RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

End-of-study project dissertation for obtaining

the State Engineer's degree in Electronics

Development of an autonomous 6DOF robotic arm using machine learning

BELKHIRI Djamel Ibrahim & BENARAB Adel

Under the supervision of **Pr. LARBES Cherif** ENP

Presented and defended on June 24th, 2025 in front of the members of jury:

President:	Mr. TAGHI Mohamed	ENP
Supervisor:	Pr. LARBES Cherif	ENP
Examiner:	Dr. BOUADJENEK Nesrine	ENP

ENP 2025

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

End-of-study project dissertation for obtaining

the State Engineer's degree in Electronics

Development of an autonomous 6DOF robotic arm using machine learning

BELKHIRI Djamel Ibrahim & BENARAB Adel

Under the supervision of **Pr. LARBES Cherif** ENP

Presented and defended on June 24th, 2025 in front of the members of jury:

President:	Mr. TAGHI Mohamed	ENP
Supervisor:	Pr. LARBES Cherif	ENP
Examiner:	Dr. BOUADJENEK Nesrine	ENP

ENP 2025

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

Mémoire de projet de fin d'études

Pour l'obtention du diplôme d'Ingénieur d'État en Électronique

Développement d'un bras autonome a 6DDL a l'aide du machine learning

BELKHIRI Djamel Ibrahim & BENARAB Adel

Sous la direction de **Pr. CHERIF Larbes** ENP

Présenté et soutenu publiquement le 24 Juin, 2025 auprès des membres du jury :

Président:	Mr. TAGHI Mohamed	ENP
Promoteur:	Pr. LARBES Cherif	ENP
Examinatrice:	Dr. BOUADJENEK Nesrine	ENP

2025

ملخص

يقدّم هذا البحث تطوير ذراع روبوتية تعاونية مستقلة بست درجات حرية، ويشمل التصميم الميكانيكي، والتحليل الحركي والديناميكي، والطباعة ثلاثية الأبعاد، والتكامل الكامل للنظام الكهربائي. يتم التحكم في الروبوت باستخدام روبوديكا و غوص من أجل المحاكاة وتخطيط الحركة. تم تحقيق الاستقلالية من خلال التعلم الخاضع للإشراف، والتعلم المعزز، وتقدير الوضعية باستخدام علامات اروكو. يُظهر النظام النهائي ذراعًا روبوتية مستقلة ومرنة وقوية، قادرة على أداء المهام الصناعية.

الكلمات المفتاحية : ست درجات حرية، الذراع الروبوتية، التصميم الميكانيكي، الاستقلالية، التعلم الخاضع للإشراف، التعلم المعزز، اروكو، غوص، روبوديكا

Résumé

Ce mémoire présente le développement d'un bras robotique collaboratif autonome à 6 degrés de liberté, incluant la conception mécanique, l'analyse cinématique et dynamique, l'impression 3D, ainsi que l'intégration complète du système électrique. Le robot est contrôlé via ROS et Robodk pour la simulation et la planification des mouvements. L'autonomie est obtenue grâce à l'apprentissage supervisé, à l'apprentissage par renforcement, et à l'estimation de pose basée sur les marqueurs ArUco. Le système final démontre un bras robotique autonome, robuste et flexible, capable d'exécuter des tâches industrielles.

Mots-clés : 6-DOF, bras robotique, conception mécanique, autonomie, apprentissage supervisé, apprentissage par renforcement, ArUco, ROS, Robodk.

Abstract

This thesis presents the development of a 6-DOF autonomous collaborative robotic arm, covering mechanical design, kinematic & dynamic analysis, 3D printing, and full electrical system integration. The robot is controlled via ROS and Robodk for simulation and motion planning. Autonomy is achieved through supervised learning, reinforcement learning, and ArUco marker-based pose estimation. The final system demonstrates a robust and flexible autonomous robotic arm capable of performing industrial tasks.

Keywords: 6-DOF, robotic arm, mechanical design, autonomy, supervised, reinforcement, ArUco, ROS, Robodk.

Dedication

I would like to offer this work for my family, especially my mother who truly supported me a lot during this work. She knew how to raise me well, she taught me how to be respectful and endure all the work stress. She helped me a lot, she suffered for us and pushed us to our limits. Thanks to her, I am writing down this acknowledgment to express my great gratitude to her.

I would love to express my great gratitude to my father, who truly suffered with me during this work. Even through all the work stress and problems, he always tried to help me find a solution, has always been patient with me, supported every idea I had, and did his best to make this work easier for me.

I want to specially mention my brother ***Hichem****. He was very patient with me, he never complained about picking me up at late hours, sacrificing sometimes his own studies so I could have time to work on my project. Even though I don't express my gratitude to you all the time, brother, you have been such a great older brother to me.*

I want to express my special gratitude to my older sister, who was like a second mother to me. I'll always be thankful for your help and support. Even though you moved far away, you always checked up on me and called to make sure that the work was going well. And I'll never forget my two little sisters the lovely ones.

I want to express my great, great, and great gratitude to my partner ***BELKHIRI Djamel Ibrahim****. It was such a great pleasure to work by your side and I truly hope we can finish this work together in the future. You were very patient and very supportive during the work. Honestly, you're the best partner I've had the chance to work with. You have been, and always will be, like a brother to me.*

I want to thank and express my huge gratitude to my great and best friend one of the greatest people I've met in my life ***DJERAOUI Houda****. You were such a great supporter of mine during this work, the owner of the arm named BRAS-DEL. You are one of the most respectful people I've seen in my life and I would love to express my gratitude to you.*

To my dearest friends ***MALEK Mohamed Sidali, Cheroufhi Mohamed Abdelhadi, LABBAS Mohammed Amine & BOUZNAD Tarek****, I want to say thank you for your support and for the EUROBOT experience we shared. You are like family to me.*

A special thanks to ***Rahal Mohamed Nadjib****, who was like an idol to me when I entered the electronics department. He was the reason I became passionate about robotics. He taught me countless things in this field, and I would have loved for you to be present during my presentation.*

To ***MESKALI Mohamed Ali & SAADANE Rched****, the most intelligent people I've met during my studies. I wish you both all the success in the future.*

Finally, to the members of ***TALEB BOT*** *and all my classmates thank you for the time we spent together and the great experiences we went through together.*

BENARAB Adel

Dedication

Starting my acknowledgments with a special person,a person that represents my first half, a person that made me the man that i am today, a person that after my dad passed away played two roles in parenting a loving mother and a caring father. ***My precious Mom*** *they say behind every great man a great women, for me its a great mother or let me say the greatest mother. Remembering the first day at school(crying my heart out for you) and now the last day at school probably the same thing. Thank you for everything & may Allah bless you.*

The second person is my other half, an example that i chase in life in order to be a better person, a person that keeps me going forward always behind me always supporting me, it's my beloved brother ***Omar Abdelaziz****, despite of 3 years age gap, but i always felt like having a father more than a brother, with certainty i can say that i wouldn't have reach the 0% of my career without you. And i dedicate all of my contribution in this project to you with all gratitude and love & may Allah protect and preserve you.*

I thank all my family, ***BELKHIRI Family*** *and* ***EZZEROUG family*** *especially my uncle* ***Soufiane****, my uncle* ***Djamel*** *and my favorite cousin* ***Younes****. Having you by my side has been an incredible source of strength. Thank you.*

The next person to thank, is a person that made this project come true and made this journey joyful, my partner, my best friend and my brother ***BENARAB Adel****. Thank you for your seriousness, your help, your support, your determination and above all your respect. I've grown so much with you, both personally and professionally. By far the best partner i ever had.*

To the best people I've known in the past 5 years, my best friends ***Amine****,* ***Abdelhadi****,* ***Tarek*** *and* ***Sidali****. Thank you for filling this journey with laughter, joy, support and unforgettable memories.*

Special thanks to ***VIC members*** *especially* ***VIC committee 2023*** *,****Eurobot 2024 team*** *and all my classmates.*

BELKHIRI Djamel Ibrahim

Contents

List of Tables

List of Figures

Acronym list

- **ABS**: Acrylonitrile Butadiene Styrene
- **API**: Application Programming Interface
- **AR**: Augmented Reality
- **CAD**: Computer-Aided Design
- **CNN**: Convolutional Neural Network
- **CSV**: Comma-Separated Values
- **DNN**: Deep Neural Network
- **DH**: Denavit Hartenberg
- **DOF**: Degrees Of Freedom
- **EEF**: End Effector
- **FWD**: Forward Kinematic Modeling
- **GPIO**: General-Purpose Input/Output
- **GUI**: Graphical User Interface
- **HDF5**: Hierarchical Data Format version 5
- **IK**: Inverse Kinematics
- **IMU**: Inertial Measurement Unit
- **ISO**: International Organization for Standardization
- **MAE**: Mean Absolute Error
- **MDP**: Markov Decision Process
- **ML**: Machine Learning
- **MSE**: Mean Squared Error
- **POV**: Point Of View
- **PLA**: Polylactic Acid
- **PPO**: Proximal Policy Optimization
- **RL**: Reinforcement Learning

- **ROS**: Robot Operating System

- **Rviz**: ROS Visualize

- **SGD**: Stochastic Gradient Descent

- **STEP**: Standard for the Exchange of Product Data

- **STL**: Standard Tessellation Language

- **URDF**: Unified Robotics Description Format

- **VR**: Virtual Reality

General Introduction

In the industrial sector, robotic arms have seen a meaningful evolution passing from classic industrial robots to the collaborative ones, or Cobots. Unlike the industrial ones, the Cobots are designed to be able to work alongside humans with no risk and full autonomy, which made them suitable for various applications such as : surgery, factory loading and unloading, and assisting workers in various tasks.

And since humanity is in continuous progress, we are working now on integrating Machine learning and image processing on these collaborative robots in the purpose of making them autonomous and able to take safe and secure decisions. One of the commonly used methods we have : Reinforcement learning next to supervised and imitation learning for robotic arms. This can help them integrate and adapt to any task in any work environment.

To this end, our final study project aims to design and develop a 6DOF autonomous collaborative arm product using machine learning and image processing techniques under the idea of creating our own startup company, making it intelligent and usable with different open-source softwares like ROS (Robot operating system) and Robodk for all industrial tasks and applications.

Problem statement

The main question here is : How can we first design this robot ? and what are the most suitable techniques that can ensure a robust autonomy and meet the industrial requirements at the same time ?

Work Methodology

In order to answer this question, we should firstly address the mechanical aspect involving each design with Soldiworks, the cinematic and dynamic analyses, the torque calculations to get a final designed model.

After that, we move to the simulation of the arm movements using the proposed softwares that will be presented in the thesis in order to study the design feasibility before getting to the realization part.

Once approved, we pass to the manufacturing or 3D printing of the arm components alongside the electrical and electronics aspect, in which we address circuitry design and soldering, the choice of the suitable motors, sensors, microcontroller and communication protocols, as well as conducting the electrical analysis to get the required current and voltage to ensure the proper power supply for best performance.

Next, we will move to the assembly part that allows real life testing using the presented softwares in the thesis on some basic tasks to ensure the secure movement for our robotic arm.

Finally, we are going to integrate the machine learning and images processing autonomy based techniques proposed in the thesis and test them on the robot to compare their efficiency and ability to be used in industrial and real-time applications.

Work Structure

To ensure a clear and coherent presentation of the thesis, and to make it easier for readers to understand the different stages of our work, it will be organized as follows :

- **Chapter 1 General definitions :** covers basic concepts and definitions of robotic arms and modeling, command softwares & different autonomy approaches.

- **Chapter 2 State of the art :** Exploring state of the art and related works in our field of studies.

- **Chapter 3 Mechanical Conception and Realization :** Presents the process of design, modeling, conception and realization of our robotic arm.

- **Chapter 4 ROS & Robodk Based Command and Electronic Integration of the Autonomous Robotic Arm :** Introduce different robotic arms command softwares as well as conducting the electrical analysis and the hardware integration.

- **Chapter 5 Robotic arm autonomy using machine learning :** Focuses on the autonomy based approaches we worked on, the performed tests and experiments and the obtained results in simulation and reality.

- **Chapter 6 Conclusion and Perspectives :** Summarizes the global work, the obtained results alongside future perspectives.

Chapter 1

General definitions

1.1 Introduction :

The following chapter representing the first chapter of the thesis, will handle the basic definitions that the reader should know and keep in mind before being able to understand and assimilate the explained work coming next. Taking into account both general aspects for robotic arms, some basic mechanical concepts, general idea about the machine learning and images processing techniques we went through and the softwares essential for the work.

1.2 Robotic arms

1.2.1 Definition of Robotic arm :

Based on the International Standard of Organization (ISO), a robotic arm is defined as reprogrammable, multifunctional manipulator designed to move material, parts and tools between 2 different positions through variable programmed motions for the performance of a variety of tasks[1].

And in other terms, it can be defined as a simple open-structure composed of (n+1) rigid body also called link denoted as C_n and (n) joint, starting from the base link which is the first rigid body and ending with the end effector which is the hand of our manipulator. Each joint J_n is made of the connection between 2 rigid bodies C_{n-1} and C_n, and each one of them might have either a rotational movement or a prismatic one based on the type of the joint, and the number of these joints will define to us the degree of freedom of our robot.

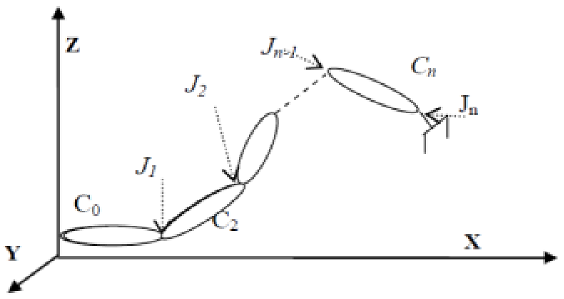


Figure 1.1: Simple open-structure robotic arm[2].

1.2.2 Classification of robotic arms :

To understand the main subject of this thesis, we should know that robotic arms can be classified based on their type, their geometry or finally their use. For our work the main part of our robot is its use.

1.2.2.1 The use

Based on the task and the comportment, we can either define the industrial robots which are the most common ones, known by their good efficiency but lack of safety especially if it required human interaction, but at the same time their high speed and precision too. Next to them we have the Collaborative robots (Cobots), which were designed for better execution alongside human-beings, they are characterized by their flexibility and adaptability. enabling the fact of using them in different tasks with no problem.

Figure 1.2: Kuka industrial arm[3].

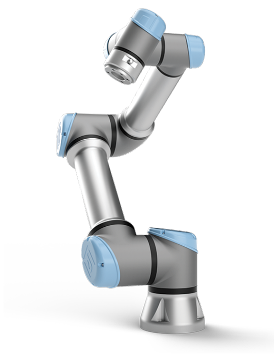


Figure 1.3: Ur5 Collaborative arm[4].

1.2.3 Modelisation of robotic arms

In order to control our arm's hand position, we need to control the movement of each one of its joints. This control can be done by applying some arm modeling techniques on our own arm. But to understand these techniques, we should start by the fundamentals first.

1.2.3.1 Homogeneous transformation

A **homogeneous transformation** is defined as a 4 × 4 matrix that represents any transformation (rotation or translation) from a coordinate frame R_i to a coordinate frame R_j:

$$T_i^j = \begin{bmatrix} s_i^j & n_i^j & a_i^j & p_i^j \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & n_x & a_x & p_x \\ s_y & n_y & a_y & p_y \\ s_z & n_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where s_i^j, n_i^j, and a_i^j are the unit vectors along the X_j, Y_j, and Z_j axes of frame R_j, respectively, expressed in frame R_i. The vector p_i^j represents the origin of frame R_j expressed in frame R_i.

1.2.3.2 Geometric modeling

The geometric modeling of a robotic arm is based on getting a mathematical representation of the robot's physical structure and configuration. It relies on the geometry of the links, the joints and the way they are interconnected. So the main idea is to find a relation between the position of the end effector and the base link of the robot based on the movements applied on each joint. For this purpose, we have to methods of this modeling :

1.2.3.2.1 Forward geometric modeling : Computes the end-effector's position based on joint parameters using the***Denavit-Hartenberg (D-H)*** Convention which simplifies the assignment of coordinate frames through four transformation parameters by the following steps :

1.2.3.2.1.1 Assigning the joint frames : The z-axis is aligned with the joint motion (rotation for revolute, translation for prismatic). The x-axis lies along the common normal between consecutive z-axes, pointing from one joint to the next. The y-axis is determined using the right-hand rule. This setup may lead to coordinate frame origins located outside the joint itself for consistency[5].

1.2.3.2.1.2 Determining DH Parameters : The transformation between two joints is then fully described by the following four parameters [5] :

1. **The length a** of the common normal between the z axis of two joints i and $i-1$ (link length).
2. **The angle** α between the z-axis of the two joints with respect to the common normal (link twist), i.e., the angle between the old and the new z-axis, measured about the common normal.
3. **The distance d** between the joint axes (link offset), i.e., the offset along the previous z-axis to the common normal.
4. **The rotation** θ around the common axis along which the link offset is measured (joint angle), i.e., the angle from the old x-axis to the new x-axis, about the previous z-axis.

1.2.3.2.1.3 Calculating the transformation matrix : The coordinate transform from one link $i-1$ to another i can now be constructed using the following matrix:

$$A_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

1.2.3.3 Dynamic modeling

Dynamic modeling is a robotic essential part for the design as it incorporates factors like joint torques, gravity, and link masses. It helps determine the maximum torque needed for smooth joint movements critical for selecting appropriate actuators.

1.2.3.3.1 Torque : It represents the required force to move an object with a linear or angular acceleration. For robotic arms often denoted by the Greek letter tau τ, it represents the required force in order to rotate an object around an axis or a pivot.
In general, it is defined as the product of the force (F) applied and the perpendicular distance (r) from the axis of rotation to the point where the force is applied :

$$\tau = r \cdot F \cdot \sin\theta \tag{1.2}$$

Where τ is the required torque, F is the applied force, r is the distance between the axis of rotation and the point of force application and θ is the angle between them.
For robotic arm use, this formula can change a little bit, in our case we are going to calculate it using this new one :

$$\tau = g \cdot m \cdot r \tag{1.3}$$

This time, the $g$represents the gravity factor in (m/s), m represents the mass of all the links related to that joint in (kg) and r remains the distance between the axis of rotation and the gravity center of all the mass we need to rotate.
After doing this for each joint, we should take into account that not all the motors will be able to provide us the required torques, for that many systems have been created to increase the output torque of motors.

1.2.3.3.2 Torque increasing systems : For our case of work, for revolute joints there are many systems that could be used to increase torque and we opted for the planetary gearbox system :

1. **Planetary gearbox :**The commonly used system in rotational movements, it relies on 4 important elements which are :

   - **The sun gear :** Which is the center of the movement, generally represents the input of the movement so it is fixed with the rotation axis of the motor.
   - **The ring gear :** An outer gear with internal teeth that mesh with the planet gears.
   - **The planetary gear :** Usually three or more identical gears that mesh with the sun gear externally and the ring gear internally. They rotate around their own axes and revolve around the sun gear (hence the "planetary" name).
   - **The carrier :** A structure that holds the planet gears in place and connects their axes. As the planet gears revolve around the sun gear, they cause the planet carrier to rotate, and is typically the output of the gearbox.

   After assembling all these together, the planetary gearbox system will look like :

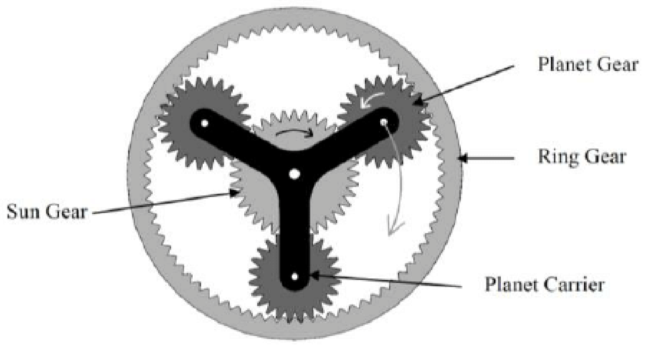
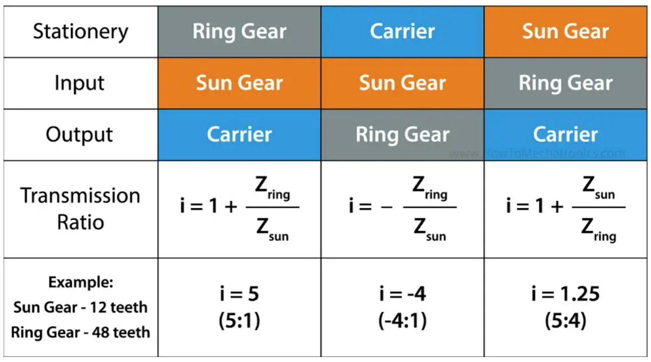


Figure 1.4: Planetary gearbox[6].

For this to work, 2 main important characteristics should be taken into consideration. Each gear is defined by its number of teeth and the global system ratio is defined by the combination we use, so it means we should define the input , the output and the fixed part in it. The following table will explain the impact of the combination on the global ratio of the gearbox.



Stationery	Ring Gear	Carrier	Sun Gear
Input	Sun Gear	Sun Gear	Ring Gear
Output	Carrier	Ring Gear	Carrier
Transmission Ratio	$i = 1 + \frac{Z_{ring}}{Z_{sun}}$	$i = -\frac{Z_{ring}}{Z_{sun}}$	$i = 1 + \frac{Z_{sun}}{Z_{ring}}$
Example: **Sun Gear - 12 teeth** **Ring Gear - 48 teeth**	**i = 5 (5:1)**	**i = -4 (-4:1)**	**i = 1.25 (5:4)**

Figure 1.5: The impact of the combination on the global ratio of the gearbox[6].

After determining the ratio of the gearbox, the torque of the motor we choose will be multiplied in order to reach the desired torque for each joint.

1.2.3.4 Actuators and sensors

1.2.3.4.1 Actuators : Starting with the actuators, which represents the main part responsible for the movement of our structure. Generally we use one of the 2 well known motors :

1. **Stepper Motors :** A stepper motor, also known as step motor or stepping motor, is a brushless DC electric motor that rotates in a series of small and discrete angular steps. Stepper motors can be set to any given step position without needing a position sensor for feedback which is the best case of use since the are open loop motors. The step position can be rapidly increased or decreased to create continuous rotation, or the motor can be ordered to actively hold its position at one given step. Motors vary in size, speed, step resolution, and torque[7].

Figure 1.6: Stepper motor[7].

In a stepper motor, discrete rotor movements are produced by selectively energizing its stator windings: when you power one winding (Phase A), you generate a magnetic field that the permanent-magnet rotor aligns to; then you de-energize Phase A and energize Phase B, causing the rotor to “jump” to the next position.

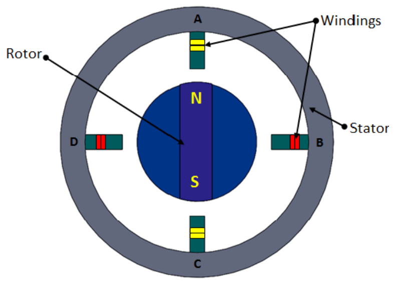


Figure 1.7: Stepper motor magnets[8].

2. **Servo motors :** A servo motor is a complete closed-loop system, integrating a motor, a position sensor (like an encoder), and a controller. It operates by constantly comparing the commanded position (the "setpoint") with the actual position reported by its feedback sensor. The controller uses any difference, or "error signal," to drive the motor, dynamically adjusting speed and torque to eliminate the error and maintain the target angle with high accuracy.

Figure 1.8: Servo motor[9].

1.2.3.4.2 Sensors : In the robotic arm field, many types of sensors are used where each one of them has a specific role in the execution of the task. But at the same time, their presence in the work is not necessary because it depends on the task and the construction of the arm.

- **Limit Switch :** A limit switch is an electromechanical sensor, called this way since it detects the presence of an object from the contact with its limit.

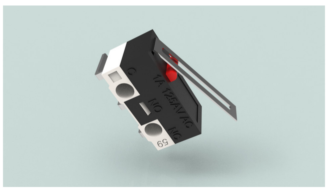


Figure 1.9: Limit switch[10].

In robotic arms, it can be used in order to detect a certain desired position for each joint range of movement, either by detecting the initial position (mostly used in case of joint's initialization) , or can be used when we reach the limit of possible movement.
One of the easiest sensors to set, it has 3 pins to connect and based on that it can be set to either normally opened (NO) or normally closed (NC), This 2 will be connected to the ground (GND) with a pull-up or pull-down resistance and the desired pin at the same time, while the 3rd pin which is the common (COM) will be connected to the power source (+5V) as showing the figure.

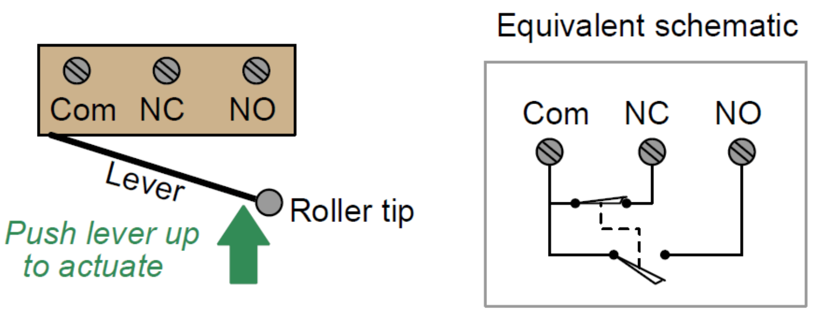


Figure 1.10: Limit switch principle[10].

So whenever we get contact between the subject and the roller tip of the limit switch. The switch will be activated and will read a high or low signal on the reading pin based on the configuration we choose.

- **Inertial Measurement Unit (IMU) :** The inertial measurement unit is the most common sensor used in order to measure the acceleration and orientation of a specific object in space. It is mainly composed of 3 elements which are :
    - **Accelerometer :** Which is used to measure the linear acceleration on the 3 axis.
    - **Gyroscope :** That we use to measure the angular velocity this time on each axis.
    - **Magnetometer :** Which is an optional part in most of the IMU's available, which is better used alongside the gyroscope for tuning and data filtering since it gives information of the magnetic field of earth on each axis.

  As the following figure shows, the most standard IMU is called MPU-6050.

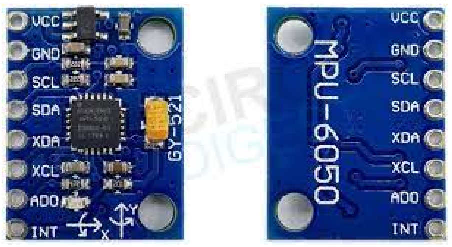


Figure 1.11: MPU-6050[11].

- **Encoders :** Principally, it is a position sensor that we use in order to get the position of a certain object in 2D space. Based on its type, it converts the mechanical movement of the object to an electrical signal that will be used after to calculate the distance or the position of the subject.

The following figure shows us the 2 types of encoders available, either a magnetic encoder attaches a magnetized disk or ring to the rotating shaft and uses stationary Hall-effect or magneto-resistive sensors to "read" the changing magnetic field as the shaft turns. The sensors generate two digital pulse trains (channels A and B) in quadrature allowing both position counting and direction detection. Or an optical encoder, that has a slotted disk rotates between an LED and photodiode array. So, as the disk turns, light passes through slots and it's interrupted, creating A/B pulse trains.

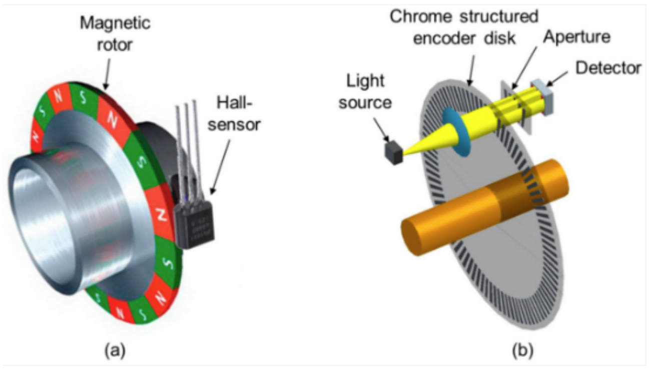


Figure 1.12: Types of encoders[12].

- **RGB Camera :** It is simply a visual sensor that collects images by assembling them in 3 channels (Red, Green and Blue). Each camera is defined by some of its special characteristics, such as: Spatial and temporal resolution (Fps and height*width), Lens and factor of distortion and inclination, etc.

Figure 1.13: Havit RGB camera[13].

1.3 Robot Operating System

1.3.1 Robot Operating System (ROS)

The robot operating system, also called ROS. It's an open-source middle-ware framework for robotics development tasks. Alone, it's not really an operating system but it can be used on real ones, specifically Linux.

Figure 1.14: Robot Operating System[14].

The main reason behind calling it an operating system even though it does not represent a real one, is because ROS ensures a structured communication system alongside many useful tools and packages that facilitates the visualization and study of robot comportment.

1.3.2 ROS Architecture

In order to maintain the communication system, ROS has a got special unsynchronised architecture based on 2 important components :

1.3.2.1 Nodes :

Which represents simply the code or the function that executes a certain task, for example reading from sensors or giving action for command. It might be a python, C or C++ code that differentiates between each other by only the way of calling and setting up the node.
But in communication systems as clearly known, we should have 2 important pairs involved in this communication; a sender and a receiver. In the ROS system these 2 has special names :

- **Publisher :** The part that sends information of a specific type in the ROS architecture, it represents the sender in the communication systems.
- **Subscriber :** This one is supposed to receive the data sent by the publisher. And since each one of them has got a certain role, each one of them has got a special architecture in the code.

1.3.2.2 Topics :

For the topics, it is simply the name we use to define the information or the variables exchanged between the publisher and subscriber whatever the type of it. In the architecture it is used to define the channel that transports data from the transmitter to the receiver.

So as a global definition, the main architecture of the robot operating system is based on 2 nodes or more exchanging data under channels that have the same name of the variables called Topics, where the parties involved in the communication system are classified between publishers that send the data and subscribers that receive it. All this can be explained in the following figure.

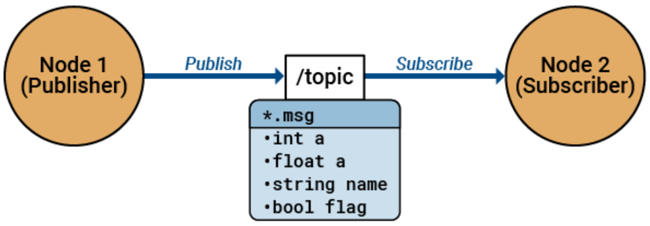


Figure 1.15: Simple example of ROS architecture[15].

1.3.3 ROS tools

In addition to the architecture property, as mentioned before the robot operating system is commonly used thanks to the tools and packages that can be used easily for robot manipulation and simulation.

1.3.3.1 Tools

Starting with the tools available within ROS. We should first understand that tools represent the set of applications and utilities that can be used for simulation specially in ROS. The mainly 2 for robot simulation and visualization are :

1.3.3.1.1 Gazebo : The Gazebo tool is a 3D design simulator widely used for robotic purposes. It gives the ability to study the kinematic and dynamic of any system in or outdoors while simulating the environment at the same time. The next figure represents the general logo of Gazebo.

Figure 1.16: Gazebo[16].

This tool allows accurate simulation of rigid body physics, enabling robots to interact with their surroundings for example, by picking up or pushing objects, rolling, or sliding on surfaces while also being affected by physical forces such as gravity and collisions with obstacles. The following figure shows an example of a simulated arm in its environment using Gazebo.

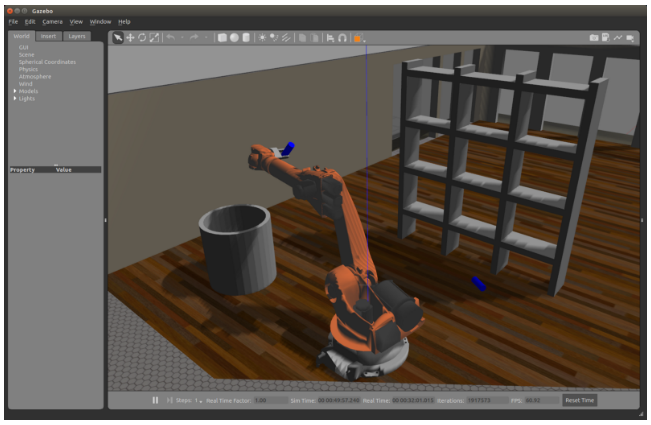


Figure 1.17: Gazebo world[17].

In the next sections, we will be explaining the possibility and how to set up a gazebo with your own robot for accurate simulations.

1.3.3.1.2 RviZ : Standing for Ros visualizer, is a famous 3D simulation tool for ROS. Same as the gazebo it gives the ability to simulate the movement of the robot with the ability to set up the environment too.

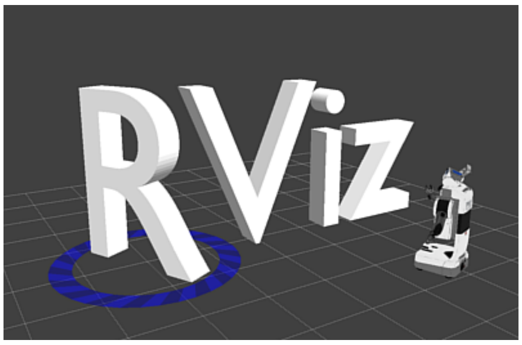


Figure 1.18: Rviz logo[18].

In graphical comparison , Rviz is less convenient since it doesn't really take into account the graphical representation, it mainly focuses on the representation of the robot in a state that will help us study its comportment. So we can say basically that it is less resource consuming if compared to Gazebo. But at the same time, it gives the minimum required for graphical visualization of the robots comportment so it might be sometimes better to use for basic simulation and work.

1.3.3.2 Packages

Talking now about the ROS packages, they represent the set of programs and code folders that we can use and set up directly for our use. Sometimes it might require some digging and fixing in order to make it work with our project and sometimes it's so easy to launch it with no problem. In our case we have :

1.3.3.2.1 Rosserial : The rosserial package is a built-in function of ROS that ensures the serial communication (UART, USB) between the publishers and subscribers. This package is the most crucial part of the ROS since as previously mentioned it's architecture relies principally on ensuring that communication with no problems.
In a more detailed way, the rosserial protocol is aimed at point-to-point ROS communications over a serial transmission line. We use the same serialization/deserialization as standard ROS messages, simply adding a packet header and tail which allows multiple topics to share a common serial link. This page describes the low-level details of the packet header and tail, and several special topics used for synchronization[19].

1.3.3.2.2 MoveIt : The moveit package is the number 1 package of ROS for robotic arm simulation, motion planning and manipulation. It has got many built-in function used for trajectory planification , inverse kinematics solving next to its main important part which is the moveit setup assistant package that helps for the integration of your own inside of it in some few steps that will be explained after in the ROS chapter.

1.4 Reinforcment learning

1.4.1 Reinforcement learning definition

Reinforcement learning is an approach used to train a specific agent the way to perform a certain task, the agent is the entity doing the action in our case that represents the robotic arm. The main idea is that the robot starts to learn based on its action and the reward or the feedback that he gets from its external environment and its sensors [20].This whole approach is known by the markov decision process.

1.4.2 Markov Decision Process (MDP)

The markov decision process or MDP, is a mathematical approach used in reinforcement learning that relies on the feedback of the environment of the agent to give a rait on the action performed. So basically, for each action the agent taks, we will have a reward or punishment value for that and by accumulating these with the time the agent or the robot in this case will know how to avoid prohibited movements so as to accomplish the task correctly and safely.

- **The agent :** Which is as explained the entity we are trying to teach how to perform a certain task.
- **The environment :** Which is the surrounding space of the agent, in the case of a robotic arm its workspace in the factory or a shop or whatever place it's going to be used in.
- **The action :** The movement performed by the agent, either by joint rotations for a robotic arm, for the displacement of a navigation robot or flying around the sky for drones.
- **The state :** Which represents the current position of the agent after performing the action.
- **The reward :** Which is the value or the feedback we give to that action taken based on the current state, but this one should rely primarily on the reward tree which will be explained afterwards.

As it's shown in the figure, the agent will visualize its environment then decide to take an action Ai. As a result, it will have a new state S_{i+1} and will get a reward value R_{i+1} for the action taken. And with that we go through a loop again and again until the end of the episode or end of the training.

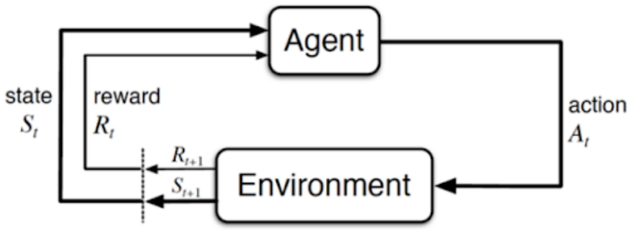


Figure 1.19: Markov decision process[21].

So the MDP can be represented mathematically by a 4 elements tuple (s,a,P,R)[22]:

- **States(S) :** Which is a set of states possible for the robot, in our case can represent the position of the end effector in its workspace.
- **Actions(A) :** Which are the possible movements of the agent, for our robot it's the rotation values for each joint that will lead to an end effector position.
- **Transition probability function** (P) **:** It's the probability $P(s' \mid s, a)$ that maps for each action A at a certain state S the new state possible for the robot after taking that action S'.
- **Reward function (R) :** Which is the rewarding function $R(s, a, s')$ that gives for each action taken a feedback value. These values are given based on the reward tree.

And the main purpose of this MDP is to maximize a π policy that accumulates the reward values of the agent during the training[23].

$$\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})\right] \tag{1.4}$$

Where S_t and A_t are the state and action for the time t, and gamma is a factor between 0 and 1 that gives importance to the next reward in the training specially used in continuous space of actions.

1.4.3 Reward Tree

The reward tree is an interpretable representation of the reward function R, it defines rewards through a hierarchy of decisions based on features of states and actions. Used to facilitate the interpretation of all possible reward values given to the action[24]. As the following figure represents a 3 states reward tree and their probabilities.

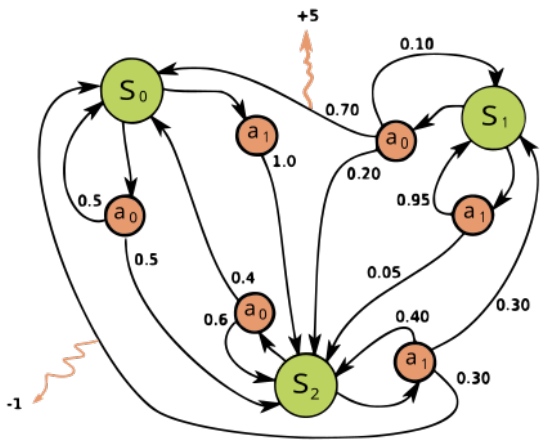


Figure 1.20: Reward tree[22].

1.4.4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is an on-policy, model-free reinforcement learning algorithm designed to facilitate stable and efficient policy learning[25]. The main idea behind this PPO, is to use a 'PPO-Clip' variant to ensure stable policy improvement by preventing overly large updates that could destabilize learning.

The main structure of the PPO relies on experience collection and evaluation by the agent, then trying to maximize the clipped surrogate objective function given by :

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\ \hat{A}_t\right)\right] \tag{1.5}$$

Which takes into consideration a trust ratio $r_t(\theta)$ that refers to the possibility of taking an action a' knowing that the previous one is a. In this case if we have a higher value then 1 means that the new policy would like to take this action, but if it's lower than 1 so the policy will be avoiding to take this kind of action in the future. And we have the main part of the $PPO - Clip$ which is ϵ which is used to define a range of how much can the policy change with the time and avoid aggressive changes directly. So if the action taken was very good, we will have the trust ratio very high but this will lead the policy to prioritise this action a lot, so it is constrained with $1 + \epsilon$ .The same way if its a very bad action , it doesnt let the policy consider it as a prohibited one to enable the exploration of the environment by limiting it with $1-\epsilon$.

In essence, the main idea of PPO (specifically PPO-Clip) is to modify the policy function itself in a simple way (clipping) that implicitly constrains the policy updates, preventing them from deviating too far from the previous policy, thereby ensuring more stable and reliable learning with first-order optimization.

1.5 Supervised learning

1.5.1 Supervised learning

Supervised learning is a branch of machine learning where an algorithm is trained on a dataset comprising pairs of input features and their corresponding output labels. During training, the model learns the underlying patterns linking inputs to outputs. Once trained, it can generalize this mapping to predict outcomes for new, unseen data[26].

1. Classification tasks involve predicting discrete labels (e.g., cat vs. dog).
2. Regression tasks involve predicting continuous values (e.g., house price).

Supervised learning relies on labeled training data to learn the relationship between input features and their corresponding outputs. Data scientists prepare these datasets by assigning known output labels to each input example. The goal of supervised learning is to train a model that can accurately predict the correct output for new, unseen inputs based on the patterns learned from the training data.

During the training process, the algorithm analyzes large amounts of input–output pairs to identify correlations. The model's performance is then evaluated using a separate test dataset to ensure it generalizes well. Cross-validation is often used to assess model performance by dividing the dataset into training and validation segments, allowing the model to be tested on unseen data during training.

One of the most widely used optimization techniques in supervised learning is gradient descent, including variants like stochastic gradient descent (SGD). These algorithms adjust the model's parameters to minimize the loss function, which quantifies the difference between predicted and actual output values. The algorithm follows the gradient of the loss function to find the parameter values that produce the most accurate predictions.

1.5.2 Convolutional Neural Network (CNN)

convolutional Neural Networks (CNNs), also known as ConvNets, are a type of feed-forward neural network where data flows sequentially from input to output through multiple layers. CNNs are particularly effective in tasks involving image recognition and classification due to their ability to automatically detect and learn spatial hierarchies and patterns within visual data
Unlike traditional neural networks, CNNs include specialized layers such as :

- **Convolutional layers :** Convolution is the first step in a convolutional Neural Network (CNN) used to extract important features from an input image. In this layer, a small matrix called a filter or kernel slides over the image and performs a mathematical operation to highlight specific patterns, such as edges, corners, or textures.

- **Pooling layers :** which reduce the spatial dimensions of the data to highlight the most significant features and reduce computational cost.

- **Activation functions :** enhance model performance and introduce non-linearity.

- **Fully connected layers :** used to connect every neuron in one layer to all the neurons in another layer. We flatten our pooled feature map matrix into vector and then feed that vector into a fully connected layer.

