigid material such as ABS or PETG must be used. The toothed profile of the flexible spline must be flexible to be deformed by the wave generator and TPU plastic with Shore hardness 95A was chosen. This level of hardness was selected because it can be easily sorted on the main eCommerce platforms and it is the easiest elastic filament to be printed, requiring minimum precautions.

It can be said that printing the toothed profile with ABS or PETG could be feasible, but thickness of the part should be minimum to guarantee deformability. Indeed, in this way structural resistance is compromised because the deposition direction causes shear stress along the layers and a short lifespan is expected if rigid polymers are used. Moreover, printing thin parts in ABS is critical due to delamination. PETG has better layer adhesion but the ideal material is TPU because it has even better layer adhesion.

3.3.5. Sacrificial features

For an easier manufacturing some features can be modeled in the CAD software that are removed once the part is completed. Some examples are:

- sacrificial layers to avoid the use of support material. These layers of plastic can be trimmed easily with a cutter or, if used for holes, can be removed with a small drill bit.
- brim to extend the contact surface of small features. The brim can be manually modeled to have full control on the width and where it is placed, since slicer software put a brim everywhere around the perimeter of the part.
- model a custom support structure to be realized as normal structure.

In fact, some sacrificial features have been used to overcome limitations imposed by the slicer software.

3.3.6. Solutions to tools limits

Since in the software there was no possibility to block generation of support material in some regions, to avoid support in the small slots for square nuts the support distance from the part was increased to half the width of the slot. This caused a reduction of quality of some outer surfaces where the overhang required support, but it was not generated

having altered the before mentioned distance.

In the base part the only required support was under the slot for the potentiometer (Figure 3.37). Hence it was modeled in CAD and printed with the same material as the part to avoid support generation in the square nut slots. A thin sacrificial layer was modeled under the "U"-shaped slot to facilitate bridging.

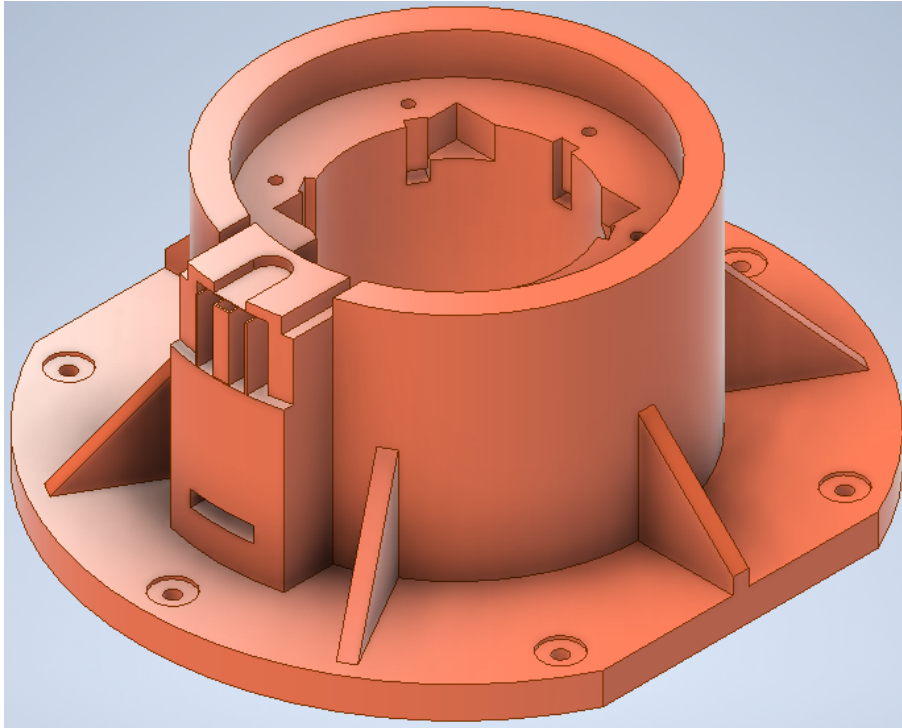


Figure 3.37: Base with CAD modeled support

3.4. Theoretical limits of the robot arm

Still considering the simplified 2R model, an assessment of the robot limits was done to find out a preliminary range of motion. A multibody simulation of the complete arm would give more accurate results but for sake of simplicity a reduced and simplified model is used.

Now it is possible to consider the actual mass of the links since the CAD geometry is available. Indeed, it is possible to obtain a mass value closer to the final one slicing the geometry of links L1 and L2 preparing the G-code and thus simulating the manufacturing result. Those masses are reported in Table 3.7. For safety mass is still increased of 5% to account for additional components as potentiometers, gearwheels and screws.

Link	Mass
L1	620g
L2	510g
L3	250g

Table 3.7: Masses estimated with Slicer software

Link L3 is still considered as cylinder but its mass is reduced to be more similar to the one of links L1 and L2. From the CAD software it is possible to obtain the value of the components of the inertia tensor. Anyway, manufacturing through 3D printing affects the internal structure of the workpiece (it is no more homogeneous) and thus the values indicated by the CAD are no more useful. A similar reasoning can be done for the center of gravity. Moments of inertia of cylinders are still considered, and the center of gravity is set in the middle of the links. Harmonic drives inertial characteristics can also be set with more accurate values. Mass can be directly measured once it is manufactured or estimated with the slicing software. It's important to notice that for the HDs bearings are the components that most affect the mass and therefore moment of inertia. The latter can be obtained from the CAD model. Anyway, to keep the model as simple as possible inertia of cylinders is still considered.

As said, for modeling constraints the limit lengths computed were exceeded. In particular the length of link 2 was significantly increased.

Limits of interest to be investigated are:

1. whether the arm can bear the fully outstretched horizontal.
2. minimum starting angle for a swing up in the 1st quadrant to reach 90deg. An angle as close as possible to 0deg is desired to have the biggest possible workspace.

3. whether it's possible to move the arm in 4th quadrant and amplitude of a symmetric swing from 4th to 1st quadrant. The biggest range of motion is $[-90, 90]$ deg but due to construction limitations it will be reduced to avoid self-collision between links L0 and L1.
4. angle in the first quadrant for which the motor on J2 can bear maximum speed and acceleration at same time. An angle as close as possible to 0deg is desired.

Limit 1

The arm is simulated in the fully outstretched horizontal position and the torques required to J2 and J3 motors are computed. Acceleration is set to the maximum value. Deliverable torques are computed with the simplified datasheet torque vs. speed curves, considering the null speed torque. Hence, a sudden departure from still is simulated.

Result of this test tells that the robot can withstand the loads. The required torques compute for motors of joints J2 and J3 are respectively 0.5648Nm and 0.2924Nm while the torque deliverable at zero speed are respectively 0.7Nm and 0.38Nm.

Limit 2

As said many times, the most critical condition for the arm is an upward movement in the first quadrant since inertial loads sum up with gravity. If the arm is capable of doing a 90deg swing starting from 0deg then the range of motion is maximum.

The feasible starting angle is searched iteratively using a vector that starts from 0deg. Links L2 and L3 are kept horizontal to maximize the lever arm for inertial loads and weight forces. The iterative computation ends when the computed torque profile is always smaller or equal (so tangent as limit condition) to the deliverable torque curve (Figure 3.38). The preliminary result is that the minimum angle searched is indeed 0deg.

Limit 3

Similarly to what is done for the second limit, the starting angle is searched iteratively using a vector starting from -90 deg. Again, links L2 and L3 are kept horizontal to maximize the lever arm for inertial loads and weight forces. The iterative computation ends when the computed torque profile is always smaller or equal (so tangent as limit condition) to the deliverable torque curve (Figure 3.39).

The preliminary result is that the minimum angle searched is indeed -90 deg.

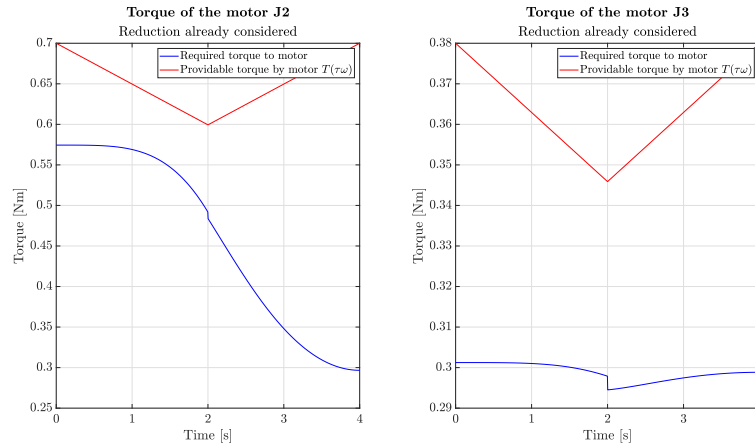


Figure 3.38: 90deg swing starting from 0deg torque curves

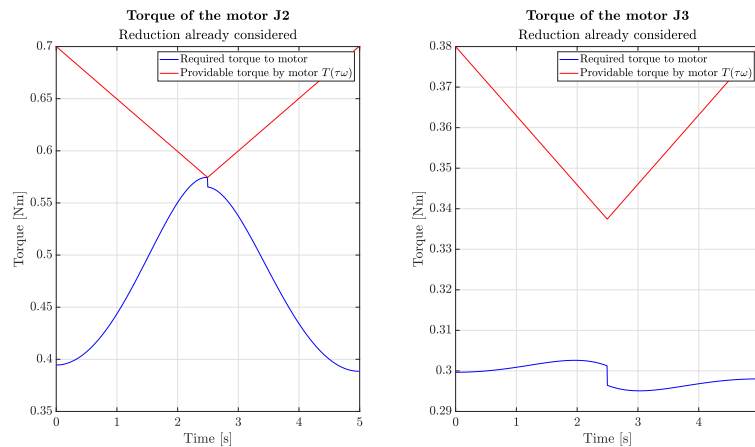


Figure 3.39: Symmetric swing up form 4th to 1st quadrant

Limit 4

The robot arm is simulated with the link L1 angled while links L2 and L3 are kept horizontal. Maximum speed and acceleration are set at the same time.

This limit becomes useful when programming a movement in the first quadrant since it may be that maximum speed and acceleration happen in a angle smaller than this limit. Indeed, this situation is very unlikely to happen because very high speed are reached only with very wide required rotations.

The required torque to motor on joint J2 is computed for values of β angle from 0deg to 90deg. The angle for which the required torque equals the deliverable one at max speed is found (Figure 3.40). This angle is indeed 0deg.

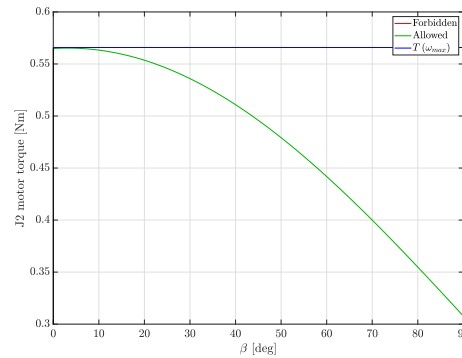


Figure 3.40: Limit β angle

Summing up

Summarizing, preliminary assessments of the limits of the arm suggest that:

- the arm can accelerate with maximum acceleration upwards starting from a fully outstretched horizontal pose.
- the robot can assume a pose with vertical arm and horizontal forearm starting from fully outstretched horizontal pose.
- the arm can do a symmetric swing up between 4th and 1st quadrant in a significantly wide range ($[-90, 90]$ deg), assuming fully outstretched horizontal pose with high speed.

4 | Electronics and Control

4.1. Electronics

4.1.1. Stepper driver description and characteristics

To actuate the stepper motors, stepper drivers were used. The stepper driver boards used were given free of charge from R.T.A. They are BSD 02.V boards. According to the manufacturer specsheet, the main characteristics are reported in Table 4.1. The pinout of the board is reported in Figure 4.1.

Parameter	Value
Nominal supply voltage range	24 – 48V
Min/Max phase current	0.7 – 2.2A
Max stepping frequency	60kHz (50% duty cycle wave)
Microstepping	400-800-1600-3200

Table 4.1: BSD 02.V characteristics

Number of steps per revolution (microstepping) can be adjusted with dip switches on the board.

These boards are controlled as any other stepper driver with two wires, one for the turning direction (DIR pin), the other for the step generation (STEP pin), that are placed on terminals 4 and 3 respectively. Given that the robot has 6 axes, to control all of them 12 wires are required.

Terminals from 12 to 15 are for motor phases wires. Phases have been arranged so that a positive rotation angle requested makes the link rotating in counterclockwise direction. This choice was done to comply with the convention of a right-hand reference system.

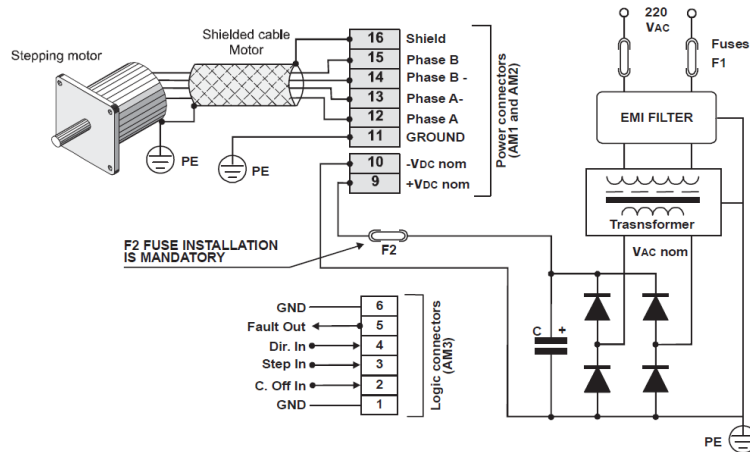


Figure 4.1: BSD 02.V connections

4.1.2. Other electronics

A list of other electric and electronic items is reported:

- 24V power supply. It is used as the only power source for the robot. The voltage is set by the motors. The nominal power must be higher than the total power required by all the motors and every else electronic subsystem of the robot.
- 2 relays for the brakes on J2 and J3 motors. Since these motors have a brake to hold position when there's no power, two relays are needed to disengage the brake at start. A module with at least two relays is needed. If more relays are present they can be exploited for the tool on the end effector.

4.1.3. Selection of the microcontroller

A microcontroller is needed to generate the signals for the stepper driver boards. Requirements for the microcontroller are:

- To have enough digital output pins to control all stepper drivers. As seen, it must have at least 12 digital outputs. Moreover, two additional digital pins are needed for the relays. The total is therefore 14 digital outputs required.
- To have an ADC converter built in or be compatible with external ADCs (communication protocols must be compatible). This is for sensing the analog signals from the potentiometers, one for each joint, therefore at least 6 analog inputs are necessary.
- Be fast (have high clock speed) to guarantee good performance and response time.

A list of all the discussed microcontroller is presented.

- Arduino UNO. It is the simplest and cheapest (not genuine board) available option and the microcontroller used for the preliminary testing. Some characteristics are reported in Table 4.2.

Parameter	Value
Microcontroller	ATmega328P
Clock speed	16MHz
Operating voltage	5V
DC current per I/O pins	20mA
Digital input/output (of which PWM capable)	14 (6)
Analog input pins (ADC resolution)	6 (10bit)
Programming language	C++
Flash memory	32KB
SRAM	2KB
EEPROM	1KB
Cost (original board)	20€ + VAT

Table 4.2: Arduino UNO characteristics

Hence, an Arduino UNO board is capable to control up to 5 motors (because two pins must be reserved to the relays) and read all 6 potentiometers signals. Pins 0 and 1 must not be used for generation of the signals for the stepper drivers because are used for communication (for instance with I2C protocol) with the ADC board or other slave peripherals. Arduino UNO is programmed with the Arduino IDE.

- Arduino MEGA. An Arduino MEGA can be used as an upgraded alternative to the UNO since it has more digital output pins. Some characteristics are reported in Table 4.3.

Parameter	Value
Microcontroller	ATmega2560
Clock speed	16MHz
Operating voltage	5V
DC current per I/O pins	20mA
Digital input/output (of which PWM capable)	54 (15)
Analog input pins (ADC resolution)	16 (10bit)
Programming language	C++
Flash memory	256KB
SRAM	8KB
EEPROM	4KB
Cost (original board)	35€ + VAT

Table 4.3: Arduino MEGA characteristics

As one can see it is possible to control all the 6 stepper drivers and the relays. Arduino MEGA is programmed with the Arduino IDE.

- Raspberry Pi Pico. It's the smallest and simplest board developed by the Raspberry organization. Differently from their other products this board it's a microcontroller and not a minicomputer since it cannot run an operating system. Some characteristics are reported in Table 4.4.

Parameter	Value
Processor	ARM Cortex-M0+ dual core
Clock speed	133MHz
Operating voltage	3.3V
DC current per I/O pins	< 50mA
Digital input/output	23
Analog input pins	3
Programming language	C++, MicroPython
Flash memory	2MB
SRAM	264KB
Cost (original board)	4€

Table 4.4: Raspberry Pi Pico characteristics

- Teensy 3.6. It's one of the fastest microcontroller boards available on the market. It was taken into consideration because it is the only one that can theoretically assure real time operation of the robot. Some characteristics are reported in Table 4.5.

Parameter	Value
Processor	ARM Cortex-M4 dual core
Clock speed	180MHz
Operating voltage	3.3V
DC current per I/O pins	< 50mA
Digital input/output (of which PWM capable)	60 (22)
Programming language	C++
Flash memory	1024KB
SRAM	256KB
EEPROM	4KB
Cost (original board)	29.25\$ + VAT

Table 4.5: Teensy 3.6 characteristics

This board is programmed with Arduino IDE with the Teensyduino add-on.

An Arduino UNO was chosen as the controller for the preliminary testing of stepper driver boards and motors attached to the HD speed reducers. Then an Arduino MEGA was used to have all the necessary outputs.

4.1.4. Analog to digital converter (ADC) sizing

The target is to measure the rotation of a joint as a consequence of the rotation of one step of the stepper motor. The motors available are built with $200^{\text{steps/rev}}$ corresponding to $1.8^{\text{deg/step}}$. Harmonic drives have a reduction ratio $i = 35$, hence one step of the motor corresponds to $\frac{1.8}{35} \approx 0.05^{\text{deg/step}}$ of rotation of the output flange of the HD. Multiplication of rotation is instead achieved for the gear on the potentiometer (see Section 3.2.2).

The transducer can be supplied with 5V (Arduino boards) or 3.3V (Raspberry Pi Nano boards, Teensy 3.6). We used an Arduino MEGA and hence the range of voltage for the measure is $0 \div 5\text{V}$. The sensitivity (V/deg) is $\frac{5\text{V}}{5.360^{\text{deg}}} \approx 2.78^{\text{mV/deg}}$, meaning that 1deg of rotation of the potentiometer shaft corresponds to 2.78mV output of the potentiometer. The ADC is sized for the smallest τ value obtained in Section 3.2.2 since it causes a smaller resolution. In particular the resolution required is $\tau \cdot 0.05^{\text{deg/step}} \cdot 2.78^{\text{mV/deg}} \approx 0.61^{\text{mV/step}}$. Resolution required for the ADC is given by Eq.4.1 where N is the number of bits.

$$\Delta V = \frac{5\text{V}}{2^N} \rightarrow \Delta V < 0.61\text{mV} \Rightarrow N \geq 14 \Rightarrow \Delta V = 0.305\text{mV} \quad (4.1)$$

The number of bits required for the ADC is bigger than the one of Arduino boards (10bit). Hence, an external ADC is needed having at least 14bit resolution. More precisely, six ADCs modules or one module with at least 6 channels is needed.

As it is possible to see in Table 4.1, in reality the minimum number of steps per revolution is 400. Hence, to see half a step of the motor the resolution required is half the one computed few lines above, precisely $0.305^{\text{mV/step}}$. This is the same value computed in Eq.4.1. Therefore more than 14bit are required.

Searching on one of the most famous eCommerce, ADS1115 ADC where found (Figure 4.2). This board has 4 channels and a resolution of 16bit that corresponds to $\Delta V = \frac{5\text{V}}{2^{16}} = 0.0076\text{mV}$ that is more than enough to measure even half a step of the motor. Two ADS1115 boards are required to measure rotation of all six joints. The spare two channels can be used for other sensors on the end effector.



Figure 4.2: ADS1115

4.1.5. Potentiometer wiring

Looking to Bourns data-sheet for the potentiometer code 3548 Single Gang, Bushing Mount, pinout is as in Figure 4.3.

Since we set counterclockwise rotation direction positive for the links, potentiometer shaft rotates in clockwise direction being actuated by a gear system. Therefore potentiometer pins must be connected to Arduino as in Table 4.6.

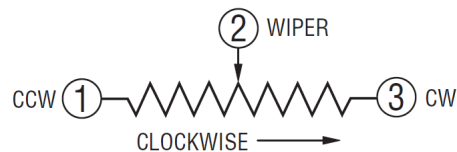


Figure 4.3: Potentiometer pinout

Potentiometer Pins	Function	Arduino Pins
1	POWER	GND
2	SIGNAL	Analog Pin (A)
3	POWER	5V

Table 4.6: Potentiometer wiring

4.2. Control

4.2.1. Strategies for actuating the stepper motors

To actuate a stepper motor driven by a stepper driver board two pins are required. One pin sets the direction of rotation and it is kept at high (5V) or low (0V) state. To perform a step a square wave signal must be provided to the other pin. The step is performed when the signal goes from high to low. Rotation speed of the motor is determined by the

frequency (and so the period) of the square wave. It's possible to generate a square signal with the very simple Arduino code in Listing 4.1.

```
1 void loop() {
2     digitalWrite(myPin, HIGH);
3     delay(some_milliseconds);
4     digitalWrite(myPin, LOW);
5     delay(some_milliseconds);
6 }
```

Listings 4.1: Interrupting square wave

The `delay()` functions completely block the microcontroller so no operation can be performed while the microcontroller waits. This is therefore a blocking strategy.

Another way, slightly more complex, is to use a non-blocking Arduino code as in Listing 4.2.

```
1 unsigned long previousMicros = 0; //[us]
2 void loop() {
3     unsigned long currentMicros = micros();
4     if((unsigned long)(currentMicros - previousMicros) >=
5     wave_period){
6         digitalWrite(myPin, !digitalRead(myPin));
7         previousMicros = currentMicros;
8     }
9 }
```

Listings 4.2: Non interrupting square wave

Summarizing, the microcontroller keeps track of time with the `micros()` function and performs the action in the `if()` statement only when the elapsed time is greater than the set period. By keeping track of the elapsed time in the variable `previousMicros` it is possible to generate a square wave. The state of the pin (high or low) is changed reading its current state using `!digitalRead(myPin)`.

4.2.2. Preliminary testings

Synchronized movement

The first thing we wanted to achieve is a synchronized movement of all the robot joints. Preliminary actuation of the robot motors is done moving them synchronously with a

constant acceleration motion law (triangular or trapezoidal speed profile). As said, an Arduino UNO microcontroller was used for the preliminary testing then we switched to an Arduino MEGA. Already implemented libraries can be exploited to make easier and faster the task. On the library manager for Arduino a useful library was found¹. This library is called *StepperDriver* and was implemented by Laurentiu Badea.

A *BasicStepperDriver* object can be defined that is associated with the pins for direction and step on the Arduino board. Moreover, it requires the number of steps per revolution of the motor. The defined object must be "activated" in the `setup()` loop that is run only once. The available methods (= functions) allow to easily set rotation angle and speed. Listing 4.3 shows what just described with words.

```
1 // Include library
2 #include "BasicStepperDriver.h"
3 // Define pins on Arduino board
4 const int pinDir= 2;
5 const int pinStep = 3;
6 // Define step per revolution
7 const int step_per_rev = 200;
8 // Define stepper driver object
9 BasicStepperDriver myStepper(step_per_rev, pinDir, pinStep);
10 void setup() {
11   myStepper.begin();
12 }
13 void loop() {
14   // Set speed in RPM
15   myStepper.setRPM(mySpeed);
16   // Request rotation of the specified angle
17   myStepper.rotate(myAngle);
18 }
```

Listings 4.3: Definition of a stepper driver object

To control multiple stepper driver at a time it's possible to define a *MultiDriver* object or a *SyncDriver* object that collect all the *BasicStepperDriver* objects of the system to actuate. The difference between a *MultiDriver* and a *SyncDriver* object is that with the first each motor moves independently, while for the latter motors move synchronously. Only synchronous movement of all the joints is of interest, hence

¹<https://www.arduino.cc/reference/en/libraries/stepperdriver/>

a `SyncDriver` object is defined. An example of definition of two stepper driver to be controlled synchronously is given in Listing 4.4.

```
1 // Include libraries
2 #include "BasicStepperDriver.h"
3 #include "SyncDriver.h"
4 // Define pins on Arduino board
5 const int pinDir1 = 2;
6 const int pinStep1 = 3;
7 const int pinDir2 = 4;
8 const int pinStep2 = 5;
9 // Define step per revolution
10 const int step_per_rev = 200;
11 // Define stepper driver objects
12 BasicStepperDriver myStepper1(step_per_rev, pinDir1, pinStep1);
13 BasicStepperDriver myStepper2(step_per_rev, pinDir2, pinStep2);
14 // Define the synchronized controller
15 SyncDriver myController(myStepper1, myStepper2);
16 void setup() {
17     myStepper1.begin();
18     myStepper2.begin();
19 }
20 void loop()
21 {
22     // Rotate the two motors synchronously
23     myController.rotate(myAngle1, myAngle2);
24 }
```

Listings 4.4: Definition of two synchronized stepper drivers

This library allows to control up to 3 stepper motors, thus the library needed to be modified to bring the number of controllable motor up to 6. This task was easy and simply performed opening the source C++ code and respective header .h files in an editor like Code::Blocks. With a find-and-replace operation in each defined method the number of defined stepper driver objects, angles, speeds and other quantities was brought to 6. A simple script was written in which some rotation angles were set together with speeds.

The main concern with this library is that it uses a blocking strategy to generate the square wave signal for the step pin. Thus, while the motors are rotating it's not possible

to have the microcontroller performing other tasks. Anyway both Arduino UNO and MEGA succeeded in moving all six motors.

Another possibility of the used library is to set the type of speed profile for the stepper motor. It is possible to choose between constant speed profile and linear speed profile. In Listing 4.5 is reported the method to set the speed profile.

```
1 // Include library
2 #include "BasicStepperDriver.h"
3 // Define pins on Arduino board
4 const int pinDir= 2;
5 const int pinStep = 3;
6 // Define step per revolution
7 const int step_per_rev = 200;
8 const int accel = 1000; // [step/s^2]
9 const int decel = 1000; // [step/s^2]
10 // Define speed profile type
11 #define mode BasicStepperDriver::LINEAR_SPEED
12 // OR
13 // #define mode BasicStepperDriver::CONSTANT_SPEED
14 // Define stepper driver object
15 BasicStepperDriver myStepper(step_per_rev, pinDir, pinStep);
16 void setup() {
17     myStepper.begin();
18     myStepper.setSpeedProfile(mode, accel, decel);
19 }
20 void loop() {
21     // Set speed in RPM
22     MyStepper.setRPM(mySpeed);
23     // Request rotation of the specified angle
24     MyStepper.rotate(myAngle);
25 }
```

Listings 4.5: Definition of speed profile

It's important to say that one has to provide acceleration as $steps/s^2$. To convert angular acceleration into the required units of measure one has to know how many steps per revolution are set. The minimum number of steps per revolution that can be set on the stepper driver boards is $400^{set}/_{rev}$. Therefore:

$$\frac{step}{s^2} = \frac{rad}{s^2} \times \frac{180deg}{\pi rad} \times \frac{1rev}{360deg} \times \frac{400step}{1rev}$$

Constant speed profile would correspond to a rectangular speed profile that translates in infinite acceleration. In reality constant speed means that the stepper motor is accelerating with its maximum acceleration. The library considers constant speed as default and it is not required to define acceleration and deceleration values. According to RTA datasheets, NEMA17 stepper motors have a maximum acceleration of $69000\text{rad/s}^2 = 3953408\text{deg/s}^2$ while for NEMA23 stepper motors $38500\text{rad/s}^2 = 2205887\text{deg/s}^2$, that is practically infinite and the speed rising phase can be considered vertical. Linear speed profile corresponds to triangular or trapezoidal speed profile. The acceleration value for the links was set to 22.5deg/s^2 that translates to:

$$22.5 \frac{deg}{s^2} \times \frac{400step}{360deg} \times 35 = 875 \frac{step}{s^2}$$

The next step is to be sure that the requested rotation angle and speed are actually performed.

Rotation angle assessment

A preliminary assessment of the accuracy of the rotation angle of the HD was performed. We measured the rotation angle in open loop of an HD with no load. “Open loop” means that one angle is requested to the HD and no feedback is produced by means of a potentiometer. The experimental setup is very simple: on the output flange of the HD a 3D printed indicator was mounted exploiting the holes for the robot links and a sheet of paper with a goniometer printed on it was used to read angle actually performed by the drive. Uncertainty on the read numbers is of $\pm 1\text{deg}$ since the goniometer has 1deg steps and the indicator has an amplitude of almost 1deg .

The code used is reported in Appendix A.1. It is possible to see that the test was automated and that the HD is required to go back to the initial position after each angle to be tested. In this way errors of each run are accumulated. A total of 8 repetitions of the test were done, 4 using a constant speed profile (maximum acceleration of the stepper motor) and 4 with linear speed profile setting the maximum acceleration for the robot links (875step/s^2). Results obtained with the two types of speed profiles were averaged.

In Figure 4.4 is reported the trend of the averaged tests together with the ideal behavior. As it possible to see the accuracy is quite good since the error bars are very close to

the ideal trend. An indication on repeatability is given comparing the error and the cumulated one: if the two are similar (laying on the first quadrant bisector) means that a new measure is affected by previous errors. As it is possible to see in Figure 4.5 this is not the case being the points really scattered. Anyway, since a position feedback system is designed this is not a big problem.

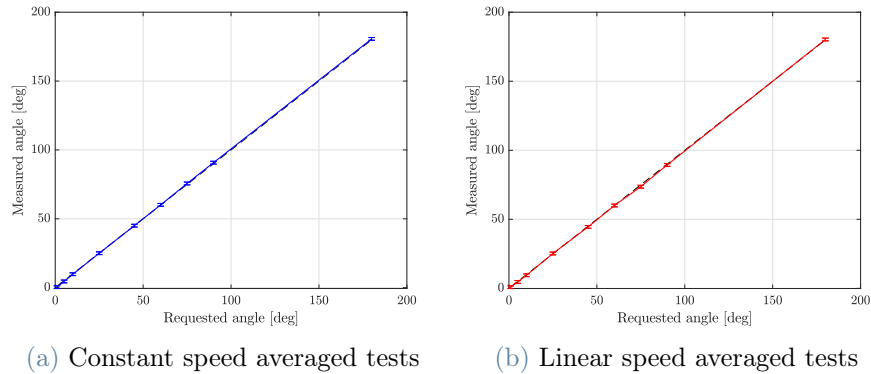


Figure 4.4: Rotation angle assessment trends

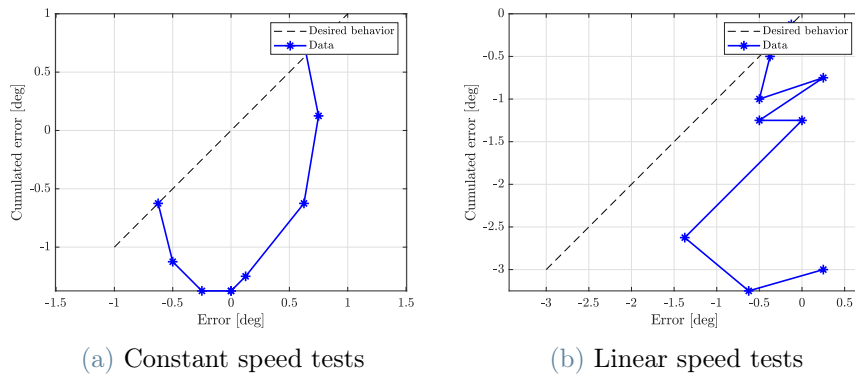


Figure 4.5: Error vs. Cumulated error

Rotation speed assessment

We measured the rotation speed of the joint J1 for different requested speeds by measuring the time required to perform one full revolution. We marked the initial position on the base and on Link 0 of the robot and using a stopwatch the time was measured. Since the operation was manual, starting and stopping of the stopwatch are affected by the reaction time of the person. Therefore, the reaction time of the writer was estimated using some online tools and resulted to be on average 0.3068s. The measured times were

then corrected by subtracting two times such reaction time. In Table 4.7 results of the test are reported. Considered the very imprecise method used, we can state that the used library is quite reliable in terms of speed. The code used is reported in Appendix A.2.

Requested speed	Measured time	Corrected time	Measured speed
1RPM	60,55s	59.9364s	1.0011RPM
2RPM	30.26s	29.6464s	2.0239RPM
4RPM	15.54s	14.9264s	4.0197RPM
5RPM	12.42s	11.8064s	5.0820RPM
8RPM	7.96s	7.3464s	8.1673RPM
9RPM	7.17s	6.5564s	9.1514RPM
10RPM	6.53s	5.9164s	10.1413RPM

Table 4.7: Speed assessment results

4.2.3. Rotation sensing

Arduino ADC

A first tentative for measuring the rotation of the joints was done using the internal ADC of the Arduino MEGA used for actuating the motors. Analog to Digital conversion is performed by the Analog Pins on the Arduino boards, identified with a capital A in front of the pin number. The function to perform an analog acquisition is `analogRead()` that receives as input the analog pin number on which perform the measure. The Arduino ADC has 10bit resolution and returns a number in the range 0-1024. It's important to define all the constants for the conversion of the measure in the desired units of measure. The most important are the resolution (in V/bit) of the ADC and sensitivity of the potentiometer (in V/deg). To obtain the measure of rotation of the link rather than the potentiometer shaft one, the transmission ratio of the gearing system must be used, contained in the `tau_gear` constant. Since the target is to see 0V in the position 0deg and considered that the range of voltage for the measure is $0 \div 5V$, 2.5V should be subtracted from the converted voltage measure to have values in range $-2.5 \div 2.5V$. In this way the angular position will be as desired.

Anyway, during the assembly task, the 0deg position can change due to misalignment of the gears. Therefore it's required a setup procedure every time a link is mounted. The procedure is reported in the following list and the Arduino code in Listing 4.6. Results are displayed on the serial monitor.

1. Rotate the potentiometer shaft till 2.5V is displayed on the serial monitor.
2. Mount the potentiometer in its housing and insert the 3D printed gear, trying not to rotate its shaft. If necessary adjust the shaft rotation to have 2.5V displayed.
3. Mount the link on the joint aligning the reference marks. The voltage displayed will change slightly because the gears will align themselves. For the potentiometers on Joint 1 and 4 meshing of the gears is performed manually (since they are pushed radially) but what said still holds.
4. Annotate the new voltage displayed and set the `home_volt` constant to that value.
5. Annotate the value displayed of `current_position` variable and set the `home_offset` constant to this value.

```
1 // Setting PINs
2 // A0 -> J1 potentiometer
3 // A1 -> J2 potentiometer
4 // A2 -> J3 potentiometer
5 // A3 -> J4 potentiometer
6 // A4 -> J5 potentiometer
7 const int potPin = A4;
8 // Setting constants
9 const int tau = 35; // HD reduction ratio
10 const float tau_gear_non_axial = 75/17; // Gear reduction ratio
11 double resolution = 5/pow(2,10); //[V/bit]
12 double sensitivity = 360; //[deg/V]
13 // Potentiometer setup constants
14 double home_volt = 0; //[V]
15 double home_offset = 0; //[deg]
16
17 void setup() {
18 // Setup Serial communication
19 Serial.begin(9600);
20 }
21
22 void loop() {
23 // Perform acquisition
24 double current_reading = analogRead(potPin); // in range [0
    1023]
25 // Do conversion
26 double volt = current_reading * resolution - home_volt;
27 double current_position = volt * sensitivity /
    tau_gear_non_axial - home_offset; //[deg]
28 // Display results
29 Serial.print(volt); Serial.print(" V"); Serial.print(" ");
30 Serial.print(current_position); Serial.println(" deg");
31 delay(100);
32 }
```

Listings 4.6: Potentiometer setup procedure

ADS1115 ADC

To read values with the external ADC board ADS1115 a different approach is needed. The A/D conversion is performed by the board and data is sent through I^2C communication protocol. Therefore, SDA e SCL dedicated pins on the Arduino MEGA board must be used. We referred to the online guide² by Adafruit for wiring and knowledge. Adafruit provides also an Arduino library that makes the whole process easier.

4.2.4. PID control

As first tentative we wanted to control the complete arm only using the Arduino MEGA micro-controller.

We thought about implementing a PID controller running on the Arduino. The target is to control in position all the joints, reaching a set of constant angular positions, one for each axis. The quantity that is fed back is clearly the angular position of each link measured by means of the potentiometers. The control action computed as Eq.4.2 is then translated in the angle of rotation requested to each stepper motor.

$$u(t) = K_p e(t) + K_d \frac{de}{dt} + K_i \int_0^t e(\tau) d\tau \quad (4.2)$$

A discrete time PID algorithm is needed because we are dealing with a digital device. To implement a discrete time PID controller a loop executed with constant frequency is required. It is relatively simple to implement a clock with the Arduino using the same non blocking algorithm presented in Listing 4.2 for the square wave generation.

Anyway, a problem is immediately evident: the library used to control and synchronize the stepper drivers doesn't allow to realize the clock because it blocks the Arduino. In fact, the instruction to perform a rotation (`myStepper.rotate(myAngle)`) must be finished before the micro-controller can proceed with further instructions. In other words, any rotation requested would take longer to complete than the clock of the PID controller, making it unfeasible because any new control action after the first one issued cannot be executed at the correct time.

With a non blocking strategy to actuate the stepper drivers instead it would be possible to implement a proper discrete time PID controller placing in the `loop()` two independent `if()` blocks. One is for the PID controller clock that every fixed time acquires the current position and computes the error and the new control action. The other one is

²<https://learn.adafruit.com/adafruit-4-channel-adc-breakouts/>

for the generation of the square waves for actuation of the motors and it uses the control action computed by the other `if()`. Clearly, the controller frequency must be smaller than the one of the square wave that corresponds to the maximum speed for the motor. In Listing 4.7 the algorithm and a pseudo Arduino code are reported.

```

1 // Set PID clock
2 const int PID_clock = 1000; //[us] = 1ms for instance
3 // Initialize clocks
4 unsigned long previousMicros_PID = 0; //[us]
5 unsigned long previousMicros_wave = 0; //[us]
6
7 void setup() {
8   // ...
9 }
10 void loop() {
11   /*
12    * IF (one PID clock time is elapsed)
13    * compute new action and update square wave parameter
14    * ELSE continue with previous square wave parameter
15    */
16   unsigned long currentMicros_PID = micros();
17   unsigned long currentMicros_wave = micros();
18   // PID clock
19   if((unsigned long) (currentMicros_PID - previousMicros_PID) >=
20     PID_clock) {
21     // Acquire current position
22     // ...
23     // Compute error
24     error_new = reference_position - current_position;
25     // Compute new action
26     // Proportional action
27     p_new = Kp*error_new;
28     // Derivative action from real continuous time derivative
29     contribution
30     d_new = (Td/(N*PID_clock+Td))*d_old + (Kp*Td*N)/(N*
31     PID_clock+Td)*(e_new-e_old);
32     // Integral action
33     i_new = i_old + Kp/Ti*PID_clock*error_new; // Backward

```



```

Euler
31   u_new = p_new + d_new + i_new;
32   // Comupte wave_period from u_new in some way
33   // ...
34   previousMicros_PID = currentMicros_PID;
35   error_old = error_new;
36   d_old = d_new;
37   i_old = i_new
38 }
39 // Square wave clock
40 if((unsigned long)(currentMicros_wave - previousMicros_wave)
    >= wave_period){
41     digitalWrite(myPin, !digitalRead(myPin));
42     previousMicros_wave = currentMicros_wave;
43 }
44 }

```

Listings 4.7: Example of PID control of a square wave period

Hence, a mock PID control logic was implemented using instead of a clock a `while()` loop that corrects the position of a joint till convergence given a certain tolerance. We reckon and strongly state that this is not a true PID controller. It is a workaround to simply control the robot without using high level software (that was indeed developed in a parallel thesis work).

The idea is to compute the new angle to be requested to the stepper motor as in continuous time (Eq.4.2) but approximating the time derivative of the error with a finite difference (Eq. 4.3) and the integral of the error with the area of a trapezoid (Eq.4.4).

$$\frac{de}{dt} \approx \frac{e(t_2) - e(t_1)}{t_2 - t_1} \quad (4.3)$$

$$\int_{t_1}^{t_2} e(t) dt \approx \frac{1}{2} (e(t_2) + e(t_1)) (t_2 - t_1) \quad (4.4)$$

As said, the library used causes the stepper motor to reach the requested position without possibility to interrupt the movement. Given this, some considerations can be done:

- The error is not computed every fixed time interval but when the arm concludes the requested rotations. This for sure causes poor performance if long rotations

are performed because the finite difference and the area of the trapezoid are not representative of the true trend of the error. Therefore K_p should be smaller than one to mitigate this problem.

- If the proportional gain K_p is bigger than one also overshoot in the first run of the `while()` loop is expected. To avoid big overshoots should be $K_p < 1$.
- Positive derivative and integral contributions cause the requested angle to increase possibly causing overshoot. Anyway, derivative and integral contributions are initialized to zero among global variables and at the first iteration of the `while()` loop are still set to zero. This is correct because the link starts moving with null speed and so the time derivative of error is null, and moreover only one point is available for the computation of the integral of the error and therefore the area is null.
- As the error decreases the derivative contribution becomes negative decreasing the value of the new requested rotation angle for each joint. This is expected and correct. Unit of measure of time are very important as shown later.
- The integral contribution theoretically can be positive or negative depending on the trend of the error. Anyway, since $K_p < 1$ and in the first run of the `while()` loop derivative and integral contributions are null, a big initial overshoot is not expected and therefore the error should be contained. This means that the target position of each link is expected to be reached within the tolerance with minimal oscillations and from above, since to have a null integral contribution the error must be both positive and negative in the same time history.

In Appendix A.3 is reported the code for whole robot arm. With this code a set of five relative angular positions with respect to the fully vertical position is requested to the user through the Serial Monitor of the Arduino IDE. Only when a valid input is given the arm will start moving towards the desired positions. Acquisition of the starting position is done starting from joint J1 to joint J5. As soon as an acquisition is done the current time instant is saved with the Arduino function `millis()` to increase precision, and conversion to the correct units of measure is done afterwards. It's important to notice that the `millis()` function returns a value in ms so having order of magnitude of 10^3 and that's why it is needed to divide by 1000. Keeping time in ms means having a derivative term very small because time is at the denominator of Eq.4.3 and an integral contribution very big since time is at numerator in Eq.4.4. Hence, instead of dividing by 1000 it is equivalent to use $K_d \approx 10^3$ and $K_i \approx 10^{-3}$. Anyway, it is preferable to have all PID gains of the same order of magnitude to understand relative importance at a glance.

```

1 float current_reading_J1 = analogRead(potPin_J1); // in range
   [0 1023]
2 // Saving the time in which the measure is done as early as
   possible
3 // Needed to compute after the derivative term
4 time_old_J1 = millis()/1000; //[ms] -> [s]
5 float volt_J1 = current_reading_J1 * resolution - home_volt_J1;
6 current_position_J1 = volt_J1 * sensitivity / tau_gear_J1 -
   home_offset_J1; //[deg]
7 error_J1 = desired_position_J1 - current_position_J1;
8 // Do the same for all other joints
9 // ...

```

Listings 4.8: Initial position measure

Afterwards the `while()` begins and its exiting condition is when all absolute values of errors are smaller than the tolerance. Then the control actions are computed starting from joint J1 to joint J5. Since the sixth joint has not a potentiometer for position feedback an action cannot be computed. Then, the synchronizer of all stepper drivers can be called

```

1 // PID-controller
2 float action_J1 = Kp*error_J1 + Kd*d_dt_error_J1 + Ki*
   int_error_J1_dt; //[deg] but for the link
3 // The same for J2, J3, J4, J5
4 // ...
5 // Release brakes
6 digitalWrite(pinBrake_J2, LOW);
7 digitalWrite(pinBrake_J3, LOW);
8 delay(10);
9 // The motor must rotate of action_Joint*tau
10 arm.rotate(tau*action_J1, tau*action_J2, tau*action_J3, tau*
   action_J4, tau*action_J5, tau*action_J6);
11 delay(10);
12 // Engage brakes
13 digitalWrite(pinBrake_J2, HIGH);
14 digitalWrite(pinBrake_J3, HIGH);

```

Listings 4.9: Control action computation

After the movement of the whole arm is completed, the new positions can be acquired

together with new time values and the new errors, time derivatives and integrals can be computed and variables updated. Again, the `millis()` is divided by 1000.

```

1 // Acquire new position
2 // J1
3 current_reading_J1 = analogRead(potPin_J1);
4 time_new_J1 = millis()/1000; //[ms] -> [s]
5 volt_J1 = current_reading_J1 * resolution - home_volt_J1;
6 current_position_J1 = volt_J1* sensitivity / tau_gear_J1 -
    home_offset_J1; //[deg]
7 error_J1 = desired_position_J1 - current_position_J1;
8 d_dt_error_J1 = (error_J1 - error_J1_old)/(time_new_J1 -
    time_old_J1);
9 int_error_J1_dt = int_error_J1_dt_old + trapz(time_new_J1,
    time_old_J1, error_J1, error_J1_old);
10 error_J1_old = error_J1;
11 time_old_J1 = time_new_J1;
12 int_error_J1_dt_old = int_error_J1_dt;
13 // The same for J2, J3, J4, J5
14 // ...

```

Listings 4.10: Updating position measure and variables

After the positions are reached within the defined tolerance, the code waits the user to input another value. Thus, this Arduino code is for interaction with the robot to request a pose and not to maintain it.

4.2.5. ROS interface

In this thesis the digital model of the robot arm realized is developed. A motion planner is developed using the ROS development environment. The Arduino code to be deployed on the micro-controller itself to let it interface with ROS is needed, and it's reported in Appendix A.4.

ROS is based on a network of processes exchanging messages and processing of messages takes place in nodes that constitutes the graph structure of the ROS network [24]. One node can act as a Publisher and a Subscriber, also simultaneously. A Publisher can send messages in the network but only about a defined topic that are named busses over which messages are passed. A Subscriber can receive messages belonging to a specified topic, being subscribed to that topic. The Arduino MEGA is a Subscriber of messages

of a topic about actuation and speed while is a Publisher of messages about feedback positions. Messages contain data types that may be of various nature but are fixed by the user. This means that the Arduino expects always the same data type and it cannot be changed during running.

With reference to Figure 4.6 [6], Arduino is inside the Real Robot (thick red box) of a ROS Control system, so it can be considered an embedded piece of hardware. Arduino doesn't act as an Embedded Controller since it doesn't run a PID. Instead it is responsible for the communication, receiving control inputs from and sending Mechanism States to the `hardware_interface::RobotHW` class of the `hardware_interface` package.

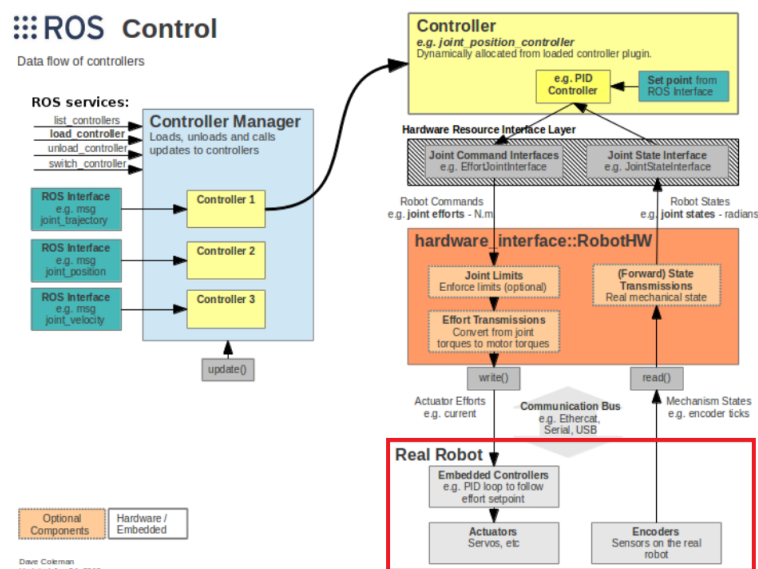


Figure 4.6: ROS Control overview

The Arduino MEGA acts as a ROS node that receives the joints angles and the maximum desired speed and sends the measured positions of each link. Communication is done through Serial communication protocol and in detail `rosserial_arduino` client library of `rosserial` protocol [17]. To let Arduino use `rosserial` protocol the `ros.h` Arduino library was used.

A brief description of how the Arduino firmware is developed is presented hereafter.

In addition to the `ros.h`, another Arduino library to set the data type of the messages exchanged must be defined. Angles and speed can be `float`, since the smallest angle that motors can do is 0.9deg. Moreover, since we are dealing with six motors, messages can be arranged in arrays to make manipulation easier. Hence the `std_msgs/Float32MultiArray.h` is used. The ROS node for the Arduino MEGA board must

be declared in the firmware. A node handle variable is created with `ros::NodeHandle` preceding the name of the variable itself.

To actuate the motors when a proper message is received a callback function must be defined.

```

1 // Callback function to make the arm move when a message is
  received
2 void move_arm_cb(const std_msgs::Float32MultiArray& cmd_msg) {
3   // Release the brakes on Joints J2 and J3 before start moving
4   digitalWrite(pinBrake_J2, LOW);
5   digitalWrite(pinBrake_J3, LOW);
6   delay(10); // Wait a short time to be sure the breaks are
  released
7   // Note that if the rotation of the link is requested, motors
  must rotate of tau*angle_for_the_link
8   float new_position[5] = {cmd_msg.data[0], cmd_msg.data[1],
  cmd_msg.data[2], cmd_msg.data[3], cmd_msg.data[4]};
9   float new_command[5]; // Command for the motors
10  for (int i = 0; i < 5; ++i) {
11    new_command[i] = tau*(new_position[i] - old_position[i]);
12  }
13  arm.rotate(new_command[0], new_command[1], new_command[2],
  new_command[3], new_command[4], new_command[5]);
14  // Store last requested positions coming from cmd_msg.data in
  old_position
15  for(int i = 0; i < 5; ++i){
16    old_position[i] = cmd_msg.data[i];
17  }
18  delay(10);
19  // Engage the brakes when finished moving
20  digitalWrite(pinBrake_J2, HIGH);
21  digitalWrite(pinBrake_J3, HIGH);
22 }

```

Listings 4.11: Definition of callback function for actuation

The command message `cmd_msg` is structured variable containing a data field called `data` that is an array of `float`. Since positions are requested from a higher level, the

firmware must be able to remember the joint positions previously requested. In this way the inputted positions are absolute. For this purpose the arrays `new_position`, `old_position` and `new_command` are used, where `new_command` is the difference to reach the new position knowing the starting one. There is certainty on the starting position thanks to the position feedback system.

