Master thesis on Inteligent Interactive Systems

Universitat Pompeu Fabra

Adaptation of Opeanai Gym for Moveo Platform

Mario Acera Mateos

Supervisor: Mario Ceresa

Septembre 2020

Master thesis on Intelligent Interactive Systems

Universitat Pompeu Fabra

Adaptation of OpeanAI Gym for Moveo Platform

Mario Acera Mateos

Supervisor: Mario Ceresa

Septembre 2020

Table of Contents

Abstract

In this project we aim to adap Openai Reinforcement Learning platform to the Moveo robot, a low-cost, 3D printed robotic manipulator. This research will widen the field of application of Moveo, concretely for robotics surgeries and also will make the research in robotics available and affordable for a large portion of the research community. Using only open-source technologies we experiment with the different required modules and pose the global architecture to train robots focused on manipulation with Reinforcement Learning techniques. This project is constructed by the combination of three main components: A real robot, a simulation platform and the Reinforcement Learning module. These components are interconnected using the robotic middleware ROS which will provide the corresponding communication interface. As a realistic simulation platform the Gazebo software is used. The Reinforcement Learning module is the OpenAI Gym which interfaces with ROS based robots through the package Openai-ROS. In this document we provide a detailed analysis of the functioning of the package Openai-ROS besides a robotic model in Gazebo with the corresponding functionalities for it to be used along with Openai-ROS and perform the training.

Keywords: Moveo; OpenAI; ROS; Openai-ROS;

1. Chapter 1: Introduction

With the final goal of applying these advances in the field of robotics surgery, in this work we experiment with training a low-cost robotic arm through Deep Reinforcement Learning techniques. This transversal project covers from setting a real robotic arm in motion to developing the corresponding interfaces for setting up a manipulation problem as a Deep Reinforcement Learning problem. Therefore, a guideline to integrate a robot from scratch with the corresponding set up to interface state of the art robotic softwares and state of the art Reinforcement Learning benchmark platforms is provided.

Concretely this project is carried out using a robotic arm called Moveo. The final objective is to reach a level of control through Deep Reinforcement Learning that allows the use of Moveo for epilepsy surgery procedures research, where electrode placing has to be performed with high level of precision.

1.1. Objectives

- Finish the mechanical assembly and perform the electronic assembly of Moveo Robotic Arm. Once the robotic arm is fully operable the objective is to control it from a robotic software platform, ROS.
- Create a simulated Moveo model that allows the application of Deep Reinforcement Learning Techniques.
- Reproduce Deep Reinforcement Learning training in a manipulator robot from the bibliography.

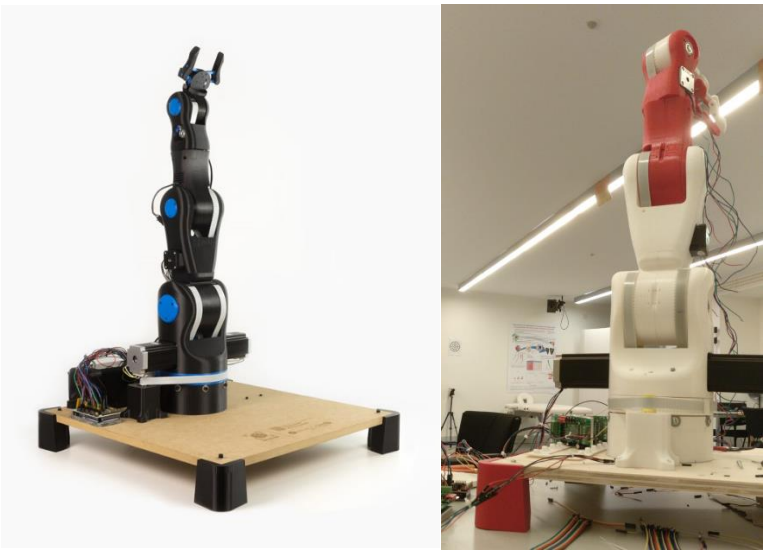
2. Chapter 2: State of the Art

The world of robotics and mechatronics has been growing during the last decades with a high popularity within students all over the world. Nevertheless, the possibilities for universities to offer hand-on works on certains areas of mechatronics has been a struggle, most due to the high budget needed to acquire the materials, morove the chances for a student to carry out a thesis on these areas were even more remote. As interesting and useful as most mechatronics applications are, they are also highly expensive and used to be restricted to industrial use. Focusing on robotics arms, if we go only a few years back, the budget needed to obtain a commercial robotic arm for research was over 15.000 € [1], just to set a lower bound. However, robotics arms are not overpriced, but they need high quality mechanical parts and controllers to achieve sufficiently good performance and accuracy.

Nowadays the overall picture for robotics is better, with the development of simulation softwares is easier to research kinematics and motion control and there are plenty of robotics models to work with [2]. Also, as fabrication technologies improve, the price of robotics components has lowered. Furthermore, innovative and fairly new fabrication technologies have become more popular and common. Concretely 3D printing technologies have experienced notable advances and standardizations making the use of this technology more wide-spread and becoming a common method for rapid prototyping [3].

2.1. Moveo as robotic platform

In the broad picture robotics, specifically, robotics arms, Moveo [4] Fig. 2.1, were born as an Open Source project with fully academic and educational purposes and bringing to the robotic field innovative ideas that have encouraged the research group to choose this precise robot to work with over many others. Moveo has been built from scratch to have a fully 3D printed structure using additive manufacturing techniques and controlling its electronics through Arduino software [4].



a) b)

Figure 2.1 : Moveo robotic arm. a) From [4] b) Built in UPF laboratory

As Moveo, many other robots have been designed with similar technology and are available for free from different sources and makers, many of which are also open source and modifiable to meet the needs of the user [5]. Nevertheless, there are not that many robots of this kind that can meet the needs of a fully operative robotic arm that can be employed for educational and research purposes [5], not to say on medical researchs on surgeries.

One of the main characteristics that make robotics arms really useful and interesting nowadays is the capability of being simulated and controlled from a software, so a robot like Zortrax [6], despite it advanced design, has to be rejected since some of its joints are not motorized, making it unusable control softwares [5].

So Moveo gathers all the requirements of a robust and advanced robotic arm, with 6 DoF. Also it can be built out of 3D printed parts, making it achievable from an economic and logistic point of view, and, on top of everything else, its motors are controlled with Arduino software, making it compatible with plenty of robotic software. Therefore, Moveo is a perfect target for study and research basic and concurrent programming, mechatronics, electronics, forward and inverse kinematics, path planning, dynamics and control, including application of modern AI techniques and its software base control makes it compatible even with some computer vision applications.

In this work we will test the assembling and commissioning of the Moveo robotic arm and its use with reinforcement learning techniques.

2.2. Reinforcement Learning

Reinforcement Learning is a popular branch of AI that currently and for the last few years have led to a great excitement in the field due to its combination with deep learning. RL is based on an agent that interacts with its environment driven by a policy that maps states into actions. As presented in [7] the application of deep neural networks in Reinforcement Learning setups for developing novel agent behaviours, termed Deep Q-Network, led to notable performance on policy learning employing end-to-end reinforcement learning in high-dimensional domains. The target of using Deep Learning in Reinforcement Learning tasks is to be able to define a policy through a Neural Network via parameters that the Neural Network can tune during the training. The fact that a policy is defined by parameter allows to perform gradient optimization techniques [8], this was the main advance that allowed to exploit reinforcement learning. This policy gradient optimization path has given rise to several techniques to improve the method that focus different aspects of the training, from improving the stability and convergence of the process by means of an actor critic [9] to reducing or optimizing the training time reusing episodes during training in continuous control [10].

2.2.1. Deep Reinforcement learning techniques for robotics

The ideas that led Deep Reinforcement Learning and Deep Q-Learning to success are adapted to continuous control in [11] using a model-free algorithm based on deterministic policy gradient and an actor-critic architecture. This way more than 20 simulated physics problems were solved with the same network architecture. The problems setups were for solving tasks like vehicle driving, legged locomotion or dexterous manipulation. Since then, the repertory of control problems have been continuously growing and the Deep Reinforcement Learning framework has made it possible to develop simulated agents that can handle highly dynamic motor skills [12].

Nevertheless, there is difference between simulation and real world and most of the algorithms that achieve excellent performances in simulated environments really struggle to be applied in the real world, not demonstrating many of the capabilities proved in simulations. There are many reasons, the main one is the difficulty of exploration without endangering the robot or its environment which leads to sample complexity. However, there are algorithms such as Guided Policy Search (GPS) [13] capable of training policies directly on real robots, developing complex manipulation skills, nonetheless the interaction with tit environment is reduced. Also, there has been the attempt to extend this method to learning vision-based manipulation policies [14]. Furthermore, parallelizing training across multiple robots was also explored [15].

An obvious alternative for training a policy directly into the real world is to train the policy in a simulation and then transfer it to the real world. This approach brings new challenges in bridging the discrepancies between the simulated environment and the real world.

This problem can be approached from two different perspectives. The first one is to try to model an accurate environment simulation to reproduce faithfully the real world. This approach was thoroughly studied before the rise of DRL techniques.

Nevertheless, only a few basic problems suit this approach, like rigid articulated robots [16] , but since many physics effects may not be accurately reproduced, many problems are not properly tackled with this approach, such as flexible bodies [17], area contact [18] or interaction with fluid. [19]. Also more accurate simulators have been built based on Finite Element Methods [20] that closely match the previous real world effects and interactions but when it comes to robotic problems such as motion planning or prolonged interactions these simulators can become extremely computationally expensive and furthermore depending on the problem tackle the simulation can become numerically ill conditioned. On top of everything else, even if there were a simulator capable of reproducing and modeling faithfully all the physical effects of interest, detailed and accurate model parameters would still be needed. Mass distribution and material properties are the most basic ones, and each robot would need to be deeply studied, there have been several attempts on different robot platforms, such as legged robot [21], helicopter [22] or UAVs [23].

On the other hand, as a second approach to bridge the discrepancies between simulation and real environments, is to analyse the problem of transferring control policies from simulation to real world. This problem can be posed as a domain adaptation problem, where an agent trained in a simulated domain is transferred to an alternative one domain, the real world. To succeed in this approach the main assumption is that alternative domains have characteristics in common, not physical ones, but that the knowledge acquired in the simulation such representations or behaviours can be also useful in alternative real domains. [24], [25] developed an encouraging approach taking advantage of shared characteristics for learning invariant features. Also in works such as [24] is explored the use of pairwise constraints to potentiate networks to acquire similar embeddings during the learning phase from samples from alternative domains that are labeled as being similar. This was achieved by means of weakly aligned pairs of images from source and target domain for adapting deep visuomotor representation.

Adversarial losses has been another technique applied in this task of transferring policies between different domains [26] and making agents to perform similar behaviours in a variety of environments, in this case the environments are simulated.

In this concrete domain of robotic arms, progressive networks have been also employed to transfer policies [27]. Moreover, the previous method is able to reduce notably the amount of information required from physical systems by reusing learned features. Another way to transfer policies from simulation to a real environment is to use data from the real world to train an inverse-kinematic model , where the desired next state of the simulated policy is given to the real world policy, instead of providing the direct kinematics [21]. Nevertheless, as promising as these methods are, they require data from the real world during training which limits the application and utility of these techniques.

New approaches have born recently, like third-person imitation learning [28], where instead of providing first-person experience to the agent, it is only provided with unsupervised third-person demonstration, so it must learn from imitation, as humans would address the problem. This method insight novel advances from domain confusion and provokes the

agent to yield domain agnostic features. This approach has successfully tackled some basic RL problems.

Following the imitation trend, other authors have focused on one-shot imitation learning [29] oriented for robots that must perform a very large set of tasks on the same domain. So the aim of these authors is to build an agent that, once it has learned a set of tasks on the training phase, it is able to learn another one by imitation in one-shot, using techniques like soft attention and allowing the model to generalize to tasks and conditions previously unseen. So the model can convert any demonstrations into a robust policy able to accomplish an immense variety of tasks. The performance of this algorithm has been tested on tasks like block stacking.

Even though these are all promising approaches and, in theory, would be able to be implemented in real robots, none of them has been tested yet on real robots. So the gap between simulation and reality would be still a challenge but for the technique of Domain randomization.

Researchers have been able to train deep neural network for object detection only with 3D simulated RGB images with non-realistic textures and transfer it to work in the real world without any pre-training with real images [30], they built a detector agent robust to distractors and partial occlusions and achieved an accuracy of 1.5 cm.

Following a similar trend, the same group of researchers, achieve to transfer a policy from simulation to a real robot for nonprehensile manipulation with dynamic randomization [31], being able to maintain the level of performance when bridging the reality gap. This technique has been further applied to more complex tasks such as dexterous in-hand manipulation [32], Fig 2.2, proving its potential and becoming the main state of the art algorithm for robotics when it comes to training a real robot with Deep Reinforcement Learning. Furthermore, in the simulation domain, this technique has also managed to perform more intrincated tasks for robotic grasping with >90% success rate on previously unseen realistic objects at test time in simulation despite having only been trained on random objects [33].