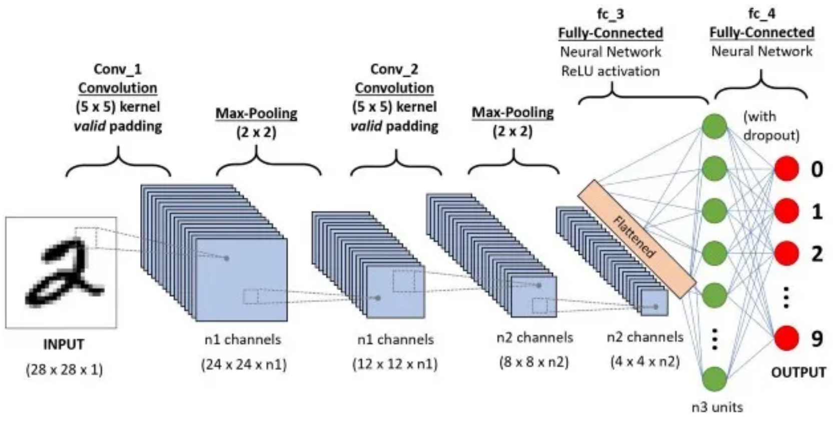


Figure 1.21: Convolutional neural network architecture[27].

CNNs learn to identify simple patterns (such as edges, curves, or textures) in the early layers and progressively build up to more complex structures (like objects, faces, or scenes) in deeper layers, making them highly suitable for tasks involving visual data.

1.6 Aruco pose estimation

1.6.1 Aruco markers

1.6.1.1 Definition

An Aruco marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (id). The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of error detection and correction techniques. The marker size determines the size of the internal matrix. For instance a marker size of 4x4 is composed of 16 bits.
So each Aruco marker is specially composed of 2 main parts :

1. **Thick black border :** This is the most important feature. The solid black border makes it extremely easy and fast for a computer vision algorithm to find the marker in an image, even under different lighting conditions or at various angles.
2. **Inner binary pattern :** Inside the black border is a grid of smaller black and white squares. This pattern represents a binary number, which is the marker's unique ID. These patterns come from a predefined dictionary that will be explained in the next point.

1.6.1.2 Types of classification

In Aruco, a dictionary is simply the collection of all marker patterns you'll recognize in your application. Each marker is defined by its unique binary grid ("codification"), and dictionaries differ in two key ways:

1. **Dictionary size :** how many distinct markers it contains, so each dictionary contains a certain number of markers so we make the difference between them in the use.
2. **Marker size :** the dimensions of each marker's grid, exactly the size of the inner binary part of the Aruco.

So as the following figure shows, we got different Aruco markers from different dictionaries. As we can simply deduce, there is a difference in the shape of them, the quantity and the pattern of the inner part so difference in the binary pattern we are using to define each one of them.

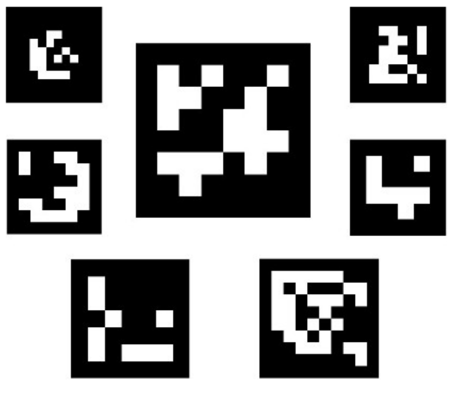


Figure 1.22: Different Aruco markers.

We should know that the difference between these dictionaries has a serious reason, that will be next explained.

1.6.2 Aruco tags detection

1.6.2.1 How to detect Aruco tags

In the robotics field, the Aruco marker detection is such a big need, because it will help for position estimation of either the robot in its environment or a specific we reach to track or localize in the robot frame. So we should principally understand that we are using visual sensors for this work, mainly an RGB camera.

Now, this detection requires some important steps in order to detect the desired Aruco marker in space.
Using an RGB camera or any type of cameras, we should capture the image frame and apply some image processing techniques on it. Starting by converting the image in grayscale to reduce the calculations and the resources, additionally because the Aruco are already in white and black pixels, so the information from RGB channels is not required. Taking for an example the captured image shown in the figure.

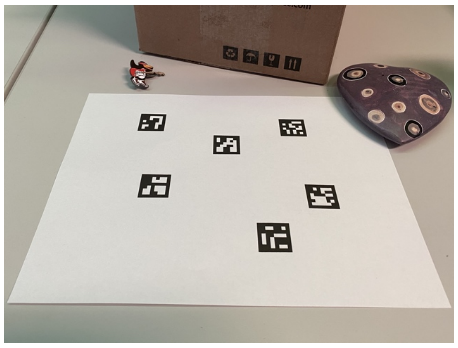


Figure 1.23: Captured images with RGB camera.

Next, we apply a binary threshold on it and that to detect the outer black border of the Aruco tag. So we get the following figure from the previous one.

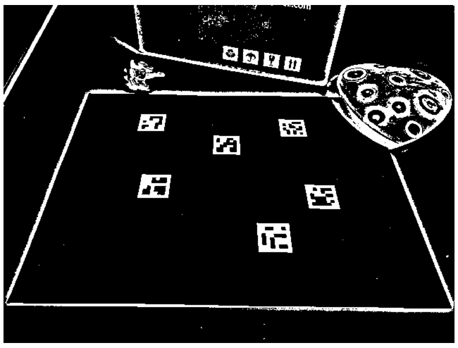
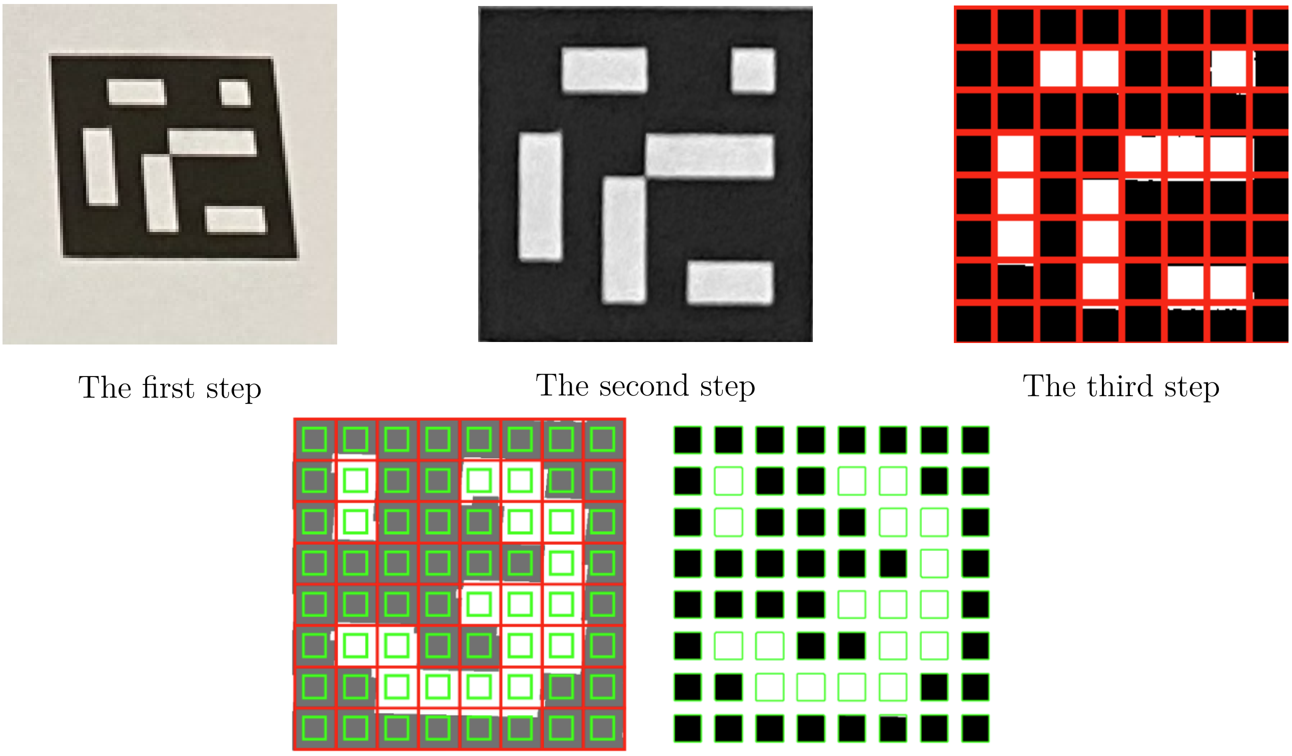


Figure 1.24: Binary threshold image.

After applying the threshold, we will be searching for squared contours that represent to us the black border of the Aruco marker. Taking for example in the following figure, we detected the Aruco marker of the image that we divide based on the predefined dictionary and marker size we are using, as for the figure we used 8*8 Aruco markers.



The first step The second step The third step

The last step

Figure 1.25: Aruco grid and matrix extraction.

1.6.3 Aruco pose estimation

Getting now to the importance of Aruco tags and all of this work. The main idea behind the markers detection is to finally use them with position estimation techniques for 2 main reasons :

1. Either localizing the robot position in its space using the predefined Aruco tags in its area commonly used with navigation robots.
2. Localizing the Aruco itself in the robot space in order to perform a task on the object linked with the Aruco, which is the most used case with robotic arms.

When we say pose estimation of an Aruco marker, we mean that we want to get its distance from the visual source (The RGB camera), and using some frame transformation techniques previously explained, while working with robotic arms we are going to use that for localizing the required object in the robot general frame. So we are trying to get the X,Y,Z coordinates and orientation of the desired object.
For Now this is all that we are going to explain about the Aruco pose estimation, since it is a full approach that we are going to use and explain in the Autonomy chapter.

1.7 Conclusion :

As a conclusion for the chapter, all essential informations and basic concepts required for the good comprehension of the thesis were clearly defined.

Chapter 2

State of the art

2.1 Introduction

After discussing and explaining the main definition of our work, now we should explore the state of the art and related work in the subject of machine learning techniques for robotic arms. And after digging for related works, the main 2 promising techniques were either the supervised learning or the reinforcement learning but for each one we have some special methods and a global architecture for the work to be done.

In what's coming next, we are going to explore and explain the main related works in this field using both methods, explain the main points for each one and end it with a general conclusion based on the informations collected from those works.

The main point of this part, is to get an idea about the most suitable techniques for our case of use, try to understand the work done and use it for better results and even close to the already done work.

2.2 Supervised learning

Supervised learning has shown significant promise in enabling robotic arms to learn complex manipulation behaviors from data. In recent years, neural networks, particularly convolutional and recurrent architectures, have been applied to predict robot control actions from sensory data such as RGB images, proprioception, or both. These models are typically trained on large datasets where robot actions are paired with observations.
Several notable datasets and benchmarks (e.g., RoboNet, RoboTurk, and DeepMind Control Suite) have enabled supervised policy learning and behavioral cloning. These datasets contain trajectories where robots perform various manipulation tasks such as pushing, grasping, reaching, and object relocation.

2.2.1 Datasets

These datasets provide multimodal information such as RGB or RGB-D video, joint angles, and actions from various robot platforms, enabling supervised and self-supervised learning of motor policies. In Table 2.1, we summarize key characteristics of widely-used datasets including RoboNet, RoboTurk, and the BAIR Robot Pushing dataset, which support different tasks such as object pushing, grasping, and pick-and-place operations. These datasets form the foundation for training models that generalize across environments and robot types.

	RoboNet	**RoboTurk**	**BAIR Robot Pushing**
Authors	Aravind Rajeswaran et al.	Mandlekar et al.	Finn, Levine et al.
Year	2019	2018	2017
Reference	[28]	[29]	[30]
Data Modality	RGB images, depth, joints commands, actions	RGB-D, joint states	RGB video, robot states
Robot Platforms	Sawyer, WidowX, Franka, UR5e, Baxter	WidowX, UR5	Sawyer
Type of Task	Pushing, reaching, pick-and-place	Grasping, pushing, real-world manipulation	Pushing small objects on table

Table 2.1: Comparison of Robotic Arm Datasets for Manipulation Tasks

2.2.2 Pose estimation

This part provides an overview of significant advancements in the field of robot pose determination from visual data. It highlights various methodologies, with a particular focus on deep learning-based approaches, which offer promising solutions to the challenges of accuracy, robustness, and real-time performance.

The core problem is to accurately estimate the joint angles of a robot manipulator from visual input. This is critical for various applications, including robot control, human-robot interaction, teleportation, and autonomous navigation, particularly in scenarios where traditional encoders might fail or where external sensing is preferred. The system aims to provide a robust alternative or complement to joint encoders.

	Pose Determination System Based on Neural Networks[31]	**Accurate Robot Arm Attitude Estimation Based on Multi-View Images and Super-Resolution Keypoint Detection Networks**[32]
Author	Sergio Rodríguez-Miranda et al	Ling Zhou et al
Year	2023	2022
Input	RGB image of robot manipulator and encoder values (for training)	Multi-view RGB images of the robotic arm
Output	Estimated robot joint angles (pose)	3D joint keypoints and estimated attitude
Approach	- CNN for pose classification - MLP for regression - Forward kinematics used for final pose	- Super-resolution keypoint detection - Multi-view image fusion - Deep learning-based 3D attitude estimation
Note	Combines CNN and MLP for classification and regression, using kinematics for pose estimation	Uses super-resolution and multi-view keypoint detection for precise pose estimation

Table 2.2: Comparison of Neural Network-Based Robot Pose Estimation Methods

2.3 Reinforcement learning

For this section, we will be presenting some of the related works in reinforcement for robotic arms tested on some known robots and personal designed ones for some.

2.3.1 Related works

Before talking about our own training structure for reinforcement learning on BRAS-DEL robot, we will be exploring the related work in reinforcement learning for robotic arms shown in the following table.

Name	**Year**	**Author(s)**	**Trainer Type**	**Input Size**	**Results**
Inverse kinematics solution and control method of 6-DoF manipulator based on deep reinforcement learning[33]	2024	Zhao et al.	PPO	16	6mm in position, 10° in orientation
PPO for 6-DoF Grasping in Space Robotics[34]	2023	Høgsbro et al.	PPO	19	91.5% success (standard), 88.7% (occluded)

Table 2.3: Summary of selected PPO-based approaches for robotic arm control and grasping

Here we explored previous work in reinforcement learning for robotic arms. As we can the 3 listed works used the PPO trainer since its the most suitable generally for continuous space of action, and we can see that most of the work are really recent so it's still under development.

In addition we can see different ways in result presentation, some of them explore the efficiency of their work by calculating the mean error in distance and orientation from the end effector of the robot to the goal object as in the Zhao et al. work[33], we can see that the work of hogsbro et al[34]. presented their results in other form by giving the accuracy percentage of task accomplishing correctly from various of tests.Last thing to mention,is that the works differ from each other in the input space dimensions, so each one has included its available data based on its case of training and simulation setup.

2.3.2 Overview of the previous work architectures and results

Aspect	**Zhao et al[33].**	**Høgsbro et al [34].**
Robot	6-DoF industrial robot	Franka Emika Panda (7-DoF)
Task	Pick and place (IK solver training)	Rock grasping in simulated Mars terrain
Simulation Environment	Unity ML-Agents	Isaac Gym
Reward Function	Based on distance and orientation error with penalties	3-phase reward: reaching, grasping (50), lifting (variable) + success bonus
Input Size	16 (EE position/orientation, goal pose, joint angles)	19 (joint positions/velocities, object pose)
Output	Joint angles	8 outputs (7 joint angles + 1 gripper command)
Trainer	PPO	PPO
Learning Rate	0.0002	0.0002
Max Steps	30 million	70 million
Special Configs	Used ResNet as encoder, larger buffer and batch size	Entropy 0.005, domain randomization
Results	Continuous improvement in reward, distance error 6mm, orientation error 10°	91.51% success rate over 100k grasps, increasing reward

Table 2.4: Comparison of two DRL-based approaches for robotic manipulation

2.4 Conclusion

In our study of the state of the art for robotic arm autonomy using supervised learning, we didn't follow the existing works directly. Instead, we used them mainly to understand what kinds of datasets are commonly used. One important dataset we chose is RoboNet, which contains useful examples of robotic arms performing manipulation tasks.

We also looked at the neural networks used in pose estimation and found that CNNs (Convolutional Neural Networks) are the most common, since they work well with visual input like images.

Based on this, we proposed our own idea: instead of just estimating poses, our work will be done to predict the next joint position $Qpos_{i+1}$ given the current joint position $Qpos_i$ and the current image Img_i. In simple terms, our model learns the pattern:$Qpos_i + Img_i = Qpos_{i+1}$. This allows the robot to learn how to move like other robotic arms doing the same tasks, such as pick and place, by cloning their behavior.

As for the reinforcement learning part, based on the cited papers we concluded that the PPO trainer is the most suitable one for our case since it's the most used one for continuous space of actions, as for the reward tree we found out that their non general definition or function to use, each work has got to define its own reward tree as long as it respects the desired work or objective we want to achieve. In addition to that, the input data can change based on the available feedback and information in the scene.

Chapter 3

Mechanical Conception and Realization

3.1 Introduction

This chapter mainly focuses on the mechanical conception and realization of the robot. In which we will be starting with the used software Solidworks, the links and end effector conception, then we will talk about the motors choice and specifications. After we have the gearboxes part in which we talk about their conception and torques then the general simulated assembly of the robot. In addition we will have a section for the robot modeling, forward and inverse geometric modeling and the dynamic study of the torques. We do have after a special section for the realization in which we are going to explain the steps we went through for the final assembly (the printing and all the required mechanical tools we needed). Then at the end, we will have a general conclusion of the chapter.

The main idea of this work is to make an autonomous 6 DOF robotic arm that can be used in different industrial and collaborative tasks. The idea of the work was inspired by the Universal Robots company that is well known in the robotic arms industry. The design was primarily inspired from the Ur3 model. It gave us the general idea of how the robot should look, and we used our own inspiration to design the links one by one.

3.2 Robot Conception

3.2.1 SolidWorks

SOLIDWORKS is a 3D computer-aided design (CAD) software that runs on Windows. It is used to develop mechatronic systems from start to finish. In the initial stage, the software is used for planning, modeling, feasibility assessment, prototyping, and project management. The software is then used for the design and construction of mechanical, electrical, and software components. Finally, it can be used for device management, analysis, data automation, and cloud services[35].

Figure 3.1: SolidWorks logo[36].

3.2.2 Links and end effector conception

Starting now with the conception and design of the links and the end effector of the robot. As mentioned in the Introduction, we got inspired from the Universal Robot Ur3 model shown in the following image.

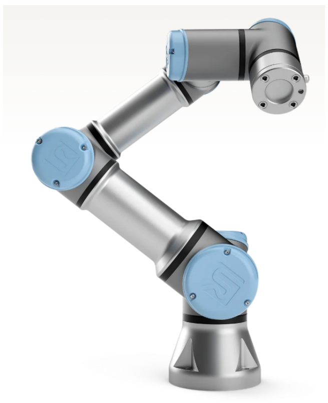


Figure 3.2: UR3 robotic arm[37].

Based on this model, we started designing our own robot. Our robot is a 6 DOF robotic arm of 6 revolute joints, 4 of them horizontal and 2 are vertical.

3.2.2.1 Base Link :

Starting with the base link, which is the first link of the robot which is going to be fixed on the operation table of the robot with a 188mm height, internal circle of 80mm diameter and external one of 125mm diameter with 6 holes will be used for fixation with the table. Inside of it, we have a place for the motor we are going to use, a nema23 stepper motor with a hole from the bottom used for cables and a weight of 300g as shown in the following figures.

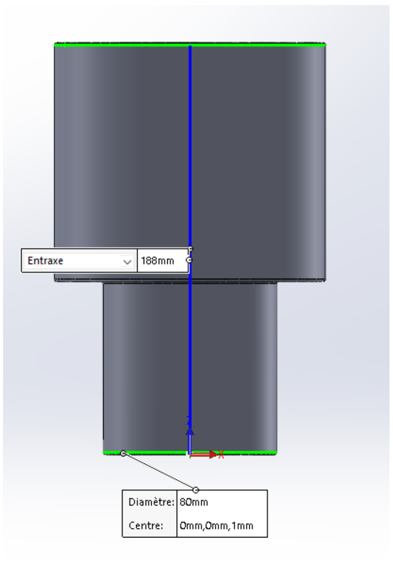


Figure 3.3: Base link face view

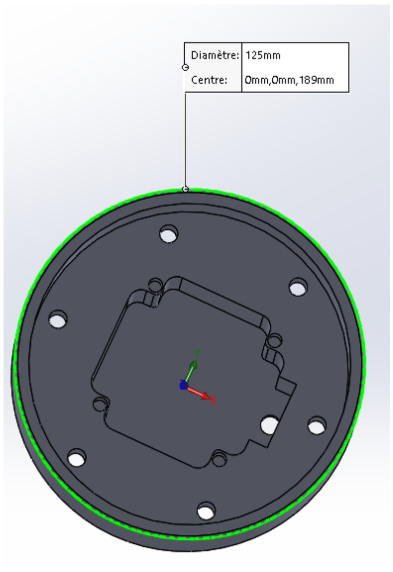


Figure 3.4: Base link upper view

3.2.2.2 Link1 :

This is the next link we worked on, it is a circle of 140mm diameter and 158 mm height. At the top of it there is our robot name BRAS-DEL, from the bottom side we have 6 holes that are going to be used after for connection with the base link to form the first joint J1.

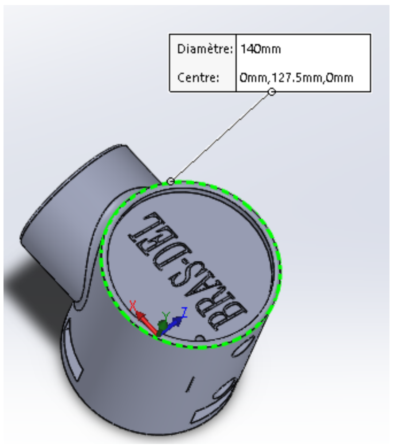


Figure 3.5: Link1 upper view

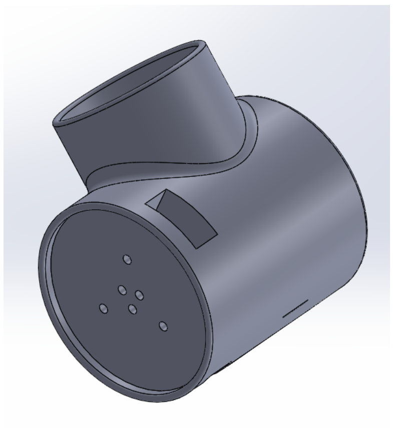


Figure 3.6: Link1 bottom view

Same as the previous link, the link1 has a special compartment for the stepper motor we are going to use inside of it with a special hole too for cables. This can be shown in the following image of the side view.

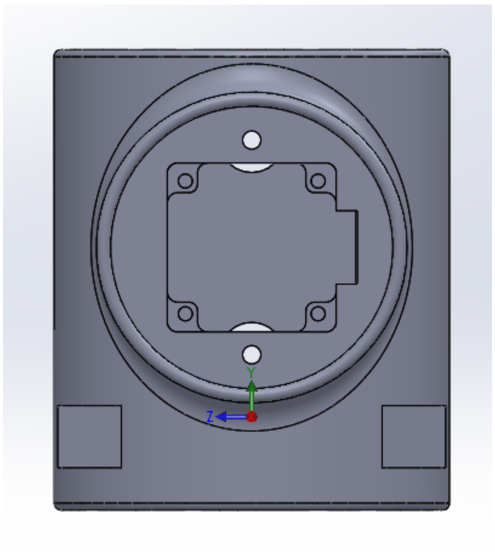


Figure 3.7: Link1 side view

And whit all this, we get a 700g wight for the link1.

3.2.2.3 Link2 :

This is composed principally of 2 parts connected with each other and moving along with each other so can be considered as a 1 link after all with a height of 326.99mm as it is shown in the following figure.

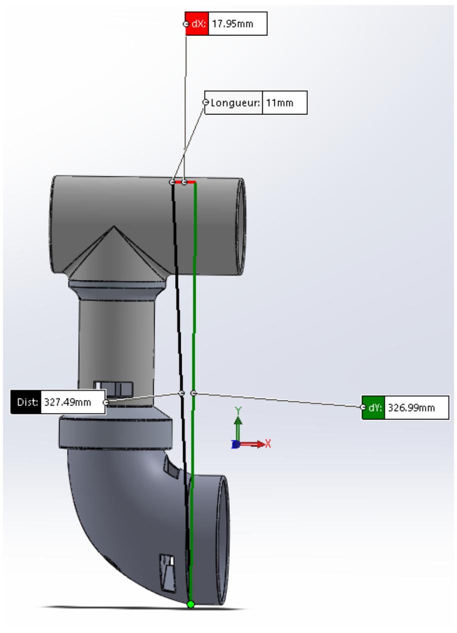


Figure 3.8: Link2 side view

And the same as the link1, from the bottom it has 6 holes for connection with the output of the link1 forming the joint 2 and from the upper side, we have the compartment of the stepper motor we are going to use a nema17 giving us a link of 570g weight as it is shown in this image.

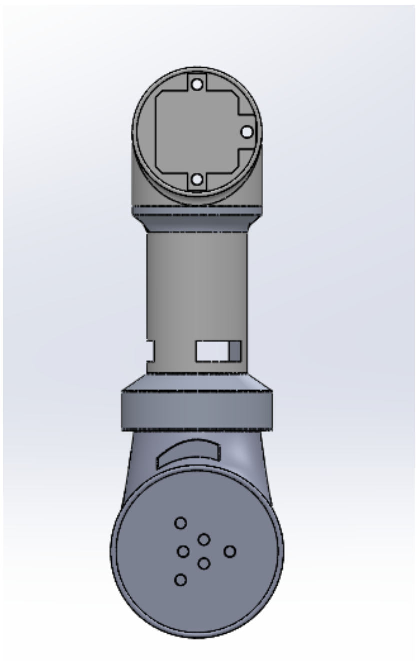


Figure 3.9: Link2 face view

3.2.2.4 Link3 :

This link is kind of the same as the previous link2, but with some small changes as the height is reduced to 292.01mm since it will carry less load and has less tension compared to the previous one, as shown in the figure.

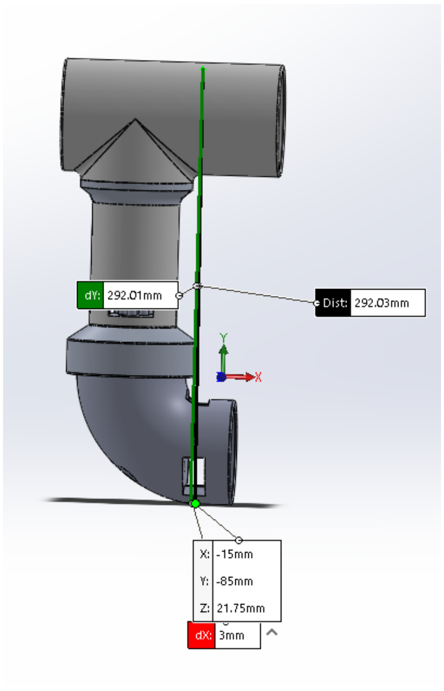


Figure 3.10: Link3 side view

At the same time, we also have a compartment for the stepper using a nema17 on the top of it but this time only 3 holes for the joint connection with the link2 since we just said it will have less tension on it, giving us a 400g weight.

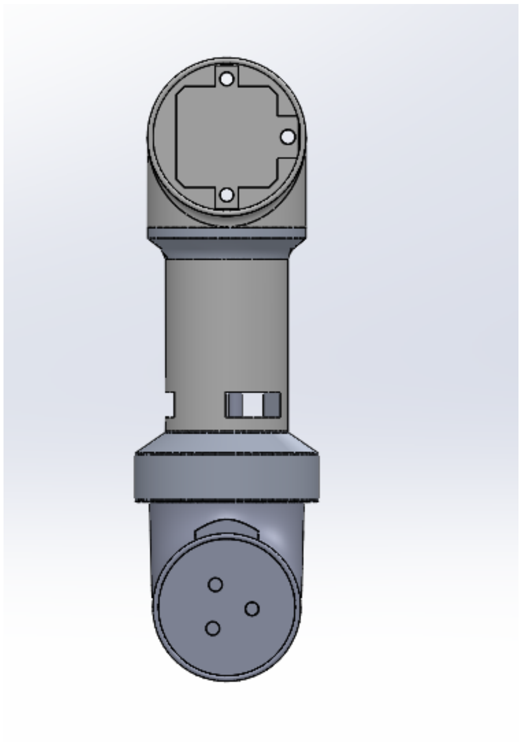


Figure 3.11: Link3 face view

The main idea behind the 3 or 6 holes is to distribute the load and tension in different points to prevent torsions in the system. So for the 3 first joints since they are the ones resisting much load we used 6 holes and for the oher 3 we only needed 3. The connection system will be easier to understand when we explain the gearbox conception.

3.2.2.5 Link4 :

For the link4, it is like a cylindre of 70mm diameter with a 147,72mm height. From the side we have the robot name BRAS-DEL, the bottom we have the 3 holes we use for connection with link3 forming the joint 4, and from the upper side we have the nema 17 compartment, giving us a 212g weight link.

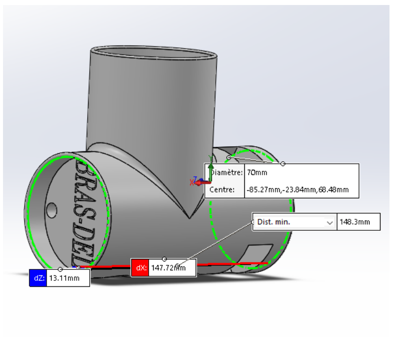


Figure 3.12: Link4 side view

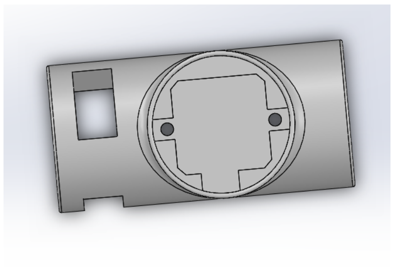


Figure 3.13: Link4 upper view

3.2.2.6 Link5 :

Same as before, we have a cylinder of 70mm diameter and 148mm height. It has the arm name from the side, the 3 holes for connection with link5 forming joint5 and the stepper motor nema 17 space from up with a total of 190g weight as can be shown in the following figures.

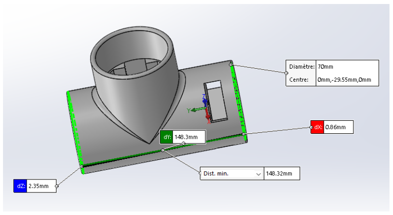


Figure 3.14: Link5 side view

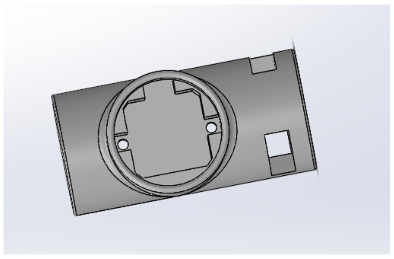


Figure 3.15: Link5 upper view

3.2.2.7 Link6 :

Which is the final link, we call it the effector holder since it will be holding from its upper side the hand or end effector of the robotic arm. Same as before, it is a cylinder of 70mm diameter and 50mm height. As the 2 figures show, it has from its upper side the servo motor compartment for end effector command and from the bottom side the 3 holes for connection with link5.

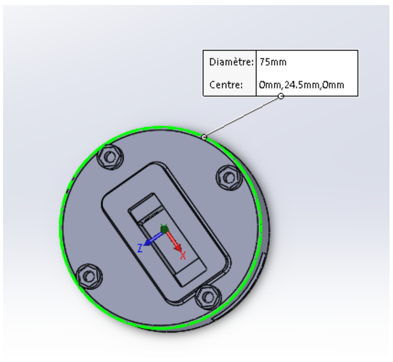


Figure 3.16: Link6 upper view

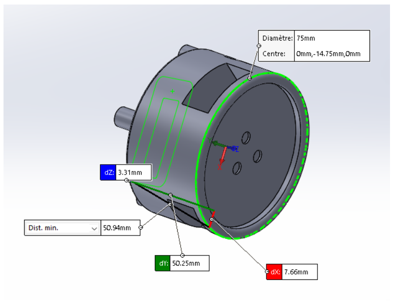


Figure 3.17: Link6 bottom view

This gives us a 61g weight for link6.

3.2.2.8 End effector :

For the end effector, we choose a prismatic movement gripper with 70mm fingers height for object picking. This will be connected with the link6 or the effector hold, giving us a effector of 50g weight.

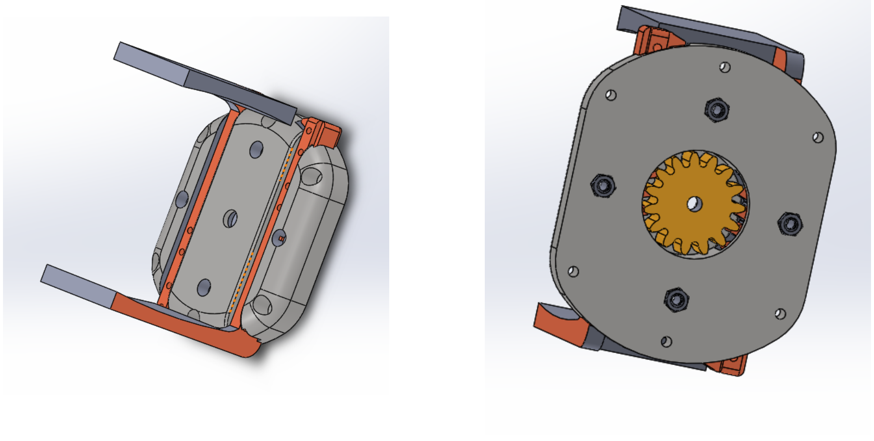


Figure 3.18: End effector design

3.2.3 Actuators

For the actuators choice, as we previously mentioned we are going to use stepper motors from two different families, nema 23 and nema 17 and for the end effector we choose a simple servo motor.

In our choice, we took into consideration 2 main points, firstly we chose the stepper motor since it is an open loop actuator, this will help us reduce the number of sensors that should be used because we don't need feedback information in the command process. Secondly, we chose for each family of nema the model that has the highest torque output inorder to simplify the work after when we need to increase the torque, having the simplest system possible. For the gripper, we chose the servo motor since it has an integrated feedback sensor, and we don't really need a high torque value for just grasping the desired object.

3.2.3.1 Nema 17 42HS48 :

The following figure shows the mechanical design and dimensions of the stepper motor.

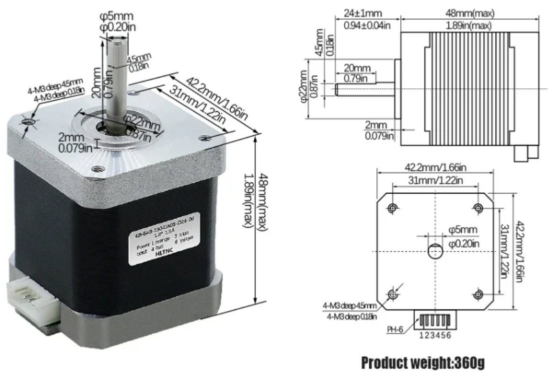


Figure 3.19: Nema 17 motor conception

And we have the following table, that shows the electrical characteristics of the motor :

Model	Step angle (°)	Nominal tension (V)	Nominal current (A)	Holding torque (Kg.cm)	Motor torque (g.cm max)	Motor weight (kg)
42HS48	1.8	4.2	1.5	5.5	280	0.36

Table 3.1: Nema 17 stepper motor characteristics

So for 4 joints, we chose to work with the nema 17 stepper motor with 0.5N.m torque.

3.2.3.2 Nema 23 57HS112 :

The following figure shows the mechanical design and dimensions of the stepper motor.

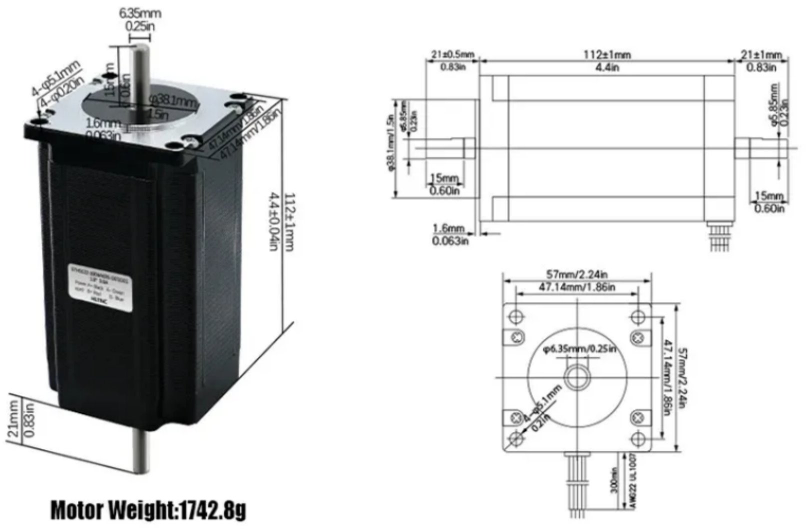


Figure 3.20: Nema 23 motor conception

And we have the following table, that shows the electrical characteristics of the motor :

Model	Step angle (°)	Nominal tension (V)	Nominal current (A)	Holding torque (Kg.cm)	Motor torque (g.cm max)	Motor weight (kg)
57hs112	1.8	2.45	3.5	28	1200	1.7

Table 3.2: Nema 23 stepper motor characteristics

For the first 2 joints, we work with the nema23 since they are the ones that will have to support more load so more torque is required, so they have a stepper motor of 2.7N.m torque.

3.2.3.3 Servo motor MG-90 :

We choose a simple servo motor reference MG-90 for the end effector command. The following image shows its conception :

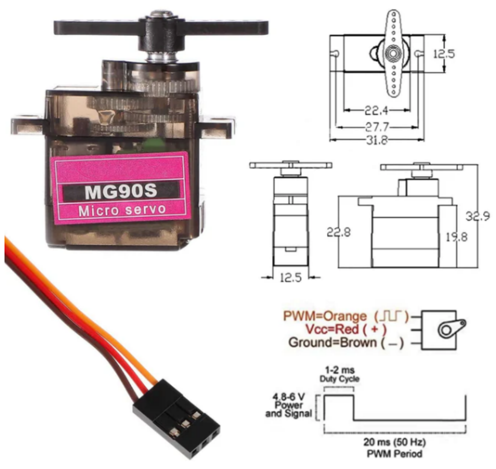


Figure 3.21: MG-90 servo motor

And this table, explain its important specifications :

Modèle	Weight	Max angle	Stall torque (Kg/cm)	Operating speed (sec/60°)	Temperature (C°)
MG-90	15g	180°	2.0	28	55

Table 3.3: Caractéristiques du moteur MG-90

3.2.4 Gearboxes

After explaining the conception of the joints and the motors choice, we should explain the chosen system for torque increasing. In our case we chose the planetary gearbox since it's easier to design and to realize in reality.

As it was explained in the general definitions chapter, it is composed of the 4 important elements :

3.2.4.1 The sun gear :

Which is the input of the movement so it is related to the axis of the motor. It is defined by its number of teeth, and the following figure shows a sun gear example of one of the gearboxes we designed.

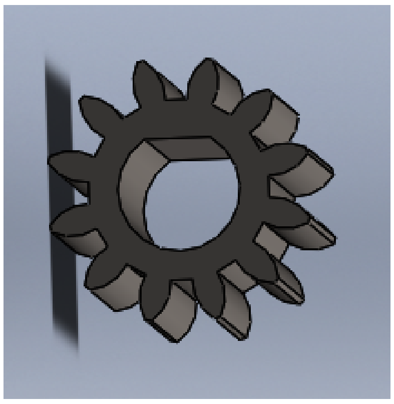


Figure 3.22: Sun gear design

3.2.4.2 The ring gear :

For our system, we chose it to be the one fixed so the output will be taken from the planetaries movement. The same as the sun gear, it is defined by its number of teeth and this figure shows as an example of a ring gear we designed.

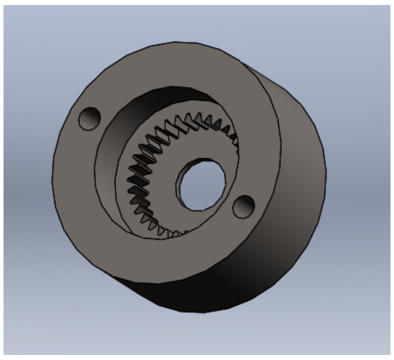


Figure 3.23: Ring gear design

3.2.4.3 The planetary gear :

For our system, we chose them to be the output of the movement. Using 3 similar planetaries to transfer the movement of the input sun gear and increase its torque. This figure shows an example of a planetary gear.

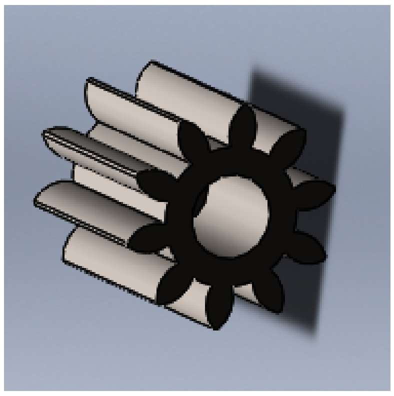


Figure 3.24: Planetary gear design

3.2.4.4 The carrier :

Which is going to set the axis of rotation for the planetaries so they turn around themselves, and at the same time they transfer their movement to the carrier which is going to be the output of the movement.

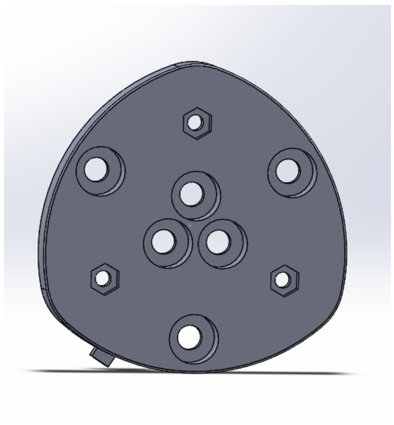


Figure 3.25: Carrier design

As we previously discussed about the holes, the output of the gearbox is going to be the carrier, and it is going to be related to the next link to the joint. So from it we might have 3 or 6 holes made for 6 axes that are going to be connected to the next link to transfer the movement from the gearbox to the next link.

By assembling all these together, we can make a 1 stage planetary gearbox as shown in the figure.