After having declared the callback function, the Subscriber for position must be declared.

```
1 // Subscriber for position
2 ros::Subscriber<std_msgs::Float32MultiArray> sub_pos("/pvbot/
   actuate", move_arm_cb);
3 }
```

Listings 4.12: Definition of the Subscriber

The same procedure is needed to set maximum speed. A callback function and a Subscriber is needed.

```
1 // Callback function to set joint speed
2 void set_speed_cb( const std_msgs::Float32MultiArray& speed_msg
   ) {
3   // speed_msg.data is the speed for the joint so speed for the
   motor is tau*speed_msg.data
4   J1.setRPM(tau * speed_msg.data[0]);
5   J2.setRPM(tau * speed_msg.data[1]);
6   J3.setRPM(tau * speed_msg.data[2]);
7   J4.setRPM(tau * speed_msg.data[3]);
8   J5.setRPM(tau * speed_msg.data[4]);
9 }
10 // Subscriber for setting speed
11 ros::Subscriber<std_msgs::Float32MultiArray> sub_speed("/pvbot/
   set_speed", set_speed_cb);
```

Listings 4.13: Definition of callback function for setting speed

The Publisher for the joints positions requires to define the message and to create the Publisher itself.

```
1 // Setting Publisher for position feedback
2 std_msgs::Float32MultiArray feedback_msg;
3 ros::Publisher pub_state("/pvbot/state", &feedback_msg);
```

```
4 // feedback_msg.data will be an array of 5 element with the
   current position of the links
```

Listings 4.14: Definition of Publisher for position feedback

A function to create an array containing the positions read from the potentiometers is written and it helps in the composition of the data field in `feedback_msg.data`

In the `setup()` of the Arduino sketch initialization of `BasicStepperDriver` objects and ROS node is done and subscription and advertising of Subscribers and Publisher is done.

```
1 void setup() {
2   // Initialize stepper driver object
3   J1.begin();
4   J1.setSpeedProfile(mode, 875, 875);
5   J2.begin();
6   J2.setSpeedProfile(mode, 875, 875);
7   J3.begin();
8   J3.setSpeedProfile(mode, 875, 875);
9   J4.begin();
10  J4.setSpeedProfile(mode, 875, 875);
11  J5.begin();
12  J5.setSpeedProfile(mode, 875, 875);
13  // Setup brakes of joint J2 and J3
14  pinMode(pinBrake_J2, OUTPUT);
15  // Engage the brake = switch off relay -> set relay command
   pin to HIGH
16  digitalWrite(pinBrake_J2, HIGH);
17  pinMode(pinBrake_J3, OUTPUT);
18  // Engage the brake = switch off relay -> set relay command
   pin to HIGH
19  digitalWrite(pinBrake_J3, HIGH);
20  // Go home
21  home_axes();
22  // Initialize ROS node, subscribe and advertise topics
23  nh.initNode();
24  // Subribe node nh to defined objects
25  nh.subscribe(sub_pos);
```



```

26 nh.subscribe(sub_speed);
27 // Dimension of the message must be declared
28 feedback_msg.data_length = 5;
29 // Make the node nh advertise the defined object
30 nh.advertise(pub_state);
31 }

```

Listings 4.15: Firmware setup

In the `setup()` the command `home_axes()` can be found and it is used to home the robot at every power on. A home pose is defined by filling the `home_array` variable.

```

1 // Definition of HOME POSITION
2 const float home_position_J1 = 90.0; // CCW
3 const float home_position_J2 = 45.0; // CCW
4 const float home_position_J3 = 90.0; // CW => keep positive =>
   positive angle = CW wrt J2
5 const float home_position_J4 = 0.0; // KEEP ALIGNED
6 const float home_position_J5 = -90.0; // CW
7 const float home_array[5] = {home_position_J1, home_position_J2
   , home_position_J3, home_position_J4, home_position_J5};
8 // Define function for homing the robot (home position values
   are set for all axes)
9 // Conceptually similar to the actuation call back function
10 void home_axes() {
11 // Read current positions and save them in positions_array
12 feedback_position();
13 float command_angle[5];
14 // Filling the command array
15 for(int i = 0; i < 5; ++i){
16     command_angle[i] = tau * (home_array[i] - positions_array[i
   ]);
17 }
18 // Release the brakes on Joints J2 and J3 before start moving
19 digitalWrite(pinBrake_J2, LOW);
20 digitalWrite(pinBrake_J3, LOW);
21 delay(10); // Wait a short time to be sure the breaks are
   released
22 // Actuate to home

```

```
23 arm.rotate(command_angle[0], command_angle[1], command_angle
    [2], command_angle[3], command_angle[4], command_angle[5]);
24 // Engage the brakes when finished moving
25 delay(10);
26 digitalWrite(pinBrake_J2, HIGH);
27 digitalWrite(pinBrake_J3, HIGH);
28 }
```

Listings 4.16: Homing function

Finally, the firmware `loop()` can be described. Every time the `loop()` is run the joints positions are sent to the ROS controller.

```
1 void loop() {
2   feedback_position();
3   feedback_msg.data = positions_array;
4   pub_state.publish(&feedback_msg);
5   nh.spinOnce();
6   delay(200);
7 }
```

Listings 4.17: Firmware loop

5 | Programming approaches

Once the firmware for communication with the physical system is developed, it's possible to dive in the field of programming methods for robots. In this chapter an overview of the principal programming strategies will be done and the most suitable solution will be selected among the proposed ones.

In general, to control a robot and make it perform specific tasks the user must acknowledge some basic information. Its kinematic chain is important to understand the kind of movements that the robot is able to perform and to solve the direct kinematics problem. Then, motion planning must be carried out to obtain the desired motion that the end effectors of the robot must follow, for instance this is possible by exploiting some algorithms for path planning and time parametrization. Furthermore, the inverse kinematic problem must be solved to obtain the joints configurations during the execution of a planned motion and finally, to control a robot, clearly also the control logic must be defined with the type of controllers and the respective gains.

From the beginning of this work the idea was not just to program the tasks on the real robot but also to implement a simulation environment in which the user could test different tasks on the virtual model without the need of the actual robot.

Since there are different ways to approach the programming phase and since programming is a time consuming and expensive process is important to show the pros and cons of each different programming strategy and explain the choice of a ROS-based offline programming method for the work performed in this thesis.

In general there are mainly three categories of programming approaches:

- Online programming
- Offline programming
- Hybrid programming (Augmented Reality)

5.1. Online programming

Online programming consists in programming directly on the robot controller so, this strategy requires direct access to the physical robot. For this reason, before starting the programming phase, the robot must be already designed and manufactured. This is a limitation because in many circumstances realizing the production and programming phases in series is a big waste of time.

Furthermore, during the teaching phase the robot can't perform its activities and must be stopped so causing another waste of time and money.

In the field of online programming there are mainly two different strategies:

1. Teach pendant

The teach pendant is an handheld device with different buttons, Figure 5.1, that allow the operator to move the robot and place the end-effector in precise positions and orientations.

On the teach pendant there is always an emergency button which is the primary safety device to stop the robot due to the small working area between human and robot.

The operator stores in the robot controllers the coordinates for each position that must be reached to perform the desired application, these positions are stored in a series of movement instructions and, finally, once the whole program is learned the robot is ready to work.

During the programming phase the technician is also allowed to change the velocity of the robot, in fact during programming the velocity can't be too high for safety reason. Once the programming phase is completed, it's possible to perform the task at full speed.

This approach is the most popular in industries, in particular it is used a lot with collaborative robots in which a safe close interaction between operator and robot can be considered as safe without the need of a barrier. By the way, for certain applications is not so easy to program in this way. In fact, as written in the article [7], in applications like spray painting, sealant application, glue dispensing and arc welding, where the tool must keep a constant orientation with respect to the working surface and must follow a smooth path, programming with a teach pendant can be tedious and a lot of time is required to define all the positions and to store them in the controller.

Teach pendant allows to reach positions with higher precision than lead through method. These positions are defined with respect to specific reference frames, for instance in ABB these are called "work object" [1]. They are frames defined with respect to an object added in the workspace. The position of the target can also be defined from the world frame or from the frame placed on the tool. But the problem is that not always it's easy to define and reach these positions with the teach pendant as mentioned before.



Figure 5.1: Teach pendant

2. Lead through method

To program the robot following this approach the operator physically drags the manipulator and stops it in the positions that must be reached by the robot to perform the task. Once the robot is stopped in a certain position it's possible to store it in the controller and then the recorded positions are used to program the movement.

This programming approach is very intuitive and an operator with no programming skill can easily learn to program in this way.

Also, as written in 1, there are applications (reported in the article [7]), like arc welding, in which teach pendant is a tedious programming strategy because the tool must be kept orthogonal to the surface and must follow a smooth path. Instead, lead-through method can be an efficient solution to program the robot in these circumstances without putting too much effort.

Often the lead-through teaching handle is mounted on the robot tool as is shown in

Figure 5.2.

Clearly there are some limitations, it's difficult to move manually a robot if it's too big and also this close interaction between robot and human can be a problem for safety reason. In fact, this approach is used mainly with collaborative robots which are of limited dimensions and are designed to allow a safe close interaction with the operator.

Finally, since the robot is moved directly by the operator is impossible to reach very precise positions in performing the task.

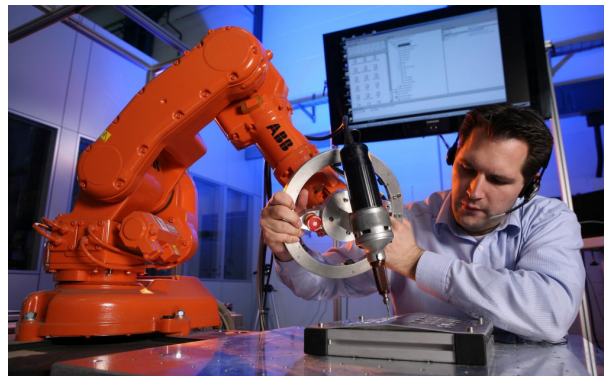


Figure 5.2: Lead through method

5.1.1. Pros and Cons

Below, there are some pros and cons of teach pendant method, lead through method and in general of online programming.

Pros

- Intuitive programming approach, so there is no need of a high skilled programmer. This is related particularly to lead through method.
- Low development time: the software is provided by a manufacturer and the operator can directly program by means of the teach pendant or with a lead through method.
- Teach pendant strategy allows to reach positions of the tool center point with higher precision than lead through method.

Cons

- Programming must be performed after designing and manufacturing the robot causing waste of time and money.

- During the programming phase the robot must be stopped from working, so there is downtime problem.
- Quality in the execution of the programmed task depends a lot on the ability of the programmer.
- Lead through method does not allow to reach positions with high precision because the robot is directly dragged by the operator.
- Lack of flexibility and reusability of the program, also when there is a slight change of the task reprogramming is needed.
- Teach pendant approach requires more skilled programmers with respect to lead through method which is more intuitive.
- Teach pendant strategy with text based input requires the programmers to learn different programming languages for each robot brand (e.g. RAPID for ABB robots, JBI for Motoman robots, etc.).

5.2. Offline programming

5.2.1. General overview

Offline programming allows the design of programs without having access to the actual robot. Programming takes place on an external system and then the program is downloaded to the real robot.

Commonly this strategy is characterized by different phases, as it's explained in the article [15]:

- 3D CAD model

Offline programming is an approach that starts from the 3D CAD model of the complete working cell so, it represents the robot and the different objects with which the robot interacts to perform a certain task. The CAD model can be generated by common modelling software or in some situation a 3D scanner can be used to obtain the model of the workpiece. The 3D CAD model is the base for offline programming, it allows to verify if there are some obstruction between the robot and the different objects and if this happens the user can change the position of the different components to perform the task in the right way. The CAD model of the robot is also of crucial importance to identify its kinematic chain. For instance, Figure 5.3 shows the simulating model of a five degrees of freedom robot and this

picture is taken from the article [11].

Offline programming needs just the 3D model, this allows to program without the physical robot and so, during the programming phase, the robot is not stopped from production avoiding the downtime problem. Furthermore, for the same reason, the programming phase can be carried out in parallel with the production of the robot saving a lot of time.

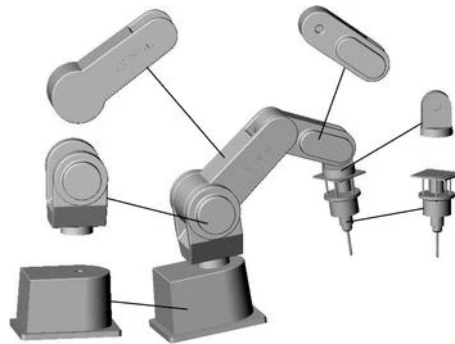


Figure 5.3: Simulating model of a 5 dof robot

- Motion planning

In general once the CAD model is provided, the second step in offline programming is path planning. There are software that have included algorithms for generating the trajectory that the end effectors must follow to perform the task. Usually, the choice of algorithm is specified inside the program in which at least the initial and final position of the task are defined.

There are some interesting applications, like the one explained in the article [13]. In which it's possible to generate the path of a welding application without programming but just drawing the path in the CAD model, enabling in this way a fast and simple programming since the program is automatically generated by the CAD file. Furthermore, for each welding path the operator can also specify the welding parameters, like velocity, voltage, torch distance to the surface. In fact, from the CAD file is generated a DXF file with all the information about the welding process.

Then, from the path generated is obtained the motion law exploiting the knowledge of some kinematic parameters. These are the maximum velocity and acceleration that the joints of the robot can reach, they depend on the properties of the motors that actuate the joints and there can be some limitation related to the application.

To generate the motion law also the inverse kinematics problem must be solved. This problem consists in finding a set of joints so that the robot places each end

effector (specified points on the links of the robot) at its target position and the way to formulate this problem is explained in the article [4].

The set of joints $\theta_1, \dots, \theta_n$ describes the configuration of the robot (n is the number of joints). The end effectors are specific points on the links and their position are defined in the column vector: $\vec{s} = (s_1, \dots, s_k)^T$. k is the number of end effector positions but each of these positions is defined by three scalars if the problem is in three dimensions. The end effector positions are functions of the joint angles: $\vec{s} = \vec{s}(\vec{\theta})$. The target positions are instead defined in the following vector: $\vec{t} = (t_1, \dots, t_k)^T$. Now, it's possible to determine the desired change in position from the vectors defined above: $\vec{e} = \vec{t} - \vec{s}$.

Solving the IK problem means finding $\vec{\theta}$ so that:

$$\vec{t} = \vec{s}(\vec{\theta})$$

But, since not always is possible to find a solution of this problem an iterative method is considered to approximate a good solution. The functions \vec{s} are approximated with the Jacobian matrix. This matrix is defined so:

$$J(\vec{\theta}) = \left(\frac{\partial s_i}{\partial \theta_j} \right)_{i,j}$$

It is a kxn matrix with the k elements which are vectors defined in 3 dimensions. With the Jacobian matrix is possible to determine the velocity of the end effectors:

$$\dot{\vec{s}} = J(\vec{\theta})\dot{\vec{\theta}}$$

This equation is expressed in continuous but if this is considered in discrete so, substituting the derivatives with finite increments the resultant formula will be:

$$\Delta \vec{s} \approx J \Delta \vec{\theta}$$

One approach is to set Δs equal to \vec{e} and so to solve the IK problem the solution of the following equation must be found:

$$\vec{e} = J \Delta \vec{\theta}$$

This is a possible way to formulate the IK problem but it's not the only one.

By the way, usually, software have already included algorithms for inverse kinemat-

ics. There are different methods to solve IK problems once the problem is formulated like is explained above and these methods are present in the article [4]. Strategies like the Jacobian transpose method and the pseudoinverse will oscillate in the neighborhoods of singularities. Instead, for instance the damped least squares method can perform well with unreachable target solution. So, once the user chooses the kind of strategy to solve the IK problem, these methods bring to a set of non linear equations that must be solved. So, also a numerical method to solve this problem is needed and a possible option can be the Newton-Raphson method.

- Simulation

Another great advantage is the presence of a simulation environment in which is possible to test the robot and try different applications. Hence, it's possible to improve and refine the program still not working on the actual robot. So, offline programming allows to improve productivity and make a lot of test in the virtual world before going in the real one improving also the safety of the operators.

- Calibration

Clearly, once the program is applied to the physical robot, a calibration phase is needed. Creating a digital twin of the working cell is very powerful but there is always a slight difference between real and virtual world.

In offline programming there are three different kind of software as explained in the article [15]:

- Manufacturer software

Almost every robot manufacturer has its own software. This is an advantage to make it more compatible to the hardware and since software and hardware are packaged together the cost is not too high. Some example of manufacturer software are RobotStudio from ABB, KUKA-Sim from KUKA etc.

Clearly the big disadvantage of this kind of software is that the user can just pick a robot of the same brand of the software.

Furthermore, since the software is provided by a manufacturer it can be very restrictive so, if some features are not present usually it's not possible to add them modifying the software.

- Commercial software

This software is characterized by an higher flexibility than manufacturer software

because they can be used with hardware from different manufacturers. By the way, with some robot manufacturer, there can be compatibility issues. The more popular generic software are Delmia from Dessault Systems, RobCAD from Technomatix and RobotDK.

- Open source software

This kind of software are often used in academic field, even if open source does not mean necessary that the software is cost free they are usually much less expensive than the other kind of software explained above.

Furthermore, the fact that it is open source means not only that the user can freely modify the software and its source code but also he can share it with any other user.

In this way a software can be developed by different users and any user can modify the software to improve it, for instance, adding some features.

5.2.2. Pros and Cons

From what has been described so far some pros and cons of offline programming are emerged:

Pros

- No need to stop the production during the programming phase.
- Possibility to program the robot in parallel with the production saving a lot of time.
- High flexibility, in fact if there is a slight change in the task often is easy and quick to adapt the program to this change.
- The presence of a simulation environment is a great advantage with respect to online programming. This environment allows to make a lot of tests without stopping the robot, in this way it's possible to improve productivity and safety.
- Allow to verify the position of the robot with respect to the different objects which are present in the environment so, to verify if the robot operates and interacts with these objects in the correct way.
- Allow to verify that the robot is able to assume the different configurations to perform the motion planning of a certain task.

Cons

- Software for offline programming are often expensive.

- Programming is time consuming, in particular customizing the software to a specific application.
- Require high level programming skills.
- Offline programming is not popular in small and medium enterprises, it is too expensive for the reasons reported above.

5.3. Hybrid programming (Augmented Reality)

AR is a trending technology in Industry 4.0 and it consists in considering 3D virtual objects in a real environment.

AR exploited in programming a robot can be classified as hybrid programming because it mixes real elements, typical of online programming, with virtual components that characterize an offline programming approach.

The article [3] explains how AR can be useful in assembly application. An head mounted displays, such as Microsoft Hololens (Figure 5.4), is used by the operator which can visualize the workpiece and the working environment in virtual reality. It's possible to generate this 3D hologram by a permanent tracking of the real environment. The real components are recognized by means of trackers and reproduced in the virtual reality by the Microsoft Hololens. Then, the operator with its gestures is able to interact with the virtual components and perform the assembly task in AR. The gestures are recognized by integrated camera and the steps to perform the assembly procedure are recorded by the Microsoft Hololens.

So, this application highlights how this programming strategy can take some positive aspects from online programming. In fact, the robot learns the task by reproducing the movement performed by the operator and this is a very intuitive way to program a robot that do not require a high skilled operator. In its simplicity and intuitiveness this kind of programming strategy recall online programming approaches like lead through method.

Furthermore, AR allows to make a step further. In fact, using a HMD this programming approach is not just intuitive but also much safer than lead through method.



Figure 5.4: Microsoft HoloLens

Another interesting example is the one in the article [15] in which a virtual model of an aeroplane washing robot is inserted in the same environment of a scaled real model of an aeroplane as is shown in Figure 5.5. So, it's possible to simulate the washing task performed by the virtual robot to the aeroplane model. Then, this program will be used by the real robot on the real aeroplane.

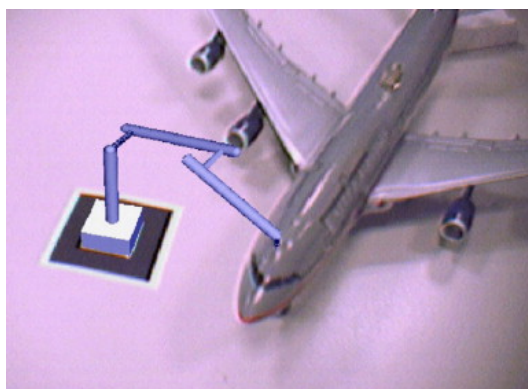


Figure 5.5: AR: miniature airplane washing robot

This application highlights some similarities with offline programming, in fact using a virtual model of the washing robot allow to program without actually having available the real robot.

Furthermore, AR allows to use a scale model of the aeroplane without having to model the aeroplane in the virtual environment saving in this way time and money.

AR is a promising technology for the future of robotics, as explained above it can be used for programming taking some feature of online and offline programming and making some improvements. By the way, nowadays this technology is not so diffuse in industries but it's limited to research and gaming fields.

5.4. ROS-based offline programming

In this thesis a ROS-based offline programming approach is chosen to program the robot.

First of all, with an offline programming approach the user is able to program without the need of the actual robot. So, programming and production phases can be performed in parallel saving in this way a lot of time.

In addition to this reason a digital twin of the system allows to program new tasks without stopping the actual robot and this is not possible with an online strategy. Furthermore, offline programming is flexible so if there are little changes on a task it's easy to adapt the code already written without having to reprogram from scratch the robot. Finally, in offline programming is also feasible to generate a simulation environment which is very useful to test and refine the programs on the virtual model.

So, once explained the choice of offline programming for this project, must also be understood why, in particular, ROS has been picked to generate a software for offline programming.

ROS [16] is an open source framework and it is cost free with respect to software owned by companies which are often really expensive. Furthermore, being open source the user can modify the source code, adding the features that are useful for its project and also he is free to share the generated files with any other user. During the years a vast ecosystem of software for robotics have been developed by the community from which the user can get useful information. So, it's clear that the choice of ROS is a smart decision at an academic level. ROS allows also to have free access to MoveIt! [12] which is a ROS-package for motion planning. The user can exploit algorithms of path planning from the OMPL (Open Motion Planning Library), this is an open source motion planning library which doesn't have the concept of a robot in fact, MoveIt! is needed to configure OMPL to work with robots. In this way it's possible to define the desired target position of the end effectors as a function of time.

To plan the motion law the inverse kinematic problem must be solved to get the joints value in all the different configurations that the robot must assume to perform the task. ROS provides different algorithms of inverse kinematics and if these algorithms was not fast enough or did not solve, for some reason, the kinematic of the robot the user would be free to generate its own algorithm of inverse kinematics. The target joints values obtained by solving the IK problem are of crucial importance to control the system in fact, this information is sent to ROS control. ROS control is a list of packages in which the user can define all the different type of controllers needed for the specific application and takes

as input also the joint data coming in feedback from the potentiometers. These inputs allow to compute the error between the target joints value and the actual one.

ROS is also compatible to any kind of hardware, in the case of this project since the robot is designed and manufactured by ourselves and so, it doesn't come from a brand it wouldn't be possible to use a manufacturer software for compatibility issues. ROS control is able to communicate by means of an Hardware Interface to the virtual robot in a dynamic simulation or to the physical robot.

Finally, ROS can be integrated with Gazebo [8] which is a cost free 3D dynamic simulator. This environment is very useful to test and refine the different programs without the need of the actual robot.

6 | ROS-based offline programming

ROS is an open source framework which is exploited to program the robot. It allows to design software for robotic applications and in this project the ROS distribution Melodic is used in Ubuntu 18.04.5 LTS.

In this chapter is explained from scratch how to program the robot in a modular way, generating different environments.

First of all, a workspace must be generated. This folder will contain all the environments needed to program the robot. Inside the workspace there are all the packages and codes compiled with catkin which is the build system of ROS.

- src folder (source space): contains the catkin packages.
- build folder: in this folder CMake is invoked to build the packages inside src folder.
- devel folder: here built targets are placed prior to being installed.

Then, inside the src folder different packages have been created:

- Robot model
- MoveIt! motion planning
- Gazebo modelling

These packages have been created both for a robot model with a gripper, since it can be a future addition to the real robot, and for a robot model without an end effector and with joint limits which represent the features of the real robot.

In general the robot model is needed to visualize the robot and specify the kinematic chain, with MoveIt! the user is able to plan the motion law that must be followed by the end effectors and by the joints so, in this ROS-package (MoveIt! package) is considered just the kinematics of the robot and not its dynamics. Then, to define the controllers is used ROS control and if the real robot is controlled there is also an hardware like Arduino

otherwise ROS control acts on the Gazebo environment controlling the digital version of the robot. Gazebo is a dynamic simulation environment so, it considers the robot model with also its dynamics and in this way the user is able to replicate the behaviour of the actual robot.

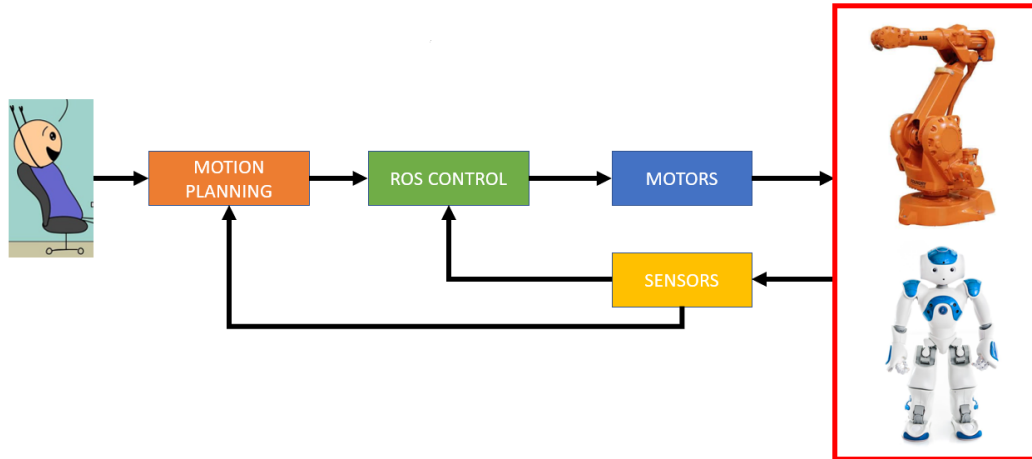


Figure 6.1: Block diagram to control a robot

6.1. Robot model with gripper

6.1.1. Visualization in RViz

In this chapter is simply reported the robot visualization, it is generated exploiting the xacro file of the robot model that will be described in the following chapters.

During the design phase a gripper has not been designed. But, when generating the digital twin of the robot, a gripper is added to the xacro model to obtain a complete virtual model and so to be able to program some task like pick and place in MoveIt!.

RViz stands for ROS Visualization, in the Figure 6.2 is shown the robot with a GUI that allows to define each joint values. Specifying the value of each joint with the GUI means working in forward kinematics. In this way just one configuration of the robot can be obtained and can be directly visualized in Rviz. Forward kinematics consists in determining the end effectors positions knowing the joints position.

The robot is composed by 6 DOF related to the manipulator and 1 DOF that allows to open and close the gripper. This tool is very useful to verify that the model has been generated in the correct way.

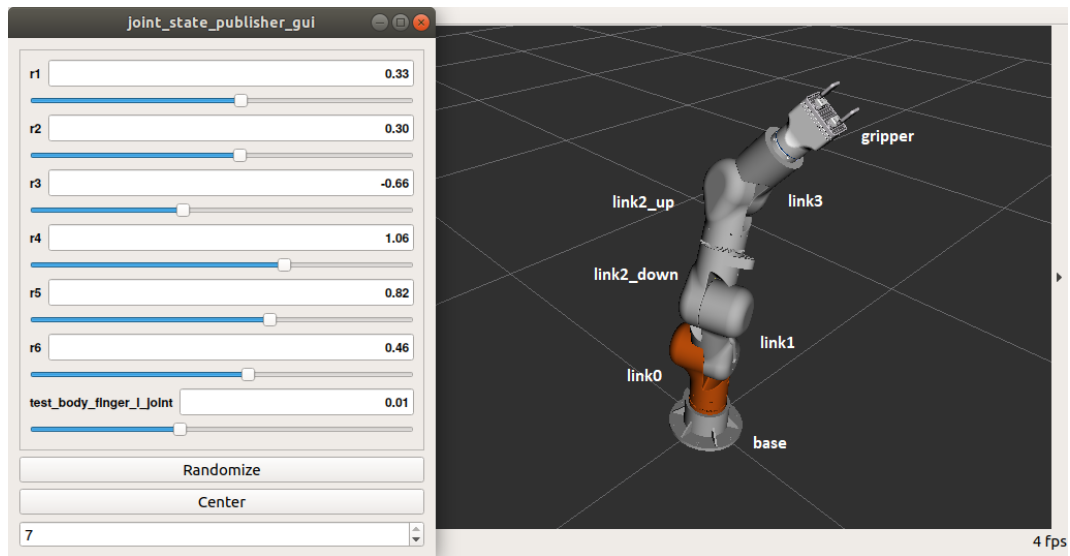


Figure 6.2: Robot model

To obtain the visualization of the robot model, it's necessary to launch a file in which, respectively, different nodes have been launched. So, calling in terminal the command `rqt_graph` it allows to obtain a clear graph with the nodes and topics that work when the robot is launched in Rviz. In Figure 6.3 it's possible to see the graph, the first node is the `joint_state_publisher`. It reads the `robot_description` parameter from the `xacro` file of the robot model, find all the non fixed joint, the type of joints and the respective limits. The value of each joint state can be given in input to the `joint_state_publisher` by a GUI and their default value is obtained from the `xacro` file. Then, the joints value is published to the topic `joint_states` and the node `robot_state_publisher` subscribes to it. This node uses the `xacro` file specified in the parameter `robot_description` to get the kinematic chain of the robot so, it is able to perform the direct kinematics and to obtain the position and orientation of each frame of the robot. Finally, the last node allows the visualization of the robot model in RViz.



Figure 6.3: rqt graph in Rviz

In Figure 6.4 there are represented the frames associated with each joint to have a better understanding of the position of each revolute joint present in the manipulator. The frame

placed at the bottom of the base is the one fixed to the ground, instead the others are related to the revolute joints. Then, to better understand the picture it must be said that for each frame x axis is the red one, y axis is the green one and z axis is the blue one.

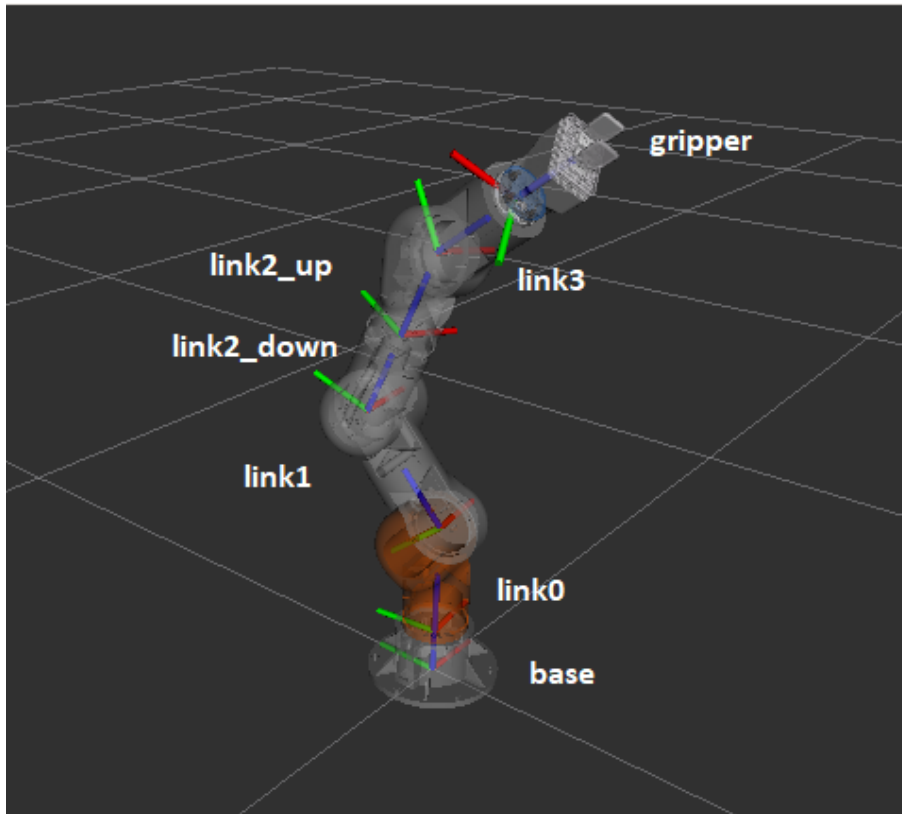


Figure 6.4: Robot model - name of components and frames

The names given to each joint during the programming phase, the respective frame and the axis of rotation are defined in the following Table. Clearly, also look at Figure 6.4 to understand to which reference frame each joint is related.

Joint name	Frame position	Axis of rotation
r1	between base and link0	z
r2	between link0 and link1	x
r3	between link1 and link2_down	x
r4	between link2_down and link2_up	z
r5	between link2_up and link3	x
r6	between link3 and gripper	z

Table 6.1: Joints name - Frame position of each joint - Axis of rotation

6.1.2. Manipulator model

In this section a description of the different passages to obtain the virtual model of the manipulator is performed. The CAD file of the robot is a prerequisite to generate the URDF file of the robot model. The CAD file was already realized in Autodesk Inventor. So, now it's possible to describe how a URDF file is built and in particular a xacro file. In fact, in this work has been chosen to generate a xacro file because it allows to reduce the amount of code present in a URDF file and also to define some parameters that can be used throughout the code.

Looking at the code A.5 on page 189, the first two lines are mandatory to define a xacro file and also they highlight that the file is written in XML language.

```
1 <?xml version="1.0"?>
2 <robot name="lucabot" xmlns:xacro="http://www.ros.org/wiki/xacro">
```

As it has been said previously a xacro file allows to define parameters, in this project just one kind of parameter is defined and it is a constant that multiplies each mass and inertia component of the links. This value is related to the 3D printer and in particular it depends on the infill. Below, the constant `i1` is reported which multiplies mass and inertia components of link1. It is obtained by dividing the actual mass of link1 by the mass obtained from the CAD file in Autodesk Inventor.

```
1 <xacro:property name="i1" value="0.3477285474" />
```

Then, in the following part of the code is defined the link0 (shown in Figure 6.2). This is an example to explain how each link has been introduced in the xacro file.

Visual tag is needed to define the shape of the component for visualization purposes. The shape is defined by importing the mesh of the CAD file modelled in Autodesk Inventor and it is imported in .dae format. The origin consists in defining the reference frame of the visual element with respect to the reference frame of the link.

Collision tag is used to define the geometry of link0 when there is the need to consider collisions with other components. For instance this tag is important during the planning phase, trajectories in which collisions are detected are not feasible. Usually, like in this project, the geometry of the collisions is much simpler than the actual geometry of the component to reduce computational time. As is shown in Figure 6.5 just boxes and cylinders are used as geometry for collisions and, in this case, link0 is approximated with a box able to include the entire shape of the link. So, these simple geometries guarantee collision avoidance with a safe margin.

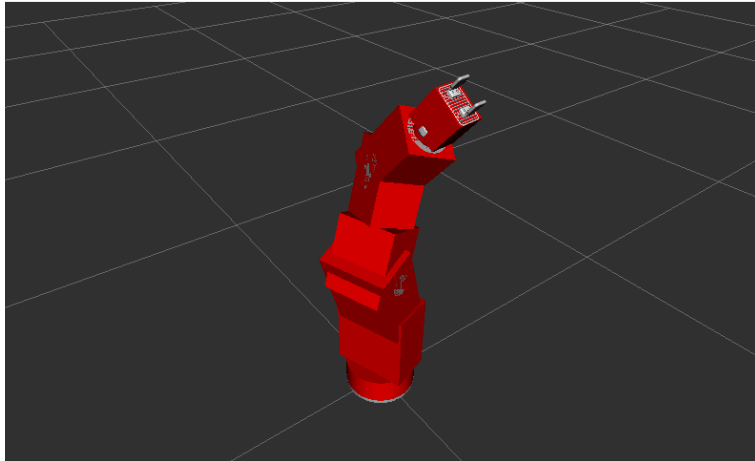


Figure 6.5: Collisions geometry

Inertial tag presents the origin of the inertial reference system with respect to the link reference system, the inertial reference system is placed in the same position of the center of gravity (position taken from the CAD file). Then, the mass and inertia properties are taken from the CAD file, using Autodesk Inventor, in the tab iProperties and considering ABS plastic material. The mass and each component of the inertia matrix are multiplied by the constant related to the 3D printer.

The inertial tag is here present but it's not useful to visualize the robot in RViz. These characteristics of the links will be important when the dynamics of the system is considered and influences the behaviour of the robot, for instance in the Gazebo environment.

```

1 <link name="link0">
2   <visual>
3     <origin rpy="0 0 0" xyz="0 0 0"/>
4     <geometry>
5       <mesh filename="package://lucabot_description/model/
6         link0_denti_incassati.dae"/>
7     </geometry>
8   </visual>
9   <collision>
10    <origin rpy="0 0 0" xyz="0 0 0.1255"/>
11    <geometry>
12      <box size="0.206 0.120 0.229"/>
13    </geometry>
14  </collision>
15  <inertial>
16    <origin xyz="-0.030697 0.000014 0.136133" rpy="0 0 0"/>
17    <mass value="$${i*1.385}" />

```

```

17 <inertia ixx="{i*0.006060034}" ixy="{i*0.000000665}" ixz="{i
    *0.001682159}"
18         iyy="{i*0.006600492}" iyz="{i*-0.000001679}"
19         izz="{i*0.003229145}" />
20 </inertial>
21 </link>

```

Once all the links are defined also the joints must be considered. In fact, the xacro file is really important because contains information about the kinematic chain of the robot. All the joints related to the manipulator are revolute joints and here after there is the definition of the joint between link0 and the base. In the robot tree the parent link is the base and the child is link0. The origin of the joint is located in the origin of the child link. For a revolute joint are also defined: the limit on the maximum joint effort expressed in Nm, the upper and lower limits for the value of the joint (in this case link0 can basically perform a complete revolution), the maximum velocity reachable by the joint expressed in rad/s and finally the axis of rotation.

```

1 <joint name="r1" type="revolute">
2   <origin xyz="0 0 0.07" rpy="0 0 0" />
3   <parent link="base" />
4   <child link="link0" />
5   <limit effort="40" lower="-3.142" upper="3.142" velocity="1" />
6   <axis xyz="0 0 1" />
7 </joint>

```

6.1.3. OnRobot gripper model

As anticipated in the section 6.1.1 a gripper has been added to the virtual model of the robot. In particular, an OnRobot gripper has been picked [14].

The choice of this gripper could not be permanent, it could be modified in the future by other students who will work on this project and maybe the gripper will be substitute by one that will be designed. By the way, adding a gripper now allow to address the programming phase with more completeness and even if the gripper will be substituted the main passages to add it in the xacro model, use it in MoveIt! and for programming the pick and place task remain the same.

Looking at the Figure 6.6, the gripper used for this project is composed by different components like the quick changer (qc link) which is fixed to the body of the gripper. The quick changer is useful to perform an easy and quick substitution of the gripper.

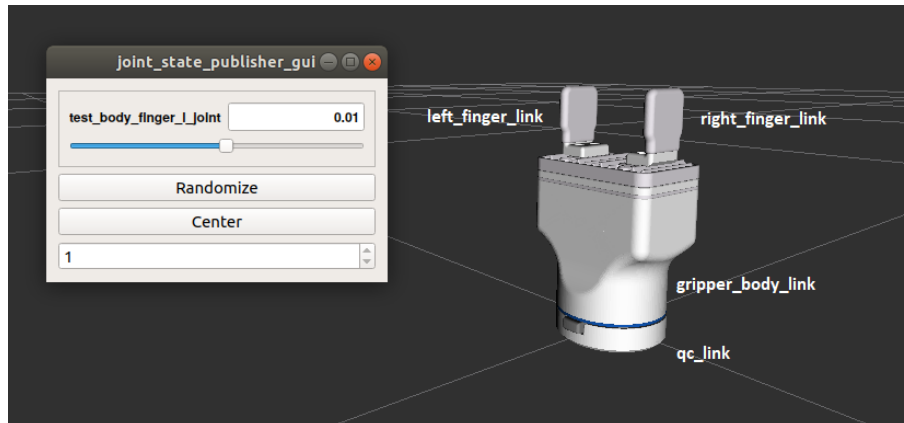


Figure 6.6: OnRobot gripper

Then, there are just two fingers and the left one is connected to the body of the gripper with a prismatic joint defined in the following lines of code which are extracted from A.6 on page 195.

```

1 <joint name="${name}_body_finger_l_joint" type="prismatic">
2   <axis xyz="0 1 0"/>
3   <limit effort="1" lower="-0.001" upper="0.025" velocity="1.0"/>
4   <origin xyz="0 0.0195 0.1098"/>
5   <parent link="${name}_gripper_body_link"/>
6   <child link="${name}_left_finger_link"/>
7 </joint>

```

One peculiar characteristic of the gripper is that it exploits mimic joint to define the prismatic joint between the right finger and the body of the gripper. The mimic joint allows the right finger to move with the left one. In this way, as it's possible to see in the GUI, the two fingers can't be moved independently but there is just one joint that moves both fingers.

The xacro file A.6 on page 195 defines the gripper model and since it's necessary to link the gripper to the manipulator this file has been included in the file of the manipulator model. Below, there are some lines of code taken from the file A.5 on page 189 that allows to do this and also enable the gripper to be placed at the end of link3 of the manipulator. In particular, the parent of the gripper is tool0 which is a dummy link placed at the end of link3.

```

1 <xacro:include filename="$(find 2fgt_onrobot_description)/urdf/2
   fgt_urdf_simplified_collision.xacro" />
2 <xacro:boh parent="tool0" name="test">
3   <origin xyz="0 0 0" rpy="0 0 0" />
4 </xacro:boh>

```


6.2. Robot model of real robot

In this section it is explained the digital twin of the real robot. First of all there is no gripper so, the model has 5 DOF which are all revolute joints as the real robot. The model can be visualized in RViz as shown in Figure 6.7.

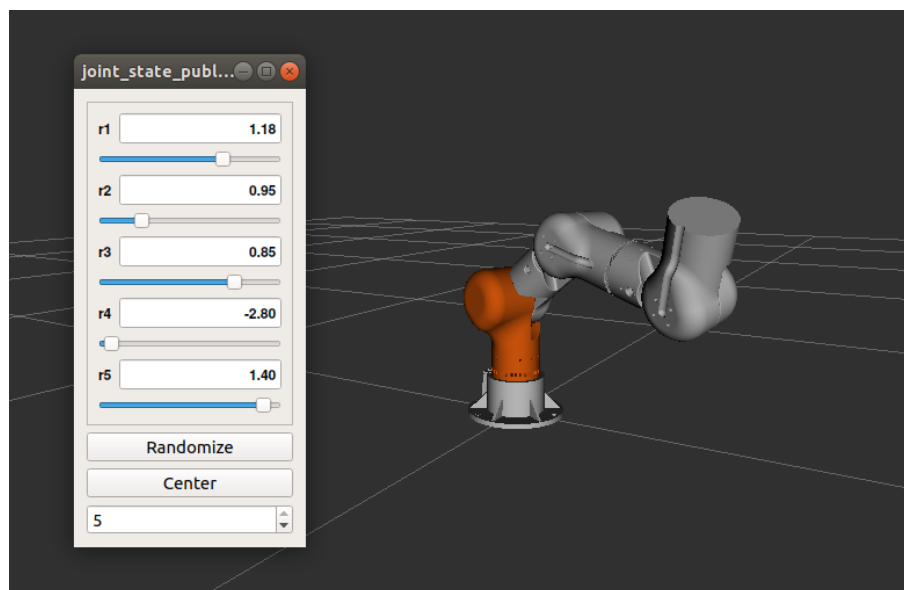


Figure 6.7: Real robot model

To obtain this model a xacro file has been created following the same procedure explained in section 6.1.2 on page 119. In this file (A.16 on page 218) there are lower and upper bounds of the joints limited with respect to the ones defined in the model with the gripper in which the joints are able to perform complete rotations. The limitations on the movement of each joint (reported here 6.2) are related to the design and manufacturing phase and they are obtained by testing the real robot.

Joints name	Lower and upper bounds [deg]
r1	-170:+170
r2	+45:+90
r3	-90:+90
r4	-170:+170
r5	-90:+90

Table 6.2: Lower and upper bounds of each joint

Furthermore, the maximum angular velocity of each joint is set in the xacro file at 1 rad/s

and also the limits on the effort must be defined. The limit effort is the maximum value of torque that can be applied on a joint, to find this value the torque-speed curves that characterize each motor are considered. First of all, the velocity limit of each joint equal to 1 rad/s is converted in RPM and multiplied by 35 (mechanical reduction) to obtain the maximum value of motor speed which is approximated down to 330 RPM. Also by approximating the characteristic curves of each motor with straight lines the value of the torque deliverable by each motor at the maximum speed of 330 RPM can be graphically obtained. This value is multiplied by the mechanical reduction to obtain the limit effort since it represents the torque on the joints. These values of limit effort, reported here 6.3, are very conservative since the torque-speed curves of each motor are all monotonous decreasing so, the joints will work at speed lower than the maximum one and the motor would be able to provide torque higher than the one obtained at the maximum speed.

Joints name	Limit effort [Nm]
r1	19.8
r2	19.8
r3	11.7
r4	6.9
r5	6.9

Table 6.3: Limit effort of each joint

6.3. MoveIt! motion planning

The MoveIt! motion planning package has the aim to generate the motion law that the end effectors should follow to perform a desired task. To program the robot and making it performing a desired task MoveIt! is able to process scripts written in Python or C++ or the robot can be programmed through a GUI as it is explained in [12].

As shown in Figure 6.8, the node `move_group` takes as input three parameters: the URDF file (to be precise the xacro file in this case). The SRDF file which contains different useful information like joint and link groups, pre-defined robot configuration, end effector group, virtual joint, information on collision checking. This file is a product of the MoveIt! setup assistant that will be explained later. The last input is the MoveIt! configuration which is also generated by the MoveIt! setup assistant and contains different information like joint limits, kinematics, motion planning etc. (these information are stored in the config directory generated in the MoveIt! package).

Finally, the `move_group` node communicate to the robot by means of topics, for instance it listens on `joint_state` topic to have always information of the current robot state, it also gets point clouds from the robot 3D sensors. Furthermore, it talks to the controllers by means of the `JointTrajectoryAction` interface.

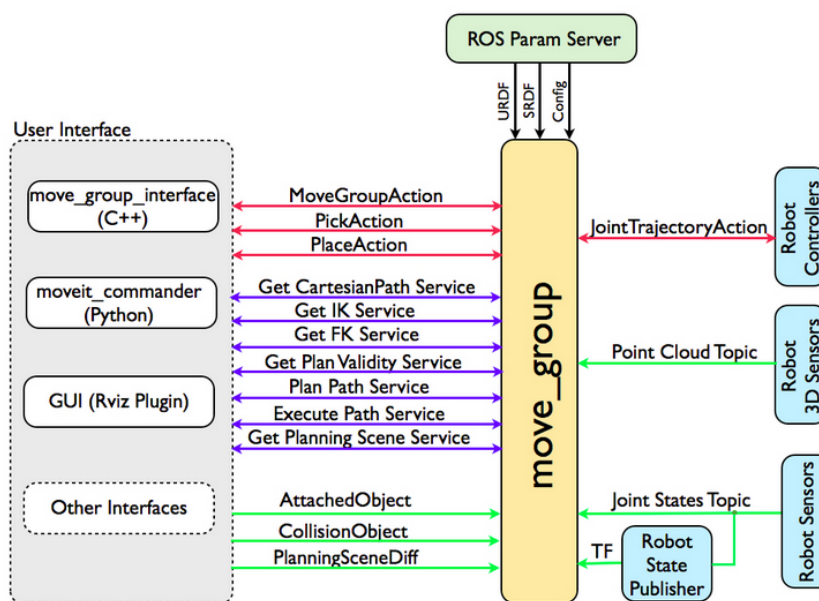


Figure 6.8: MoveIt! system architecture

6.3.1. MoveIt! Setup Assistant

MoveIt! Setup Assistant is a GUI used to configure a specific robot model for using it with MoveIt!. It allows to generate the SRDF file and all the other configuration file mentioned previously.

There are different steps that must be performed to configure the robot:

- Load the xacro file of the robot model.
- Generate self collision matrix.

This step exploits the collision meshes generated in the xacro file which are a simplification of the shape of the different link. In fact, the number of triangles of the meshes influences the computational time of collision checking. Checking for self collision means investigate random robot poses and disable pair of links that are always in collision, never in collision, adjacent, in collision already in the starting pose. Disable some link from collision checking is useful because decreases the motion planning time. In this project has been used a sampling density of 10000 which

is the default value and indicates the number of random robot poses to check for self collision.

- Add virtual joint.

In this step a virtual joint is generated between the root link of the robot (the dummy link: world) with respect to the world reference frame. This joint is fixed because the robot in this project is attached to the world. Instead, for instance, with a mobile robot the virtual joint should be a planar joint to allow the movement of the robot on the plane.

- Planning groups.

Planning groups are a crucial aspect in MoveIt!, in fact MoveIt! acts on each single group. So, it's possible to plan the motion of one group while keeping the joints of other groups stationary.

Considering the robot model with the gripper there are two groups: manipulator and gripper. The manipulator group is created by considering the joints of the robot arm. Instead, the gripper group is generated from the links of the gripper.

As shown in Figure 6.9, for the manipulator group is specified a kinematic solver to solve the inverse kinematics: KDL kinematic plugin. This is the default inverse kinematic solver and as explained in the article [9] the method consists in finding the solution of the differential kinematics equation:

$$\dot{s} = J(\theta)\dot{\theta}$$

the vector \dot{s} defines the velocity of the end effectors, J is the Jacobian matrix, $\dot{\theta}$ is the vector with the angular velocity of the joints.

The solution of this equation is that of a linear least squares problem that minimizes:

$$\|\dot{s} - J(\theta)\dot{\theta}\|$$

To minimize this value the Newton-Raphson method is used.

By the way, a user could implement its own solver, like the IKFast package, to solve in a better and faster way the inverse kinematics. Furthermore, a planner must be considered and in this case RRT is selected but when a precise task is programmed it's possible to specify the desired planner in the script. The user can choose between different planner given by OMPL (Open Motion Planning Library).

Just two algorithms: RRT, RRTstar are explained below, RRT is the one that will be used for the pick and place program and it's worth giving an explanation of RRTstar because it derives from RRT. These planners are algorithms for path planning, they allow to generate the geometrical path which the robot must follow to perform a certain task. To generate the motion law MoveIt! takes in consideration some kinematic parameter: max velocity of each joint predefined in the xacro file of the robot model and max acceleration of each joint defined in the joint_limits.yaml file obtained from the MoveIt! setup assistant.