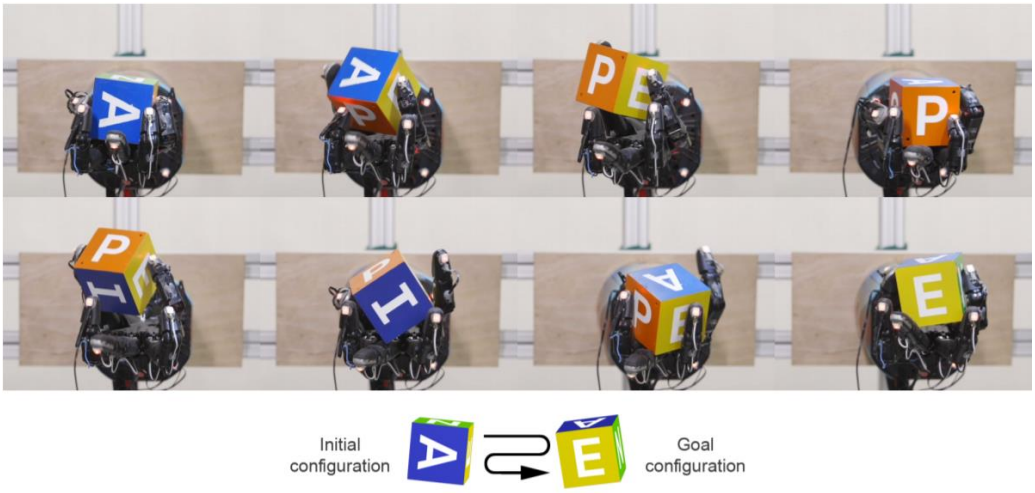


Figure 2.2 Robotic humanoid hand trained with reinforcement learning manipulating a block from an initial to a final configuration, using vision for sensing. From [32]

2.2.2. Deep Reinforcement Learning Benchmarks

For all the novel Deep Reinforcement Learning techniques a benchmark platform to compare algorithm performance became mandatory and a variety of benchmarks were released tackling different applications of DRL techniques, such as solving the Atary games [34] or problems of continuous control [35]. Nowadays we can categorize Open Gym [36] as one of the state-of-the-art benchmarks for Deep Reinforcement Learning, for developing and comparing algorithms. Open Gym claims to be a software package that is maximally convenient and accessible, having combined the best element of previous benchmarks.

Open Gym was developed and is currently maintained and updated by Open AI [37], a non-profit artificial intelligence research company. Open AI is focused on improving and advancing digital intelligence in a way that humanity as a whole can get benefit from. Being free from financial obligations Open AI research can better focus on positive human impact. This Open Gym benchmark is based on a collection of tasks, called environments, with a common interface where the users are able to develop its own agent. Beside the software library, more information about this platform can be found in its website (gym.openai.com). Currently we can find several environments for various kinds of DRL applications such as "Toy text", simple environments to get started, "Classic control" where typical RL literature

control problems are posed, "Box2D" for continuous control simulation in 2D, "Atari" for training high score agents in Atari 2600 games, "MuJoCo" for continuous control task in 3D with a fast physics simulator and a "Robotics" environment for goal-oriented task in manipulation for two specific robotics arms.

Also the Open Gym API can be successfully integrated with robotic hardware, validating RL in real environment [38], extending widely the use of Open Gym DRL techniques. This is made using a robotic middleware Robot Operating System [39] that serves as a common structure for organizing everything needed in the creation of a robot training environment from scratch and also provides the necessary interfaces to interact with simulation platforms. This architecture has been formalized as a ROS package called openai-ros [40]. But there are also other toolkits that use ROS [41], precisely ROS2, as a tool to connect the OpenAI gym with the simulation platform and successfully train a robot using Reinforcement Learning techniques.

2.2.3. Robotic middleware and simulation platform

Since the simulation of a robotic arm tends to have as a final goal the transfer of the learned policy into a real robot, the simulation platform must be robust but also there is the need for a middleware that performs the communication between modules. Previously has already been mentioned the use of ROS for extending the usability of the OpenAI API acting as a middleware between the RL Gym and the robotic hardware but the use of ROS goes far beyond. Nowadays ROS has become the state of the art of robotic softwares due to this modular architecture and its way of handling and passing information. ROS, an open source project, allows new implementations through packages and can be integrated with plenty of softwares. For the case of simulating a manipulator robotic arm, ROS provides efficient tools for data process and analysis such as 3D visualization with Rviz (ROS Visual) module [42]. Also packages like Moveit [43] can be found, really helpful when it comes to perform motion planning in a robotic arm.

Regarding the simulation platform, it is known that most OpenAI applications in continuous control are tested in the MuJoCo simulator [44] but since it is not open source an alternative seems mandatory to develop this project.

As it has been said in the previous section, if there is the chance of integrating the OpenaAI with ROS, we can easily find a simulator that fits our needs which already has the necessary package to communicate with ROS. The simulator that completely suits this case is Gazebo. A creator of accurate 3D dynamic multi-robot environment capable of recreating complex worlds with fine grained control and high fidelity, and also with a completely open source status [45]. It generates both, realistic sensor feedback and physically plausible interactions between objects (accurate simulation of rigid-body physics) [42]. But most importantly it provides the required interfaces that are necessary to to simulate a robot in Gazebo via ROS [41].

3. Chapter 3: Materials and methods

3.1. Description of the architecture

Multiple modules and areas within the robotic research field are merged to conform this project, thus, before examining each of the different aspects involved one by one is worth to have a global overview of the problem.

Paying attention to Fig. 3.1 It can be intuited how the whole learning set up is structured. Each of the modules depicted in the scheme are going to be tackled along this section and the relation between them will also be explained. Basically a robotic middleware ROS is used as a connection between all the required softwares and its functionalities and also interacts with the controllers of the real robot. To train custom robots in an open source realistic simulator, the Gazebo simulator is employed and to perform motion planning for robotic arm manipulation tasks, the Moveit package is also used. These softwares are already integrated with ROS through the corresponding packages. Nevertheless, the module in charge of providing a learning set up for the robot to learn is not fully integrated with ROS and the package Opeanai-ROS is used in order to create a functional interface.

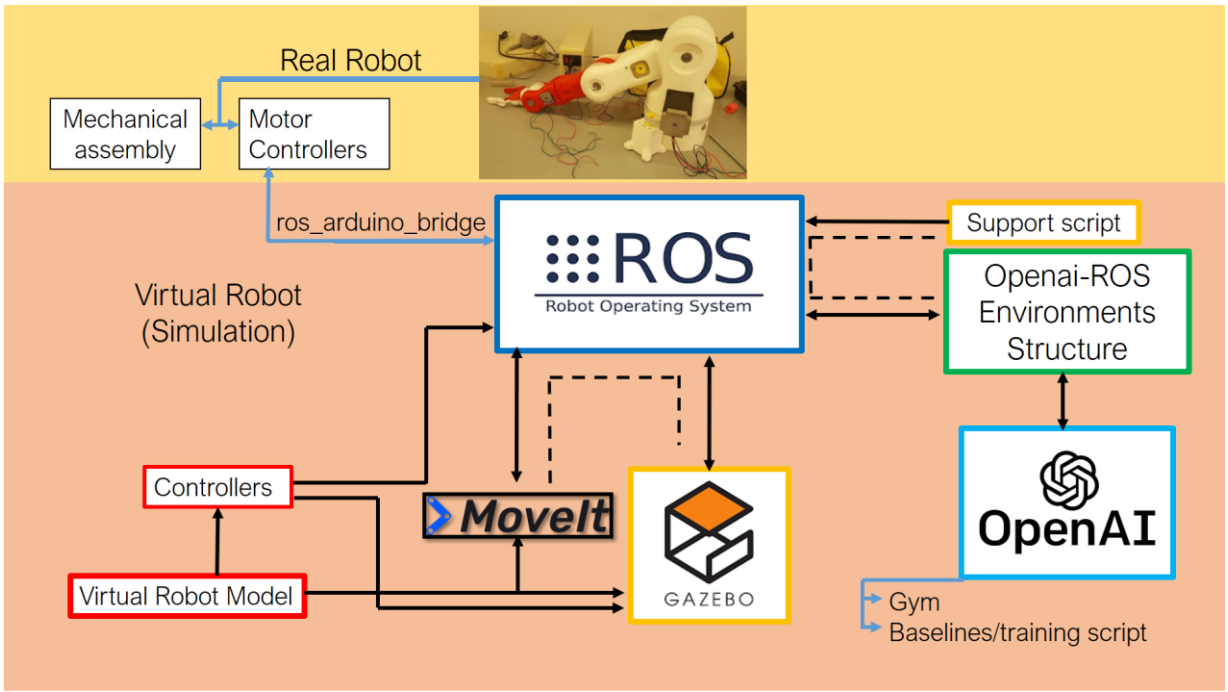


Figure 3.1 Basic Scheme of the structure for deploying intelligent behaviour in a robotic platform

3.2. BCN3D Moveo Robotic Arm.

Moveo is a fully operable 6 DoF robotics arm, built out of 3D printed parts and controlled by Arduino software, versatile from design to applications. Moveo was born as an Open Source initiative by BCN3D in collaboration with the Departament d'Ensenyament de la Generalitat de Catalunya with the aim of bringing robotics arms and additive technologies closer to the public in general and to students in particular.

As an open source technology a Github repository [46] can be found with all the information and files needed to build Moveo from scratch.

3.3. Experiments Description.

In this project different experiments are performed in order to step solidly toward the final goal and are going to be explained along this section. Since the project final objective of deploying intelligent behaviours on the Moveo robot was hard to fulfil in time, the main objective has been segmented in three different experiments. As a guide, the schema shown in Fig. 3.2 can be used.

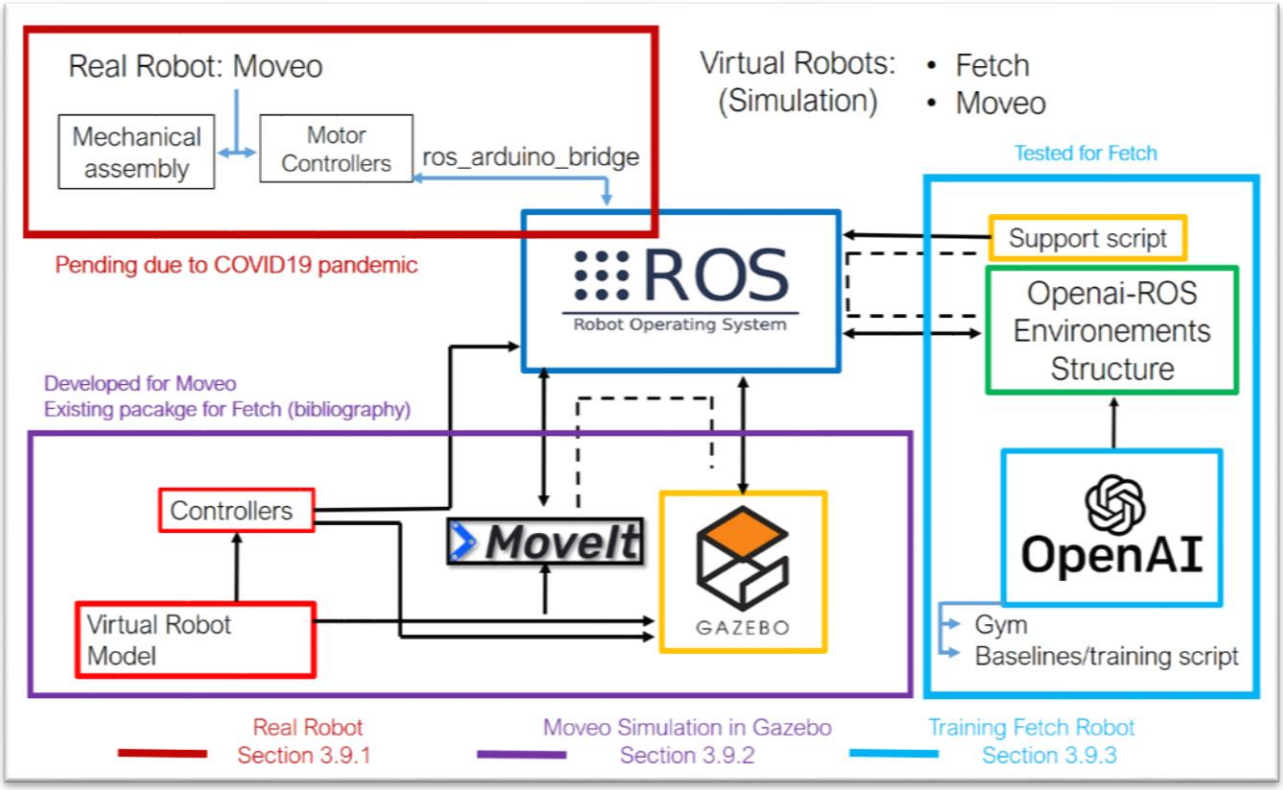


Figure 3.2 Domain of the overall problem that is covered by each experiment

3.3.1. Mechanical Assembly fo the Moveo Robotic Arm

This experiment covers issues related to the mechanical and electronics assemblage in order to obtain a correct functioning of the robot. The attention is focused on achieving a proper transmission along the kinematic chain in the manipulator avoiding power losses due to slips in motors shafts or impaired joint mechanisms.

3.3.2. Simulating the Moveo Robot

This experiment is focused on developing a functional replica of the real Moveo in the simulation platform (Gazebo) that will be used for the Reinforcement Learning module (OpenAI) to simulate the training episodes. The robot model needs it associated modules for being controlled inside the simulation, those modules are mandatory since are also used by during the training to perform the corresponding action.