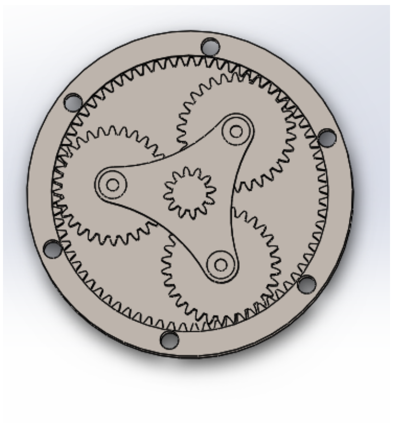


Figure 3.26: 1 stage planetary gearbox

In order to get better torque increase, we can work on a 2 stages gearbox, and that simply by connecting the stages so they get to be cascading. Making this will multiply the torque by itself, so by cascading a stage of 5:1 ratio with itself that will give us an output ratio of 25:1. The following figure shows and example of a 2 stages gearbox.

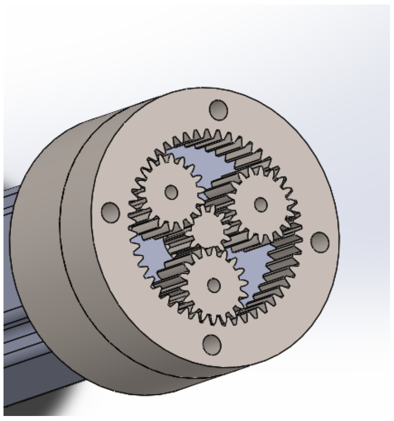


Figure 3.27: 2 stages planetary gearbox

Now in-order to make sure that the movement will be transferred to the next link so we create our joint, and to ensure that the structure of our gearbox remains fixed and no part of it slips out of it. We are going to introduce the second carrier of the system that will be connected to the 2nd stage of the gearbox, and the cover which is going to cover the gearbox and make sure that it staies compressed so nothing move out of it.

3.2.4.5 The second carrier :

The second carrier has the same form as the first one, he will have the 6 or 3 holes for connection with the next link, and in addition it will have a special part which is going to be used for joint position initialization with the limit switch.

This following figure shows the second carrier format, and we can see that it has a part getting out of it which is the one used in contact with the limit switch to know that we reached the initialization position.

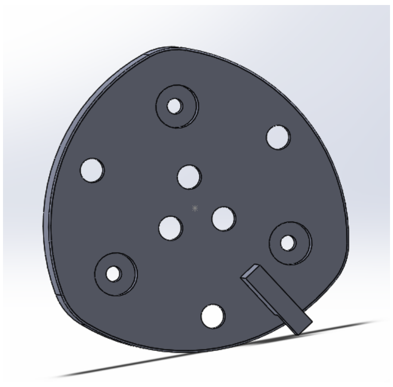


Figure 3.28: Second carrier

3.2.4.6 The cover :

Which is in a spherical shape, has big hole in the middle to permit the axis getting out of the holes of the second carrier to be fixed with the next link, it has 2 holes on the extremity which will be used to fix the gearbox with the link and the stepper and 2 little holes in which we are going to fix the limit switch in it's initialization position. The following image shows the design of the cover.

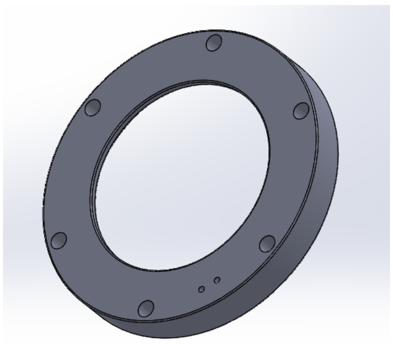


Figure 3.29: Gearbox cover

Based on what was just explained, the final structure of our gearbox of 2 stages and positioned limit switch will look like as shown in the figure.

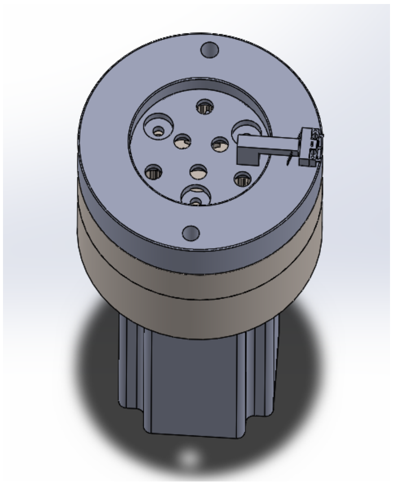


Figure 3.30: Full gearbox design

3.2.5 General assembly

After explaining each link, the actuators choice and the chosen system for torque increasing, we will go to the joint connections and general assembly of the robot. As previously explained, the link is formed by the connection of 2 links or rigid bodies C_{n-1} and C_n, in which the C_{n-1} will be called the parent link and the C_n called the child link. In the connection of this 2, we are going to use the gearbox to increase the motor torque fixed in the parent link, and the axis getting out of the second carrier will do the job of transmitting the movement to the child link.

To simplify the arm composition, we will have the following table to resume the links and their wights free from the motor then with added motors.

Link	Weight (g)	Motor used	Link final weight (g)
Base link	300	Nema 23	2000
Link 1	700	Nema 23	2400
Link 2	570	Nema 17	930
Link 3	400	Nema 17	760
Link 4	212	Nema 17	572
Link 5	190	Nema 17	550
Link 6	61	MG-90	76
End effector	50	–	–

Table 3.4: Robot links weight

Then we have the following table that will resume the joints, the parent and child link for each one, the motor used, and the gearbox ratio used for each one.

Joint number	Parent link	Child link	Motor used	Motor torque (N.m)	Gearbox ratio	Final torque (N.m)
Joint 0	Base link	Link1	Nema 23	2.7	49:1	132.3
Joint 1	Link1	Link2	Nema 23	2.7	25:1	67.5
Joint 2	Link2	Link3	Nema 17	0.5	36:1	18
Joint 3	Link3	Link4	Nema 17	0.5	25:1	12.5
Joint 4	Link4	Link5	Nema 17	0.5	14:1	7
Joint 5	Link5	Link6	Nema 17	0.5	5:1	2.5

Table 3.5: Robot Joints Structure

Based on what's motioned in the table and previously explained, we have got a full 6 DOF robotic arm, of a 1m height and 7338g load with a max payload of 3Kg within a security margin. And we have the following figure that shows the final assembly of the robot.

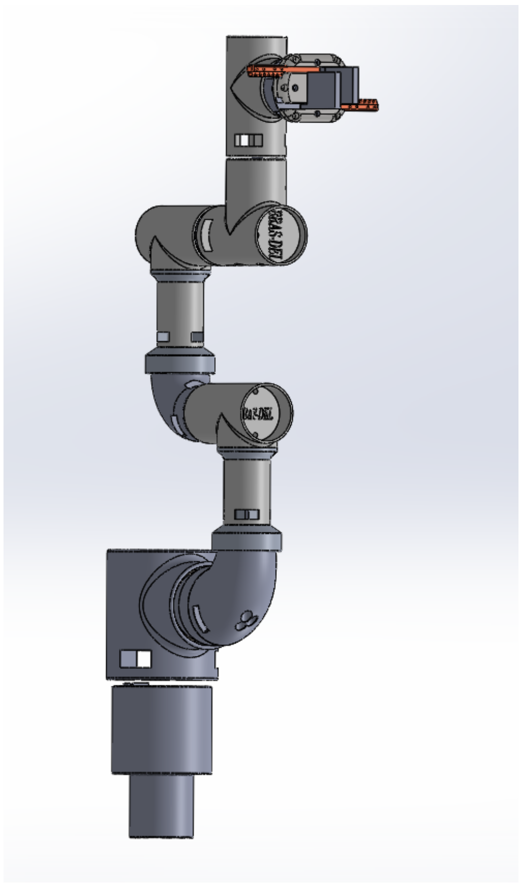
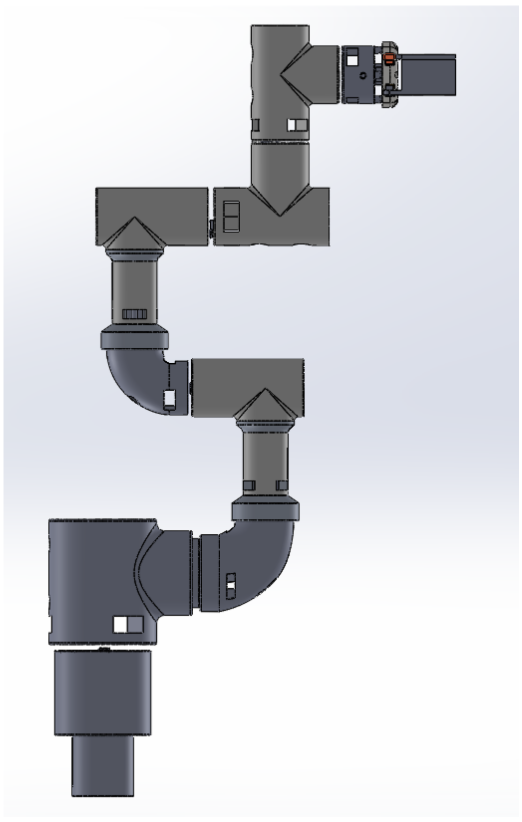



Figure 3.31: Final robot assembly

3.2.6 BRAS-DEL URDF file

3.2.6.1 The unified robotics description format :

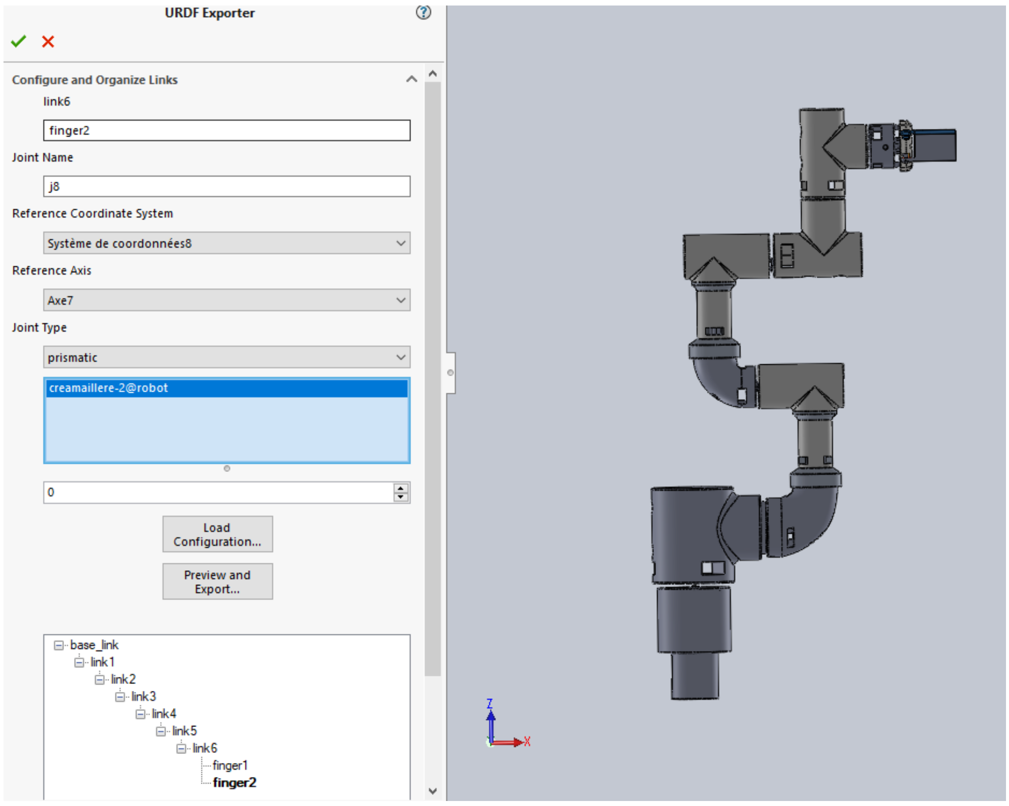
The unified robotics description format (URDF) is an extensible markup language (XML) file type that includes the physical description of a robot. It is essentially a 3D model with information around joints, motors, mass, etc. The files are then run through the Robot Operating System (ROS). The data from the file informs the human operator what the robot looks like and is capable of before they begin operating the robot[38].

3.2.6.2 URDF generation :

Working with SolidWorks, it allows the generation of a basic URDF file using the "SW2URDF" tool. This tool enables the definition of each robot link in hierarchical order (parent-child), specifying the joint types, rotation axes, origins, and the working environment.
Using this SOLIDWORKS extension, we can generate our robot URDF file by specifying the axes of rotations, the relations parent child between links to create the joints and the limits of the movement for each joint.

3.2.6.2.1 Parent child relation and axis of rotations : First step in URDF generation using the SolidWorks extension is to setup the parent child relation in order to create the joints. For this case we should remember that the parent represent the link C_{n-1} and the child link is the next one C_n creating the joint J_n.

At the same step, we are going to setup the axis of rotation and the frames (noted coordinate systems in SolidWorks) for each joint.
The following figure represents the first step using the urdf exporter.



URDF Exporter

Configure and Organize Links

link6

finger2

Joint Name

j8

Reference Coordinate System

Système de coordonnées8

Reference Axis

Axe7

Joint Type

prismatic

creamaillere-2@robot

0

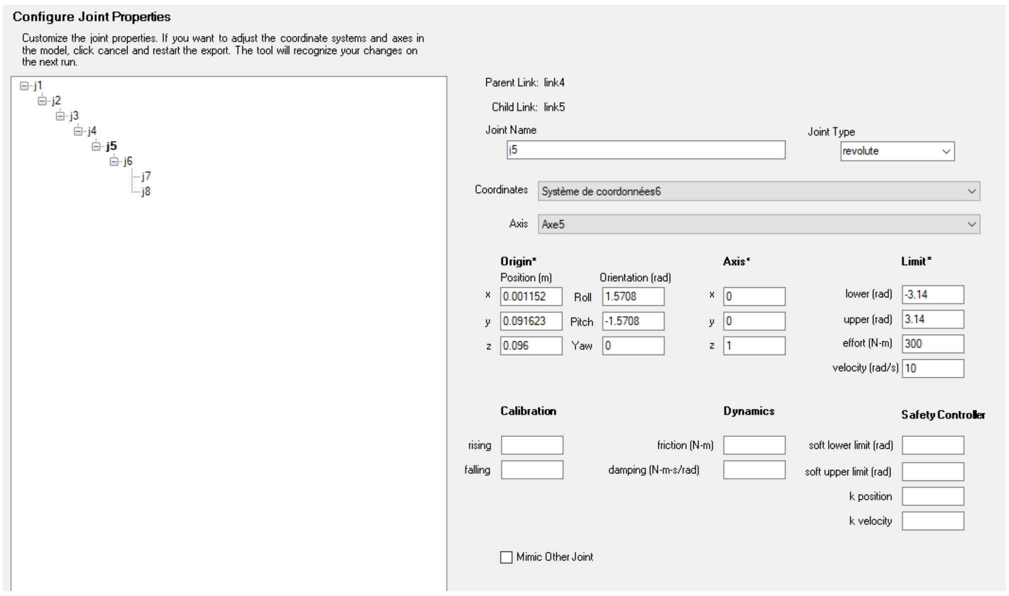
Load Configuration...

Preview and Export...

- base_link
  - link1
    - link2
      - link3
        - link4
          - link5
            - link6
              - finger1
              - **finger2**

Figure 3.32: Step1 of Urdf exportation

After that, we will have to set the movement constrains for each joint, so each joint has its lower and higher limit of rotation alongside a maximum velocity possible. And that can be shown in the following image.



Configure Joint Properties

Customize the joint properties. If you want to adjust the coordinate systems and axes in the model, click cancel and restart the export. The tool will recognize your changes on the next run.

- j1
  - j2
    - j3
      - j4
        - **j5**
          - j6
            - j7
            - j8

Parent Link: link4

Child Link: link5

Joint Name: j5

Joint Type: revolute

Coordinates: Système de coordonnées6

Axis: Axe5

Origin*

Position (m)		Orientation (rad)	
x	0.001152	Roll	1.5708
y	0.091623	Pitch	-1.5708
z	0.096	Yaw	0

Axis*

x	0
y	0
z	1

Limit*

lower (rad)	-3.14
upper (rad)	3.14
effort (N-m)	300
velocity (rad/s)	10

Calibration

rising	
falling	

Dynamics

friction (N-m)	
damping (N-m-s/rad)	

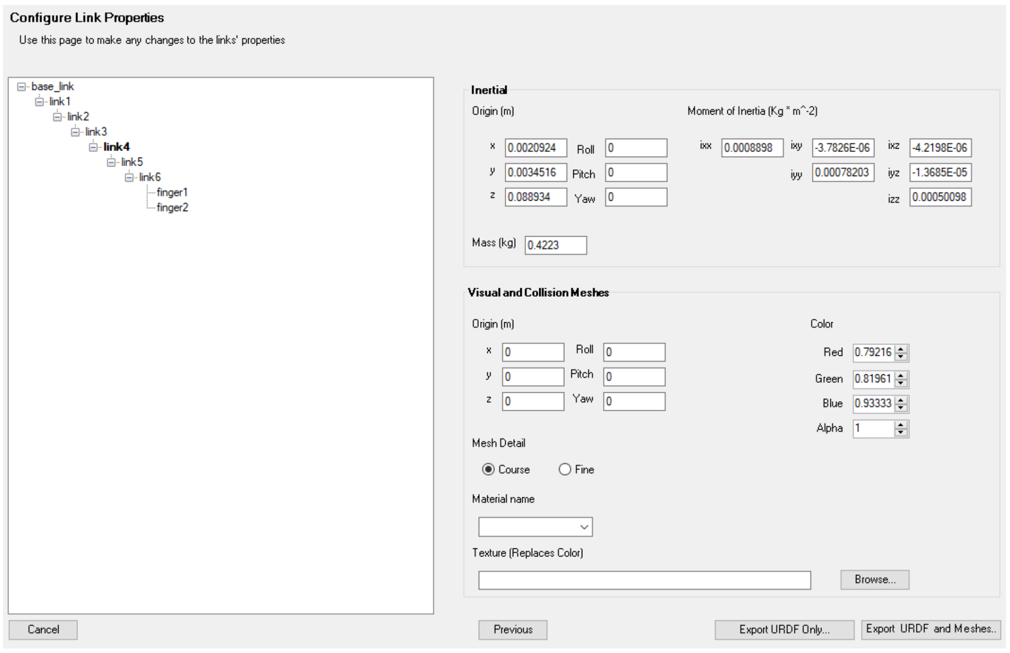
Safety Controller

soft lower limit (rad)	
soft upper limit (rad)	
k position	
k velocity	

☐ Mimic Other Joint

Figure 3.33: Step2 of Urdf exportation

Then we have the last step in which u check for any mistakes. We should make sure that for each link a mass is attributed otherwise we didn't set it correctly before saving the required files as shown in the next image.



Configure Link Properties

Use this page to make any changes to the links' properties

- base_link
  - link1
    - link2
      - link3
        - **link4**
          - link5
            - link6
              - finger1
              - finger2

Inertial

Origin (m)

x	0.0020924	Roll	0
y	0.0034516	Pitch	0
z	0.088934	Yaw	0

Moment of Inertia (Kg * m^-2)

ixx	0.0008898	ixy	-3.7826E-06	ixz	-4.2198E-06
		iyy	0.00078203	iyz	-1.3685E-05
				izz	0.00050098

Mass (kg) 0.4223

Visual and Collision Meshes

Origin (m)

x	0	Roll	0
y	0	Pitch	0
z	0	Yaw	0

Color

Red	0.79216
Green	0.81961
Blue	0.93333
Alpha	1

Mesh Detail

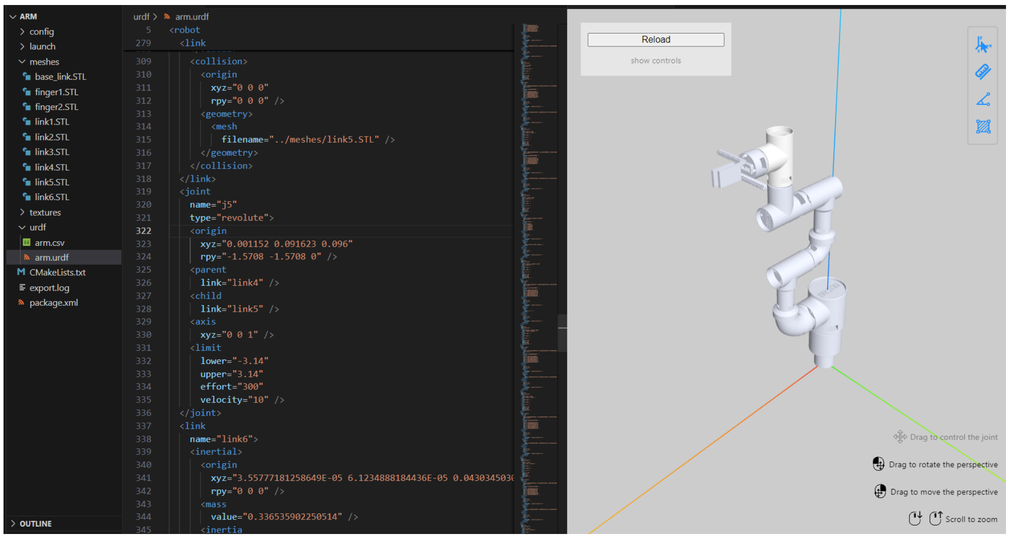
[✓ Course] Fine

Material name

Texture (Replaces Color)

Figure 3.34: Step3 of Urdf exportation

At the end of it, we can use some of Vscode extensions in order to check if our URDF is generated correctly. The following figure shows at the left side of it the generated folder from the Urdf exporter which contains the meshes folder and the urdf file, in the middle we have the URDF file of the robot and at the right part we have a visualization of the urdf using the urdf viewer extension.



ARM
- config
- launch
- meshes
  - base_link.STL
  - finger1.STL
  - finger2.STL
  - link1.STL
  - link2.STL
  - link3.STL
  - link4.STL
  - link5.STL
  - link6.STL
- textures
- urdf
  - arm.csv
  - arm.urdf
- CMakeLists.txt
- export.log
- package.xml

urdf > arm.urdf

```
<robot
  <link
      <collision>
        <origin
          xyz="0 0 0"
          rpy="0 0 0" />
        <geometry>
          <mesh
            filename="../meshes/link5.STL" />
        </geometry>
      </collision>
  </link>
  <joint
    name="j5"
    type="revolute">
    <origin
      xyz="0.001152 0.091623 0.096"
      rpy="-1.5708 -1.5708 0" />
    <parent
      link="link4" />
    <child
      link="link5" />
    <axis
      xyz="0 0 1" />
    <limit
      lower="-3.14"
      upper="3.14"
      effort="300"
      velocity="10" />
  </joint>
  <link
    name="link6">
    <inertial>
      <origin
        xyz="3.55777181258649E-05 6.1234888184436E-05 0.0430345030
        rpy="0 0 0" />
      <mass
        value="0.336535902250514" />
      <inertia
```

OUTLINE
Reload
show controls
Drag to control the joint
Drag to rotate the perspective
Drag to move the perspective
Scroll to zoom

Figure 3.35: Urdf visualization in Vscode

3.3 Robot modeling

After designing our robotic model, and before being able to use it in reality, we should pass the modeling of our robot using the technics we previously explained in the general definitions section.

For our case of use, we only need the Forward geometrical modeling , we don't need to go through the kinematic or the inverse geometric one since our command methods which will be explained later does not require it and the dynamic modeling was previously made in the part where we discussed the torques and gearbox ratios for each joints.

3.3.1 Forward Geometric modeling

For the forward geometric modeling, we will be using the DH parameters that was explained previously and for that we should start by setting up the frames.
The following figure shows the joint frames we chose with respect to the DH convention.

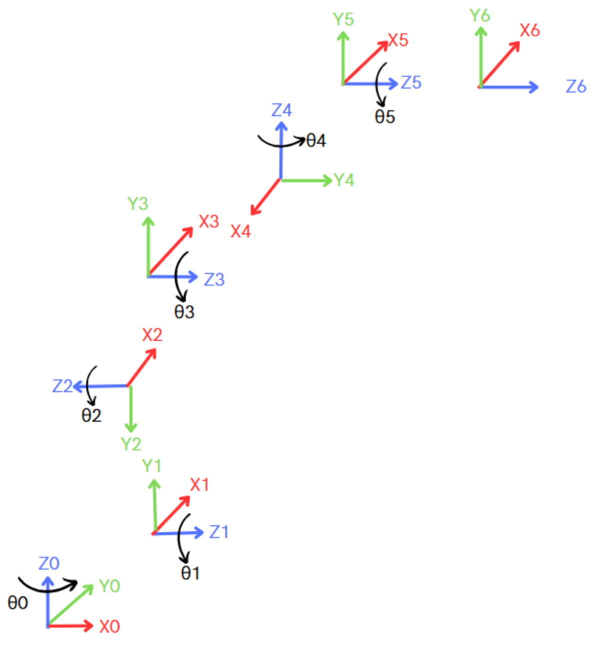


Figure 3.36: Robot joint frames

From these frames and by following the DH parameters extraction steps, we got the following table that resumes the DH parameters for BRAS-DEL robot.

Joint (i)	θ_i (rad)	d_i (mm)	a_i (mm)	α_i (rad)
0	θ_0	104	0	$-\pi/2$
1	θ_1	11	206	$\pi/2$
2	θ_2	9	172	$\pi/2$
3	θ_3	72	0	$-\pi/2$
4	θ_4	65	0	$-\pi/2$
5	θ_5	48	0	0

Table 3.6: Extracted DH parameters

Now that we obtained DH values, we should calculates the transformation matrix for each joint then the global one for the robot.

$$T_1^0 = \begin{bmatrix} \cos\theta_0 & 0 & \sin\theta_0 & 0 \\ \sin\theta_0 & 0 & -\cos\theta_0 & 0 \\ 0 & 1 & 0 & 107 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^1 = \begin{bmatrix} \cos\theta_1 & 0 & -\sin\theta_1 & 241\cos\theta_1 \\ \sin\theta_1 & 0 & \cos\theta_1 & 241\sin\theta_1 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^2 = \begin{bmatrix} \cos\theta_2 & 0 & -\sin\theta_2 & 223\cos\theta_2 \\ \sin\theta_2 & 0 & \cos\theta_2 & 223\sin\theta_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_4^3 = \begin{bmatrix} \cos\theta_3 & 0 & -\sin\theta_3 & 0 \\ \sin\theta_3 & 0 & \cos\theta_3 & 0 \\ 0 & -1 & 0 & 96 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_5^4 = \begin{bmatrix} \cos\theta_4 & 0 & \sin\theta_4 & 0 \\ \sin\theta_4 & 0 & -\cos\theta_4 & 0 \\ 0 & 1 & 0 & 66 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_6^5 = \begin{bmatrix} \cos\theta_5 & 0 & -\sin\theta_5 & 0 \\ \sin\theta_5 & 0 & \cos\theta_5 & 0 \\ 0 & 1 & 0 & 67 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After calculating the transformation matrix for each joint, we can calculate the overall trasnformation matrix T_6^0 where :

$$T_6^0 = T_1^0 \cdot T_2^1 \cdot T_3^2 \cdot T_4^3 \cdot T_5^4 \cdot T_6^5$$

This will give us the full matrix :

$$T_6^0 = \begin{bmatrix} A_{11} & A_{21} & A_{31} & A_{14} \\ A_{12} & A_{22} & A_{32} & A_{24} \\ A_{13} & A_{23} & A_{33} & A_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With :

$$A_{11} = (-\sin\theta_0 \sin\theta_4 + \cos\theta_0 \cos\theta_4 \cos(\theta_1 - \theta_2 + \theta_3)) \cos\theta_5 - \cos\theta_0 \sin(\theta_1 - \theta_2 + \theta_3) \sin\theta_5$$

$$A_{12} = (\sin\theta_0 \sin\theta_4 - \cos\theta_0 \cos\theta_4 \cos(\theta_1 - \theta_2 + \theta_3)) \sin\theta_5 - \cos\theta_0 \sin(\theta_1 - \theta_2 + \theta_3) \cos\theta_5$$

$$A_{13} = \sin\theta_0 \cos\theta_4 + \cos\theta_0 \sin\theta_4 \cos(\theta_1 - \theta_2 + \theta_3)$$

$$\begin{aligned} A_{14} = {} & 67 \sin\theta_0 \cos\theta_4 + 96 \sin\theta_0 + 67 \sin\theta_4 \cos\theta_0 \cos(\theta_1 - \theta_2 + \theta_3) \\ & - 66 \sin(\theta_1 - \theta_2 + \theta_3) \cos\theta_0 + 241 \cos\theta_0 \cos\theta_1 + 223 \cos\theta_0 \cos(\theta_1 - \theta_2) \end{aligned}$$

$$A_{21} = (\sin\theta_0 \cos\theta_4 \cos(\theta_1 - \theta_2 + \theta_3) + \cos\theta_0 \sin\theta_4) \cos\theta_5 - \sin\theta_0 \sin(\theta_1 - \theta_2 + \theta_3) \sin\theta_5$$

$$A_{22} = -(\sin\theta_0 \cos\theta_4 \cos(\theta_1 - \theta_2 + \theta_3) + \cos\theta_0 \sin\theta_4) \sin\theta_5 - \sin\theta_0 \sin(\theta_1 - \theta_2 + \theta_3) \cos\theta_5$$

$$A_{23} = \sin\theta_0 \sin\theta_4 \cos(\theta_1 - \theta_2 + \theta_3) - \cos\theta_0 \cos\theta_4$$

$$\begin{aligned} A_{24} = {} & 67 \sin\theta_0 \sin\theta_4 \cos(\theta_1 - \theta_2 + \theta_3) - 66 \sin\theta_0 \sin(\theta_1 - \theta_2 + \theta_3) \\ & + 241 \sin\theta_0 \cos\theta_1 + 223 \sin\theta_0 \cos(\theta_1 - \theta_2) \\ & - 67 \cos\theta_0 \cos\theta_4 - 96 \cos\theta_0 \end{aligned}$$

$$A_{31} = \sin\theta_5 \cos(\theta_1 - \theta_2 + \theta_3) + \sin(\theta_1 - \theta_2 + \theta_3) \cos\theta_4 \cos\theta_5$$

$$A_{32} = -\sin(\theta_1 - \theta_2 + \theta_3)\cos\theta_4 \sin\theta_5 + \cos(\theta_1 - \theta_2 + \theta_3)\cos\theta_5$$

$$A_{33} = \sin\theta_4 \sin(\theta_1 - \theta_2 + \theta_3)$$

$$A_{34} = 241\sin\theta_1 + 67\sin\theta_4 \sin(\theta_1 - \theta_2 + \theta_3) + 223\sin(\theta_1 - \theta_2) + 66\cos(\theta_1 - \theta_2 + \theta_3) + 107$$

3.4 Robot Realization

3.4.1 3D printing

3D printing, also known as additive manufacturing, is a process of creating three-dimensional objects by building them layer by layer from a digital 3D model. Unlike traditional manufacturing, 3D printing adds material only where needed, making it efficient for prototyping, customization, and even final product production.

3.4.1.1 Materials :

During the printing process, we needed 2 types of filament where each one is going to be used for special pieces of the robot due to the difference in characteristics of each one of them.

3.4.1.1.1 PLA : Polylactic Acid or PLA, and it is one of the most popular and widely used 3D printing materials. It is commonly used for simple structures with no risk of high temperatures is possible. Using this PLA, we are going to print only the parts that are not in direct contact with the released heat from the stepper motors during the work so basically only the links.

3.4.1.1.2 ABS : Acrylonitrile Butadiene Styrene or ABS , durable, strong, and impact-resistant thermoplastic commonly used in 3D printing, especially for functional parts. Its main characteristic that it resists the high temperatures during the work, so for that we are going to use it for gearbox parts printing only.

The following table shows the difference in setup between the 2 types we are going to use :

Filament type	Nozzle temperature	Bed temperature	Heat resistance
PLA	$200°C$	$60°C$	High
ABS	$250°C$	$90°C$	LOW

Table 3.7: Filament parameters

3.4.2 Required tools

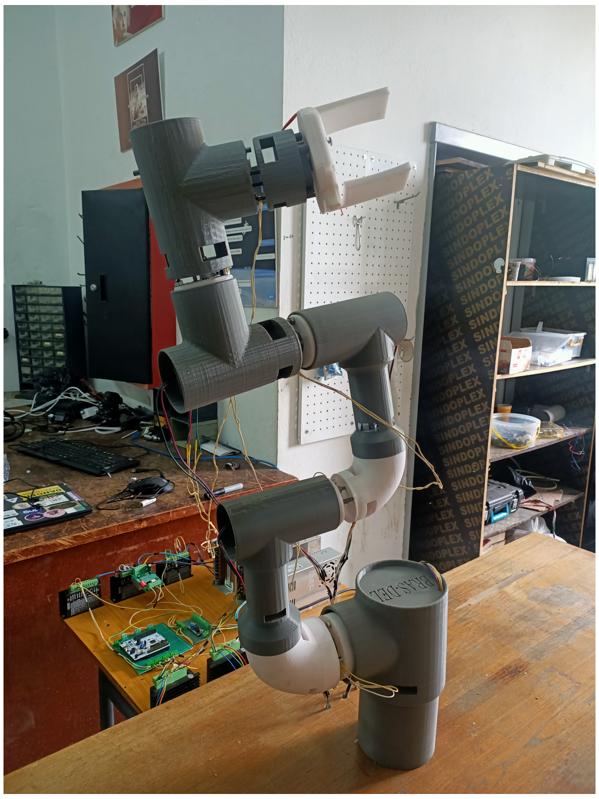
During the assembly, we needed many mechanical tools that can be resumed in the following table :

Tool	Quantity
Screw M4	37
Screw M5	6
Screw M6	21
Nut M4	37
Nut M5	6
Nut M6	54
Threaded rod M6	2 meters

Table 3.8: Mechanical tools required

3.4.3 Real life Assembly

After putting together all the previously explained work from printing the different links and gearboxes of the robot to assembling everything, we could get the final assembly of the robot in real life shown in the following figures.



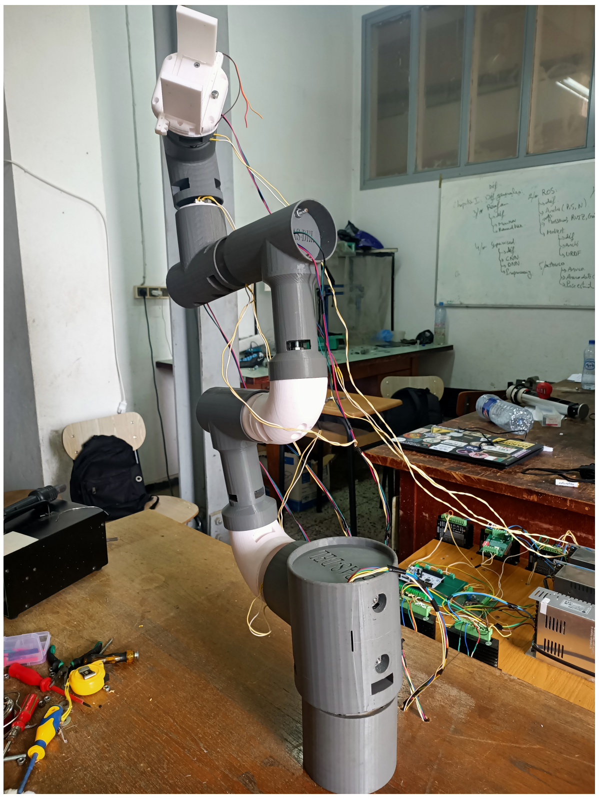


Figure 3.37: Final robot assembly

3.5 Conclusion

By the end of the chapter, all the essential parts of the mechanical work we done were explained to facilitate readers to understand the work we have done and the reason behind every step we took.

In addition to that, this was a crucial part for our work since it is intended for industry and the main goal of this work is to present the idea as a startup idea. So it was a very hard part we suffered a lot during the conception since the design was made by us, and even during the printing we had to face severe problems such as the luck of material and 3D printers, so we had to learn how to fix the available ones in FABLAB and the electronic department before being able to start the real life assembly and which took from us too much time.

But even though all the problems we faced, we were able to deliver at the end a prototype for our project in order to simulate and test the movements and be able to present it to the jury the D-day.

Chapter 4

ROS & Robodk Based Command and Electronic Integration

4.1 Introduction

After introducing our mechanical design and conception of our robotic arm, we will jump now to the main part which includes the electrical circuit and design to make the previously explained work true in reality.

In addition to that we are going to explain the 2 main approaches we worked on for the robot command, we did rely on the most used based commands in robotics and industry as explained in the definition part, but this time we are going to get through more details where we're going to explain each step of the work, the perfect setup steps required for the perfect results.

4.2 Hardware and circuit design

4.2.1 Microcontroller

In our robotic arm, we used an **STM32 Nucleo-64 F446 RE** development board as the main controller. The board served as an interface between the ROS or RoboDK system and the physical hardware. It was responsible for receiving high-level motion commands and generating low-level control signals to drive six stepper motors. Additionally, it monitored six input limit switches to ensure accurate homing of the joints. The STM32's powerful microcontroller, flexible I/O options, and built-in debugging capabilities made it an ideal choice for real-time robotic control applications.

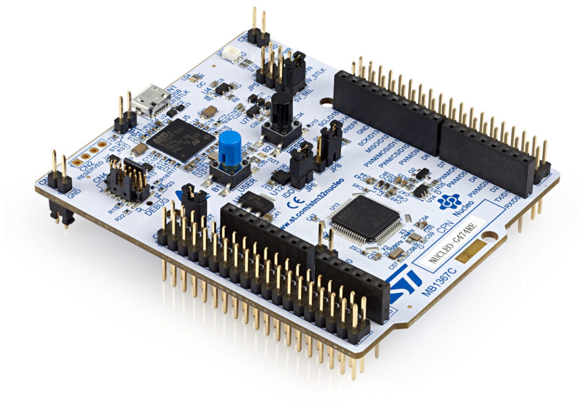


Figure 4.1: STM32 Nucleo 64 F446RE[39].

Feature	**Specification**
Core	ARM Cortex-M4 @ 180 MHz with FPU and DSP
Performance	225 DMIPS (Dhrystone 2.1)
Flash Memory	512 KB
SRAM	128 KB + 4 KB Backup SRAM
GPIOs	Up to 114 I/O pins (5V tolerant, 90 MHz capable)
Timers	17 total (incl. 2 watchdogs, 2×32-bit, PWM support)
Communication Interfaces	4×USART, 2×UART, 4×SPI, 4×I2C, 2×CAN, USB FS/HS
Debug Interfaces	SWD, JTAG, ETM Trace Macrocell
Package	LQFP64 or LQFP48
Operating Voltage	1.7V to 3.6V
External Interface Support	FSMC for SRAM/NOR/SDRAM, QuadSPI

Table 4.1: STM32 nucleo-64 F446 RE features

4.2.2 Motors and Drivers

Our robotic arm is actuated using a total of six stepper motors: four **Nema 17 (42HS48)** motors(used for joints 2,3,4 & 5) shown in figure 3.19, and two higher-torque **Nema 23 (57HS112)** motors (used for joint 0 & 1) also shown in figure 3.20, which require greater torque.

Each motor is driven using a **TB6600 stepper motor driver**, which supports external control using **STEP** and **DIR** signals from the STM32 microcontroller. The **STEP pin** receives a series of digital pulses that determine the number of steps the motor takes, while the **DIR pin** determines the direction of rotation (high for one direction, low for the opposite). The following connections are made for each TB6600:

- **DIR+** is connected to a GPIO pin on the STM32 (for setting the motor direction).
- **PUL+ (STEP+)** is connected to another GPIO pin (for sending step pulses).
- **DIR-**, **PUL-**, and **ENA-** are all connected to GND.

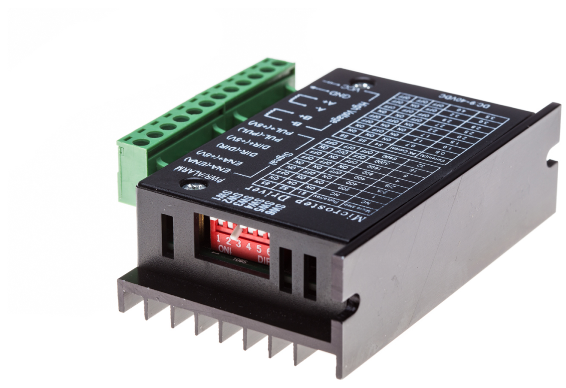


Figure 4.2: Drive TB6600[40].

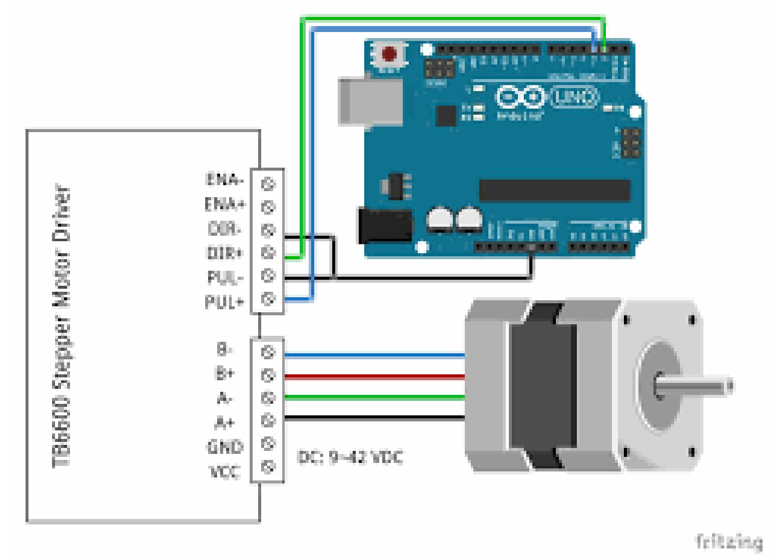


Figure 4.3: Stepper, drive and microcontroller wiring[41].

To achieve high positioning accuracy and smooth motion, the TB6600 drivers are configured for **3200 microsteps per revolution** by adjusting the DIP switches on the driver. This allows for finer angular resolution in our movement.
Furthermore, the current limit settings on the TB6600 are configured to the **maximum allowable current** supported by each motor model. This ensures the stepper motors operate with maximum available torque and performance.

4.2.3 Limit Switches

To home and init each joint of our robotic arm, we used six KW10 micro limit switches. Each switch was configured for Normally Open (NO) logic, meaning the circuit is open (inactive) when not pressed and closed (active) when pressed.