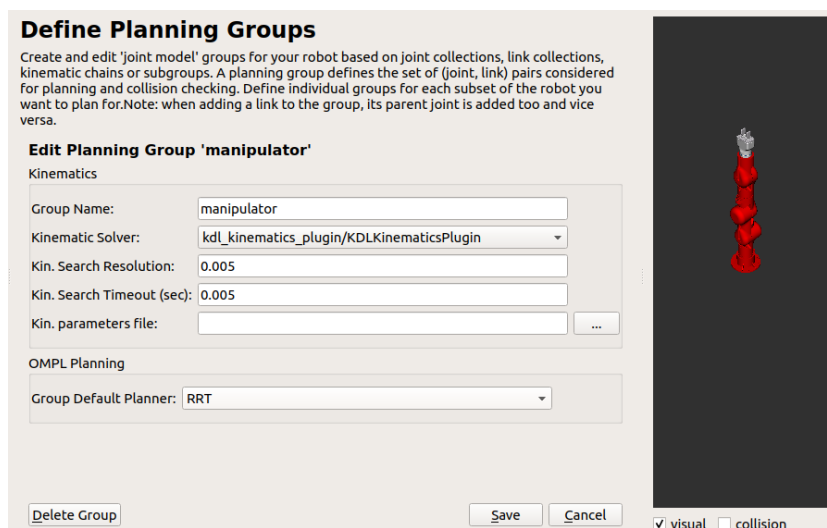


Figure 6.9: manipulator group

RRT

RRT is a single query sample based algorithm, it works by generating a tree of nodes and this is performed step by step, each step a random place in the workspace is considered and the tree is expanded in the direction of this random place starting from the nearest node to that position generating in this way a new node. The tree is expanding till a node fall into the goal position and so a feasible trajectory that link the starting position to the final one is found. This is not an optimum solution and if the number of nodes increases the trajectory remains always the same. Looking at Figure 6.10 it's shown an example of RRT algorithm and is evident that the trajectory obtained is not the optimum one.

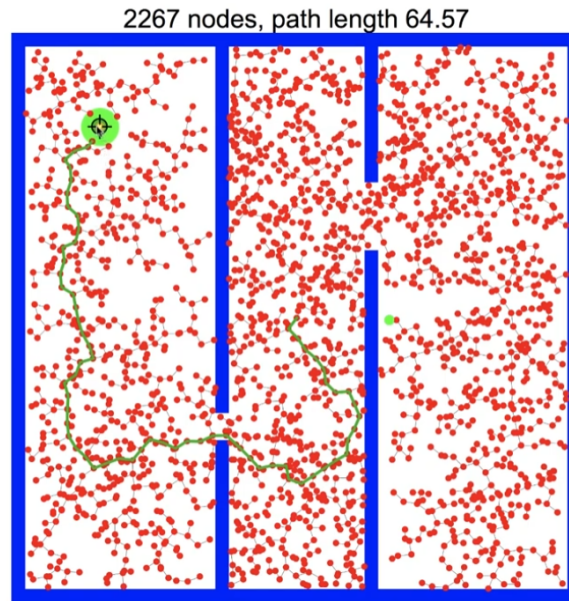


Figure 6.10: RRT example

RRTstar

This algorithm has been generated from the RRT, basically it works in the same way but it doesn't stop at the first feasible trajectory that is found. But, increasing the number of nodes, if another shorter trajectory is obtained this will be picked. Clearly the trajectory found will not be the shortest since this happens with the number of nodes tending to infinite. By the way, it's an optimum solution. In Figure 6.11 it's possible to observe the trajectory obtained with RRTstar which is much shorter than the one of Figure 6.10 obtained with RRT algorithm.

It is also possible to notice that the tree generated with RRTstar is different from the one of RRT, in fact at each random pose considered a new node towards that position is generated starting from the nearest node. But, the tree is expanding to the new node not necessarily from the nearest node. In fact, each vertex has a cost function which is its distance from the starting point. Once the new node is generated an area with a certain radius, centered in the new node, is explored and the node is connected to the vertex in this area with the minimum cost. In this way the tree obtained is smoother but the problem is that it suffers from a reduction in performance. Finding the optimum path requires more time.

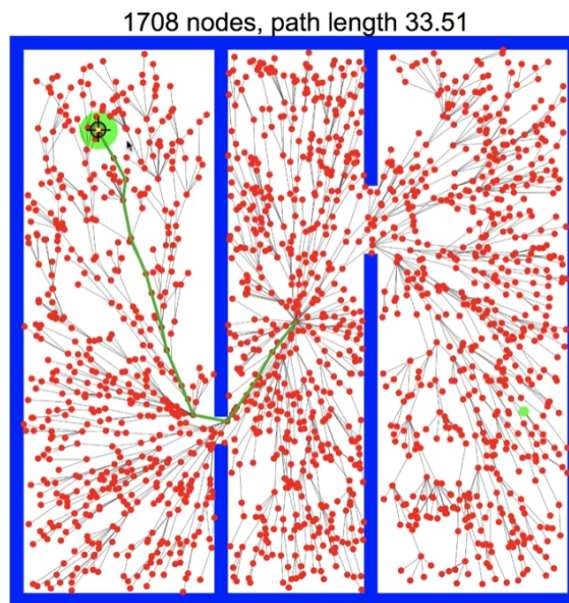


Figure 6.11: RRTstar example

- Define end effector group.
- ROS Control

In MoveIt! a `position_controllers/JointPositionController` type of controller is chosen to control the joints of the manipulator.

6.3.2. MoveIt! GUI

Once the robot model has been configured in MoveIt!, it's possible to plan some simple movement by launching the code A.10 on page 204. In Figure 6.12 there is the robot model with the gripper configured in MoveIt! and visualized by means of RViz. Then, once the Motion Planning Plugin is loaded the user can start planning. By setting as planning group the manipulator, the user can interact with the robot model by means of the interactive marker and he can drag the robot in a feasible position. In this way, he can define the initial and final position of a desired movement. Instead, by setting as planning group the gripper the user can decide its configuration for instance picking a close or open configuration which was defined in the MoveIt! setup assistant. Finally, before planning and executing the movement the user can set in the context section which algorithm of path planning the system must use. So, planning a movement in this way it is very easy. Instead, a user with some coding experience can plan a more complex task by programming in Python or C++ as will be explained in the next section.

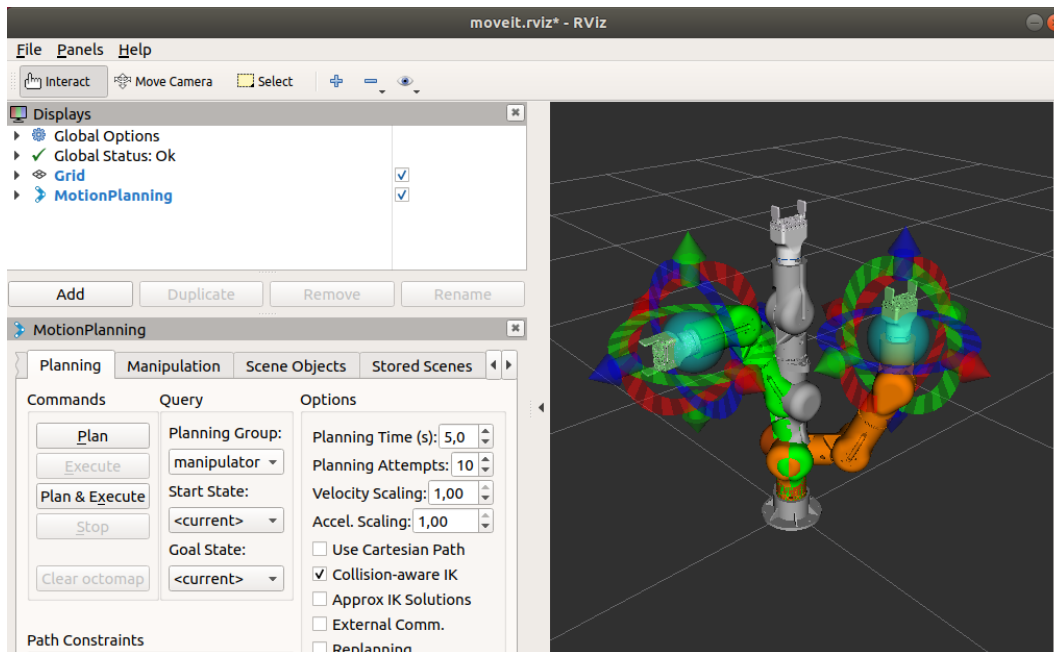


Figure 6.12: MoveIt! GUI

6.3.3. Pick and place application

In this section the aim is to make a program of pick and place to move a box from one position to another. This program is written in Python language and the result will be the planning and execution of the pick and place motion law using MoveIt!. In this way, the motion has been planned just considering the kinematics of the robot, the dynamics will be considered once the planned motion is used to control a real robot or the virtual model of the robot in Gazebo.

Below there is a general explanation of the code A.15 on page 209.

First of all, to write the program a list of modules must be imported to have access to different useful functionality.

```

1 from copy import deepcopy
2 from moveit_msgs.msg import *
3 from moveit_commander import *
4 from trajectory_msgs.msg import JointTrajectoryPoint
5 import tf
6 from os import path
7 from time import strftime, localtime
8 from geometry_msgs.msg import *
9 import matplotlib.pyplot as plt
10 from matplotlib.figure import Figure, Axes, rcParams
11 from math import pi

```



```

12 from tf.transformations import quaternion_from_euler
13 import os

```

Initialize the moveit_commander and a rospy node called plan_motion:

```

1 # Initialization of Moveit!
2 rospy.logininfo('Starting the Initialization')
3 roscpp_initialize(sys.argv)
4 # Initialization of the Node
5 rospy.init_node('plan_motion', anonymous=True, log_level=rospy.INFO)

```

Now, instantiate the robot object of the RobotCommander class which provides information on the robot's kinematic model and the robot's current joint states. Instantiate also the scene object of the PlanningSceneInterface class, this provides a remote interface for getting, setting, and updating the robot's internal understanding of the surrounding world.

Then, instantiate two objects of the MoveGroupCommander class, these objects are interfaces for the planning groups defined in the MoveIt! setup assistant which are the manipulator and gripper groups for the robot of this project. These interfaces will be used to plan and execute motions of the manipulator and gripper groups.

To perform path planning the user assign RRT algorithm for the manipulator and RRT-Connect algorithm for the gripper. Furthermore, the number of time path planning is performed from scratch before the shortest solution is returned is set to 1000 and the goal tolerance is set to 0.01.

```

1 robot = RobotCommander()
2 scene = PlanningSceneInterface()
3
4 planner = "RRT"
5 group = MoveGroupCommander("manipulator")
6 group.set_pose_reference_frame("base")
7 group.set_planner_id(planner)
8 group.set_num_planning_attempts(1000)
9 group.allow_replanning(False) # Allow the replanning if there are changes
   in environment
10
11 arm = "tool0"
12
13 group_gripper = 'gripper'
14 group_gripper2 = MoveGroupCommander("gripper")
15 group_gripper2.set_planner_id('RRTConnect')
16 group_gripper2.set_num_planning_attempts(1000)

```

```

17 group_gripper2.allow_replanning(False)
18 group_gripper2.set_goal_tolerance(0.01)

```

To perform this task the user set an home position for the dummy link called tool0 placed at the end of link3. The robot will go in the home position before performing the task and once the task is completed. The position also contains information on the orientation of the dummy link which is defined so that the gripper points downward and in this way it has already the right orientation to pick the box. The orientation is defined with Euler angles which are more intuitive but then a conversion in quaternions is needed.

```

1 # Home position
2 quaternion = tf.transformations.quaternion_from_euler(pi, 0, 0)
3 home_pos = [0.300, 0, 0.4, quaternion[0], quaternion[1], quaternion[2],
              quaternion[3]]

```

In the program is also defined the TestBox class, different attributes are considered for each object of this class like the name, its position in the space and its dimensions. In particular a box with each side of 5 cm is defined. Furthermore, there are different functions for this class like `add_object` to add a box in the space, `remove_object` to remove it, `attach_object` which is associated to the gripper group and will be needed by the robot to grasp the box and finally `detach_object` that will be useful to release the box.

```

1 class TestBox:
2
3     def __init__(self, name, x=0.0, y=0.0):
4         self.name = name
5         self.x_dim = 0.05
6         self.y_dim = 0.05
7         self.z_dim = 0.05
8         self.pose_msg = PoseStamped()
9         self.pose_msg.header.frame_id = "base"
10        self.pose_msg.pose.position.x = x
11        self.pose_msg.pose.position.y = y
12        self.pose_msg.pose.position.z = 0
13
14        def add_object(self):
15            rospy.sleep(0.5)
16            scene.add_box(self.name, self.pose_msg, size=(self.x_dim, self.
17            y_dim, self.z_dim))
18            rospy.sleep(0.5)
19
20        def remove_object(self):
21            if scene.get_attached_objects(self.name):

```

```

21         rospy.logerr('Object attached: check it please!')
22     else:
23         scene.remove_world_object(self.name)
24
25     def attach_object(self, arm):
26         touch_links = robot.get_link_names(group_gripper)
27         scene.attach_box(arm, self.name, touch_links=touch_links)
28         rospy.sleep(1.0)
29
30     def detach_object(self, arm):
31         scene.remove_attached_object(arm, self.name)
32         rospy.sleep(0.5)

```

Below, different functions are defined. The first one is called `getting_joints_from_plan`, once a trajectory is planned this function allows to pick the value of the joints at the final time instant so, provide the list of joints value at the end of the trajectory.

Then, there is the function: `create_robotstate`. This function allows to plan the motion starting from the final state of the robot so, it allows to plan different motions in sequence.

The function `evaluate_time` provides simply the amount of time needed to plan a certain motion.

```

1 def getting_joints_from_plan(plan):
2     # type: (RobotTrajectory) -> list
3     """
4     Provide the joints of the last point of the trajectory
5     :param plan: RobotTrajectory msg: Plan of a trajectory
6     :type plan: RobotTrajectory
7     :rtype: list
8     :return: List of joints of the end of the trajectory
9     """
10    positions = plan.joint_trajectory.points[-1] # type:
11    JointTrajectoryPoint
12    return positions.positions
13
14 def create_robotstate(plan):
15     """
16     Return a RobotState() msg of the end of a trajectory
17     :param plan: RobotTrajectory msg of the trajectory - planning
18     :type plan: RobotTrajectory
19     :rtype: RobotState
20     :return: robot_state: RobotState msg of the trajectory
21     """
22     if plan.joint_trajectory.points:

```

```

22     # Creating a RobotState for evaluate the next trajectory
23     joint_state = JointState()
24     joint_state.header.stamp = rospy.Time.now()
25     joint_state.name = plan.joint_trajectory.joint_names
26
27     positions = getting_joints_from_plan(plan)
28     joint_state.position = positions
29     robot_state = RobotState()
30     robot_state.joint_state = joint_state
31     return robot_state
32
33 else:
34     raise RuntimeError("Error in creating the robotstate: points
35 empty")
36
37 def evaluate_time(plan, info=''):
38     """
39     It returns the time duration of the trajectory
40     :param plan: Plan msg of the trajectory
41     :type plan: RobotTrajectory
42     :param info: info about the time
43     :type info: str
44     :rtype: float
45     :return: duration time of the trajectory
46     """
47     # Check if the plan is not empty
48     if plan.joint_trajectory.points:
49         p_last = plan.joint_trajectory.points[-1] # type:
50         JointTrajectoryPoint
51         duration = p_last.time_from_start.to_sec()
52         rospy.loginfo('Estimated time of planning {}: {} s'.format(info,
53 duration))
54         return duration
55     else:
56         rospy.logwarn('The plan is empty')
57         duration = 2000.0 # penalty
58         return duration

```

Now, there are the functions which define the different kind of motions that the robot will be able to perform. The home function allows to make the manipulator go in the home position.

The function picking plans two motions: with the pick motion the robot goes above the box in a position in which the gripper is able to grasp the box without compenetration and then with the homing motion the robot goes back in the home position.

The function placing also plans two motions: with the plan motion the robot goes from the home position to where the box must be placed and then with the return home motion it goes back in the home position.

Finally, the functions opening and closing make the fingers of the gripper open or close to release or grasp the box.

```

1 def home():
2     rospy.loginfo('home')
3     group.set_pose_target(home_pos, arm)
4     plan = group.plan()
5     home_duration = evaluate_time(plan, "homing")
6     group.execute(plan)
7     group.stop()
8     return home_duration
9
10 def picking(obj, arm, info=''):
11     # type: (TestBox, str, str) -> List[float, RobotTrajectory,
12     RobotTrajectory]
13     """
14     Wrapper for picking
15     :param obj: Object to pick
16     :param arm: Arm used
17     :param info Info about what is doing
18     :rtype: list[float, RobotTrajectory, RobotTrajectory] or tuple[float,
19     RobotTrajectory, RobotTrajectory]
20     :return: Duration time for picking and RobotTrajectories for picking
21     """
22     pose_P = deepcopy(obj.pose_msg)
23     pose_P.pose.position.z += 0.15
24     pose_P.pose.orientation.x = quaternion[0]
25     pose_P.pose.orientation.y = quaternion[1]
26     pose_P.pose.orientation.z = quaternion[2]
27     pose_P.pose.orientation.w = quaternion[3]
28
29     group.set_pose_target(pose_P, arm)
30     pick = group.plan()
31     # Evaluate the time for picking
32     t1 = evaluate_time(pick, info + "_t1")
33     if pick.joint_trajectory.points:
34         # Creating a RobotState for evaluate the next trajectory
35         robot_state = create_robotstate(pick)
36         group.set_start_state(robot_state)
37         group.set_pose_target(home_pos, arm)
38         homing = group.plan()

```

```

37     # Evaluate the duration of the planning
38     t2 = evaluate_time(homing, info + "_t2")
39     return [(t1 + t2), pick, homing]
40 else:
41     rospy.logerr('Planning failed')
42     pass
43
44 def closing():
45     rospy.loginfo('closing')
46     joint_goal = group_gripper2.get_current_joint_values()
47     joint_goal[0] = 0.0
48     group_gripper2.go(joint_goal, wait=True)
49     group_gripper2.stop()
50
51 def placing(obj, info=''):
52     # type: (TextBox, str) -> List[float, RobotTrajectory, RobotTrajectory]
53     place_pos = [0, 0.3, 0.1, quaternion[0], quaternion[1], quaternion[2],
54                 quaternion[3]]
55     group.set_pose_target(place_pos, arm)
56     placePlan = group.plan() # type: RobotTrajectory
57     # Evaluate the time of the trajectory
58     t1 = evaluate_time(placePlan, info + "_t1_placing")
59     if placePlan.joint_trajectory.points:
60         # Creating a RobotState for evaluate the next trajectory
61         group.clear_pose_targets()
62         place_state = create_robotstate(placePlan)
63         group.set_start_state(place_state)
64         group.set_pose_target(home_pos, arm)
65         return_home = group.plan() # type: RobotTrajectory
66         t2 = evaluate_time(return_home, info + "_t2_placing") # type:
67         float
68         return [(t1 + t2), placePlan, return_home]
69     else:
70         raise RuntimeError("Error: Planning failed for placing")
71
72 def opening():
73     rospy.loginfo('opening')
74     joint_goal = group_gripper2.get_current_joint_values()
75     joint_goal[0] = 0.025
76     group_gripper2.go(joint_goal, wait=True)
77     group_gripper2.stop()

```

Finally, there is the function run in which the planning and execution of the pick and place task is performed exploiting what has been defined above.

Once all the objects in the space are removed a box is added in the space in the position: (x=0.3m, y=0, z=0). The sequence of motion planned and executed are: the robot goes in the home position by calling the function home, with the function pick it goes above the box and very close to it, using the function attach_object the box is attached to the gripper and so can be picked up. Then, with the function closing the fingers of the gripper close and with homing the robot goes back in the home position with the box attached to the gripper. Now, there is the place function so the robot goes in the position where the box must be placed, with opening the fingers of the gripper open and with detach_object the box is released. Finally, the robot goes back in the home position with the function return home.

```

1 def run():
2     # Remove detached object
3     if scene.get_attached_objects():
4         scene.remove_attached_object(arm)
5
6     # Remove all the objects
7     scene.remove_world_object()
8     rospy.sleep(1.0)
9
10    # Add the test box
11    B = TestBox('box', x=0.3, y=0)
12    rospy.loginfo('Placing box')
13    rospy.sleep(0.5)
14
15    # Setting up the test box
16    B.add_object()
17    rospy.sleep(2.0)
18
19    # Going home
20    group.set_start_state_to_current_state()
21    home()
22
23    # # Evaluate the time for the arm cycle
24    (t1, pick, homing) = picking(B, arm, info="pick\t")
25    home_robotstate = create_robotstate(homing)
26    group.set_start_state(home_robotstate)
27    (t2, place, return_home) = placing(B, info="place\t")
28
29    # Create the initial state after placing the tube
30    return_home_state = create_robotstate(return_home)
31    group.set_start_state(return_home_state)
32

```

```

33     # Execute Picking
34     group.execute(pick)
35     group.stop()
36
37     # Attach Test box
38     B.attach_object(arm)
39     closing()
40
41     group.execute(homing)
42     group.stop()
43
44     # Execute Placing
45     group.execute(place)
46     group.stop()
47     opening()
48     B.detach_object(arm)
49
50     group.execute(return_home)
51     group.stop()

```

In the Figure 6.13 is possible to read on the terminal the time needed to plan the different motions. Around 3 seconds are required to plan the motion of the robot that move from the initial position in which it is vertical to the home position. The function picking is composed by two motions as written above and both movements require 2 seconds to be planned. Then, also the function placing is composed by two motions and in this case both movements take 2.6 seconds to be planned.

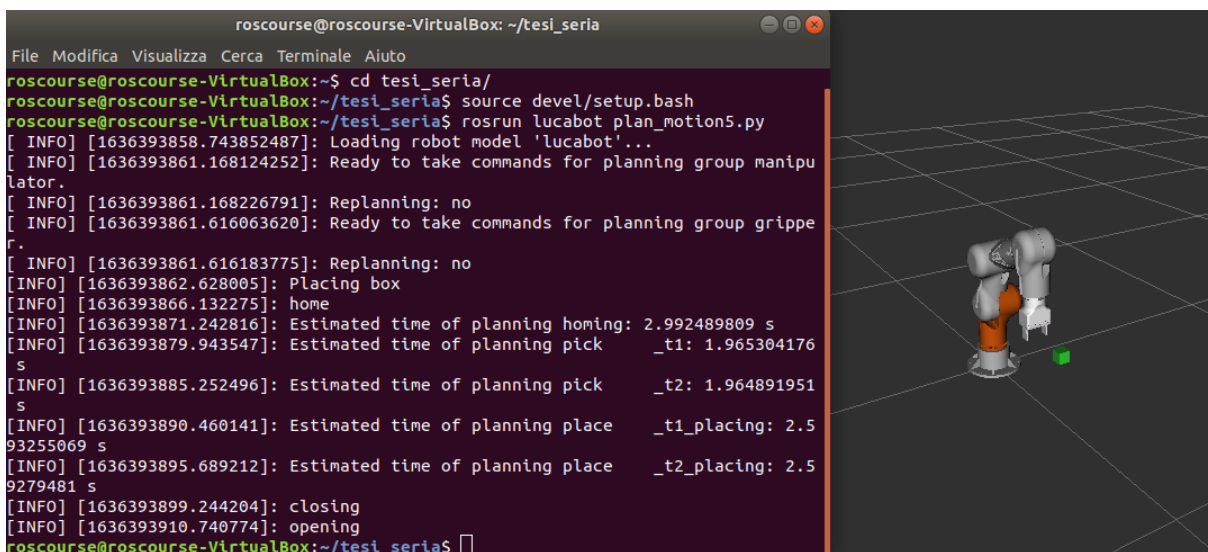


Figure 6.13: Pick and place

Below, there are the plots of the motion law of different movements: the pick and homing motions which are defined in the function picking. Then, the place and return home motions defined in the function placing.

These plots represent how the position, velocity and acceleration of the revolute joints of the manipulator change to perform the movements explained above.

For instance, from the plot of the pick motion is highlighted that just the joints j3 and j5 change significantly to perform this motion.

Furthermore, looking at both the pick and homing plots the behaviour of the joints position is the same but during the pick phase the motion is performed in one direction instead during the homing phase the motion goes in opposite direction. In fact, the initial and final positions of the pick motion are respectively the final and initial positions of the homing motion. This can be pointed out also looking at the velocity and acceleration of the two plots, the behaviour of these quantities is the same but just with opposite sign. This characteristic is present also for the place and return home motions.

Finally, looking at the velocity plots the limit of 1 rad/s defined in the xacro file is respected. Looking at the acceleration plots also the limit of 1 rad/s^2 which is set by default is respected.

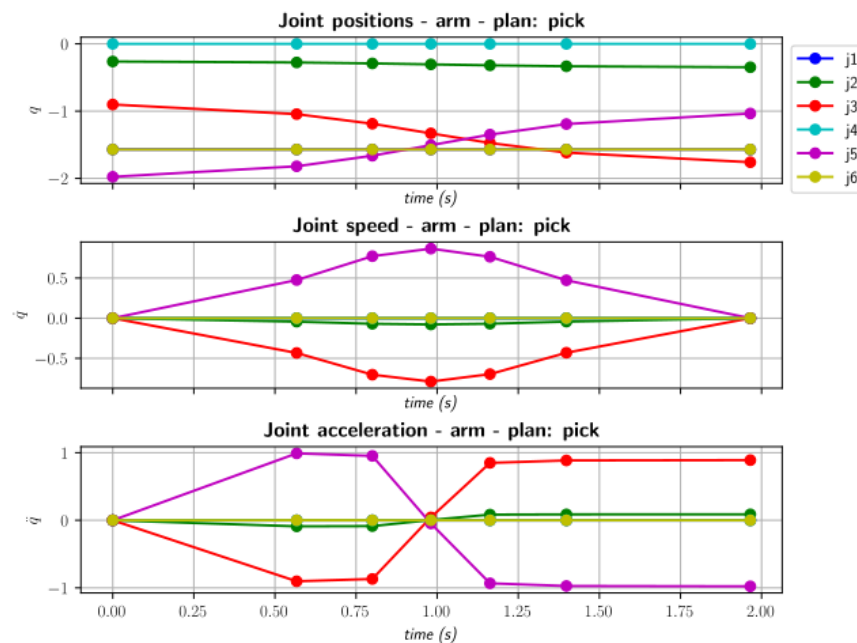


Figure 6.14: Pick

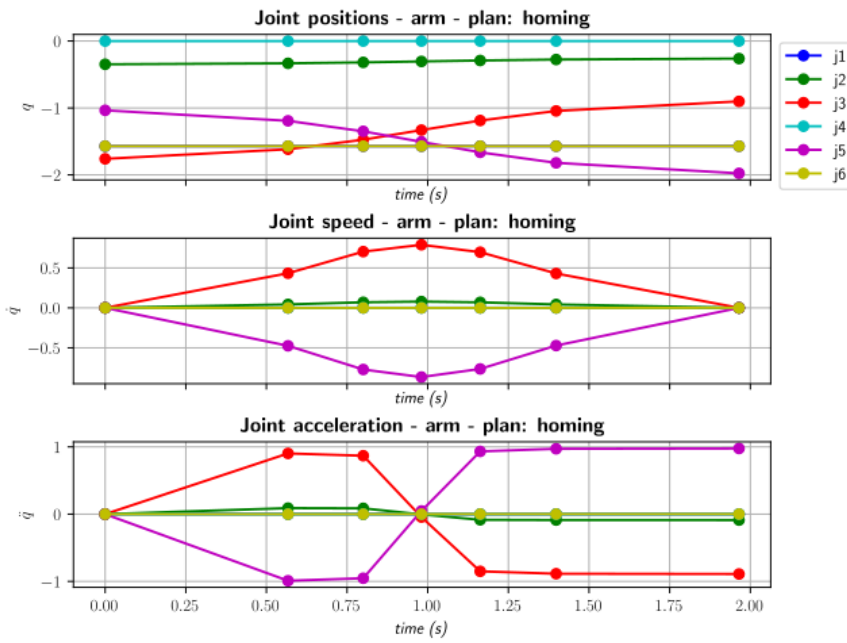


Figure 6.15: Homing

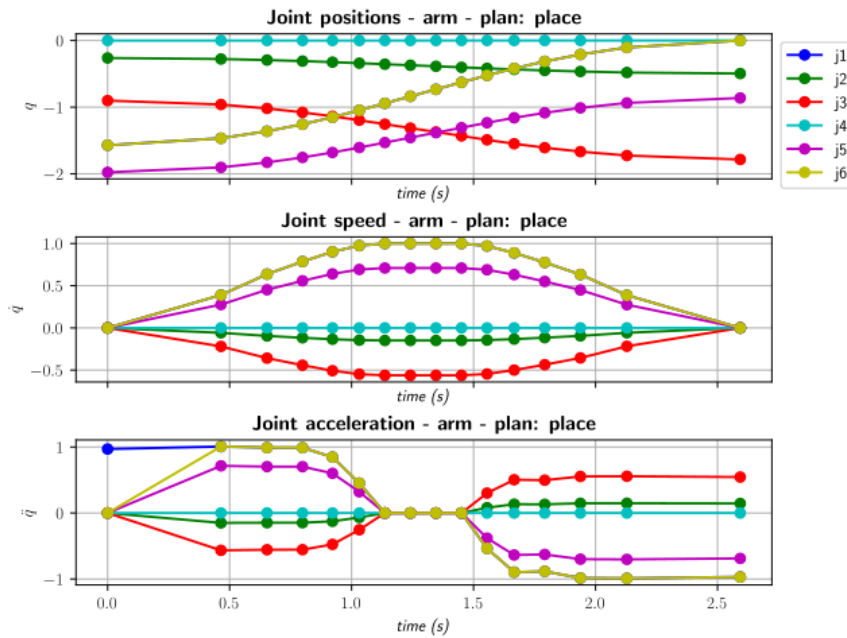


Figure 6.16: Place

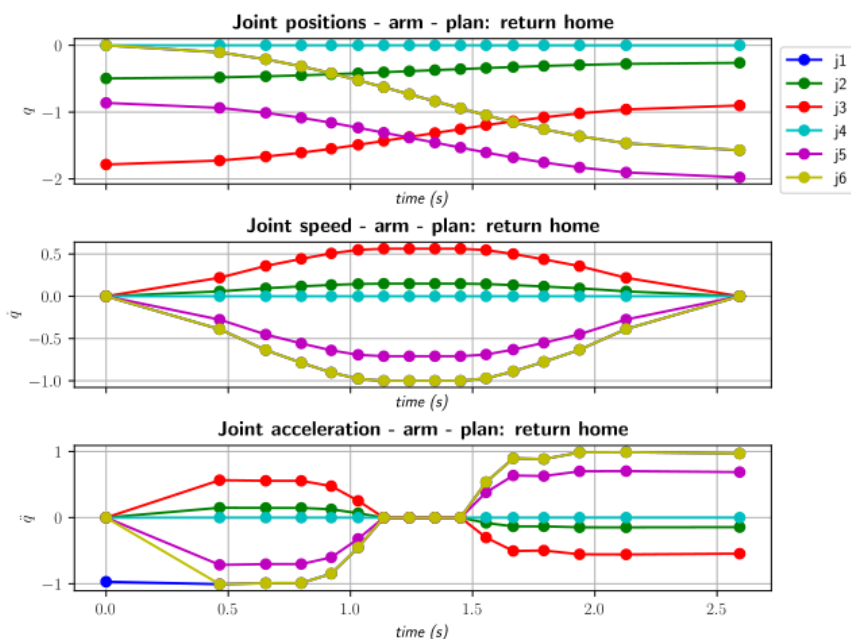


Figure 6.17: Return home

6.4. Gazebo modelling

Gazebo is a 3D dynamic simulator and allows to test different programs on the virtual robot [8]. Since also the dynamic of the robot is considered the behaviour of the actual robot can be reproduced. The presence of this simulation environment is one of the advantages of offline programming since it allows to test and refine the programs on the virtual robot without stopping the actual one and so keeping it active without wasting time and money. Here is explained how to launch and control the multibody model in Gazebo both for the robot model with the gripper and for the one which represents the real robot.

6.4.1. Robot model with gripper

Xacro file in Gazebo

First of all, there is the need of adding the following lines of code to the xacro file of the robot model (A.5 on page 189) to allow the interaction between ROS and Gazebo giving the chance of controlling the robot. This is the plug in that has been added:

```

1 <gazebo>
2   <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">

```

```

3   <robotNamespace>/lucabot</robotNamespace>
4   </plugin>
5 </gazebo>

```

Then, to consider the dynamic of the robot, mass and inertia of each link have been added as explained in the section above: Manipulator model.

Furthermore, for each non-fixed joint of the robot a transmission is needed. This allows to describe the relationship between actuator and joint. For instance, in the following lines is present the transmission related to the first revolute joint (r1). The hardware interface is an EffortJointInterface because the controller related to this joint is of type `effort_controllers/JointPositionController` as it will be shown later. Finally, in the actuator tag the mechanical reduction is defined at the joint actuator transmission and it is set to 35 for each joint.

```

1 <transmission name="tran1">
2   <type>transmission_interface/SimpleTransmission</type>
3   <joint name="r1">
4     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
5   </joint>
6   <actuator name="motor1">
7     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
8     <mechanicalReduction>35</mechanicalReduction>
9   </actuator>
10 </transmission>

```

Gazebo environment

In this section is explained how to build the Gazebo environment and, in particular, `lucabot.world` (A.12 on page 207) is the file in which the environment is defined.

In the environment static objects can be included. For instance, a table has been added on which the robot is placed. Furthermore, the sun is included to have a good lighting. This kind of environment can be seen in Figure 6.19.

ROS Control + Gazebo

To control the virtual robot in the Gazebo simulation ROS control is exploited.

As is shown in Figure 6.18 the controllers take as input the joint state data coming from the Joint State interface and typically it uses a control loop feedback mechanism like a

simple PID control logic. The other input of ROS control is the motion law of the joints planned in MoveIt!. These inputs allow to obtain the error between the desired joint values and the actual one. Instead, the output of ROS control is an effort sent to actuate the joints.

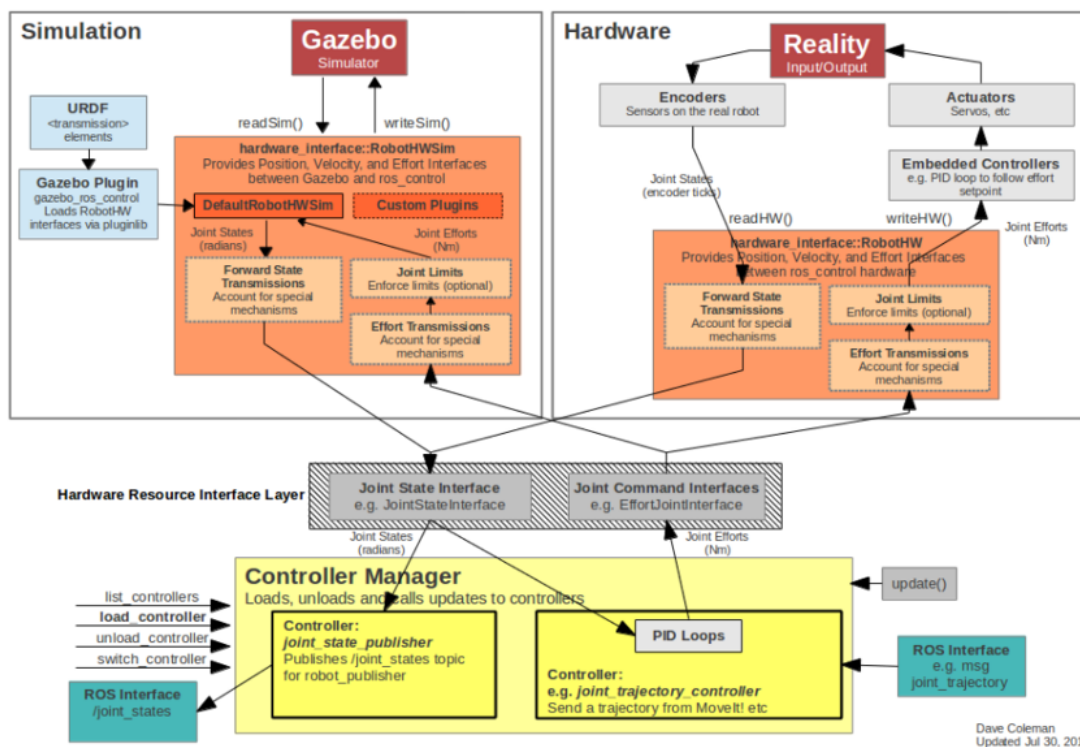


Figure 6.18: ROS Control + Gazebo

To define the controllers that are needed in the Gazebo environment two configuration files must be created:

robot_controllers.yaml

This file (A.13 on page 207) presents the type of controllers and the respective gains for each joint.

In the following lines of code it's shown how to define the controller for the first revolute joint, the other controllers for the joints of the group manipulator are not reported here because are defined in the same way.

```

1 joint1_position_controller:
2   type: effort_controllers/JointPositionController
3   joint: r1
4   pid: {p: 10000.0, i: 0.5, d: 1.0}

```

The type of controller: `effort_controllers/JointPositionController` takes as input the position of the joint and gives in output the force/torque to the joint.

Then, for the gripper, the controller is defined just for the joint of the left finger, this is due to the presence of mimic joint.

These are the lines of code associated to this controller:

```

1 gripper_controller:
2   type: position_controllers/JointTrajectoryController
3   joints:
4     # Check "test" is the name parameter
5     - test_body_finger_l_joint
6   constraints:
7     goal_time: 0.6
8     stopped_velocity_tolerance: 0.05
9     gripper_finger_joint: { trajectory: 0.2, goal: 0.2 }
10  stop_trajectory_duration: 0.5
11  state_publish_rate: 125
12  action_monitor_rate: 10

```

As shown above the controller is of type: `position_controllers/JointTrajectoryController`.

There are also some parameters that must be specified for this type of controller like:

- goal time: the gripper follows the trajectory with success if it reaches the goal in 0.6 seconds.
- stopped velocity tolerance: the maximum velocity at the end of the trajectory for the joint to be considered stopped.
- joint goal: position tolerance to reach the goal.
- joint trajectory: position tolerance throughout the trajectory.
- stop trajectory duration: time to bring current state to a stop.
- state publish rate: frequency of publication of controller state.
- action monitor rate: frequency at which the action goal status is monitored.

gripper_gains.yaml

In this file (A.14 on page 208) there are simply specified the gains of the controllers for the gripper. Here below just the gains of the left finger joint are present because the one for the right finger are the same.

```

1 pid_gains:

```

```
2 test_body_finger_l_joint:  
3   p: 200.0  
4   i: 0.1  
5   d: 0.0  
6   i_clamp: 0.2  
7   antiwindup: false  
8   publish_state: true
```

Gazebo simulation

In the previous sections different files have been defined, these are exploited and launched in another file called `lucabot.launch` which is reported in the listing A.11 on page 206. In this file the environment is defined by launching `lucabot.world`, the robot model is considered by exploiting the `xacro` file which has been modified to adapt it to the Gazebo simulation. Finally, also the controllers defined above are loaded in this file.

In the Figure 6.19 is shown the robot in the Gazebo environment.

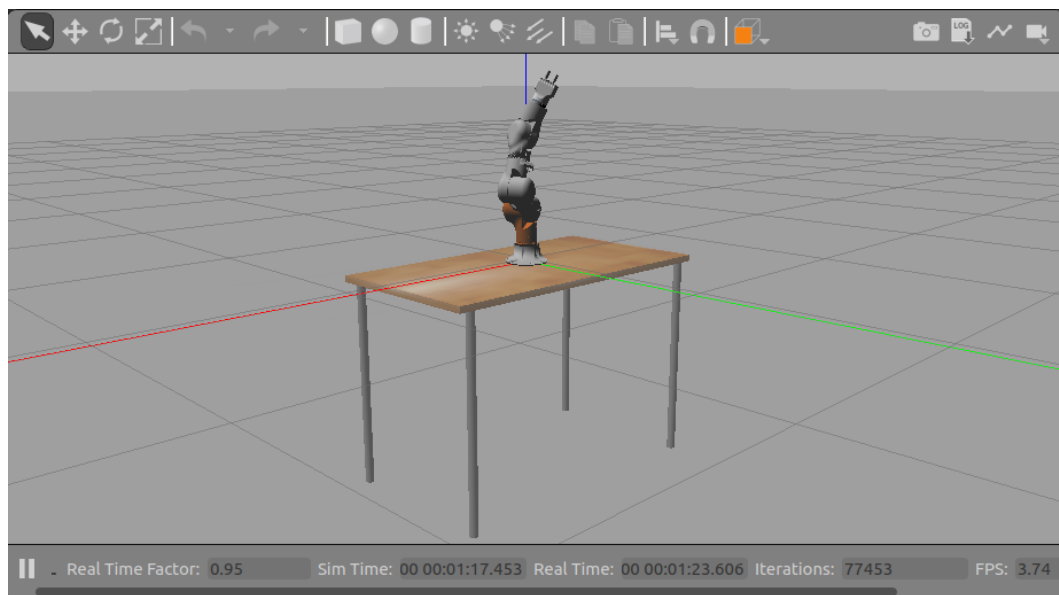


Figure 6.19: Gazebo simulation

Then, it's possible to change the position of the revolute joints of the manipulator group by exploiting `rqt` tool (Figure 6.20). This tool can be used to publish the value of the joint on the topic associated to the specific joint. For instance, to change the position of the first revolute joint it's simply required to publish the desired value on the topic `joint1_position_controller`.

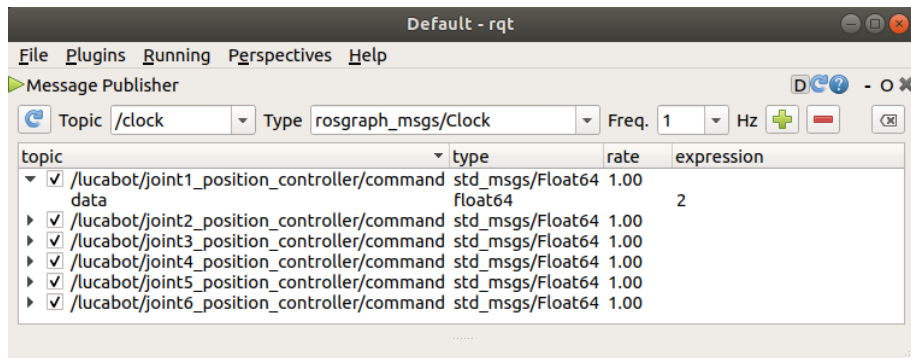


Figure 6.20: rqt

Finally, to control the joint of the gripper, the tool joint trajectory controller, shown in Figure 6.21, can be used. First of all, select the controller (grripper controller) and enable it. Then, it's possible to select the value. In this case, for the gripper, the controller acts only on the left finger joint since the right finger joint moves simultaneously and in the same way thanks to the mimic joint.

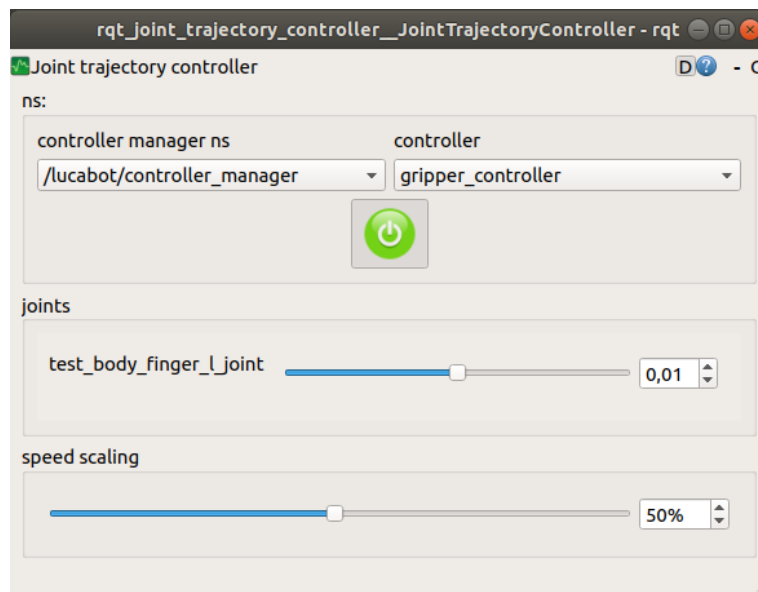


Figure 6.21: joint trajectory controller

6.4.2. Robot model of real robot

As explained here 6.4.1 on page 141 also for the model of the real robot a plugin to allow the interaction between ROS and Gazebo is needed and also the transmissions related to each joint must be added.

Furthermore some additional modifications must be performed to the xacro file of the real robot (A.16 on page 218), taking in consideration the sources [12] and [10]:

- Add damping to each joint

```
1 <dynamics damping="1.0"/>
```

- Add friction forces in order to have realistic dynamics

```
1 <xacro:macro name="lucabot_gazebo" params="name">
2   <gazebo reference="{name}_body">
3     <material>Gazebo/White</material>
4     <mu1>0.2</mu1>
5     <mu2>0.2</mu2>
6   </gazebo>
7 </xacro:macro>
```

- Overwrite the `ros_controller.yaml` file automatically generated by the MoveIt! setup assistant with the following snippet (A.20 on page 227). In generating this file pay attention to the namespace in which you work, in this case it is called `lucabot`. In this file the controller is defined for each joint, in particular the type is `effort_controllers/JointTrajectoryController` so, ROS control takes in input the position+velocity trajectory and give in output the torque that actuate each joint. Also the gains are defined, PID controller are used with the following values: p:100, d:1, i:0.1.

```
1 # MoveIt-specific simulation settings
2 moveit_sim_hw_interface:
3   joint_model_group: controllers_initial_group_
4   joint_model_group_pose: controllers_initial_pose_
5 # Settings for ros_control control loop
6 generic_hw_control_loop:
7   loop_hz: 300
8   cycle_time_error_threshold: 0.01
9 # Settings for ros_control hardware interface
10 hardware_interface:
11   joints:
12     - r1
13     - r2
14     - r3
15     - r4
16     - r5
17   sim_control_mode: 1 # 0: position, 1: velocity
18 # Publish all joint states
19 lucabot:
20   # Creates the /joint_states topic necessary in ROS
```

```

21   joint_state_controller:
22     type: joint_state_controller/JointStateController
23     publish_rate: 50
24   lucabot_arm_controller:
25     type: effort_controllers/JointTrajectoryController
26     joints:
27       - r1
28       - r2
29       - r3
30       - r4
31       - r5
32     gains:
33       r1: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
34       r2: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
35       r3: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
36       r4: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
37       r5: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
38
39     constraints:
40       goal_time: 2.0
41       state_publish_rate: 25
42
43   controller_list:
44     - name: lucabot/lucabot_arm_controller
45       action_ns: follow_joint_trajectory
46       type: FollowJointTrajectory
47       default: true
48       joints:
49         - r1
50         - r2
51         - r3
52         - r4
53         - r5
54

```

- Adjust the file `ros_controller.launch` by loading the controllers in this way and paying attention to the namespace (A.23 on page 231):

```

1   <!-- Load the controllers -->
2   <node name="controller_spawner" pkg="controller_manager" type="
3     spawner" ns="/lucabot" respawn="false"
4     output="screen" args="--namespace=/lucabot
5       joint_state_controller
6       lucabot_arm_controller
       --timeout 20"/>

```

7

Now, it is possible to launch the file `gazebo.launch` (A.22 on page 231) to obtain the multi-body model of the real robot in the Gazebo simulator. This file is generated automatically by the MoveIt! setup assistant and running the `rqt_joint_trajectory_controller` a GUI appears which enable the user to move each joint as shown in Figure 6.22.

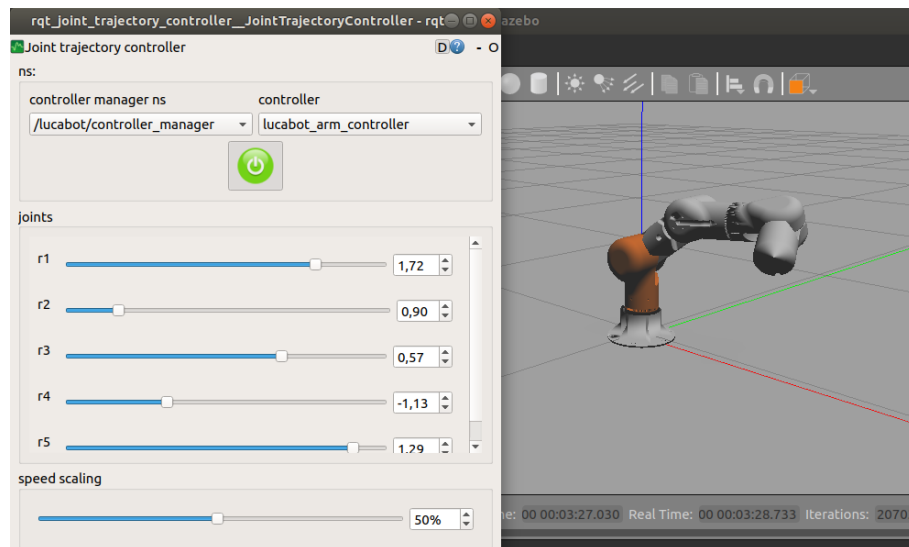


Figure 6.22: Real robot model in Gazebo

6.5. Dynamic simulation integrating Gazebo into MoveIt!

In this section the aim is to launch the real robot model in MoveIt! and Gazebo at the same time. In this way a task can be planned in MoveIt! and can be executed by the multibody model in Gazebo. So, the user can try to plan a task and verify if the multibody model which reproduce the dynamic behaviour of the real robot is able to perform it.

Furthermore, we will see how to get the torque that actuate each joint during the execution of a movement.