3.3.3. Learning to control the Moveo Robot using OpenAI

This experiment explores the interface between the Reinforment Learning module and the simulation platform through the robotic middleware ROS employing the package `opeanai_ros` from ROS library. In order control a simulated robot through the opeanai_ros package, a fully functional model of the robot must be implemented in gazebo with all the corresponding features that will be needed for the training. We do not yet count with a functional model of Moveo for the Gazebo simulator, due to issues that will be further explained in this document. Nevertheless, the interface generated by the package openai_ros is robot related but focused on the application field of the robot. So even though we will not train the Moveo robot, we will train a robotic manipulator just as Moveo is. Therefore, once the Moveo robot model is functional in Gazebo, we will be able to perform the training it using our custom constructed interface obtained by performing slight adjustments to the interface used in this experiment.

4. Chapter 4: Methods-Mechanical Assembly fo the Moveo Robotic Arm

A precise guide to build the robot can be found in the Instructables website [47], besides, the BCN3D Github repository provides all the information needed to get the differents pieces of the arm, Fig 4.1, and also an assembly instruction manual. The arm core can be built using a 3D printer or if you are not able to access one also can be ordered on web pages like 3DHubs [48], the CAD files for the 3D pieces can be found on the BCN3D repository.

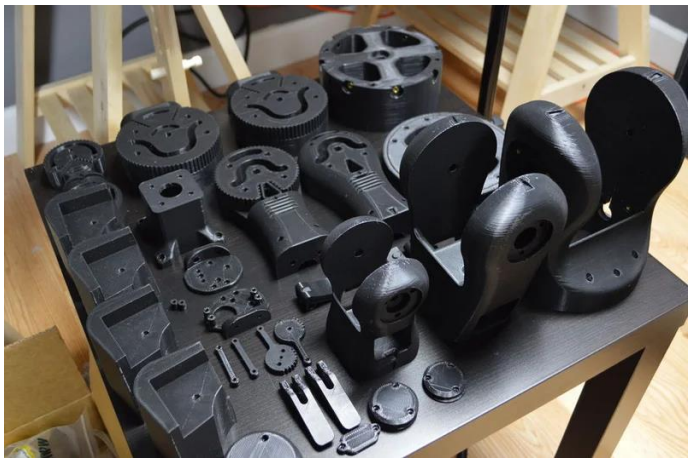


Figure 4.1: 3D printed parts that conform the Moveo robotic arm. From [47]

In the assembling phase, metal screws are used to join fixed parts. To achieve robust unions heat-set inserts are attached to the screws counterpart, the plastic 3D printed parts are large enough and therefore compact but have notable low heat resistance. So the printed pieces are malleable under heat and the heat-set inserts are easily attachable. In addition to that the inserts remain solidly attached once the piece gets cold. Nevertheless, the insert must be clamped fast to avoid heat transmission and deformations on the surrounding.

The locomotion of this robotic arm is driven entirely by toothed pulleys systems but for the wrist torsion joint that is coupled directly to the motor shaft end. All the toothed belts used are T5 metric and had a width of 16 mm.

The following section are dedicated to highlight concrete aspects to takes into account during the assembling phase to obtain a locomotion as functional as possible

4.1. Base joint

The base joint can be seens as a twisting joint that allows rotational motion along the global vertical axis.

The bottom part of the joint, Fig. 4.2.a, is fixed to the base table, on it there are attached eight 5 mm ball bearings on the periphery of the top surface to avoid tangential friction when rotating. Also to maintain both parts of the joint centered in the same axis a metal screwed shaft passes through the center of both parts and it will be coupled to the top part of the joint using a nut. So to avoid friction between the shaft and the bottom part of the joint another ball bearing is attached using heat to the center of the piece.

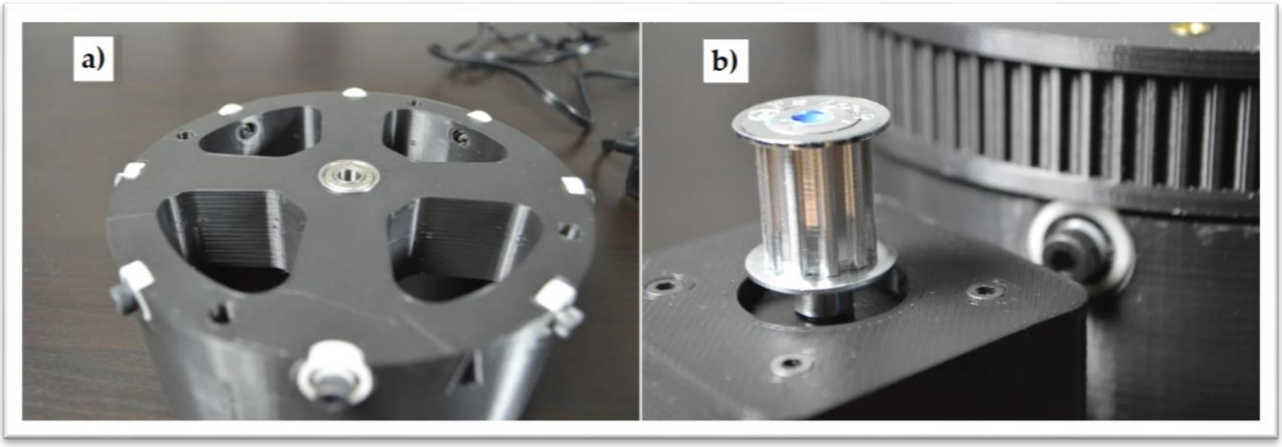


Figure 4.2: Base joint of the Moveo robotic Arm. a) Bottom part. b) Pulley. From *[47]*

The top part of this joint is a rotatory disc that is attached to the rest of the robots by 6 screws. On its periphery the surface is toothed so the toothed belt fits properly. Also there are two toothed grooves on the surface of the disc to introduce and lock both ends of the toothed belt.

The motor that locomotes this joint is settled side by side to the base of the robot Fig. 4.2.b. To obtain a good performance the pulleys attached to the motor's shaft must be glued to avoid slippage between the motor's shaft and the pulley. Moreover, the motor should be screwed to the table after the pulley is settled and tensed, this articulation has no tensioner for the belt. The motor used for this articulation is a stepper motor Nema 17HS24.

4.2. Shoulder Joint

This joint is placed on top of the rotatory disc of the base joint. The bottom part of this joint holds the locomotion that is driven by two powerful motors Nema 23HS45. The motors shaft ends pass through the piece and out of the inner side of the piece so the pulleys are glued to the motor ends.

On the inner side there are also installed belt tensioners, to achieve a good performance on the pulleys-belt locomotion, Fig 4.3.a

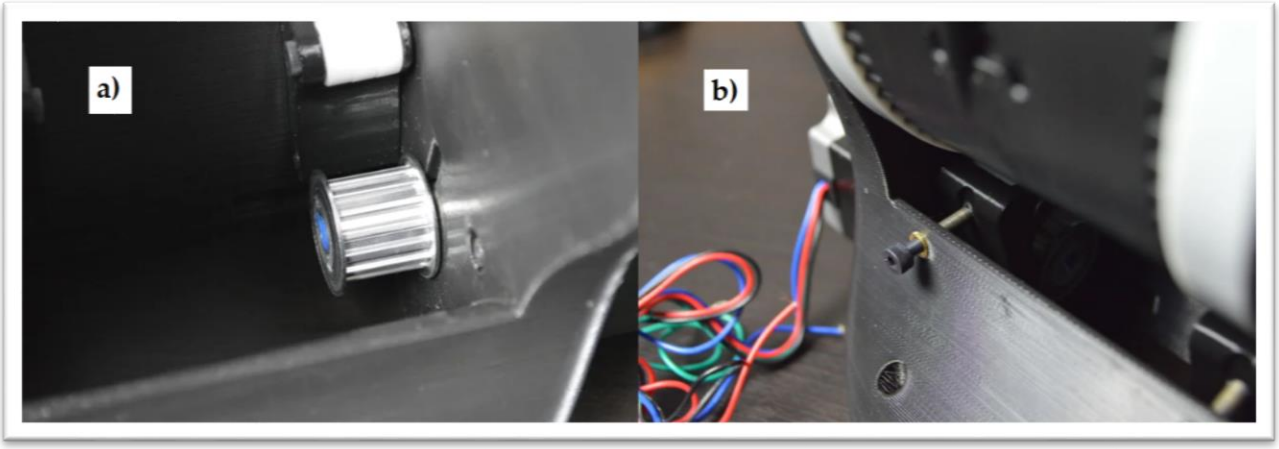


Figure 4.3: Pulley system of the shoulder joint. From [47]

The shoulder joint is assembled by placing a rotating shaft through both parts of the joint. The bottom part is the one equipped with the corresponding bell bearing for avoiding friction. The top part has the belts attached so the locomotion is achieved by placing the belts around the pulleys. Finally, the belts are tensed using the tensioners Fig. 4.3.b

4.3. Elbow joint

This joint is very similar in terms of kinematics to the shoulder joint, and therefore the mechanical distribution of the locomotion system is alike. On the bottom part of the joint, the part that is attached to the top part of the shoulder joint, are placed the motor, a stepper motor Nema 17HS13 with gear ratio 5:1, and the pulley. In this case this joint is locomoted only by one motor, since the torque force that is going to be needed is notably less than in the shoulder joint.

The top part of this joint also holds the wrist motor for the torsional joint that has to be assembled before finishing the mounting of the elbow joint.

To set the locomotion system the mechanism is similar to the shoulder one, the belt is attached to the top part of the joint, then the belt is placed around the pulley and the rotating shaft is passed through both parts of the joint, then the belt is tensed screwing in the tensioner.

Figure 4.4: Assembled elbow joint. From [47]

4.4. Wrist Joint

The wrist joint is a two degrees of freedom joint. Firstly, a torsional or twisting join is found, which provides a rotational motion around the longitudinal local axis of the robotic arm. As seen in the picture adobe, Fig. 4.4, the threaded rod that emerged out the top part of the elbow joint is the shaft that will drive the torsion. The motor of the wrist joint, a stepper Nema 14HS13, is therefore located inside the top part of the elbow joint and the threaded rod is linked to the motor's end through a shaft coupler like the one in the Fig. 4.5.

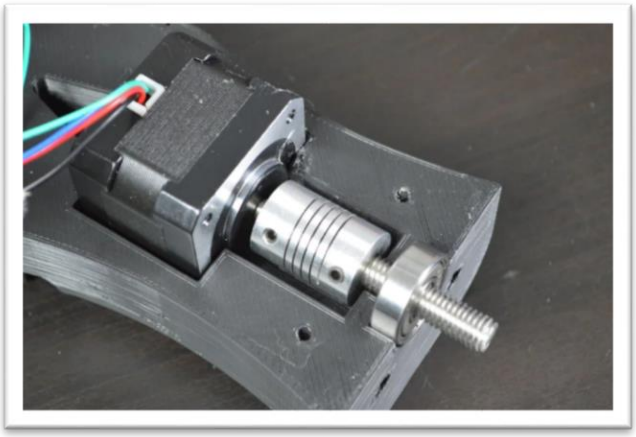


Figure 4.5: Nema 14HS13 assembled to the transmission system for the torsional joint of the wrist. From [47]

The coupler is used to dampen the bending forces and avoid damaging the motor mechanism. Besides, to avoid bending displacements and friction, a bell bearing is placed for the threaded rod at the end of the piece.

The second joint of the wrist is another rotational joint very alike to the previous joints of its same kind. The most curious aspect here is how the top part of the elbow joint connects with the bottom part of the rotational joint.

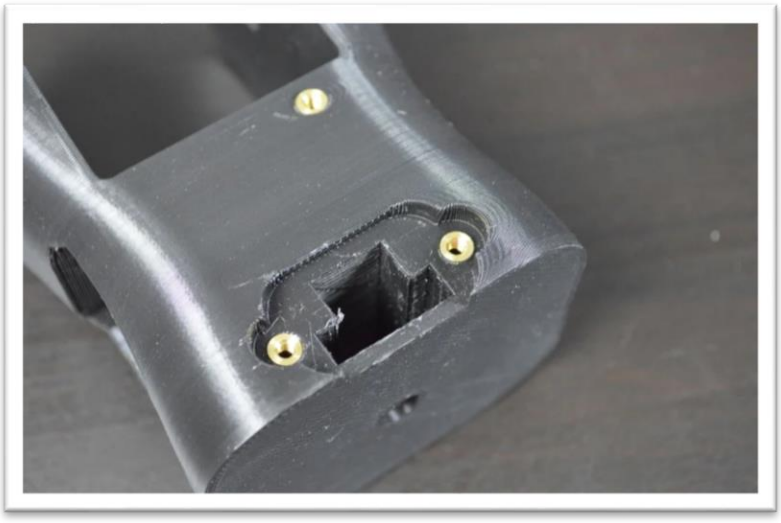


Figure 4.6: Top part of the torsional joint of the wrist. From [47]

Attending the adobe picture, which presents the bottom part of the wrist rotational joint, Fig. 4.6, a hole in the bottom face of the part can be spotted, there goes the threaded rod of the top part of the elbow. To obtain a fixed union, through the side slot two items are going to

be placed. The slot has a step pyramid shape, see Fig. 3.7, so it can be analysed as if each step is a particular slot. Firstly, a bell bearing through the base (bigger) slot and secondly a locknut through the smaller one. This locknut will lean on the bell bearing and the centers of both pieces will be coaxials to the threaded rod, see Fig 3.6. Therefore, if the threaded rod is introduced in the hole it can be screwed on the locknut and blocked. For this attachment to work there must be two things carefully measured. The length of the threaded rod, so when it is fully screwed both parts ends up tight enough and the joint is not limp. And the size of the smaller hole. Its width must be exactly the same so the locknut does not spin inside the slot and the twisting motion is done properly. To ensure a correct behaviour of this joint the locknut has been slightly modified with time-hardening mastic, this way any chance of spinning is prevented.