The wiring for each switch is as follows:

- **Common (C)** pin is connected to VCC (3.3V or 5V).
- **Normally Open (NO)** pin is connected to a GPIO input pin on the STM32.
- A **pull-down resistor** is connected between the GPIO pin and GND to ensure a defined LOW state when the switch is open.

Logic in code : When the limit switch is pressed, the NO contact closes and the GPIO input pin receives HIGH (VCC). so :

- `HIGH` signal = switch is **activated** (pressed)
- `LOW` signal = switch is **inactive** (not pressed)

4.2.4 Power supplies

To power our whole robotic arm we used 3 power supplies and based on tables 3.1 and 3.2 by respecting the maximal current of each stepper motor we powered the system like this :

AC Input	**DC Output**	**Number of Outputs**	**Powers**
220V AC	24V / 20A	3 Outputs	Joint 3 (NEMA 17), Joint 4 (NEMA 17), Joint 5 (NEMA 17)
220V AC	24V / 20A	3 Outputs	Joint 2 (NEMA 17), VCC (for limit switches), Common GND(for steppers and limit switches)
220V AC	36V / 10A	3 Outputs	Joint 0 (NEMA 23), Joint 1 (NEMA 23)

Table 4.2: Summary of power supplies and their assignments

Figure 4.4: The first power supply

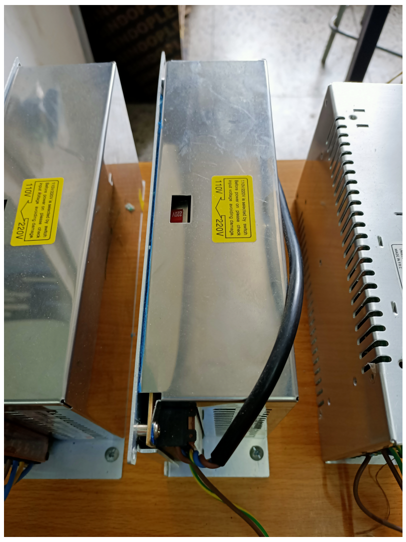


Figure 4.5: The second power supply

Figure 4.6: The third power supply

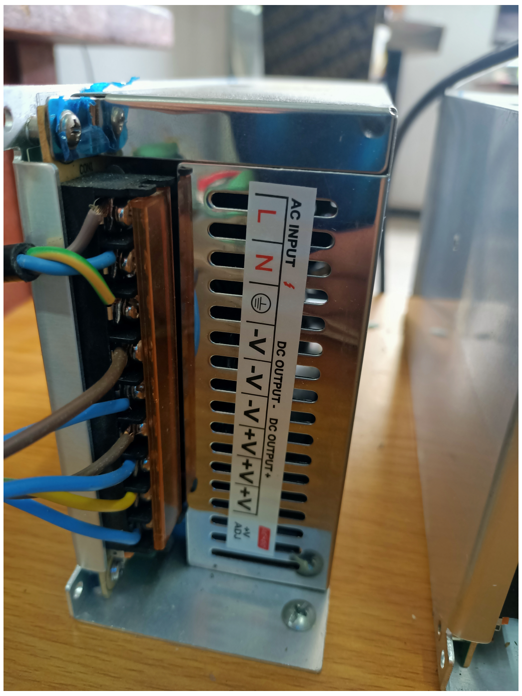


Figure 4.7: Power supply inputs-outputs

4.2.5 Circuit Design

4.2.5.1 Final Circuit Design Summary

The complete wiring design integrating six stepper motors and six limit switches, controlled and monitored by the STM32 microcontroller. Below is an overview of the major components and their interconnections:

1. **Stepper Motors (6 Total):**
   - Each motor is controlled via a TB6600 driver.
   - Two control signals per driver are used: `STEP+` and `DIR+`, connected to STM32 GPIO pins.
   - The `STEP-`, `DIR-`, and `ENA-` pins of each driver are connected to **common GND**.
2. **Limit Switches (6 Total):**
   - Each switch is wired in **Normally Open (NO)** configuration.
   - The `Common (COM)` terminal of each switch is connected to a shared **VCC** line.
   - The `NO` terminal is connected to a dedicated STM32 GPIO input pin.
   - A pull-down resistor is used between each GPIO pin and **GND** to stabilize the logic level.
3. **Power and Logic Reference:**
   - A shared **VCC** line is used to supply the limit switch COM terminals and optionally ENA+ if required.
   - A shared **GND** is used for all `ENA-`, `DIR-`, `STEP-` terminals and for GPIO ground reference.

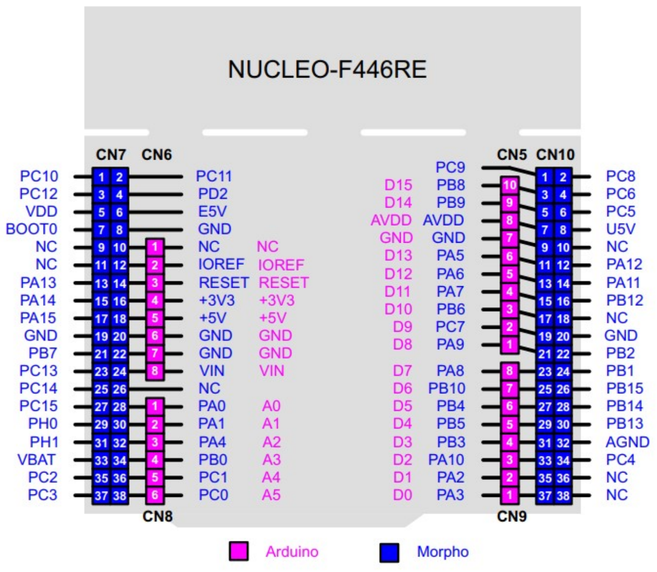

NUCLEO-F446RE
CN7 CN6
PC10 PC12 VDD BOOT0 NC NC PA13 PA14 PA15 GND PB7 PC13 PC14 PC15 PH0 PH1 VBAT PC2 PC3
PC11 PD2 E5V GND NC IOREF RESET +3V3 +5V GND GND VIN NC PA0 PA1 PA4 PB0 PC1 PC0
NC IOREF RESET +3V3 +5V GND GND VIN A0 A1 A2 A3 A4 A5
CN8
CN5 CN10
PC9 PB8 PB9 AVDD GND PA5 PA6 PA7 PB6 PC7 PA9 PA8 PB10 PB4 PB5 PB3 PA10 PA2 PA3
D15 D14 AVDD GND D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
PC8 PC6 PC5 U5V NC PA12 PA11 PB12 NC GND PB2 PB1 PB15 PB14 PB13 AGND PC4 NC NC
CN9
Arduino
Morpho

Figure 4.8: STM32 pinout[42].

And now based on the STM32 F446 RE shown in figure 4.8, here's a table that illustrates the whole wiring and pins assignments.

Joint (Motor)	**DIR_PIN**	**STEP_PIN**	**LIMIT_PIN**
G5 (Joint 5)	PC12	PC10	PC8
G4 (Joint 4)	PA14	PA13	PC6
G3 (Joint 3)	PB7	PA15	PA11
G2 (Joint 2)	PC3	PC2	PB15
G1 (Joint 1)	PB0	PA4	PB13
G0 (Joint 0)	PA1	PA0	PC4

Table 4.3: STM32 GPIO Pin Assignments for Stepper Control and Limit Switches

4.2.5.2 Real life electrical Assembly

For the real-world assembly of our robotic arm control system, we carefully organized the hardware components in a compact and efficient setup that include :

- **3 power supplies**.
- **6 TB6600 stepper drivers**.
- **VCC & common GND power board**.
- **STM32 control board**.

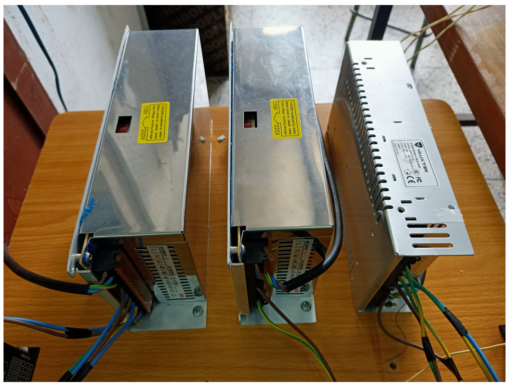


Figure 4.9: Power supplies

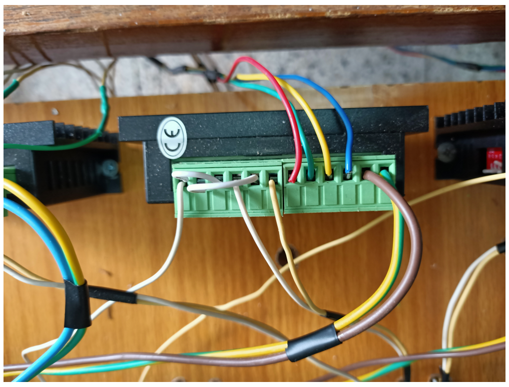


Figure 4.10: Driver TB6600 setup

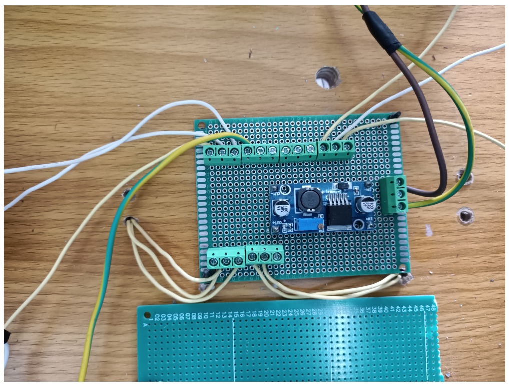


Figure 4.11: VCC & common GND board

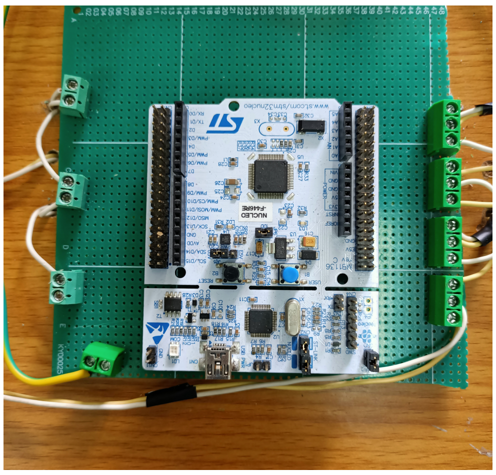


Figure 4.12: STM32 control board

All connections were made with proper cable management, ensuring safe, stable operation of the motors and switches. The full assembled hardware are shown below :

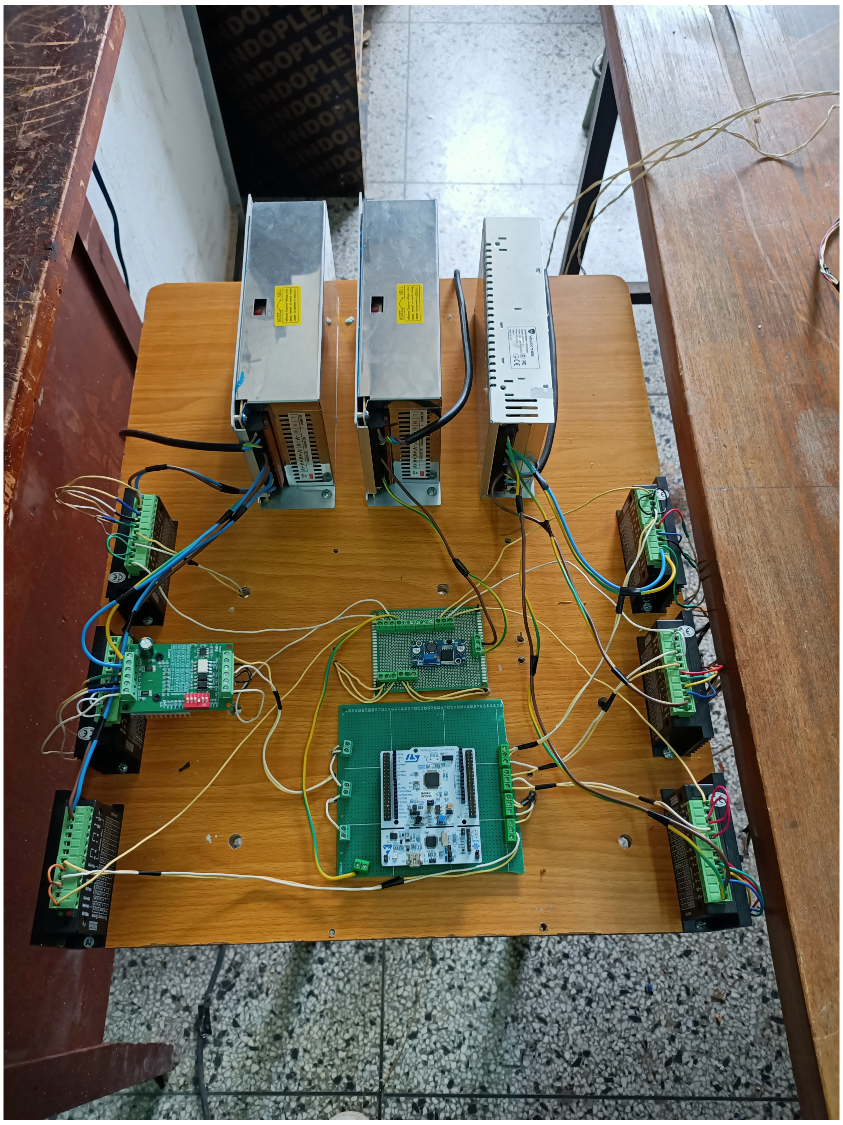


Figure 4.13: The final electrical assembly

4.3 ROS-based motion planning with moveIt

4.3.1 ROS-based motion planning approach

In order to command our robotic arm, the first approach that we worked on is the ROS-based planning using moveIt pkg which is a really robust method for robotic arms, the main idea of this approach is :

1. **Motion Planning (ROS MoveIt):** The user provides a goal pose in the MoveIt RViz. MoveIt computes a trajectory and publishes the joint angles over the /joint_states topic.

2. **Trajectory buffering (Joint_cmd.py Node):** A custom Python node Joint_cmd.py listens to /joint_states, extracting the trajectory. This node:
   - Buffers all trajectory waypoints.
   - Samples the trajectory down to 10 intermediate waypoints for execution efficiency.
   - For each sampled waypoint [joint1, joint2, ..., joint6], it publishes six float topics: /joint1, /joint2, ..., /joint6, each corresponding to a motor joint angle.

3. **Low-Level Execution (STM32 via rosserial):** The STM32 board, interfaced via rosserial, listens to the /jointX topics. Upon receiving a full set of joint angles, it:
   - Commands each motor driver with the respective angle.
   - Executes the movement to the specified positions.
   - Publishes a /done flag (bool) set to true once the current waypoint is fully executed.

4. **Synchronized Execution:** The Joint_cmd.py node waits for /done == true before sending the next waypoint, ensuring smooth, sequential motion execution.

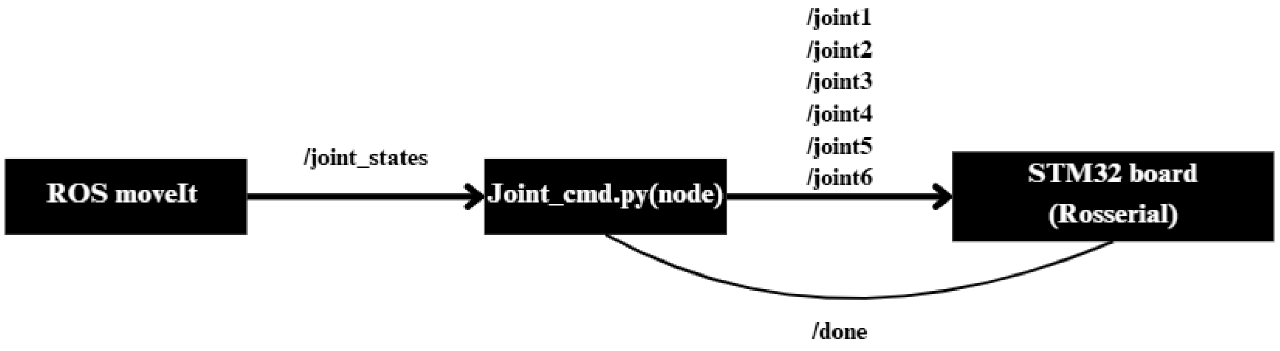


Figure 4.14: ROS moveIt approach

4.3.1.1 Reality vs Simulation testing

Now we move on to the best part of the whole process, which consists on testing the ROS path planning and execution simulation in the real robot. Our test consists on sending the robot from a init pose (starting pose) into the second and third pose passing by mid points and try to match those poses with real life robot.

4.3.1.1.1 First target pose We sent our robot to the first target pose , and we see that the real life robot is clearly following the simulation based on it poses (starting, mid and final poses) as well as the trajectory.

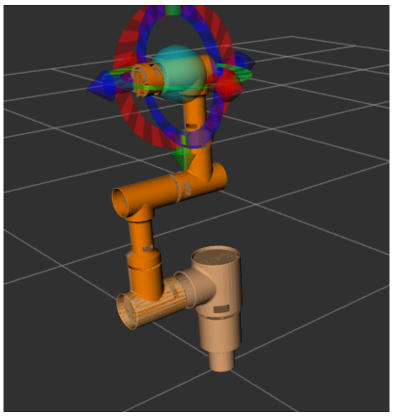


Figure 4.15: The initial pose on simulation

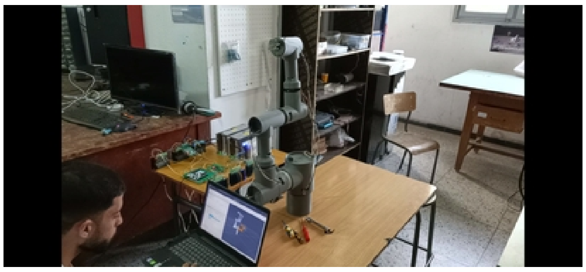


Figure 4.16: The initial pose in reality

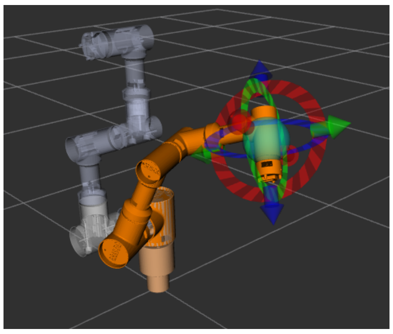


Figure 4.17: The first target pose mid trajectory on simulation

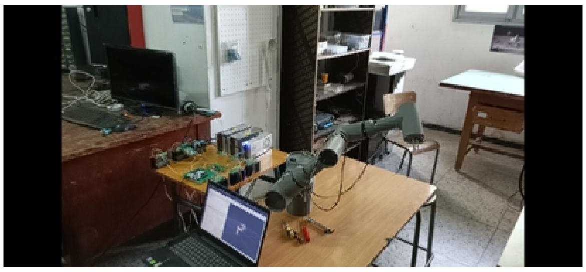


Figure 4.18: The first target pose mid trajectory in reality

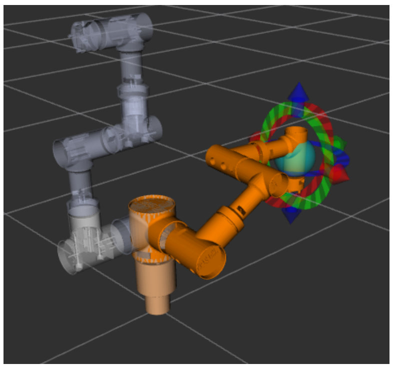


Figure 4.19: The first target pose end trajectory on simulation

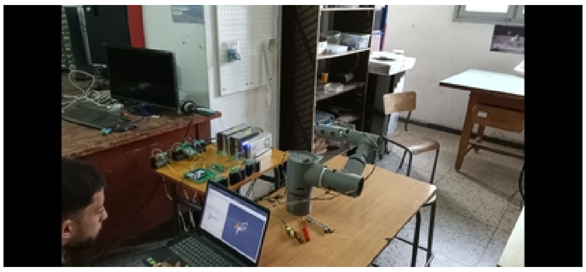


Figure 4.20: The first target pose end trajectory in reality

4.3.1.1.2 The second target pose With the same logic and from the last checkpoint(the final pose of the first target) we send our robot to a second target.

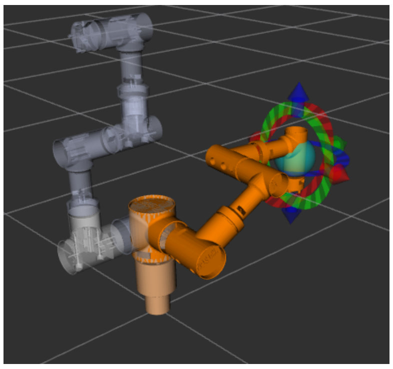


Figure 4.21: The initial pose on simulation

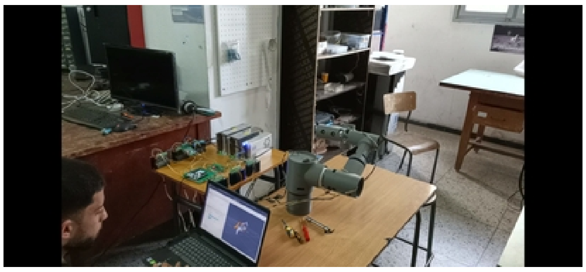


Figure 4.22: The initial pose in reality

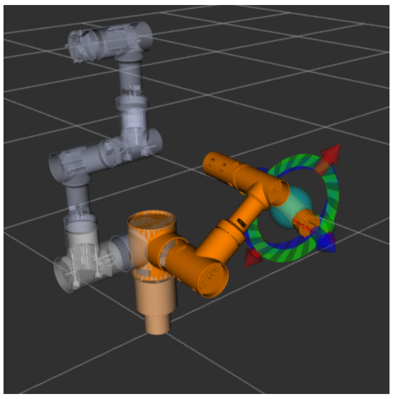


Figure 4.23: The second target pose mid trajectory on simulation

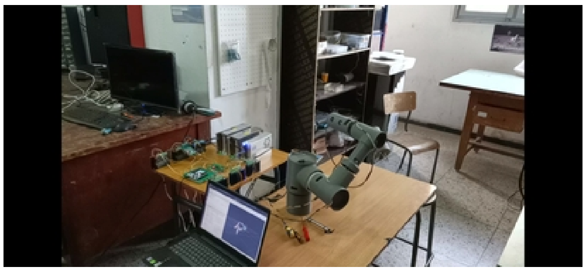


Figure 4.24: The second target pose mid trajectory in reality

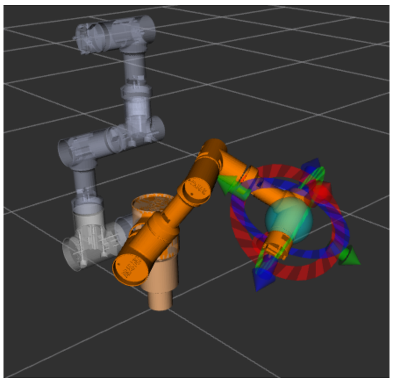


Figure 4.25: The second target pose end trajectory on simulation

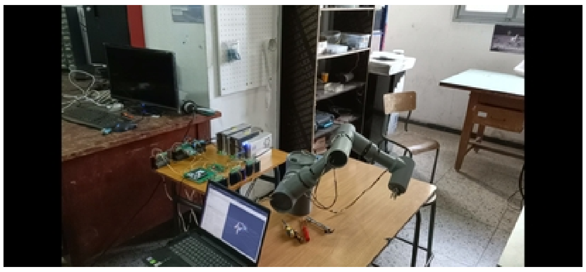


Figure 4.26: The second target pose end trajectory in reality

So at the end of both tests where we sent the robot from an initial pose on sim and reality to 2 consecutive target poses and we saw that the robot in reality matches the robot in simulation in final poses and even in the followed trajectory

4.4 Conclusion

As a conclusion for this chapter, we were able to integrate the actuators, sensors and control board (The STM32) and realize the designed circuit in reality as shown by the pervious figure 4.13, in addition to that we did the setup of our robotic arm using ROS and used it for some movement simulations on the robot that gave great results.

Based on what was presented so far, we're going to use it for next based autonomy approaches specially the arUco pose estimation based approach.

Chapter 5

Robotic arm autonomy using machine learning

5.1 Introduction

Getting now the most interesting part, which is the machine learning based autonomy approaches we worked on alongside the test and results we got for that.

As a first autonomy approach, we started with the supervised learning technique, in which we tried to estimate the next joint angles for our robot joints based on the current frame image and joint angles, working on that we're trying to perform a pick and place task and specially the goal reaching aspect. For this purpose, we got reference to some free available datasets online and applied our approach on it in order to clone the behavior of the reference robotic arms performing the same task efficiently.And to get the results on our case of use robotic arm BRAS-DEL, we performed a motion transfer from the robot dataset we worked on which the sawyer robot to our BRAS-DEL robot.

In addition to that, we worked on the reinforcement learning approach in which we tried to teach our own robot on how to perform the task without the need of dataset of any external robot. The main idea was to create our own IK solver without the need of using the available ones on MoveIt or Robodk as previously explained.

As a final approach, since we want to work on the industrial aspect at the same time. We proposed to work using the arUco tags since its the less consuming approach and the one with higher rate of success specially for the industrial environment.

5.2 Supervised learning

5.2.1 Introduction

Supervised learning is a type of machine learning where a model learns to map inputs to outputs using a labeled dataset. In the context of robotics, supervised learning is often used to train a robot to perform tasks by learning from recorded demonstrations. Each data sample in the dataset consists of an input (such as an image from the robot's camera) and a corresponding output (such as a control command or action executed by the robot). The idea here is to train our robotic arm based on the demonstrations of other well known robotic arms performing similar tasks that we want to perform (Pick & place tasks), and then transforming the results to ours.

5.2.2 Training process

The main idea is to learn a direct mapping from visual observations to joint-space control commands using demonstrations collected from other robotic arms performing similar tasks. We begin by selecting a dataset containing video demonstrations of robotic arms successfully completing manipulation tasks. Each video represents a temporal sequence where the robot transitions from an initial configuration to a final state that completes the task. Given that the robot in the dataset at the end of the video performs the task successfully, our objective is to replicate that behavior in our own robotic system. To train the model, each video is decomposed into a sequence of image frames. For every image frame at time step i, we pair it with the corresponding 7-dimensional vector of joint angles, denoted as $Qpos_i$. The supervised learning task is then defined as predicting the next joint configuration, $Qpos_{i+1}$, based on the current visual observation Img_i . So we have as input :

$$Input = Qpos_i + Img_i \tag{5.1}$$

$$Output = Qpos_{i+1} \tag{5.2}$$

5.2.3 Dataset

5.2.3.1 Dataset Description

The dataset used for training in our case is RoboNet, a large-scale robotic dataset collected and maintained by researchers at the University of California, Berkeley and Pennsylvania. RoboNet is publicly available and can be accessed through the RoboNet project website.

5.2.3.2 Dataset Overview

RoboNet contains over 15 million frames of robotic interaction data collected across multiple robot arms (e.g., Sawyer, WidowX, UR5e, etc.) using a variety of camera perspectives and physical configurations.

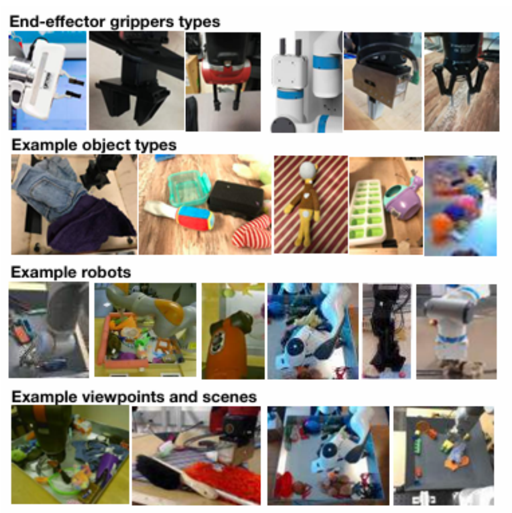
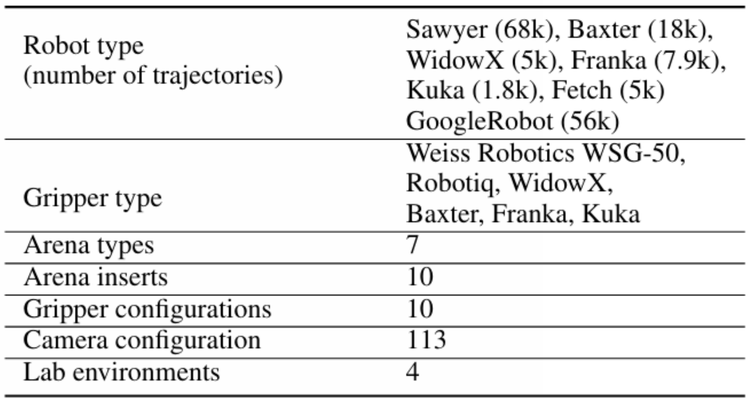


Figure 5.1: Qualitative examples of the various attributes in the RoboNet dataset[28].



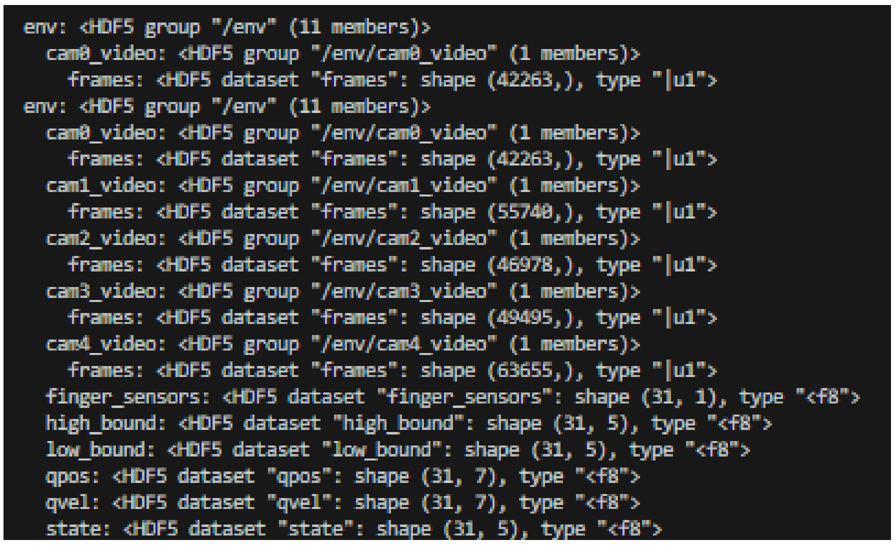
Robot type (number of trajectories)	Sawyer (68k), Baxter (18k), WidowX (5k), Franka (7.9k), Kuka (1.8k), Fetch (5k) GoogleRobot (56k)
Gripper type	Weiss Robotics WSG-50, Robotiq, WidowX, Baxter, Franka, Kuka
Arena types	7
Arena inserts	10
Gripper configurations	10
Camera configuration	113
Lab environments	4

Figure 5.2: Quantitative overview of the various attributes in the RoboNet dataset, including the 6 different robot arms and 6 different grippers[28].

5.2.3.3 Data structure

The dataset contains multiple robots and for each robot's experiment is saved as $robot_i$trajectory.hdf5. Each datapoint.hdf5 typically consists of:

- RGB images observations from different camera angles.
- Robot states (e.g., joints efforts, velocities, positions and angles, eef position)
- Time series data for each trajectory and other metadata.



```
env: <HDF5 group "/env" (11 members)>
  cam0_video: <HDF5 group "/env/cam0_video" (1 members)>
    frames: <HDF5 dataset "frames": shape (42263,), type "|u1">
env: <HDF5 group "/env" (11 members)>
  cam0_video: <HDF5 group "/env/cam0_video" (1 members)>
    frames: <HDF5 dataset "frames": shape (42263,), type "|u1">
  cam1_video: <HDF5 group "/env/cam1_video" (1 members)>
    frames: <HDF5 dataset "frames": shape (55740,), type "|u1">
  cam2_video: <HDF5 group "/env/cam2_video" (1 members)>
    frames: <HDF5 dataset "frames": shape (46978,), type "|u1">
  cam3_video: <HDF5 group "/env/cam3_video" (1 members)>
    frames: <HDF5 dataset "frames": shape (49495,), type "|u1">
  cam4_video: <HDF5 group "/env/cam4_video" (1 members)>
    frames: <HDF5 dataset "frames": shape (63655,), type "|u1">
  finger_sensors: <HDF5 dataset "finger_sensors": shape (31, 1), type "<f8">
  high_bound: <HDF5 dataset "high_bound": shape (31, 5), type "<f8">
  low_bound: <HDF5 dataset "low_bound": shape (31, 5), type "<f8">
  qpos: <HDF5 dataset "qpos": shape (31, 7), type "<f8">
  qvel: <HDF5 dataset "qvel": shape (31, 7), type "<f8">
  state: <HDF5 dataset "state": shape (31, 5), type "<f8">
```

Figure 5.3: HDF5 file robot trajectory structure

5.2.4 Data Prepossessing

Before starting the training phase, we need to perform a crucial step which is the "Data Preprocessing"

Parsing and frames extraction :
Each trajectory is stored as an HDF5 file containing multiple modalities such as RGB frames, robot joint positions (qpos), actions, and metadata.

Resizing & Grayscaling :
All RGB frames were resized to a uniform dimension of 64×64×1 pixels to reduce computational complexity, decrease memory usage and ensure consistent input size for the neural network.

Normalization :
Each grayscale pixel value was normalized to the range [0, 1] by dividing by 255. This normalization ensures faster convergence during training.

Input/Output Pairing :
We created an input consisting of:

- The grayscale 64*64 image frame img_i.
- The corresponding joints angles vector $Qpos_i$ (7-dimensional).

And the output as:

- The next joints angles vector $Qpos_{i+1}$.

Data Export :
All processed input-output pairs were saved in a CSV file format, which is more faster and efficient during training.

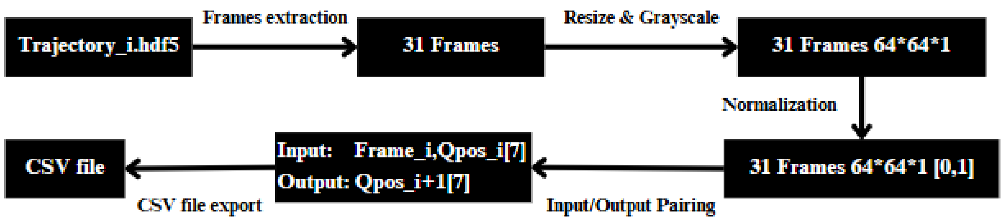


Figure 5.4: Preprocessing architecture

5.2.5 Deep Learning model

The proposed network architecture is composed of two main parts. First, a CNN is used for feature extraction, consisting of five convolutional layers, each followed by max pooling. This deep convolutional structure increases the network's capacity to learn multiple significant feature maps from the input images. Second, the output feature map from the CNN is concatenated with the current joint position vector $Qpos_i$, forming the input to a deep fully connected network. This DNN is designed for predicting the next joint angles $Qpos_{i+1}$. To improve prediction accuracy and reduce overfitting, the number of layers in both the CNN and DNN parts was empirically adjusted to achieve the best performance.

So this is our Deep learning model architecture that we used in this training phase :

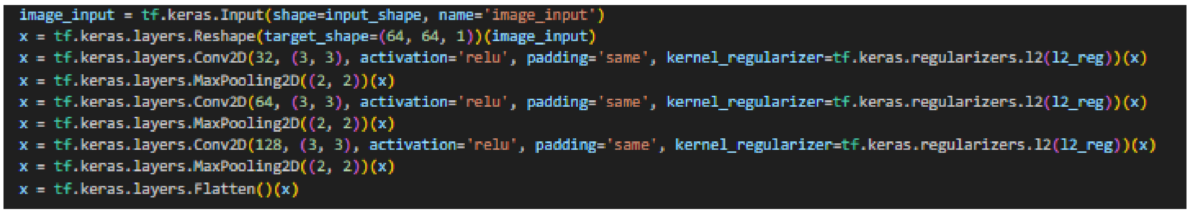
Layer Type	**Kernel Size / Units**
Reshape	(64, 64, 1)
Conv2D	(3,3) - 32 filters
MaxPool2D	(2,2)
Conv2D	(3,3) - 64 filters
MaxPool2D	(2,2)
Conv2D	(3,3) - 128 filters
MaxPool2D	(2,2)
Conv2D	(3,3) - 256 filters
MaxPool2D	(2,2)
Conv2D	(3,3) - 512 filters

Table 5.1: CNN architecture

Layer Type	**Kernel Size / Units**
MaxPool2D	(2,2)
Concatenate	CNN + qpos_input
Dense	1028 units
Dense	512 units
Dense	256 units
Dense	128 units
Dense	64 units
Output	7 classes

Table 5.2: Fully Connected Layers

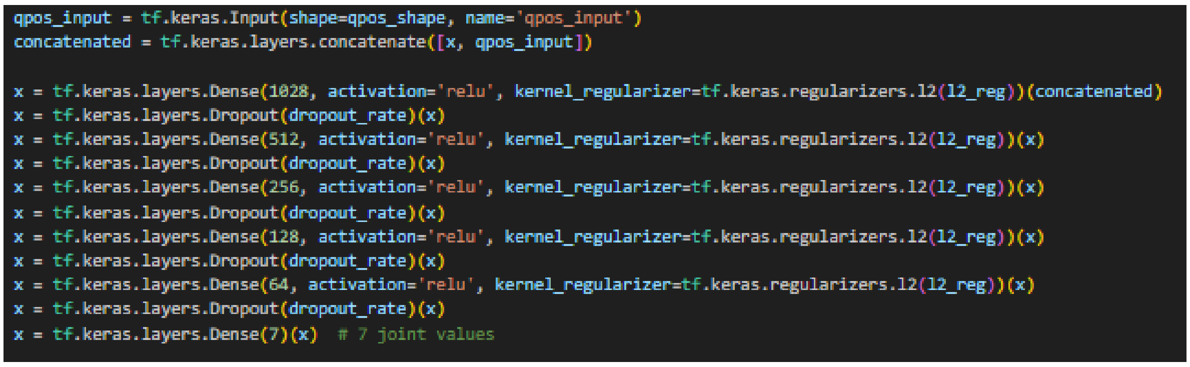
Figure 5.5 shows the Python code used to construct the CNN part of the model, including the convolutional and pooling layers for feature extraction.



```
image_input = tf.keras.Input(shape=input_shape, name='image_input')
x = tf.keras.layers.Reshape(target_shape=(64, 64, 1))(image_input)
x = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(x)
x = tf.keras.layers.MaxPooling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(x)
x = tf.keras.layers.MaxPooling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(x)
x = tf.keras.layers.MaxPooling2D((2, 2))(x)
x = tf.keras.layers.Flatten()(x)
```

Figure 5.5: Convolutional layers code

Figure 5.6 presents the implementation of the fully connected layers (DNN), which takes the CNN output concatenated with $Qpos_i$ and predicts the next joint angles $Qpos_{i+1}$



```
qpos_input = tf.keras.Input(shape=qpos_shape, name='qpos_input')
concatenated = tf.keras.layers.concatenate([x, qpos_input])

x = tf.keras.layers.Dense(1028, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(concatenated)
x = tf.keras.layers.Dropout(dropout_rate)(x)
x = tf.keras.layers.Dense(512, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(x)
x = tf.keras.layers.Dropout(dropout_rate)(x)
x = tf.keras.layers.Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(x)
x = tf.keras.layers.Dropout(dropout_rate)(x)
x = tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(x)
x = tf.keras.layers.Dropout(dropout_rate)(x)
x = tf.keras.layers.Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(l2_reg))(x)
x = tf.keras.layers.Dropout(dropout_rate)(x)
x = tf.keras.layers.Dense(7)(x)  # 7 joint values
```

Figure 5.6: Fully connected layers code

5.2.6 Training phase

5.2.6.1 Train hyper-parameters

In this section, we present the final hyperparameters selected after multiple tuning during the training phase of the neural network model used for our case of studies. Each hyperparameter plays an important role in the training dynamics and the model's generalization capability. The learning rate determines how quickly the model adapts to the data, while the number of epochs defines the total training iterations. The batch size influences both memory efficiency and gradient stability during updates. Strategies such as dropout penalty was applied during training to randomly deactivate 30% of neurons in specific layers, helping to prevent overfitting. A rate of 0.3 provides a good trade-off between learning and regularization. The choice of optimizer and loss function directly affects convergence behavior and training robustness and they we choosed based on the literature; in this case, the Adam optimizer and Mean Squared Error (MSE) loss function are well-suited for continuous value prediction. The fully connected layers, organized in a gradually decreasing architecture, enable progressive feature abstraction, transitioning from high-dimensional feature maps to the final joint angle outputs. The complete set of hyperparameters is summarized in Table 5.3 after multiple tuning.

Hyper-parameters	**Value**
Learning Rate	0.001
Epochs	80
Batch Size	512
Dropout Rate	30.0
L2 Regularization	0.0
Optimizer	Adam
Loss Function	Mean Squared Error
Activation Function (Conv/Dense)	ReLU
Fully Connected Layers	1028, 512, 256, 128, 64

Table 5.3: Hyper-parameters Settings for the Deep Learning Model

Additionally, the dataset was split into 80% for training, 15% for validation, and 5% for testing. This split allowed for efficient model evaluation and hyperparameter tuning while ensuring a reliable test set for final performance assessment.

5.2.6.2 Train results

5.2.6.2.1 Metric : Mean absolute error(MAE) :

In the context of a regression model training problem for predicting joint angles, the Mean Absolute Error (MAE) is a common evaluation metric. When the target variable is joint angles (typically in radians or degrees), the MAE measures the average absolute difference between the predicted and true joint angles.

$$\text{MAE}_{\text{total}} = \frac{1}{N \cdot J} \sum_{i=1}^{N} \sum_{j=1}^{J} \left| \hat{\theta}_{i,j} - \theta_{i,j} \right| \tag{5.3}$$

Where:N be the number of data samples and J the number of joints. The term $\hat{\theta}_{i,j}$ represents the predicted angle (in radians or degrees) of joint j for sample i, while $\theta_{i,j}$ denotes the ground-truth (true) angle of the same joint and sample. The notation $| \cdot |$ is used to indicate the absolute value.

At the end of the training we got a test mean absolute error(MAE) :

$$\boxed{\textbf{Final Test MAE: 0.06 rad}}$$

- A final MAE of 0.06 radians (3.4 degrees) means the average prediction error per joint is very low, which is quite good for robotic manipulation tasks.
- Validation MAE remains stable and closely follows the training MAE so no signs of overfitting.
- The model learns quickly in the first 10 epochs.