The different adjustments made in the section 6.4.2 are needed also to integrate Gazebo with MoveIt!. To execute these two environments at the same time the file `demo_gazebo.launch` obtained automatically by the MoveIt! setup assistant must be launched (A.24 on page 232). By the way, a little change is performed: the parameter source list of `joint_state_publisher` must be prefixed by the namespace.

```
1 <node name="joint_state_publisher" pkg="joint_state_publisher" type="
  joint_state_publisher" unless="$(arg use_gui)">
```

```

2   <rosparam param="source_list">[move_group/fake_controller_joint_states]
   </rosparam>
3   <rosparam param="source_list">[lucabot/joint_states]</rosparam>
4 </node>
5 <node name="joint_state_publisher" pkg="joint_state_publisher_gui" type="
   joint_state_publisher_gui" if="$ (arg use_gui) ">
6   <rosparam param="source_list">[move_group/fake_controller_joint_states]
   </rosparam>
7   <rosparam param="source_list">[lucabot/joint_states]</rosparam>
8 </node>

```

So, launching the `demo_gazebo.launch` file the model pop up both in MoveIt! and Gazebo. In particular, we make the robot go from the initial pose shown in Figure 6.23 to the final one shown in Figure 6.24. For these two poses the information on the joints value and effort to keep the configuration are reported in the following Tables 6.4 and 6.5. To perform this movement in the `xacro` file a limit value of joint velocity is set at 1 rad/s and in the `joint_limits.yaml` file obtained from the MoveIt! setup assistant the limit in the acceleration is set at $0,3927 \text{ rad/s}^2$

Joints name	Initial joint values [deg]	Initial effort [Nm]
r1	16.73	$8.71e - 05$
r2	87.11	-3.37
r3	-29.90	-1,16
r4	148.95	0.08
r5	62.18	-0.02

Table 6.4: Initial pose information of a simple movement

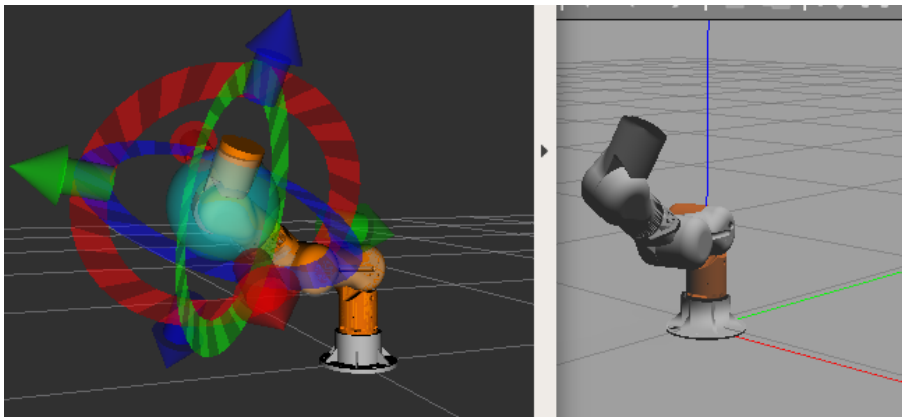


Figure 6.23: Initial pose

Joints name	Final joint values [deg]	Final effort [Nm]
r1	-87.81	0.00
r2	74.60	-2.01
r3	-74.91	0.12
r4	-29.99	0.00
r5	-48,62	0.13

Table 6.5: Final pose information of a simple movement

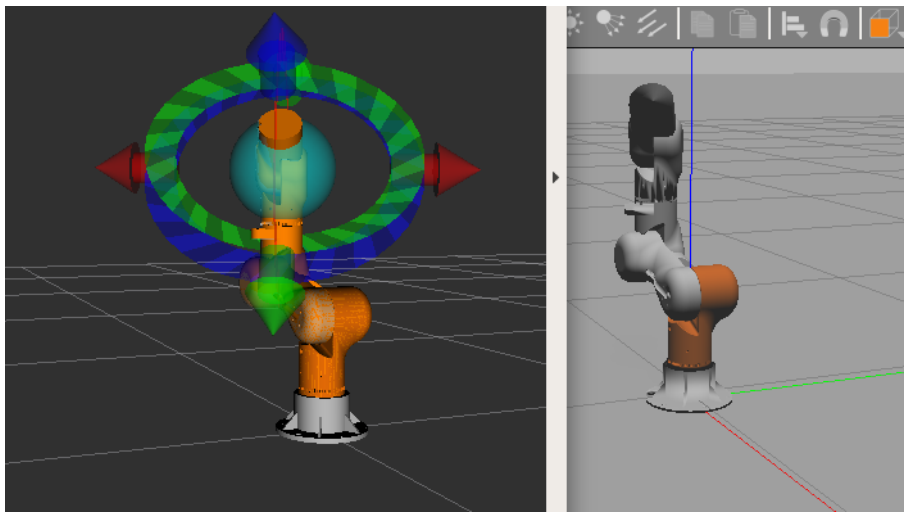


Figure 6.24: Final pose

Finally, we plot the torque that actuate each joint to go from the starting position to the final one. To do this `rqt` is used and in particular is considered the topic `joint_states`. From this topic the message related to the effort is plotted which is an array of 5 elements since 5 are the joints of the robot.

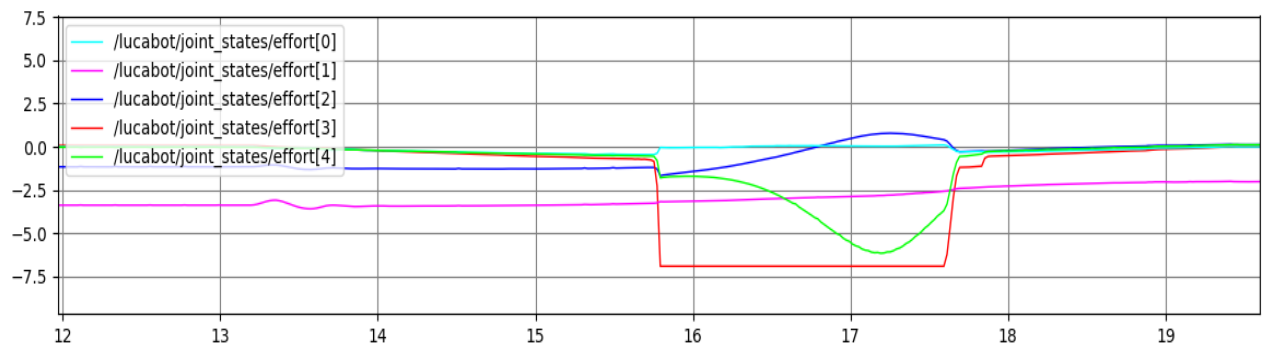


Figure 6.25: Torque plot

From the Figure 6.25 which represents the torque plot, it's possible to notice that the limits effort of each joint are respected throughout the whole movement. In particular, it is evident that joint 4 reaches the limit value of effort equal to 6.9 to perform the movement. The limits effort are assigned in the xacro file of the model of the real robot and they are reported in the Table 6.3.

7 | Results and Conclusions

7.1. Results

7.1.1. Cost estimation

The estimation of the total cost of parts and items used to build the robot are given hereafter. The prices reported are to be considered as at the time of writing this document.

Prices are reported as someone wants to individually source the components to build its own robot, as happens in many DIY projects that can be found on the Internet.

In Table 7.1 a summary of the cost VAT included is provided, while in the following a more detailed overview for each main category is given.

Category	Total cost
Motors and drivers	770€
3D printing overheads	85€
Material	145€
Bearings	80€
Hardware and fasteners	50€
Potentiometers and ADCs	155€
Spares	150€
TOTAL	1435€

Table 7.1: Cost per category

Stepper motors and stepper drivers

As said elsewhere in this thesis work, motors and stepper drivers were given free of charge as samples from the firm RTA. Anyway, just doing an internet research of the codes, the product page on RTA eStore appears also containing the cost of the item. In Table 7.2 are reported the prices for one item of each code. Prices are to be considered as at the time of writing this document and are given without VAT.

Joints	Motor code	Price
J1	103-H7123-5040	39.10€+VAT
J2	103-H7123-5010.B	118.50€+VAT
J3	103-H5210-4512.B	113.60€+VAT
J4	103-H5205-4240	22.50€+VAT
J5	103-H5205-4240	22.50€+VAT
J6	103-H5205-4240	22.50€+VAT
Total		338.70€+VAT

Table 7.2: Motor prices

BSD 02.V stepper driver board were used and a total of 6 boards are needed. The unit price for purchasing quantity bigger than 2 pieces is 48.80€+VAT, for a total of 292.80€+VAT.

We reckon that these motors are very expensive for such an application. Anyway, these motors and drivers are industry grade and hence of high quality. Consumer stepper motors can be sourced elsewhere to limit the cost but feature like the brake can be lost.

3D printing overheads

To better estimate the cost of the realized prototype, overhead costs should be considered. Among these costs there are for sure depreciation and cost for electricity.

About depreciation we can imagine that a 3D printer can have an expected lifespan of five years. After this period it is reasonable to think that technological obsolescence will occur nullifying the market value of the printer. We can also imagine, similarly to what actually did, to use the 3D printer 5 days a week continuously for 24 hours (weekend is not available due to lack of surveillance for two full days). Therefore, a total of 31200h of lifespan is expected. The cost of the 3ntr A4v3 printer used is not known and could not be found on the Internet but, being an Industry grade machine, we can hypothesize that the price would have ranged around 10.000€. The Prusa MK3S+ used has a price of around 1000€. Even if the Prusa is capable of realizing all the parts of the robot (it has big enough build volume and high enough precision), we consider a professional 3D printer in the range of 3000€ for the calculations. Hence, the hourly cost for depreciation is 0.096€/h. Total printing time is estimated around 400h and thus the cost is 38€.

For what concerns electricity cost, since a power meter was not available, an estimation on the consumed power is done considering the data available on the machine. The manufacturer state a peak power consumption of 1150W, and it is reasonable to consider an average power consumption (once the heated bed and chamber are at regime temperature)

of 50% the peak one (maintain the chamber heated required much power). Considering 0.19€/kWh the total cost for energy is 43.70€.

3D printing filament

Estimation of the price for the ABS filament used is done considering an average price of filament spools found on the Internet and for the quantity the sum of the used material for each part. Total weight of filament is almost 2.5kg (not including support material). The 3ntr printer used SSU0 filament to print support material. We estimate that around 1kg of support material to be generated by the slicer. Since no information about this type of polymer was found (available retail channel and price) we consider to use ABS filament also for support material, therefore approximately 3.5kg of ABS filament are required. Filament spools are sold typically in 1kg so at least 4 spools are needed. Considering an indicative price of 20€/kg, filament would cost 80€ VAT included. Estimation of material cost depends on where the filament is sourced, the quality of the material and the chosen color.

For other parts PETG and TPU were used, respectively around 25€/kg and 40€/kg. Considering one spool of 1kg each (even if the whole spool won't be used), we need to add 65€ VAT included.

3D printing filament would cost around 145€ VAT included.

Bearings

The cost of bearings depends on where they are sourced and their quality. Considering the application, high quality bearings are overkilled so branded ones are not strictly necessary. Purchasing unbranded bearings on the mainstream eCommerce is fine. In Table 7.3 are reported the prices VAT included of the bearings codes considering an average of the products available on one of the most famous eCommerce that allows for fast shipping. Purchasing from Asian eCommerce equivalent products allows for further savings but the shipping time increases significantly.

For the prototype build in the University laboratory we purchased branded bearings from seller having an agreement with the University. Consequently, the cost for bearings was much higher. We don't reported the actual cost because we reckon it would be misleading.

Bearing code	Unit cost	Quantity	Total
61810-2RS	7.25€/pc	6	43.50€
61805-2RS	2.14€/pc	12	25.70€
624ZZ	0.99€/pc	12	11.88€
Total			81.10€ VAT included

Table 7.3: Cost for bearings

Hardware and Fasteners

The cost of fasteners highly depends on where parts are sorted. Again, unbranded and low performance screws are enough for the application, since the plastic would break before the fasteners. On the mainstream eCommerce platforms assorted sets of screws and nuts are available. Therefore, one should select the best combination and the price varies consequently. A rough high-ball estimate is around 50€ VAT included.

Potentiometers and ADCs

The used potentiometers are of high quality and this affects the cost. They were purchased on a famous online retailer of electronics that has an agreement with the University. The unitary cost was at the time of purchasing 18.796€/pc+VAT for a total of 93.98€+VAT.

For potentiometers dedicated ADCs are required as said in Section . ADS1115 16bit 4 channel ADC was selected. Since the robot has 6 joints, 2 ADS1115 boards are required. An indicative price for one ADC is 20€, therefore 2 boards for 8 channels are around 40€ VAT included. Spare channels can be used for external sensors.

Other electronics and spares

Here is reported a non exhaustive list of other items that would be need to build the robot but that can be already available or are not necessary.

- An Arduino MEGA board.
- A power supply (24V 15A to have safety margin). The type used in 3D printers is fine.
- Relay module 24V – 5V to be commanded by Arduino to release stepper motors brakes.
- Electrical wires to extend the motor wires (phases and brakes) and for supplying

the stepper driver board. Gauge 22 wires are needed for NEMA23 stepper motors phases and gauge 20 wire for J2 brake wires (22 gauge is fine too). For NEMA17 stepper motors thinner wires are needed.

- Wires to supply the potentiometers and for signal. Since 5V supply is used Arduino cables can be used.
- Electric connectors (self-soldering and heat shrink) to extend motor wires.
- Mammoth terminal block for power distribution. Since a PCB is not used, to safely perform a parallel circuit between all stepper drivers this kind of blocks were used. They are also useful to share GND and 5V from the Arduino board without using a breadboard, since Arduino has few pins dedicated to power supply.
- Male Dupont terminal connectors to connect signal wires from potentiometers to Arduino pins.

For all this electric equipment and electronics 150€ maximum are required.

7.1.2. Limits to range of motion

Limits of joint J2

As said we experienced a failing meshing when we actuated the joint J2 with only the link L1 attached to it. We noticed that the loss of the meshing was happening when the joint was overcoming the weight force while when in favor of gravity it did not happen. This suggests that when the motor lifts the link the wave generator starts to turn freely on the flexible spline causing the link to fall instead of rising. This happens as soon as the link L1 is at a critical angle from the vertical position. We managed to have a smooth motion up to 25deg from the vertical position considering the original design of the HD (circular spline with semicircular teeth and 3D printable flexible spline).

We also performed some empirical and qualitative tests to see whether it was the stepper motor that was losing steps, whether it was the set screw on the mounting hub that was losing grip on the motor shaft or was indeed the flexible spline failing meshing. We opened one HD keeping only the flexible spline, circular spline and the wave generator and gripped the first component till slip of the meshing occurred. With the original circular spline design we noticed that the required torque to induce slipping was indeed very low, but we could not quantitatively measure it. Instead with the modified circular spline with HTD3M complementary teeth profile the torque to induce slip feels higher (again no quantitative data could be acquired). We excluded that the set screw on the mounting

hub was failing because after slipping of the flexible spline it was always resting on the flat side of the NEMA stepper. To exclude the step loss of the motor itself we set a very low acceleration for the movement and when it was required to lift the first link the issue still occurred. We want to remark that the sound the drive produced while failing was giving a clear clue on what was happening but for the sake of completeness we performed these test to exclude other causes.

We want to state clearly that the harmonic drive was not tested under load before the beginning of this thesis work. A parallel activity was run to create a test bench for the characterization of the drive, but no results were given before we faced the issue before described.

Tests on J2 harmonic drive

We tried first different combinations between circular spline and modified flexible spline to understand if meshing would be improved without increasing stress on the flexible spline with a wider wave generator. Results are reported in Table 7.4.

		Circular spline	
		Original semi-circular teeth	HTD3M complementary teeth
Flexible spline	Original S3M belt	J2 fails at 20deg with only Link 1	J2 fails at 20deg with Link 1 and Joint 3
	HTD3M belt as single piece	J2 fails at 25deg with only Link 1	J2 fails at 30deg with Link 1 and Joint 3
	HTD3M TPU -modified	J2 fails at 20deg with only Link 1	J2 fails at 30deg with Link 1 and Joint 3
	HTD3M half PETG half TPU - modified	J2 fails at 25deg with only Link 1	J2 fails at 30deg with Link 1 and Joint 3
	HTD3M PETG - modified	J2 fails at 45deg with Link 1 and Joint 3	J2 fails at 65deg with Link 1 and Joint 3
	Commercial HTD3M belt	NOT TESTED NOT RELEVANT	NOT TESTED NOT RELEVANT

Table 7.4: Results of tests on Joint 2 using original wave generator wheelbase. Comparison between different combinations of flexible spline and circular spline.

As it is possible to see, the most promising combination is with the HTD3M circular spline and PETG modified flexible spline. Moreover, it's clear that the complementary HTD3M teeth profile introduces a big improvement because the weight lifted and the maximum angle of Joint 2 are always bigger than the results obtained with the original circular spline.

Since no solution was definitive we tried with wave generators having progressively increasing wheelbase. The most interesting results were obtained with increased distance of +0.50mm, +0.80mm and +1.00mm.

Now, it is necessary to distinguish between the improvements introduced by the two best components separately.

We first fixed the PETG modified flexible spline and compared original and modified circular splines increasing the generator wheelbase. If the flexible spline made of PETG has much more importance than the modified circular spline, then original and modified circular splines would have very similar performance when increasing the wave generator wheelbase. Results are reported in Table 7.5.

		Circular spline	
		Original semi-circular teeth	HTD3M complementary teeth
Increase in wave generator wheel base	+0.50mm	J2 fails at 80deg with Link 1 + Joint 3	J2 fails at 70deg with Link 1 + Joint 3 + half Link 2 + Joint 4
	+0.80mm	J2 fails at 80deg with Link 1 + Joint 3 + half Link 2 + Joint 4	J2 achieves 90deg with Link 1 + Joint 3 + half Link 2 + Joint 4
	+1.00mm	Flexible spline broke	Flexible spline broke

Table 7.5: Results of tests on Joint 2 using the PETG modified flexible spline with HTD3M teeth. Comparison between original and modified circular splines increasing wave generator wheelbase.

We increased this distance up to 1.00mm and this value was the limit one because it caused the rupture of the flexible spline. The failure was in practices static (very few cycles of periodic shear stress were performed). The broken flexible spline showed a fracture along the layers and several splits perpendicular to the layers (Figure 7.1).



Figure 7.1: Broken flexible spline

Hence, it's not possible to affirm the cause of the failure. There are two possibilities:

1. the applied torque on the output flange was too high for the layer adhesion and so the part sheared before fracturing also perpendicularly to the layers.
2. the pressing force was too high and this caused the toothed section to split open and then the layers sheared when hit by the wave generator.

Anyway, looking the results in Table 7.5, we see that when using +0.80mm wheelbase the two circular splines perform quite similarly, thus the flexible spline has more importance.

As a check, we fixed the modified circular spline and compared all the other designs of the flexible spline available, increasing the wave generator wheelbase. Results are reported in Table 7.6. Since with the previous set of test +1.00mm cause the fracture of the 3D printed flexible spline, we tested with this increase of wheelbase only the commercial flexible spline. If the modified circular spline has more importance than the modified flexible spline, then all the typologies of flexible spline (hard plastic and soft rubber) would perform quite similarly when increasing the wave generator wheelbase.

		Increase in wave generator wheel base		
		+0.50mm	+0.80mm	+1.00mm
Flexible spline	Original S3M belt	NOT RELEVANT	NOT RELEVANT	J2 fails at 40deg with Link 1 and Joint 3
	HTD3M belt as single piece	J2 fails at 20deg with Link 1 and Joint 3	NOT RELEVANT	NOT APPLICABLE
	HTD3M TPU - modified	NOT RELEVANT	J2 fails at 45deg with Link 1 and Joint 3	NOT APPLICABLE
	HTD3M half PETG half TPU - modified	NOT RELEVANT	J2 fails at 55deg with Link 1 and Joint 3	NOT APPLICABLE
	HTD3M PETG - modified	J2 fails at 70deg with Link 1 + Joint 3 + half Link 2 + Joint 4	J2 achieves 90deg with Link 1 + Joint 3 + half Link 2 + Joint 4	Flexible spline broke
	Commercial HTD3M belt	NOT RELEVANT	J2 achieves 90deg with Link 1 + Joint 3 + half Link 2 + Joint 4	Harmonic drive completely blocks when loaded

Table 7.6: Results of tests on Joint 2 using the circular spline with HTD3M complementary teeth. Comparison between different combinations of flexible splines and increased wave generator wheelbase.

The solutions with hard plastic flexible spline and commercial HTD3M belt, using for both the circular spline with complementary HTD3M teeth profile and wave generator with +0.80mm wheelbase were further tested to establish the best flexible element for this robot. We loaded the joint J2 with the full weight of the arm (from joint J3 on, no J6 and payload) mounting the link L3 to be always vertical so that there is no lever arm. The commercial HTD3M belt can only withstand around $35 \div 40$ deg from vertical position, then the wave generator starts spinning freely on the belt. The 3D printed hard plastic flexible spline can instead reach 90deg and go back up.

Given the results above, we can state that having a flexible spline made of hard plastic is more important than having a circular spline with complementary teeth profile. Anyway, the combination of both is the only one that promise a decent range of motion of the full robot arm.

We also reckon that teeth made of TPU are too soft to withstand the compression of the wave generator with increased wheelbase. The flexible splines made of TPU fail meshing because the shape of the teeth cannot be maintained and therefore they slip on the circular spline.

Commercial HTD3M rubber belt are radially more rigid than TPU ones and this avoid slipping, but friction comes too high and the HD can block if a too high interference is introduced with the wave generator.

7.1.3. Arduino library concerns

Rotation speed of the stepper motor resulted not to be so straightforward. We consider that there may be some incompatibility between the stepper driver board from RTA and the Arduino library used. In fact, during simple tests we were not able to approach the maximum speed reported on the manufacturer datasheet. Anyway, the maximum speed set for the robot arm was achieved and accurate.

Moreover, if two commands to move a joint are sent one after the other without a sufficient pause in between, the two movements are not blended but the motor tries to brake and then suddenly it accelerates again causing noise and vibration of the arm.

The library can be modified to make it leaner and to have full control on what the code does. In this way it would take up less memory on the microcontroller. Moreover, it would be of advantage to write code *ad hoc* for the RTA stepper driver since the Arduino library only consider consumer stepper driver (A4988, DRV88xx). In fact, RTA drivers have some functionality activable by the pinout that cannot be currently used. Lastly,

it would be useful to develop a custom library to implement features like embedded rotation sensing, automatic homing procedure and guided calibration routine without need of further computation devices or many Arduino sketches. This task was out of the competencies of the writer and is not compulsory, thus is left for future work.

7.1.4. Structural concerns

During testing of the robot, as the weight of the arm increased adding more and more links, we notice a significant bending of the link L0 column on J1 joint, mainly caused by loosening screws. The design of the HD uses a big 61810 ball bearing to support bending moment. Anyhow, for safety another bearing should be added in the base column using the HD cover as shaft. Unfortunately, the HD cover has an external diameter of 78mm while existing ball bearings have internal diameter of 75mm or 80mm, thus a resizing of the HD cover is needed. Moreover, the smallest external diameter available is 100mm that is exactly the actual outer dimension of the links. Therefore, a redesign of the base column and the HD would be required to implement this structural upgrade.

7.2. Conclusions

Looking to the list of requirements in Chapter 1, several points have been satisfied.

The designed robot turned out to be competitive in terms of cost with respect the products described in Chapter 2. In particular, the price it's close and lower to the one of the Niryo One by NiryoRobotics, the closest identified competitor. The actual cost of our robot is still high because components have been sourced in small batches and high quality motors and drivers have been used. Moreover, it has a longer outreach and the payload is higher. We reckon that our product is in a primitive state and more effort is required to implement an application at this stage compared with the already implemented development environment by NiryoRobotics. In this regard, a user interface can be developed to make interaction with the robot easier. Moreover, some tools for the end effector can be developed to identify a precise task for the robot.

The robot has smooth and rounded edges and speed reducers are not exposed. Cable management has been cured as much as possible considering that the speed reducers are not hollow shaft. Cables are routed externally and pass through the arm links to go to a main harness. An enclosure for all the electronics was not addressed since the robot is at a very early prototyping phase. Anyway, a 3D printed base for electronics was designed

to tidy and separated high from low voltages.

Being fully 3D printable, speed reducers included, potentially makes this robot accessible to a very big community of Universities and Makers because the BOM has very few parts and production can be fully done locally. No proprietary hardware, electronic board or software was used control the robot.

Indeed, the 3D printed harmonic drives used as speed reducers showed some limitations regarding performance under load that have been partially mitigated during this thesis work but that are still open. Future investigations can be done to solve the loss of meshing issue to its roots, following the path traced in this work. Moreover, the pronounced backlash, increasing after some load cycles, reduces accuracy and repeatability of the robot. This problem also caused to switch from a high resolution ADC system to the Arduino built-in ADC since resolution would have been wasted by HD backlash and gearing system backlash. Reliability is compromised by the 3D printed flexible splines with hard plastic.

A low level control of the robot was achieved: the user can directly send target joints positions on Arduino MEGA on the Arduino IDE. The high level control was also addressed. The Arduino firmware for interaction with ROS is ready. The virtual model of the robot and tools for motion planning are completed. The MoveIt! GUI allows to plan simple motion and also more complex tasks can be programmed by means of coding languages. In particular, an example task of pick and place to move a box is available.

For the moment an Hardware Interface which allow the interaction between ROS control and the physical hardware is missing. Once available, it will be possible to plan a task in MoveIt! but also to make the real robot execute it.

The files to launch the multibody model in the dynamic simulation environment (Gazebo) and the packages to control it are prepared. Also the integration of Gazebo into MoveIt! is performed and allow to plan a task in MoveIt! and test it in Gazebo before making the robot execute it. In this way the user can understand if the robot is able to perform the task and if some refinements must be carried out.

A original look was given to the robot. Thanks to 3D printing, customization of the robot aesthetic is very simple to adapt the arm to tastes and blend its design to the one of other products.

The robot base can be mounted on any table top. It can be possible to mount the arm on a Automated Guided Vehicle for a mobile robot application.

The flexible spline of the harmonic drive realized with hard PETG plastic showed poor

reliability, being prone to fracture due to the high shear stresses, especially when mounted in the most stressed joint, the robot shoulder.

The modular design proven to be effective in easing disassembling and assembling for maintenance and repair.

The modular design proven to be effective in easing disassembling and assembling for maintenance and repair.

Bibliography

- [1] ABB. Abb robotstudio. URL <https://new.abb.com/products/robotics/robotstudio>.
- [2] I. Askin. Design of a 3d-printed harmonic drive for low-cost modular robot. Master's thesis, Politecnico di Milano Scuola di Ingegneria Industriale e dell'Informazione, 2018/2019.
- [3] S. Blankemeyer, R. Wiemann, L. Posniak, C. Pregizer, and A. Raatz. Intuitive robot programming using augmented reality. *Procedia CIRP*, 76:155–160, 2018. ISSN 2212-8271. doi: <https://doi.org/10.1016/j.procir.2018.02.028>. URL <https://www.sciencedirect.com/science/article/pii/S2212827118300933>. 7th CIRP Conference on Assembly Technologies and Systems (CATS 2018).
- [4] S. R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17(1-19):16, 2004.
- [5] Y.-C. Chen, Y.-H. Cheng, J.-T. Tseng, and K.-J. Hsieh. Study of a harmonic drive with involute profile flexspline by two-dimensional finite element analysis. *Engineering Computations*, 34(7):2107–2130, oct 2017. doi: 10.1108/ec-03-2017-0086. URL <https://doi.org/10.1108/EC-03-2017-0086>.
- [6] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo. ros_control: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2017. doi: 10.21105/joss.00456. URL <http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf>.
- [7] S. Choi, W. Eakins, G. Rossano, and T. Fuhlbrigge. Lead-through robot teaching. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–4, 2013. doi: 10.1109/TePRA.2013.6556347.
- [8] Gazebo. Gazebo. URL <http://gazebo.org/tutorials>.

- [9] K. Khokar, P. Beeson, and R. Burrige. Implementation of kdl inverse kinematics routine on the atlas humanoid robot. *Procedia Computer Science*, 46: 1441–1448, 2015. ISSN 1877-0509. doi: <https://doi.org/10.1016/j.procs.2015.02.063>. URL <https://www.sciencedirect.com/science/article/pii/S1877050915001271>. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.
- [10] T. Kose. Moveit gazebo. URL <https://medium.com/@tahsincankose/custom-manipulator-simulation-in-gazebo-and-motion-planning-with-moveit>
- [11] S. Mitsi, K.-D. Bouzakis, G. Mansour, D. Sagris, and G. Maliaris. Off-line programming of an industrial robot for manufacturing. *The International Journal of Advanced Manufacturing Technology*, 26(3):262–267, Aug 2005. ISSN 1433-3015. doi: 10.1007/s00170-003-1728-5. URL <https://doi.org/10.1007/s00170-003-1728-5>.
- [12] MoveIt! Moveit! URL https://ros-planning.github.io/moveit_tutorials/.
- [13] J. Norberto Pires, T. Godinho, and P. Ferreira. Cad interface for automatic robot welding programming. *Industrial Robot: An International Journal*, 31(1):71–76, Jan 2004. ISSN 0143-991X. doi: 10.1108/01439910410512028. URL <https://doi.org/10.1108/01439910410512028>.
- [14] OnRobot. 2fg7 onrobot gripper. URL <https://onrobot.com/it/prodotti/2fg7>.
- [15] Z. Pan, J. Polden, N. Larkin, S. Van Duin, and J. Norrish. Recent progress on programming methods for industrial robots. *Robotics and Computer-Integrated Manufacturing*, 28(2):87–94, 2012. ISSN 0736-5845. doi: <https://doi.org/10.1016/j.rcim.2011.08.004>. URL <https://www.sciencedirect.com/science/article/pii/S0736584511001001>.
- [16] ROS. Ros wiki. URL <http://wiki.ros.org/it>.
- [17] ROS.org. roserial package summary. <http://wiki.ros.org/roserial>. URL <http://wiki.ros.org/roserial>.
- [18] M. Rotulo. Ridimensionamento di un riduttore armonico per un robot stampato in 3d. 2020.
- [19] H. D. SE. Riduttore armonico harmonic drive®.

- URL <https://harmonicdrive.de/it/glossario/riduttore-armonico-harmonic-drive>.
- [20] UniversalRobots. Carta delle idee della robotica collaborativa 2021. robotica collaborativa, quali sfide per l'education, Sept. 2021.
- [21] Wikipedia contributors. Strain wave gearing — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Strain_wave_gearing&oldid=1031130185, 2021. [Online; accessed 18-November-2021].
- [22] Wikipedia contributors. Acrylonitrile butadiene styrene — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Acrylonitrile_butadiene_styrene&oldid=1052546143, 2021. [Online; accessed 8-November-2021].
- [23] Wikipedia contributors. Polyethylene terephthalate — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Polyethylene_terephthalate&oldid=1049252865, 2021. [Online; accessed 8-November-2021].
- [24] Wikipedia contributors. Robot operating system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Robot_Operating_System&oldid=1051240875, 2021. [Online; accessed 8-November-2021].
- [25] S. Wu. Euclid design and optimization of a 3d printed robotic arm. Master's thesis, Politecnico di Milano School of Design, 2019.

A | Codes

A.1. Rotation assessment

```
1 #include "BasicStepperDriver.h"
2 // Setting PINs
3 const int pinStep_HD = 2;
4 const int pinDir_HD = 3;
5 // Setting motor and board parameters
6 const int MOTOR_STEP = 200;
7 const int MICROSTEPS = 2;
8 // Setting constants
9 const int tau = 35; // reduction ratio
10 const int step_per_rev = MICROSTEPS*MOTOR_STEP; // Number of
    steps per revolution of the motor [step/rev]
11 // Set type of motion law
12 // #define mode BasicStepperDriver::CONSTANT_SPEED
13 #define mode BasicStepperDriver::LINEAR_SPEED
14 // Define stepper driver object
15 BasicStepperDriver HD(step_per_rev, pinDir_HD, pinStep_HD);
16 // Global variables declaration
17 bool START = 0; // start trigger
18 bool CONTINUE = 0; // continue trigger
19 int i = 0; // counter
20 int incomingByte = 0; // for incoming serial data
21 // Test parameters
22 double angle[] = {1, 5, 10, 25, 45, 60, 75, 90, 180}; // [deg]
23 double velocity = 2; // [rpm]
24
25 void setup() {
26     // Initialize stepper driver object
```

```
27 HD.begin();
28 HD.setSpeedProfile(mode, 875, 875);
29 HD.setRPM(velocity*tau);
30 // Initialize serial communication
31 Serial.begin(9600);
32 // Wait untill start trigger is given
33 while ( !START ) {
34     if ( Serial.available() ) {
35         incomingByte = Serial.read();
36         // 's' = start test
37         if (incomingByte == 's'){
38             START = 1;
39         }
40     }
41 }
42 }
43
44 void loop() {
45     // Perform current test -> i counter
46     Serial.println(angle[i]);
47     HD.rotate(angle[i]*tau);
48     delay(100);
49     // Wait untill continue trigger is given
50     while ( !CONTINUE ) {
51         if ( Serial.available() ) {
52             incomingByte = Serial.read();
53             // 'c' = continue with next angle
54             if (incomingByte == 'c'){
55                 HD.rotate(-angle[i]*tau);
56                 delay(100);
57                 i++;
58                 CONTINUE = 1;
59             }
60         }
61     }
62     delay(10);
63     // Disable continue trigger
```

```
64  CONTINUE = 0;
65 }
```

Listings A.1: Rotation assessment

A.2. Rotation speed assessment

```
1  #include "BasicStepperDriver.h"
2  // Setting PINs
3  const int pinStep_HD = 2;
4  const int pinDir_HD = 3;
5  // Setting motor and board parameters
6  const int MOTOR_STEP = 200;
7  const int MICROSTEPS = 2;
8  // Setting constants
9  const int tau = 35;
10 const int step_per_rev = MICROSTEPS*MOTOR_STEP; // Number of
    steps per revolution of the motor [step/rev]
11 // Set type of motion law
12 #define mode BasicStepperDriver::CONSTANT_SPEED
13 // Define stepper driver object
14 BasicStepperDriver HD(step_per_rev, pinDir_HD, pinStep_HD);
15
16 void setup() {
17     // Initialize stepper driver object
18     HD.begin();
19     HD.setSpeedProfile(mode, 875, 875);
20 }
21
22 void loop() {
23     double angle = 360; //[deg]
24     double velocity = 10; // [rpm] {1 2 4 5 8 9 10}
25     HD.setRPM(velocity*tau);
26     HD.rotate(angle*tau);
27     delay(4000);
28 }
```

Listings A.2: Speed assessment

A.3. Mock Arduino PID controller

```
1  #include "BasicStepperDriver.h"
2  #include "SyncDriver.h"
3  // Setting PINs ...
4  // ... for Stepper Drivers
5  const int pinStep_J1 = 2;
6  const int pinDir_J1 = 3;
7  const int pinStep_J2 = 4;
8  const int pinDir_J2 = 5;
9  const int pinStep_J3 = 6;
10 const int pinDir_J3 = 7;
11 const int pinStep_J4 = 8;
12 const int pinDir_J4 = 9;
13 const int pinStep_J5 = 10;
14 const int pinDir_J5 = 11;
15 const int pinStep_J6 = 12;
16 const int pinDir_J6 = 13;
17 // ... for Potentiometers
18 const int potPin_J1 = A0;
19 const int potPin_J2 = A1;
20 const int potPin_J3 = A2;
21 const int potPin_J4 = A3;
22 const int potPin_J5 = A4;
23 //const int potPin_J6 = A5; // NO potentiometer was bought
for J6
24 // ... for Brakes
25 const int pinBrake_J2 = 22;
26 const int pinBrake_J3 = 23;
27 // Setting motor and board parameters
28 const int MOTOR_STEP = 200;
29 const int MICROSTEPS = 2;
30 const int step_per_rev = MICROSTEPS*MOTOR_STEP; // Number
of steps per revolution of the motor [step/rev]
31 // Setting constants
32 const int tau = 35; // HD reduction ratio
33 const float tau_gear_J1 = 83.0/17;
```

```
34     const float tau_gear_J2 = 75.0/17;
35     const float tau_gear_J3 = 75.0/17;
36     const float tau_gear_J4 = 83.0/17;
37     const float tau_gear_J5 = 75.0/17;
38     //const float tau_gear_J6 = 83.0/17;
39     // Setting ADC parameters
40     const float resolution = 5/pow(2,10); //[V/bit]
41     const float sensitivity = 360; //[deg/V]
42     /*
43      * ATTENTION: EVERY TIME ONE LINK IS DISASSEMBLED YOU NEED
44      * CALIBRATE AGAIN ITS POTENTIOMETER AND UPDATE VALUES
45      * HEREAFTER
46      */
47     const float home_volt_J1 = 2.52; //[V]
48     const float home_offset_J1 = -0.04; //[deg]
49     const float home_volt_J2 = 2.50; //[V]
50     const float home_offset_J2 = 0; //[deg]
51     const float home_volt_J3 = 2.53; //[V]
52     const float home_offset_J3 = -0.06;
53     const float home_volt_J4 = 2.52;
54     const float home_offset_J4 = -0.04;
55     const float home_volt_J5 = 2.47;
56     const float home_offset_J5 = 0.04;
57     //const float home_volt_J6 =
58     //const float home_offset_J6 =
59     // Set type of motion law
60     #define mode BasicStepperDriver::LINEAR_SPEED
61     // Define stepper driver objects
62     BasicStepperDriver J1(step_per_rev, pinDir_J1, pinStep_J1);
63     BasicStepperDriver J2(step_per_rev, pinDir_J2, pinStep_J2);
64     BasicStepperDriver J3(step_per_rev, pinDir_J3, pinStep_J3);
65     BasicStepperDriver J4(step_per_rev, pinDir_J4, pinStep_J4);
66     BasicStepperDriver J5(step_per_rev, pinDir_J5, pinStep_J5);
67     BasicStepperDriver J6(step_per_rev, pinDir_J6, pinStep_J6);
68     // Define object to synchronize all stepper motors
69     SyncDriver arm(J1, J2, J3, J4, J5, J6);
70     // Global variables declaration
```

```
70     float toll = 0.9; //[deg] tollerance on rotation (half
motor step)
71     float current_position_J1, current_position_J2,
current_position_J3, current_position_J4,
current_position_J5;
72     float desired_joint_position[5];
73     float desired_position_J1, desired_position_J2,
desired_position_J3;
74     float desired_position_J4, desired_position_J5; //
desired_position_J6;
75     float error_J1=0, error_J2=0, error_J3=0, error_J4=0,
error_J5=0;
76     float error_J1_old=0, error_J2_old=0, error_J3_old=0,
error_J4_old=0, error_J5_old=0;
77     uint8_t count = 0;
78     float d_dt_error_J1=0, d_dt_error_J2=0, d_dt_error_J3=0,
d_dt_error_J4=0, d_dt_error_J5=0;
79     //float int_error_J1_dt[256], int_error_J2_dt[256],
int_error_J3_dt[256], int_error_J4_dt[256], int_error_J5_dt
[256];
80     float int_error_J1_dt=0, int_error_J2_dt=0, int_error_J3_dt
=0, int_error_J4_dt=0, int_error_J5_dt=0;
81     float int_error_J1_dt_old=0, int_error_J2_dt_old=0,
int_error_J3_dt_old=0, int_error_J4_dt_old=0,
int_error_J5_dt_old=0;
82     float time_old_J1, time_new_J1, time_old_J2, time_new_J2,
time_old_J3, time_new_J3;
83     float time_old_J4, time_new_J4, time_old_J5, time_new_J5;
84     float Kp = 0.95; // Proportional gain
85     float Kd = 0.1; // Derivative gain
86     float Ki = 0; // Integral gain
87     // Function to compute area of trapezoid
88     float trapz(float t2, float t1, float e2, float e1){
89         float integral = 0.5*(e2+e1)*(t2-t1);
90         return integral;
91     }
92
```

```
93     void setup() {
94         // Initialize stepper driver object
95         J1.begin();
96         J1.setSpeedProfile(mode, 875, 875);
97         J2.begin();
98         J2.setSpeedProfile(mode, 875, 875);
99         J3.begin();
100        J3.setSpeedProfile(mode, 875, 875);
101        J4.begin();
102        J4.setSpeedProfile(mode, 875, 875);
103        J5.begin();
104        J5.setSpeedProfile(mode, 875, 875);
105        /*
106        * J6.begin();
107        * J6.setSpeedProfile(mode, 875, 875);
108        */
109        pinMode(pinBrake_J2, OUTPUT);
110        // Engage the brake = switch off relay
111        digitalWrite(pinBrake_J2, HIGH);
112        pinMode(pinBrake_J3, OUTPUT);
113        // Engage the brake = switch off relay
114        digitalWrite(pinBrake_J3, HIGH);
115        // Initialize serial communication
116        Serial.begin(9600);
117        Serial.println("To control the positions of the robot
joints, please type the desired positions (also floats) as:
");
118        Serial.println("number(space)number(space)... from J1 to
J5 and finally (return)");
119    }
120
121    void loop() {
122        // Wait to receive a set of 5 floats
123        if( Serial.available() ){
124            for(byte i = 0; i < 5; i++){
125                desired_joint_position[i] = Serial.parseFloat();
126            }
```

```
127     // Assign the parsed floats to the variables
128     desired_position_J1 = desired_joint_position[0];
129     desired_position_J2 = desired_joint_position[1];
130     desired_position_J3 = desired_joint_position[2];
131     desired_position_J4 = desired_joint_position[3];
132     desired_position_J5 = desired_joint_position[4];
133     /* DEBUG
134     Serial.print("You typed: ");
135     for(byte i = 0; i < 5; i++){
136         Serial.print(desired_joint_position[i]); Serial.print
137         (" ");
138         Serial.println("");
139     */
140     float current_reading_J1 = analogRead(potPin_J1); // in
141     range [0 1023]
142     // Saving the time in which the measure is done as
143     early as possible
144     // Needed to compute after the derivative term
145     time_old_J1 = millis()/1000; //[s]
146     float volt_J1 = current_reading_J1 * resolution -
147     home_volt_J1;
148     current_position_J1 = volt_J1 * sensitivity /
149     tau_gear_J1 - home_offset_J1; //[deg]
150     error_J1 = desired_position_J1 - current_position_J1;
151     // Do the same for all other joints
152     // J2
153     float current_reading_J2 = analogRead(potPin_J2); // in
154     range [0 1023]
155     time_old_J2 = millis()/1000;
156     float volt_J2 = current_reading_J2 * resolution -
157     home_volt_J2;
158     current_position_J2 = volt_J2 * sensitivity /
159     tau_gear_J2 - home_offset_J2; //[deg]
160     error_J2 = desired_position_J2 - current_position_J2;
161     // J3
162     float current_reading_J3 = analogRead(potPin_J3); // in
```



```
    range [0 1023]
156     time_old_J3 = millis()/1000;
157     float volt_J3 = current_reading_J3 * resolution -
home_volt_J3;
158     current_position_J3 = volt_J3 * sensitivity /
tau_gear_J3 - home_offset_J3; //[deg]
159     error_J3 = desired_position_J3 - current_position_J3;
160     // J4
161     float current_reading_J4 = analogRead(potPin_J4); // in
    range [0 1023]
162     time_old_J4 = millis()/1000;
163     float volt_J4 = current_reading_J4 * resolution -
home_volt_J4;
164     current_position_J4 = volt_J4 * sensitivity /
tau_gear_J4 - home_offset_J4; //[deg]
165     error_J4 = desired_position_J4 - current_position_J4;
166     // J5
167     float current_reading_J5 = analogRead(potPin_J5); // in
    range [0 1023]
168     time_old_J5 = millis()/1000;
169     float volt_J5 = current_reading_J5 * resolution -
home_volt_J5;
170     current_position_J5 = volt_J5 * sensitivity /
tau_gear_J5 - home_offset_J5; //[deg]
171     error_J5 = desired_position_J5 - current_position_J5;
172     while(abs(error_J1) > toll || abs(error_J2) > toll ||
abs(error_J3) > toll || abs(error_J4) > toll || abs(error_J5
) > toll){
173         // PID-controller
174         float action_J1 = Kp*error_J1 + Kd*d_dt_error_J1 + Ki
*int_error_J1_dt; //[deg] but for the link
175         float action_J2 = Kp*error_J2 + Kd*d_dt_error_J2 + Ki
*int_error_J2_dt; //[deg] but for the link
176         float action_J3 = Kp*error_J3 + Kd*d_dt_error_J3 + Ki
*int_error_J3_dt; //[deg] but for the link
177         float action_J4 = Kp*error_J4 + Kd*d_dt_error_J4 + Ki
*int_error_J4_dt;
```

```
178     float action_J5 = Kp*error_J5 + Kd*d_dt_error_J5 + Ki
*int_error_J5_dt;
179     float action_J6 = 0;
180     // Release brakes
181     digitalWrite(pinBrake_J2, LOW);
182     digitalWrite(pinBrake_J3, LOW);
183     delay(10);
184     // The motor must rotate of action_Joint*tau
185     arm.rotate(tau*action_J1, tau*action_J2, tau*
action_J3, tau*action_J4, tau*action_J5, tau*action_J6);
186     delay(10);
187     // Engage brakes
188     digitalWrite(pinBrake_J2, HIGH);
189     digitalWrite(pinBrake_J3, HIGH);
190     // Acquire new position
191     //J1
192     current_reading_J1 = analogRead(potPin_J1);
193     time_new_J1 = millis()/1000;
194     volt_J1 = current_reading_J1 * resolution -
home_volt_J1;
195     current_position_J1 = volt_J1* sensitivity /
tau_gear_J1 - home_offset_J1; //[deg]
196     error_J1 = desired_position_J1 - current_position_J1;
197     d_dt_error_J1 = (error_J1 - error_J1_old)/(
time_new_J1 - time_old_J1);
198     //int_error_J1_dt[count++] = int_error_J1_dt[count] +
trapz(time_new_J1, time_old_J1, error_J1, error_J1_old);
199     int_error_J1_dt = int_error_J1_dt_old + trapz(
time_new_J1, time_old_J1, error_J1, error_J1_old);
200     error_J1_old = error_J1;
201     time_old_J1 = time_new_J1;
202     int_error_J1_dt_old = int_error_J1_dt;
203     //J2
204     current_reading_J2 = analogRead(potPin_J2);
205     time_new_J2 = millis()/1000;
206     volt_J2 = current_reading_J2 * resolution -
home_volt_J2;
```

```
207     current_position_J2 = volt_J2* sensitivity /
tau_gear_J2 - home_offset_J2; //[deg]
208     error_J2 = desired_position_J2 - current_position_J2;
209     d_dt_error_J2 = (error_J2 - error_J2_old)/(
time_new_J2 - time_old_J2);
210     //int_error_J2_dt[count++] = int_error_J2_dt[count] +
trapz(time_new_J2, time_old_J2, error_J2, error_J2_old);
211     int_error_J2_dt = int_error_J2_dt_old + trapz(
time_new_J2, time_old_J2, error_J2, error_J2_old);
212     error_J2_old = error_J2;
213     time_old_J2 = time_new_J2;
214     int_error_J2_dt_old = int_error_J2_dt;
215     // J3
216     current_reading_J3 = analogRead(potPin_J3);
217     time_new_J3 = millis()/1000;
218     volt_J3 = current_reading_J3 * resolution -
home_volt_J3;
219     current_position_J3 = volt_J3* sensitivity /
tau_gear_J3 - home_offset_J3; //[deg]
220     error_J3 = desired_position_J3 - current_position_J3;
221     d_dt_error_J3 = (error_J3 - error_J3_old)/(
time_new_J3 - time_old_J3);
222     //int_error_J3_dt[count++] = int_error_J3_dt[count] +
trapz(time_new_J3, time_old_J3, error_J3, error_J3_old);
223     int_error_J3_dt = int_error_J3_dt_old + trapz(
time_new_J3, time_old_J3, error_J3, error_J3_old);
224     error_J3_old = error_J3;
225     time_old_J3 = time_new_J3;
226     int_error_J3_dt_old = int_error_J3_dt;
227     // J4
228     current_reading_J4 = analogRead(potPin_J4);
229     time_new_J4 = millis()/1000;
230     volt_J4 = current_reading_J4 * resolution -
home_volt_J4;
231     current_position_J4 = volt_J4* sensitivity /
tau_gear_J4 - home_offset_J4; //[deg]
232     error_J4 = desired_position_J4 - current_position_J4;
```

```

233     d_dt_error_J4 = (error_J4 - error_J4_old)/(
time_new_J4 - time_old_J4);
234     //int_error_J4_dt[count++] = int_error_J4_dt[count] +
trapz(time_new_J4, time_old_J4, error_J4, error_J4_old);
235     int_error_J4_dt = int_error_J4_dt_old + trapz(
time_new_J4, time_old_J4, error_J4, error_J4_old);
236     error_J4_old = error_J4;
237     time_old_J4 = time_new_J4;
238     int_error_J4_dt_old = int_error_J4_dt;
239     // J5
240     current_reading_J5 = analogRead(potPin_J5);
241     time_new_J5 = millis()/1000;
242     volt_J5 = current_reading_J5 * resolution -
home_volt_J5;
243     current_position_J5 = volt_J5* sensitivity /
tau_gear_J5 - home_offset_J5; //[deg]
244     error_J5 = desired_position_J5 - current_position_J5;
245     d_dt_error_J5 = (error_J5 - error_J5_old)/(
time_new_J5 - time_old_J5);
246     //int_error_J5_dt[count++] = int_error_J5_dt[count] +
trapz(time_new_J5, time_old_J5, error_J5, error_J5_old);
247     int_error_J5_dt = int_error_J5_dt_old + trapz(
time_new_J5, time_old_J5, error_J5, error_J5_old);
248     error_J5_old = error_J5;
249     time_old_J5 = time_new_J5;
250     int_error_J5_dt_old = int_error_J5_dt;
251 }
252 // Request new desired postions through Serial Monitor
253 Serial.println("Type the desired positions: ");
254 }
255 }

```

Listings A.3: Mock Arduino PID controller

A.4. Arduino Firmware for ROS

```

1 #include "BasicStepperDriver.h"

```

```
2   #include "SyncDriver.h"
3   #include <ros.h>
4   #include <std_msgs/Float32MultiArray.h>
5   // Setting PINs of motors
6   const int pinStep_J1 = 2;
7   const int pinDir_J1 = 3;
8   const int pinStep_J2 = 4;
9   const int pinDir_J2 = 5;
10  const int pinStep_J3 = 6;
11  const int pinDir_J3 = 7;
12  const int pinStep_J4 = 8;
13  const int pinDir_J4 = 9;
14  const int pinStep_J5 = 10;
15  const int pinDir_J5 = 11;
16  const int pinStep_J6 = 12;
17  const int pinDir_J6 = 13;
18  // ... for Potentiometers
19  const int potPin_J1 = A0;
20  const int potPin_J2 = A1;
21  const int potPin_J3 = A2;
22  const int potPin_J4 = A3;
23  const int potPin_J5 = A4;
24  // Brakes
25  const int pinBrake_J2 = 22;
26  const int pinBrake_J3 = 23;
27  // Setting motor and board parameters
28  const int MOTOR_STEP = 200;
29  const int MICROSTEPS = 2;
30  const int step_per_rev = MICROSTEPS*MOTOR_STEP; // Number
of steps per revolution of the motor [step/rev]
31  // Setting constants:
32  const int tau = 35; // HD reduction ratio
33  // Gear ratio for Potentiometers
34  const float tau_gear_J1 = 83.0 / 17;
35  const float tau_gear_J2 = 75.0 / 17;
36  const float tau_gear_J3 = 75.0 / 17;
37  const float tau_gear_J4 = 83.0 / 17;
```

```

38     const float tau_gear_J5 = 75.0 / 17;
39     // Setting ADC parameters
40     const float resolution = 5 / pow(2, 10); // [V/bit]
41     const float sensitivity = 360; // [deg/V]
42     const float home_volt_J1 = 2.52; // [V]
43     const float home_offset_J1 = -0.04; // [deg]
44     const float home_volt_J2 = 2.50; // [V]
45     const float home_offset_J2 = 0; // [deg]
46     const float home_volt_J3 = 2.53; // [V]
47     const float home_offset_J3 = -0.06;
48     const float home_volt_J4 = 2.52;
49     const float home_offset_J4 = -0.04;
50     const float home_volt_J5 = 2.47;
51     const float home_offset_J5 = -0.38;
52     // Definition of HOME POSITION
53     const float home_position_J1 = 90.0; // CCW
54     const float home_position_J2 = 45.0; // CCW
55     const float home_position_J3 = 90.0; // CW => keep positive
=> positive angle = CW wrt J2
56     const float home_position_J4 = 0.0; // KEEP ALIGNED
57     const float home_position_J5 = -90.0; // CW
58     const float home_array[5] = {home_position_J1,
home_position_J2, home_position_J3, home_position_J4,
home_position_J5};
59     // Service variables
60     const float tau_gear[] = {tau_gear_J1, tau_gear_J2,
tau_gear_J3, tau_gear_J4, tau_gear_J5};
61     const float home_volt[] = {home_volt_J1, home_volt_J2,
home_volt_J3, home_volt_J4, home_volt_J5};
62     const float home_offset[] = {home_offset_J1, home_offset_J2
, home_offset_J3, home_offset_J4, home_offset_J5};
63     float old_position[5] = {home_position_J1, home_position_J2
, home_position_J3, home_position_J4, home_position_J5};
64     float positions_array[5];
65     // Set type of motion law
66     #define mode BasicStepperDriver::LINEAR_SPEED
67     // Define stepper driver object