As previously said, on the top part of the torsional joint, it is found the bottom part of the second joint of the wrist, another rotational joint. The design is really similar to the previous rotational joints but a bit smaller in size. On the bottom part of the joint the motor, a stepper Nema 14HS13, and the pulley are placed and also the belt tensioner and the bell bearing for the shaft. The top part of the rotation wrist joint has the toothed surface for the belt to fit and also the slot to lock the belt, bot parts are assembled using a smooth shaft that will serve as rotation axis

4.5. End effector

The end effector is attached to the top part of the rotational joint of the wrist, nevertheless, the end effector of the Moveo robotic arm is out of the domain of this work. It is a parallel-jaw gripper driven by a servo-motor but the aim of this work is for the robotic arm to be compatible with several different end effectors that would broaden the use of this low-cost robotic arm. So future works might tackle this topic.

4.6. Controllers assembly

The controller of the robotic arm is composed of several parts. For the controller itself it is used an Arduino Mega 2560 that is coupled with a RAMPS shield so the Arduino can interface with the motor. As the stepper motors that have been used on the robotic arm, the

RAMPS shield is normally used on 3D printers so these items have excellent synergy to control the robotic arm as well.

The RAMPS must be equipped with correct stepper drivers, typically a TB6560 would do the work. But since some motors used on Moveo are considerably larger than those used on 3D printers it must be taken into account that these motors might work at higher current. Therefore, the drivers used must be able to handle high currents.

Assembling the control, electronics is not an over difficult task. It starts by placing the RAMPS shield onto the Arduino Mega and connecting both by applying firm pressure, making sure that the power screws of the RAMPS are on the same face as the USB port of the Arduino Mega 2560.

When the RAMPS is mounted the stepper motor drivers must be assembled following the singleline schema in the Appendix. This task can be simplified using pin drivers, like DRV8825.

5. Chapter 5: Methods - Simulating the Moveo Robot

5.1. ROS (Robotic Operating System)

ROS [39] is a software interface that serves as a catalyst of every component involved in training a robot. As a collection of tools and libraries, ROS provides the background to manage all the different processes involved in a robotic project and the communication between them.

The most basic unit of communication used in ROS is called topics. Using topics, we have a way, firstly, of informing the robot to perform an action and, secondly, a way of retrieving any information about the robot state. Relying on topics, ROS has more sophisticated communication systems such as services or actions.

ROS uses packages to organize its programs. Those packages can be thought of as a repository for all the files involved in a specific ROS program. The files in a packages are organized following this structure:

- Launch folder: Contains the launch files, that are in charge of executing programs
- Src folder: Contains source files, as cpp and python files
- CMakeList.txt: Is a list of cmake rules for compilation.
- Package.xml: Contains package information and dependencies.
- Config folder: Optional folder for parameters definition files.

When a lunch file is executed, it is indicating ROS to create a node, which basically is a process that performs computation, it also can be seen as a program made in ROS. A concrete node is related to a certain package and performs what it is specified in the concrete source file that is being launched.

ROS also has a parameter server, a dictionary that ROS uses to store parameters. The parameter server is used by the nodes, normally to store information about configuration parameters.

In order to manage all the processes, ROS has a main process that manages all of the ROS system, it is called roscore, it has to be launched before any communication with ROS is

available. Also a set of Linux system environment variables are used by ROS to work properly, those variables store information such as ros packages path, ros distro(version) or the ROS root.

5.1.1. Topics

Topics can be seens as an information channel where nodes can either write or read information. A node that sends information into a topic is called Publisher that literally publishes information into a topic for other nodes, so they can communicate. But also the information related to the different topic is always available for the user.

The information that is sent through topics is handled by massages. There can be found many different types of messages. The number of variables that a message contains as well as the variable type can vary. Messages are defined by .msg files allocated in the msg directory of a package.

Equivalent to Publisher nodes but to read information that is being published in a topic ROS has Subscriber nodes.

5.1.2. Services - Actions

Topics allow us to induce plenty of behaviours on a robot but they might be insufficient for some applications. For certain tasks there will be necessary a continuous flow of information provided by a publisher node (related to a sensor measurement eg.), and during that task different actions might be needed and actions will send a continuous feedback along the process. So we can deduce that actions work in an asynchronous manner. While other tasks will need for the system to wait until a certain service is performed and the results from the service are received. So we can deduce that services work in a synchronous wait and basically it must be used when the program can continue without certain information.

5.2. Gazebo Model for Moveo.

There have been some developers testing Moveo,[49] for instance , that uses ROS and Moveit as a platform control for the real robot alternatively to using G-code as is the procedural

instruction set used for 3D printers which is not really useful for feedback control. Even though in [49] the author defined a 3D model of the robot for Rviz and created a Moveit package, he directly transfer the information of the motion plan to the real robot, and skip the step of simulating the task, which is an option for simple tasks where there is no aim of using RL techniques. While Rviz is used for visualization, to analyse the states of the robot, basically the relative position of links, Gazebo is by far more complex. What is done in Gazebo can be regarded as generating an experimental replica of the robot in a virtual world. So a Gazebo model for the Moveo must be built.

5.2.1. Manipulator Modeling.

Modeling up a robot from scratch is not a hard task. Robots are defined in gazebo and ROS as Unified Robot Description Format (URDF) file, where all the information of a virtual robot is recorded. The 3D modelling must be carried out in another kind of format as CAD using any of the commertial modelling software, such as Solidworld or Blender. What is stated in the URDF format is the relation between joints and links (parts of the robots). No matter how complex the robot could be the basic structure remains the same, Fig 5.1.

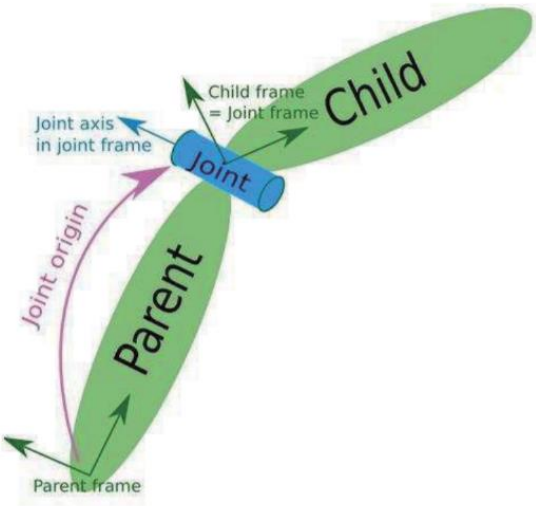


Figure 5.1: Revolute joint example along with it parent and child links. Joints modify the frame of the local reference system of the robot.

In a URDF description, there are defined the different relations set by the different joints between the different links of the robot. The whole coded elements between tag <link> and the tag </link> are defining one specific link. The same way happens with the joints description, between the tags <joint> and </joints>. In the link definition there is the chance of importing the STL to provide the link of the robot with a concrete 3D shape, modelled in CAD.

Also in each link definition the physical properties must be defined, such as the inertia rotation coefficient. The coefficient should be computed or approximated. But actually, in medium size robots the inertia coefficients do not play an important role since the values are not large enough. Nevertheless, those values must be defined carefully or the simulated model will end up collapsing, Fig 5.2.

The same files used to print the different parts of the Moveo can also be used in the URDF file to define the correct geometries. For this project the model built by [49] for obtaining a virtual Moveo definition in Rviz will be recycled. The information concerning the parts and joints definition will be the same as in [49] but for the inertia coefficient, that were tuned to avoid the collapse of the model, Fig. 5.2. Also, for the joints of the model to stay static under gravity forces in the Gazebo simulation, the controllers of the joints have also to be defined in the model and will be explained in the following section.

Simulating the adjusted model of Moveo in Gazebo provides the following, Fig 5.3.

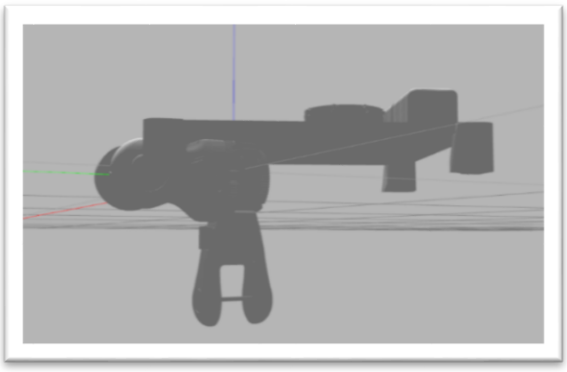


Figure 5.2: Collapsed model of the Moveo robotic arm

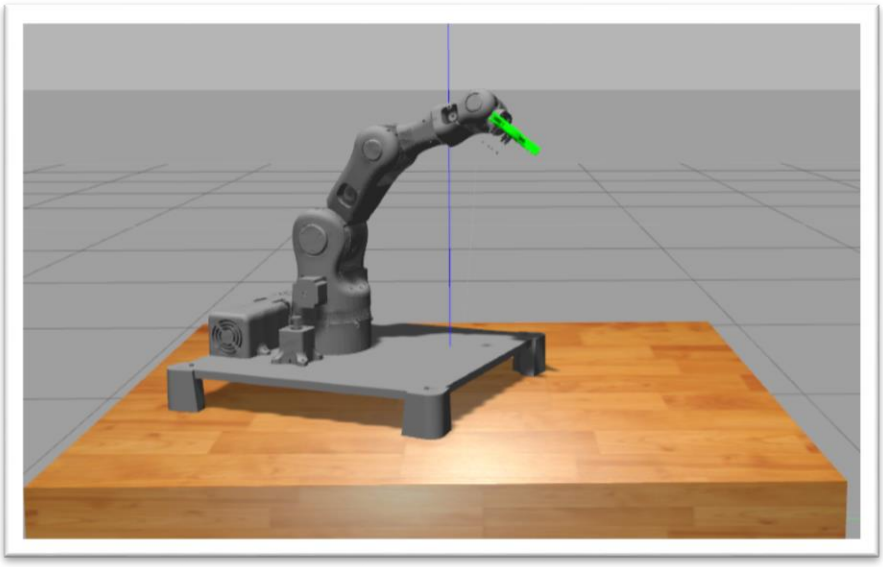


Figure 5.3: Moveo robot model in Gazebo

Focusing only in the 3D model, as were previously mentioned, there was the chance of modifying the end effector depending on the task to perform. For the case of this project would be a good choice to implement a simpler one since in our application there is no need to grasp. Furthermore, some artefact related to the end-effector component can be seen in the simulation, Fig 5.4. Nevertheless, here is no assurance about if those artefacts would affect the simulation.

Figure 5.4: Frame of the Moveo end-effector were floating artifacts are detected.

For robot model definition there also exists the chance of using xacro language [50] to shorten the XML file through the use of macros and optimize the code.

5.2.2. Gazebo-ROS control interface

Those motorized joints that require control during the simulation need a transmission definition in the URDF file associated with the controller for the specific joint. Transmission is an element of the control pipeline that transforms efforts/flow variables [42]. The effort controller transmission of a joint can be rapidly configured using a basic transmission code block, can be checked in the guide [51].

There are various types of hardware interfaces that can be defined alternatively to the effort controllers such as velocity or position controllers.

Additionally, to the transmission tags for each motorized joint a Gazebo plugin must be added to the URDF file to parse the transmission tags, loading the appropriate hardware interfaces and controller manager [42]. Adding the gazebo_ros_control plugin code allows to create a list of controller_manager services, used to manage the controllers in order to list, start, switch or stop them. The controllers play an essential job in the behaviour of the manipulator. For manipulation the trajectory controllers are typically employ, a sort of position tracking controllers. The controller type, joint names and PID parameters have to be defined in a YAML file [52].

Additionally, for manipulation task and motion planning Moveit [43] is the current stay of the art employed tool. Moveit is a software that is comprised in the extense ROS collection of packages, integrated with the Rviz interface which makes motion planning a trivial task for the user. In order to use Moveit in a custom robot a particular ROS package has to be created, luckily the GUI (Graphical user interface) provided by Moveit makes it also a trivial task. Furthermore, it can be managed from Python scripts making it a useful tool for AI applications.

Nevertheless, performing the motion in the Rviz interface of Moveit does not perform the motion in the simulator (which actually acts as a real robot), everything is performed in the Moveit internal simulator. So Moveit is just providing the necessary ROS services and actions in order to plan and execute trajectories but it is not able to pass the information to the robot.

This is solved by performing certain modifications in the particular Moveit package created for the custom robot.

Precisely, what is done is defining an Action Server that will be used to control the joints of the robot. Firstly, a YAML file indicating the controller type and the associated joints is created in the Config folder of the Moveit package for the custom robot and its corresponding Launch file for the YAML. Secondly, another YALM file only with the names of the joint to be controlled is created and finally the Launch file `my_robot_planning_execution.launch` is created, allowing, by launching it, the communication between Moveit and the simulated robot model.

6. Chapter 6: Methods - Learning to control the Moveo Robot using OpenAI

6.1. Reinforcement Learning Techniques

Being aware of the importance of Deep Reinforcement Learning techniques, the technical aspect that allowed the adaptation of the conventional RL algorithm to new domains are going to be reviewed in this section. Focusing the attention on the HER algorithm since is the algorithm that will be used for the training during the testing of the Openai-ROS interface. Also the ideas behind the HER algorithm are the ones that allowed the extension of RL to continuous and extense environments such as manipulation.