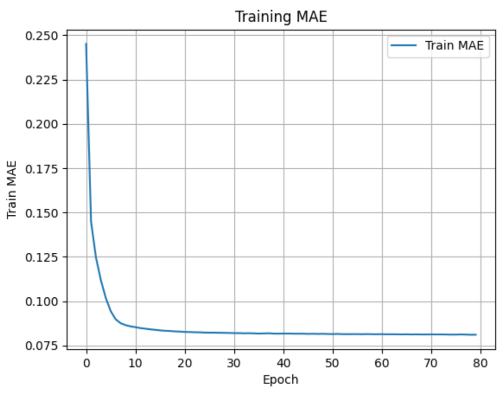


Figure 5.7: Train MAE

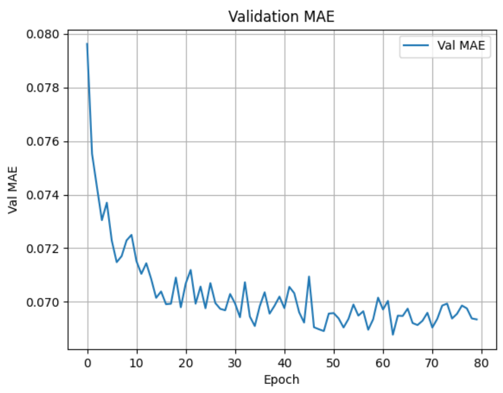


Figure 5.8: Validation MAE

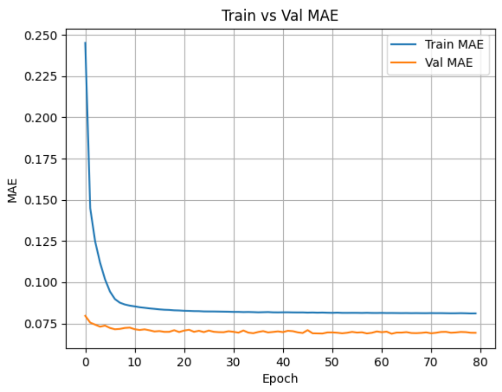


Figure 5.9: The combined MAEs

5.2.6.2.2 Loss : Mean squared error(MSE) :

The Mean Squared Error (MSE) serves as a common and effective loss function to evaluate the model's performance during training and testing.

$$\text{MSE}_{\text{total}} = \frac{1}{N \cdot J} \sum_{i=1}^{N} \sum_{j=1}^{J} \left(\hat{\theta}_{i,j} - \theta_{i,j}\right)^2 \tag{5.4}$$

Where:N be the number of data samples and J the number of joints. The predicted angle of joint j for sample i is denoted by $\hat{\theta}_{i,j}$, while $\theta_{i,j}$ represents the true (ground-truth) angle. The notation $(\cdot)^2$ indicates the squared difference.

At the end of the training we got a test mean squared error(MSE):

$$\boxed{\textbf{Final Test Loss (MSE): 0.0045 } rad^2}$$

- The training loss decreases rapidly and converges smoothly.
- Validation loss remains low and stable after early epochs, showing no signs of overfitting.
- A test loss (MSE) of 0.0045 indicates that the average squared error in predicting each joint angle is very small.

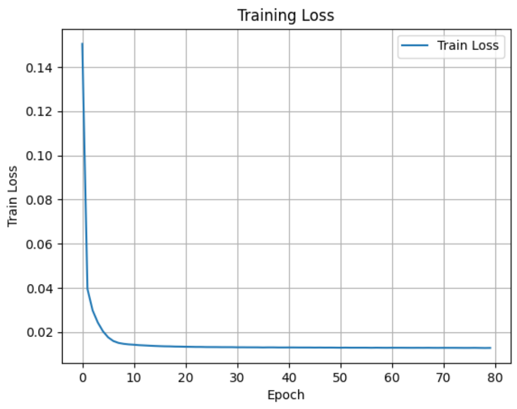


Figure 5.10: Train MSE

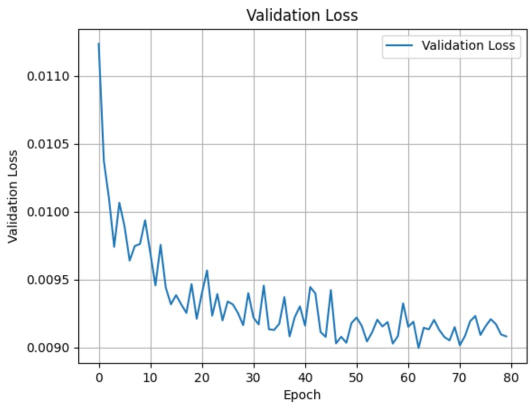


Figure 5.11: Validation MSE

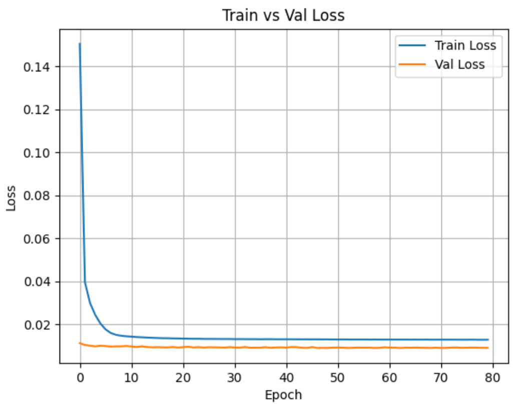


Figure 5.12: The combined MSEs

5.2.7 Experiments and tests

To see the effectiveness and generalization capability of our model, we conduct experiments on two different robotic platforms:

- The Sawyer robot, which is the source of the original dataset and serves as a benchmark for evaluating the model under conditions similar to training.
- Our custom robot, which was not used in training but we will transfer the behaviour of the sawyer robot which is the reference robot to our robot platform.

The evaluation focuses on comparing the ground-truth joint configurations and end-effector (EEF) positions against those predicted by the model. This is done to evaluate both joint-level accuracy and the resulting task-space performance.

5.2.7.1 Testing the result on Sawyer robot

The first set of experiments is conducted on the Sawyer robot—the same robot from the dataset. We integrate the Sawyer robot into the ROS(Robot Operating System) environment using its URDF description and generate a moveIt workspace for motion planning and control.

For each test case, we use a single image frame Img_i and then current joint config $Qpos_i$ from the dataset and feed it into the trained model to predict the next joint configuration $Qpos_{i+1}$ We then perform two separate forward kinematics evaluations:

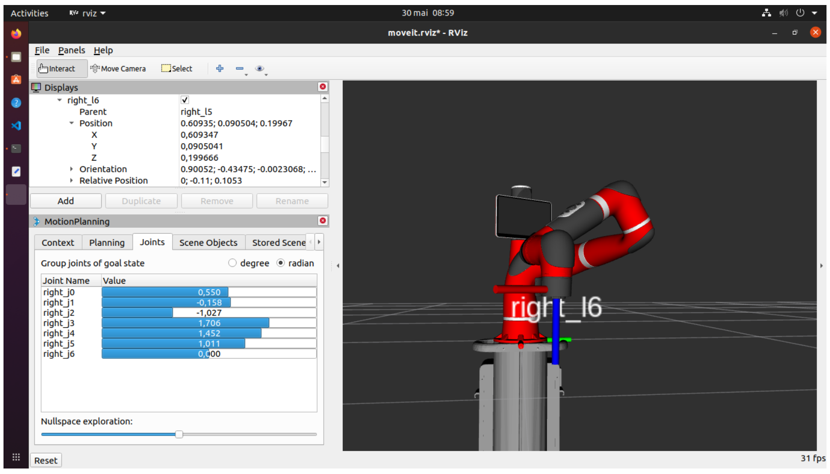
- One using the ground-truth $Qpos_{i+1}$ (from the dataset).
- One using the predicted $Qpos_{i+1}$ (from the model).

By comparing the resulting global poses and end-effector positions from both configurations, we compare how accurately the model can replicate the behavior encoded in the dataset. This evaluation provides insight into the model's precision in reproducing learned trajectories.

5.2.7.1.1 Test trajectory n° 1 first frame : The first test trajectory in the 40005th traj that contains 31 frame and Qpos , we tested the predicted angles on the first frame (1st. And this table showcase the ground-truth angles and the predicted ones.

The frame	Qpos[0]	Qpos[1]	Qpos[2]	Qpos[3]	Qpos[4]	Qpos[5]
Ground-truth (rad)	0.551	-0.163	-1.033	1.709	1.450	1.1017
Predicted (rad)	0.555	-0.185	-1.095	1.729	1.437	1.075
Error (rad)	0.004	0.022	0.062	0.020	0.013	0.0267
Error (deg)	0.229	1.261	3.553	1.146	0.745	1.53
Mean error (deg)	**1.411**					

Table 5.4: Comparison of angular values (in radians and degrees) between Ground-truth and Predicted for traj40005 frame 2



Displays

right_l6	✓
Parent	right_l5
Position	0.60935; 0.090504; 0.19967
X	0,609347
Y	0,0905041
Z	0,199666
Orientation	0.90052; -0.43475; -0.0023068; …
Relative Position	0; -0.11; 0.1053

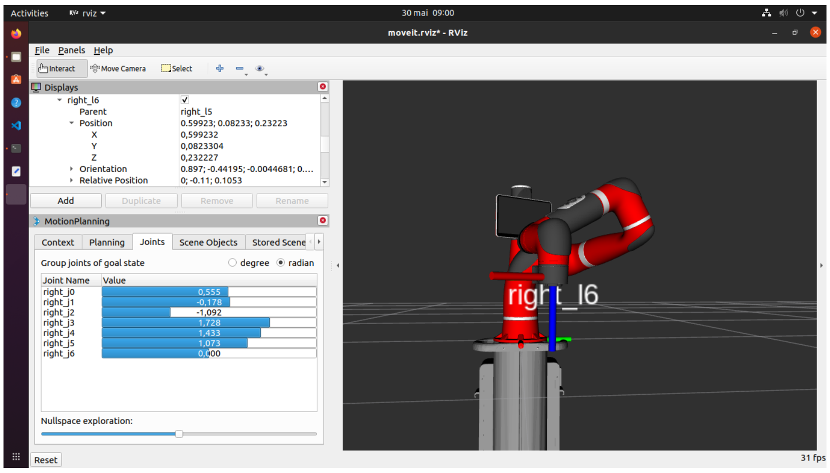
MotionPlanning

Group joints of goal state ○ degree [✓ radian]

Joint Name	Value
right_j0	0,550
right_j1	-0,158
right_j2	-1,027
right_j3	1,706
right_j4	1,452
right_j5	1,011
right_j6	0,000

Nullspace exploration:

Figure 5.13: The ground-truth Qpos frame 2



Displays

right_l6	✓
Parent	right_l5
Position	0.59923; 0.08233; 0.23223
X	0,599232
Y	0,0823304
Z	0,232227
Orientation	0.897; -0.44195; -0.0044681; 0.…
Relative Position	0; -0.11; 0.1053

MotionPlanning

Group joints of goal state ○ degree [✓ radian]

Joint Name	Value
right_j0	0,555
right_j1	-0,178
right_j2	-1,092
right_j3	1,728
right_j4	1,433
right_j5	1,073
right_j6	0,000

Nullspace exploration:

Figure 5.14: The predicted Qpos frame 2

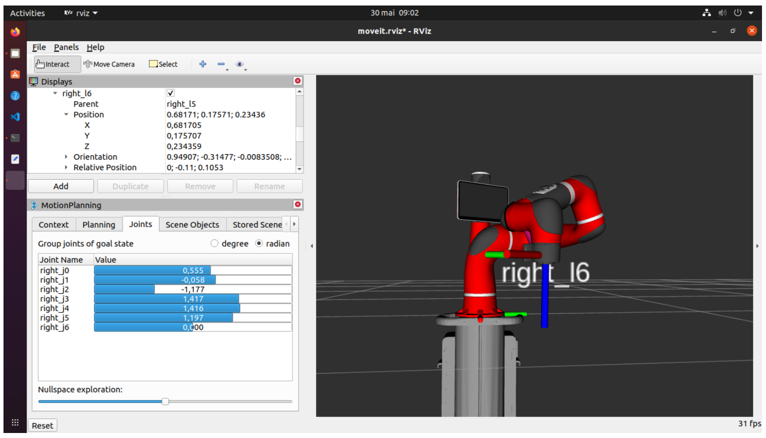
For the comparison, the predicted angles pose got a really close pose and behavior as the real pose. And for the EEF position we got a difference of

- 0.609-0.5992=0.0098 m = 0.98 cm on the X axe.
- 0.0905-0.0823=0.0082 m = 0.82 cm on the Y axe.
- 0.1999-0.2322=-0.0323 m= 3.23 cm on the Z axe.

5.2.7.1.2 Test trajectory n° 1 last frame :The second test that we will perform is on 40005th traj too, but we will test the results after 31 successive predictions (so we take the first img (Img_0) and the first Qpos ($Qpos_0$) and keep predicting based on all the rest 30 frames to see the cumulative error of the trajectory. And this table showcase the ground-truth angles and the predicted ones.

The frame	Qpos[0]	Qpos[1]	Qpos[2]	Qpos[3]	Qpos[4]	Qpos[5]
Ground-truth (rad)	0.553	-0.060	-1.178	1.4125	1.417	1.193
Predicted (rad)	0.427	-0.037	-1.290	1.292	1.447	1.297
Error (rad)	0.126	0.023	0.112	0.1205	0.030	0.104
Error (deg)	7.22	1.32	6.42	6.90	1.72	5.96
Mean error (deg)	**4.92**					

Table 5.5: Comparison of angular values (in radians and degrees) between Ground-truth and Predicted for traj40005 frame 31



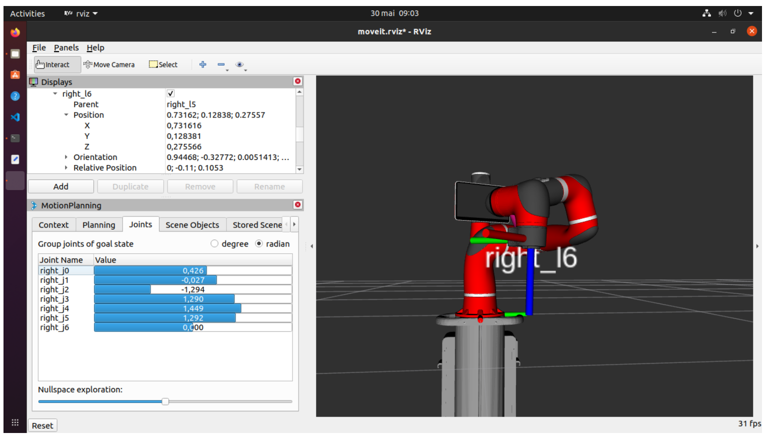
right_l6	✓
Parent	right_l5
Position	0.68171; 0.17571; 0.23436
X	0,681705
Y	0,175707
Z	0,234359
Orientation	0.94907; -0.31477; -0.0083508; ...
Relative Position	0; -0.11; 0.1053

Group joints of goal state ○ degree [✓ radian]

Joint Name	Value
right_j0	0,555
right_j1	-0,058
right_j2	-1,177
right_j3	1,417
right_j4	1,416
right_j5	1,197
right_j6	0,000

Nullspace exploration:

Figure 5.15: The ground-truth Qpos frame 31



right_l6	✓
Parent	right_l5
Position	0.73162; 0.12838; 0.27557
X	0,731616
Y	0,128381
Z	0,275566
Orientation	0.94468; -0.32772; 0.0051413; ...
Relative Position	0; -0.11; 0.1053

Group joints of goal state ○ degree [✓ radian]

Joint Name	Value
right_j0	0,426
right_j1	-0,027
right_j2	-1,294
right_j3	1,290
right_j4	1,449
right_j5	1,292
right_j6	0,000

Nullspace exploration:

Figure 5.16: The predicted Qpos frame 31

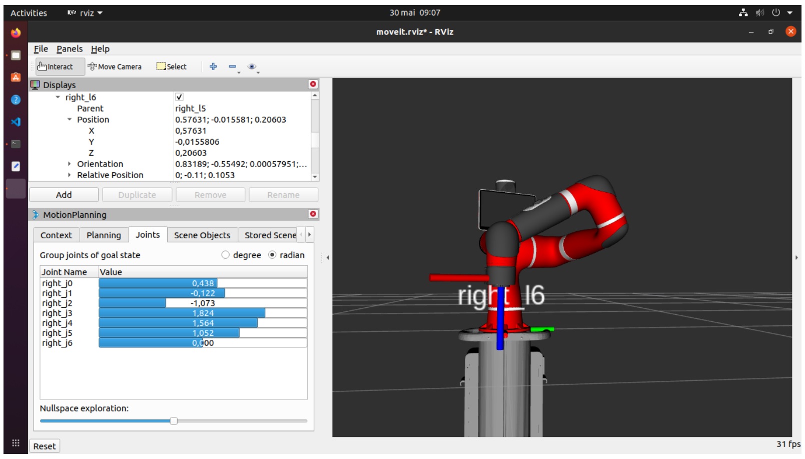
For the comparison, the predicted angles pose got a really close pose and behavior as the real pose. And for the EEF position we got a difference of

- 0.681-0.7316= -0.0506 m = -5.06 cm on the X axe.
- 0.175-0.128= 0.047 m = 4.7 cm on the Y axe.
- 0.234-0.275= -0.041 m = 4.1 cm on the Z axe.

5.2.7.1.3 Test trajectory n° 2 first frame : The third test trajectory in the 40007th traj that contains 31 frame and Qpos too , we tested the predicted angles on the first frame (1st. And this table showcase the ground-truth angles and the predicted ones.

The frame	Qpos[0]	Qpos[1]	Qpos[2]	Qpos[3]	Qpos[4]	Qpos[5]
Ground-truth (rad)	0.440	-0.116	-1.079	1.820	1.565	1.056
Predicted (rad)	0.291	-0.177	-1.176	1.730	1.4436	1.152
Error (rad)	0.149	0.061	0.097	0.090	0.1214	0.096
Error (deg)	8.54	3.50	5.56	5.15	6.96	5.50
Mean error (deg)	**5.87**					

Table 5.6: Comparison of angular values (in radians and degrees) between Ground-truth and Predicted for traj40007 frame 2

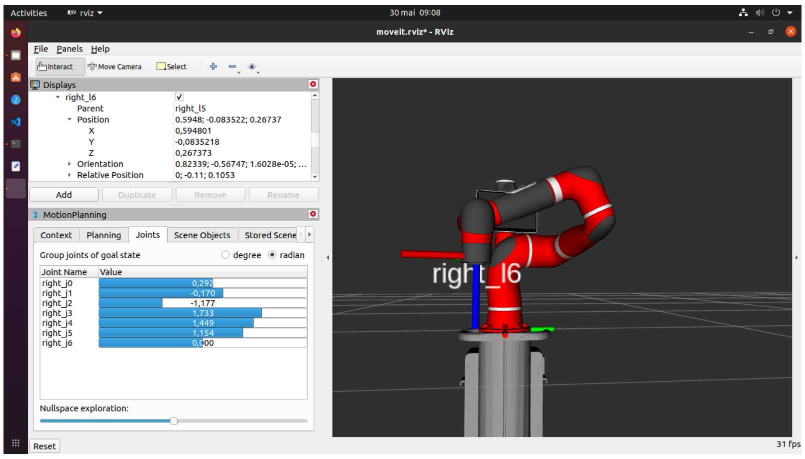


right_l6	✓
Parent	right_l5
Position	0.57631; -0.015581; 0.20603
X	0,57631
Y	-0,0155806
Z	0,20603
Orientation	0.83189; -0.55492; 0.00057951;...
Relative Position	0; -0.11; 0.1053

Group joints of goal state ○ degree ● radian

Joint Name	Value
right_j0	0,438
right_j1	-0,122
right_j2	-1,073
right_j3	1,824
right_j4	1,564
right_j5	1,052
right_j6	0,000

Figure 5.17: The ground-truth Qpos frame 2



right_l6	✓
Parent	right_l5
Position	0.5948; -0.083522; 0.26737
X	0,594801
Y	-0,0835218
Z	0,267373
Orientation	0.82339; -0.56747; 1.6028e-05; ...
Relative Position	0; -0.11; 0.1053

Group joints of goal state ○ degree ● radian

Joint Name	Value
right_j0	0,292
right_j1	-0,170
right_j2	-1,177
right_j3	1,733
right_j4	1,449
right_j5	1,154
right_j6	0,000

Figure 5.18: The predicted Qpos frame 2

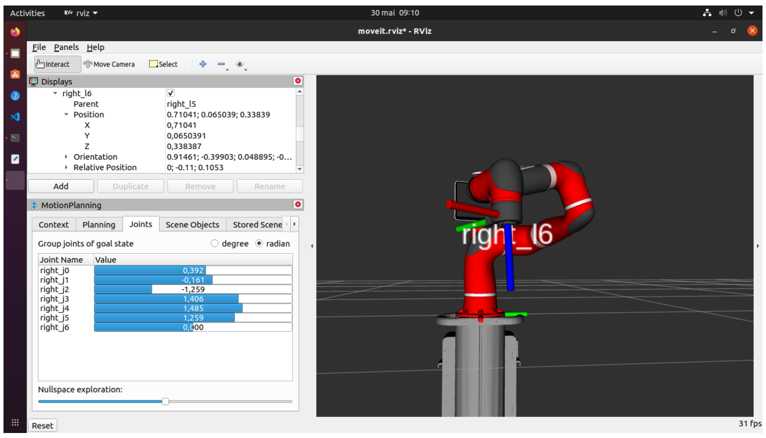
For the comparison, the predicted angles pose got a really close pose and behavior as the real pose. And for the EEF position we got a difference of

- 0.576-0.5948= -0.0188 m = -1.88 cm on the X axe.
- -0.015+0.08352= 0.06852 m = 6.852 cm on the Y axe.
- 0.206-0.267= -0.061 m = -6.1 cm on the Z axe.

5.2.7.1.4 Test trajectory n° 2 last frame :The fourth test that we will perform is on 40007th traj too, but we will test the results after 31 successive predictions (so we take the first img (Img_0) and the first Qpos ($Qpos_0$) and keep predicting based on all the rest 30 frames to see the cumulative error of the trajectory. And this table showcase the ground-truth angles and the predicted ones.

The frame	Qpos[0]	Qpos[1]	Qpos[2]	Qpos[3]	Qpos[4]	Qpos[5]
Ground-truth (rad)	0.394	-0.016	-1.259	1.397	1.485	1.259
Predicted (rad)	0.168	0.017	-1.364	1.183	1.499	1.3607
Error (rad)	0.226	0.033	0.105	0.214	0.014	0.1017
Error (deg)	12.95	1.89	6.02	12.27	0.80	5.83
Mean error (deg)	**6.62**					

Table 5.7: Comparison of angular values (in radians and degrees) between Ground-truth and Predicted for traj40007 frame 31



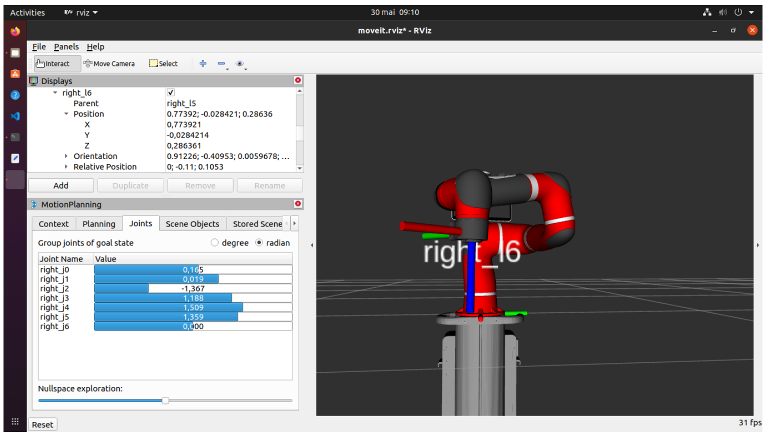
right_l6	✓
Parent	right_l5
Position	0.71041; 0.065039; 0.33839
X	0,71041
Y	0,0650391
Z	0,338387
Orientation	0.91461; -0.39903; 0.048895; -0...
Relative Position	0; -0.11; 0.1053

Group joints of goal state ○ degree ● radian

Joint Name	Value
right_j0	0,392
right_j1	-0,161
right_j2	-1,259
right_j3	1,406
right_j4	1,485
right_j5	1,259
right_j6	0,000

Nullspace exploration:

Figure 5.19: The ground-truth Qpos frame 31



right_l6	✓
Parent	right_l5
Position	0.77392; -0.028421; 0.28636
X	0,773921
Y	-0,0284214
Z	0,286361
Orientation	0.91226; -0.40953; 0.0059678; ...
Relative Position	0; -0.11; 0.1053

Group joints of goal state ○ degree ● radian

Joint Name	Value
right_j0	0,165
right_j1	0,019
right_j2	-1,367
right_j3	1,188
right_j4	1,509
right_j5	1,359
right_j6	0,000

Nullspace exploration:

Figure 5.20: The predicted Qpos frame 31

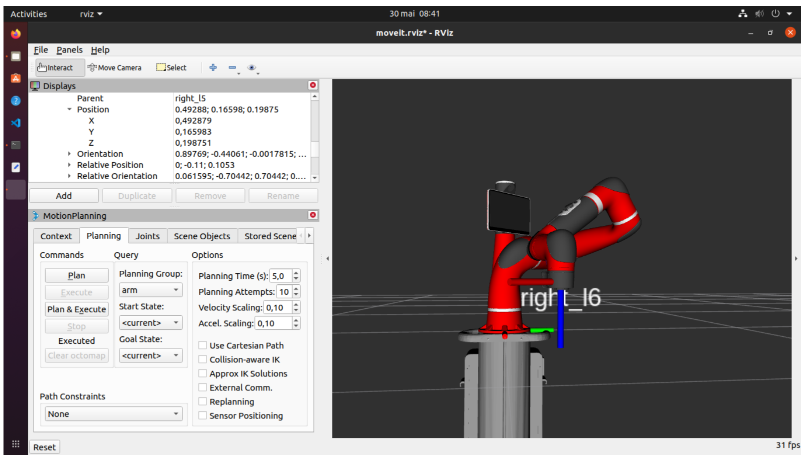
For the comparaison, the predicted angles pose got a really close pose and behaviour as the real pose. And for the EEF position we got a differnce of

- 0.7104-0.7739= -0.0635 m = 6.35 cm on the X axe.
- 0.0650+0.0284= 0.0934 m = 9.34 cm on the Y axe.
- 0.338-0.286= 0.052 m = 5.2 cm on the Z axe.

5.2.7.1.5 Test trajectory n° 3 first frame : The fifth test trajectory in the 40009th traj that contains 31 frame and Qpos too , we tested the predicted angles on the first frame (1st and on the last 31th. And this table showcase the ground-truth angles and the predicted ones.

The frame	Qpos[0]	Qpos[1]	Qpos[2]	Qpos[3]	Qpos[4]	Qpos[5]
Ground-truth (rad)	0.828	-0.238	-1.00	2.03	1.583	0.952
Predicted (rad)	0.871	-0.255	-1.03	1.97	1.516	0.970
Error (rad)	0.043	0.017	0.03	0.06	0.067	0.018
Error (deg)	2.46	0.97	1.72	3.44	3.84	1.03
Mean error (deg)	**2.24**					

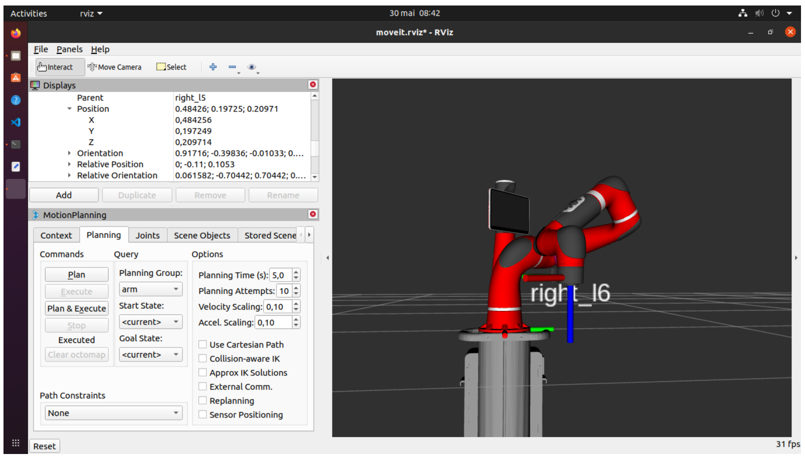
Table 5.8: Comparison of angular values (in radians and degrees) between Ground-truth and Predicted for traj40009 frame 2



Parent: right_l5
Position: 0.49288; 0.16598; 0.19875
X: 0,492879
Y: 0,165983
Z: 0,198751
Orientation: 0.89769; -0.44061; -0.0017815; ...
Relative Position: 0; -0.11; 0.1053
Relative Orientation: 0.061595; -0.70442; 0.70442; 0....

Planning Group: arm
Start State: <current>
Goal State: <current>
Path Constraints: None
Planning Time (s): 5,0
Planning Attempts: 10
Velocity Scaling: 0,10
Accel. Scaling: 0,10
☐ Use Cartesian Path
☐ Collision-aware IK
☐ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning

Figure 5.21: The ground-truth Qpos frame 2



Parent: right_l5
Position: 0.48426; 0.19725; 0.20971
X: 0,484256
Y: 0,197249
Z: 0,209714
Orientation: 0.91716; -0.39836; -0.01033; 0....
Relative Position: 0; -0.11; 0.1053
Relative Orientation: 0.061582; -0.70442; 0.70442; 0....

Planning Group: arm
Start State: <current>
Goal State: <current>
Path Constraints: None
Planning Time (s): 5,0
Planning Attempts: 10
Velocity Scaling: 0,10
Accel. Scaling: 0,10
☐ Use Cartesian Path
☐ Collision-aware IK
☐ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning

Figure 5.22: The predicted Qpos frame 2

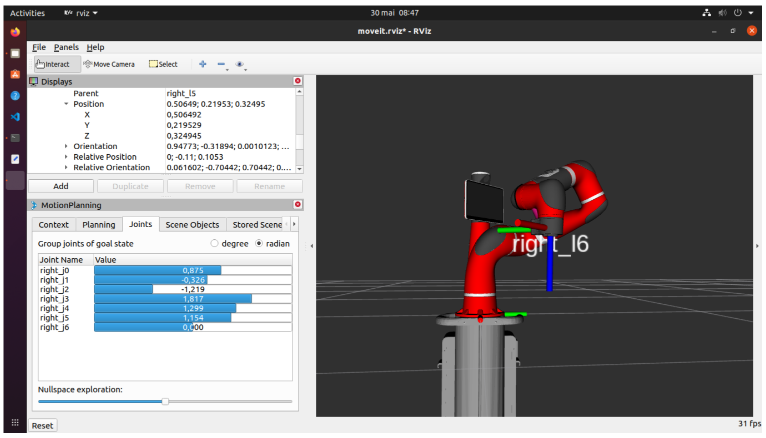
For the comparison, the predicted angles pose got a really close pose and behavior as the real pose. And for the EEF position we got a difference of

- 0.484-0.492= -0.008 m = -0.8 cm on the X axe.
- 0.197-0.165= 0.032 m = 3.2 cm on the Y axe.
- 0.209-0.198= 0.011 m = 1.1 cm on the Z axe.

5.2.7.1.6 Test trajectory n° 3 last frame : The sixth test that we will perform is on 40009th traj too, but we will test the results after 31 successive predictions (so we take the first img (Img_0) and the first Qpos ($Qpos_0$) and keep predicting based on all the rest 30 frames to see the cumulative error of the trajectory. And this table showcase the ground-truth angles and the predicted ones.

The frame	Qpos[0]	Qpos[1]	Qpos[2]	Qpos[3]	Qpos[4]	Qpos[5]
Ground-truth (rad)	0.878	-0.325	-1.216	1.821	1.303	1.156
Predicted (rad)	0.787	-0.263	-1.080	1.845	1.422	1.030
Error (rad)	0.091	0.062	0.136	0.024	0.119	0.126
Error (deg)	5.22	3.55	7.79	1.38	6.82	7.22
Mean error (deg)	**5.33**					

Table 5.9: Comparison of angular values (in radians and degrees) between Ground-truth and Predicted for traj40009 frame 31



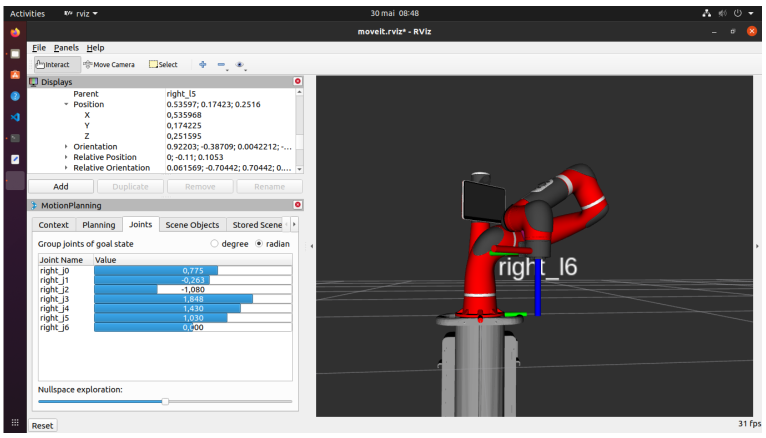
Parent	right_l5
Position	0.50649; 0.21953; 0.32495
X	0,506492
Y	0,219529
Z	0,324945
Orientation	0.94773; -0.31894; 0.0010123; ...
Relative Position	0; -0.11; 0.1053
Relative Orientation	0.061602; -0.70442; 0.70442; 0....

Group joints of goal state ○ degree [✓ radian]

Joint Name	Value
right_j0	0,875
right_j1	-0,326
right_j2	-1,219
right_j3	1,817
right_j4	1,299
right_j5	1,154
right_j6	0,000

Nullspace exploration:

Figure 5.23: The ground-truth Qpos frame 31



Parent	right_l5
Position	0.53597; 0.17423; 0.2516
X	0,535968
Y	0,174225
Z	0,251595
Orientation	0.92203; -0.38709; 0.0042212; -...
Relative Position	0; -0.11; 0.1053
Relative Orientation	0.061569; -0.70442; 0.70442; 0....

Group joints of goal state ○ degree [✓ radian]

Joint Name	Value
right_j0	0,775
right_j1	-0,263
right_j2	-1,080
right_j3	1,848
right_j4	1,430
right_j5	1,030
right_j6	0,000

Nullspace exploration:

Figure 5.24: The predicted Qpos frame 31

For the comparison, the predicted angles pose got a really close pose and behavior as the real pose. And for the EEF position we got a difference of

- 0.506-0.535= -0.029 m = -2.9 cm on the X axe.
- 0.219-0.174= 0.045 m = 4.5 cm on the Y axe.
- 0.324-0.251= 0.073 m = 7.3 cm on the Z axe.

5.2.7.1.7 Results : Joints angles Mean Absolute Error We compare the joints angles MAE results that we got from the 3 test experiments of the first frames only with the state of the art proposed method.
Starting with MAE results of the [31] proposed method :

	Joint 1	**Joint 2**	**Joint 3**
Joint angle MAE(degree)	0.75	0.69	1.69
Mean MAE (degree)	**1.04**		

Table 5.10: Joints angles MAE[31].

From table 5.4,5.6 & 5.8 , we have our robot joints angles MAE in degrees:

Tests	**Joint 1**	**Joint 2**	**Joint 3**	**Joint 4**	**Joint 5**	**Joint 6**	**MAE(degree)**
Trajectory 40005	0.229	1.261	3.553	1.146	0.745	1.53	1.41
Trajectory 40007	8.54	3.50	5.56	5.15	6.96	5.50	5.87
Trajectory 40009	2.46	0.97	1.72	3.44	3.84	1.03	2.24
Mean MAE (degree)							**3.17**

Table 5.11: Our robot joints angles MAE

Table 5.10 shows that the method by Rodríguez-Miranda et al[31] achieved a low mean error of **1.04°** using a two-step process: first classifying the image into a pose group, then refining it with a neural network.

Our method, shown in Table 5.11, had a higher error of **3.17°**, but it works differently. Instead of using pose groups, we train the model to directly predict the next joint angles using both the current image and joint positions.

There are a few reasons why our error is higher:

- We work with a 6-joint robot, while theirs has only 3.
- We predict movement over time , not just static poses.
- We use both image and joint data, which makes the input more complex.

5.2.7.1.8 Results : End Effector (EEF) position Mean Absolute Error Now we compare the End Effector(EEF) pose MAE result that we got from the 3 test experiments of the first frames only with the state of the art proposed method.

	Test 1	**Test 2**	**Test 3**
MAE(cm)	0.245	0.5	0.552
Mean MAE(cm)	**0.436**		

Table 5.12: EEF all axis MAE[31].

From table 5.4,5.6 & 5.8 , we have the robot end effector (EEF) in cm:

	X axis	**Y axis**	**Z axis**	**All axis**
Trajectory 40005 (cm)	0.98	0.82	3.23	1.67
Trajectory 40007 (cm)	1.88	6.852	6.1	4.94
Trajectory 40009 (cm)	0.8	3.2	1.1	1.7
Mean MAE(cm)				**2.77**

Table 5.13: Our robot all axis MAE

Based on tables 5.12 & 5.13, the [31] approach achieved a lower EEF mean error of **0.436 cm**, while our method resulted in a mean error of **2.77 cm**.

- The state-of-the-art method was tested on a 3-DOF robot, where only 3 joints contribute to the final EEF position.
- In contrast, our robot is a 6-DOF robot, meaning twice as many joints influence the EEF, each with its own prediction error. These errors accumulate, which can significantly affect the final EEF position.
- Furthermore, our model performs direct regression from images Img_i and current joint states $Qpos_i$ to the next configuration $Qpos_{i+1}$. This includes temporal transitions (learning how the robot moves over time), making the task more complex than simply regressing static positions from single images, as in the baseline method.

5.2.7.1.9 Results : EEF position error after 31 successive predictions

- For the End Effector (EEF) pose error here's a table that summarizes the cartesian axes X,Y & Z absolute error between the ground-truth position and the predicted position if the last frame of that trajectory because it holds the cumulative error for each prediction from the first frame .

The trajectory	EEF X absolute error	EEF Y absolute error	EEF Z absolute error
Trajectory 40005	5.06	4.7	4.1
Trajectory 40007	6.35	9.34	5.2
Trajectory 40009	2.9	4.5	7.3
Mean absolute error	4.77	6.18	5.533

Table 5.14: EEF X,Y & Z positions absolute error of the last frame for each test trajectory

- The evaluation was conducted over a set of test trajectories from the original dataset. The comparison focused on the positional accuracy of the EEF in Cartesian space. The mean absolute error (MAE) between the predicted and ground-truth EEF positions across the X, Y, and Z axes was found to be:

  ○ X-axis: 4.77 cm.
  ○ Y-axis: 6.18 cm.
  ○ Z-axis: 5.53 cm.

  So These results indicate that the model is capable of generating joint-space commands that result in accurate and task-relevant EEF positioning.

5.2.7.2 Testing the result on BRAS-DEL robot

The model was trained on the Sawyer robot, to test the model on the BRAS-DEL robot we need to transfer the obtained result on the reference robot to ours.

5.2.7.2.1 Test setup The test setup requires this next important steps:

- **Prediction (Keras Model)**: The first frame of the test trajectory, along with the initial joint angles $Qpos_i$, is passed to the trained Keras model. The model predicts the next joint configuration $Qpos_{i+1}$ for the Sawyer robot based on the visual input and current joint state.
- **Forward Kinematics on MoveIt (Sawyer Robot)**: The predicted joint angles are applied to the Sawyer robot using its URDF and MoveIt configuration. MoveIt performs forward kinematics to compute the resulting pose (position and orientation) of the end-effector (EEF) in the global reference frame.
- **Navigation to the Goal Pose**: The pose obtained from Sawyer's predicted configuration is considered the target goal. This goal represents the desired EEF pose in Cartesian space that the new robot (BRAS-DEL) must reach. It includes both the position (X, Y, Z) and orientation (typically as a quaternion or rotation matrix).
- **Transfer Node – Inverse Kinematics on MoveIt (BRAS-DEL)**: The goal pose (from the Sawyer EEF) is sent to the MoveIt IK solver for BRAS-DEL. The solver computes the corresponding joint angles $Qpos_{i+1}^{\text{our\_robot}}$ that would result in the same EEF pose on the new robot, considering its different kinematics.
- **Predicted Angles for BRAS-DEL**: The final output is a 7-dimensional joint angle vector for BRAS-DEL. This vector can be executed directly on the new robot to achieve the desired behavior inferred from the Sawyer robot's demonstration.