```

```
68   BasicStepperDriver J1(step_per_rev, pinDir_J1, pinStep_J1);
69   BasicStepperDriver J2(step_per_rev, pinDir_J2, pinStep_J2);
70   BasicStepperDriver J3(step_per_rev, pinDir_J3, pinStep_J3);
71   BasicStepperDriver J4(step_per_rev, pinDir_J4, pinStep_J4);
72   BasicStepperDriver J5(step_per_rev, pinDir_J5, pinStep_J5);
73   BasicStepperDriver J6(step_per_rev, pinDir_J6, pinStep_J6);
74   // Define object to synchronize all stepper motors
75   SyncDriver arm(J1, J2, J3, J4, J5, J6);
76   // Create ROS nodehandler
77   ros::NodeHandle nh;
78   // Callback function to make the arm move when a message is
   received
79   void move_arm_cb(const std_msgs::Float32MultiArray& cmd_msg
   ){
80       // Release the brakes on Joints J2 and J3 before start
   moving
81       digitalWrite(pinBrake_J2, LOW);
82       digitalWrite(pinBrake_J3, LOW);
83       delay(10); // Wait a short time to be sure the breaks are
   released
84       // Note that if the rotation of the link is requested,
   motors must rotate of tau*angle_for_the_link
85       float new_position[5] = {cmd_msg.data[0], cmd_msg.data
   [1], cmd_msg.data[2], cmd_msg.data[3], cmd_msg.data[4]};
86       float new_command[5]; // Command for the motors
87       for (int i = 0; i < 5; ++i) {
88           new_command[i] = tau*(new_position[i] - old_position[i
   ]);
89       }
90       arm.rotate(new_command[0], new_command[1], new_command
   [2], new_command[3], new_command[4], new_command[5]);
91       // Store last requested positions coming from cmd_msg.
   data in old_position
92       for(int i = 0; i < 5; ++i){
93           old_position[i] = cmd_msg.data[i];
94       }
95       delay(10);
```

```

96     // Engage the brakes when finished moving
97     digitalWrite(pinBrake_J2, HIGH);
98     digitalWrite(pinBrake_J3, HIGH);
99     }
100    // Subscriber for position
101    ros::Subscriber<std_msgs::Float32MultiArray> sub_pos("/
pvbot/actuate", move_arm_cb);
102    // Callback function to set joint speed
103    void set_speed_cb( const std_msgs::Float32MultiArray&
speed_msg) {
104        // speed_msg.data is the speed for the joint so speed for
the motor is tau*speed_msg.data
105        J1.setRPM(tau * speed_msg.data[0]);
106        J2.setRPM(tau * speed_msg.data[1]);
107        J3.setRPM(tau * speed_msg.data[2]);
108        J4.setRPM(tau * speed_msg.data[3]);
109        J5.setRPM(tau * speed_msg.data[4]);
110    }
111    // Subscriber for setting speed
112    ros::Subscriber<std_msgs::Float32MultiArray> sub_speed("/
pvbot/set_speed", set_speed_cb);
113    // Setting Publisher for position feedback
114    std_msgs::Float32MultiArray feedback_msg;
115    ros::Publisher pub_state("/pvbot/state", &feedback_msg);
116    // feedback_msg.data will be an array of 5 element with the
current position of the links
117    // Define function for reading 5 potentiometers
118    void feedback_position(){
119        for(int i = 0; i<5; i++){
120            float current_reading = analogRead(i); // in range [0
1023]
121            float volt = current_reading * resolution - home_volt[i
];
122            float current_position = volt * sensitivity / tau_gear[
i] - home_offset[i]; //[deg]
123            positions_array[i] = current_position;
124        }

```



```
125     }
126     // Define function for homing the robot (home_position_
values are set for all axes)
127     // Conceptually similar to the actuation call back function
128     void home_axes(){
129         // Read current positions and save them in
positions_array
130         feedback_position();
131         float command_angle[5];
132         // Filling the command array
133         for(int i = 0; i < 5; ++i){
134             command_angle[i] = tau * (home_array[i] -
positions_array[i]);
135         }
136         // Release the brakes on Joints J2 and J3 before start
moving
137         digitalWrite(pinBrake_J2, LOW);
138         digitalWrite(pinBrake_J3, LOW);
139         delay(10); // Wait a short time to be sure the breaks are
released
140         // Actuate to home
141         arm.rotate(command_angle[0], command_angle[1],
command_angle[2], command_angle[3], command_angle[4],
command_angle[5]);
142         // Engage the brakes when finished moving
143         delay(10);
144         digitalWrite(pinBrake_J2, HIGH);
145         digitalWrite(pinBrake_J3, HIGH);
146     }
147
148     void setup() {
149         // Initialize stepper driver object
150         J1.begin();
151         J1.setSpeedProfile(mode, 875, 875);
152         J2.begin();
153         J2.setSpeedProfile(mode, 875, 875);
154         J3.begin();
```

```
155     J3.setSpeedProfile(mode, 875, 875);
156     J4.begin();
157     J4.setSpeedProfile(mode, 875, 875);
158     J5.begin();
159     J5.setSpeedProfile(mode, 875, 875);
160     // Setup brakes of joint J2 and J3
161     pinMode(pinBrake_J2, OUTPUT);
162     // Engage the brake = switch off relay -> set relay
command pin to HIGH
163     digitalWrite(pinBrake_J2, HIGH);
164     pinMode(pinBrake_J3, OUTPUT);
165     // Engage the brake = switch off relay -> set relay
command pin to HIGH
166     digitalWrite(pinBrake_J3, HIGH);
167     // Go home
168     home_axes();
169     // Initialize ROS node, subscribe and advertise topics
170     nh.initNode();
171     nh.subscribe(sub_pos);
172     nh.subscribe(sub_speed);
173     // Dimension of the message must be declared
174     feedback_msg.data_length = 5;
175     nh.advertise(pub_state);
176 }
177
178 void loop() {
179     feedback_position();
180     feedback_msg.data = positions_array;
181     pub_state.publish(&feedback_msg);
182     nh.spinOnce();
183     delay(200);
184 }
```

Listings A.4: Arduino Firmware for ROS

A.5. Codes for the digital model of the robot with gripper

Below, there is the xacro model of the robot which is called lucabot. In this file is included also the model of the OnRobot gripper which is reported here A.6 on page 195.

```

1 <?xml version="1.0"?>
2 <robot name="lucabot" xmlns:xacro="http://www.ros.org/wiki/xacro">
3
4   <xacro:property name="i" value="0.3" />
5   <xacro:property name="i1" value="0.3477285474" />
6   <xacro:property name="i2" value="0.3072289157" />
7   <xacro:property name="i3" value="0.212585034" />
8
9   <link name="world"/>
10
11  <link name="base">
12    <visual>
13      <origin rpy="0 0 0" xyz="0 0 0"/>
14      <geometry>
15        <mesh filename="package://lucabot_description/model/base.dae"/>
16      </geometry>
17    </visual>
18    <collision>
19      <origin rpy="0 0 0" xyz="0 0 0.0415"/>
20      <geometry>
21        <cylinder length="0.083" radius="0.100"/>
22      </geometry>
23    </collision>
24    <inertial>
25      <origin xyz="-0.000313 0 0.029159" rpy="0 0 0"/>
26      <mass value="{i*0.848}" />
27      <inertia ixx="{i*0.001992393}" ixy="{i*0.0}" ixz="{i*0.000007016}"
28              iyy="{i*0.002050412}" iyz="{i*0.0}"
29              izz="{i*0.003102151}" />
30    </inertial>
31  </link>
32
33  <joint name="fixed" type="fixed">
34    <origin xyz="0 0 0" rpy="0 0 0" />
35    <parent link="world" />
36    <child link="base" />
37  </joint>
38

```

```

39 <link name="link0">
40   <visual>
41     <origin rpy="0 0 0" xyz="0 0 0"/>
42     <geometry>
43       <mesh filename="package://lucabot_description/model/
link0_denti_incassati.dae"/>
44     </geometry>
45   </visual>
46   <collision>
47     <origin rpy="0 0 0" xyz="0 0 0.1255"/>
48     <geometry>
49       <box size="0.206 0.120 0.229"/>
50     </geometry>
51   </collision>
52   <inertial>
53     <origin xyz="-0.030697 0.000014 0.136133" rpy="0 0 0"/>
54     <mass value="{i*1.385}" />
55     <inertia ixx="{i*0.006060034}" ixy="{i*0.000000665}" ixz="{i
*0.001682159}"
56               iyy="{i*0.006600492}" iyz="{i*-0.000001679}"
57               izz="{i*0.003229145}" />
58   </inertial>
59 </link>
60
61   <joint name="r1" type="revolute">
62     <origin xyz="0 0 0.07" rpy="0 0 0" />
63     <parent link="base" />
64     <child link="link0" />
65     <limit effort="40" lower="-3.142" upper="3.142" velocity="1" />
66     <axis xyz="0 0 1" />
67   </joint>
68
69   <link name="link1">
70     <visual>
71       <origin rpy="{pi/2} {pi} 0" xyz="0 0 0"/>
72       <geometry>
73         <mesh filename="package://lucabot_description/model/link1_v2_intero
.dae"/>
74       </geometry>
75     </visual>
76     <collision>
77       <origin rpy="0 0 0" xyz="0 0 0.1"/>
78       <geometry>
79         <box size="0.190 0.120 0.320"/>

```

```

80     </geometry>
81 </collision>
82 <inertial>
83   <origin xyz="0.028857 -0.000241 0.12016" rpy="0 0 0"/>
84   <mass value="{i1*1.783}" />
85   <inertia ixx="{i1*0.014019562}" ixy="{i1*-0.000003339}" ixz="{i1
86     *0.001190259}"
87     iyy="{i1*0.013996178}" iyz="{i1*0.000032161}"
88     izz="{i1*0.003295798}" />
89 </inertial>
90 </link>
91
92   <joint name="r2" type="revolute">
93     <origin xyz="0 0 0.180" rpy="0 0 0" />
94     <parent link="link0" />
95     <child link="link1" />
96     <limit effort="40" lower="-3.142" upper="3.142" velocity="1" />
97     <axis xyz="1 0 0" />
98 </joint>
99
100 <link name="link2_down">
101   <visual>
102     <origin rpy="{pi/2} {pi} 0" xyz="0 0 0"/>
103     <geometry>
104       <mesh filename="package://lucabot_description/model/link2_down.dae"
105       />
106     </geometry>
107   </visual>
108   <collision>
109     <origin rpy="0 0 0" xyz="0 0 0.03725"/>
110     <geometry>
111       <box size="0.158 0.120 0.1945"/>
112     </geometry>
113   </collision>
114   <inertial>
115     <origin xyz="-0.016579 0.000539 0.045038" rpy="0 0 0"/>
116     <mass value="{i2*0.803}" />
117     <inertia ixx="{i2*0.002857544}" ixy="{i2*0.000001774}" ixz="{i2
118       *0.000629942}"
119       iyy="{i2*0.002802888}" iyz="{i2*0.000000346}"
120       izz="{i2*0.001398029}" />
121   </inertial>
122 </link>

```

```

121     <joint name="r3" type="revolute">
122     <origin xyz="0 0 0.2" rpy="0 0 0" />
123     <parent link="link1" />
124     <child link="link2_down" />
125 <limit effort="40" lower="-3.142" upper="3.142" velocity="1" />
126     <axis xyz="1 0 0" />
127 </joint>
128
129     <link name="link2_up">
130         <visual>
131         <origin rpy="-${pi/2} 0 0" xyz="0 0 -0.1355"/>
132         <geometry>
133             <mesh filename="package://lucabot_description/model/
link2_up_denti_incassati.dae"/>
134         </geometry>
135     </visual>
136         <collision>
137         <origin rpy="0 0 0" xyz="0 0 0.09725"/>
138         <geometry>
139             <box size="0.120 0.120 0.1945"/>
140         </geometry>
141     </collision>
142         <inertial>
143         <origin xyz="-0.020402 0.000025 0.097778" rpy="0 0 0"/>
144         <mass value="${i2*0.857}" />
145         <inertia ixx="${i2*0.003054318}" ixy="${i2*-0.000000437}" ixz="${i2
*0.000510103}"
146             iyy="${i2*0.002847609}" iyz="${i2*0.000001337}"
147             izz="${i2*0.001356364}" />
148     </inertial>
149 </link>
150
151     <joint name="r4" type="revolute">
152     <origin xyz="0 0 0.1355" rpy="0 0 0" />
153     <parent link="link2_down" />
154     <child link="link2_up" />
155 <limit effort="40" lower="-3.142" upper="3.142" velocity="1" />
156     <axis xyz="0 0 1" />
157 </joint>
158
159     <link name="link3">
160         <visual>
161         <origin rpy="-${pi/2} 0 0" xyz="0 0 0"/>
162         <geometry>

```

```

163     <mesh filename="package://lucabot_description/model/link3_v2.dae"/>
164   </geometry>
165 </visual>
166   <collision>
167 <origin rpy="0 0 0" xyz="0 0 0.045"/>
168   <geometry>
169     <box size="0.120 0.120 0.210"/>
170   </geometry>
171 </collision>
172 <inertial>
173   <origin xyz="0.013097 0 0.066543" rpy="0 0 0"/>
174   <mass value="{i3*1.176}" />
175   <inertia ixx="{i3*0.004822326}" ixy="{i3*0.0}" ixz="{i3
176     *0.000933744}"
177     iyy="{i3*0.004764996}" iyz="{i3*0.0}"
178     izz="{i3*0.001777042}" />
179 </inertial>
180 </link>
181
182 <joint name="r5" type="revolute">
183 <origin xyz="0 0 0.1345" rpy="0 0 0" />
184 <parent link="link2_up" />
185 <child link="link3" />
186 <limit effort="20" lower="-3.142" upper="3.142" velocity="1" />
187 <axis xyz="1 0 0" />
188 </joint>
189
190 <link name="tool0"/>
191
192 <joint name="r6" type="revolute">
193 <origin xyz="0 0 0.150" rpy="0 0 0" />
194 <parent link="link3" />
195 <child link="tool0" />
196 <limit effort="25" lower="-3.142" upper="3.142" velocity="1" />
197 <axis xyz="0 0 1" />
198 </joint>
199
200 <xacro:include filename="{find 2fgt_onrobot_description)/urdf/2
201   fgt_urdf_simplified_collision.xacro" />
202 <xacro:boh parent="tool0" name="test">
203   <origin xyz="0 0 0" rpy="0 0 0" />
204 </xacro:boh>

```

```
205 <transmission name="tran1">
206   <type>transmission_interface/SimpleTransmission</type>
207   <joint name="r1">
208     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
209   </joint>
210   <actuator name="motor1">
211     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
212     <mechanicalReduction>35</mechanicalReduction>
213   </actuator>
214 </transmission>
215
216 <transmission name="tran2">
217   <type>transmission_interface/SimpleTransmission</type>
218   <joint name="r2">
219     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
220   </joint>
221   <actuator name="motor2">
222     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
223     <mechanicalReduction>35</mechanicalReduction>
224   </actuator>
225 </transmission>
226
227 <transmission name="tran3">
228   <type>transmission_interface/SimpleTransmission</type>
229   <joint name="r3">
230     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
231   </joint>
232   <actuator name="motor3">
233     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
234     <mechanicalReduction>35</mechanicalReduction>
235   </actuator>
236 </transmission>
237
238 <transmission name="tran4">
239   <type>transmission_interface/SimpleTransmission</type>
240   <joint name="r4">
241     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
```



```

242     </joint>
243     <actuator name="motor4">
244         <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
245         <mechanicalReduction>35</mechanicalReduction>
246     </actuator>
247 </transmission>
248
249 <transmission name="tran5">
250     <type>transmission_interface/SimpleTransmission</type>
251     <joint name="r5">
252         <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
253     </joint>
254     <actuator name="motor5">
255         <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
256         <mechanicalReduction>35</mechanicalReduction>
257     </actuator>
258 </transmission>
259
260 <transmission name="tran6">
261     <type>transmission_interface/SimpleTransmission</type>
262     <joint name="r6">
263         <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
264     </joint>
265     <actuator name="motor6">
266         <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
267         <mechanicalReduction>35</mechanicalReduction>
268     </actuator>
269 </transmission>
270
271 <gazebo>
272     <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
273         <robotNamespace>/lucabot</robotNamespace>
274     </plugin>
275 </gazebo>
276
277 </robot>

```

Listings A.5: lucabot.xacro

```

1 <?xml version="1.0"?>

```

```

2 <robot name="robot"
3   xmlns:xacro="http://www.ros.org/wiki/xacro">
4   <!--Creo il gripper come macro in modo da poterla importare semplicemente
5     -->
6   <!--parametro *origin significa che passo il blocco origin-->
7   <xacro:macro name="boh" params="parent name *origin robot_namespace:=''">
8     <link name="${name}_qc_link">
9       <inertial>
10        <origin xyz="0 0 0" rpy="0 0 0"/>
11        <mass value="0.1"/>
12        <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0"/>
13      </inertial>
14
15      <visual>
16        <origin xyz="0 0 0" rpy="-${pi / 2} 0 0"/>
17        <geometry>
18          <mesh filename="package://2fgt_onrobot_description/model/quick-
19            changer.dae"/>
20        </geometry>
21      </visual>
22
23      <collision>
24        <origin xyz="0 0 0.0055" rpy="0 0 0"/>
25        <geometry>
26          <cylinder length="0.0155" radius="0.036"/>
27        </geometry>
28      </collision>
29    </link>
30
31    <link name="${name}_gripper_body_link">
32      <inertial>
33        <origin xyz="0 0 0" rpy="0 0 0"/>
34        <mass value="0.1"/>
35        <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0"/>
36      </inertial>
37
38      <visual>
39        <origin xyz="0 0 0" rpy="${pi / 2} 0 ${pi / 2}"/>
40        <geometry>
41          <mesh filename="package://2fgt_onrobot_description/model/grip-
42            block-origin.dae"/>
43        </geometry>
44      </visual>

```

```

43
44     <collision>
45         <origin xyz="0 0 0.06" rpy="0 0 0"/>
46         <geometry>
47             <box size="0.07 0.093 0.098"/>
48         </geometry>
49     </collision>
50 </link>
51
52 <!--Joint tra parent e quick changer-->
53 <joint name="${name}_qc_tool0_joint" type="fixed">
54     <xacro:insert_block name="origin"/>
55     <parent link="${parent}"/>
56     <child link="${name}_qc_link"/>
57 </joint>
58
59 <!--Joint tra quick changer e body pinza-->
60 <joint name="${name}_qc_body_joint" type="fixed">
61     <origin xyz="0 0 0.0025"/>
62     <parent link="${name}_qc_link"/>
63     <child link="${name}_gripper_body_link"/>
64 </joint>
65
66 <link name="${name}_left_finger_link">
67     <inertial>
68         <origin xyz="0 0 0" rpy="0 0 0"/>
69         <mass value="0.1"/>
70         <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0"/>
71     </inertial>
72
73     <visual>
74         <origin xyz="0 0 0" rpy="${pi / 2} 0 ${pi / 2}"/>
75         <geometry>
76             <mesh filename="package://2fqt_onrobot_description/model/left-
77             finger-origin.dae"/>
78         </geometry>
79     </visual>
80
81     <collision>
82         <origin xyz="0 0 0" rpy="${pi / 2} 0 ${pi / 2}"/>
83         <geometry>
84             <mesh filename="package://2fqt_onrobot_description/model/left-
85             finger-origin.dae"/>
86         </geometry>

```

```

85     </collision>
86 </link>
87
88 <!--Joint tra body gripper e dito sinistro-->
89 <joint name="\${name}_body_finger_l_joint" type="prismatic">
90   <axis xyz="0 1 0"/>
91   <limit effort="1" lower="-0.001" upper="0.025" velocity="1.0"/>
92   <origin xyz="0 0.0195 0.1098"/>
93   <parent link="\${name}_gripper_body_link"/>
94   <child link="\${name}_left_finger_link"/>
95 </joint>
96
97 <link name="\${name}_right_finger_link">
98   <inertial>
99     <origin xyz="0 0 0" rpy="0 0 0"/>
100    <mass value="0.1"/>
101    <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0"/>
102  </inertial>
103
104  <visual>
105    <origin xyz="0 0 0" rpy="\${pi / 2} 0 \${pi / 2}"/>
106    <geometry>
107      <mesh filename="package://2fgt_onrobot_description/model/right-
finger-origin.dae"/>
108    </geometry>
109  </visual>
110
111  <collision>
112    <origin xyz="0 0 0" rpy="\${pi / 2} 0 \${pi / 2}"/>
113    <geometry>
114      <mesh filename="package://2fgt_onrobot_description/model/right-
finger-origin.dae"/>
115    </geometry>
116  </collision>
117 </link>
118
119 <joint name="\${name}_body_finger_r_joint" type="prismatic">
120   <axis xyz="0 -1 0"/>
121   <limit effort="1" lower="-0.001" upper="0.025" velocity="1"/>
122   <!--Mimic the joint: moving together-->
123   <mimic joint="\${name}_body_finger_l_joint" multiplier="1" offset="0"
/>
124   <origin xyz="0 -0.0195 0.1098"/>
125   <parent link="\${name}_gripper_body_link"/>

```

```

126     <child link="${name}_right_finger_link"/>
127 </joint>
128
129 <!-- Adding the transmission only on the left finger because the right
is already mimic that-->
130 <transmission name="${name}_left_finger_transmission">
131     <type>transmission_interface/SimpleTransmission</type>
132     <joint name="${name}_body_finger_l_joint">
133         <hardwareInterface>hardware_interface/PositionJointInterface</
hardwareInterface>
134     </joint>
135     <actuator name="${name}_left_joint_motor">
136         <mechanicalReduction>1</mechanicalReduction>
137     </actuator>
138 </transmission>
139
140 <!--Adding the Gazebo plug-in for ROS_control-->
141 <gazebo>
142     <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
143         <xacro:unless value="${robot_namespace == ''}">
144             <robotNamespace>${robot_namespace}</robotNamespace>
145         </xacro:unless>
146         <controlPeriod>0.001</controlPeriod>
147     </plugin>
148
149 <!--Adding a Plugin for joint mimic in gazebo-->
150 <plugin name="${name}_body_finger_l_joint_mimic_joint_plugin"
filename="libroboticsgroup_gazebo_mimic_joint_plugin.so">
151     <joint>${name}_body_finger_l_joint</joint>
152     <mimicJoint>${name}_body_finger_r_joint</mimicJoint>
153     <!--In teoria andrebbe il true qui: ma bisogna fixare i parametri
del PID e della dinamica del gripper...-->
154     <xacro:if value="false">         <!-- if set to true, PID
parameters from "/gazebo_ros_control/pid_gains/${mimic_joint}" are
loaded -->
155         <hasPID />
156     </xacro:if>
157     <multiplier>1</multiplier>
158     <offset>0</offset>
159     <sensitiveness>0</sensitiveness>         <!-- if absolute difference
between setpoint and process value is below this threshold, do nothing;
0.0 = disable [rad] -->
160     <maxEffort>1</maxEffort>         <!-- only taken into account if
has_pid:=true [Nm] -->

```

```
161     <xacro:unless value="${robot_namespace == ''}">
162         <robotNamespace>${robot_namespace}</robotNamespace>
163     </xacro:unless>
164     </plugin>
165 </gazebo>
166
167 </xacro:macro>
168
169 </robot>
```

Listings A.6: 2fgt_urdf_simplified_collision.xacro

A.6. MoveIt! configuration files for robot model with gripper

The following configuration files are a part of the whole list of files generated by the MoveIt! setup assistant in the config directory. The ones reported below are considered more relevant for the project developed in this thesis.

```
1 # joint_limits.yaml allows the dynamics properties specified in the URDF to
   # be overwritten or augmented as needed
2 # Specific joint properties can be changed with the keys [max_position,
   # min_position, max_velocity, max_acceleration]
3 # Joint limits can be turned off with [has_velocity_limits,
   # has_acceleration_limits]
4 joint_limits:
5   r1:
6     has_velocity_limits: true
7     max_velocity: 1
8     has_acceleration_limits: false
9     max_acceleration: 0
10  r2:
11     has_velocity_limits: true
12     max_velocity: 1
13     has_acceleration_limits: false
14     max_acceleration: 0
15  r3:
16     has_velocity_limits: true
17     max_velocity: 1
18     has_acceleration_limits: false
19     max_acceleration: 0
20  r4:
21     has_velocity_limits: true
```

```

22   max_velocity: 1
23   has_acceleration_limits: false
24   max_acceleration: 0
25 r5:
26   has_velocity_limits: true
27   max_velocity: 1
28   has_acceleration_limits: false
29   max_acceleration: 0
30 r6:
31   has_velocity_limits: true
32   max_velocity: 1
33   has_acceleration_limits: false
34   max_acceleration: 0
35 test_body_finger_l_joint:
36   has_velocity_limits: true
37   max_velocity: 1
38   has_acceleration_limits: false
39   max_acceleration: 0
40 test_body_finger_r_joint:
41   has_velocity_limits: true
42   max_velocity: 1
43   has_acceleration_limits: false
44   max_acceleration: 0

```

Listings A.7: joint_limits.yaml

```

1 manipulator:
2   kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
3   kinematics_solver_search_resolution: 0.005
4   kinematics_solver_timeout: 0.005

```

Listings A.8: kinematics.yaml

```

1 <?xml version="1.0" ?>
2 <!--This does not replace URDF, and is not an extension of URDF.
3   This is a format for representing semantic information about the robot
4   structure.
5   A URDF file must exist for this robot as well, where the joints and the
6   links that are referenced are defined
7 -->
8 <robot name="lucabot">
9   <!--GROUPS: Representation of a set of joints and links. This can be
10  useful for specifying DOF to plan for, defining arms, end effectors, etc
11 -->
12   <!--LINKS: When a link is specified, the parent joint of that link (if
13 it exists) is automatically included-->

```

```

9   <!--JOINTS: When a joint is specified, the child link of that joint (
which will always exist) is automatically included-->
10  <!--CHAINS: When a chain is specified, all the links along the chain (
including endpoints) are included in the group. Additionally, all the
joints that are parents to included links are also included. This means
that joints along the chain and the parent joint of the base link are
included in the group-->
11  <!--SUBGROUPS: Groups can also be formed by referencing to already
defined group names-->
12  <group name="manipulator">
13      <joint name="fixed" />
14      <joint name="r1" />
15      <joint name="r2" />
16      <joint name="r3" />
17      <joint name="r4" />
18      <joint name="r5" />
19      <joint name="r6" />
20  </group>
21  <group name="gripper">
22      <link name="test_qc_link" />
23      <link name="test_right_finger_link" />
24      <link name="test_gripper_body_link" />
25      <link name="test_left_finger_link" />
26  </group>
27  <!--GROUP STATES: Purpose: Define a named state for a particular group,
in terms of joint values. This is useful to define states like 'folded
arms'-->
28  <group_state name="open" group="gripper">
29      <joint name="test_body_finger_l_joint" value="0.025" />
30      <joint name="test_body_finger_r_joint" value="0" />
31  </group_state>
32  <group_state name="close" group="gripper">
33      <joint name="test_body_finger_l_joint" value="-0.001" />
34      <joint name="test_body_finger_r_joint" value="0" />
35  </group_state>
36  <!--END EFFECTOR: Purpose: Represent information about an end effector.
-->
37  <end_effector name="gripper" parent_link="tool0" group="gripper" />
38  <!--VIRTUAL JOINT: Purpose: this element defines a virtual joint
between a robot link and an external frame of reference (considered
fixed with respect to the robot)-->
39  <virtual_joint name="W1" type="fixed" parent_frame="world" child_link="
world" />
40  <!--DISABLE COLLISIONS: By default it is assumed that any link of the

```



```
robot could potentially come into collision with any other link in the
robot. This tag disables collision checking between a specified pair of
links. -->
41 <disable_collisions link1="base" link2="link0" reason="Adjacent" />
42 <disable_collisions link1="link0" link2="link1" reason="Adjacent" />
43 <disable_collisions link1="link1" link2="link2_down" reason="Adjacent"
/>
44 <disable_collisions link1="link2_down" link2="link2_up" reason="
Adjacent" />
45 <disable_collisions link1="link2_up" link2="link3" reason="Adjacent" />
46 <disable_collisions link1="link2_up" link2="test_gripper_body_link"
reason="Never" />
47 <disable_collisions link1="link2_up" link2="test_left_finger_link"
reason="Never" />
48 <disable_collisions link1="link2_up" link2="test_qc_link" reason="Never
" />
49 <disable_collisions link1="link2_up" link2="test_right_finger_link"
reason="Never" />
50 <disable_collisions link1="link3" link2="test_gripper_body_link" reason
="Never" />
51 <disable_collisions link1="link3" link2="test_left_finger_link" reason=
"Never" />
52 <disable_collisions link1="link3" link2="test_qc_link" reason="Adjacent
" />
53 <disable_collisions link1="link3" link2="test_right_finger_link" reason
="Never" />
54 <disable_collisions link1="test_gripper_body_link" link2="
test_left_finger_link" reason="Adjacent" />
55 <disable_collisions link1="test_gripper_body_link" link2="test_qc_link"
reason="Adjacent" />
56 <disable_collisions link1="test_gripper_body_link" link2="
test_right_finger_link" reason="Adjacent" />
57 <disable_collisions link1="test_left_finger_link" link2="test_qc_link"
reason="Never" />
58 <disable_collisions link1="test_left_finger_link" link2="
test_right_finger_link" reason="Never" />
59 <disable_collisions link1="test_qc_link" link2="test_right_finger_link"
reason="Never" />
60 </robot>
```

Listings A.9: lucabot.srdf

A.7. Code to launch the virtual model with gripper in MoveIt!

```

1 <launch>
2 <!-- specify the planning pipeline -->
3 <arg name="pipeline" default="ompl" />
4
5 <!-- By default, we do not start a database (it can be large) -->
6 <arg name="db" default="false" />
7 <!-- Allow user to specify database location -->
8 <arg name="db_path" default="$(find move_it_lucabot)/
  default_warehouse_mongo_db" />
9
10 <!-- By default, we are not in debug mode -->
11 <arg name="debug" default="false" />
12
13 <!-- By default, we will load or override the robot_description -->
14 <arg name="load_robot_description" default="true"/>
15
16 <!-- Set execution mode for fake execution controllers -->
17 <arg name="execution_type" default="interpolate" />
18
19 <!--
20 By default, hide joint_state_publisher's GUI
21
22 MoveIt!'s "demo" mode replaces the real robot driver with the
  joint_state_publisher.
23 The latter one maintains and publishes the current joint configuration of
  the simulated robot.
24 It also provides a GUI to move the simulated robot around "manually".
25 This corresponds to moving around the real robot without the use of
  MoveIt!.
26 -->
27 <arg name="use_gui" default="false" />
28 <arg name="use_rviz" default="true" />
29
30 <!-- If needed, broadcast static tf for robot root -->
31
32
33 <!-- We do not have a robot connected, so publish fake joint states -->
34 <node name="joint_state_publisher" pkg="joint_state_publisher" type="
  joint_state_publisher" unless="$(arg use_gui)">
35   <rosparam param="source_list">[move_group/fake_controller_joint_states]

```

```

    </rosparam>
36 </node>
37 <node name="joint_state_publisher" pkg="joint_state_publisher_gui" type="
    joint_state_publisher_gui" if="$(arg use_gui)">
38   <rosparam param="source_list">[move_group/fake_controller_joint_states]
    </rosparam>
39 </node>
40
41 <!-- Given the published joint states, publish tf for the robot links -->
42 <node name="robot_state_publisher" pkg="robot_state_publisher" type="
    robot_state_publisher" respawn="true" output="screen" />
43
44 <!-- Run the main MoveIt! executable without trajectory execution (we do
    not have controllers configured by default) -->
45 <include file="$(find move_it_lucabot)/launch/move_group.launch">
46   <arg name="allow_trajectory_execution" value="true"/>
47   <arg name="fake_execution" value="true"/>
48   <arg name="execution_type" value="$(arg execution_type)"/>
49   <arg name="info" value="true"/>
50   <arg name="debug" value="$(arg debug)"/>
51   <arg name="pipeline" value="$(arg pipeline)"/>
52   <arg name="load_robot_description" value="$(arg load_robot_description)
    "/>
53 </include>
54
55 <!-- Run Rviz and load the default config to see the state of the
    move_group node -->
56 <include file="$(find move_it_lucabot)/launch/moveit_rviz.launch" if="$(
    arg use_rviz)">
57   <arg name="rviz_config" value="$(find move_it_lucabot)/launch/moveit.
    rviz"/>
58   <arg name="debug" value="$(arg debug)"/>
59 </include>
60
61 <!-- If database loading was enabled, start mongodb as well -->
62 <include file="$(find move_it_lucabot)/launch/default_warehouse_db.launch
    " if="$(arg db)">
63   <arg name="moveit_warehouse_database_path" value="$(arg db_path)"/>
64 </include>
65
66 </launch>

```

Listings A.10: demo.launch

A.8. Codes to launch the virtual model with gripper in Gazebo

```

1 <?xml version="1.0"?>
2 <launch>
3   <!--Loading the parameter of the robot-->
4   <param name="robot_description" command="$(find xacro)/xacro --inorder $(
5     find lucabot_description)/urdf/lucabot.xacro"/>
6   <!-- Load gazebo controller configurations -->
7   <!-- Note: You MUST load these PID parameters for all joints that are
8     using
9     the PositionJointInterface, otherwise the arm + gripper will act
10    like a
11    giant parachute, counteracting gravity-->
12   <rosparam file="$(find 2fgt_onrobot_description)/config/gripper_gains.
13     yaml" command="load"/>
14
15   <!--Launch Gazebo with empty world-->
16   <include file="$(find gazebo_ros)/launch/empty_world.launch">
17     <arg name="paused" value="true"/>
18     <arg name="world_name" value="$(find lucabot_gazebo)/world/lucabot.world"
19     />
20     <arg name="use_sim_time" value="true"/>
21     <!-- more default parameters can be changed here -->
22   </include>
23
24   <!--load ros_control config: joints state and controller for joints-->
25
26   <rosparam file="$(find lucabot_control)/config/robot_controllers.yaml"
27     command="load"/>
28   <!--Spawn robot into gazebo-->
29   <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
30     robot_description -urdf -model lucabot"/>
31
32   <!--Robot state publisher-->
33   <node name="robot_state_publisher" pkg="robot_state_publisher" type="
34     robot_state_publisher" output="screen">
35     <remap from="/joint_states" to="/lucabot/joint_states"/>
36   </node>
37
38   <!--Start controllers-->
39   <node name="controller_spawner" pkg="controller_manager" type="spawner"

```

```

output="screen" ns="/lucabot" respawn="false" args="
  joint_state_controller joint1_position_controller
  joint2_position_controller joint3_position_controller
  joint4_position_controller joint5_position_controller
  joint6_position_controller gripper_controller"/>
33
34 <!--Load joint trajectory controller-->
35 <node name="rqt_joint_trajectory_controller" pkg="
  rqt_joint_trajectory_controller" type="rqt_joint_trajectory_controller"/
  >
36 </launch>

```

Listings A.11: lucabot.launch

```

1 <?xml version="1.0" ?>
2 <sdf version="1.4">
3   <world name="default">
4     <include>
5       <uri>model://ground_plane</uri>
6     </include>
7     <include>
8       <uri>model://sun</uri>
9     </include>
10    <include>
11      <uri>model://table</uri>
12      <pose>0 0 -1.03 0 0 0</pose>
13    </include>
14  </world>
15 </sdf>

```

Listings A.12: lucabot.world

A.9. Codes to control the virtual model with gripper in Gazebo

The files below: `robot_controllers.yaml` and `gripper_gains.yaml` are needed to define the type of controllers and the gains which are exploited to control the dynamic model of the robot in Gazebo. These yaml files are loaded in the launch file A.11 on the preceding page.

```

1 lucabot:
2   joint_state_controller:
3     type: joint_state_controller/JointStateController
4     publish_rate: 500

```

```

5 joint1_position_controller:
6   type: effort_controllers/JointPositionController
7   joint: r1
8   pid: {p: 10000.0, i: 0.5, d: 1.0}
9 joint2_position_controller:
10  type: effort_controllers/JointPositionController
11  joint: r2
12  pid: {p: 10000.0, i: 0.5, d: 1.0}
13 joint3_position_controller:
14  type: effort_controllers/JointPositionController
15  joint: r3
16  pid: {p: 10000.0, i: 0.5, d: 1.0}
17 joint4_position_controller:
18  type: effort_controllers/JointPositionController
19  joint: r4
20  pid: {p: 10000.0, i: 0.5, d: 1.0}
21 joint5_position_controller:
22  type: effort_controllers/JointPositionController
23  joint: r5
24  pid: {p: 10000.0, i: 0.5, d: 1000.0}
25 joint6_position_controller:
26  type: effort_controllers/JointPositionController
27  joint: r6
28  pid: { p: 10000.0, i: 0.5, d: 1.0 }
29 gripper_controller:
30  type: position_controllers/JointTrajectoryController
31  joints:
32    # Check "test" is the name parameter
33    - test_body_finger_1_joint
34  constraints:
35    goal_time: 0.6
36    stopped_velocity_tolerance: 0.05
37    gripper_finger_joint: { trajectory: 0.2, goal: 0.2 }
38  stop_trajectory_duration: 0.5
39  state_publish_rate: 125
40  action_monitor_rate: 10

```

Listings A.13: robot_controllers.yaml

```

1 # Note: You MUST load these PID parameters for all joints that are using
   the
2 # PositionJointInterface, otherwise the arm + gripper will act like a giant
3 # parachute, counteracting gravity, and causing some of the wheels to lose
4 # contact with the ground, so the robot won't be able to properly navigate.
   See

```

```
5 # https://github.com/ros-simulation/gazebo_ros_pkgs/issues/612
6 gazebo_ros_control:
7   pid_gains:
8     # these gains are used by the gazebo_ros_control plugin
9     test_body_finger_l_joint:
10      p: 200.0
11      i: 0.1
12      d: 0.0
13      i_clamp: 0.2
14      antiwindup: false
15      publish_state: true
16     # the following gains are used by the gazebo_mimic_joint plugin
17     test_body_finger_r_joint:
18      p: 200.0
19      i: 0.1
20      d: 0.0
21      i_clamp: 0.2
22      antiwindup: false
23      publish_state: true
```

Listings A.14: gripper_gains.yaml

A.10. Code to plan the pick and place task in MoveIt!

```
1 #!/usr/bin/env python
2
3
4 from copy import deepcopy
5 from moveit_msgs.msg import *
6 from moveit_commander import *
7 from trajectory_msgs.msg import JointTrajectoryPoint
8 import tf
9 from os import path
10 from time import strftime, localtime
11 from geometry_msgs.msg import *
12 import matplotlib.pyplot as plt
13 from matplotlib.figure import Figure, Axes, rcParams
14 from math import pi
15 from tf.transformations import quaternion_from_euler
16 import os
17
18 os.environ["PATH"] += os.pathsep + '/usr/bin'
19
20 # Set Latex font
```

```
21 rcParams['text.usetex'] = True
22 rcParams['text.latex.unicode'] = True
23
24 # Initialization of Moveit!
25 rospy.loginfo('Starting the Initialization')
26 roscpp_initialize(sys.argv)
27 # Initialization of the Node
28 rospy.init_node('plan_motion', anonymous=True, log_level=rospy.INFO)
29
30 robot = RobotCommander()
31 scene = PlanningSceneInterface()
32 rospy.sleep(1.0)
33
34 planner = "RRT"
35 group = MoveGroupCommander("manipulator")
36 group.set_pose_reference_frame("base")
37 group.set_planner_id(planner)
38 group.set_num_planning_attempts(1000)
39 group.allow_replanning(False) # Allow the replanning if there are changes
    in environment
40
41 arm = "tool0"
42
43 group_gripper = 'gripper'
44 group_gripper2 = MoveGroupCommander("gripper")
45 group_gripper2.set_planner_id('RRTConnect')
46 group_gripper2.set_num_planning_attempts(1000)
47 group_gripper2.allow_replanning(False)
48 group_gripper2.set_goal_tolerance(0.01)
49
50 # Home position
51 quaternion = tf.transformations.quaternion_from_euler(pi, 0, 0)
52 home_pos = [0.300, 0, 0.4, quaternion[0], quaternion[1], quaternion[2],
    quaternion[3]]
53
54 images_path = path.join(path.expanduser("~"), "tesi_seria/images/{}/".
    format(planner))
55
56 class TestBox:
57
58     def __init__(self, name, x=0.0, y=0.0):
59         self.name = name
60         self.x_dim = 0.05
61         self.y_dim = 0.05
```



```

62     self.z_dim = 0.05
63     self.pose_msg = PoseStamped()
64     self.pose_msg.header.frame_id = "base"
65     self.pose_msg.pose.position.x = x
66     self.pose_msg.pose.position.y = y
67     self.pose_msg.pose.position.z = 0
68
69     def add_object(self):
70         rospy.sleep(0.5)
71         scene.add_box(self.name, self.pose_msg, size=(self.x_dim, self.
72 y_dim, self.z_dim))
73         rospy.sleep(0.5)
74
75     def remove_object(self):
76         if scene.get_attached_objects(self.name):
77             rospy.logerr('Object attached: check it please!')
78         else:
79             scene.remove_world_object(self.name)
80
81     def attach_object(self, arm):
82         touch_links = robot.get_link_names(group_gripper)
83         scene.attach_box(arm, self.name, touch_links=touch_links)
84         rospy.sleep(1.0)
85
86     def detach_object(self, arm):
87         scene.remove_attached_object(arm, self.name)
88         rospy.sleep(0.5)
89
90     def getting_joints_from_plan(plan):
91         # type: (RobotTrajectory) -> list
92         """
93         Provide the joints of the last point of the trajectory
94         :param plan: RobotTrajectory msg: Plan of a trajectory
95         :type plan: RobotTrajectory
96         :rtype: list
97         :return: List of joints of the end of the trajectory
98         """
99         positions = plan.joint_trajectory.points[-1] # type:
100 JointTrajectoryPoint
101         return positions.positions
102
103     def create_robotstate(plan):
104         """
105         Return a RobotState() msg of the end of a trajectory

```

```

104 :param plan: RobotTrajectory msg of the trajectory - planning
105 :type plan: RobotTrajectory
106 :rtype: RobotState
107 :return: robot_state: RobotState msg of the trajectory
108 """
109 if plan.joint_trajectory.points:
110     # Creating a RobotState for evaluate the next trajectory
111     joint_state = JointState()
112     joint_state.header.stamp = rospy.Time.now()
113     joint_state.name = plan.joint_trajectory.joint_names
114
115     positions = getting_joints_from_plan(plan)
116     joint_state.position = positions
117     robot_state = RobotState()
118     robot_state.joint_state = joint_state
119     return robot_state
120 else:
121     raise RuntimeError("Error in creating the robotstate: points
122 empty")
123
124 def evaluate_time(plan, info=''):
125     """
126     It returns the time duration of the trajectory
127     :param plan: Plan msg of the trajectory
128     :type plan: RobotTrajectory
129     :param info: info about the time
130     :type info: str
131     :rtype: float
132     :return: duration time of the trajectory
133     """
134     # Check if the plan is not empty
135     if plan.joint_trajectory.points:
136         p_last = plan.joint_trajectory.points[-1] # type:
137         JointTrajectoryPoint
138         duration = p_last.time_from_start.to_sec()
139         rospy.loginfo('Estimated time of planning {}: {} s'.format(info,
140 duration))
141         return duration
142     else:
143         rospy.logwarn('The plan is empty')
144         duration = 2000.0 # penalty
145         return duration
146
147 def home():

```

```
145     rospy.loginfo('home')
146     group.set_pose_target(home_pos, arm)
147     plan = group.plan()
148     home_duration = evaluate_time(plan, "homing")
149     group.execute(plan)
150     group.stop()
151     return home_duration
152
153 def picking(obj, arm, info=''):
154     # type: (TestBox, str, str) -> List[float, RobotTrajectory,
RobotTrajectory]
155     """
156     Wrapper for picking
157     :param obj: Object to pick
158     :param arm: Arm used
159     :param info Info about what is doing
160     :rtype: list[float, RobotTrajectory, RobotTrajectory] or tuple[float,
RobotTrajectory, RobotTrajectory]
161     :return: Duration time for picking and RobotTrajectories for picking
162     """
163     pose_P = deepcopy(obj.pose_msg)
164     pose_P.pose.position.z += 0.15
165     pose_P.pose.orientation.x = quaternion[0]
166     pose_P.pose.orientation.y = quaternion[1]
167     pose_P.pose.orientation.z = quaternion[2]
168     pose_P.pose.orientation.w = quaternion[3]
169
170     group.set_pose_target(pose_P, arm)
171     pick = group.plan()
172     # Evaluate the time for picking
173     t1 = evaluate_time(pick, info + "_t1")
174     if pick.joint_trajectory.points:
175         # Creating a RobotState for evaluate the next trajectory
176         robot_state = create_robotstate(pick)
177         group.set_start_state(robot_state)
178         group.set_pose_target(home_pos, arm)
179         homing = group.plan()
180         # Evaluate the duration of the planning
181         t2 = evaluate_time(homing, info + "_t2")
182         return [(t1 + t2), pick, homing]
183     else:
184         rospy.logerr('Planning failed')
185         pass
186
```

```

187 def closing():
188     rospy.loginfo('closing')
189     joint_goal = group_gripper2.get_current_joint_values()
190     joint_goal[0] = 0.0
191     group_gripper2.go(joint_goal, wait=True)
192     group_gripper2.stop()
193
194 def placing(obj, info=''):
195     # type: (TestBox, str) -> List[float, RobotTrajectory, RobotTrajectory]
196     place_pos = [0, 0.3, 0.1, quaternion[0], quaternion[1], quaternion[2],
197                 quaternion[3]]
198     group.set_pose_target(place_pos, arm)
199     placePlan = group.plan() # type: RobotTrajectory
200     # Evaluate the time of the trajectory
201     t1 = evaluate_time(placePlan, info + "_t1_placing")
202     if placePlan.joint_trajectory.points:
203         # Creating a RobotState for evaluate the next trajectory
204         group.clear_pose_targets()
205         place_state = create_robotstate(placePlan)
206         group.set_start_state(place_state)
207         group.set_pose_target(home_pos, arm)
208         return_home = group.plan() # type: RobotTrajectory
209         t2 = evaluate_time(return_home, info + "_t2_placing") # type:
210         float
211         return [(t1 + t2), placePlan, return_home]
212     else:
213         raise RuntimeError("Error: Planning failed for placing")
214
215 def opening():
216     rospy.loginfo('opening')
217     joint_goal = group_gripper2.get_current_joint_values()
218     joint_goal[0] = 0.025
219     group_gripper2.go(joint_goal, wait=True)
220     group_gripper2.stop()
221
222 def joint_diagram(plan, info=''):
223     # type: (RobotTrajectory, str) -> None
224     points = [p for p in plan.joint_trajectory.points] # type: List[
225     JointTrajectoryPoint]
226     # Create an object figure subplot
227     fig, axes = plt.subplots(3, sharex=True) # type: Figure, List[Axes]
228     # Get width and height
229     (width_fig, height_fig) = rcParams["figure.figsize"]
230     # Get hspace between subplots + 20%

```

```

228     hspace = rcParams["figure.subplot.hspace"] + 0.20
229     # Add half inch to the figure's size
230     fig.set_size_inches(width_fig + 1, height_fig + 1.1)
231     fig.subplots_adjust(hspace=hspace)
232     # For each point, I separate each joint position
233     t = [tt.time_from_start.to_sec() for tt in points]
234     j1 = [jj.positions[0] for jj in points]
235     j2 = [jj.positions[1] for jj in points]
236     j3 = [jj.positions[2] for jj in points]
237     j4 = [jj.positions[3] for jj in points]
238     j5 = [jj.positions[4] for jj in points]
239     j6 = [jj.positions[5] for jj in points]
240     axes[0].plot(
241         t, j1, 'bo-',
242         t, j2, 'go-',
243         t, j3, 'ro-',
244         t, j4, 'co-',
245         t, j5, 'mo-',
246         t, j6, 'yo-',
247     )
248     axes[0].grid()
249     # axes[0].set_title(r"\text{Joint positions - arm - plan: {0}}".format(
250     info))
251     axes[0].set_title(r"$\textbf{Joint positions - arm - plan: %s}$" % info
252     )
253     axes[0].set_xlabel(r"$\textit{time (s)}$")
254     axes[0].set_ylabel(r"$q$")
255     axes[0].legend(['j1', 'j2', 'j3', 'j4', 'j5', 'j6'], loc='best',
256     bbox_to_anchor=(1.001, 1))
257     v1 = [jj.velocities[0] for jj in points]
258     v2 = [jj.velocities[1] for jj in points]
259     v3 = [jj.velocities[2] for jj in points]
260     v4 = [jj.velocities[3] for jj in points]
261     v5 = [jj.velocities[4] for jj in points]
262     v6 = [jj.velocities[5] for jj in points]
263     axes[1].plot(
264         t, v1, 'bo-',
265         t, v2, 'go-',
266         t, v3, 'ro-',
267         t, v4, 'co-',
268         t, v5, 'mo-',
269         t, v6, 'yo-',
270     )
271     axes[1].grid()

```

```

269 axes[1].set_xlabel(r"$\textit{time (s)}$")
270 axes[1].set_ylabel(r"$\dot{q}$")
271 # axes[1].set_title(r"\text{Joint speed - arm - plan: {0}}".format(info
))
272 axes[1].set_title(r"$\textbf{Joint speed - arm - plan: %s}$" % info)
273 a1 = [jj.accelerations[0] for jj in points]
274 a2 = [jj.accelerations[1] for jj in points]
275 a3 = [jj.accelerations[2] for jj in points]
276 a4 = [jj.accelerations[3] for jj in points]
277 a5 = [jj.accelerations[4] for jj in points]
278 a6 = [jj.accelerations[5] for jj in points]
279 axes[2].plot(
280     t, a1, 'bo-',
281     t, a2, 'go-',
282     t, a3, 'ro-',
283     t, a4, 'co-',
284     t, a5, 'mo-',
285     t, a6, 'yo-',
286 )
287 axes[2].grid()
288 axes[2].set_xlabel(r"$\textit{time (s)}$")
289 axes[2].set_ylabel(r"$\ddot{q}$")
290 # axes[2].set_title(r"\text{Joint acceleration - arm - plan: {0}}".
format(info))
291 axes[2].set_title(r"$\textbf{Joint acceleration - arm - plan: %s}$" %
info)
292 # print("end time: {}".format(t[-1]))
293 fig.savefig(images_path + "J_arm_{_}".format(info, strftime("%d_%b-%
H_%M", localtime()))),
294             format='svg',
295             transparent=False
296         )
297 # plt.show()
298 # Save the timings on a file:
299 with open(path.join(path.expanduser("~"), "tesi_seria/timings"), "a+")
as f:
300     f.write(
301         strftime("%d_%b-%H_%M", localtime()) +
302         "\tPLANNER: {} J_arm_{} Time: {}\n".format(planner, info, t
[-1])
303     )
304
305 def run():
306     # Remove detached object