6.1.1. Background

Considering a standard basic Reinforcement Learning problem of an agent interacting with an environment E in discrete timesteps. At each timestep t , the agent receives an observation x_t from E, takes an action a_t determined by a policy $\pi: S \rightarrow A$ (maps states into actions) and receives a reward r_t depending on the impact of the taken action in the environment. Environments can be partially observables, so to define the environment state the whole history of observation, action pair might be required $s_t = (x_1, a_1, \dots , a_{t-1}, x_t)$. In case that the environment is fully-observable the current observation of the environment is enough to define the state it $s_t = x_t$.

The policy that define the agent behaviour has the function of mapping states into the actions or into probability distribution over the actions $\pi: S \rightarrow P(A)$, depending if a deterministic or stochastic scenario, respectively, is considered. As well as the environment reaction to an action can also be stochastic. This kind of problems can be posed as a Markov Decision Process where there are: state space S , action space $A = \mathbb{R}^N$, an initial state with density $p_1(s_1)$, a conditional transition dynamics distribution $p(s_{t+1} | s_t, a_t)$ that satisfy the Markov property $p(s_{t+1} | s_1, a_1, \dots , s_t, a_t)$ and a reward function $r_t(s_t, a_t)$.

To measure the agent-policy performance each state has associated a return defined as the sum of discounted rewards:

$$R_t^{\gamma} = \sum_{t=i}^{T} \gamma^{i-t}\ r(s_i, a_i)\ \ ;\ \ \gamma \in [0,1] \qquad [1]$$

The goal in Reinforcement Learning is to optimize the policy and find the one that maximize the expected return (cumulative discounted reward) from the initial distribution $J(\pi) = \mathbb{E}\left[R_1^{\gamma} \middle| \pi\right]$.

6.1.1.1. Deterministic Policy Gradient (DPG)

Policy gradient algorithms might be the most populat class of continuous action reinforcement learining algorithm. The idea behind this algorithms, as intuitive as might it be, is to optimize the parameters θ of a function approximator of the policy following the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$. The essential result underlying these class of algorithms is the Policy Gradient Theorem [8], that can be extended to deterministic policies, posing the deterministic policy gradient theorem [56]:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\nabla_\theta \mu_\theta(s)\ Q^{\mu}(s,a)|_{a=\mu_\theta(s)}\right];\ \mu_\theta: S \to A \qquad [2]$$

Deterministic Policy Gradient Theorem allows to simplify the computation in those model-free reinforcement learning algorithms not based in policy iteration (intercalling policy evaluation and policy improvement through greedy maximization) as the majority of the algorithms [56]. Concretely in continuous action spaces, as the robotic domain is, greedy policy improvement is problematic due to the fact that global maximization is required at every step. Therefore a simpler and computationally tempting alternative is to perform policy optimization in the direction of the gradient of Q^{π} instead of maximize Q^{π}. In this approach the policy parameters θ^{k+1} are updated in proportion to the gradient $\nabla_\theta Q^{\mu^k}\big(s, \mu_\theta(s)\big)$:

$$\theta^{k+1} = \theta^k + \alpha\, \mathbb{E}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu^k}(s,a)|_{a=\mu_\theta(s)}\right] \qquad [3]$$

From this theorem both on-policy actor-critic algorithms and off-policy actor-critic algorithms have been derived and also these ideas have been applied to Deep Reinforcement Learning.

6.1.1.2. Deep Deterministic Policy Gradient (DDPG)

This algorithm provides the necessary modifications to the DPG algorithm, to allow it used combined with neural network function approximators in order to apply this technique in large states and action spaces [11].

This model-free RL algorithm is also usable for continuous spaces, it is based in two neural networks. The first one is a target policy (actor) network $\pi: S \rightarrow A$, the second one is an action-value function approximator network (critic) $Q: S \times A \rightarrow \mathbb{R}$. As it is common in actor-critic architectures, the critic's work is to approximate the actor's action-value function Q^{π}. In this off-policy technique the episodes are generated using a behavioural policy, not a random one by a noisy version of the target policy $\pi_b(s) = \pi(s) + \mathcal{N}(0,1)$. The critic is trained similarly to the Q-function in DQN but the targets y_t are obtained using actions selected by the actor, for example:

$$y_t = r_t + \gamma Q(s_{t+1}, \pi(s_{t+1}))$$

The actor is trained with mini-batch gradient descent on the loss, sampling s from the replay buffer:

$$\mathcal{L}_a = -\mathbb{E}[Q(s_t, \pi(s_t))]$$

Then the gradient of the loss $\mathcal{L}_a$ with regard to the actor parameters can be computed through backpropagation by combining the critic and the actor network.

6.1.2. Hindsight Experience Replay

This technique arises due the fact there are certain setups where standard RL algorithms are bound to fail. Usually when an agent can not learn it is due to the lack of information received from the environment. If there is no chance to shape the reward accurate enough to provide information to reach the goal, what is usually used is $r_g(s,a) = -[s \neq g]$. The previous fact combined with a large state space can lead to a scenario where the agent does not experience any reward other the -1, which can not be solved by exploration techniques either.

So the solution proposed in [10] is to re-examine padst episode trajectories $s_0, s_1, \ldots, s_T$ where $g \neq s_1, \ldots, s_T$ but with a different goal. While the past trajectory does not say anything about reaching the initial goal, definitely provide the necessary information to reach s_T. So basically

what is done in HER is to harvest the information provided by failed episodes by using off-policy learning and experience replay where the initial goal g is replaced in the replay buffer by s_T in addition to the original trajectory that is also saved in the buffer.

More formally stated, the idea of Hindsight Experience Replay is [10]: "*after experiencing some episode* $s_0, s_1, \dots, s_T$ *we store in the pelay buffer every transition* $s_t \rightarrow s_{t+1}$ *not only with the original goal used for this episode but also with a subset of othe goals*". There must be taken into account that the pursued goal influences the agent behaviour but not the environment dynamics, thus the episode can be replayed with an arbitrary goal assuming that an off-policy algorithm is being used, like DQN[7] or DDPG[11].

In order to apply HER a choice has to be made, which is to state the set of additional goals that are going to be used for replay. The simplest version is to replay each trajectory with the goal $m(s_T)$, which is the final state achieved in the episode.

6.2. OpenAI Gym

Gym is a toolkit for developing and comparing RL algorithms. It was developed by the AI research and deployment company OpenAI [37]. OpenAI Gym supports teaching agents from playing most basic games such as Pong or Pinball to walking or solving dexterous manipulation tasks.

OpenAI Gym provides an open source interface to reinforcement learning tasks. There is no assumption taken about the architecture of the agents and it is compatible with numerical computation libraries, such as the basic deep learning and artificial intelligence libraries, e.g. Tensorflow.

The `gym` library provides an easy-to-use set of RL tasks. It basically is a collection of environments, that describes different types of problems. Working with those environments that have a shared interface, agents can be taught using general RL algorithms. This infrastructure allows the reusability of AI learning algorithms.

Using this library is really easy to initialize an environment from the collection, by employing the function `gym.make( )` and the associated name of the environment in the registry.

Once an environment has been created there are some basic functions to interact with it. Probably the most important one is the `step( )` function, that will inform us of what the actions of our agent are doing to the environment. Actually the function step returns four values, that are

- `Observation (object variable)`: It is an environment-specific variable that represents the observation over the environment. There are a wild range of observations depending on the environment. It can be from joint states of joint velocities if the agent is training an articulated robot to a board state if the agent is playing a board game.
- `Reward (float variable)`: Value of the reward achieved by the action executed in the step. The type of the reward varies between the different environments but the objective is always to boost the final reward.
- `Done (boolean variable)`: Defines weather or not is time to restart(reset) the environment. Most tasks are structured into well-defined episodes and 'done' being True indicates the episode has terminated.
- `Info (dict)`: Holds diagnostic information useful for debugging.

This process gets started by calling el function `reset( )`, and also must be called every time an episode ends so the 'step' function is the implementation of the classic agent-environment loop Fig 6.1 Each timestep, the agent chooses an action, and the environment returns an observation and a reward.

When it comes to sample random actions, it is done from the environment's action space. Each environment has defined an action_space but also with an observation_space.

These attributes are variables of type Space that state the shape of valid actions and observations. It can be found in different spaces such as discrete spaces, that allows a fixed range of positive values, or box spaces that allows a representation of an n-dimensional box described by an array of n-numbers.

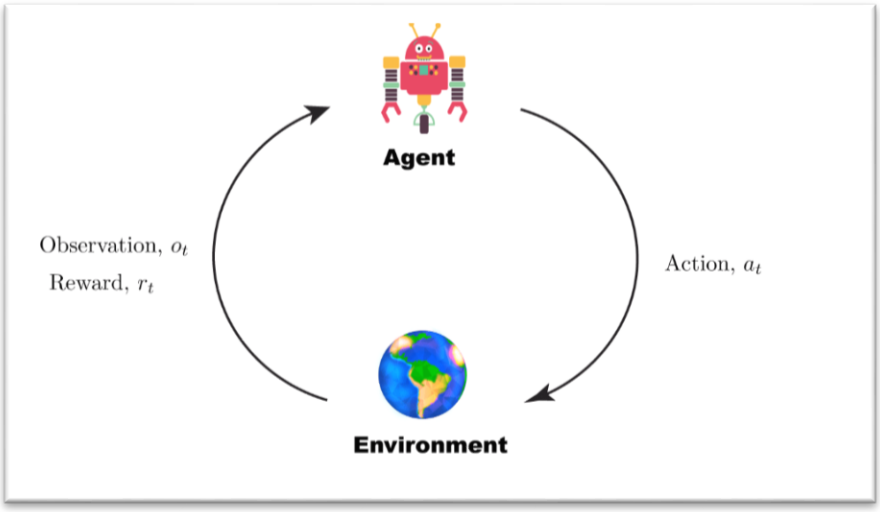


Figure 6.1: Reinforcement Learning agent-environment interaction loop

6.2.1. The registry

The main purpose of the OpenAI Gym is to provide a collection of environments that share a common interface and are improved, versioned and allow comparison between each other. There is a list of environments that can be called using the `gym.envs.registry`. It retrieves a list of EnvSpec objects that defines parameters for a particular task, including number of trials to run or maximum number of episodes but mainly are environment IDs, so when creating a custom environment, it must be added into the registry and thus make them available.

6.3. Using OpenAI with ROS - openai_ros package

Even though the OpenAI Gym allows to train robots using simulation platforms like MuJoCo, it does not provide environments to train ROS based robots using Gazebo simulations. So the company The Construct [58] has developed the openai_ros package that solves this issue by providing a common structure, Fig 6.2, for organizing everything needed to create a robot training from scratch.

Without focusing deeply into details the composition is the following:

- Training Script: Defines and sets up the Reinforcement Learning algorithm that will teach the agent. It also selects the task and the robot to use and triggers the training.

- Training Environments: Remember that an environment is just a problem definition with a minimal interface that an agent can interact with. So the training environments are in charge of providing all the needed data to the learning algorithm from the simulation and the RL agent environment, in order to make the agent learn. There can be found three different types of environments, that are organized always with the same structure and one inherit for the following, forming a pyramid structure:
    - Task Environment: Environment where all functions related to a specific training session are implemented. Task environments are robot dependent, and obviously there will be needed different task environments for different tasks to be learnt. Inherits from the Robot environment
    - Robot environment: Environment where all the functions needed during training for a specific robot are implemented. It will not include any function related to a specific task, and different robot environments will be needed for different robots to learn. Inherits from the Gazebo environment.
    - Gazebo environment: Common environment for any training or robot. Is in charge of generating all the connections between the trained robot and Gazebo. There is no need to modify it but there can be more than one type of Gazebo environment, such as Goal Oriented environment. Inherits from the Gym environment, the basic structure provided by OpenAI Gym framework.

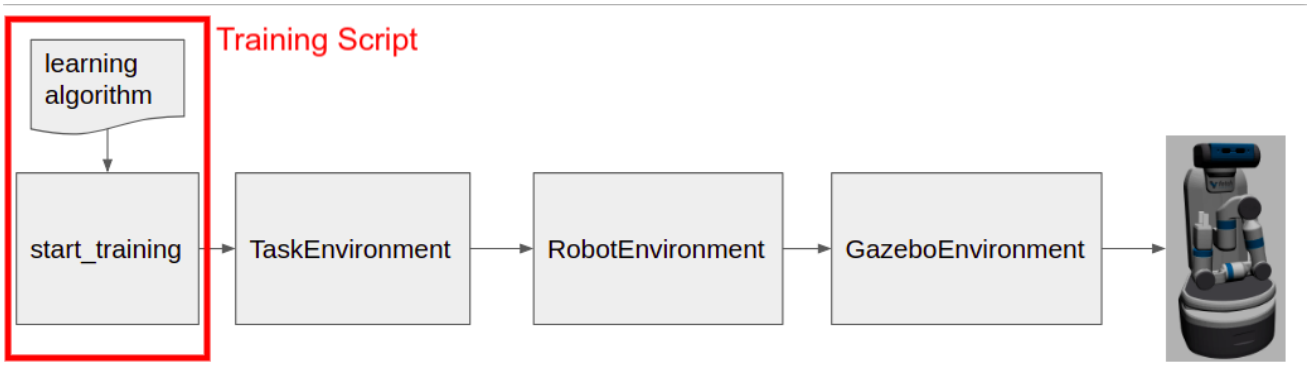


Figure 6.2: OpenAI integrated with ROS-Gazebo Structure.