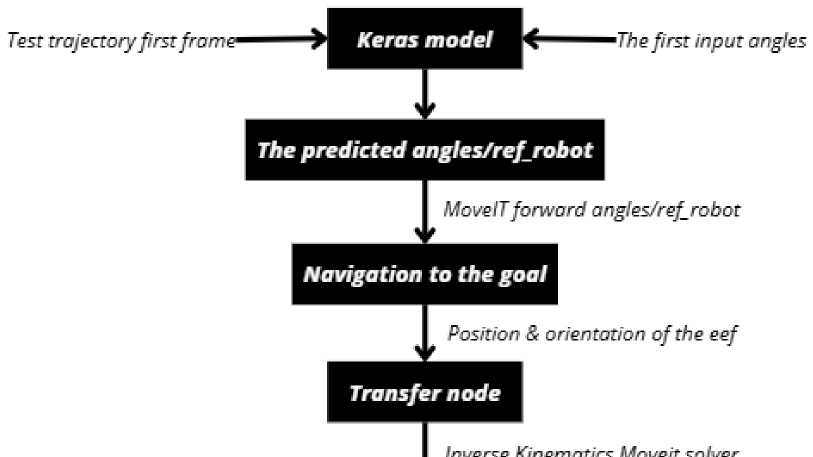
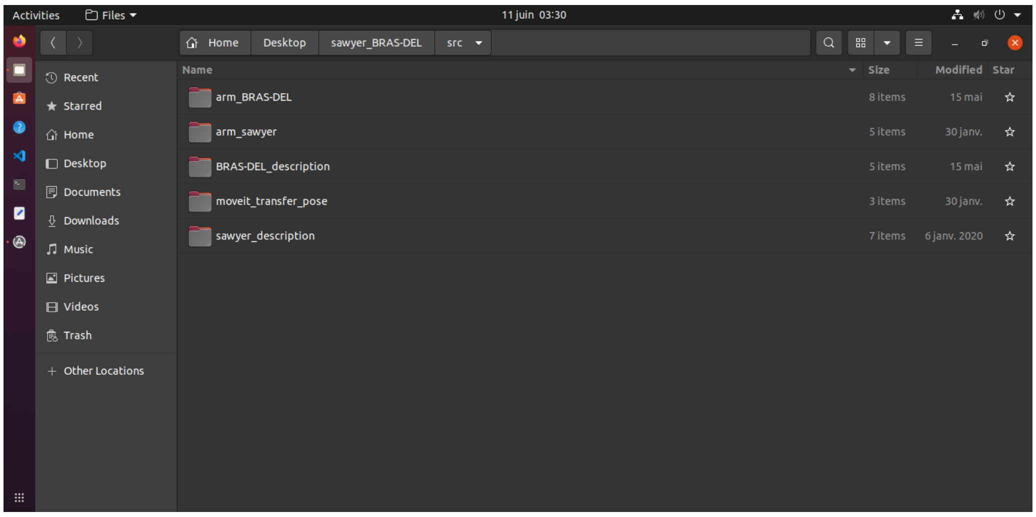

Test trajectory first frame
Keras model
The first input angles
The predicted angles/ref_robot
MoveIT forward angles/ref_robot
Navigation to the goal
Position & orientation of the eef
Transfer node
Inverse Kinematics Moveit solver
The predicted angles/our_robot

Figure 5.25: BRAS-DEL test setup architecture

5.2.7.2.2 Test setup on ROS We performed our experiment, we took a test image joined by its current $Qpos_i[6]$ and forward it into the trained model to get the predicted $Qpos_{i+1}[6]$. But the $Qpos_{i+1}[6]$ are in the Sawyer robot reference, to transfer the result into our robot (BRAS-DEL robot), we follow these steps :

5.2.7.2.2.1 ROS Workspace setup : In order to setup the test on ROS(Robot Operating System), we need to add the both robots moveIt folders into our workspace.



Name	Size	Modified	Star
arm_BRAS-DEL	8 items	15 mai	☆
arm_sawyer	5 items	30 janv.	☆
BRAS-DEL_description	5 items	15 mai	☆
moveit_transfer_pose	3 items	30 janv.	☆
sawyer_description	7 items	6 janv. 2020	☆

Figure 5.26: ROS test workspace setup

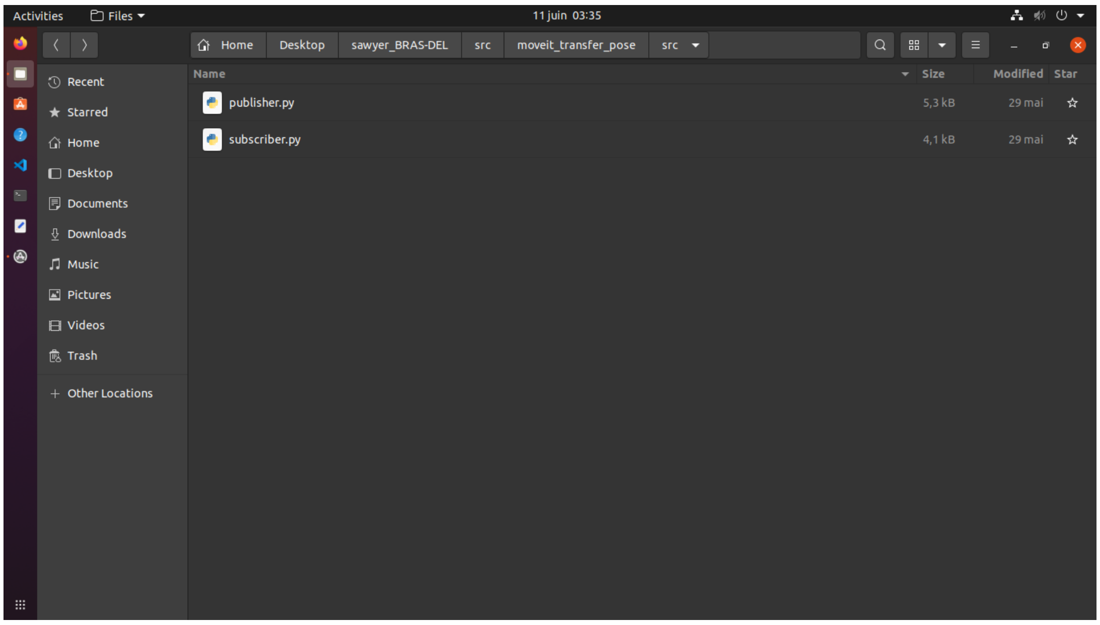
- **arm_sawyer :** The pkg that contains the moveit setup of the sawyer robot(the refernce robot).

- **arm_BRAS-DEL :** The pkg that contains the moveit setup of the BRAS-DEL robot(our robot).

- **sawyer_description :** The pkg that contains the URDF (Unified Robot Description File) in addition to the meshes of the sawyer robot.

- **BRAS-DEL_description :** The pkg that contains the URDF (Unified Robot Description File) in addition to the meshes of the BRAS-DEL robot.

- **moveit_transfer_pose :** The pkg that performs the result transfer from the sawyer robot to the BRAS-DEL robot.



Home / Desktop / sawyer_BRAS-DEL / src / moveit_transfer_pose / src

Name	Size	Modified	Star
publisher.py	5,3 kB	29 mai	☆
subscriber.py	4,1 kB	29 mai	☆

Figure 5.27: Moveit_transfer_pose pkg

5.2.7.2.2.2 ROS rqt graph architecture :

- **robot1_position_orientation_publisher(publisher.py)**:This node performs the given Qpos[6](that we got from the model prediction) on the refernce robot (sawyer) using its forward kinematics and then publishes the pose (position and orientation) of the EEF as topics.

- **robot1_position & robot_orientation (topics)** : They contain the position and orientation of the EEF published by the publisher.py node.

- **robot1_position_orientation_subscriber(subscriber.py)**:This node subscribe to the previous topics and set that pose as target pose for the second robot (BRAS-DEL robot which is our robot) and then navigates to that point using the inverse kinematics solvers to get the combinaison of joint angles in order to get to that EEF pose.

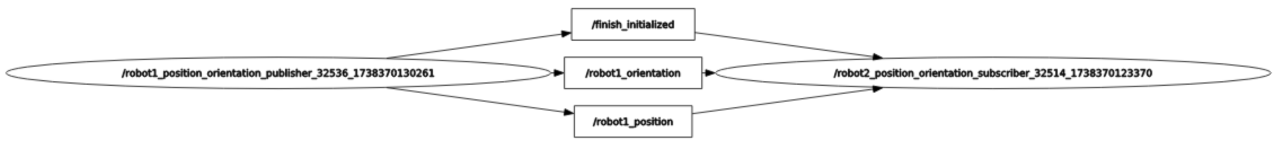

/finish_initialized
/robot1_position_orientation_publisher_32536_1738370130261
/robot1_orientation
/robot2_position_orientation_subscriber_32514_1738370123370
/robot1_position

Figure 5.28: Test ROS architecture of nodes and topics

5.2.7.2.2.3 Test result on RviZ Now we want to transfer the resulted pose from the previously obtained joint angles shown in Table 5.7 of test trajectory n° 2 used in the sawyer robot to our robot (BRAS-DEL robot):

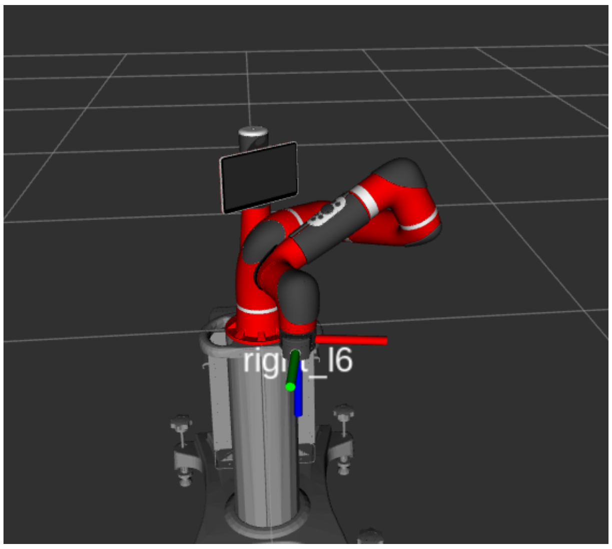


Figure 5.29: The sawyer pose

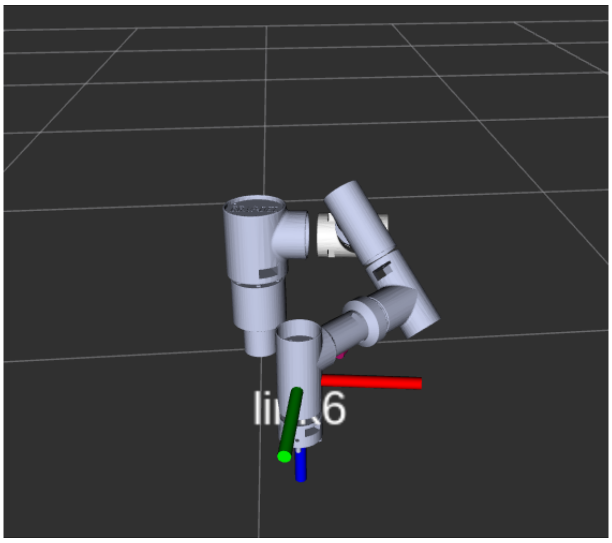


Figure 5.30: The BRAS-DEL pose

Here's another point of view (POV) of this experiment :

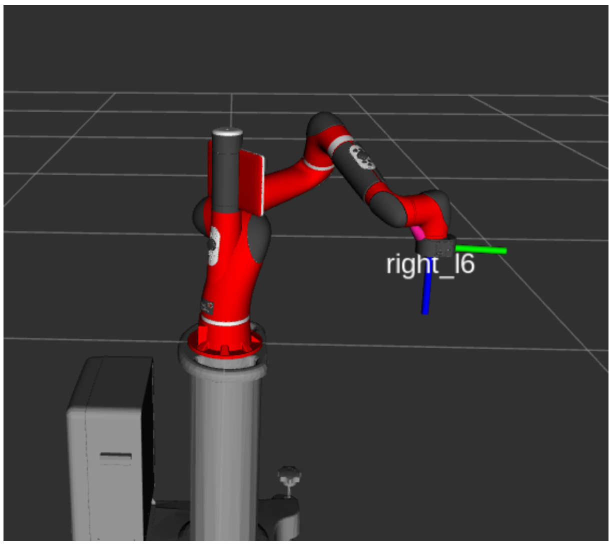


Figure 5.31: The sawyer pose with another POV

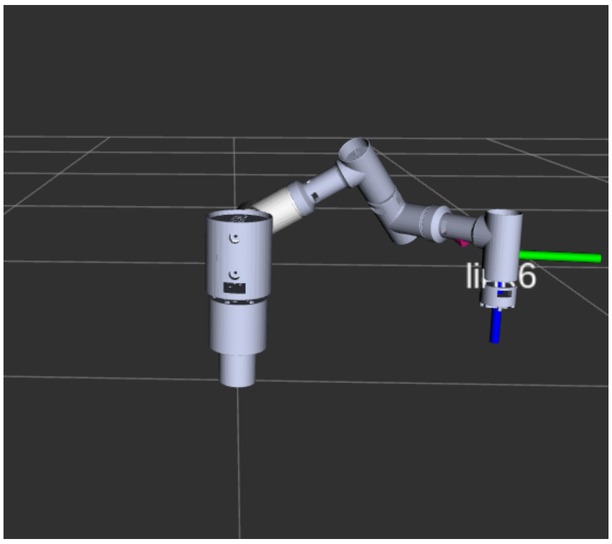
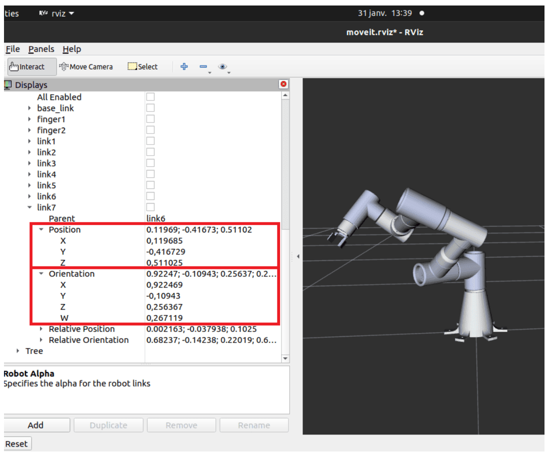


Figure 5.32: The BRAS-DEL pose with another POV

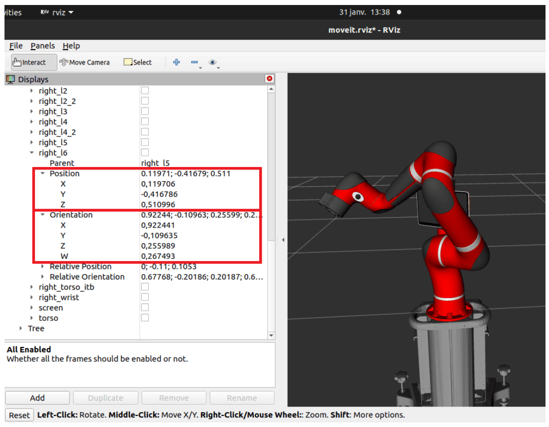
As illustrated in the figure, although the two robots differ in their geometry and kinematic structures, they both successfully reach the same end-effector pose (position and orientation) and exhibit comparable motion behavior.

5.2.7.2.2.4 Other Test results on RviZ We took random generated $Qpos[6]$ and applies them into the robot sawyer and then we perform the result transfer.



```
ties   rviz                      31 janv. 13:39
moveit.rviz* - RViz
File  Panels  Help
Interact  Move Camera  Select
Displays
    All Enabled
  ▸ base_link
  ▸ finger1
  ▸ finger2
  ▸ link1
  ▸ link2
  ▸ link3
  ▸ link4
  ▸ link5
  ▸ link6
  ▾ link7
      Parent                  link6
    ▾ Position                0.11969; -0.41673; 0.51102
        X                     0,119685
        Y                     -0,416729
        Z                     0,511025
    ▾ Orientation             0.92247; -0.10943; 0.25637; 0.2…
        X                     0,922469
        Y                     -0,10943
        Z                     0,256367
        W                     0,267119
    ▸ Relative Position       0.002163; -0.037938; 0.1025
    ▸ Relative Orientation    0.68237; -0.14238; 0.22019; 0.6…
▸ Tree
Robot Alpha
Specifies the alpha for the robot links
Add  Duplicate  Remove  Rename
Reset
```

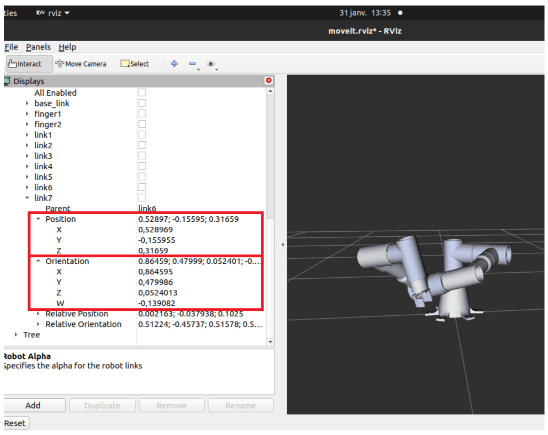
Figure 5.33: Our robot test 1



```
ities   rviz                      31 janv. 13:38
moveit.rviz* - RViz
File  Panels  Help
Interact  Move Camera  Select
Displays
  ▸ right_l2
  ▸ right_l2_2
  ▸ right_l3
  ▸ right_l4
  ▸ right_l4_2
  ▸ right_l5
  ▾ right_l6
      Parent                  right_l5
    ▾ Position                0.11971; -0.41679; 0.511
        X                     0,119706
        Y                     -0,416786
        Z                     0,510996
    ▾ Orientation             0.92244; -0.10963; 0.25599; 0.2…
        X                     0,922441
        Y                     -0,109635
        Z                     0,255989
        W                     0,267493
    ▸ Relative Position       0; -0.11; 0.1053
    ▸ Relative Orientation    0.67768; -0.20186; 0.20187; 0.6…
  ▸ right_torso_itb
  ▸ right_wrist
  ▸ screen
  ▸ torso
▸ Tree
All Enabled
Whether all the frames should be enabled or not.
Add  Duplicate  Remove  Rename
Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
```

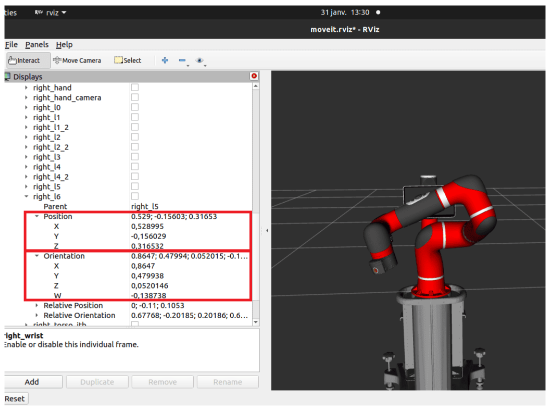
Figure 5.34: Reference robot test 1

We the same logic we performed another test with other joint angles configuration that gives us another pose :



```
ties   rviz                      31 janv. 13:35
moveit.rviz* - RViz
File  Panels  Help
Interact  Move Camera  Select
Displays
    All Enabled
  ▸ base_link
  ▸ finger1
  ▸ finger2
  ▸ link1
  ▸ link2
  ▸ link3
  ▸ link4
  ▸ link5
  ▸ link6
  ▾ link7
      Parent                  link6
    ▾ Position                0.52897; -0.15595; 0.31659
        X                     0,528969
        Y                     -0,155955
        Z                     0,31659
    ▾ Orientation             0.86459; 0.47999; 0.052401; -0…
        X                     0,864595
        Y                     0,479986
        Z                     0,0524013
        W                     -0,139082
    ▸ Relative Position       0.002163; -0.037938; 0.1025
    ▸ Relative Orientation    0.51224; -0.45737; 0.51578; 0.5…
▸ Tree
Robot Alpha
Specifies the alpha for the robot links
Add  Duplicate  Remove  Rename
Reset
```

Figure 5.35: Our robot test 2



```
ties   rviz                      31 janv. 13:30
moveit.rviz* - RViz
File  Panels  Help
Interact  Move Camera  Select
Displays
  ▸ right_hand
  ▸ right_hand_camera
  ▸ right_l0
  ▸ right_l1
  ▸ right_l1_2
  ▸ right_l2
  ▸ right_l2_2
  ▸ right_l3
  ▸ right_l4
  ▸ right_l4_2
  ▸ right_l5
  ▾ right_l6
      Parent                  right_l5
    ▾ Position                0.529; -0.15603; 0.31653
        X                     0,528995
        Y                     -0,156029
        Z                     0,316532
    ▾ Orientation             0.8647; 0.47994; 0.052015; -0.1…
        X                     0,8647
        Y                     0,479938
        Z                     0,0520146
        W                     -0,138738
    ▸ Relative Position       0; -0.11; 0.1053
    ▸ Relative Orientation    0.67768; -0.20185; 0.20186; 0.6…
right_wrist
Enable or disable this individual frame.
Add  Duplicate  Remove  Rename
Reset
```

Figure 5.36: Reference robot test 2

The robot	X position	Y position	Z position	X orientation	Y orientation	Z orientation
Sawyer robot test 1	0.119706	-0.416786	0.510996	0.922441	-0.109635	0.255989
BRAS-DEL robot test 1	0.119685	-0.416729	0.511025	0.922469	-0.10943	0.256367
Sawyer robot test 2	0.528995	-0.156029	0.316532	0.8647	0.479938	-0.138738
BRAS-DEL robot test 2	0.528969	-0.155955	0.31659	0.864595	0.479986	-0.139082

Table 5.15: Table of comparison of the EEF X,Y & Z axes positions and orientations between sawyer robot and our robot

As shown in Table 5.15, figures 5.33, 5.34, 5.35 & 5.36 , the End Effector (EEF) positions and orientations of the BRAS-DEL robot closely match those of the Sawyer robot across both test scenarios. Despite differences in the robots' physical structures and kinematics, the inverse kinematics solver applied to the BRAS-DEL robot successfully replicates the target EEF poses. The positional and orientational differences are minimal, demonstrating the system's effectiveness in achieving accurate pose imitation.

5.3 Reinforcement learning

5.3.1 The approach

After explaining the general idea of reinforcement learning and its most important components, we will start now by explaining the first approach we used during our work. The main idea of this approach is to teach our own robot model BRAS-DEL, on how to perform the pick and place task for a certain object, so reaching the goal object by minimizing the distance between the end effector and setting a good orientation too. The main goal of this approach is to get a model that will be used instead of the standard IK solver previously explained in the ROS and moveIt chapter.

5.3.2 Unity framework

Unity is a powerful and widely-used cross-platform game engine and real-time development platform, providing a comprehensive framework and integrated development environment (IDE) for creating interactive 2D and 3D content. Unity provides a robust set of tools and a structured environment (the "framework") that handles many of the complex underlying technical aspects of interactive content creation, allowing developers to focus more on the creative and design elements of their projects.

Figure 5.37: Unity framework logo

5.3.3 Scene setup

Working with the Unity framework, the first thing we need to do is setting up the simulation environment. In our case, the most important elements are the robot itself, the object we need to pick up and finally the ground space. But we should understand how to create these elements in the scene?
Starting by downloading the Unity editor, in our case we used the 2020.2.1f1 that gives us the following interface to start with an empty world.

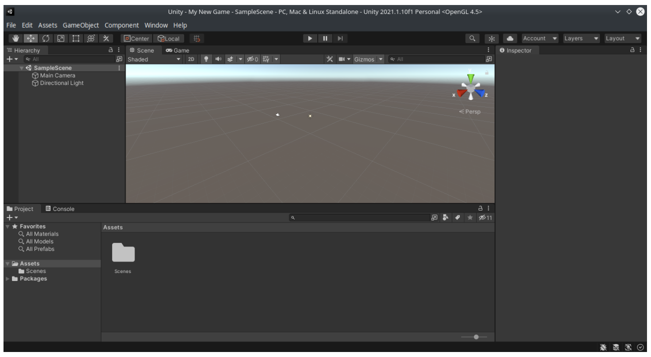

Unity - My New Game - SampleScene - PC, Mac & Linux Standalone - Unity 2021.1.10f1 Personal <OpenGL 4.5>
File Edit Assets GameObject Component Window Help
Hierarchy
SampleScene
Main Camera
Directional Light
Scene
Game
Inspector
Project
Console
Favorites
Assets
Scenes
Packages

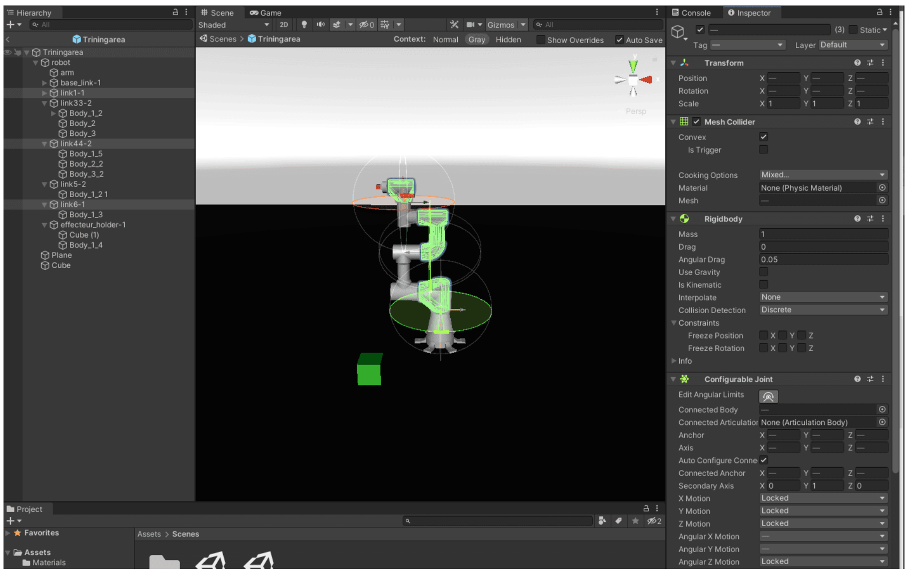
Figure 5.38: Unity 2020.2.1f1 editor interface

So basically, the editor gives you the interface to set up your scene by providing pre-created shapes (circles, squares, etc..), it provides a folder interface where u can access your codes and designs, where u can also save your created scenes too.
Now for us to create our own scene, we start by loading the STL files of our robot that we already saved during the conception part and upload them into unity.

Next, we need to set the ground space which is going to be just a simple flat square. And the required object to lift which is going to be a small square too. Using the predefined shapes and available colors in the editor, this will be an easy task to do.
For now, we just created the scene but we still need to add some modifications before we save it and start the simulation. We need to know that each elements in unity should be defined by :

- **Meshes :** This is the one we already created that represents the 3D/2D geometry of the elements in the scene.
- **The mesh colliders :** The Mesh collider builds its collision geometry to match an assigned Mesh, including its shape, position and scale. The benefit of this is that you can make the shape of the collider exactly the same as the shape of the visible Mesh for the GameObject, which creates more precise and realistic collisions[7].
- **The configurable joint :** Used to customize the movement of a ragdoll and enforce certain poses on your characters. You can also use them to adapt joints into highly specialized joints of your own design[8], using this u can set the center and axe of rotation alongside the movement limitations.

So we are going to add for the goal object, ground and each link of the robot a mesh collider in order to detect the collision between them, then we set the rotatif movement of each joint using the configurable joints. At the end, we will be getting the following figure result for our robot scene which is going to be saved for later work and simulation, where the green cube is the goal object to reach, the black space is the ground, our robotic arm and the green space around the links represented the mesh colliders.

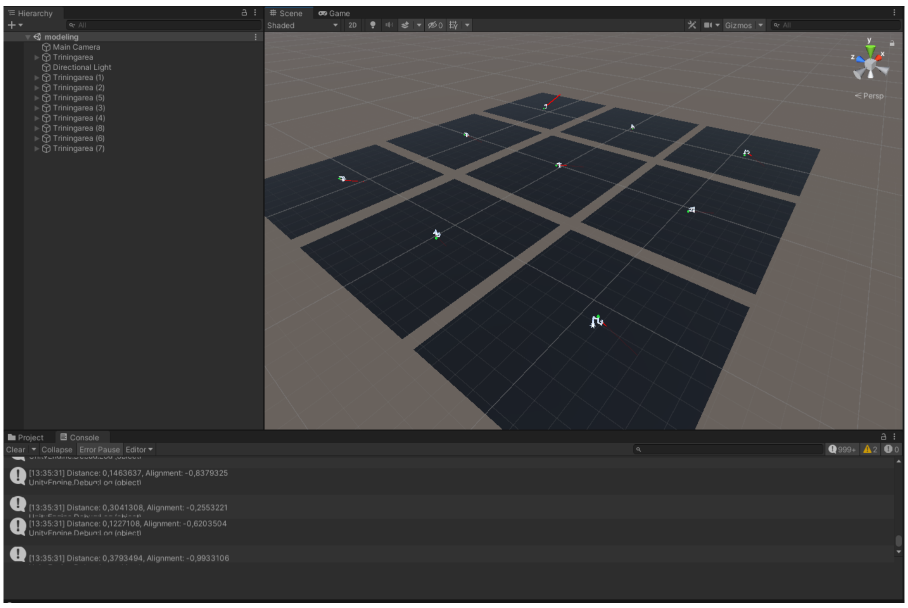


```
Triningarea
  robot
    arm
    base_link-1
    link1-1
    link33-2
      Body_1_2
      Body_2
      Body_3
    link44-2
      Body_1_5
      Body_2_2
      Body_3_2
    link5-2
      Body_1_2 1
    link6-1
      Body_1_3
    effecteur_holder-1
      Cube (1)
      Body_1_4
  Plane
  Cube
```

```
(3)  Static
Tag —            Layer Default
Transform
  Position     X —  Y —  Z —
  Rotation     X —  Y —  Z —
  Scale        X 1  Y 1  Z 1
Mesh Collider
  Convex               ✓
    Is Trigger
  Cooking Options      Mixed...
  Material             None (Physic Material)
  Mesh                 —
Rigidbody
  Mass                 1
  Drag                 0
  Angular Drag         0.05
  Use Gravity
  Is Kinematic
  Interpolate          None
  Collision Detection  Discrete
  Constraints
    Freeze Position    X Y Z
    Freeze Rotation    X Y Z
  Info
Configurable Joint
  Edit Angular Limits
  Connected Body       —
  Connected Articulatior None (Articulation Body)
  Anchor               X —  Y —  Z —
  Axis                 X —  Y —  Z —
  Auto Configure Conne ✓
  Connected Anchor     X —  Y —  Z —
  Secondary Axis       X 0  Y 1  Z 0
  X Motion             Locked
  Y Motion             Locked
  Z Motion             Locked
  Angular X Motion     —
  Angular Y Motion     —
  Angular Z Motion     Locked
```

Figure 5.39: BRAS-DEL scene setup

This same scene can be used when we start the training. Using it, we are going to create a training farm that contains many of the same scenes running in parallel. This method can help us save some time since we have more than just one robot training to reach the goal object and each one independently from the others, but this method requires high computational resources to be done. So at the end it gives us the following farm of 9 scenes for our case.



```
[13:35:31] Distance: 0,1463637, Alignment: -0,8379325
UnityEngine.Debug:Log (object)
[13:35:31] Distance: 0,3041308, Alignment: -0,2553221
[13:35:31] Distance: 0,1227108, Alignment: -0,6203504
UnityEngine.Debug:Log (object)
[13:35:31] Distance: 0,3793494, Alignment: -0,9933106
```

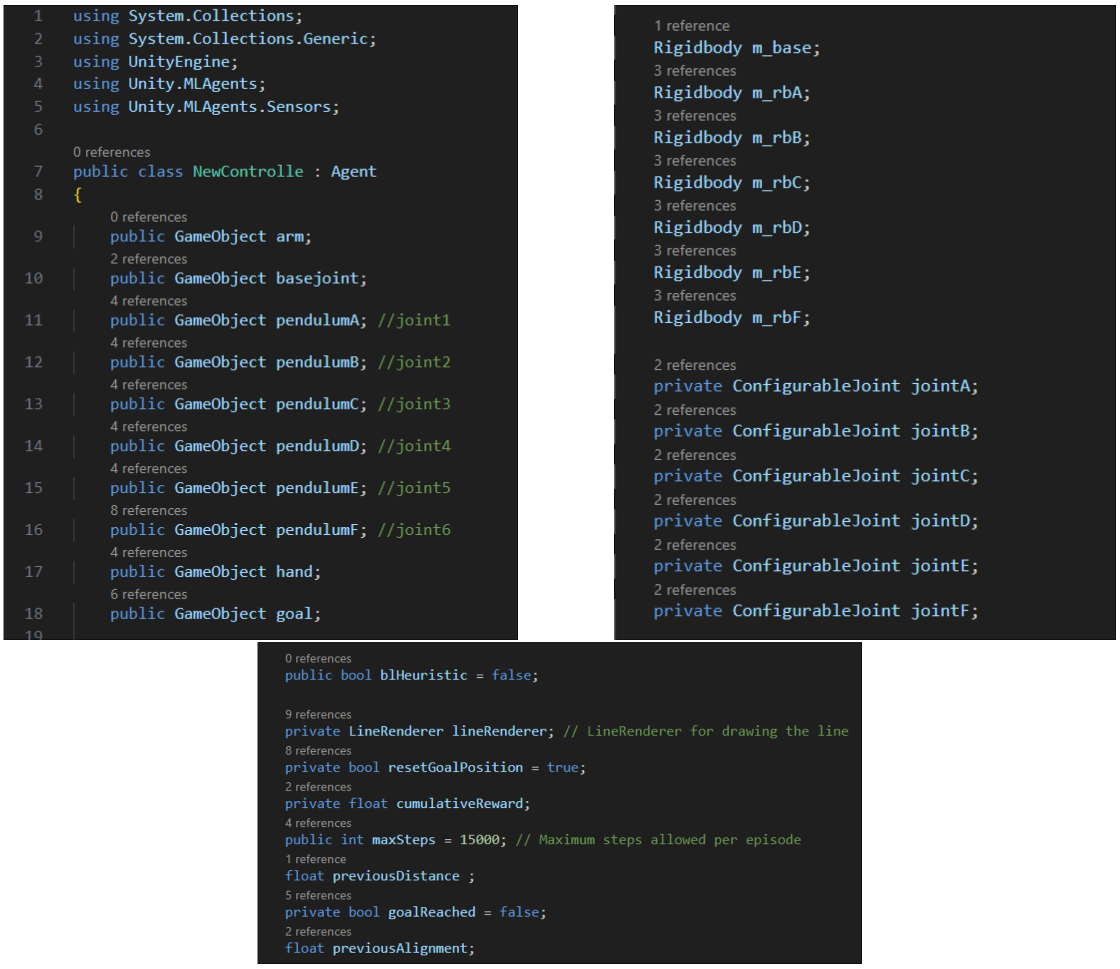
Figure 5.40: Training farm of 9 scenes

5.3.4 Training in unity

In order to start the training, we will need to go by a Unity library called ML-Agents. The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents. We provide implementations (based on PyTorch) of state-of-the-art algorithms to enable game developers and hobbyists to easily train intelligent agents for 2D, 3D and VR/AR games. Researchers can also use the provided simple-to-use Python API to train Agents using reinforcement learning, imitation learning, neuro evolution, or any other methods[9]. So, for our case of use we are going to download the ML-Agents library 1.0.8 version, which will be serving as a liaison between the unity scene that we created and the rest of python libraries we need to use such as PyTorch for the training.

After downloading the ML-Agents library, we should now create an agent script .cs which is going to be the one controlling the robot, collecting observations and in which we are going to set the reward tree of our training that we will talk about after. Now we will be breaking down the code and explaining each part of it, how it is related to the scene and how to use it for the train.
Starting by importing libraries and defining our agent class which is going to be related with the agent in the scene. The class will control the behavior of the agent (Which is the robotic arm in our case), will be attaching for each joint and link its type of movement.

So, as it is shown in the figures this is how we declare the agent class. In it we start by declaring the rigidbodies of each link because we are going to use them after, the scene objects such as the ground, goal object and obviously the arm and some random variables we use during the work such as goalreached which is a boolean variable to set if we reached the goal object or not or the max number of steps.



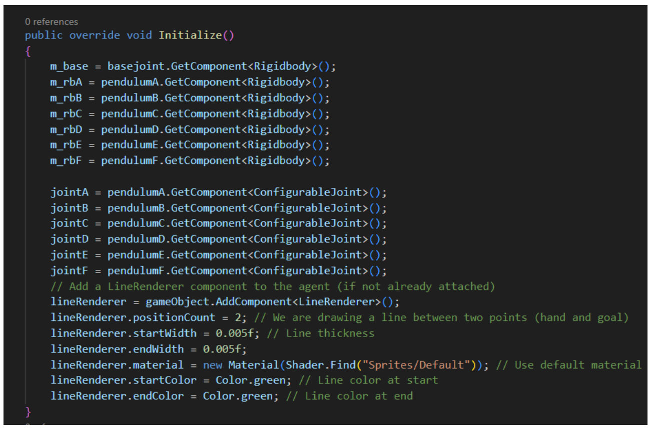
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;

0 references
public class NewControlle : Agent
{
    0 references
    public GameObject arm;
    2 references
    public GameObject basejoint;
    4 references
    public GameObject pendulumA; //joint1
    4 references
    public GameObject pendulumB; //joint2
    4 references
    public GameObject pendulumC; //joint3
    4 references
    public GameObject pendulumD; //joint4
    4 references
    public GameObject pendulumE; //joint5
    8 references
    public GameObject pendulumF; //joint6
    4 references
    public GameObject hand;
    6 references
    public GameObject goal;
```

```
1 reference
Rigidbody m_base;
3 references
Rigidbody m_rbA;
3 references
Rigidbody m_rbB;
3 references
Rigidbody m_rbC;
3 references
Rigidbody m_rbD;
3 references
Rigidbody m_rbE;
3 references
Rigidbody m_rbF;

2 references
private ConfigurableJoint jointA;
2 references
private ConfigurableJoint jointB;
2 references
private ConfigurableJoint jointC;
2 references
private ConfigurableJoint jointD;
2 references
private ConfigurableJoint jointE;
2 references
private ConfigurableJoint jointF;
```

```
0 references
public bool blHeuristic = false;

9 references
private LineRenderer lineRenderer; // LineRenderer for drawing the line
8 references
private bool resetGoalPosition = true;
2 references
private float cumulativeReward;
4 references
public int maxSteps = 15000; // Maximum steps allowed per episode
1 reference
float previousDistance ;
5 references
private bool goalReached = false;
2 references
float previousAlignment;
```

Figure 5.41: Class agent declaration

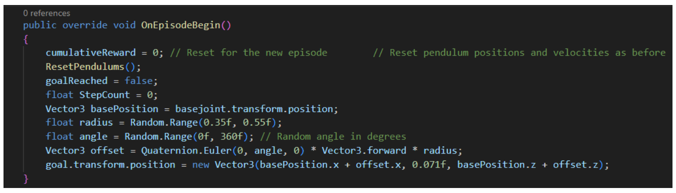
Then we will jump to the first function we use which is Initialize(). This function is used at the beginning of the train one time only, so it gives and initializes some variables as shown in the following figure, we set up the drawing lines noted LineRender, and we attach the links for each rigidbody using the predefined getcomponent function of unity.



```
0 references
public override void Initialize()
{
    m_base = basejoint.GetComponent<Rigidbody>();
    m_rbA = pendulumA.GetComponent<Rigidbody>();
    m_rbB = pendulumB.GetComponent<Rigidbody>();
    m_rbC = pendulumC.GetComponent<Rigidbody>();
    m_rbD = pendulumD.GetComponent<Rigidbody>();
    m_rbE = pendulumE.GetComponent<Rigidbody>();
    m_rbF = pendulumF.GetComponent<Rigidbody>();

    jointA = pendulumA.GetComponent<ConfigurableJoint>();
    jointB = pendulumB.GetComponent<ConfigurableJoint>();
    jointC = pendulumC.GetComponent<ConfigurableJoint>();
    jointD = pendulumD.GetComponent<ConfigurableJoint>();
    jointE = pendulumE.GetComponent<ConfigurableJoint>();
    jointF = pendulumF.GetComponent<ConfigurableJoint>();
    // Add a LineRenderer component to the agent (if not already attached)
    lineRenderer = gameObject.AddComponent<LineRenderer>();
    lineRenderer.positionCount = 2; // We are drawing a line between two points (hand and goal)
    lineRenderer.startWidth = 0.005f; // Line thickness
    lineRenderer.endWidth = 0.005f;
    lineRenderer.material = new Material(Shader.Find("Sprites/Default")); // Use default material
    lineRenderer.startColor = Color.green; // Line color at start
    lineRenderer.endColor = Color.green; // Line color at end
}
```

Figure 5.42: Initialize() function

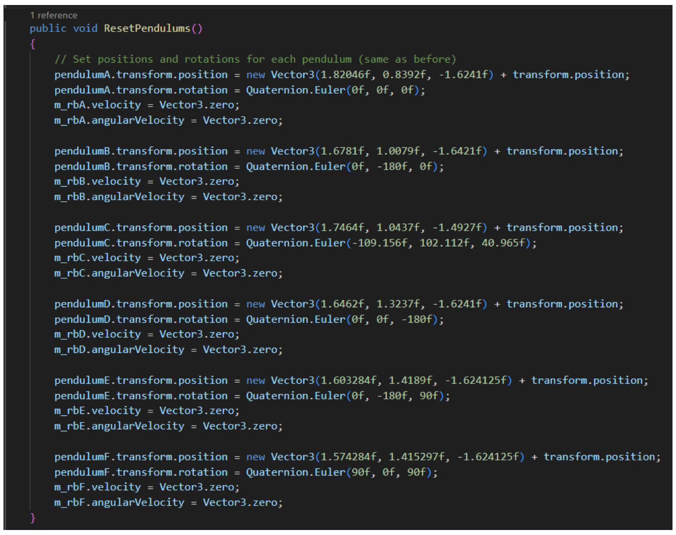
Then, we have the OnEpisodeBegin() function. This is the function we call at the beginning of each episode, so it's not used only once as the initialize one. Its role is to setup the episode, reset the number of steps, we call inside of this function some other functions we are going to explain next that reset the links to their initial position and move the goal object to a new position in the reachable space of the robotic arm as it is shown in this figure.