```

```
307     if scene.get_attached_objects():
308         scene.remove_attached_object(arm)
309
310     # Remove all the objects
311     scene.remove_world_object()
312     rospy.sleep(1.0)
313
314     # Add the test box
315     B = TestBox('box', x=0.3, y=0)
316     rospy.loginfo('Placing box')
317     rospy.sleep(0.5)
318
319     # Setting up the test box
320     B.add_object()
321     rospy.sleep(2.0)
322
323     # Going home
324     group.set_start_state_to_current_state()
325     home()
326
327     # # Evaluate the time for the arm cycle
328     (t1, pick, homing) = picking(B, arm, info="pick\t")
329     home_robotstate = create_robotstate(homing)
330     group.set_start_state(home_robotstate)
331     (t2, place, return_home) = placing(B, info="place\t")
332
333     # Create the initial state after placing the tube
334     return_home_state = create_robotstate(return_home)
335     group.set_start_state(return_home_state)
336
337     # Execute Picking
338     group.execute(pick)
339     group.stop()
340
341     # Attach Test box
342     B.attach_object(arm)
343     closing()
344
345     group.execute(homing)
346     group.stop()
347
348     # Execute Placing
349     group.execute(place)
350     group.stop()
```

```

351     opening()
352     B.detach_object(arm)
353
354     group.execute(return_home)
355     group.stop()
356
357 # Representing data and saving data
358     joint_diagram(pick, "pick")
359     joint_diagram(homing, "homing")
360     joint_diagram(place, "place ")
361     joint_diagram(return_home, "return home")
362
363 if __name__ == '__main__':
364     run()
365     roscpp_shutdown()

```

Listings A.15: pick_and_place.py

A.11. Codes for the digital model of the real robot

```

1
2 <?xml version="1.0"?>
3 <robot name="lucabot" xmlns:xacro="http://www.ros.org/wiki/xacro">
4
5 <!-- Import all Gazebo-customization elements, including Gazebo colors -->
6 <xacro:include filename="$(find lucabot_description)/urdf/lucabot.gazebo.
7   xacro" />
8
9 <xacro:property name="i" value="0.3" />
10 <xacro:property name="i1" value="0.3477285474" />
11 <xacro:property name="i2" value="0.3072289157" />
12 <xacro:property name="i3" value="0.212585034" />
13
14 <link name="world"/>
15
16 <link name="base">
17 <visual>
18 <origin rpy="0 0 0" xyz="0 0 0"/>
19 <geometry>
20 <mesh filename="package://lucabot_description/model/base.dae"/>
21 </geometry>
22 </visual>
23 <collision>
24 <origin rpy="0 0 0" xyz="0 0 0.0415"/>

```



```

24     <geometry>
25         <cylinder length="0.083" radius="0.100"/>
26     </geometry>
27 </collision>
28 <inertial>
29     <origin xyz="-0.000313 0 0.029159" rpy="0 0 0"/>
30     <mass value="{i*0.848}" />
31     <inertia ixx="{i*0.001992393}" ixy="{i*0.0}" ixz="{i*0.000007016}"
32             iyy="{i*0.002050412}" iyz="{i*0.0}"
33             izz="{i*0.003102151}" />
34 </inertial>
35 </link>
36
37 <joint name="fixed" type="fixed">
38     <origin xyz="0 0 0" rpy="0 0 0" />
39     <parent link="world" />
40     <child link="base" />
41 </joint>
42
43 <link name="link0">
44     <visual>
45         <origin rpy="0 0 0" xyz="0 0 0"/>
46         <geometry>
47             <mesh filename="package://lucabot_description/model/
link0_denti_incassati.dae"/>
48         </geometry>
49     </visual>
50     <collision>
51         <origin rpy="0 0 0" xyz="0 0 0.1255"/>
52         <geometry>
53             <box size="0.206 0.120 0.229"/>
54         </geometry>
55     </collision>
56     <inertial>
57         <origin xyz="-0.030697 0.000014 0.136133" rpy="0 0 0"/>
58         <mass value="{i*1.385}" />
59         <inertia ixx="{i*0.006060034}" ixy="{i*0.000000665}" ixz="{i
*0.001682159}"
60             iyy="{i*0.006600492}" iyz="{i*-0.000001679}"
61             izz="{i*0.003229145}" />
62     </inertial>
63 </link>
64
65     <joint name="r1" type="revolute">

```

```

66   <origin xyz="0 0 0.07" rpy="0 0 0" />
67   <parent link="base" />
68   <child link="link0" />
69   <limit effort="19.8" lower="-2.96706" upper="2.96706" velocity="1" />
70   <axis xyz="0 0 1" />
71   <dynamics damping="1.0"/>
72 </joint>
73
74   <link name="link1">
75   <visual>
76   <origin rpy="{pi/2} {pi} 0" xyz="0 0 0"/>
77   <geometry>
78     <mesh filename="package://lucabot_description/model/link1_v2_intero
79     .dae"/>
80   </geometry>
81 </visual>
82 <collision>
83 <origin rpy="0 0 0" xyz="0 0 0.1"/>
84 <geometry>
85   <box size="0.190 0.120 0.320"/>
86 </geometry>
87 </collision>
88 <inertial>
89 <origin xyz="0.028857 -0.000241 0.12016" rpy="0 0 0"/>
90 <mass value="{i1*1.783}" />
91 <inertia ixx="{i1*0.014019562}" ixy="{i1*-0.000003339}" ixz="{i1
92 *0.001190259}"
93   iyy="{i1*0.013996178}" iyz="{i1*0.000032161}"
94   izz="{i1*0.003295798}" />
95 </inertial>
96 </link>
97
98   <joint name="r2" type="revolute">
99   <origin xyz="0 0 0.180" rpy="0 0 0" />
100  <parent link="link0" />
101  <child link="link1" />
102  <limit effort="19.8" lower="{pi/4}" upper="{pi/2}" velocity="1" />
103  <axis xyz="1 0 0" />
104  <dynamics damping="1.0"/>
105 </joint>
106
107 <link name="link2_down">
108 <visual>
109 <origin rpy="{pi/2} {pi} 0" xyz="0 0 0"/>

```

```

108     <geometry>
109         <mesh filename="package://lucabot_description/model/link2_down.dae"
110     />
111     </geometry>
112 </visual>
113 <collision>
114     <origin rpy="0 0 0" xyz="0 0 0.03725"/>
115     <geometry>
116         <box size="0.158 0.120 0.1945"/>
117     </geometry>
118 </collision>
119 <inertial>
120     <origin xyz="-0.016579 0.000539 0.045038" rpy="0 0 0"/>
121     <mass value="{i2*0.803}" />
122     <inertia ixx="{i2*0.002857544}" ixy="{i2*0.000001774}" ixz="{i2
123 *0.000629942}"
124         iyy="{i2*0.002802888}" iyz="{i2*0.000000346}"
125         izz="{i2*0.001398029}" />
126 </inertial>
127 </link>
128
129     <joint name="r3" type="revolute">
130     <origin xyz="0 0 0.2" rpy="0 0 0" />
131     <parent link="link1" />
132     <child link="link2_down" />
133     <limit effort="11.7" lower="-{pi/2}" upper="{pi/2}" velocity="1" />
134     <axis xyz="1 0 0" />
135     <dynamics damping="1.0"/>
136 </joint>
137
138     <link name="link2_up">
139     <visual>
140     <origin rpy="-{pi/2} 0 0" xyz="0 0 -0.1355"/>
141     <geometry>
142         <mesh filename="package://lucabot_description/model/
143 link2_up_denti_incassati.dae"/>
144     </geometry>
145 </visual>
146 <collision>
147     <origin rpy="0 0 0" xyz="0 0 0.09725"/>
148     <geometry>
149         <box size="0.120 0.120 0.1945"/>
150     </geometry>
151 </collision>

```

```

149   <inertial>
150     <origin xyz="-0.020402 0.000025 0.097778" rpy="0 0 0"/>
151     <mass value="\${i2*0.857}" />
152     <inertia ixx="\${i2*0.003054318}" ixy="\${i2*-0.000000437}" ixz="\${i2
153       *0.000510103}"
154       iyy="\${i2*0.002847609}" iyz="\${i2*0.000001337}"
155       izz="\${i2*0.001356364}" />
156   </inertial>
157 </link>
158
159   <joint name="r4" type="revolute">
160     <origin xyz="0 0 0.1355" rpy="0 0 0" />
161     <parent link="link2_down" />
162     <child link="link2_up" />
163 <limit effort="6.9" lower="-2.96706" upper="2.96706" velocity="1" />
164     <axis xyz="0 0 1" />
165     <dynamics damping="1.0"/>
166 </joint>
167
168   <link name="link3">
169     <visual>
170       <origin rpy="-\${pi/2} 0 0" xyz="0 0 0"/>
171       <geometry>
172         <mesh filename="package://lucabot_description/model/link3_v2.dae"/>
173       </geometry>
174     </visual>
175     <collision>
176       <origin rpy="0 0 0" xyz="0 0 0.045"/>
177       <geometry>
178         <box size="0.120 0.120 0.210"/>
179       </geometry>
180     </collision>
181     <inertial>
182       <origin xyz="0.013097 0 0.066543" rpy="0 0 0"/>
183       <mass value="\${i3*1.176}" />
184       <inertia ixx="\${i3*0.004822326}" ixy="\${i3*0.0}" ixz="\${i3
185         *0.000933744}"
186         iyy="\${i3*0.004764996}" iyz="\${i3*0.0}"
187         izz="\${i3*0.001777042}" />
188     </inertial>
189 </link>
190
191   <joint name="r5" type="revolute">
192     <origin xyz="0 0 0.1345" rpy="0 0 0" />

```

```
191     <parent link="link2_up" />
192     <child link="link3" />
193 <limit effort="6.9" lower="-${pi/2}" upper="${pi/2}" velocity="1" />
194     <axis xyz="1 0 0" />
195     <dynamics damping="1.0"/>
196 </joint>
197
198     <transmission name="tran1">
199     <type>transmission_interface/SimpleTransmission</type>
200     <joint name="r1">
201         <hardwareInterface>hardware_interface/EffortJointInterface</
202 hardwareInterface>
203     </joint>
204     <actuator name="motor1">
205         <hardwareInterface>hardware_interface/EffortJointInterface</
206 hardwareInterface>
207         <mechanicalReduction>35</mechanicalReduction>
208     </actuator>
209 </transmission>
210
211 <transmission name="tran2">
212     <type>transmission_interface/SimpleTransmission</type>
213     <joint name="r2">
214         <hardwareInterface>hardware_interface/EffortJointInterface</
215 hardwareInterface>
216     </joint>
217     <actuator name="motor2">
218         <hardwareInterface>hardware_interface/EffortJointInterface</
219 hardwareInterface>
220         <mechanicalReduction>35</mechanicalReduction>
221     </actuator>
222 </transmission>
223
224 <transmission name="tran3">
225     <type>transmission_interface/SimpleTransmission</type>
226     <joint name="r3">
227         <hardwareInterface>hardware_interface/EffortJointInterface</
228 hardwareInterface>
229     </joint>
230     <actuator name="motor3">
231         <hardwareInterface>hardware_interface/EffortJointInterface</
232 hardwareInterface>
233         <mechanicalReduction>35</mechanicalReduction>
234     </actuator>
```

```

229 </transmission>
230
231 <transmission name="tran4">
232   <type>transmission_interface/SimpleTransmission</type>
233   <joint name="r4">
234     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
235   </joint>
236   <actuator name="motor4">
237     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
238     <mechanicalReduction>35</mechanicalReduction>
239   </actuator>
240 </transmission>
241
242 <transmission name="tran5">
243   <type>transmission_interface/SimpleTransmission</type>
244   <joint name="r5">
245     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
246   </joint>
247   <actuator name="motor5">
248     <hardwareInterface>hardware_interface/EffortJointInterface</
hardwareInterface>
249     <mechanicalReduction>35</mechanicalReduction>
250   </actuator>
251 </transmission>
252
253 <gazebo>
254 <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
255   <robotNamespace>/lucabot</robotNamespace>
256 </plugin>
257 </gazebo>
258
259 </robot>

```

Listings A.16: lucabot_moveit_gazebo.xacro

```

1 <?xml version='1.0' encoding='utf-8'?>
2
3 <robot name = "lucabot" xmlns:xacro="http://www.ros.org/wiki/xacro">
4
5   <xacro:macro name="lucabot_gazebo" params="name">
6     <gazebo reference="{name}_body">
7       <material>Gazebo/White</material>

```

```
8         <mu1>0.2</mu1>
9         <mu2>0.2</mu2>
10        </gazebo>
11    </xacro:macro>
12
13 </robot>
```

Listings A.17: lucabot.gazebo.xacro

A.12. MoveIt! configuration files for robot model of the real robot

```
1 # joint_limits.yaml allows the dynamics properties specified in the URDF to
   # be overwritten or augmented as needed
2 # Specific joint properties can be changed with the keys [max_position,
   # min_position, max_velocity, max_acceleration]
3 # Joint limits can be turned off with [has_velocity_limits,
   # has_acceleration_limits]
4 joint_limits:
5   r1:
6     has_velocity_limits: true
7     max_velocity: 1
8     has_acceleration_limits: true
9     max_acceleration: 0.3927
10  r2:
11    has_velocity_limits: true
12    max_velocity: 1
13    has_acceleration_limits: true
14    max_acceleration: 0.3927
15  r3:
16    has_velocity_limits: true
17    max_velocity: 1
18    has_acceleration_limits: true
19    max_acceleration: 0.3927
20  r4:
21    has_velocity_limits: true
22    max_velocity: 1
23    has_acceleration_limits: true
24    max_acceleration: 0.3927
25  r5:
26    has_velocity_limits: true
27    max_velocity: 1
28    has_acceleration_limits: true
```

```
29 max_acceleration: 0.3927
```

Listings A.18: joint_limits_real_robot.yaml

```

1 <?xml version="1.0" ?>
2 <!--This does not replace URDF, and is not an extension of URDF.
3     This is a format for representing semantic information about the robot
4     structure.
5     A URDF file must exist for this robot as well, where the joints and the
6     links that are referenced are defined
7 -->
8 <robot name="lucabot">
9     <!--GROUPS: Representation of a set of joints and links. This can be
10    useful for specifying DOF to plan for, defining arms, end effectors, etc
11 -->
12     <!--LINKS: When a link is specified, the parent joint of that link (if
13    it exists) is automatically included-->
14     <!--JOINTS: When a joint is specified, the child link of that joint (
15    which will always exist) is automatically included-->
16     <!--CHAINS: When a chain is specified, all the links along the chain (
17    including endpoints) are included in the group. Additionally, all the
18    joints that are parents to included links are also included. This means
19    that joints along the chain and the parent joint of the base link are
20    included in the group-->
21     <!--SUBGROUPS: Groups can also be formed by referencing to already
22    defined group names-->
23     <group name="manipulator">
24         <joint name="r1" />
25         <joint name="r2" />
26         <joint name="r3" />
27         <joint name="r4" />
28         <joint name="r5" />
29     </group>
30     <!--GROUP STATES: Purpose: Define a named state for a particular group,
31    in terms of joint values. This is useful to define states like 'folded
32    arms'-->
33     <group_state name="home" group="manipulator">
34         <joint name="r1" value="0" />
35         <joint name="r2" value="1.1802" />
36         <joint name="r3" value="0" />
37         <joint name="r4" value="0" />
38         <joint name="r5" value="0" />
39     </group_state>
40     <!--VIRTUAL JOINT: Purpose: this element defines a virtual joint
41    between a robot link and an external frame of reference (considered

```



```

fixed with respect to the robot)-->
28 <virtual_joint name="W1" type="fixed" parent_frame="world" child_link="
world" />
29 <!--DISABLE COLLISIONS: By default it is assumed that any link of the
robot could potentially come into collision with any other link in the
robot. This tag disables collision checking between a specified pair of
links. -->
30 <disable_collisions link1="base" link2="link0" reason="Adjacent" />
31 <disable_collisions link1="base" link2="link1" reason="Never" />
32 <disable_collisions link1="base" link2="link2_down" reason="Never" />
33 <disable_collisions link1="base" link2="link2_up" reason="Never" />
34 <disable_collisions link1="link0" link2="link1" reason="Adjacent" />
35 <disable_collisions link1="link0" link2="link2_down" reason="Never" />
36 <disable_collisions link1="link0" link2="link2_up" reason="Never" />
37 <disable_collisions link1="link0" link2="link3" reason="Never" />
38 <disable_collisions link1="link1" link2="link2_down" reason="Adjacent"
/>
39 <disable_collisions link1="link1" link2="link2_up" reason="Never" />
40 <disable_collisions link1="link1" link2="link3" reason="Never" />
41 <disable_collisions link1="link2_down" link2="link2_up" reason="
Adjacent" />
42 <disable_collisions link1="link2_down" link2="link3" reason="Never" />
43 <disable_collisions link1="link2_up" link2="link3" reason="Adjacent" />
44 </robot>

```

Listings A.19: lucabot_real_robot.srdf

```

1 # MoveIt-specific simulation settings
2
3 moveit_sim_hw_interface:
4   joint_model_group: controllers_initial_group_
5   joint_model_group_pose: controllers_initial_pose_
6 # Settings for ros_control control loop
7 generic_hw_control_loop:
8   loop_hz: 300
9   cycle_time_error_threshold: 0.01
10 # Settings for ros_control hardware interface
11 hardware_interface:
12   joints:
13     - r1
14     - r2
15     - r3
16     - r4
17     - r5
18   sim_control_mode: 1 # 0: position, 1: velocity

```

```
19 # Publish all joint states
20 lucabot:
21     # Creates the /joint_states topic necessary in ROS
22     joint_state_controller:
23         type: joint_state_controller/JointStateController
24         publish_rate: 50
25     lucabot_arm_controller:
26         type: effort_controllers/JointTrajectoryController
27         joints:
28             - r1
29             - r2
30             - r3
31             - r4
32             - r5
33         gains:
34             r1: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
35             r2: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
36             r3: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
37             r4: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
38             r5: { p: 100, d: 1, i: 0.1, i_clamp: 0.1 }
39
40         constraints:
41             goal_time: 2.0
42             state_publish_rate: 25
43
44     controller_list:
45         - name: lucabot/lucabot_arm_controller
46           action_ns: follow_joint_trajectory
47           type: FollowJointTrajectory
48           default: true
49           joints:
50               - r1
51               - r2
52               - r3
53               - r4
54               - r5
```

Listings A.20: ros_controllers.yaml

A.13. Code to launch the virtual model of the real robot in MoveIt!

```
1 <launch>
```

```

2
3 <!-- specify the planning pipeline -->
4 <arg name="pipeline" default="ompl" />
5
6 <!-- By default, we do not start a database (it can be large) -->
7 <arg name="db" default="false" />
8 <!-- Allow user to specify database location -->
9 <arg name="db_path" default="$(find move_it_gazebo)/
   default_warehouse_mongo_db" />
10
11 <!-- By default, we are not in debug mode -->
12 <arg name="debug" default="false" />
13
14 <!-- By default, we will load or override the robot_description -->
15 <arg name="load_robot_description" default="true"/>
16
17 <!-- Set execution mode for fake execution controllers -->
18 <arg name="execution_type" default="interpolate" />
19
20 <!--
21 By default, hide joint_state_publisher's GUI
22
23 MoveIt!'s "demo" mode replaces the real robot driver with the
   joint_state_publisher.
24 The latter one maintains and publishes the current joint configuration of
   the simulated robot.
25 It also provides a GUI to move the simulated robot around "manually".
26 This corresponds to moving around the real robot without the use of
   MoveIt.
27 -->
28 <arg name="use_gui" default="false" />
29 <arg name="use_rviz" default="true" />
30
31 <!-- If needed, broadcast static tf for robot root -->
32
33
34 <!-- We do not have a robot connected, so publish fake joint states -->
35 <node name="joint_state_publisher" pkg="joint_state_publisher" type="
   joint_state_publisher" unless="$(arg use_gui)">
36   <rosparam param="source_list">[move_group/fake_controller_joint_states]
   </rosparam>
37 </node>
38 <node name="joint_state_publisher" pkg="joint_state_publisher_gui" type="
   joint_state_publisher_gui" if="$(arg use_gui)">

```

```

39   <roscparam param="source_list">[move_group/fake_controller_joint_states]
    </roscparam>
40 </node>
41
42 <!-- Given the published joint states, publish tf for the robot links -->
43 <node name="robot_state_publisher" pkg="robot_state_publisher" type="
    robot_state_publisher" respawn="true" output="screen" />
44
45 <!-- Run the main MoveIt! executable without trajectory execution (we do
    not have controllers configured by default) -->
46 <include file="$(find move_it_gazebo)/launch/move_group.launch">
47   <arg name="allow_trajectory_execution" value="true"/>
48   <arg name="fake_execution" value="true"/>
49   <arg name="execution_type" value="$(arg execution_type)"/>
50   <arg name="info" value="true"/>
51   <arg name="debug" value="$(arg debug)"/>
52   <arg name="pipeline" value="$(arg pipeline)"/>
53   <arg name="load_robot_description" value="$(arg load_robot_description)
    "/>
54 </include>
55
56 <!-- Run Rviz and load the default config to see the state of the
    move_group node -->
57 <include file="$(find move_it_gazebo)/launch/moveit_rviz.launch" if="$(
    arg use_rviz)">
58   <arg name="rviz_config" value="$(find move_it_gazebo)/launch/moveit.
    rviz"/>
59   <arg name="debug" value="$(arg debug)"/>
60 </include>
61
62 <!-- If database loading was enabled, start mongodb as well -->
63 <include file="$(find move_it_gazebo)/launch/default_warehouse_db.launch"
    if="$(arg db)">
64   <arg name="moveit_warehouse_database_path" value="$(arg db_path)"/>
65 </include>
66
67 </launch>

```

Listings A.21: demo_real_robot.launch

A.14. Codes to launch the virtual model of the real robot in Gazebo

```

1 <?xml version="1.0"?>
2 <launch>
3   <arg name="paused" default="false"/>
4   <arg name="gazebo_gui" default="true"/>
5   <arg name="urdf_path" default="$(find lucabot_description)/urdf/
6     lucabot_moveit_gazebo.xacro"/>
7
8   <!-- startup simulated world -->
9   <include file="$(find gazebo_ros)/launch/empty_world.launch">
10     <arg name="world_name" default="worlds/empty.world"/>
11     <arg name="paused" value="$(arg paused)"/>
12     <arg name="gui" value="$(arg gazebo_gui)"/>
13   </include>
14
15   <!-- send robot urdf to param server -->
16   <param name="robot_description" command="$(find xacro)/xacro '$(find
17     lucabot_description)/urdf/lucabot_moveit_gazebo.xacro'"/>
18
19   <!-- push robot_description to factory and spawn robot in gazebo at the
20     origin, change x,y,z arguments to spawn in a different position -->
21   <node name="spawn_gazebo_model" pkg="gazebo_ros" type="spawn_model" args=
22     "-urdf -param robot_description -model robot -x 0 -y 0 -z 0"
23     respawn="false" output="screen" />
24
25   <include file="$(find move_it_gazebo)/launch/ros_controllers.launch"/>
26 </launch>

```

Listings A.22: gazebo.launch

```

1 <?xml version="1.0"?>
2 <launch>
3
4   <!-- Load joint controller configurations from YAML file to parameter
5     server -->
6   <rosparam file="$(find move_it_gazebo)/config/ros_controllers.yaml"
7     command="load"/>
8
9   <!-- Load the controllers -->
10  <node name="controller_spawner" pkg="controller_manager" type="spawner"
11    ns="/lucabot" respawn="false"
12    output="screen" args="--namespace=/lucabot
13      joint_state_controller
14      lucabot_arm_controller
15      --timeout 20"/>

```

```
13
14 </launch>
```

Listings A.23: ros_controllers.launch

A.15. Codes to launch the virtual model of the real robot in both MoveIt! and Gazebo

```
1 <launch>
2 <!-- By default, we do not start a database (it can be large) -->
3 <arg name="db" default="false" />
4 <!-- Allow user to specify database location -->
5 <arg name="db_path" default="$(find move_it_gazebo)/
   default_warehouse_mongo_db" />
6
7 <!-- By default, we are not in debug mode -->
8 <arg name="debug" default="false" />
9
10 <!-- By default, we will not load or override the robot_description -->
11 <arg name="load_robot_description" default="false" />
12
13 <!-- By default, hide GUI of joint_state_publisher
14
15 "demo" mode of MoveIt! replaces the real robot driver with the
   joint_state_publisher.
16 The latter one maintains and publishes the current joint configuration of
   the simulated robot.
17 It also provides a GUI to move the simulated robot around "manually".
18 This corresponds to moving around the real robot without the use of
   MoveIt.
19 -->
20 <arg name="use_gui" default="false" />
21
22 <!-- Gazebo specific options -->
23 <arg name="gazebo_gui" default="true"/>
24 <arg name="paused" default="false"/>
25 <!-- By default, use the urdf location provided from the package -->
26 <arg name="urdf_path" default="$(find lucabot_description)/urdf/
   lucabot_moveit_gazebo.xacro"/>
27
28 <!-- launch the gazebo simulator and spawn the robot -->
29 <include file="$(find move_it_gazebo)/launch/gazebo.launch" >
30   <arg name="paused" value="$(arg paused)"/>
```

```

31   <arg name="gazebo_gui" value="$(arg gazebo_gui)"/>
32   <arg name="urdf_path" value="$(arg urdf_path)"/>
33 </include>
34
35 <!-- If needed, broadcast static tf for robot root -->
36
37
38 <!-- We do not have a robot connected, so publish fake joint states -->
39 <node name="joint_state_publisher" pkg="joint_state_publisher" type="
40   joint_state_publisher" unless="$(arg use_gui)">
41   <rosparam param="source_list">[move_group/fake_controller_joint_states]
42   </rosparam>
43   <rosparam param="source_list">[lucabot/joint_states]</rosparam>
44 </node>
45 <node name="joint_state_publisher" pkg="joint_state_publisher_gui" type="
46   joint_state_publisher_gui" if="$(arg use_gui)">
47   <rosparam param="source_list">[move_group/fake_controller_joint_states]
48   </rosparam>
49   <rosparam param="source_list">[lucabot/joint_states]</rosparam>
50 </node>
51
52 <!-- Given the published joint states, publish tf for the robot links -->
53 <node name="robot_state_publisher" pkg="robot_state_publisher" type="
54   robot_state_publisher" respawn="true" output="screen" />
55
56 <!-- Run the main MoveIt! executable without trajectory execution (we do
57   not have controllers configured by default) -->
58 <include file="$(find move_it_gazebo)/launch/move_group.launch">
59   <arg name="allow_trajectory_execution" value="true"/>
60   <arg name="fake_execution" value="false"/>
61   <arg name="info" value="true"/>
62   <arg name="debug" value="$(arg debug)"/>
63   <arg name="load_robot_description" value="$(arg load_robot_description)
64   "/>
65 </include>
66
67 <!-- Run Rviz and load the default config to see the state of the
68   move_group node -->
69 <include file="$(find move_it_gazebo)/launch/moveit_rviz.launch">
70   <arg name="rviz_config" value="$(find move_it_gazebo)/launch/moveit.
71   rviz"/>
72   <arg name="debug" value="$(arg debug)"/>
73 </include>

```

```
66 <!-- If database loading was enabled, start mongodb as well -->
67 <include file="$(find move_it_gazebo)/launch/default_warehouse_db.launch"
    if="$(arg db)">
68   <arg name="moveit_warehouse_database_path" value="$(arg db_path)"/>
69 </include>
70
71 </launch>
```

Listings A.24: demo_gazebo.launch

B | Drawings

Technical drawings of the robot parts, assemblies and exploded views are attached to this document. The list of all attachments is reported hereafter, divided by parts of the harmonic drive and robot parts.

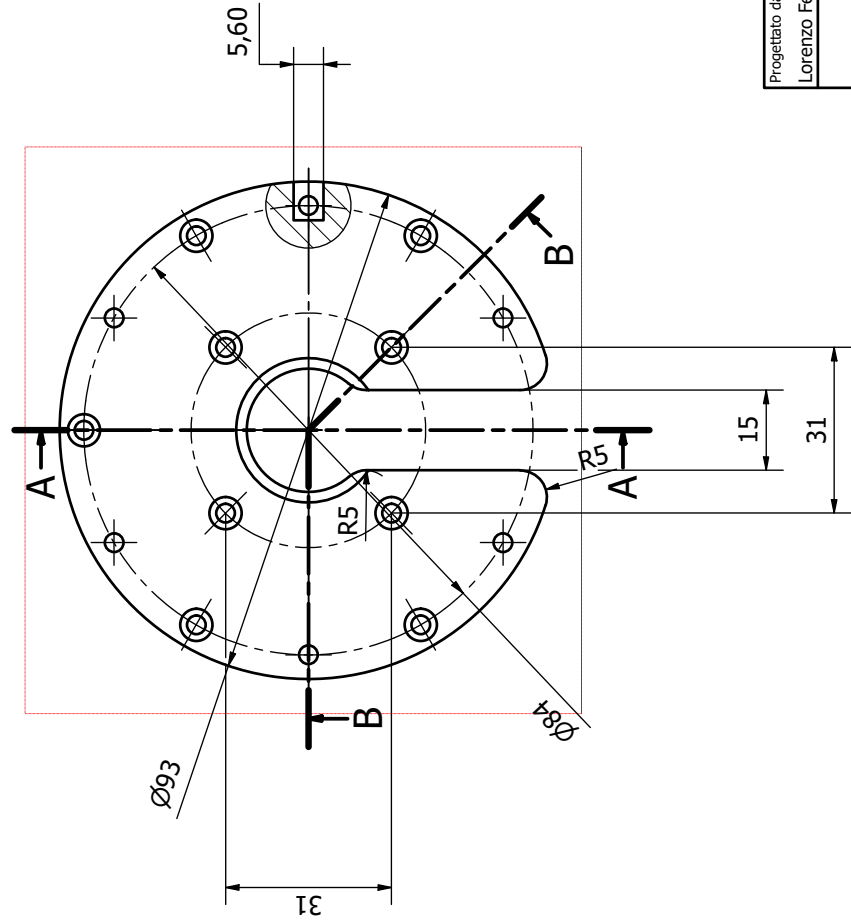
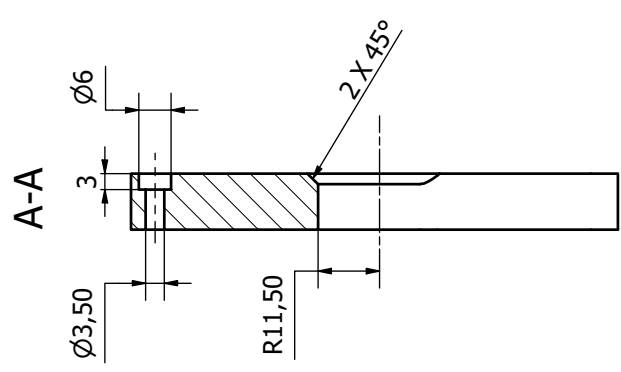
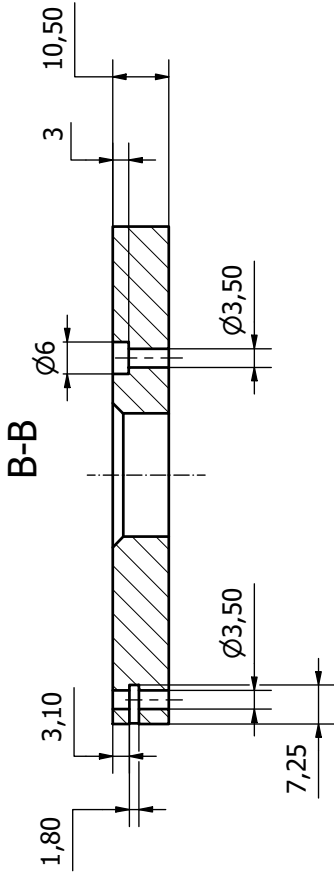
B.1. Harmonic Drive Parts


- NEMA17 base flange.
- NEMA23 base flange.
- HD cover (J1, J4, J6).
- HD cover (J2, J3, J5).
- HD circular spline v0.
- HD circular spline v1.
- Flexible spline v0.
- Flexible spline v1.
- Flexible spline v2.
- Flexible spline v3.
- NEMA17 wave generator.
- NEMA23 wave generator.
- Output flange v0.
- Output flange v1.
- Axial coupler.
- Gear ring.
- Assembled HD.

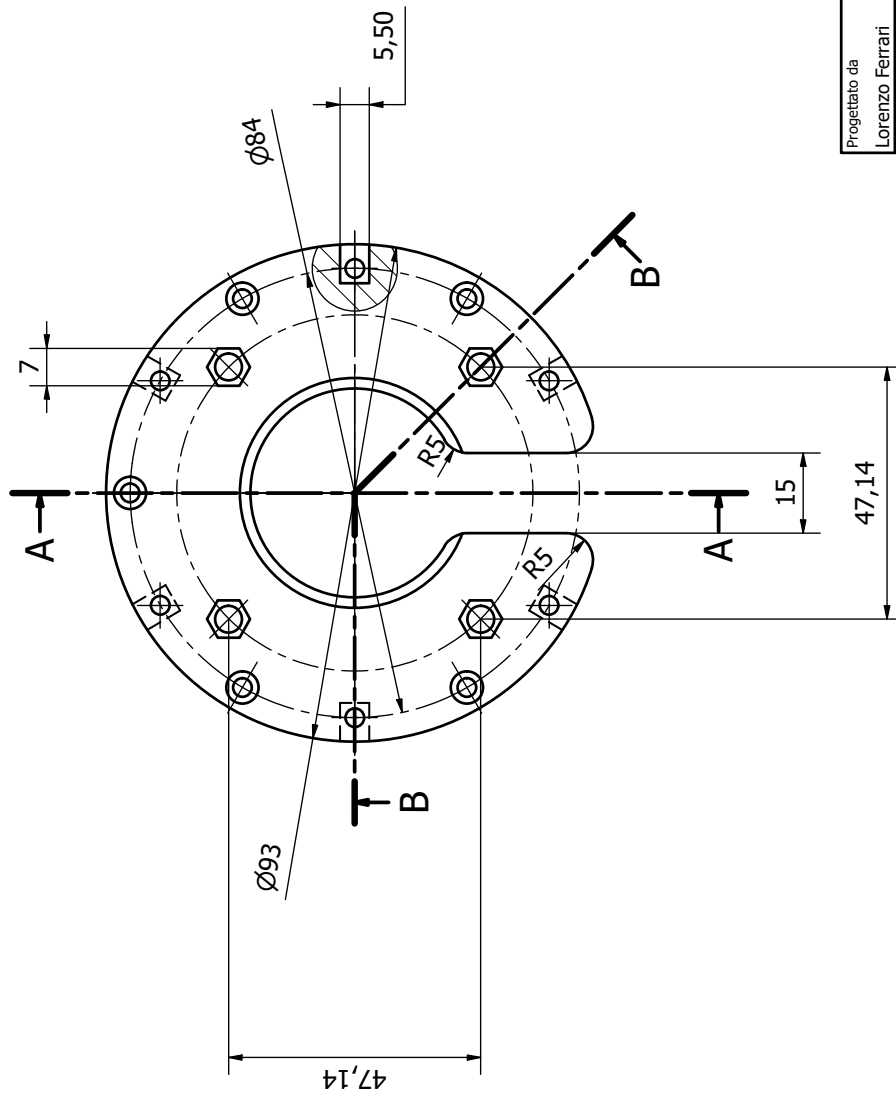
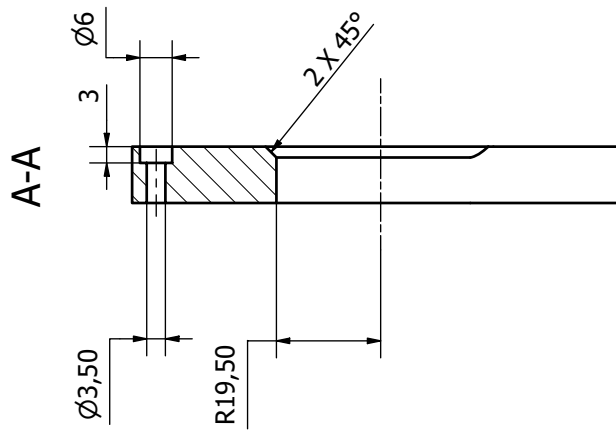
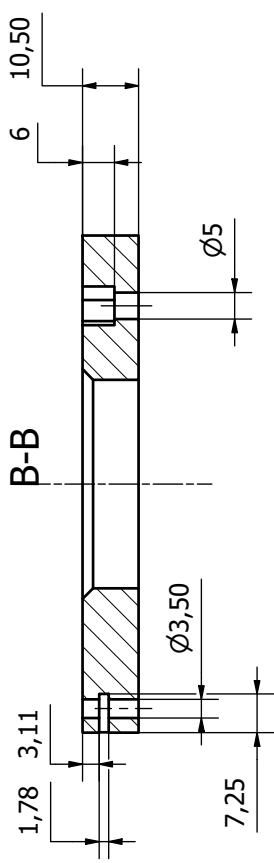
- Exploded view of HD for NEMA17 motors.
- Exploded view of HD NEMA23 motors.


B.2. Robot Parts

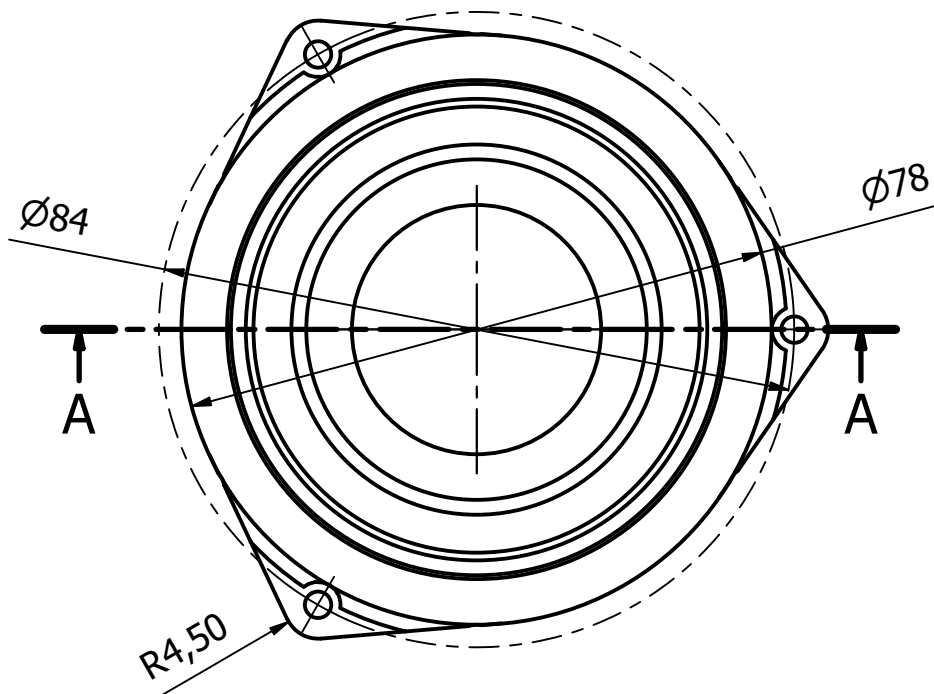
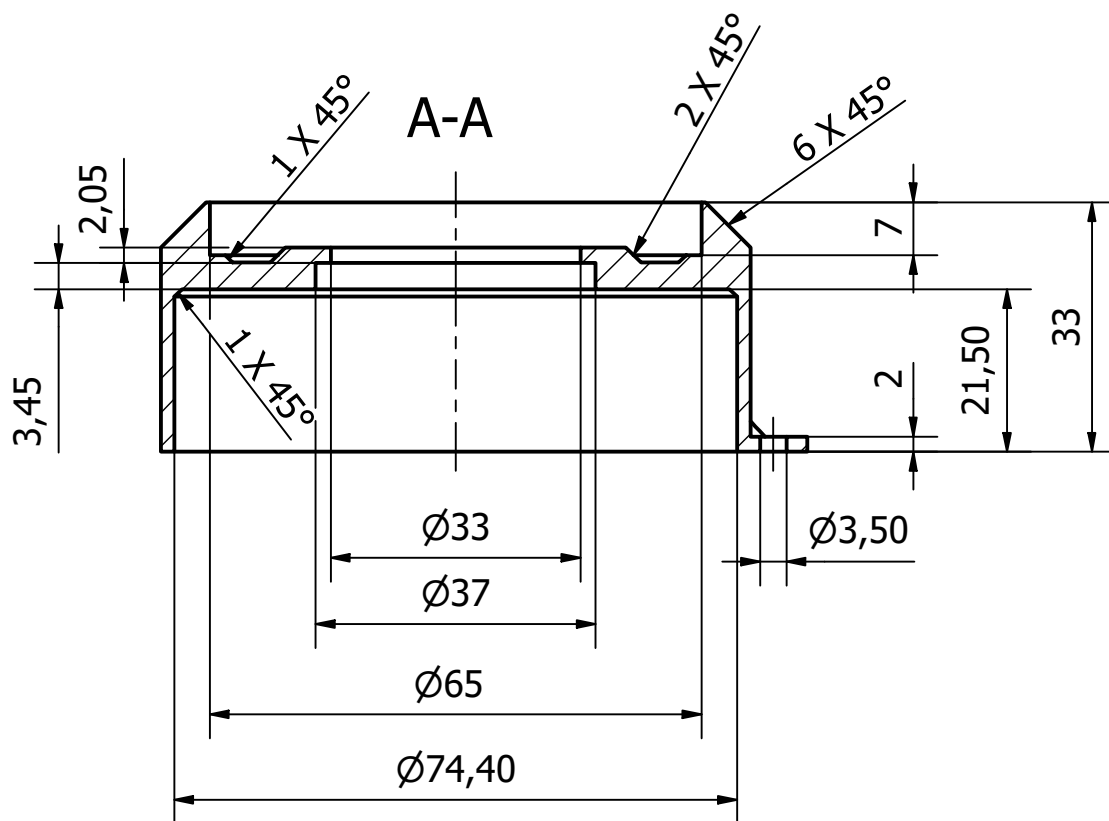
- Robot base.
- Link 0.
- Link 1.
- Link 2 bottom.
- Link 2 up.
- Module 1.
- Module 2.
- Module 3.
- Module 4.
- J4 potentiometer assembly.




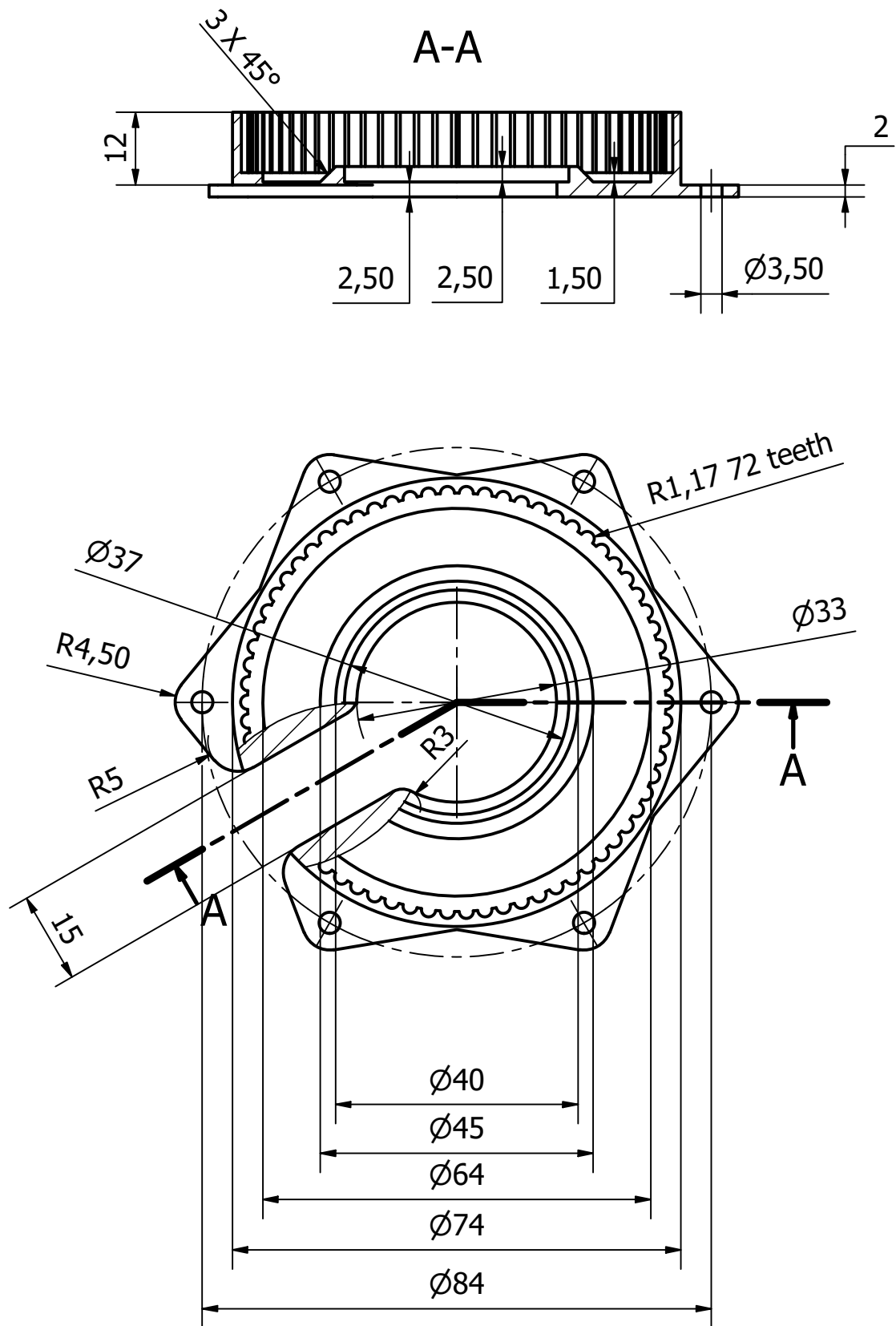
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data
 POLITECNICO MILANO 1863			Base flange for NEMA17 HD	
			NEMA17 base flange	




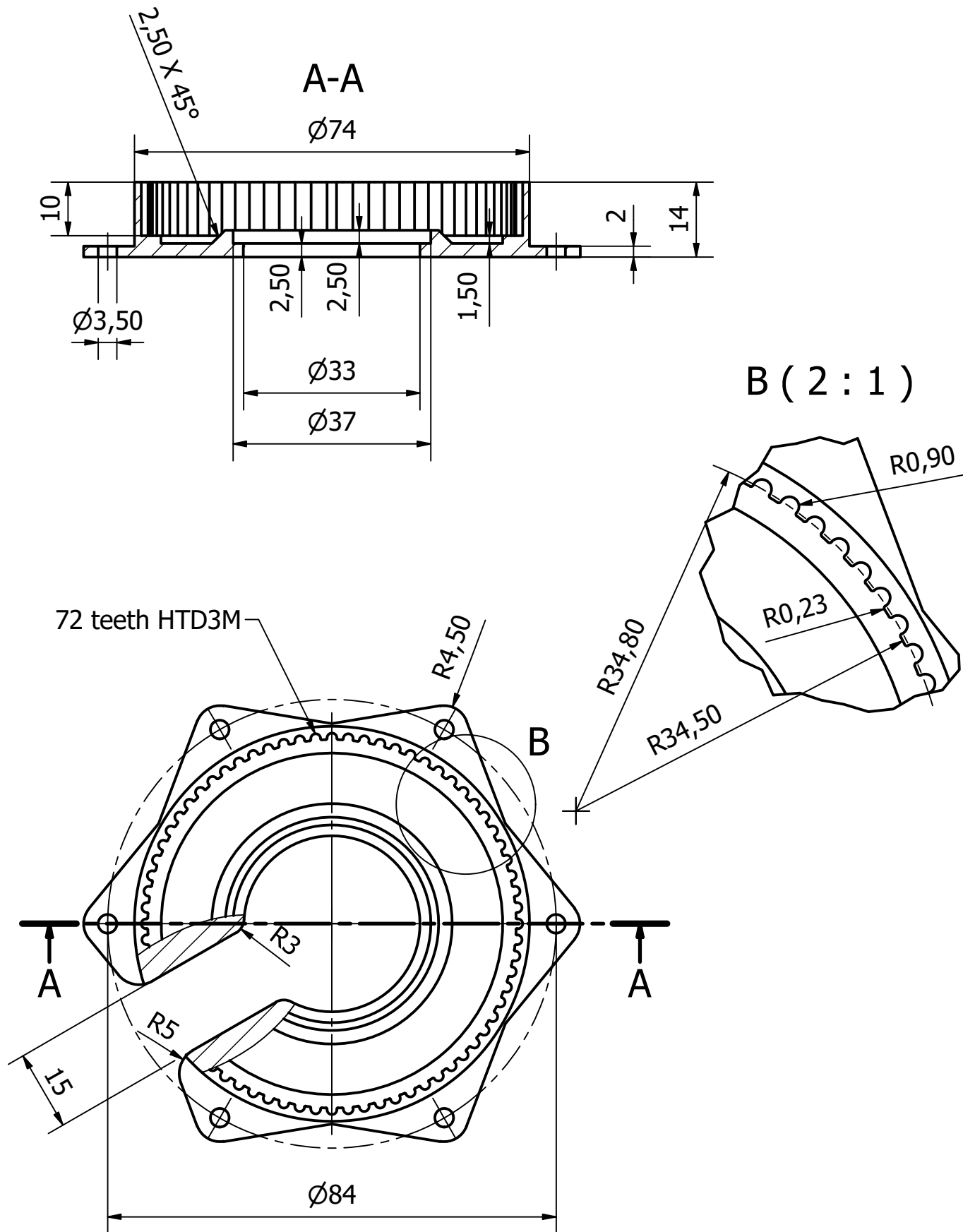
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data
 POLITECNICO MILANO 1863			Base flange for NEMA23 HD	
			NEMA23 base flange	