6.4. Fetch Robot Environment Structure for OpenAI-ROS.

The main recommendation by the developers of the openai-ros package if a new robot wants to be implemented from scratch is to seek inspiration analysing the environments of similar robots. The opeanai_ros package is just a collection of environments of different robots sharing a similar interface.

So in order to get familiarized with the functioning of the OpenAI-ROS system as a whole and to be able to build the corresponding environments for the Moveo, the environments structure for a Fetch Robot is going to be reviewed and analysed. Moreover, the Fetch Robot will be trained in order to manage every aspect of the OpenAI-ROS structure. The Fetch Robot has been selected due to several reasons. Currently is one of the main state of the art robots equipped with a robotic arm and it is used on most research concerning manipulation in the AI RL scenario. The package with the Fetch robot model for setting up the robot in the Gazebo simulation is available in various repositories and it is also one of the robots found in the package openai_ros_examples [59]. The main mismatch between Moveo and Fetch is the complexity and quality of the design, not in the kinematic aspect but in the hardware-software interface. The Fetch robot is not only equipped with a gravity compensation system and further controllers to smooth the motion of the robotic arm but also is equipped with other gadgets as a frontal camera for computer vision. This discrepancy might be determinant in the adaptation process of the Moveo for developing intelligent tasks but will not play an important role when adapting the training set up from the Fetch robot to the Moveo, since the environment is meant to control any manipulator.

The Fetch robot will be trained to perform the simplest manipulation task which is to reach certain points or objects in it configuration space. This task is notably aligned with the final objective of using Moveo for medical research, where the main task would be placing the end effector in a certain point but also with a certain orientation in the space. Those adjustments would need to be tackled in the future but for now the goal is to understand OpenAI-ROS environment structure and training phase.

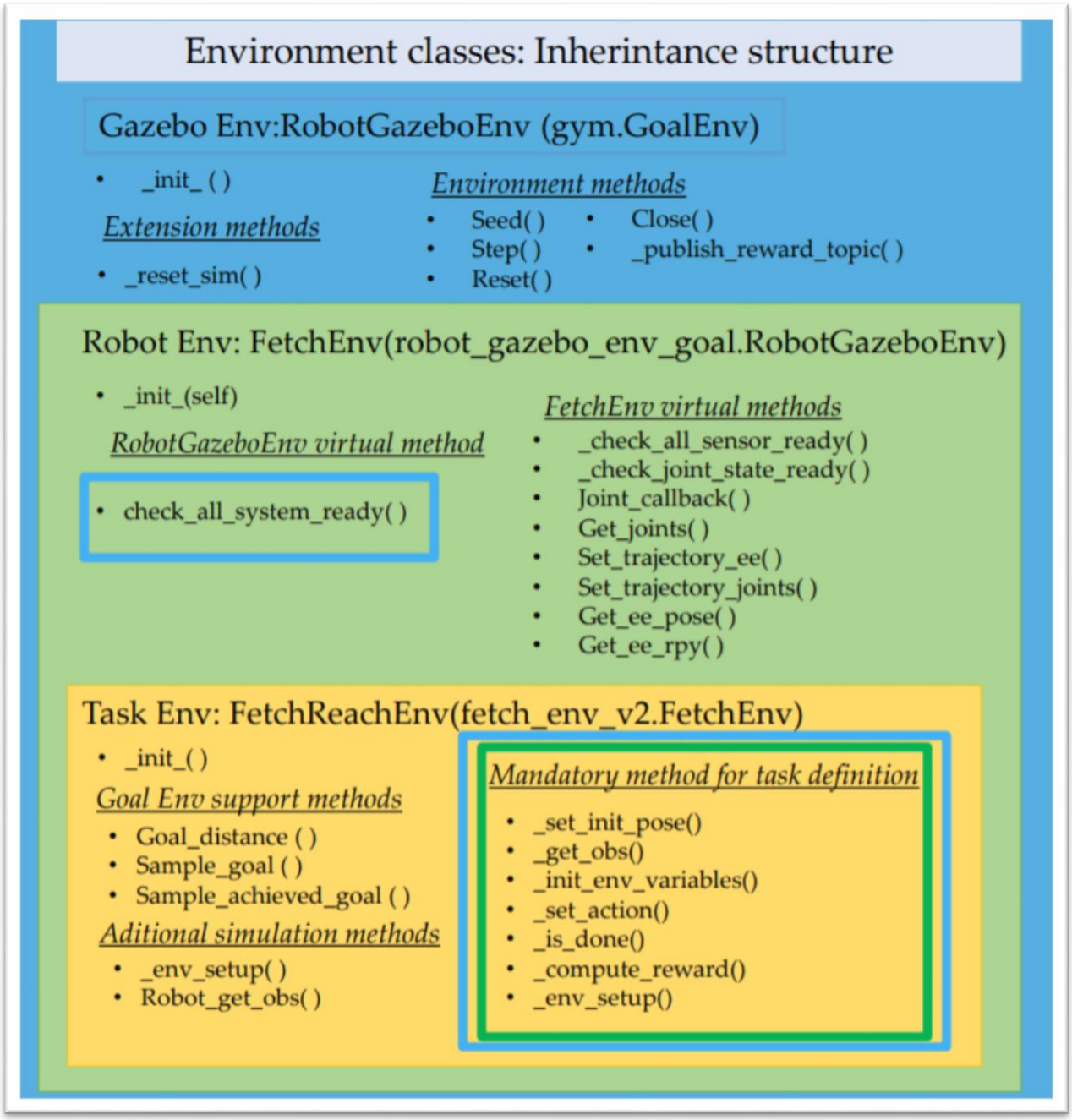


Figure 6.3 Inheritance structure. Framed functions are also initialized in the environments of the corresponding frame colors. Are the functions demanded by gym.GoalEnv.

As a reference for the environment inheritance structure Fig. 6.3 can be used. Beside it is recommended to go to our repository [60] and have the code present during this section.

6.4.1. Gazebo Environment

This environment is the base of the inheritance structure. It will remain untouched for any application, regardless of the robot or the kind of task to solve. Basically its function is to connect the Robot environment to the Gazebo simulator by taking charge for instance of the reset of the simulator each training episode or resetting the controllers if needed, see Gazebo Environment in Appendix 2.

The most important function in the Gazebo environment are the following:

- step( ) : This function defines how to handle the simulation during the training. It manages the simulation steps, is in charge of pausing and unpausing the simulation.
- reset( ) : This function states how to reset the simulation after each training episode.
- close( ) : This function shuts the ROS system.
- publish_reward_topic ( ) : This function has nothing to do with the training but with the results visualization. It publishes the reward for each step in the ROS topic /openai/reward.

Also there is one important thigh to take into account about this environment and it is that it demands three parameters in its constructor:

- robot_name_space : A string variable with the namespace of the trained robot
- controller_list : An array variable containing the name of the controllers of the robot.
- reset_controls : A boolean variable that indicates if there is required to reset the controllers when resetting the simulation.

This is an interesting information to bear in mind when creating the Robot Environment, knowing that in order for the inheritance structure to work we must fill the above three variables.

Also there is the need in this environment to initialize, define but not implement, some mandatory functions for the structure to work, Fig 4.21. These function definitions need to be implemented in the Robot or the Task Environment or on the contrary they will raise a `NotImplementedError()`. The functions are the following:

- `_set_init_pose()`
- `_check_all_systems_ready()`
- `_get_obs()`
- `_init_env_variables()`
- `_set_action()`
- `_is_done()`
- `_compute_reward()`
- `_env_setup()`

These functions are briefly described in the code and will be further analysed on them implementation.

Besides all the functions commented above there also is the function `_reset_sim()`. Even though this function is implemented, it can also be overwritten in any of the other Environments. What is implemented in this environment is how the simulation resets by default but there is the option of modifying the behaviour of how the simulation resets.

Even though it has been said that this Gazebo Environment is unique and it will not need any modification depending on the application, in some cases there is an exception, precisely the Fetch Robot case. The regular Gazebo environment that has just been reviewed is made to work with one goal, that will remain the same for the whole training process. Nevertheless, there are some training algorithms that require the modification of the goal during the training, for instance the HER algorithm. So for those cases the Gazebo Environment will be slightly modified to switch the environment it inherits from. Instead of inheriting from gym.Env it will inherit from gym.GoalEnv. It will be the only modification needed to switch from a regular Gazebo Environment to a Gazebo Goal(based)-Environment.

6.4.2. Robot Environment

The Robot Environment contains all the functions associated with a certain robot, containing all the functionalities that the specific robot needs to be controlled and serving as an interface to communicate with the Fetch Robot. Normally there can be distinguished two parts in the Robot Environment: The initialization of the class and the specific methods needed by the robot. A template for start building the Robot Environment can be found in [60]

The first thing to take into account is that the class that is going to be initialized inherits from the Gazebo Environment, in our case, from the Goal-based version of the Gazebo Environment. The fact that this Robot Environment class inherits from the Gazebo Environment simply means that all the functions defined in the Gazebo Environment are also accessible from this class.

In the initialization of the class section there are also defined all the ROS Publisher and Subscribers required for this specific robot.

Firstly, what is done in the initialization is to create a subscriber for the '/joints_state' topic. Also a variable with the `JointState( )` type is created to store the value of the different joints of the robots. Then there are defined a series of Service Clients.

Each Service Client will communicate with their corresponding Service Server. These are custom Services that will be created in a support script in order to send the trajectory information for manipulation to the robotic arm and also to get information from the End Effector of the arm.

Then the information about the controllers and the robot namespace that the Gazebo Environment needs is defined in blank. The reasons for it will be further explained but basically, in order for the whole system to work there must be two independent modules sharing information instead of one module managing all the processes. One module will be managing the Learning structure and the other one the simulation structure. This way ROS processes can be separated from the Learning algorithm methods.

The second main part is the definition of the function that will provide the methods needed by the Robot Environment.

Just for clarification, if the Robot Environment template is taken as a reference, it can be easily checked that the only mandatory method that the robot environment needs to define for the Gazebo Environment is the function `_check_all_systems_ready()`. The objective of that function is to check that all the joints are at their initial state value before starting each step. Basically this function checks that everything is working as it should.

This function is not included as a part of the methods for the Robot Environment but for the Gazebo Environment, strictly speaking. Nevertheless the only thing that needs to be done in

that function is to call the function `_check_all_sensor_ready()`, function that actually is part of the methods for the Robot Environment. This is just in order to maintain the common structure of the environments.

Focusing now on the function `_check_all_sensors_ready( )`, what it is doing is calling the following function `_check_joint_states_ready( )`, which finally is the relevant one. This last function is in charge of checking that the `/joint_states` topic is being published correctly.

Is vital to make sure that this topic is being published due to is the one that will provide the system with the data about the state of the joints at every moment.

Then it can be found the function `joints_callback( )`, it is just the callback for the Subscriber that was declared in the initialization of the class (`self.joint_states_sub`). It just stores the data from the `/joint_states` topic into the variable `self.joints`.

The function `get_joints( )` is another very basic function which labour is to simply return the self.joints variable.

A highly relevant function in this environment is the set_trajectory_ee(self, action). This precise function is the one in charge of sending the trajectories to the manipulation arm. As it can be seen, this function is getting as an input a variable named action. The action variable will contain the desired position coordinates (x,y,z) of the end effector of the manipulator. Once the desired position is received and filled into the corresponding variable, it is sent through the Service Client that is defined in the initialization of the class (`self.ee_traj_client`). Executing this function will activate the Service Server that is in charge of executing the trajectory in the simulated robot .

Similar to the previous function is the `set_trajectory_joints(self, initial_qpos)` function. It also is in charge of sending a trajectory in order to be executed by the manipulator arm, but in this case the objective is to achieve a set of joints positions instead of the desired end effector position. The desired joint positions are filled into the function in the variable `initial_qpos`.

Finally, there are only two functions left to review, concerning the particular method needed for the Fetch Robot Environment. These functions are `get_ee_pose()` and `get_ee_rpy()`

and are used to retrieve information about the position(x,y,z) and orientation(roll, pitch, yaw) of the end effector. To achieve that work they use the Service Client defined also in the initialization of the class, `self.ee_pose_client` and `self.ee_rpy_client`. These are custom Services that will be analysed in the following chapter.

Then the last lines of the Robot environment script are reserved for, as said in the template, methods that the task environment will need to define, and there is nothing to modify there. So gathering all the modifications performed in the Robot Environment template, the final Fetch Environment can be seen in [60]

6.4.3. Support Script for the Manipulator

The main reason for this support script is to avoid raising errors due to Python versions incompatibility. On the one hand ROS only officially supports Python 2.7. On the other hand, the objective is to train the Fetch robot using the HER algorithm that runs in Python 3.5 or superior, so the learning script will be launched from a Python 3.5 virtual environment. So since Python 3.5 lacks some basic ROS modules that will allow the interaction with the Moveit API there is the need of separating all the functions that will require interactions with those ROS modules from the environment structure.

Running those functions in a separate script, will make available the Services Clients needed from the environments, that will be running in Python 3.5. So from the environment structure calls will be performed to those functionalities that are actually in a separate script, that will be executed outside the Python 3.5 environment, avoiding any compatibility problem.