```
0 references
public override void OnEpisodeBegin()
{
    cumulativeReward = 0; // Reset for the new episode        // Reset pendulum positions and velocities as before
    ResetPendulums();
    goalReached = false;
    float StepCount = 0;
    Vector3 basePosition = basejoint.transform.position;
    float radius = Random.Range(0.35f, 0.55f);
    float angle = Random.Range(0f, 360f); // Random angle in degrees
    Vector3 offset = Quaternion.Euler(0, angle, 0) * Vector3.forward * radius;
    goal.transform.position = new Vector3(basePosition.x + offset.x, 0.071f, basePosition.z + offset.z);
}
```

Figure 5.43: OneEpisodeBegin() function

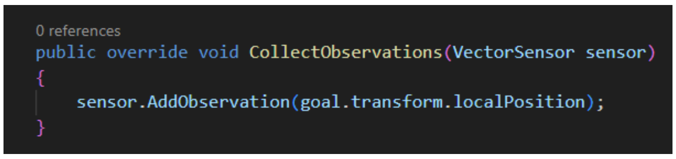
After that, we have the ResetPendulums() function. This is the function that sets the links each one for its starting position which should be the same for all angles to zero value. So after setting up the scene manually in the editor, we will need to look for the position and orientation of each link from the inspector panel, then add it manually to the function in the code so it gets to be called at the beginning of each episode by the previously explained function. This figure shows how is the ResetPendulums function declared,



```
1 reference
public void ResetPendulums()
{
    // Set positions and rotations for each pendulum (same as before)
    pendulumA.transform.position = new Vector3(1.82046f, 0.8392f, -1.6241f) + transform.position;
    pendulumA.transform.rotation = Quaternion.Euler(0f, 0f, 0f);
    m_rbA.velocity = Vector3.zero;
    m_rbA.angularVelocity = Vector3.zero;

    pendulumB.transform.position = new Vector3(1.6781f, 1.0079f, -1.6421f) + transform.position;
    pendulumB.transform.rotation = Quaternion.Euler(0f, -180f, 0f);
    m_rbB.velocity = Vector3.zero;
    m_rbB.angularVelocity = Vector3.zero;

    pendulumC.transform.position = new Vector3(1.7464f, 1.0437f, -1.4927f) + transform.position;
    pendulumC.transform.rotation = Quaternion.Euler(-109.156f, 102.112f, 40.965f);
    m_rbC.velocity = Vector3.zero;
    m_rbC.angularVelocity = Vector3.zero;

    pendulumD.transform.position = new Vector3(1.6462f, 1.3237f, -1.6241f) + transform.position;
    pendulumD.transform.rotation = Quaternion.Euler(0f, 0f, -180f);
    m_rbD.velocity = Vector3.zero;
    m_rbD.angularVelocity = Vector3.zero;

    pendulumE.transform.position = new Vector3(1.603284f, 1.4189f, -1.624125f) + transform.position;
    pendulumE.transform.rotation = Quaternion.Euler(0f, -180f, 90f);
    m_rbE.velocity = Vector3.zero;
    m_rbE.angularVelocity = Vector3.zero;

    pendulumF.transform.position = new Vector3(1.574284f, 1.415297f, -1.624125f) + transform.position;
    pendulumF.transform.rotation = Quaternion.Euler(90f, 0f, 90f);
    m_rbF.velocity = Vector3.zero;
    m_rbF.angularVelocity = Vector3.zero;
}
```

Figure 5.44: ResetPendulums() function

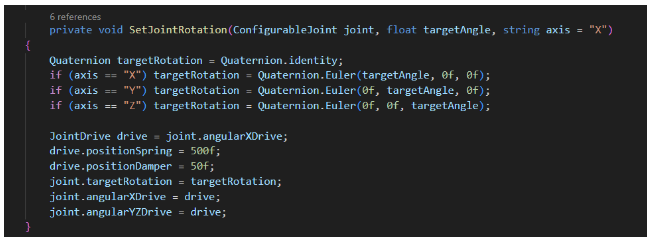
Then we declare one of the most important functions which is CollectObservation() which is going to be used to collect the required information from the scene. The next figure will show us how it is declared. We will see that we only ask for the goal position object in world space, since our approach is used to replace the IK solver and as previously explained the IK solver takes the desired position we want to reach and find out the joint angles required to reach that position. So with that the observation space of ours will only have 3 elements (X,Y,Z coordinates of the goal object).



```
0 references
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(goal.transform.localPosition);
}
```

Figure 5.45: CollectObservations() function

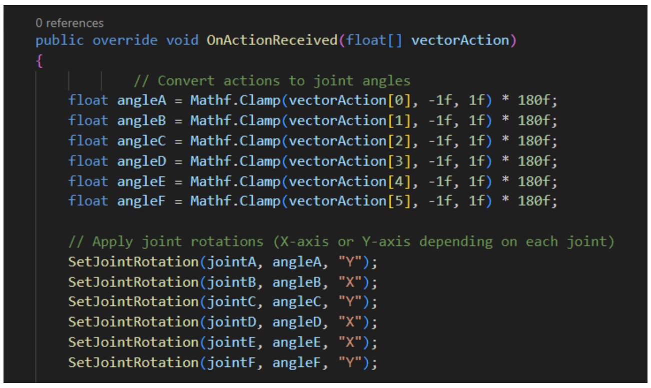
Before continuing, we are going to declare a special function which is special just with our use case, not like all the previous ones which are mandatory for Reinforcement learning using ML-agents and Unity. This function is the SetJointRotation() which sets the rotation axis and applies the angles on each joint in degrees as it is shown in the figure.



```
    6 references
    private void SetJointRotation(ConfigurableJoint joint, float targetAngle, string axis = "X")
{
    Quaternion targetRotation = Quaternion.identity;
    if (axis == "X") targetRotation = Quaternion.Euler(targetAngle, 0f, 0f);
    if (axis == "Y") targetRotation = Quaternion.Euler(0f, targetAngle, 0f);
    if (axis == "Z") targetRotation = Quaternion.Euler(0f, 0f, targetAngle);

    JointDrive drive = joint.angularXDrive;
    drive.positionSpring = 500f;
    drive.positionDamper = 50f;
    joint.targetRotation = targetRotation;
    joint.angularXDrive = drive;
    joint.angularYZDrive = drive;
}
```

Figure 5.46: SetJointRotation() function

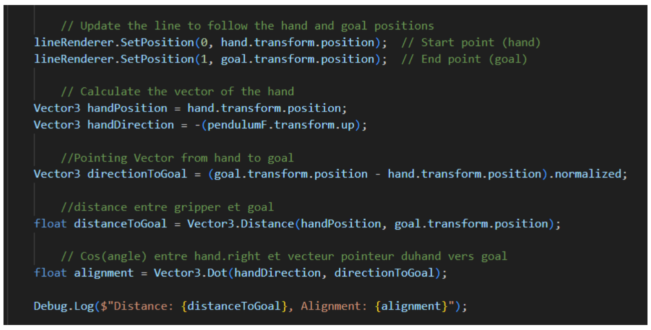
After finishing this one, we have one of the necessary functions of the training which is OneActionReceived(). This function is used to apply the received action from the policy and apply them on each joint, in addition inside of this function we declare the reward tree we're working with. We should understand that, since we have 6 joints (since it's a 6DOF robotic arm) the action space will have 6 elements (an angle value for each joint). As shown in the figure, we set the values between 1 and -1 then multiply by 180 so all the values we get are between -180 and +180 degrees for each angle. Then we use the SetJointRotation function in order to apply those values on each joint, each one on its axis of rotation.



```
0 references
public override void OnActionReceived(float[] vectorAction)
{
            // Convert actions to joint angles
    float angleA = Mathf.Clamp(vectorAction[0], -1f, 1f) * 180f;
    float angleB = Mathf.Clamp(vectorAction[1], -1f, 1f) * 180f;
    float angleC = Mathf.Clamp(vectorAction[2], -1f, 1f) * 180f;
    float angleD = Mathf.Clamp(vectorAction[3], -1f, 1f) * 180f;
    float angleE = Mathf.Clamp(vectorAction[4], -1f, 1f) * 180f;
    float angleF = Mathf.Clamp(vectorAction[5], -1f, 1f) * 180f;

    // Apply joint rotations (X-axis or Y-axis depending on each joint)
    SetJointRotation(jointA, angleA, "Y");
    SetJointRotation(jointB, angleB, "X");
    SetJointRotation(jointC, angleC, "Y");
    SetJointRotation(jointD, angleD, "X");
    SetJointRotation(jointE, angleE, "X");
    SetJointRotation(jointF, angleF, "Y");
```

Figure 5.47: OneActionReceived() function part 1

We continue with the same function, we declare some temporary variables we use during the work such as the distancetogoal which is the difference between the position of the end effector and the goal object, we also have the Alignment which is the cosine of the angle made between the directing vector from the end effector to the goal and the link6 pointing vector and some saving variables previousdistance and previous alignment which are used to save the distance and alignment values and use them for comparison if we in the reward tree.



```
    // Update the line to follow the hand and goal positions
lineRenderer.SetPosition(0, hand.transform.position);  // Start point (hand)
lineRenderer.SetPosition(1, goal.transform.position);  // End point (goal)

    // Calculate the vector of the hand
Vector3 handPosition = hand.transform.position;
Vector3 handDirection = -(pendulumF.transform.up);

    //Pointing Vector from hand to goal
Vector3 directionToGoal = (goal.transform.position - hand.transform.position).normalized;

    //distance entre gripper et goal
float distanceToGoal = Vector3.Distance(handPosition, goal.transform.position);

    // Cos(angle) entre hand.right et vecteur pointeur duhand vers goal
float alignment = Vector3.Dot(handDirection, directionToGoal);

Debug.Log($"Distance: {distanceToGoal}, Alignment: {alignment}");
```

Figure 5.48: OneActionReceived() function part 2

5.3.5 Reward tree

Before setting up the reward tree for our training, we should understand that it's very crucial. Because it does structure the way our robot is going to learn the way to perform the task so it should be chosen carefully.
At first, since the robot doesn't know anything about how to do it, we should try and keep it simple for it so it gets to understand the basics of robotic arm movement and the main idea of the application. For the beginning we chose a pretty simple tree in which we only used the distance and the alignment as essential inputs.

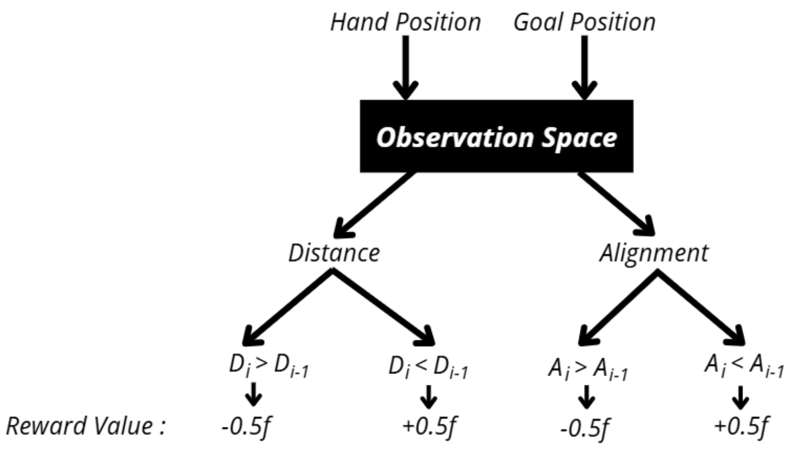
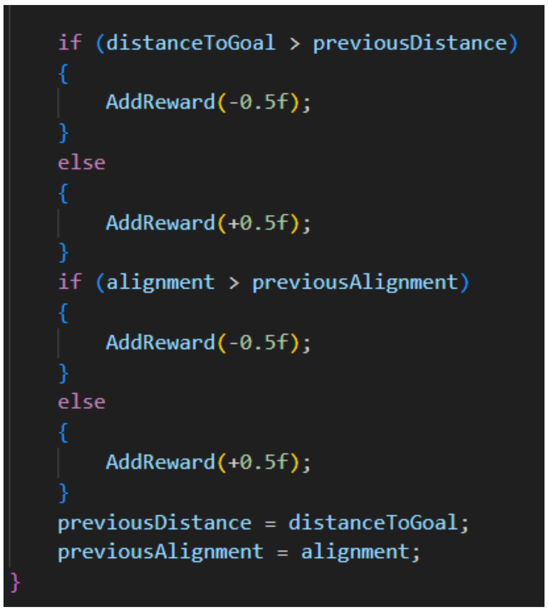


Figure 5.49: Simple reward tree

As we can see in the figure, we did use the hand position (end effector position) and the goal position for each step to calculate the distance and alignment between them. At first we did some gradual rewarding, so each time the robot gets a better alignment or reduces its distance from the goal object we give him a small positve reward +0.5f, and in the same way if he gets far or misses the alignment we give him a little negative reward of -0.5f. We should know that the reward better be between -1 and +1 at the beginning and only give some small rewards and punishment otherwise the policy will prohibit the exploration of the scene in the future and its movements will be only based on the firstly made ones.



```
    if (distanceToGoal > previousDistance)
    {
        AddReward(-0.5f);
    }
    else
    {
        AddReward(+0.5f);
    }
    if (alignment > previousAlignment)
    {
        AddReward(-0.5f);
    }
    else
    {
        AddReward(+0.5f);
    }
    previousDistance = distanceToGoal;
    previousAlignment = alignment;
}
```

Figure 5.50: Simple rewarding system

With the time going on, we should keep surveying the evolution of the reward values and behavior of the robot, so when we make sure that he learned the basics we can start making it more complex. Over steps we started to add some rewards and punishment on the too far distance so the policy understands that we can't have movement that take him 1 meter far away from the goal object, we start giving big punishments when the end effector looks directly in the opposite way of the goal object. In addition we started to add the collision problems, using the mesh colliders we set at the beginning we can detect when 2 elements collide and we use this as an aggressive punishment of -1f since these are untolerated actions except if the end effector gets in collision with the goal object that means it did reach it with good orientation so we give him a big reward of +1f . We do add some slight punishments over steps in order to make it accomplish the task in the minimum possible of steps , so for each step we give a small punishment of -0.05f, and more of a big punishment of -1f if the episode ends before the end effector reaches the goal object. At the end we have got the following complex reward tree that did lead us to some good results.

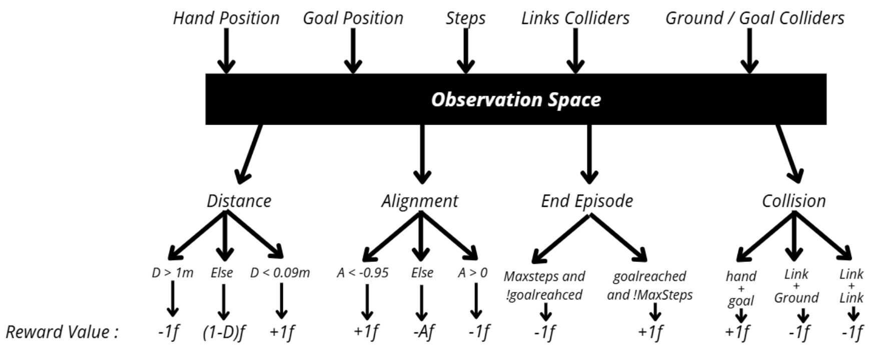
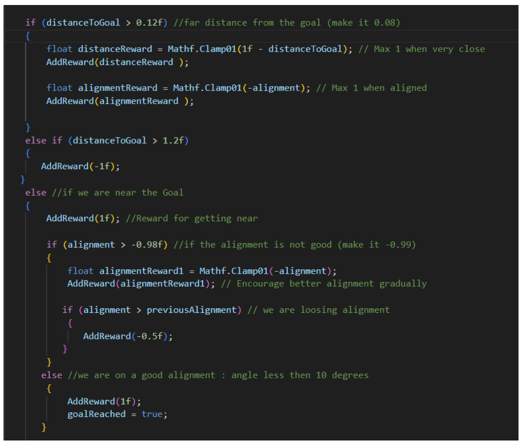
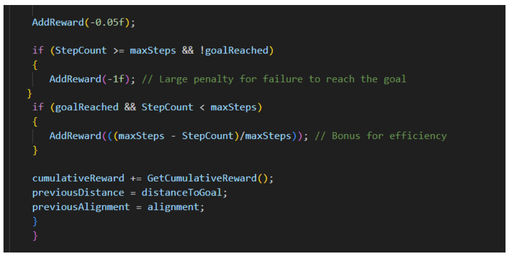


Figure 5.51: Complex reward tree



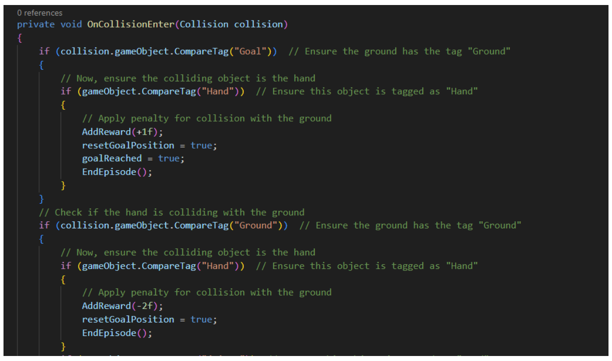
```
if (distanceToGoal > 0.12f) //far distance from the goal (make it 0.08)
{
    float distanceReward = Mathf.Clamp01(1f - distanceToGoal); // Max 1 when very close
    AddReward(distanceReward );

    float alignmentReward = Mathf.Clamp01(-alignment); // Max 1 when aligned
    AddReward(alignmentReward );

}
else if (distanceToGoal > 1.2f)
{
   AddReward(-1f);
}
else //if we are near the Goal
{
    AddReward(1f); //Reward for getting near

    if (alignment > -0.98f) //if the alignment is not good (make it -0.99)
    {
        float alignmentReward1 = Mathf.Clamp01(-alignment);
        AddReward(alignmentReward1); // Encourage better alignment gradually

       if (alignment > previousAlignment) // we are loosing alignment
        {
           AddReward(-0.5f);
       }
    }
   else //we are on a good alignment : angle less then 10 degrees
    {
        AddReward(1f);
        goalReached = true;
   }
```

Figure 5.52: Reward tree part 1



```
AddReward(-0.05f);

if (StepCount >= maxSteps && !goalReached)
{
   AddReward(-1f); // Large penalty for failure to reach the goal
}
if (goalReached && StepCount < maxSteps)
{
   AddReward(((maxSteps - StepCount)/maxSteps)); // Bonus for efficiency
}

cumulativeReward += GetCumulativeReward();
previousDistance = distanceToGoal;
previousAlignment = alignment;
}
}
```

Figure 5.53: Reward tree part 2



```
0 references
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Goal"))  // Ensure the ground has the tag "Ground"
    {
        // Now, ensure the colliding object is the hand
        if (gameObject.CompareTag("Hand"))  // Ensure this object is tagged as "Hand"
        {
            // Apply penalty for collision with the ground
            AddReward(+1f);
            resetGoalPosition = true;
            goalReached = true;
            EndEpisode();
        }
    }
    // Check if the hand is colliding with the ground
    if (collision.gameObject.CompareTag("Ground"))  // Ensure the ground has the tag "Ground"
    {
        // Now, ensure the colliding object is the hand
        if (gameObject.CompareTag("Hand"))  // Ensure this object is tagged as "Hand"
        {
            // Apply penalty for collision with the ground
            AddReward(-2f);
            resetGoalPosition = true;
            EndEpisode();
        }
```

Figure 5.54: Reward tree part 3

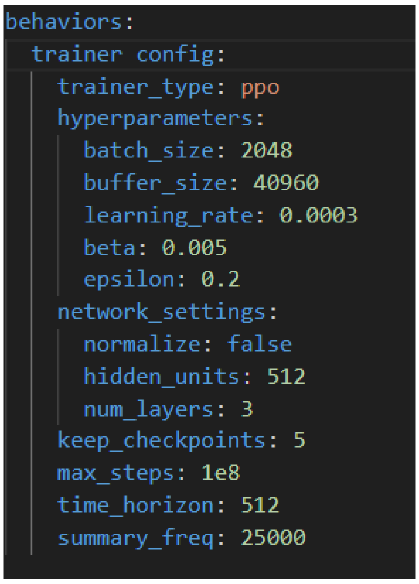
5.3.6 Model architecture

In order to start with the training now, we should set the hyperparameters of our training in a training.yaml file. From this file they will be imported to the model for the training. And the following table shows us the configuration the we chose :

Parameter	**Value**
Trainer Type	PPO
Batch Size	2048
Buffer Size	40960
Learning Rate	0.0003
Beta	0.005
Epsilon	0.2
Normalize	False
Hidden Units	512
Num Layers	3
Max steps	1.10^7
Sum frequency	25000

Table 5.16: Table of training hyperparameters

So as it is listed in the table, we went with the PPO as a trainer type since we have explained the advantage of using it specially since we are working with a continuous space of actions and as previously mentioned it does take into account the next action based on the actual one, so we are keeping the continuity of the movement. Next to it, we chose a batch size of 2048 and a buffer size of 40960 with a learning rate of 0.0003, beta of 0.005 and epsilon of 0.2 (So the trust ratio will be constrained between 0.8 and 1.2). In addition we chose a lambda of 0.95, architecture of 3 layers with 512 hidden units for each. At the end set the max step for 100 million steps before the end of the training while keeping checkpoints each 5 episodes. So at the end we got the following training.yaml file :



```yaml
behaviors:
  trainer config:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 40960
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
    network_settings:
      normalize: false
      hidden_units: 512
      num_layers: 3
    keep_checkpoints: 5
    max_steps: 1e8
    time_horizon: 512
    summary_freq: 25000
```

Figure 5.55: Configuration file

After setting all these parameters, now we can start the training for our robot.

5.3.7 Test & results

For the interesting part now, the test and results. We should firstly know that we have to keep an eye on the scene during the training since we have to change each time the reward tree and sometimes even the architecture in case of the model gets stuck at a local maximum. So during that monitoring we can see from the following pictures that we were starting to get some good results.

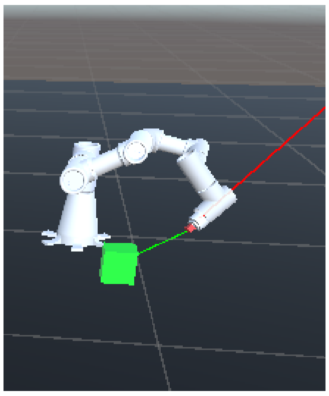


Figure 5.56: Scene 1 of the farm

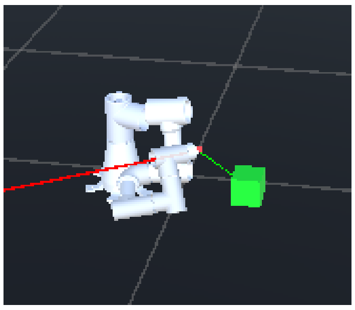


Figure 5.57: Scene 3 of the farm

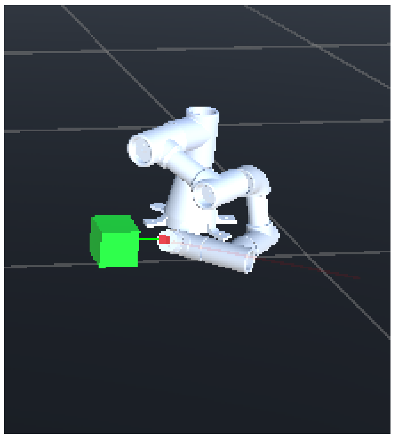


Figure 5.58: Scene 5 of the farm

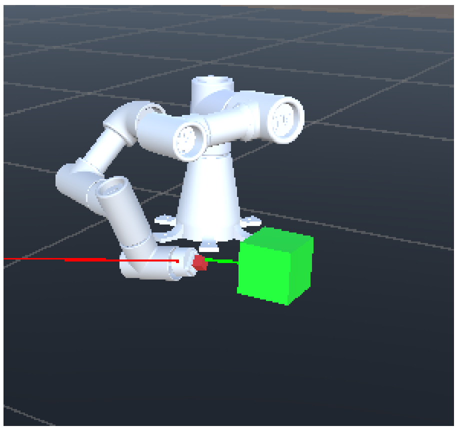


Figure 5.59: Scene 7 of the farm

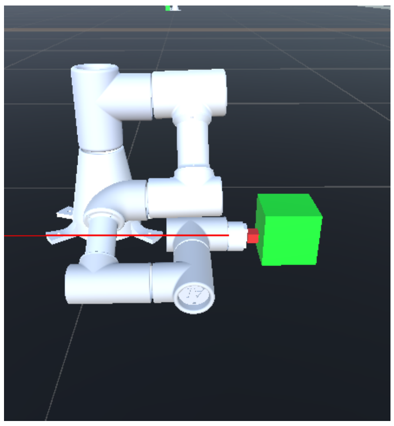


Figure 5.60: Scene 4 of the farm

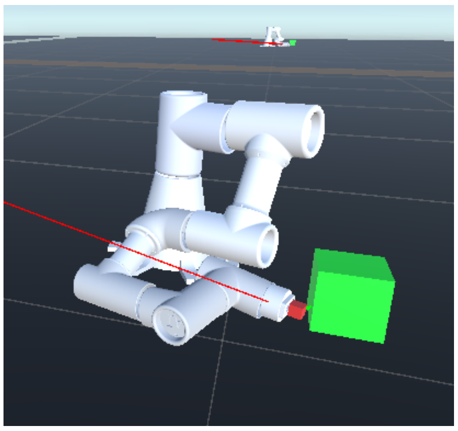


Figure 5.61: Scene 6 of the farm

This is why we pointed at the necessity of monitoring the train but never rush to stop it whenever u see that it's not getting to reach the object, since in our case we have a farm of training and the global policy is learning from the punishment and rewards of all of them at the same time, so it is a good point actually that some of them are miss applying the behavior since it will help in the future to avoid mistakes in the test.
At the end of the training, we could get the loss and cumulative reward by steps.

5.3.7.1 Cumulative reward / steps :

This graph will explain to us the comportment of our robot in the scene, how good or bad was he accomplishing the task and did he get enough reward values or punishments. We have the following figure that shows that

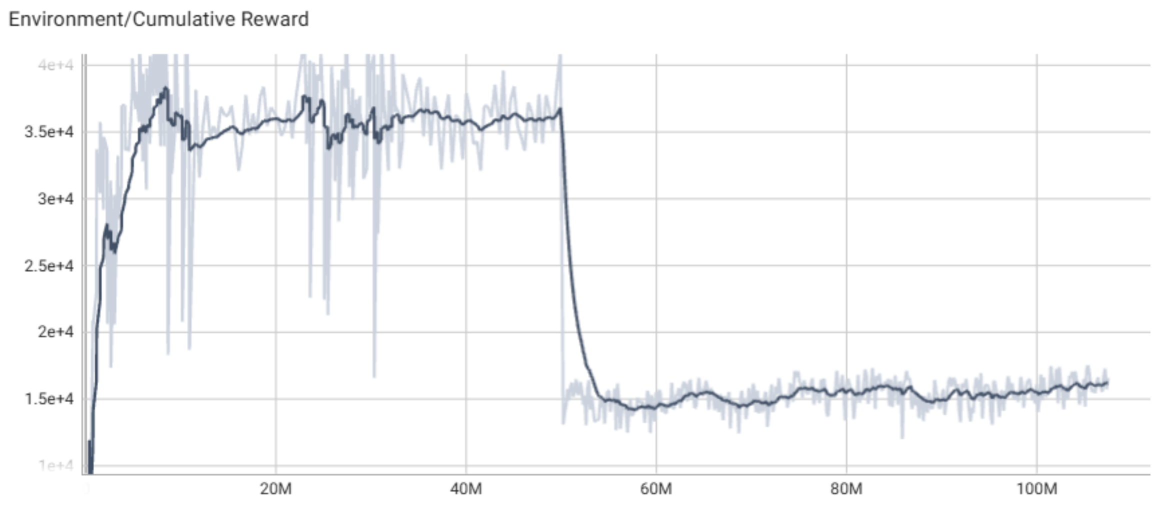


Figure 5.62: Steps/Cumulative reward value

Let's break this graph down. As we can see that the beginning fo the train wasn't that good which normal since the robot is still discovering its environment and is generating some random actions that are taking him away from the robot rather then getting closer to it but it remains a normal thing since we're still at the discovery part.
After approximately 10M steps we can see that the cumulative reward value has increased significantly entering a new phase of the train where the robot has discovered its environment and now is getting closer to the goal object. In this phase we can see the the reward is oscillating a lot around 34000 and 37000 this can be explained that each time we see that it learned the current reward tree we try to complicated a little bit this explains why we have sometimes some hard dropping in the cumulative reward value.

Over steps, we can see that it stabilizes around the 36000 value and from the simulation scene we found out that the robot has learned how to perform the task and he is getting close enough with a good orientation to the goal object, so we entered a new phase in which we try to teach the robot on how to perform the task quickly and not only effectively and for that we reduced the episode length from 25000 steps per episode to 10000 which less then the half of it. The main reason for that is to make it learn how to perform the task quickly since the episode will end early this time and that will be giving negative reward to the agent that he will try to avoid. So during this phase we can see a huge drop in the cumulative reward value but that's normal since he can't accumulate the same reward value with less available steps.

Even though we reduced the steps number but the robot didn't forget how to perform the task which can be seen in the plot, at the beginning the cumulative reward value has dropped and we can see that it's oscillating again, but after around 80M steps the agent did learn again correctly the task and stabilized the cumulative value.

5.3.7.2 Loss / Steps :

For the loss value plot, we should understand that it mean error between the generated action at an instant t and the supposed to be a good action for the same moment and not the opposite of the cumulative reward as some might understand or think ,so for continuous movements and robotic arms, the difference should not be big since we can't suddenly generate a big difference in angle for a certain joint, if it's the case that means there is something wrong. So we can say that the loss / steps graph is more of a stability indicator for our training.

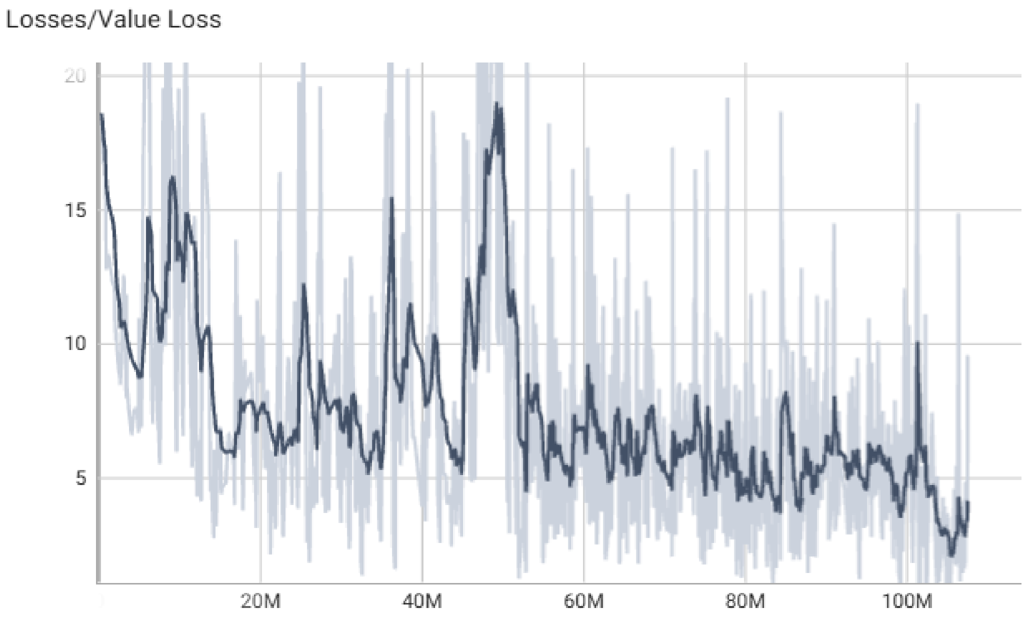


Figure 5.63: Steps/Loss value

As we can see from the figure of the Steps / Loss value. At the beginning of the training we have a high value jump in the loss since it's obvious that the robot is generating random actions in order to discover its environment. Then with the training going on, we can see clearly that the robot is minimizing the difference between its movements, so he is starting to learn how to perform a continuous action. At the same time we can see that the loss value increase so it's kind of oscillating, but this is normal since each time as previously explained we were complicating the reward tree for the agent, this will cause that the policy at the moment we upgrade the reward tree will start to receive some negative rewards a lot this will cause that the policy will re-start exploring again in order to find out which actions are more suitable now for the new reward tree.

After reaching the 50M we can see that the value loss increased a lot, which is due to decreasing the total number of steps for the episode. Same as previously motioned, this caused that the agent received huge negative reward at that moment, so the policy did re-discover the environment for a certain time then we can see that the loss value has dropped less then before the 50M because the policy has reached stability again until the end of the train around 100M steps where we can see that the loss value decreased a lot and we have less oscillations this time which means better stability of the policy.

At the end of this, to simulate the comportment of the robot we took the obtained model and tested it on 1 robot from the farm to see how it is performing the task. The following figure shows 3 phases of the test, first at the beginning at the test where the robot is initialized without movement, second in the middle of the test while the robot is trying to reach the desired goal object, and finally the last one when the robot has successfully reached the goal. We added some writings on the scene to visualize the distance and the alignment between the end effector and the goal object.

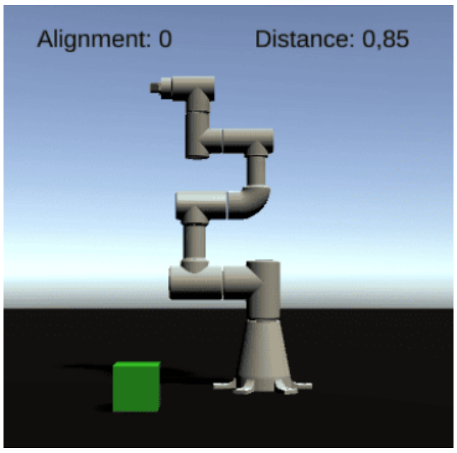


Figure 5.64: Test beginning

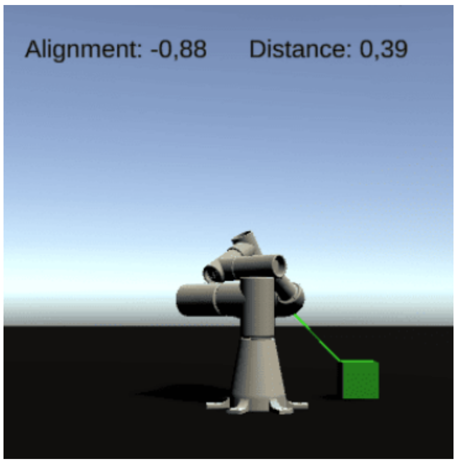


Figure 5.65: Mid test

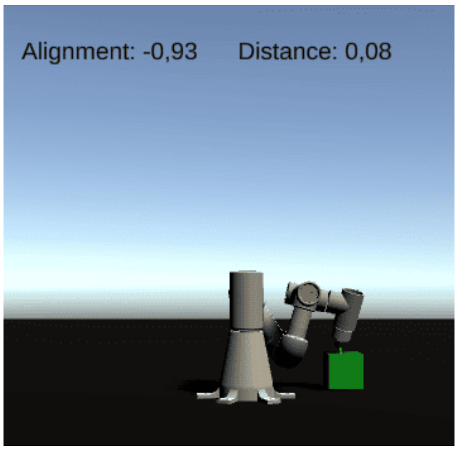


Figure 5.66: The end of the test

As we can see in the figure, the robot is initialized at its starting position and the goal object (the green cube) has spawned at a 0.85m distance from the end effector with a 0 alignment value which means 90 degrees angle between them. When we start the test the robot is trying to reach the goal object by reducing the distance by 0.39m and fixing the alignment of -0.88 which is 151 degrees. For us, the best case is to have less than 0.1m distance between the end effector and the cube. This can be visualized in the third part of the test in which we got a distance of 0.08m or 8cm while taking into account the cube size it self of 6cm, and since we calculate the distance between the center of the effector holder of the robot and the center of the green cube and not its out surface, the distance error between the end effector and the goal object reduces to 5cm in addition to that we take into account the dimensions of the gripper (the 7cm length of the fingers) which in this case reduces the error distance by 3cm so we get a global distance error of approximately 2 to 3cm, and with an alignment of -0.93 which is pretty good as a result and as we can see in the figure the robot has reached the goal object while facing it to make the grasping easy.

5.4 Aruco detection and pose estimation based approach

5.4.1 The approach

The main idea of this approach, is to estimate the position and orientation of the aruco tag fixed on the top of the goal object in order to use that information alongside the IK solvers of MoveIt package with ROS to calculate the required angles and trajectories for task accomplishing.
Before getting into more details, this figure shows the main structure of the aruco tag approach.

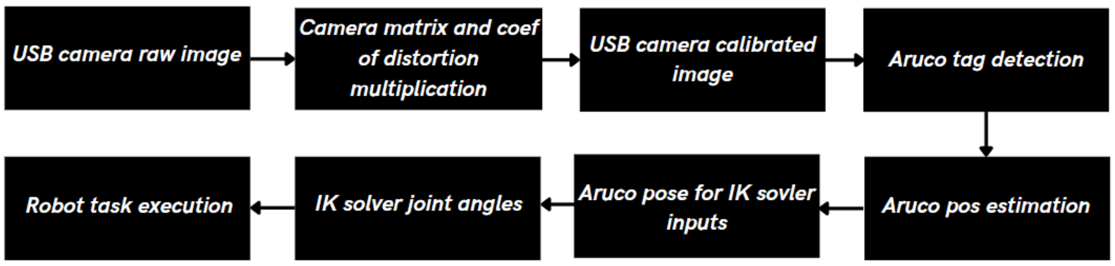

USB camera raw image
Camera matrix and coef of distortion multiplication
USB camera calibrated image
Aruco tag detection
Aruco pos estimation
Aruco pose for IK sovler inputs
IK solver joint angles
Robot task execution

Figure 5.67: Aruco pos based approach architecture

5.4.2 Camera Calibration

Since we are working with ROS, we are going to use the advantage of the available packages. The camera calibration package is a predefined package in the robot operating system that can be used to calibrate mono and stereo cameras alongside libraries of python specially the OpenCv one.

5.4.2.1 Chess Board :

Which is a printed black and white squares pattern used for camera calibration process. It is defined by the size of those squares and the number of edges vertically and horizontally. The following figure shows and example of that.

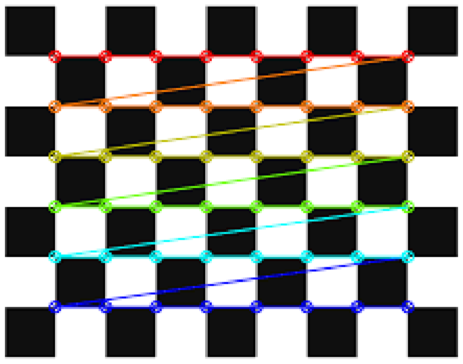


Figure 5.68: Chess board 8x6 25mm squares

5.4.2.2 Camera calibration package :

After printing the required chessboard for the work, we clone the repository of the camera calibration package in our work space and from a terminal we run the cameracalibrator.py node in which we are going to specify the number of edges, the square dimensions, and the topic of our camera USB.

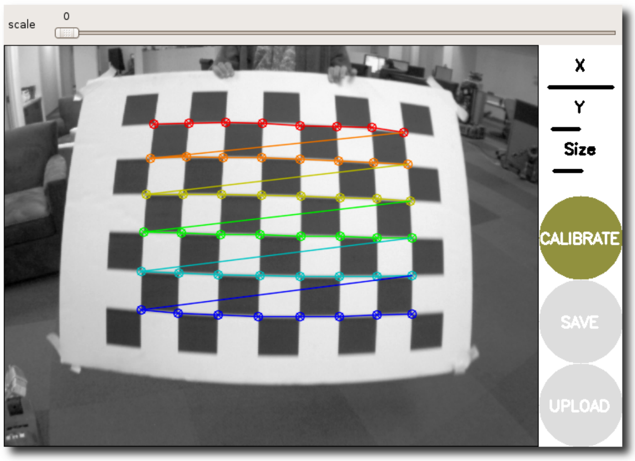
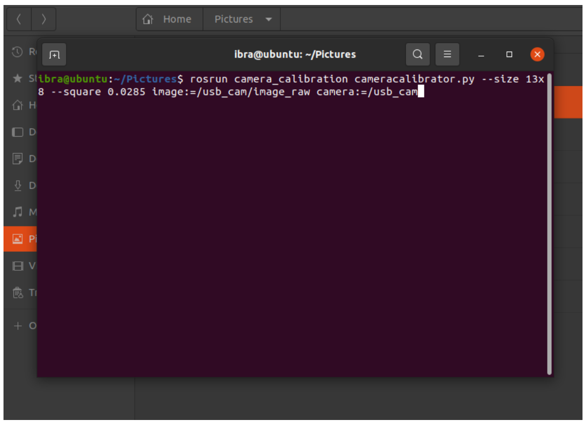


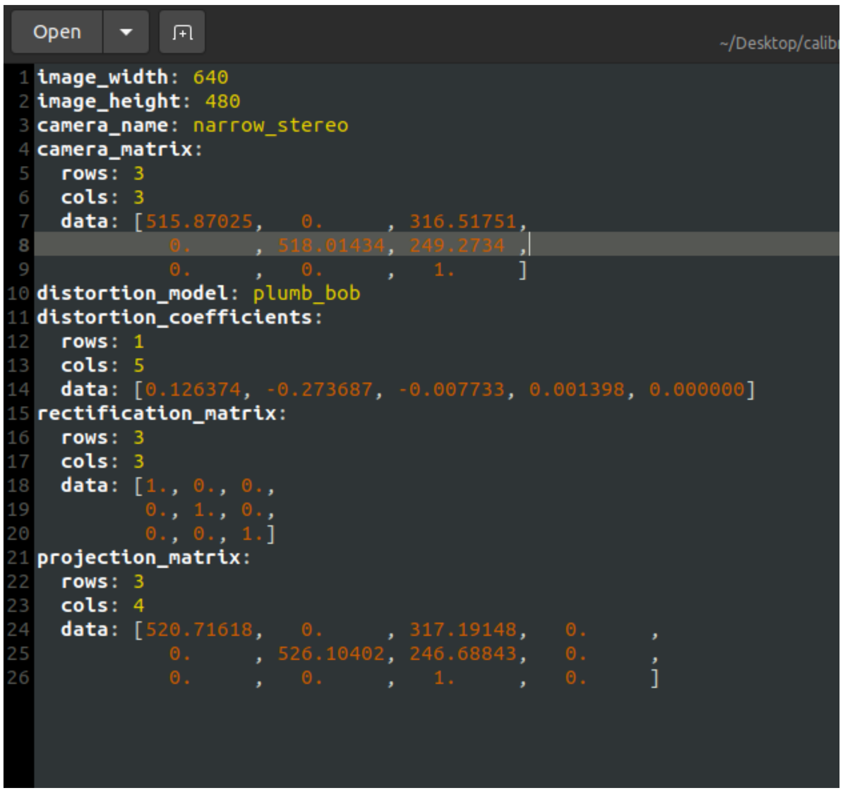
Figure 5.69: Calibration images collection



```
ibra@ubuntu:~/Pictures$ rosrun camera_calibration cameracalibrator.py --size 13x
8 --square 0.0285 image:=/usb_cam/image_raw camera:=/usb_cam
```

Figure 5.70: Running the node from terminal

After collecting enough images for the calibration while all the corners were detected and the X, Y and size bare from the right are fulled completely. The package will return to us at the end the camera matrix and coefficient of distortion in a config file as shown in this figure.