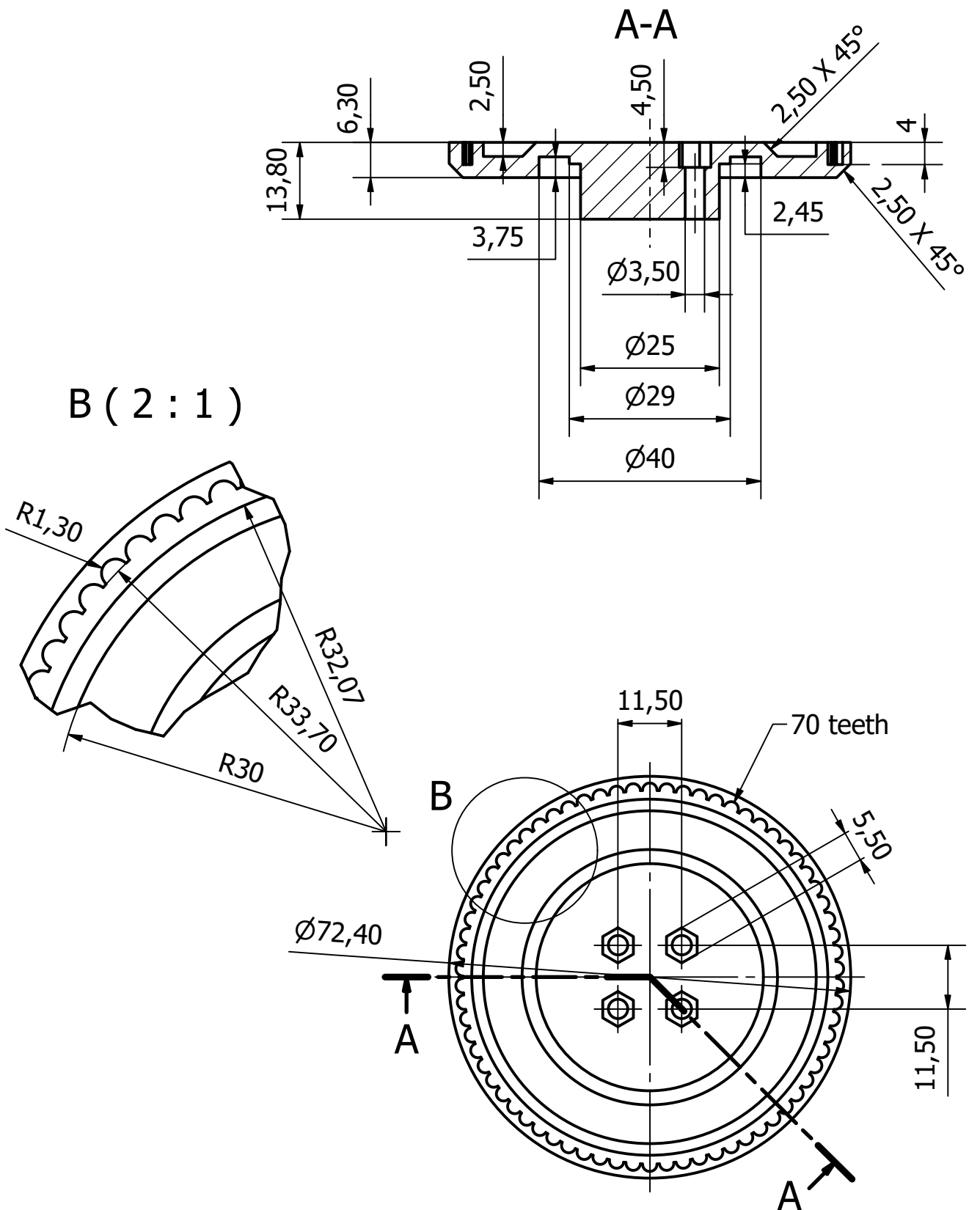
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863		HD cover for axial joints			
HD cover (J1, J4, J6)			Edizione	Foglio 1 / 1	




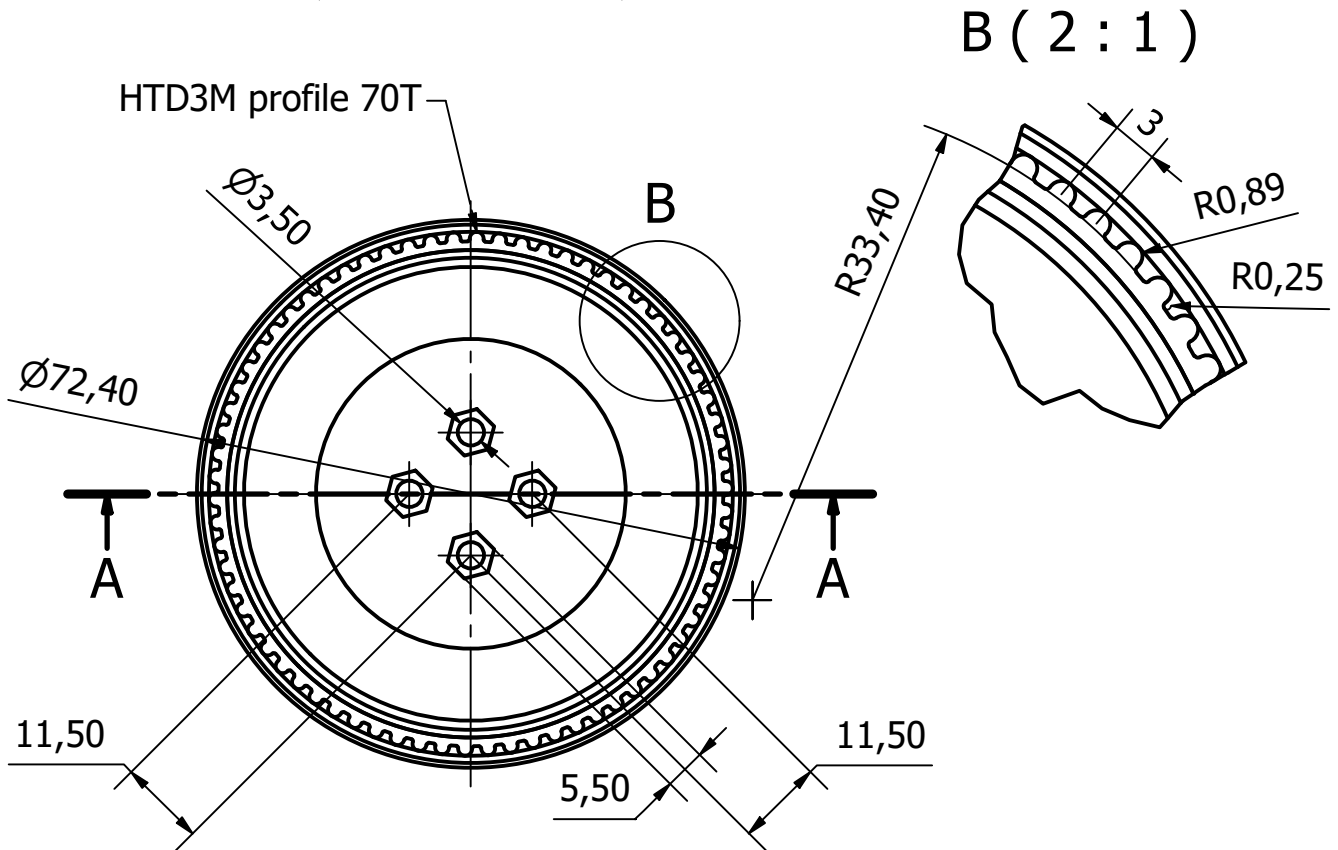
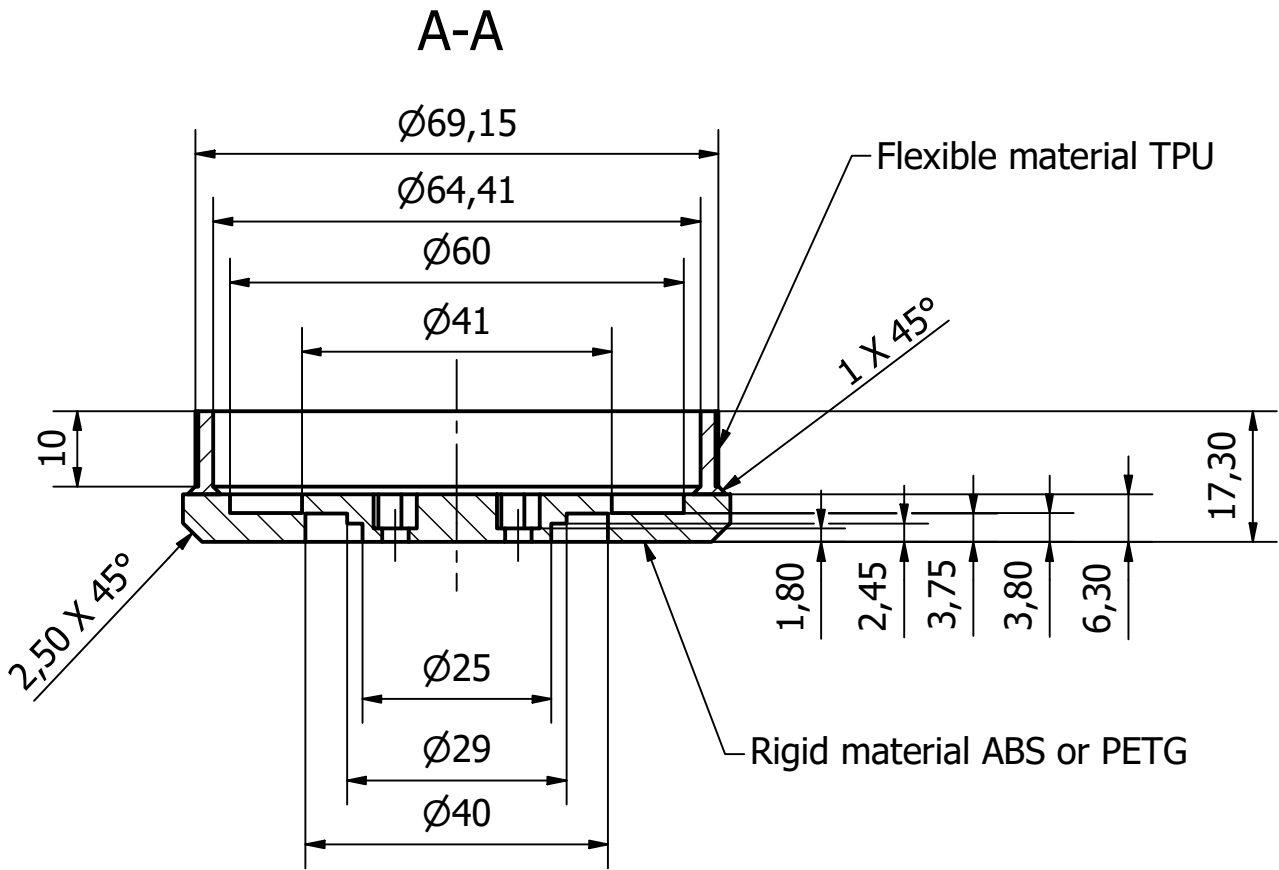
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863		Original design of the circular spline			
		HD circular spline v0	Edizione	Foglio	1 / 1




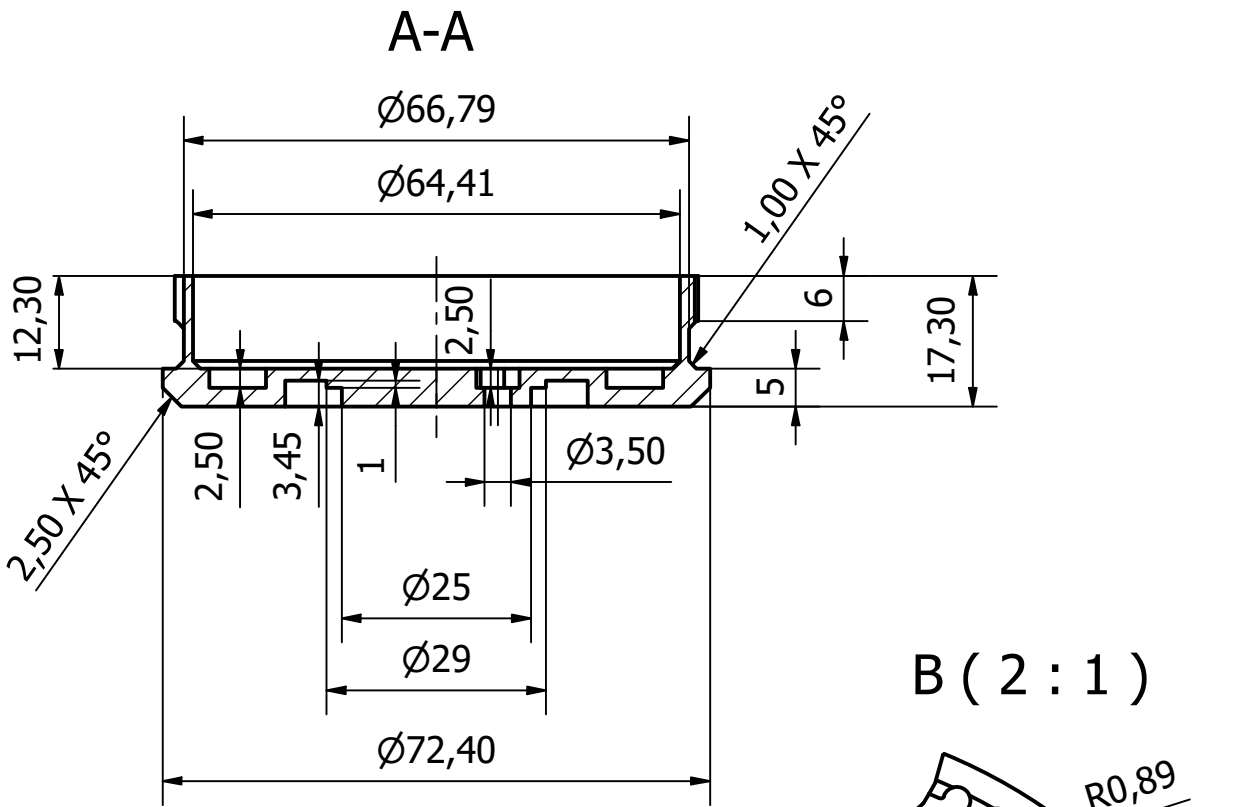
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data		Data	
 POLITECNICO MILANO 1863			Circular spline with HTD3M profile			
			HD circular spline v1		Edizione	Foglio 1 / 1



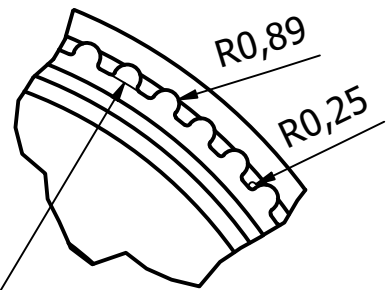
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863			Belt housing for original flexible spline		
			Flexible spline v0	Edizione	Foglio 1 / 1



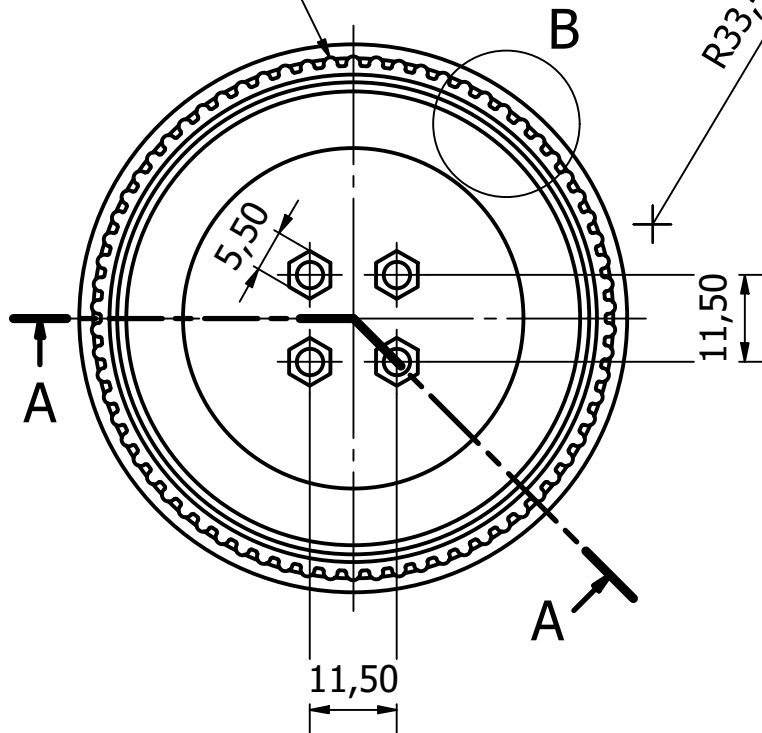
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863			3D printable flexible spline		
			Flexible spline v1		Edizione




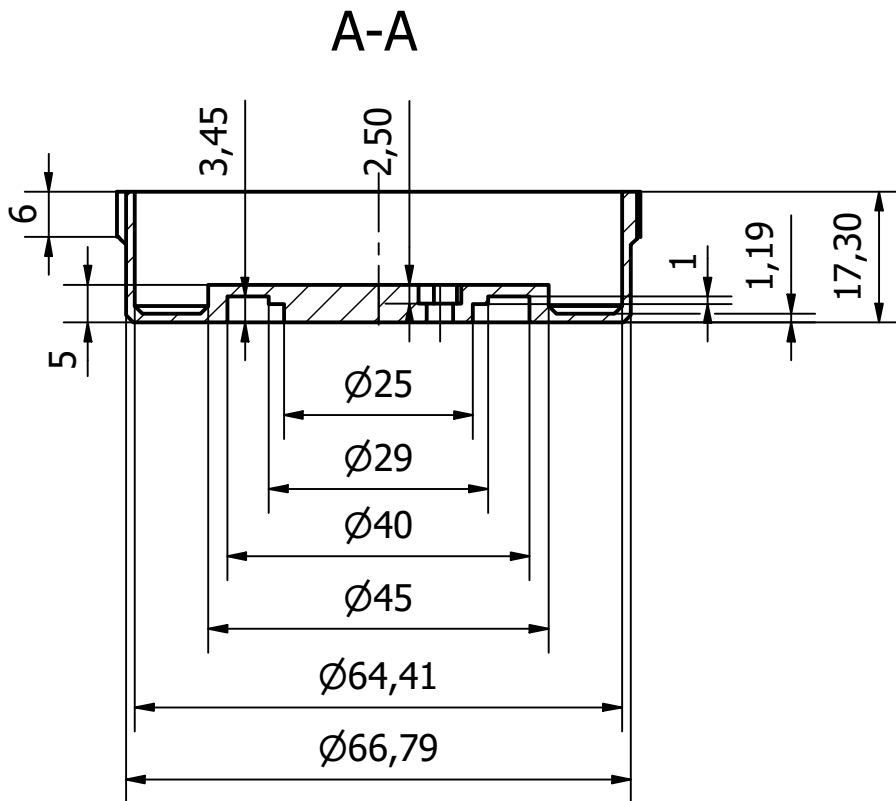
B (2 : 1)



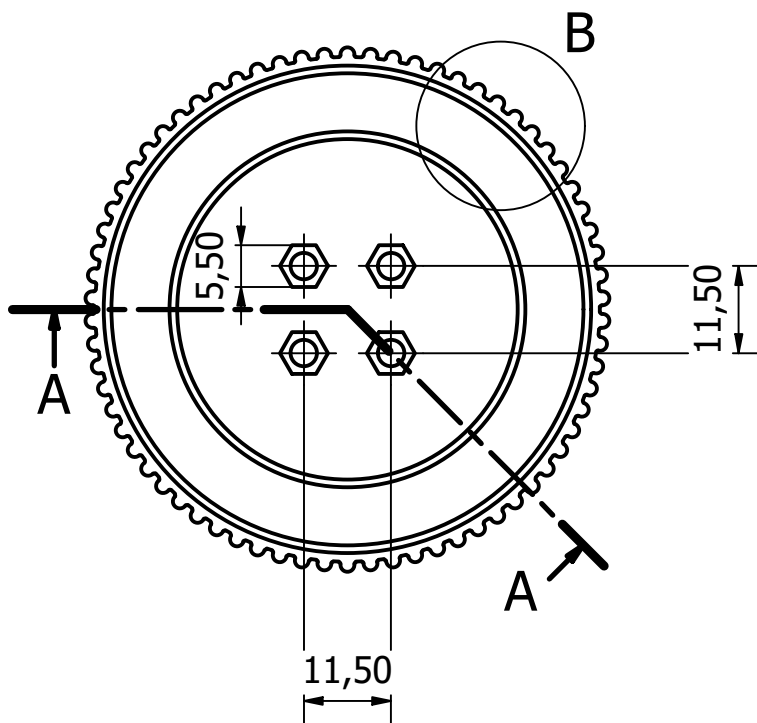
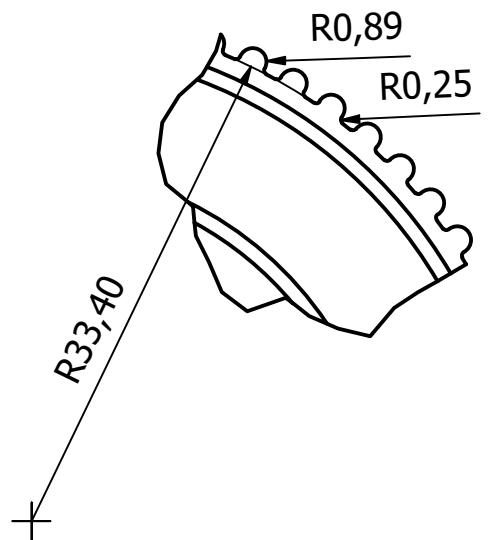
70 teeth HTD3M




Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863			Modified 3D printable flexible spline		
			Flexible spline v2		Edizione

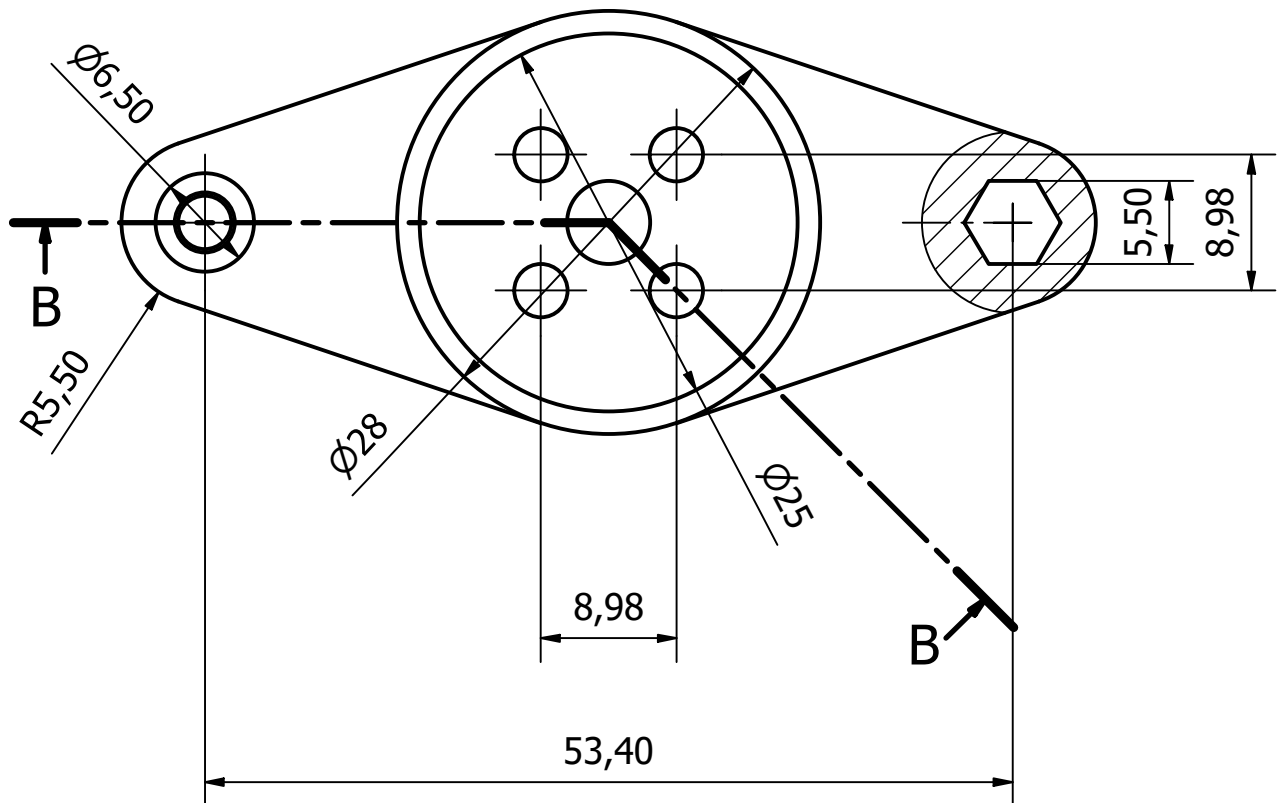
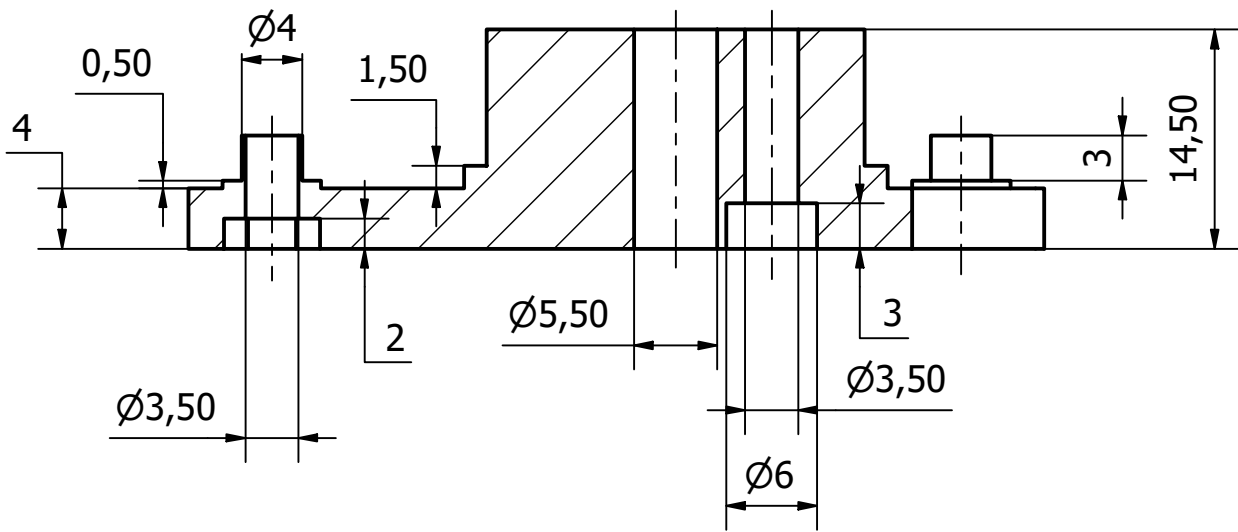



B (2 : 1)



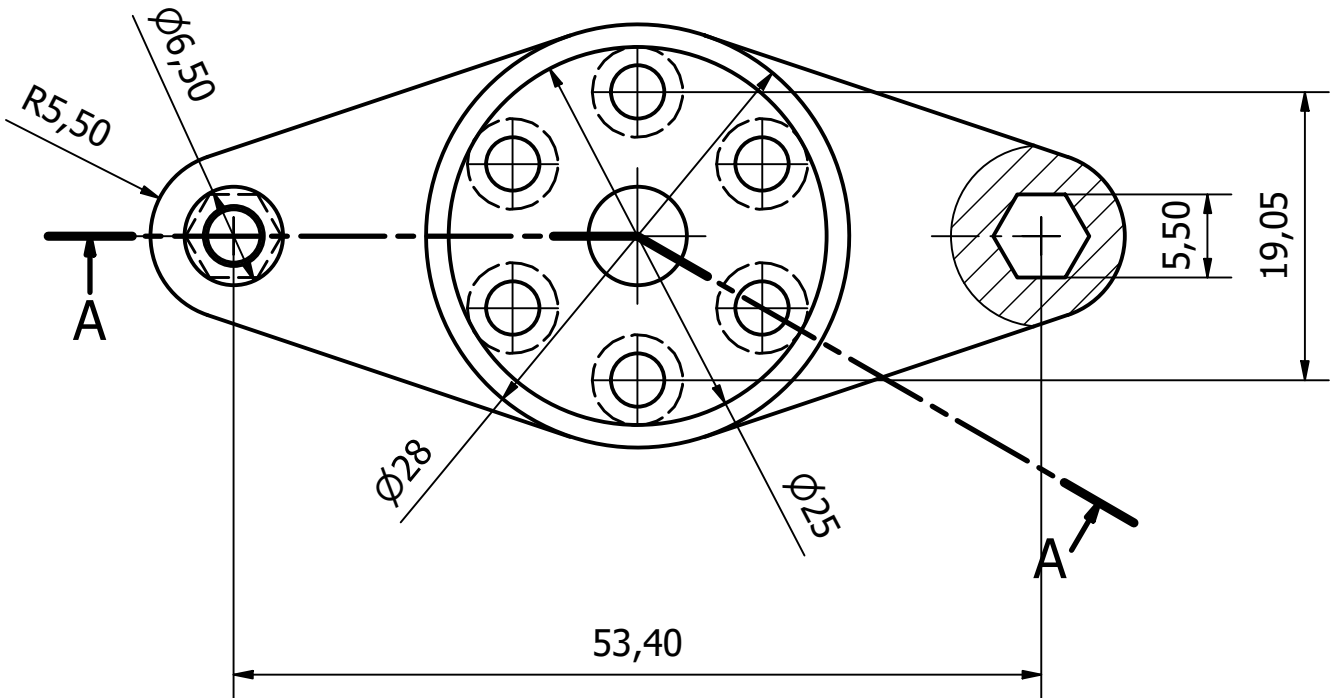
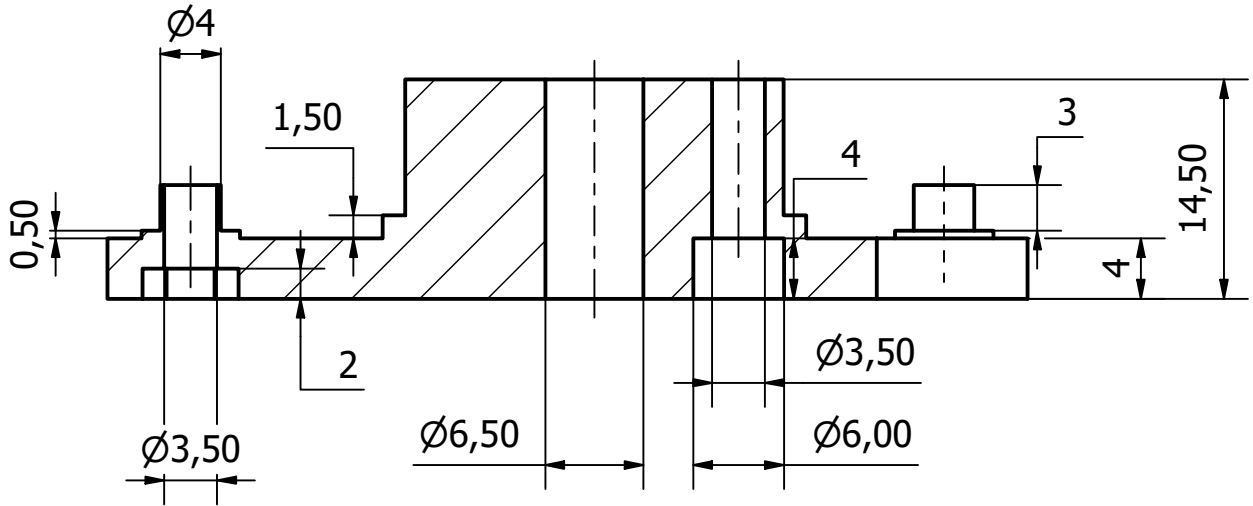
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863			Modified 3D printable flexible spline		
			Flexible spline v3		Edizione


B-B

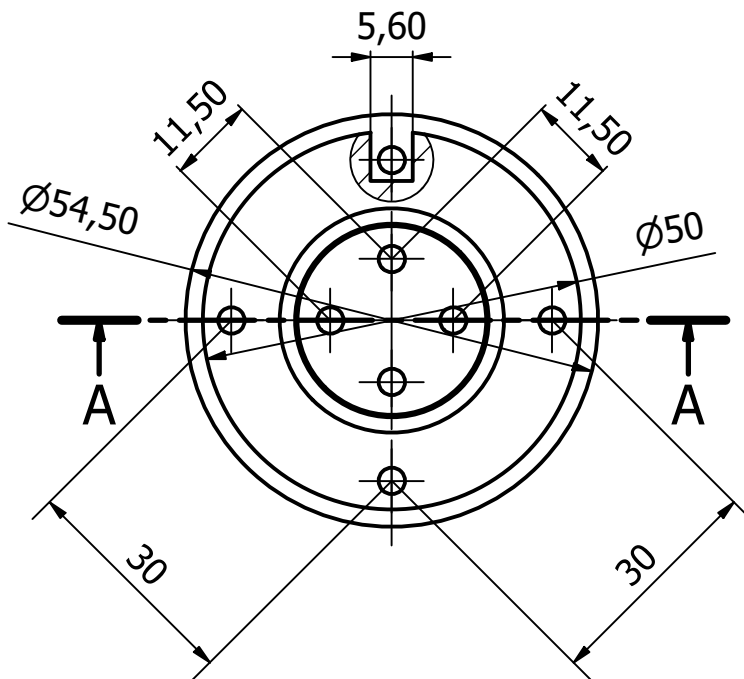
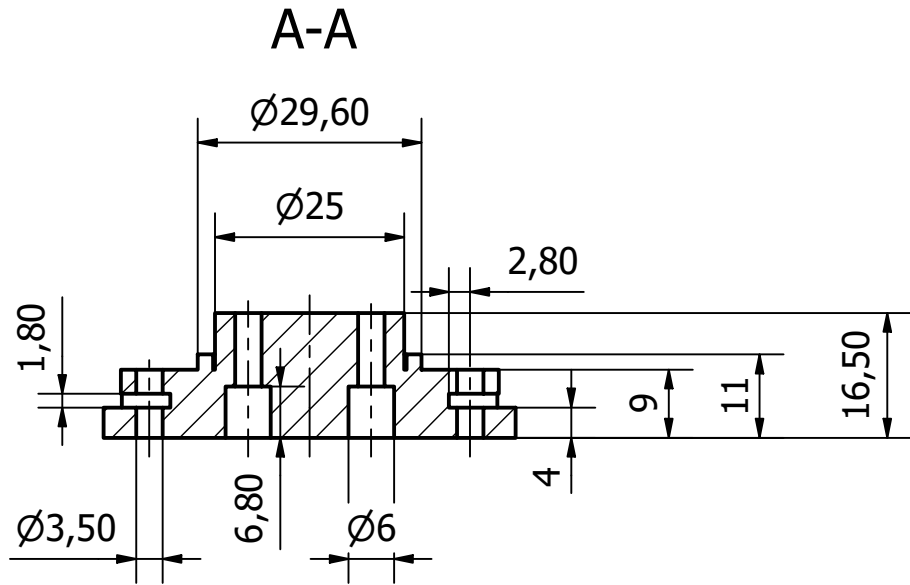



Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863		Wave generator for NEMA17 HD			
		NEMA17 wave generator	Edizione	Foglio	1 / 1

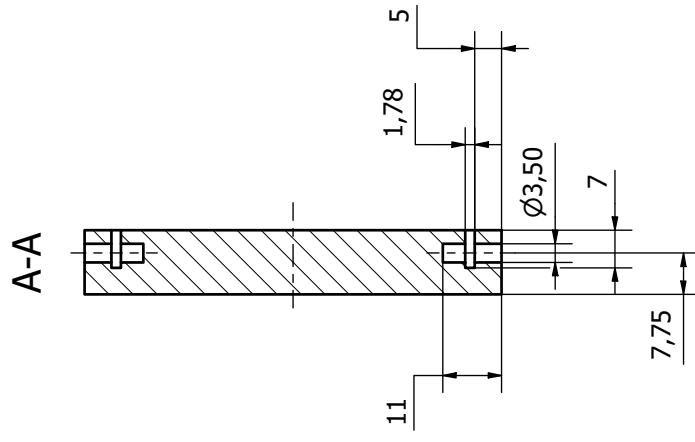
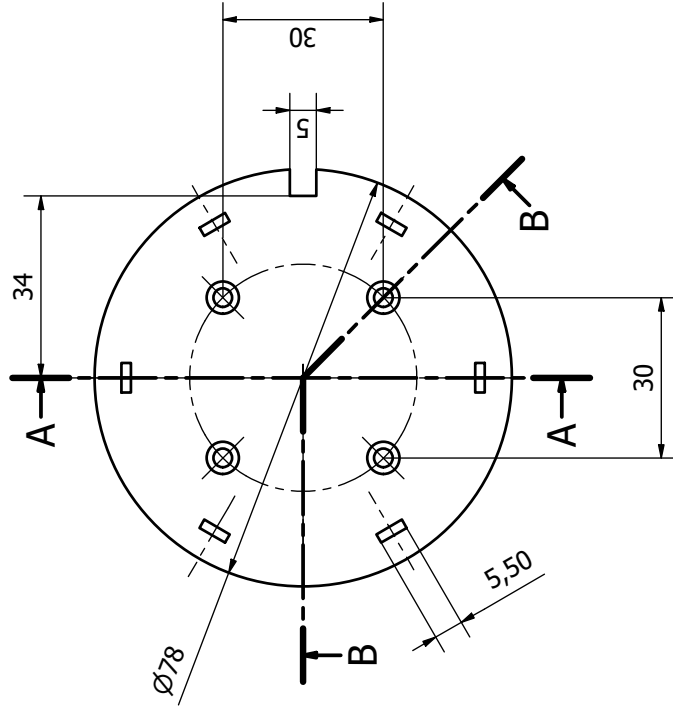
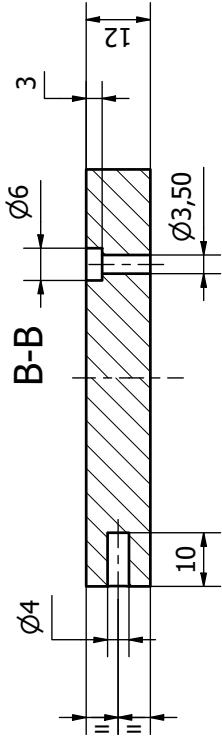
A-A




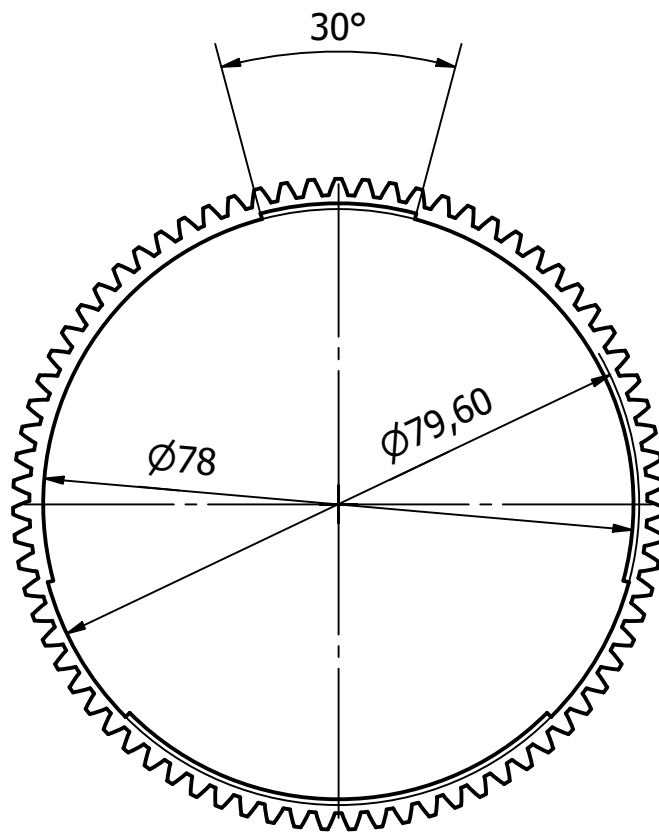
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863		Wave generator for NEMA23 HD			
		NEMA23 wave generator		Edizione	Foglio 1 / 1




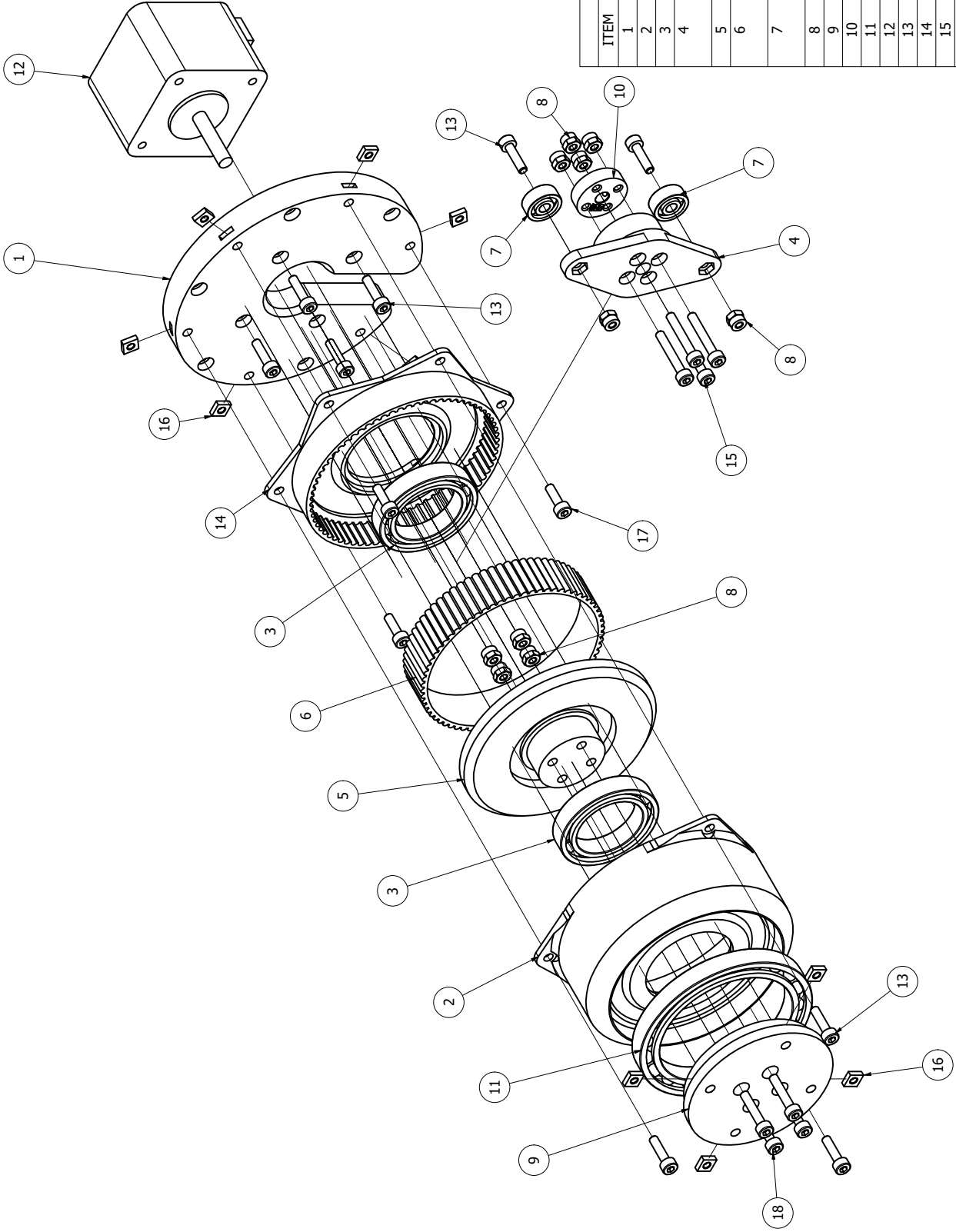
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	
 POLITECNICO MILANO 1863			Output flange for 3D printed flex spline		
			Output flange v1	Edizione	Foglio 1 / 1



Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data
 POLITECNICO MILANO 1863			Axial coupler for joints 1 and 4	
			Axial coupler	
			Edizione	Foglio
				1 / 1



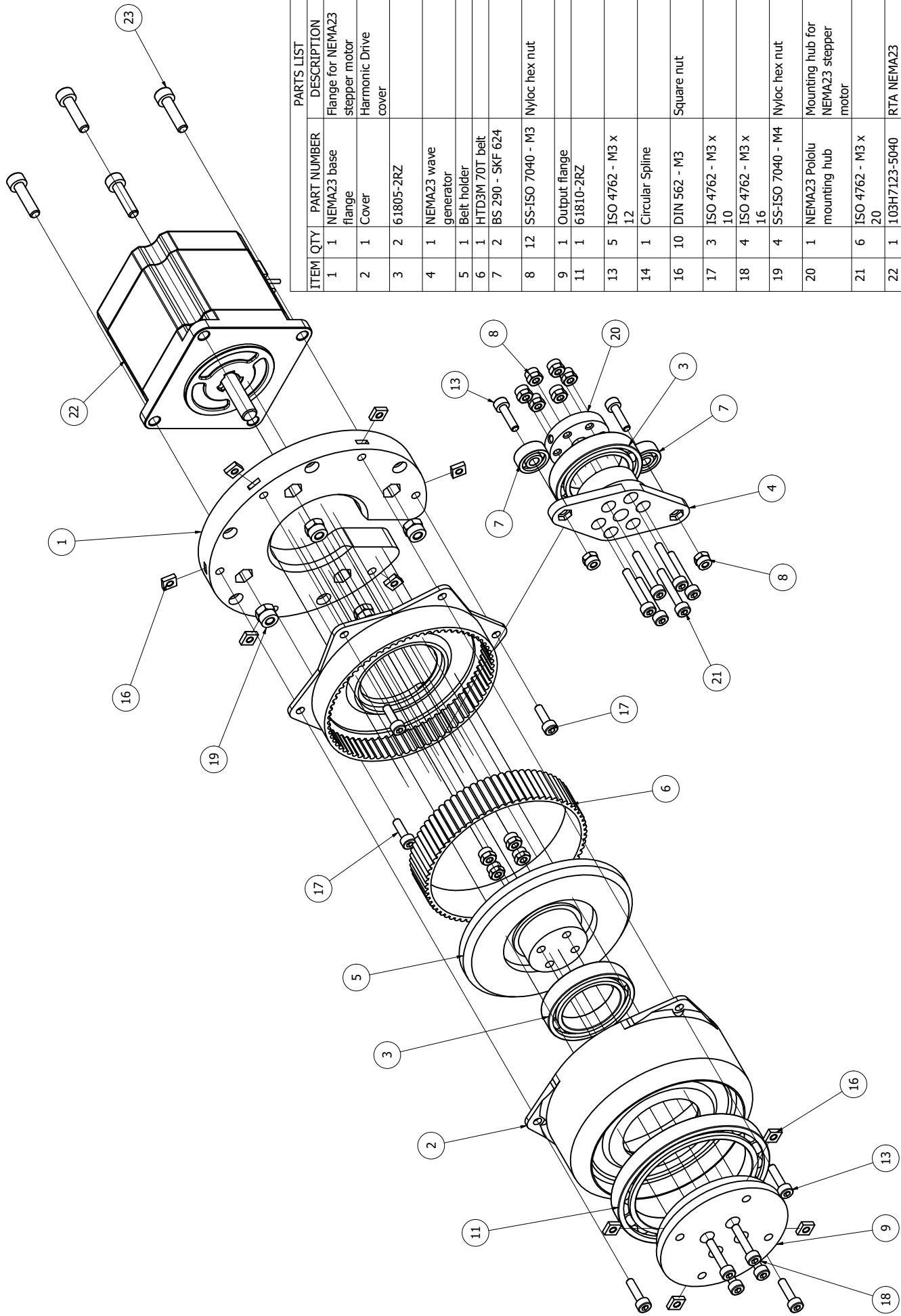
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data		Data	
 POLITECNICO MILANO 1863			Gear ring for non axial joints			
			Gear ring		Edizione	Foglio 1 / 1



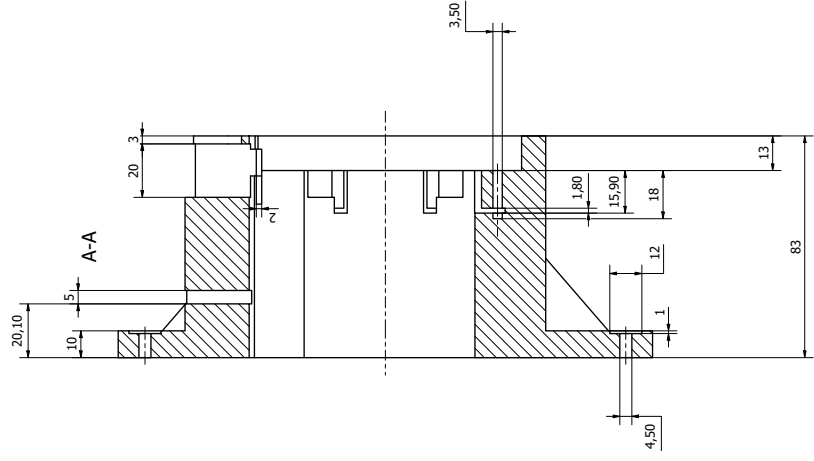
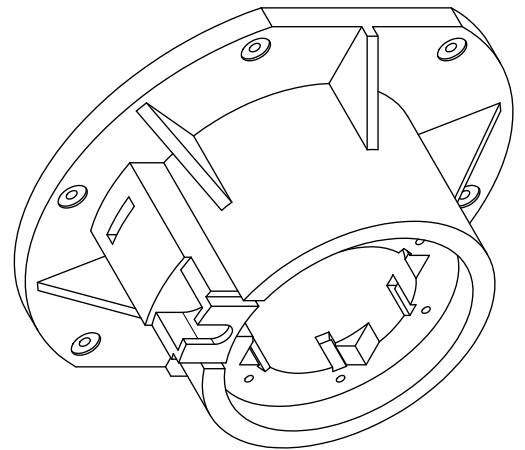
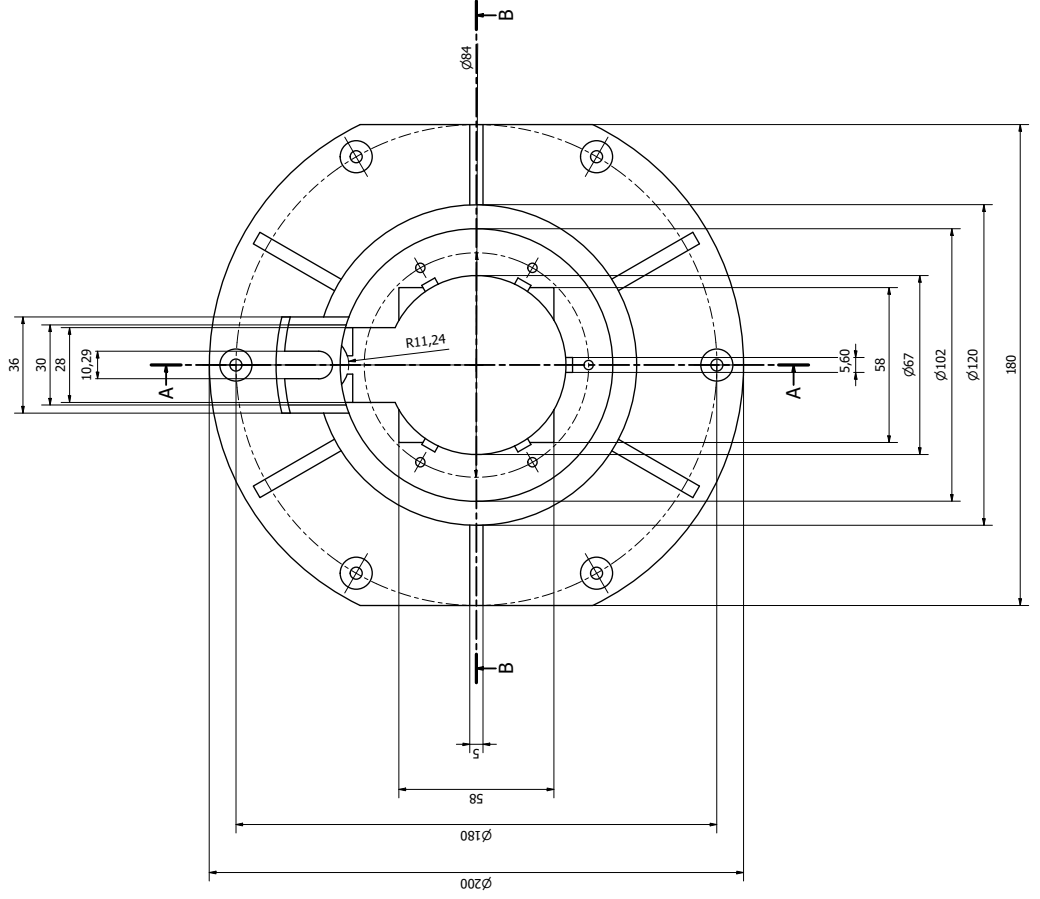
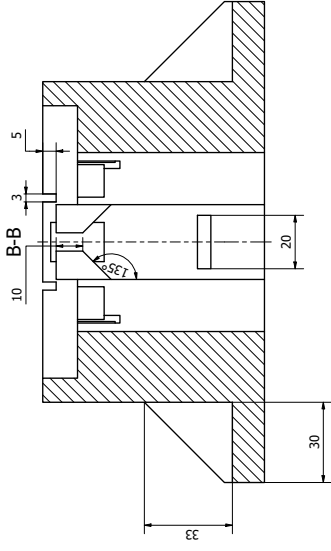
ITEM	QTY	PART NUMBER	DESCRIPTION	COMMENTS
1	1	NEMA17 base flange		3D printed
2	1	HD cover		3D printed
3	2	61805-2RZ	Ball bearing	Purchased
4	1	NEMA17 wave generator		3D printed
5	1	Flexible spline v0	Belt holder	3D printed
6	1	HTD3M synchronous belt	Rubber belt as flexible element	Purchased
7	2	624ZZ	Ball bearings for wave generator	Purchased
8	10	SS-ISO 7040 - M3	Hex nyloc nut	Purchased
9	1	Output flange v0		3D printed
10	1	Pololu mounting hub		Purchased
11	1	61810-2RZ	Ball bearing	Purchased
12	1	103H5205-4240	NEMA17 stepper motor	Purchased
13	9	ISO 4762 - M3 x 12	Hex screw	Purchased
14	1	Circular spline v0		3D printed
15	4	ISO 4762 - M3 x 20	Hex screw	Purchased
16	10	DIN 562 - M3	Square nut	Purchased
17	3	ISO 4762 - M3 x 10	Hex screw	Purchased
18	4	ISO 4762 - M3 x 16	Hex screw	Purchased