This fact is very important to bear in mind while working with Openai-ROS. This technique of separating the training algorithm and the environment structure from the code that will need ROS modules actually allows the user to work with most of the Deep Reinforcement Learning algorithm of the state of the art, that runs in Python 3.5 or superior. The support script can be seen in [60].

Firstly, focusing on the initialization of the class, it can be seen that the moveit_commander is initialized allowing the communication with the Moveit Rviz interface. Other objects are

also initialized, as the MoveGroupCommander, an interface to the manipulator group of joints to control and also a ROS publisher for the trajectories is created.

Then, below, there are being defined the Services that will be used to send motions to the manipulator:

- `ee_traj_srv`
- `joint_traj_srv`
- `ee_pose_srv`
- `ee_rpy_srv`

Just after defining the Services, there are defined the callback functions for each of the Services:

- `ee_traj_callback(self, request)`: Is in charge of computing the trajectories that are based on the end effector target position. The function receives a desired end-effector position in the request variable, that comes from the Service Client call, that can be localized in the Robot Environment inside the function `set_trajectory_ee(self, action)`. This function will compute the best motion plan to reach the desired position and then execute that plan by calling the `execute_trajectory( )` function.
- `joint_traj_callback(self, request)`: Analogously as it was done in the Robot Environment, this function is quite similar to the previous one but instead of working with the desired position of the end effector, this function takes as an input in the request variable a set of specific joint positions. So in the same way as the previous function, it computes the plan to obtain the desired joints position and then executes the trajectory by calling `execute_trajectory( )`. This function is also called in the Robot Environment by the function `set_trajectory_joints(self, initial_qpos)`:
- `ee_pose_callback(self, request)` // `ee_rpy_callback(self, request)` : This function will return the location(cartesian coordinates) and orientation(roll, pitch, yaw) of the end-effector. Their corresponding related functions in the Robot Environment are `get_ee_pose(self)` and `get_ee_rpy(self)`.

Apart from the call-back functions there is left the function `execute_trajectory(_)` that simply executes the trajectory that has been computed by the Moveit module.

Also at the end of the script there is defined an initialization of the ROS node that will hold the Services and it will be maintained open.

For all this to work a new package needs to be defined in the catkin directory, then paste into the src folder the support script `execute_trajectory.py` and also create a srv folder. Inside that folder fours service files (.srv) have to be created, one for each service we want to define. This way we can create and compile the 4 Services messages for our 4 custom Services. The service files that need to be defined in the srv folder are the following:

- EePose.srv:
```
---
geometry_msgs/Pose pose
```
- EERpy.srv:
```
---
float32 r
float32 p
float32 y
```
- EeTraj.srv:
```
geometry_msgs/Pose pose
---
bool success
string message
```
- JointTraj.srv:
```
trajectory_msgs/JointTrajectoryPoint point
---
bool success
string message
```

After creating these files, the only thing left to do is to perform the corresponding modification to the CMakeLists.txtx and the package.xml to be able to compile the package by performing a catkin_make. Now this support package would be ready to use.

6.4.4. Task Environment

The Task Environment is the highest one in the inheritance structure which also means it is the most specific one and fully depends on the specific task that the robot has to perform. All the related methods that the robot will need to learn how to perform the specific task are defined here.

In this example, the task that is going to be implemented for the Fetch Robot in this Task Environment is the Reach task. The basic goal of this task is to be able to point (and reach)

with the end-effector of the manipulator arm to one cube placed in the table the robot is in front of.

The explanation will also be based in the Task Environment template that can be found in [60].

The first thing that is done in this script is to register the specific task into the OpenAI registry, this must be done previously in order to be able to use the gym environment. There are three variables to define in the registry. The `id` variable is a label for the training and always must follow the format label_for_task-v# indicating the version number after the v. The `entry_point` indicates the Python class that is used to start the training. The `timestep_limit` variable most likely is self explanatory but to avoid confusion, it indicates the maximum number of time steps that are allowed before the episode finishes.

Then the class for the Task Environment is defined. As has been done in the Robot Environment, the environment can be split into the initialization part and the functions that implement the methods dedicated to the specific task.

In the initialization function, what must be done first is to call the `get_params()` function that will retrieve all the required parameters that the Fetch robot requires to start learning. Then the Fetch Environment (Robot Environment) is initialized and right after the function `_env_setup( )` is called. This function basically sets up everything needed to start training and send the information to the manipulator to adopt its initial position, filled in the `initial_qpos` variable.

Then, in order to generate the observation space the function `_get_obs( )` is called and finally the observation as well as the action space are defined. The action space is a Box space (`space.Box`) variable, imported from the `gym` module, the values are between -1 and 1 and the shape of the Box variable is stated by the param `n_actions`. The observation space is a dictionary that is specifically designed to be aligned with a Goal-based environment.

Continuing with the task environment, now focusing on the specific methods needed by the Task Environment, it is very important to understand that even though any function can be added if it is required, there are some functions, that are also initialized in both of the previous reviewed environments, that will always need to be denied in the task environment.

Since those functions are required by the most basic environment (`gym.Env`) they need to be defined even if they are not needed, just to avoid raising a `NotImplementedError()`. Nevertheless, they just need to be present in the code, for some of them there is not any specific variable they need to return and the user can basically make them pass. Here is the list of the functions that must be always present in the code:

- `_set_init_pose()`
- `_check_all_systems_ready()`
- `_get_obs()`
- `_init_env_variables()`
- `_set_action()`
- `_is_done()`
- `_compute_reward()`
- `_env_setup()`

Once this has been said the attention can be focused on the particular method implemented for this training.

The first function found is the `get_params()` function which basically load the required variables for the training. These variables can be defined in the script as it is done in this one or the value of the variable can also be read from the ROS system if previously those variables have been launched from a configuration YAML file. Several aspects of the task and training are reflected here as variables such as the length of the action vector, in this case `n_actions` = 4 which states the position of the end effector and the state of the gripper. Other variables like the initial pose of the robot (`init_pose`), `reward_type`, `distance_threshold` or `block_gripper` are also defined in the function `get_params()`. In this case the gripper of the robot will be blocked, since the objective of the task is simply to reach an object. Besides, another variable that notably influences the overall algorithm is `has_object`, it is a Boolean variable that states if the algorithm takes into account task related objects or not. Since the objective for this training is for the robot to learn how to reach different 3D spots it makes no difference to define an episode goal as a virtual 3D location or as an object position. Therefore, in this case `has_object = False`.

The function `_set_action()` receives an action variable as input. What the function does is simply reformat the action variable and send it through the

`set_trajectory_ee(action)` function, the custom Service reviewed previously which is defined in the support script, in order to execute the motion, this function only takes as an input the pose of the end effector but no the rotation. The rotation of the gripper is already defined in this function as a fixed rotation [1, 0, 1, 0] in quaternions.

About the function `_get_obs( )`, it is really intuitive what this function is doing. It uses the Services from the support script to retrieve the position of the end-effector and information related to the object in the scene if it is taken into account in the training, in this case the information related to the object is filled as a zeros array. This function does not also retrieve the observation array but also the achieved goal and the desired goal variables, the related functions to those variables are defined in this script below.

Continue reviewing the mandatory function to define in the task environment it can be found the `_is_done(self, observation)`. All these functions perform such an intuitive job if the user is familiarized with reinforcement learning training. Concretely this function makes use of the function `goal_distance()` to compute how far the actual goal is from the desired one. Since the reward has been defined as sparse in the function `get_params()`, if this distance is small enough, smaller than a threshold, the function will return a one otherwise it will return a cero.

Another mandatory function is the `_compute_reward(self, observation, done)` function. It calculates the reward for each step, in other word, each time an action is taken. The reward is based on the distance between the end-effector and the cube and, as previously said, in this occasion it is defined as sparse.

The last mandatory function left to review is the function `_set_init_pose(self)` is in charge of putting the manipulator of the Fetch in the initial position after each episode of the training finishes. It uses the custom Service associated with the function `set_trajectory_joints()` to send the information of the variable `self.init_pos`.

As a support function needed for this task and in order to define the goal to be reached and also the actual goal that has been reached the functions `_sample_goal(self)` and `_sample_achived_goal(self, grip_pos_array, object_pos)` are respectively used. Since in the learning phase there is no objects taken into account the initial goal for the

episode is sampled as the position of the gripper plus a random offset sample from a uniform distribution with values between -0.15 and 0.15. Thus the goal for the episode would be within a 30 centrimeters edge cube centred in the gripper tip. The achieved goal is obviously the current position of the gripper.

There also is the function `_env_setup(self, initial_qpos)` that is called in the initialization of this environment class and basically, as its name implies, sets up the environment so that everything is ready for the training to start, send the manipulator to its corresponding position, sample the goal for the episode and get the first observation.

Finally the last function is `robot_get_obs(data)` which reformats the data coming from `self.joints` into joint positions and velocities associated with the Fetch robot joints.

So gathering all the functions and modifications performed to the Task Environment template there the `fetch_reach.py` environment, see [60].

6.4.5. Training Script

There must be beared in mind that the training script is completely independent from the environment that defines the problem. In the case of this project the OpenaAI baselines [61] are going to be used. OpenAI baselines is a set of high-quality implementation of RL algorithms, created to be used with OpenAI Gym [36]. In order to use these algorithms with the OpenAI-ROS interface some slight implementations, such as initializing a ros node, has to be made but the learning algorithm is left untouched. The final training algorithm that is going to be used for the simulation can be seen in [60]. The training is going to be carried out using the HER algorithm with a DDPG agent.

Besides, a new fork of OpenAI Baselines has been released recently with the name of Stable-Baselines [62], it is a set of improved implementations of RL algorithms by far more intuitive and simple to use than the original baselines.

In this version of the baselines there are additional algorithms supported by the HER method, like Soft Actor Critic (SAC) and Twin Delayed Deep Deterministic Policy Gradient (TD3), besides, the HER algorithm code was rewritten from scratch.

6.5. Connecting AI control to Real Moveo Robot

The Moveo is still in the assembling phase. The electronic setup of the controllers TB6560 has been tested using Pronterface, a really basic controller software with an intuitive interface, to control the base motor of the Moveo. The controllers work properly and the motors spin correctly, nevertheless, due to the Moveo size and weight, there can be slip between the motor shaft and the pulley. That can be critically harmful for the control on the Moveo so to have a solid assembly for the pulleys must be glued to the motor shafts. Once the mechanical and electronic assembly is completed the hardware-software bridge between the Moveo controller and ROS has to be implemented in order to have real-time communication between the simulation and the real robot. Basically the objective here is to create a rostopic that hosts the information to be sent/received, and both, the simulation and the Arduino firmware can publish or subscribe to those topics in real time. A precise guide can be found in [63].

6.6. Connecting AI control to the Moveo Simulation

To launch a functional Moveo model in the Gazebo simulation, different files have to be managed. Firstly, the URDF is launched in a XACRO file containing the information about the link (parts) and joints and the relation between them. Within that information the dynamics of the model are included, the inertia mass of the elements and friction parameters.

As a preliminary work, this simulation is going to be launched without the end effector and the top part of the wrist articulation that is adapted for the end effector set up. This decision simplifies the work avoiding the complexity of the current end effector taking into account that for medical research the end effector should be remodeled.

Moveo claims to be a 6 DoF arm by conferring one DoF to its end effector, so by removing the second joint of the wrist left us with a 4 DoF robot. Nevertheless, adding new links and joints to the model in the future is a trivial task.

Each DoF of the robot should have an associated transmission code related to the joint, concretely a Position Joint Interface has been chosen as hardware interface in the transmission definition. For those transmission to communicate with ROS a plugin has to be

defined, in this case a basic gazebo_ros_control has been used. In the plugin definition a namespace of the robot needs to be stated clearly as this name is crucial for the controllers set up.

Once all the information is gathered in the URDF the only thing left to do to have the robot model in gazebo is to create a launch file. In the launch file of the robot, the controllers need to be defined by loading a YAML file in the ros parameter server and then loaded when the URDF spawns by using the package controller manager.

In our case, where the objective is to use Moveit, a group controller with the 4 DoF of the robot is defined, instead of defining one controller per joint. This information is stated in the moveo_controller YAML file that is launched in the generation of the Moveo model.

The launch file created to generate the moveo model has been called moveo.launch and the following is obtained when the file is launched, Fig. 3.15.

Take into account that a gazebo world was already initialized using *rosrun gazebo_ros gazebo*. Remember that the catkin directory where the packages that are going to be used are allocated always need to be sourced to ROS by running source/devel/setup.bash in the command shell being in the catking directory. And this has to be done in every shell that is used to launch files allocated in the catkin directory as well as the shell used to launch Gazebo.

Notice in Fig. 6.4 how the controllers are loaded properly and there have been created the topics related to the moveo/joint_trajectory_controller.

Then following the Moveit documentation [64] a Moveo moveit configuration package is created.

The only thing left to do is to create the Gazebo Moveit interface. There are different version of building the interface and plenty of documentation can be found online, in this case the guide [51] has been strictly followed. In order for the Moveit package to recognize the gazebo controllers in the ROS system and build the corresponding action server for the controller's topics it has to recognize the topics first. Therefore the *controllers.yaml* file in the Config folder of the Moveit package the controllers has to be defined so the Gazebo controllers are identified, as it is shown in Table 4.1.