```
image_width: 640
image_height: 480
camera_name: narrow_stereo
camera_matrix:
  rows: 3
  cols: 3
  data: [515.87025,   0.     , 316.51751,
           0.     , 518.01434, 249.2734 ,
           0.     ,   0.     ,   1.     ]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [0.126374, -0.273687, -0.007733, 0.001398, 0.000000]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1., 0., 0.,
         0., 1., 0.,
         0., 0., 1.]
projection_matrix:
  rows: 3
  cols: 4
  data: [520.71618,   0.     , 317.19148,   0.     ,
           0.     , 526.10402, 246.68843,   0.     ,
           0.     ,   0.     ,   1.     ,   0.     ]
```

Figure 5.71: Camera parameters config file

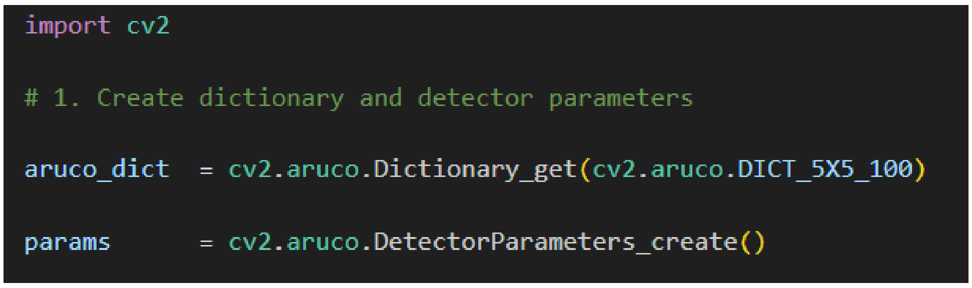
After that we are going to apply these factors on the raw image obtained from the camera each time we want to localize the arUco tag.

5.4.3 Aruco detection

5.4.3.1 Python library for aruco detection

For easy work on arUco markers detection, we are going to present the most suitable way for that which is using the OpenCv library with arUco module on python. It specially works with the 4.7.0 version of OpenCv and higher. Starting now to explain the most important and usable functions of the arUco module in OpenCv after for sure importing the library we have :

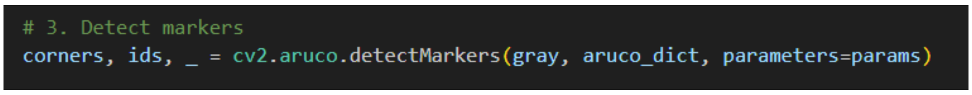
- **aruco.Dictionary_get() :** Basically this is the first function to call, in which we are going to call the dictionary we want to work with.
- **aruco.DetectorParameters_create() :** This function is used to set the parameters of the frame binarization (thresholding, corners marker, corners filtering) As shown in the following figure, this is how they are going to be used.



```
import cv2

# 1. Create dictionary and detector parameters

aruco_dict  = cv2.aruco.Dictionary_get(cv2.aruco.DICT_5X5_100)

params      = cv2.aruco.DetectorParameters_create()
```

Figure 5.72: aruco.DetectorParameters_create()

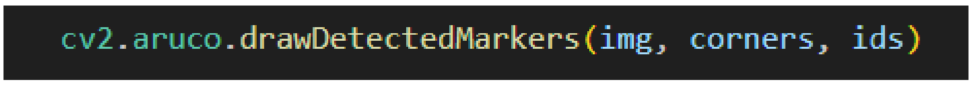
- **aruco.detectMarkers() :** This is the function that we use to detect the markers, it takes the grayscale frame from the capture image, the dictionary that we define to use and the set of parameters previously defined. All these parameters are taken as arguments for this function to detect the arUco markers, then save their outputs which are the corners, the IDs and rejected ones as shown in the figure.



```
# 3. Detect markers
corners, ids, _ = cv2.aruco.detectMarkers(gray, aruco_dict, parameters=params)
```

Figure 5.73: aruco.detectMarkers()

- **aruco.drawDetectedMarkers() :** The last important function which has to draw the ids and the frames of the detected markers on the image as an output. For the arguments, it takes the normal frame img, the IDs and corners obtained from the previous function as the next figure shows.



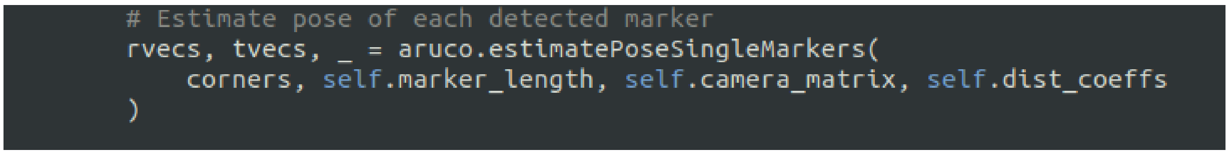
```
cv2.aruco.drawDetectedMarkers(img, corners, ids)
```

Figure 5.74: aruco.drawDetectedMarkers()

So basically, this is the simplest method u can use in order to detect arUco markers using a python script since it's the most suitable one while working on image processing.

5.4.4 Arcuo pose estimation

After detecting the arUco tag placed on the top of the goal object, we should now proceed to the main part of the approach which is the pose estimation from the calibrated image. For this purpose, we used the same aruco for OpenCv library used for the aruco detection but this time using different function arcuo.estimateposesinglemarker() as shown in the next figure.



```
# Estimate pose of each detected marker
rvecs, tvecs, _ = aruco.estimatePoseSingleMarkers(
    corners, self.marker_length, self.camera_matrix, self.dist_coeffs
)
```

Figure 5.75: Pose estimation function

The provided function takes as inputs a vector of the 4 aruco corners detected, the marker length that we specify at the beginning since we are the ones to chose the aruco tag ID and size for the application, it does take also the camera matrix and distortion coefficient that we get from the config file after the calibration process.

Based on that, the function tries to estimate the real 3D pose of the arUco tag (position and orientation) but in the camera frame.

And at the end, it outputs the rvecs which has the rotation information on each axis, and the tvecs that has the translation or positions on each axis.

5.4.5 Task execution

Based on the explained work on how to estimate the aruco pose, we can use the approach either on ROS or Robodk based on the case of work.

5.4.5.1 Using ROS

In order to implement the aruco pose navigation on , we will work using this approach :

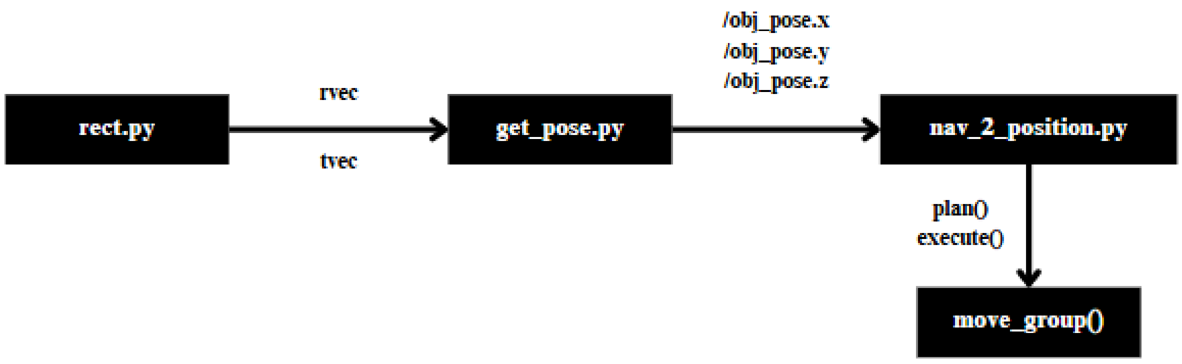
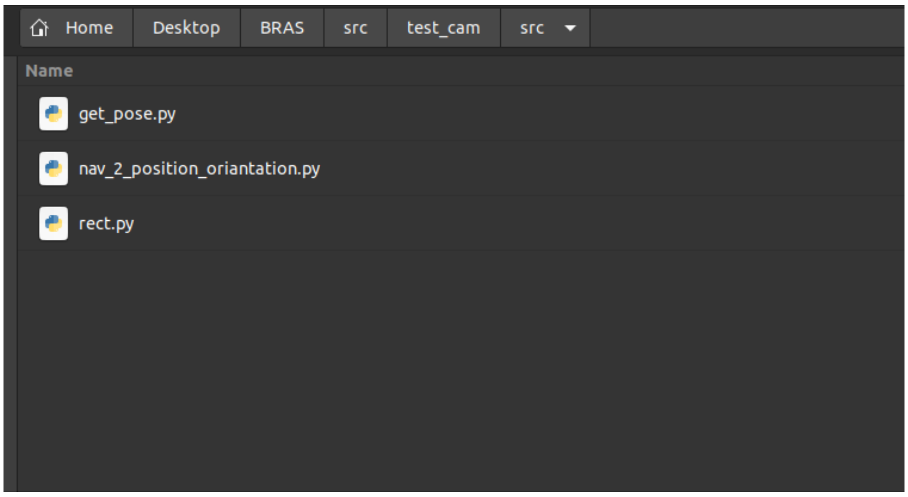


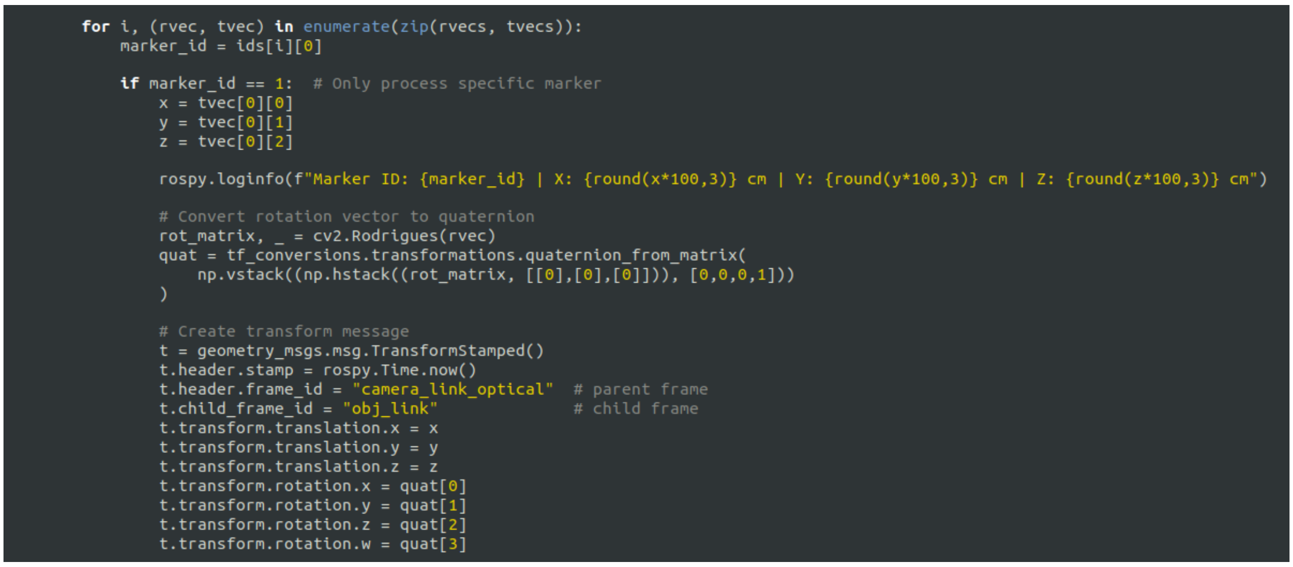
Figure 5.76: ROS based aruco navigation approach



get_pose.py
nav_2_position_oriantation.py
rect.py

Figure 5.77: Workspace ROS for aruco pose navigation

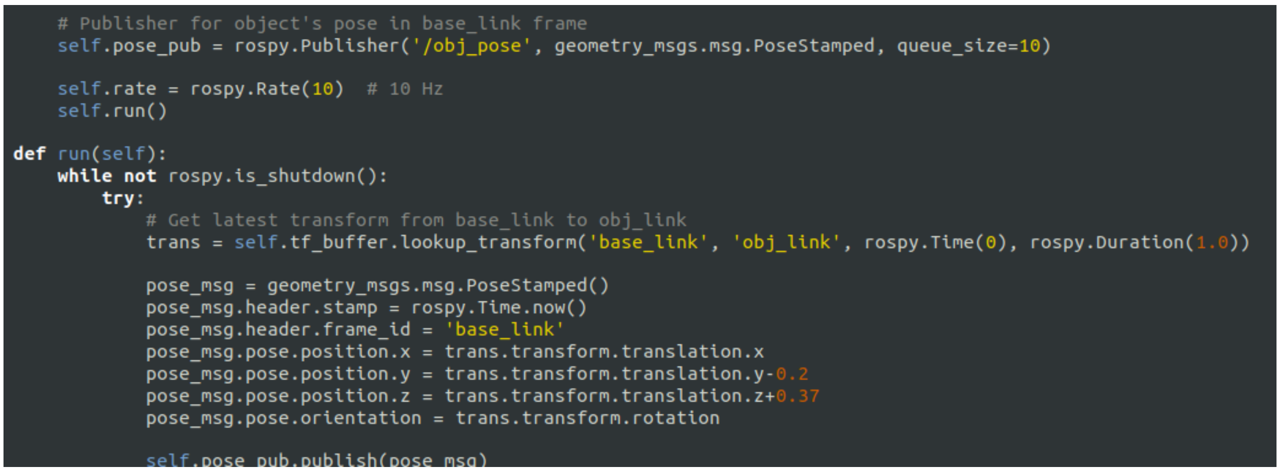
- **rect.py** : a python node responsible on detecting an aruco, performs pose estimation and create a *"obj_link"* frame that has as attributes the position and orientation estimated earlier.



```
for i, (rvec, tvec) in enumerate(zip(rvecs, tvecs)):
    marker_id = ids[i][0]

    if marker_id == 1:  # Only process specific marker
        x = tvec[0][0]
        y = tvec[0][1]
        z = tvec[0][2]

        rospy.loginfo(f"Marker ID: {marker_id} | X: {round(x*100,3)} cm | Y: {round(y*100,3)} cm | Z: {round(z*100,3)} cm")

        # Convert rotation vector to quaternion
        rot_matrix, _ = cv2.Rodrigues(rvec)
        quat = tf_conversions.transformations.quaternion_from_matrix(
            np.vstack((np.hstack((rot_matrix, [[0],[0],[0]])), [0,0,0,1]))
        )

        # Create transform message
        t = geometry_msgs.msg.TransformStamped()
        t.header.stamp = rospy.Time.now()
        t.header.frame_id = "camera_link_optical"  # parent frame
        t.child_frame_id = "obj_link"              # child frame
        t.transform.translation.x = x
        t.transform.translation.y = y
        t.transform.translation.z = z
        t.transform.rotation.x = quat[0]
        t.transform.rotation.y = quat[1]
        t.transform.rotation.z = quat[2]
        t.transform.rotation.w = quat[3]
```

Figure 5.78: rect.py

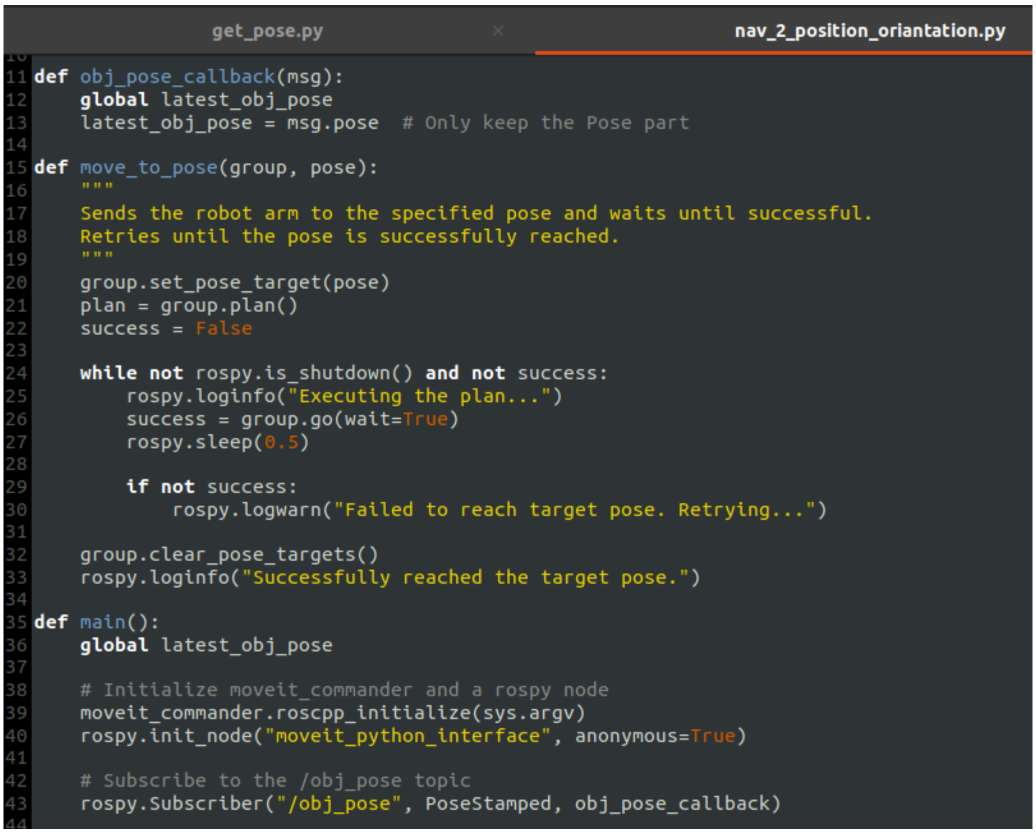
- **get_pose.py** : a python node that gets the position and orientation from "obj_link" frame and publishes it pose in respect to the *"base_link"* frame (the reference frame of moveIt in /obj_pose topic.



```
    # Publisher for object's pose in base_link frame
    self.pose_pub = rospy.Publisher('/obj_pose', geometry_msgs.msg.PoseStamped, queue_size=10)

    self.rate = rospy.Rate(10)  # 10 Hz
    self.run()

def run(self):
    while not rospy.is_shutdown():
        try:
            # Get latest transform from base_link to obj_link
            trans = self.tf_buffer.lookup_transform('base_link', 'obj_link', rospy.Time(0), rospy.Duration(1.0))

            pose_msg = geometry_msgs.msg.PoseStamped()
            pose_msg.header.stamp = rospy.Time.now()
            pose_msg.header.frame_id = 'base_link'
            pose_msg.pose.position.x = trans.transform.translation.x
            pose_msg.pose.position.y = trans.transform.translation.y-0.2
            pose_msg.pose.position.z = trans.transform.translation.z+0.37
            pose_msg.pose.orientation = trans.transform.rotation

            self.pose_pub.publish(pose_msg)
```

Figure 5.79: get_pose.py node

- **nav_2_position.py** : a python node that subscribe to /obj_pose topic gets the pose of the object(position and orientation) and navigates to that pose using move_group().



```
def obj_pose_callback(msg):
    global latest_obj_pose
    latest_obj_pose = msg.pose  # Only keep the Pose part

def move_to_pose(group, pose):
    """
    Sends the robot arm to the specified pose and waits until successful.
    Retries until the pose is successfully reached.
    """
    group.set_pose_target(pose)
    plan = group.plan()
    success = False

    while not rospy.is_shutdown() and not success:
        rospy.loginfo("Executing the plan...")
        success = group.go(wait=True)
        rospy.sleep(0.5)

        if not success:
            rospy.logwarn("Failed to reach target pose. Retrying...")

    group.clear_pose_targets()
    rospy.loginfo("Successfully reached the target pose.")

def main():
    global latest_obj_pose

    # Initialize moveit_commander and a rospy node
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node("moveit_python_interface", anonymous=True)

    # Subscribe to the /obj_pose topic
    rospy.Subscriber("/obj_pose", PoseStamped, obj_pose_callback)
```

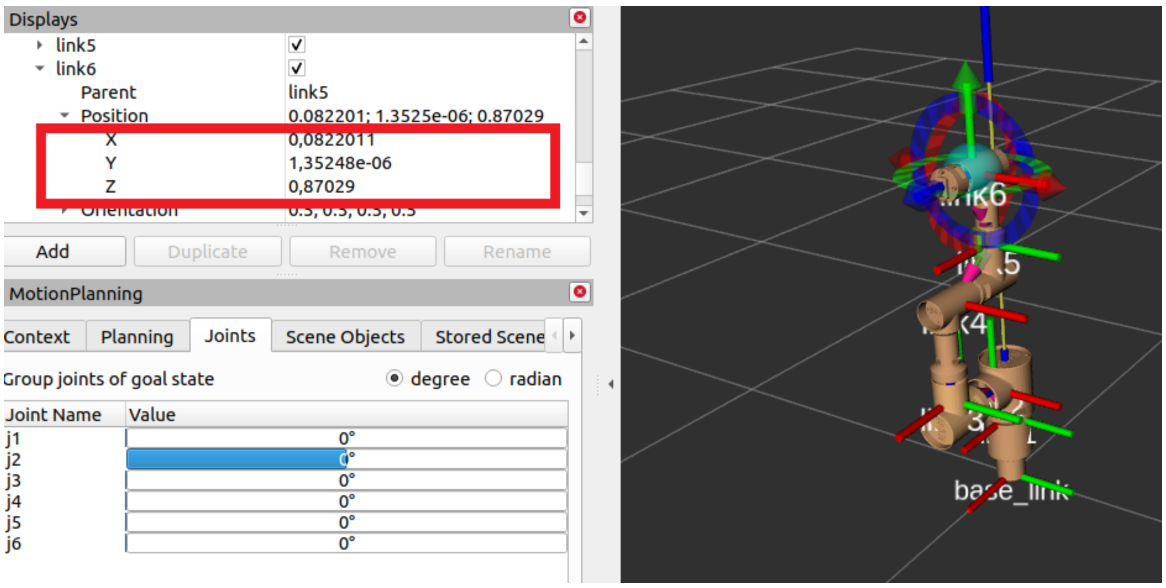
Figure 5.80: nav_2_position.py node

5.4.5.1.1 Test

5.4.5.1.1.1 First step : moveIt launch

We launch our robot on ROS moveIt and as we can see we have our eef position of the robot on the pose :

- X position = 0,08 .
- Y position = $1,35 \times 10^{-6}$.
- Z position = 0,87029.



Displays

link5	✓
link6	✓
Parent	link5
Position	0.082201; 1.3525e-06; 0.87029
X	0,0822011
Y	1,35248e-06
Z	0,87029
Orientation	0.5; 0.5; 0.5; 0.5

MotionPlanning

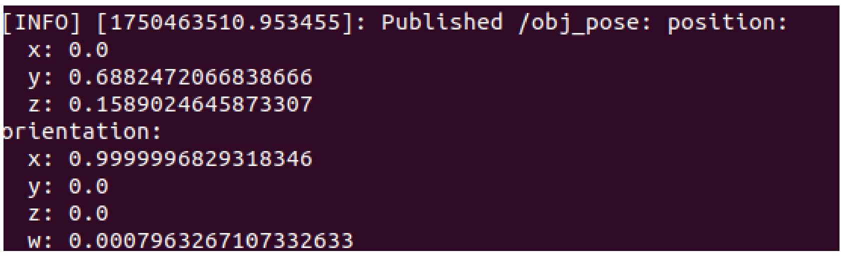
Group joints of goal state [✓ degree] radian

Joint Name	Value
j1	0°
j2	0°
j3	0°
j4	0°
j5	0°
j6	0°

Figure 5.81: 1st Moveit launch

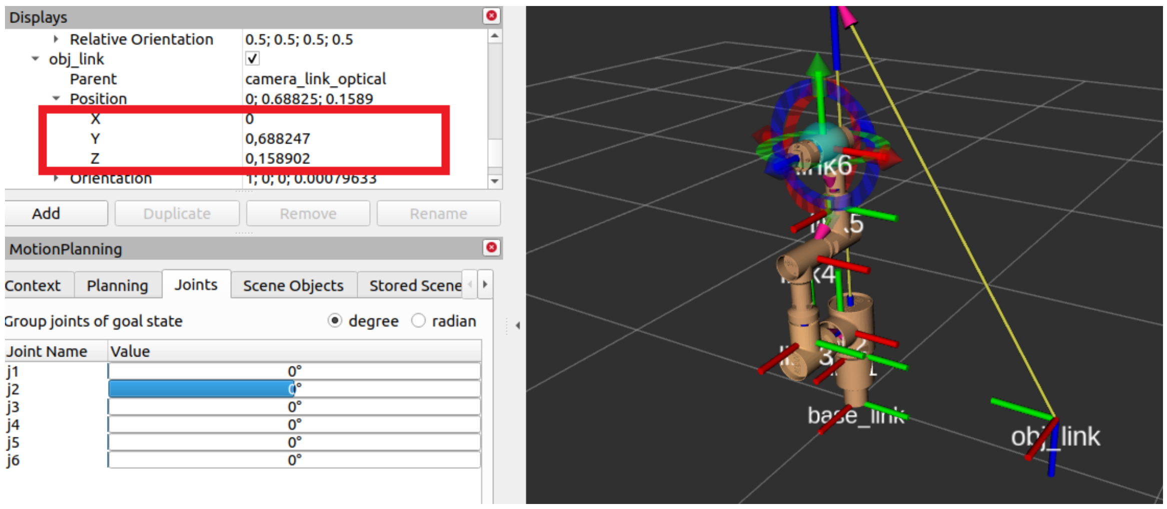
5.4.5.1.1.2 Second step : Aruco detection and pose estimation

In this step we are going to detect the aruco from its node and attribute it position and orientation to a frame called "*/obj_link*".



```
[INFO] [1750463510.953455]: Published /obj_pose: position:
  x: 0.0
  y: 0.6882472066838666
  z: 0.1589024645873307
orientation:
  x: 0.9999996829318346
  y: 0.0
  z: 0.0
  w: 0.0007963267107332633
```

Figure 5.82: Publishing the position of the aruco in respect to the camera frame



```
Displays
  ▸ Relative Orientation   0.5; 0.5; 0.5; 0.5
  ▾ obj_link               ✓
      Parent               camera_link_optical
    ▾ Position             0; 0.68825; 0.1589
        X                  0
        Y                  0,688247
        Z                  0,158902
    ▸ Orientation          1; 0; 0; 0.00079633
```

```
MotionPlanning
Context | Planning | Joints | Scene Objects | Stored Scene
Group joints of goal state      (●) degree  ( ) radian
Joint Name   Value
j1           0°
j2           0°
j3           0°
j4           0°
j5           0°
j6           0°
```

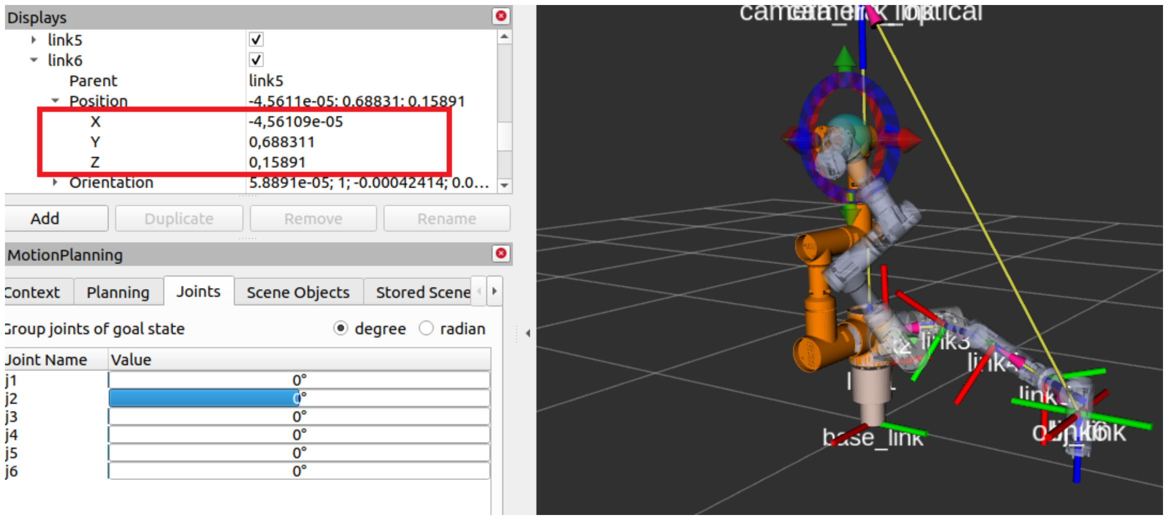
Figure 5.83: The position of the aruco in respect to the base link frame

As we can see the aruco pose its at the position :

- X position = 0.
- Y position = 0,688247.
- Z position = 0,158902.

5.4.5.1.1.3 Third step : nav to aruco pose

Using the nav_2_position.py node explained earlier to send our robot from its current position to the aruco pose.



```
Displays
  ▸ link5                  ✓
  ▾ link6                  ✓
      Parent               link5
    ▾ Position             -4.5611e-05; 0.68831; 0.15891
        X                  -4,56109e-05
        Y                  0,688311
        Z                  0,15891
    ▸ Orientation          5.8891e-05; 1; -0.00042414; 0.0...
```

```
MotionPlanning
Context | Planning | Joints | Scene Objects | Stored Scene
Group joints of goal state      (●) degree  ( ) radian
Joint Name   Value
j1           0°
j2           0°
j3           0°
j4           0°
j5           0°
j6           0°
```

Figure 5.84: Navigating to the aruco position with our robotic arm

We can see that the EEF of our robot reached the position :

- X position = $-4,56109 \times 10^{-5}$.
- Y position = 0,688311.
- Z position = 0,15891.

	X position	**Y position**	**Z position**
Aruco pose in meter	0	0,688247	0,158902
Final EEF pose in meter	$-4,56109 \times 10^{-5}$	0.688311	0,15891
Absolute error in cm	$4,56109 \times 10^{-3}$	0.051	0,0008

Table 5.17: Comparison between the aruco position and the EEF position after aruco-based navigation

The end-effector (EEF) successfully reached the ArUco-detected position with high accuracy. The absolute positional errors are minimal—approximately 0.05 cm in Y, 0.0008 cm in Z, and 0.0046 cm in X—indicating precise alignment between the desired and final poses after ArUco-based navigation.

5.5 Conclusion

As a conclusion for this work, we can say that we got some good results in each approach compared to the related works done in this field specially since the main approach we went through are kind of different from the previous works.

For the supervised learning, our work aims to predict the future movement of the robotic arm, in contrast to previous works which only estimate the robot's current position. However, despite this difference, we were able to achieve results similar to theirs.

As for the reinforcement learning approach, our work aimed to get a robust IK solver with the minimum possible of input presented by the goal object position giving a space vector of 3 elements only compared to the related work which had up to 19 elements, and for this difference our approach took much more time during the training compared to theirs. But even though we were able to get results which are slightly near to theirs.

And for the aruUco pose estimation approach, the simulation did gave accurate results, since the camera detected the position of the arUco tag on the object goal and reached the desired position as shown in the test and result part. Unfortunately the test were done only on simulation and not in reality since the required equipments and pieces manufacturing weren't available at the time.

Conclusion and Perspectives

In this thesis, we focused on designing and developing an autonomous 6 DOF robotic arm that can be an intelligent decision-making platform through the integration of machine learning and image processing techniques. Our goal was not only to achieve full autonomy for industrial tasks but also a commercial product that aligns with modern collaborative robotics standards.

Following a challenging engineering path, we successfully addressed the complete development cycle of the robotic arm — from mechanical design, kinematic and dynamic analysis, 3D modeling and printing, electrical integration, and final assembly. We ensured that each component, both mechanical and electronic, was carefully selected and tested for an optimal performance.

In terms of control and software integration, we successfully implemented two control approaches using ROS with MoveIt and Robodk. These platforms provided a robust framework for simulating, controlling, and validating the robotic arm's planning.

The most important contribution of our work is exploring, implementing and comparing different machine learning approaches for robotic arms autonomy. We explored supervised learning for pose estimation and behavior cloning, reinforcement learning to let the robot learn optimal actions through interaction with its environment, and an ArUco marker-based pose estimation method to enhance fast and real time navigation. These experiments confirmed that our robotic arm could perform intelligent tasks such as path planning and decision-making with full autonomy.

Additionally, by creating a Business Model Canvas (BMC), we explored the entrepreneurial point of vue of our project, preparing it for potential commercialization as an intelligent collaborative robotic solution adaptable to multiple industrial tasks.

In conclusion, this project highlights the potential of combining machine learning with robotic systems to create autonomous, flexible, and intelligent robotic arms suitable for a wide range of real-world applications.

For future work, we would like to start by improving our mechanical structure in order to ensure the scalability of our product and better address industrial-scale applications.

As for the intelligence and autonomy aspects, we aim to make our model more general and achieve better results by exploring larger and more diverse datasets. This includes incorporating the DH parameters of different robots alongside their respective datasets as inputs to our model, in order to ensure the generalization of our work and its applicability to various types of robots. We also plan to combine the supervised learning approach with reinforcement learning methods to further improve the accuracy of our results.

As a final improvement, we would try to collect our own robot dataset for better accuracy of the supervised learning approach, then use the obtained weights as predefined ones for the reinforcement learning approach for earlier convergence and better results.

Bibliography

1. SAHA, S. K. *Introduction to Robotics*. 2nd ed. India: McGraw-Hill Education, 2014.
2. CHEBCHEB, Abderrahmane. *Projet de fin d'études - Robotique autonome* [https://repository.enp.edu.dz/jspui/bitstream/123456789/1037/1/pfe.2020.auto.CHEBCHEB.Abderrahmane.pdf?123=123]. 2020. Accessed: 2025-06-13.
3. *KUKA ForceTorqueControl* [online]. [visited on 2025-06-13]. Available from: https://www.kuka.com/fr-fr/produits-et-prestations/syst%C3%A8mes-de-robots/logiciels/logiciels-d'application/kuka-forcetorquecontrol.
4. *UR5e Collaborative Robot* [online]. [visited on 2025-06-13]. Available from: https://www.universal-robots.com/products/ur5e/.
5. CORRELL, Nikolaus. *Introduction to Autonomous Robots*. University of Colorado at Boulder, 2016. Available online.
6. MAVRIDIS, Nikolaos. A review of auditory perception in robotics—Robotic hearing. *In-Tech*. 2013. Available from DOI: 10.5772/55170.
7. *Stepper motor* [https://en.wikipedia.org/wiki/Stepper_motor]. [N.d.]. Accessed: 2025-06-13.
8. SAMEERA; HASAN, Mohd Asif. A Study of Stepper Motors for Performance Improvement of the CNC Machine. *International Journal of Engineering Research*. 2019.
9. WIKIMEDIA COMMONS CONTRIBUTORS. *Servo Motor* [online]. 2020. [visited on 2025-06-13]. Available from: https://etechrobot.com/product/ds51150-servo-motor-high-torque-150kg/. Image of the internal structure of a servo motor, licensed under Creative Commons.
10. *Basics of Limit Switches* [https://instrumentationtools.com/basics-limit-switches/]. [N.d.]. Accessed: 2025-06-13.
11. POWERLAB ALGERIA. *MPU6050 Module Image* [online]. 2023. [visited on 2025-06-14]. Available from: https://powerlab.dz/wp-content/uploads/2023/02/download-2023-02-28T155144.345.jpg. Image of the MPU6050 gyroscope and accelerometer module.
12. SEYBOLD, Jonathan; BÜLAU, André; FRITZ, Karl-Peter; FRANK, Alexander. Miniaturized Optical Encoder with Micro Structured Encoder Disc. *Sensors*. 2017.
13. GOOGLE IMAGES. *Havit RGB Camera Image* [online]. 2025. [visited on 2025-06-14]. Available from: https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcREZ4cID6DjnI4ce9WY9 s. Image of a Havit RGB camera accessed via Google Images.
14. *ROS Logo on Wikimedia* [https://fr.wikipedia.org/wiki/Fichier:Ros_logo.svg]. [N.d.]. Accessed: 2025-06-13.
15. *ROS Topics - MathWorks Documentation* [https://ch.mathworks.com/help/ros/gs/ros-topics.html]. [N.d.]. Accessed: 2025-06-13.

16. *Robotic simulation scenarios with Gazebo and ROS* [https://www.generationrobots.com/blog/en/robotic-simulation-scenarios-with-gazebo-and-ros/]. [N.d.]. Accessed: 2025-06-13.
17. *ROS Simulations - RoboticLab HomeLab* [https://www.roboticlab.eu/homelab/en/ros/simulations]. [N.d.]. Accessed: 2025-06-13.
18. *RViz - ROS Visualization GitHub Repository* [https://github.com/ros-visualization/rviz]. [N.d.]. Accessed: 2025-06-13.
19. OPEN SOURCE ROBOTICS FOUNDATION. *rosserial - ROS Wiki* [online]. 2025. [visited on 2025-06-14]. Available from: https://wiki.ros.org/rosserial. Accessed via the official ROS Wiki.
20. KAELBLING, Leslie Pack; LITTMAN, Michael L.; MOORE, Andrew W. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*. 1996, vol. 4, pp. 237–285.
21. SPICEWORKS. *What is Markov Decision Process?* [online]. 2024. [visited on 2025-06-14]. Available from: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-markov-decision-process/. Accessed on June 14, 2025.
22. WIKIPEDIA CONTRIBUTORS. *Markov Decision Process* [online]. 2024. [visited on 2025-06-14]. Available from: https://en.wikipedia.org/wiki/Markov_decision_process. Accessed on June 14, 2025.
23. JUNG, Whiyoung; CHO, Myungsik; PARK, Jongeui; SUNG, Youngchul. *Quantile Constrained Reinforcement Learning: A Reinforcement Learning Framework Constraining Outage Probability* [https://ieeexplore.ieee.org/document/9509222]. 2021. Available on IEEE Xplore or arXiv.
24. BEWLEY, Tom; LÉCUÉ, Freddy. Interpretable Preference-based Reinforcement Learning with Tree-Structured Reward Functions. In: *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2022)*. IFAAMAS, 2022, pp. 118–126. Available from DOI: 10.5555/3525948.3545736. Also available as arXiv preprint arXiv:2112.11230 (Dec 20, 2021).
25. SCHULMAN, John; WOLSKI, Filip; DHARIWAL, Prafulla; RADFORD, Alec; KLIMOV, Oleg. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*. 2017. Available also from: https://arxiv.org/abs/1707.06347.
26. BELCIC, I.; STRYKER, C. *What Is Supervised Learning?* IBM, 2024-12. Available also from: https://www.ibm.com/topics/supervised-learning. Accessed: 2025-06-14.
27. CHOUBEY, Vijay. *Demystifying Convolutional Neural Networks* [online]. 2020-7 26. [visited on 2025-06-14]. Available from: https://vijay-choubey.medium.com/understanding-convulutional-neural-networks-9b0cbd9b3055. Published on Medium; 8min read.
28. DASARI, Sudeep; EBERT, Frederik; TIAN, Stephen; NAIR, Suraj; BUCHER, Bernadette; SCHMECKPEPER, Karl; SINGH, Siddharth; LEVINE, Sergey; FINN, Chelsea. RoboNet: Large-Scale Multi-Robot Learning. In: *CoRL 2019: Volume 100 Proceedings of Machine Learning Research*. 2019. Available from arXiv: 1910.11215 `[cs.RO]`.
29. MANDLEKAR, Ajay; BOOHER, Jonathan; SPERO, Max; TUNG, Albert; GUPTA, Anchit; ZHU, Yuke; GARG, Animesh; SAVARESE, Silvio; FEI-FEI, Li. Scaling robot supervision to hundreds of hours with roboturk: Robotic manipulation dataset through human reasoning and dexterity. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 1048–1055.
30. FINN, Chelsea; LEVINE, Sergey. Deep Visual Foresight for Planning Robot Motion. *arXiv preprint arXiv:1610.00696*. 2017. Available from arXiv: 1610.00696 `[cs.LG]`.

31. RODRÍGUEZ-MIRANDA, Sergio; YAÑEZ-MENDIOLA, Javier; CALZADA-LEDESMA, Valentin; VILLANUEVA-JIMENEZ, Luis Fernando; ANDA-SUAREZ, Juan De. Pose Determination System for a Serial Robot Manipulator Based on Artificial Neural Networks. *Machines*. 2023. Available also from: https://www.mdpi.com/2075-1702/11/6/592. Published: 26 May 2023.

32. ZHOU, Ling; WANG, Ruilin; ZHANG, Liyan. Accurate Robot Arm Attitude Estimation Based on Multi-View Images and Super-Resolution Keypoint Detection Networks. *Sensors*. 2022. Available also from: https://www.mdpi.com/1424-8220/22/24/9714. College of Mechanical & Electrical Engineering, Nanjing University of Aeronautics and Astronautics, China.

33. ZHAO, Chengyi; WEI, Yimin; XIAO, Junfeng; SUN, Yong; ZHANG, Dongxing; GUO, Qiuquan; YANG, Jun. Inverse kinematics solution and control method of 6-degree-of-freedom manipulator based on deep reinforcement learning. *Scientific Reports*. 2024, vol. 14, p. 12467. Available from DOI: 10.1038/s41598-024-62948-6.

34. HØGSBRO, Gustav; BODILSEN, Rasmus; KJAERGAARD, Morten. PPO for 6-DoF Grasping in Space Robotics. In: *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*. ESA, 2023. Available also from: https://vbn.aau.dk/en/publications/ppo-for-6-dof-grasping-in-space-robotics. Joint-control PPO policy for Franka Emika Panda in simulated off-Earth rock-grasping environments.

35. CAPITOL TECHNOLOGY UNIVERSITY. *SolidWorks for Mechatronics Design and Engineering Program* [online]. 2022. [visited on 2025-06-14]. Available from: https://www.captechu.edu/blog/solidworks-mechatronics-design-and-engineering-program. Accessed June 14, 2025.

36. 1000LOGOS.NET. *SolidWorks Logo – Design, Meaning & History* [online]. 2025-02-18. [visited on 2025-06-14]. Available from: https://1000logos.net/solidworks-logo/. Accessed June 14, 2025.

37. UNIVERSAL ROBOTS. *UR3e – Collaborative Robotic Arm* [online]. [visited on 2025-06-14]. Available from: https://www.universal-robots.com/products/ur3e/. Accessed June 14, 2025.

38. FORMANT. *What is URDF (Unified Robotics Description Format)?* [online]. [visited on 2025-06-14]. Available from: https://formant.io/resources/glossary/urdf/. Accessed June 14, 2025.

39. *6-DOF Robotic Arm Image* [https://th.bing.com/th/id/R.7c54fb3634ffca06ece1787c149881e2]. [N.d.]. Accessed: 2025-06-20.

40. 3D, Spool. *TB6600 Stepper Motor Driver Controller*. 2025. Available also from: https://spool3d.ca/tb6600-stepper-motor-driver-controller/. Accessed: 2025-06-21.

41. ROBOTIQUE, Atelier de la. *Contrôleur de moteur pas à pas TB6600 haute précision et performance*. 2025. Available also from: https://www.atelierdelarobotique.fr/produit/controleur-de-moteur-pas-a-pas-tb6600-haute-precision-et-performance. Accessed: 2025-06-21.

42. COMMUNITY, ST. *UART / USART not working with GPS module*. 2024. Available also from: https://community.st.com/t5/stm32-mcus-products/uart-usart-not-working-with-gps-module/td-p/669336. Accessed: 2025-06-20.