Progettato da: Lorenzo Ferrari
 Controllo da:
 Approvato da:
 Data:
 Data:

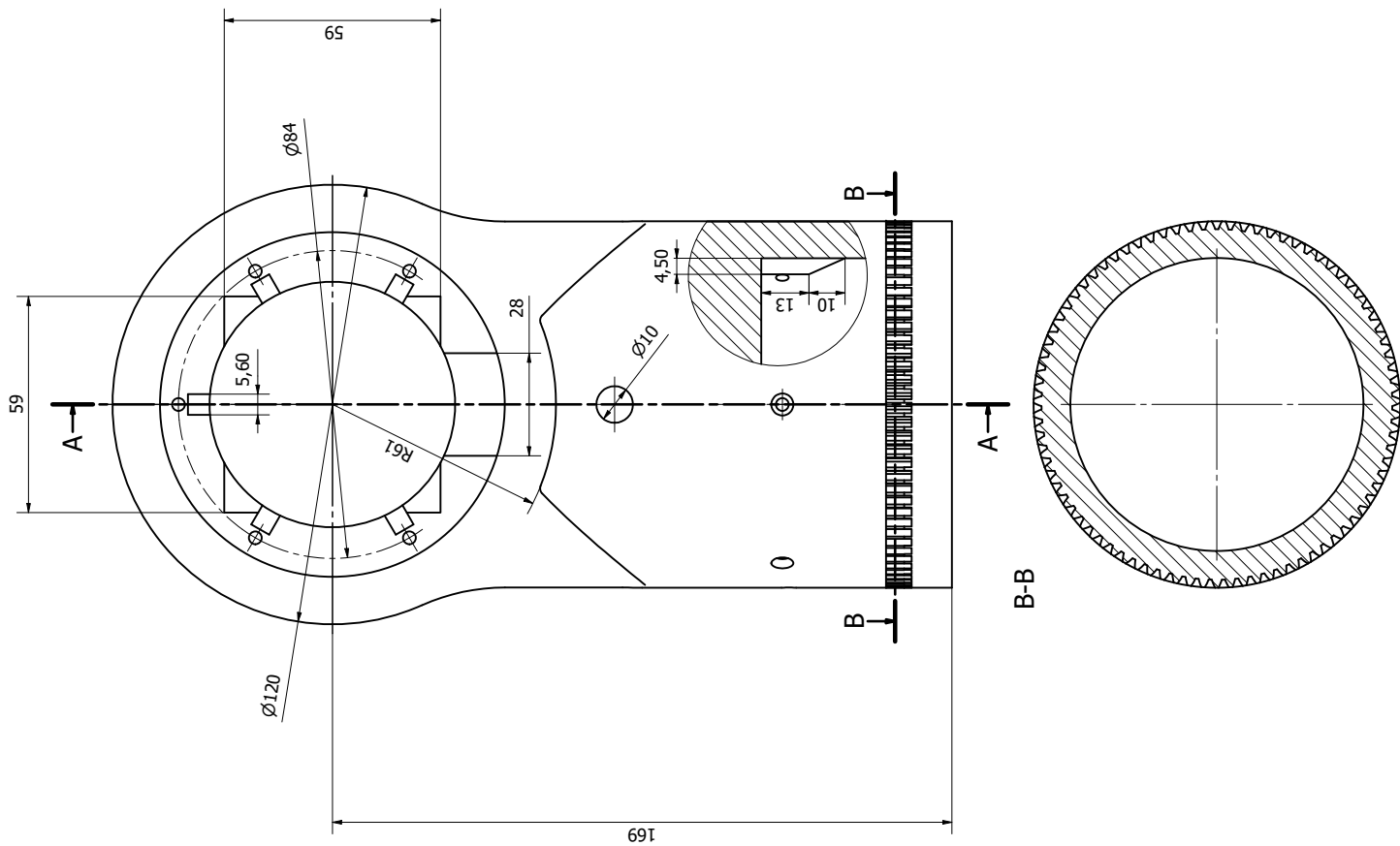
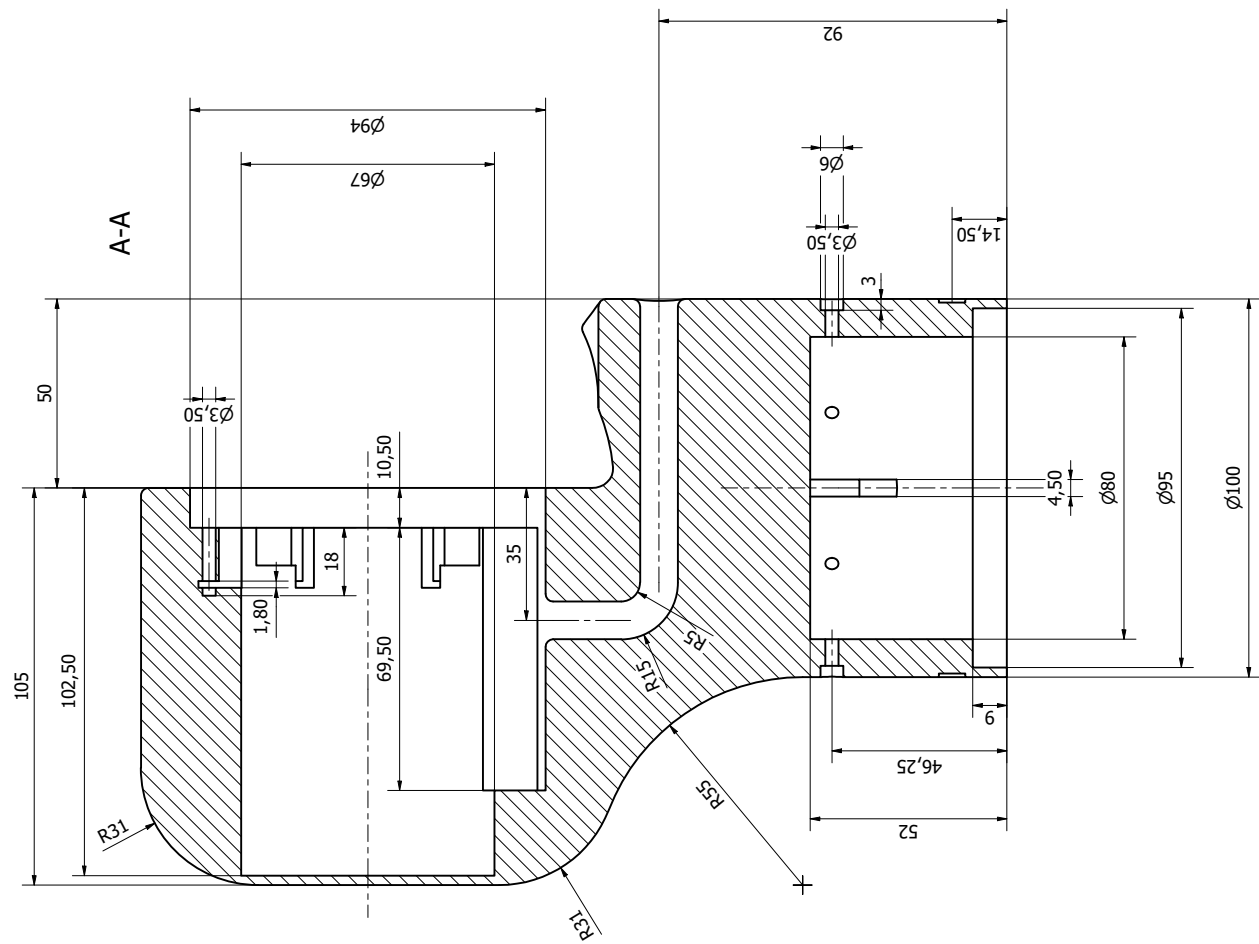
POLITECNICO MILANO 1863
 Exploded view of HD for NEMA17 motors
 Exploded HD NEMA17
 Foglio 1 / 1




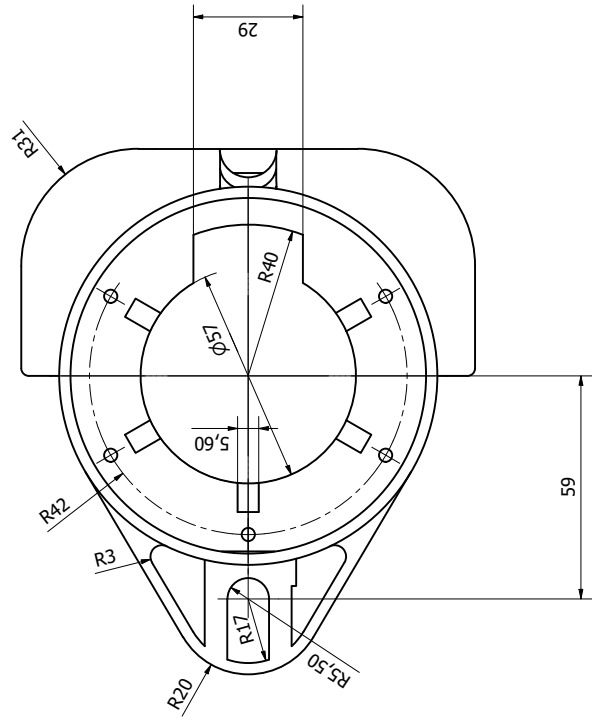
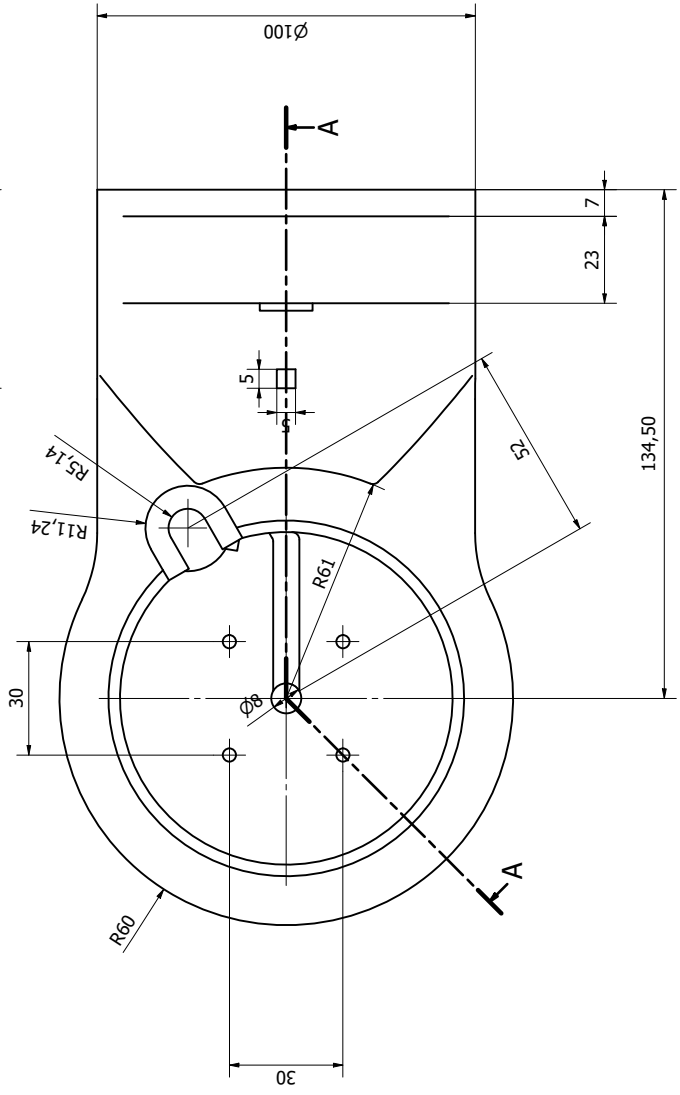
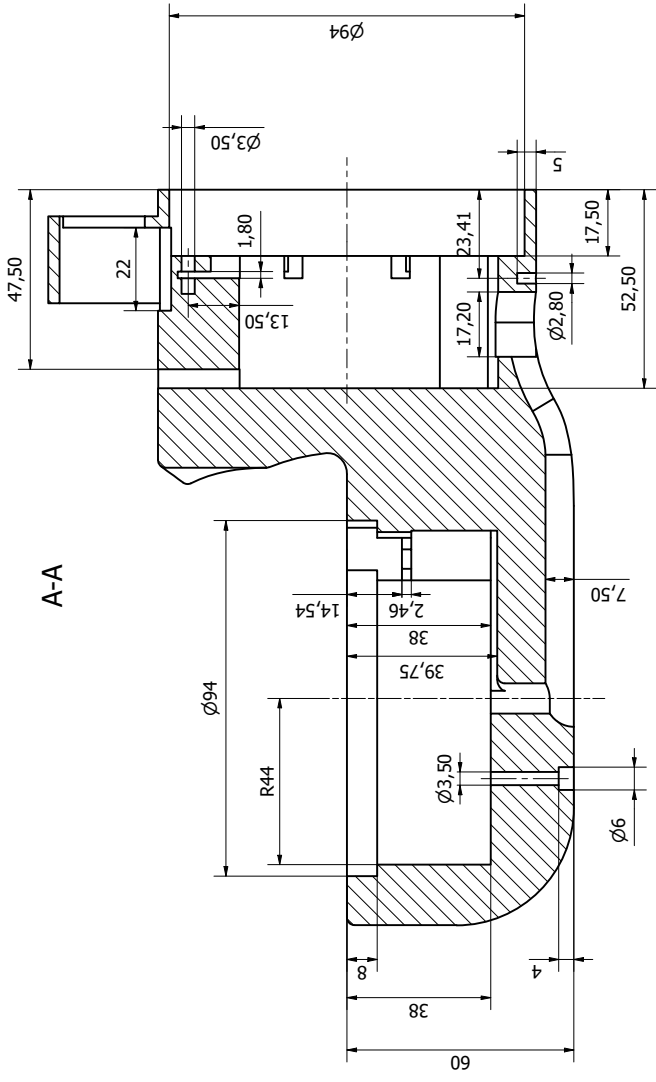
PARTS LIST					
ITEM	QTY	PART NUMBER	DESCRIPTION	MATERIAL	COMMENTS
1	1	NEMA23 base flange	Flange for NEMA23 stepper motor	ABS Plastic	3D printed
2	1	Cover	Harmonic Drive cover	ABS Plastic	3D printed
3	2	61805-2RZ		Acciaio, Dolce	Purchased
4	1	NEMA23 wave generator		ABS Plastic	3D printed
5	1	Belt holder		ABS Plastic	3D printed
6	1	HTD3M 70T belt		Rubber	Purchased
7	2	BS 290 - SKF 624			Purchased
8	12	SS-ISO 7040 - M3	Nyloc hex nut		Purchased
9	1	Output flange		ABS Plastic	3D printed
11	1	61810-2RZ			Purchased
13	5	ISO 4762 - M3 x 12			Purchased
14	1	Circular Spline		ABS Plastic	3D printed
16	10	DIN 562 - M3	Square nut		Purchased
17	3	ISO 4762 - M3 x 10			Purchased
18	4	ISO 4762 - M3 x 16			Purchased
19	4	SS-ISO 7040 - M4	Nyloc hex nut		Purchased
20	1	NEMA23 Pololu mounting hub	Mounting hub for NEMA23 stepper motor	Aluminum	Purchased
21	6	ISO 4762 - M3 x 20			Purchased
22	1	103H7123-5040	RTA NEMA23 stepper motor		Purchased
23	4	ISO 4762 - M4 x 16(1)			Purchased

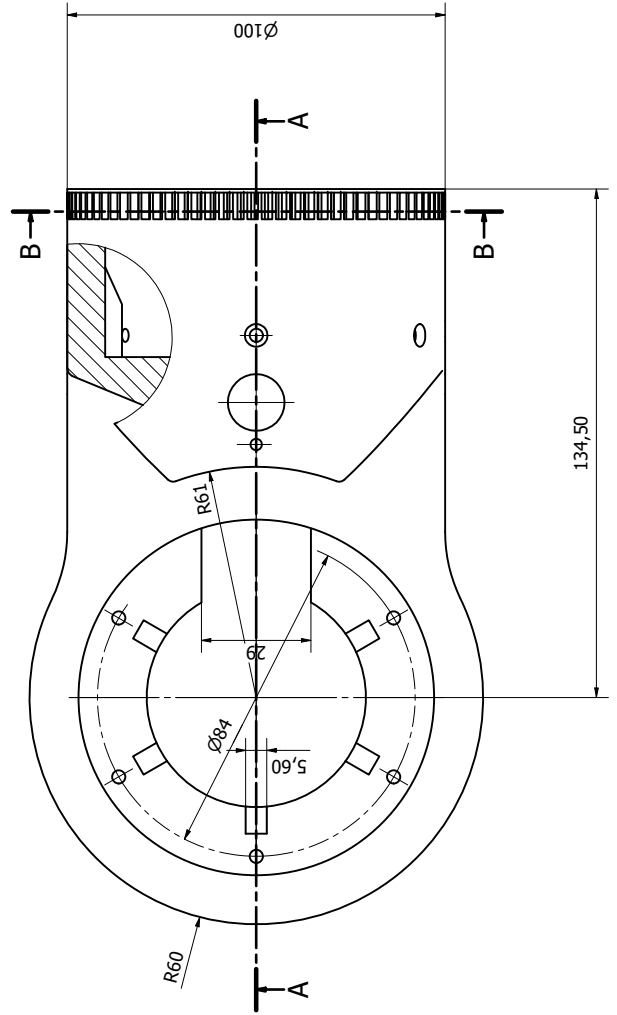
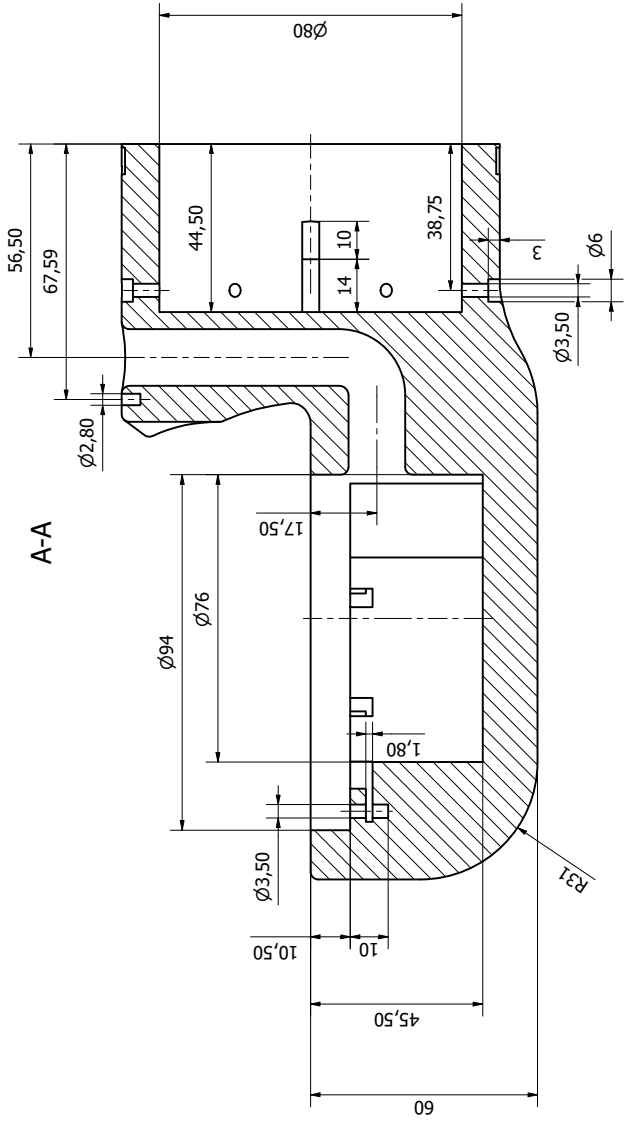


Professione Lorenzo Ferrati	Corso POLITECNICO MILANO 1863	Reparto Robot base	Data	Data	Data	Bozza 1 / 1
--------------------------------	-------------------------------------	-----------------------	------	------	------	----------------



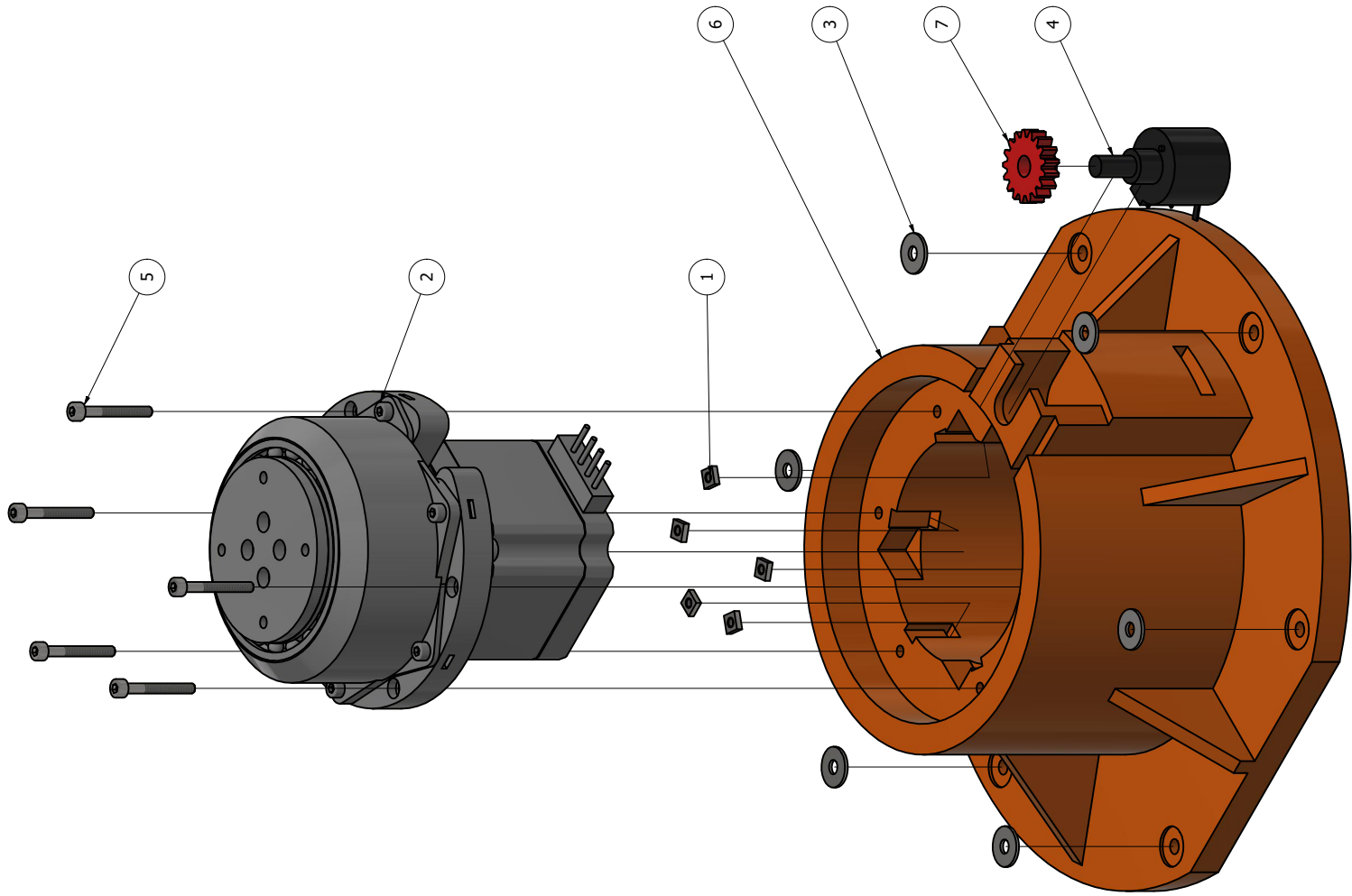
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	Link 0
 POLITECNICO MILANO 1863					Link 0
					Foglio 1 / 1



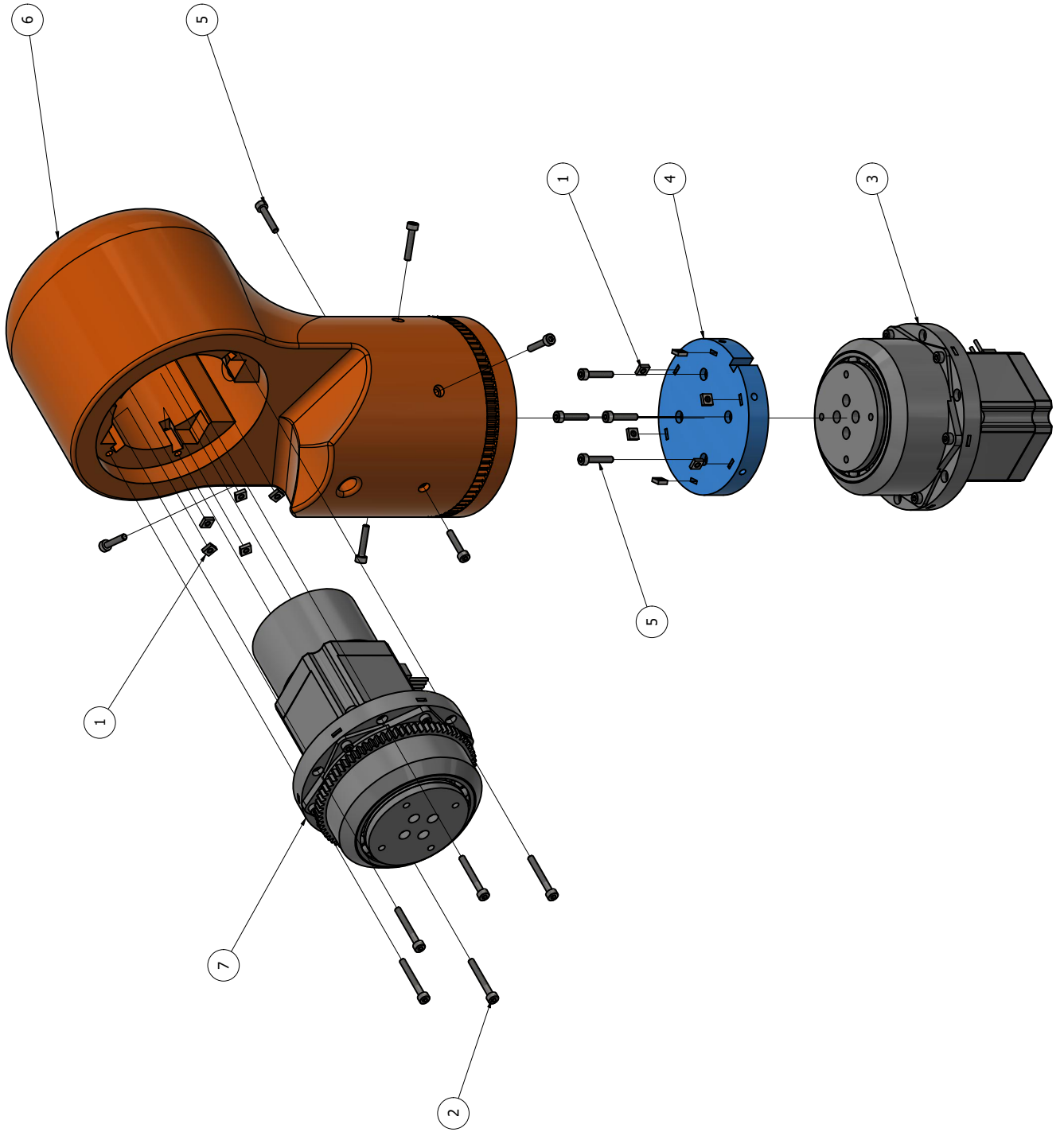


B-B

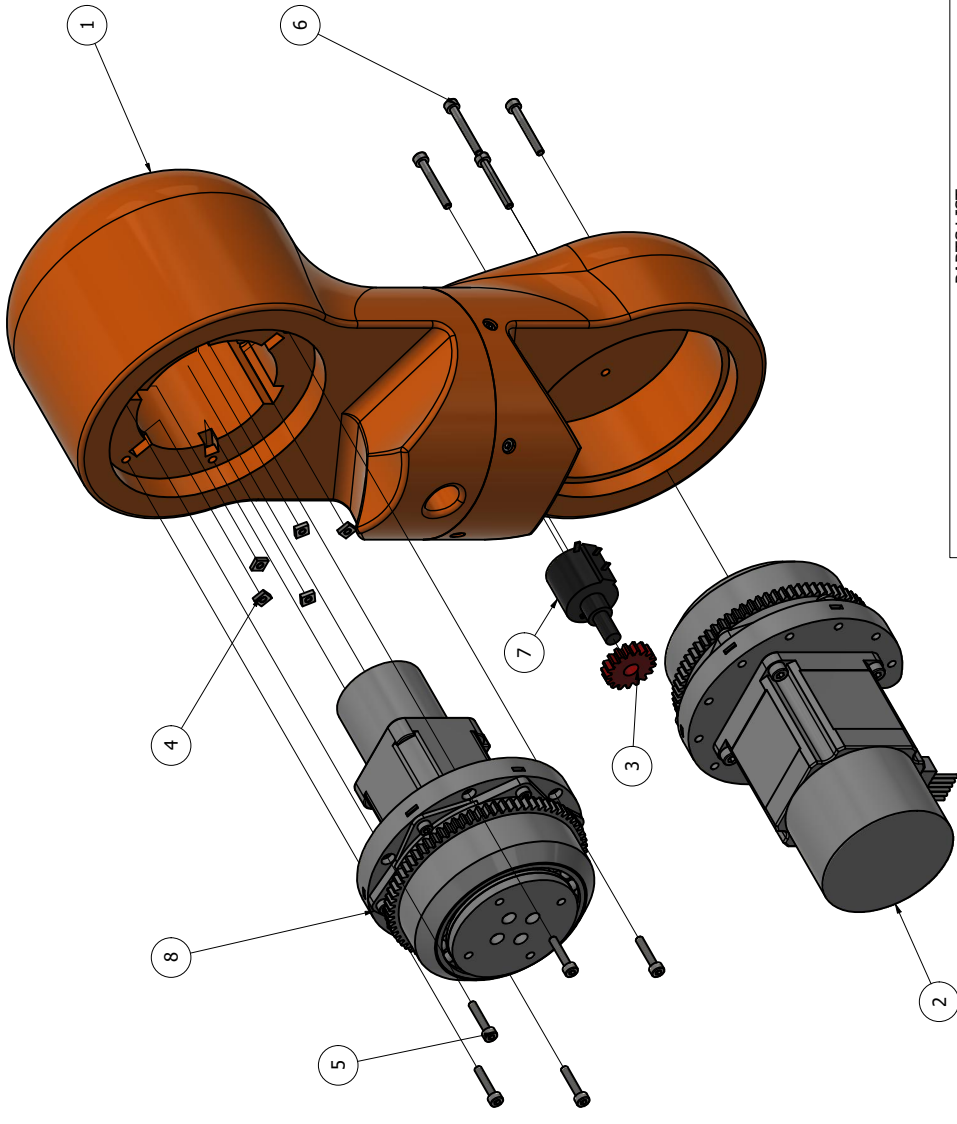
Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data	Data	Data
POLITECNICO MILANO 1863			Link 2 upper section		
			Link 2 up		
			Foglio 1 / 1		



PARTS LIST						
ITEM	QTY	PART NUMBER	DESCRIPTION	COMMENTS		
1	5	DIN 562 - M3	Square nut	Purchased		
2	1	HD J1	Assembled HD	3D printed		
3	6	ISO 7093 A - ST 4 - 140 HV	Washer	Purchased		
4	1	Bourms 3548	Potentiometer	Purchased		
5	5	ISO 4762 - M3 x 25	Potentiometer	Purchased		
6	1	Base		Purchased		
7	1	Type 1 gear	Gear for potentiometer shaft	3D printed		
Progettato da Lorenzo Ferrari		Controllato da	Approvato da	Data		
				Data		
				Explored view of Module 1		
				Module 1		
				POLITECNICO MILANO 1863		
				Edizione 1 / 1		
				Foglio 1 / 1		

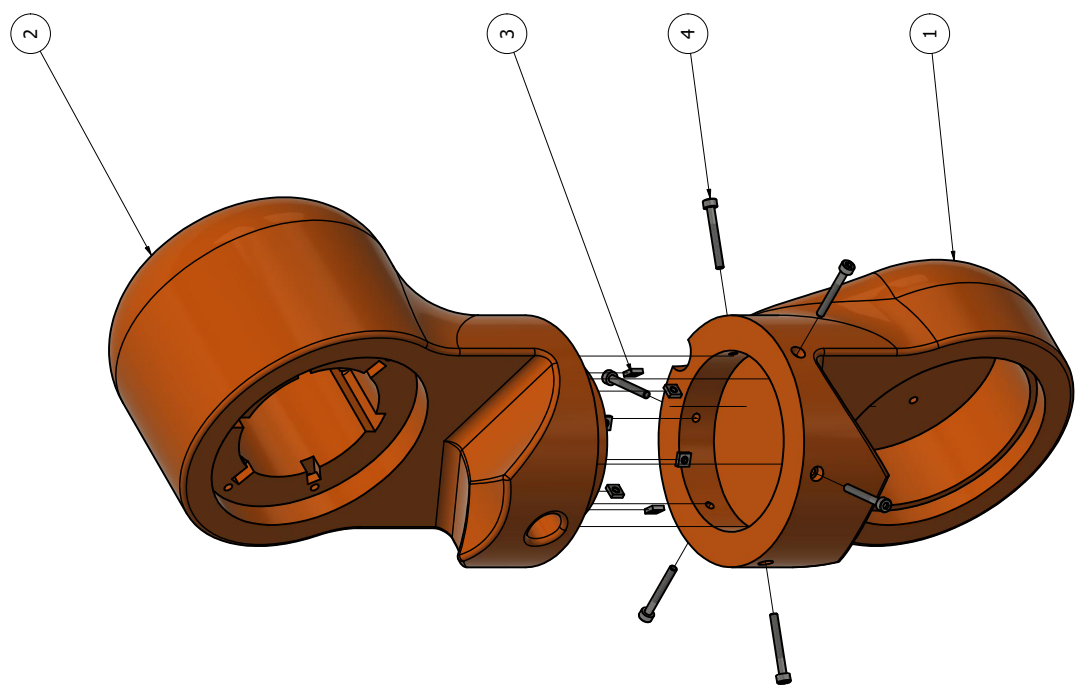


PARTS LIST						
ITEM	QTY	PART NUMBER	DESCRIPTION	COMMENTS		
1	11	DIN 562 - M3	Square nut	Purchased		
2	5	ISO 4762 - M3 x 25		Purchased		
3	1	HD J1	Assembled J1 HD (already assembled on previous module)	3D printed		
4	1	Coupler		3D printed		
5	10	ISO 4762 - M3 x 16		Purchased		
6	1	Link 0		3D printed		
7	1	HD J2	Assembled J2 HD	3D printed		
Progettato da Lorenzo Ferrari		Approvato da		Data		
				Data		
				Exploded view of Module 2		
				Module 2		
				POLITECNICO MILANO 1863		Foglio 1 / 1

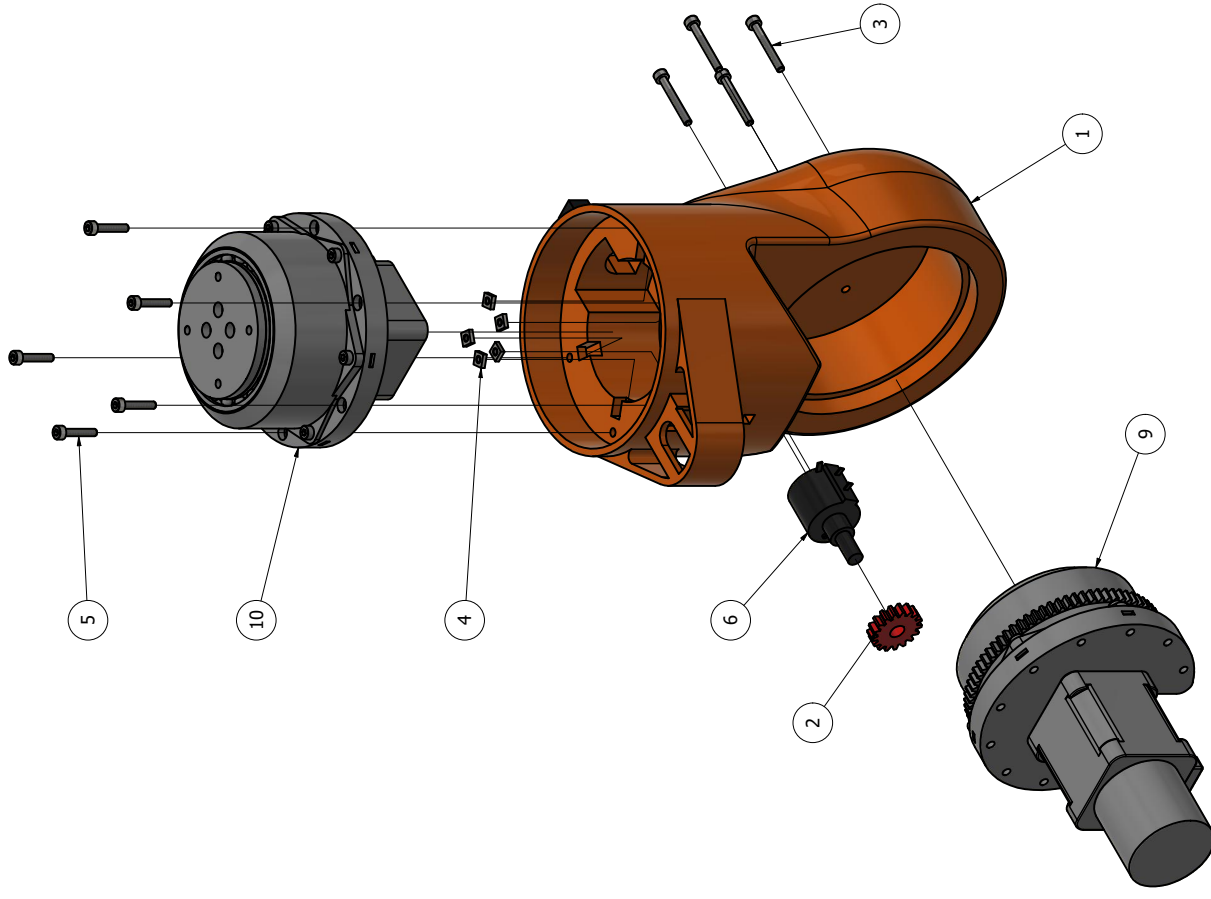


PARTS LIST				COMMENTS
ITEM	QTY	PART NUMBER	DESCRIPTION	
1	1	Link 1	Assembled Link 1	3D printed
2	1	HD J2	Assembled J2 HD (already assembled on previous Module)	3D printed
3	1	Type 2 gear	Gear for potentiometer shaft	3D printed
4	5	DIN 562 - M3	Square nut	Purchased
5	11	ISO 4762 - M3 x 16	Hex screw	Purchased
6	4	ISO 4762 - M3 x 25	Hex screw	Purchased
7	1	Bourns 3548	Potentiometer	Purchased
8	1	HD J3	Assembles J3 HD	3D printed

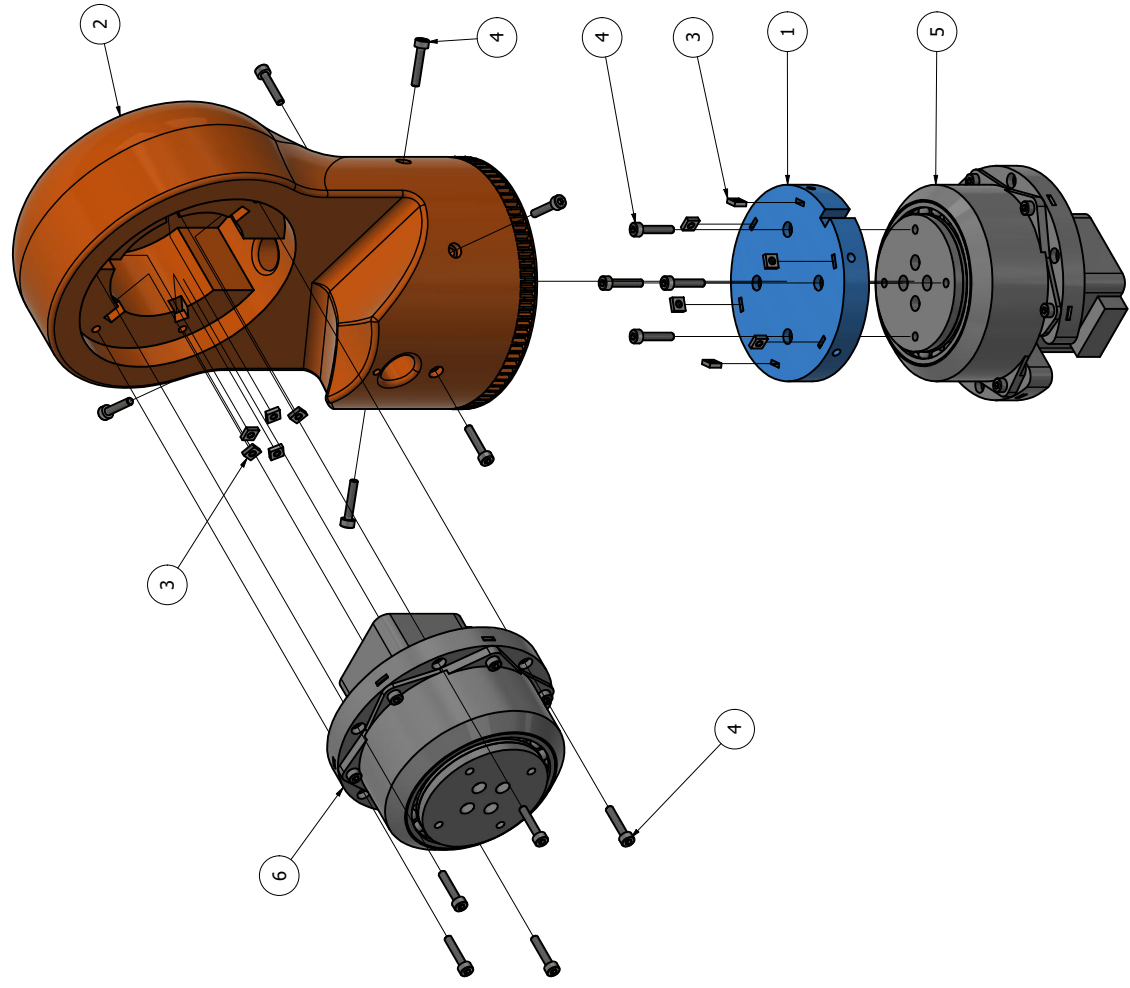
Progettato da: Lorenzo Ferrari
 Controllato da: _____
 Approvato da: _____
 Data: _____



PARTS LIST				COMMENTS
ITEM	QTY	PART NUMBER	DESCRIPTION	
1	1	Link 1 lower part		3D printed
2	1	Link 1 upper part		3D printed
3	6	DIN 562 - M3	Square nut	Purchased
4	6	ISO 4762 - M3 x 25		Purchased



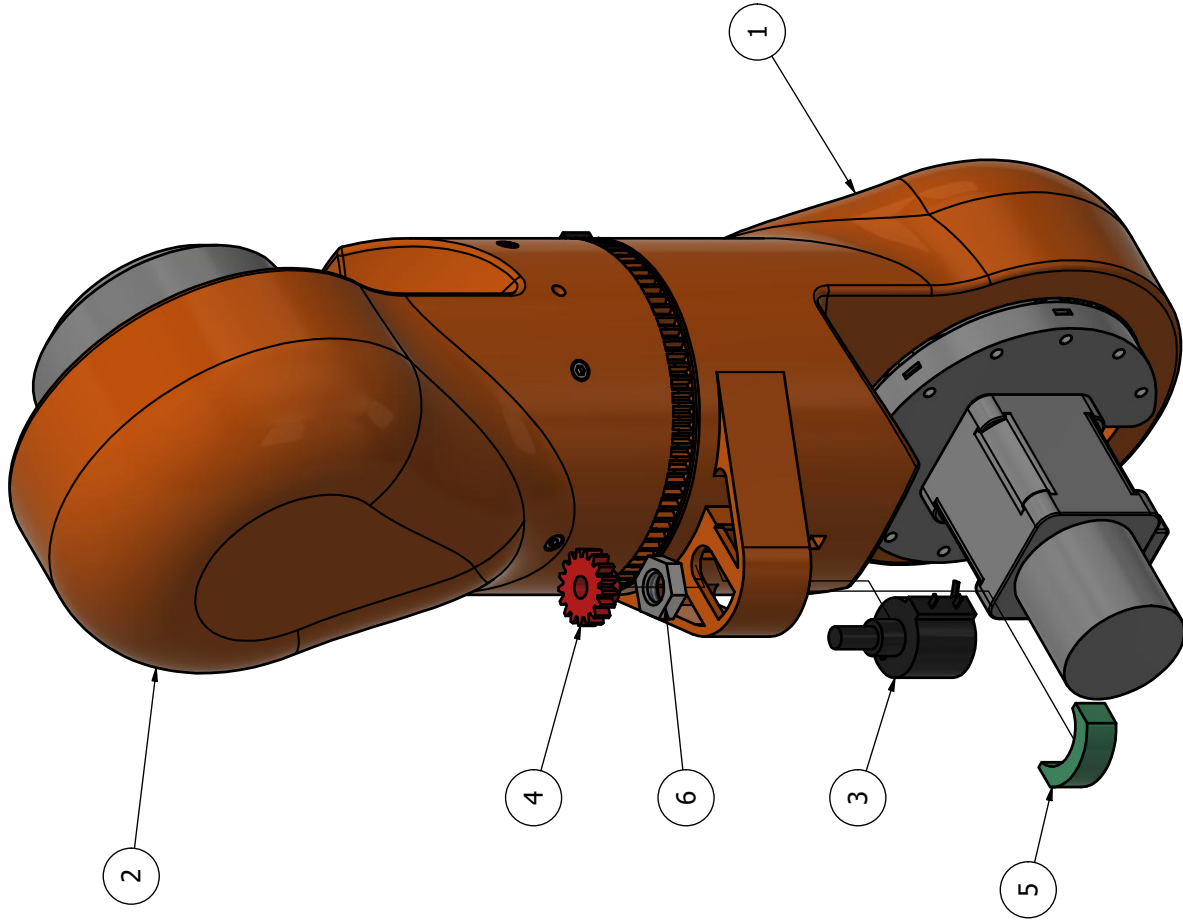
PARTS LIST			COMMENTS
ITEM	QTY	PART NUMBER	DESCRIPTION
1	1	Link 2 lower part	3D printed
2	1	Type 2 gear	Gear for potentiometer shaft
3	4	ISO 4762 - M3 x 25	Hex screw
4	5	DIN 562 - M3	Square nut
5	5	ISO 4762 - M3 x 16	Hex screw
6	1	Bourns 3548	Potentiometer
7	1	Support for wires	Strain relief support for wires
8	1	ISO 4762 - M3 x 20	Hex screw
9	1	HD J3	Assembled J3 HD (already mounted on previous module)
10	1	HD J4	Assembled J4 HD




PARTS LIST			COMMENTS
ITEM	QTY	PART NUMBER	DESCRIPTION
1	1	Axial coupler	3D printed
2	1	Link2 upper part	3D printed
3	11	DIN 562 - M3	Square nut
4	15	ISO 4762 - M3 x 16	Hex screw
5	1	HD J4	Assembled J4 HD (already mounted on link 2 lower part)
6	1	HD J5	Assembled J5 HD

Progettato da Lorenzo Ferrari	Controllato da	Approvato da	Data
----------------------------------	----------------	--------------	------

POLITECNICO MILANO 1863		Explored view of Module 4	
Module 4		Edizione 1 / 1	



PARTS LIST					
ITEM	QTY	PART NUMBER	DESCRIPTION	COMMENTS	
1	1	Link 2 bottom part		3D printed	
2	1	Link 2 upper part		3D printed	
3	1	Bourns 3548	Potentiometer	Purchased	
4	1	Type 3 gear		3D printed	
5	1	Alignment block	Tool for alignment of potentiometer	3D printed	
6	1	Hex nut	Hex nut provided with the potentiometer	Purchased	
Progettato da Lorenzo Ferrari		Controllato da		Data	Data
		Approvato da		Data	Data
 POLITECNICO MILANO 1863			Assembly procedure for J4 potentiometer		
			J4 potentiometer assembly		