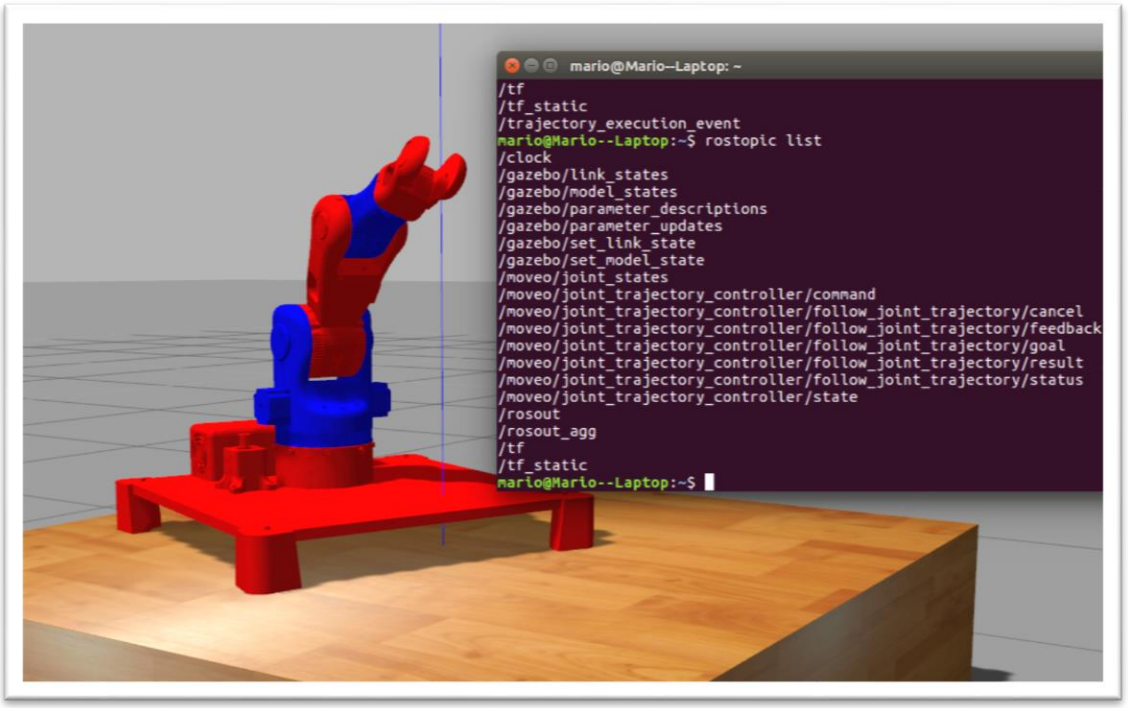


Figure 6.4: Preliminary Moveo model with controllers in Gazebo, showing the ROS topic list in the bottom terminal.

Then the corresponding launch files for the Moveit package are created, concretely by executing my_moveo_planning_execution.launch. When this file is launched the Moveit Gazebo Interface should be created properly.

Table 6.1 YAML files where the information about the controllers is located. a) File used to launch the controllers in the gazebo simulation. Constraints and PID gains has been omitted. b) File used by the Moveit package to identify the controllers and created the Gazebo-Moveit interface.

a) my_moveo_description/config/moveo_controllers.yaml

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

# Trajectory Controllers ---------------------------------
  joint_trajectory_controller:
    type: position_controllers/JointTrajectoryController
    joints:
      - Joint_1
      - Joint_2
      - Joint_3
      - Joint_4
```

b) my_moveo/my_moveo_moveit_config/config/controllers.yamls

```
controller_list:
  - name: moveo/joint_trajectory_controller
    action_ns: "follow_joint_trajectory"
    type: FollowJointTrajectory
    joints: [Joint_1, Joint_2, Joint_3, Joint_4]
```

6.7. Training a Robotic Controller with OpenAI-ROS

Since a fully operational model of the Moveo robotic arm is not available yet, due to pending designs for the end-effector, the Fetch robot is going to be used fot testing the OpenaAI-ROS package.

Fetch OpenaAI-ROS interface environments created in previous sections are built to work with the existing Gazebo Fetch model, for this work the simulation package for the Fetch model has been downloaded from The Construct public repository [59] as the folder `fetch_tc`. Actually there is a collection of packages that integrates the Fetch simulation package as a whole. It comes with all the requirements needed to perform tasks with the robotic arm, so the Moveit package is already built.

The Fetch simulation is launched using the file main.launch or, for our case, as the Fetch robot has been already used for research in manipulation tasks, there is another launch file for setting up the Fetch robot for such tasks. So the command that will be used is put_robot_in_world_HER.launch. Basically the robot is launched in the exact way but the standing pose is changed in order to not collapse with the elements needed to set up the task at the beginning of the episode.

Then the elements for the task have to be loaded into the simulation. Since the task that is going to be learned is the Reach task there only needs to be one table and one object to reach in the simulation.

It is crucial to bear in mind that all the shell used to launch files need to previously source the catkin directory to the ROS package directory by using the *source devel/setup.bash* in the shell being in the catking directory.

Once the Fetch model has been launched properly the Moveit package can be launched as well as the support script for the manipulator defined in the section 4.6.4. Finally, before the training script is launched, in order to avoid interface errors, it is recommended to run these two commands *export LC_ALL=C.UTF-8* and *export LANG=C.UTF-8.*

At this point everything is set up for learning the task. As can be seen in the Fig. 6.5, the Fetch model has been launched properly and the moveit motion planning is properly integrated with the Gazebo simulator. However, there is an ascect that must be discussed. Since we are going to be using OpenAI baselines for the training, the training script must be launched from a Python 3 virtual environment, since it is the baselines requirement. Therefore, taking into account that ROS only officially supports Python 2.7, some problems can emerge. This fact is the main reason for the creation of the support script from section 6.4.3, to separate any ROS process from the learning modules that need to run with Python 3. So in the code executed with Python 3 there only will be Service Clients which will perform calls to the functionalities that are actually in a different script running in Python 2.7. As can be seen, when the simulation is being set up, the support script execute_trajectories.py is executed outside any virtual environment.

Meanwhile, the execution of the training script is done from a virtual environment, in our case named *her_python*, where all the required modules such as tensorflow are installed. The training script has to be located in the baselines/baselines/her folder, that is downloaded from [61].

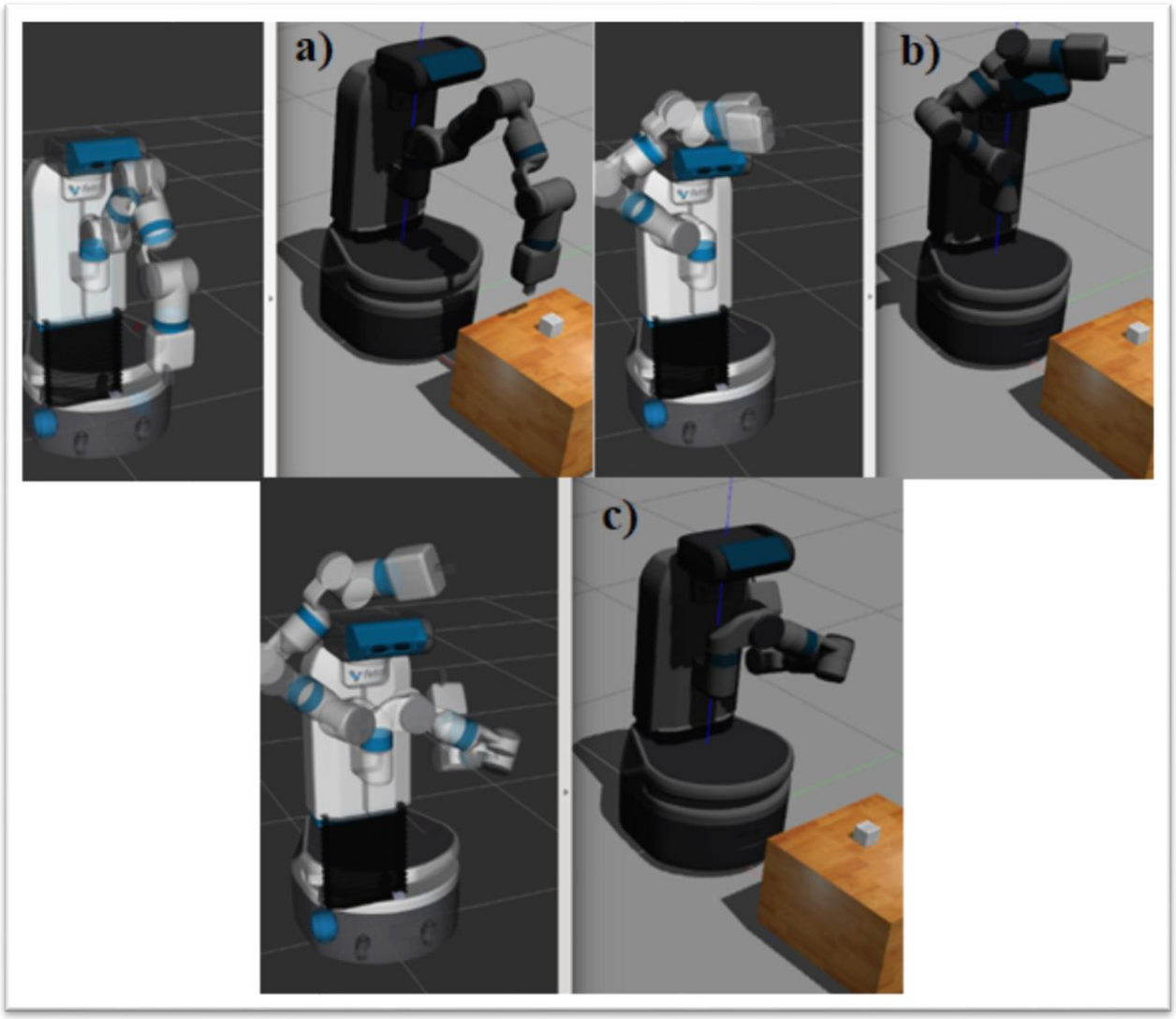


Figure 6.5. Sequence a, b, c of random motions planned with Moveit and executed in the Gazebo simulator. In the left side of each picture is the Rviz visualization for Moveit, in the right side is the Gazebo model performing the same motion.

7. Chapter 7: Results and Discussion

All the work gathered in this document has been performed in Ubuntu 16.04 as Operating System, ROS Kinetic, the corresponding ROS version for Ubuntu 16.04, and Gazebo 7.

7.1. Moveo Robot Mechanical Assembly

The Moveo Robot is not functional yet due to the restricted entry to the Universitat Pompeu Fabra Tanger building, measure taken in consideration of the COVID19 pandemic.

Nevertheless, there are certains contribution to the Moveo assembly worth to highlight.

Firstly, concerning the joints locomotion system of the robot. The rotational joints should work properly once the belt tensioners are setted up but the torsional joint of the wrist, section 4.4, was completely limp once it was mounted following the original assembly guide. This is an issue concerning geometry tolenrances between the slot in the moveo part and the locknut used to use to create relative motion between the both links of the joint. The locknut is supposed to perfectly fit in the slot but instead the nut has enough room to spin inside the slot, therefore the motion would not be correctly tranfered along the manipulator. The proposed solution is to used time-hardening mastic around the locknut to ensure it tighly fits in the slot of the Moveo part.

Secondly, we have tested the controller's setup for the base motor following the singleline schema showed in the Appendix and Fig. 7.1 for the connection between our concrete motors and controllers. Once everything was set up we used the Promterface software for sending pulses to the motor. Since the base joint, has no belt tensioner, it is important that the motor shell is screwed to the table once the motion mechanism is setted up and the belt is fully tensed or there can the chance that the belt tooths slip around the pulley and the motion is not correctly transferred. The test was performed with a 3D printed pulley fixed to the motor shaft by interference fitting. These kind of pulleys should also be avoided since the geometry can be damaged during the motion. Also during the test, slip between the motor shaft and the pulley was noticed in certain situations. Therefore, interference fits are not enough to ensure the well functioning of the locomotion system, so the pulleys must be glued to motors shafts.

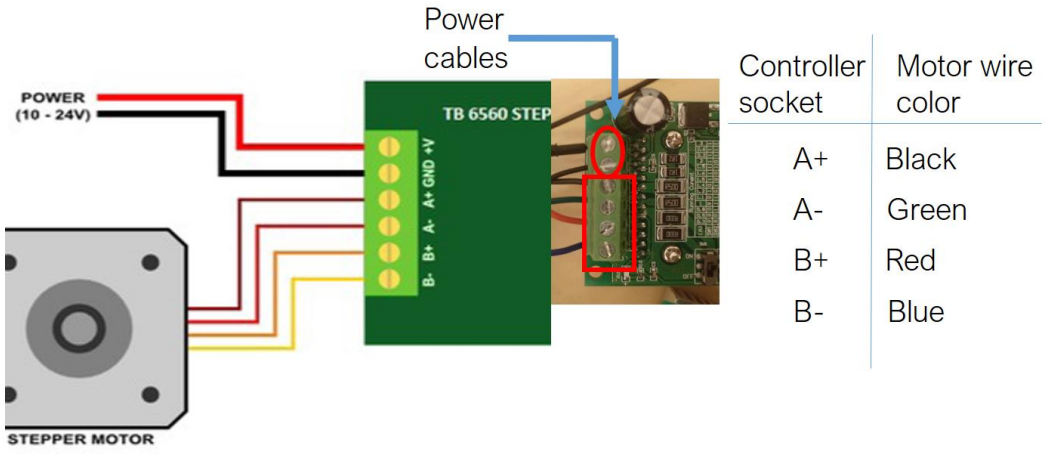


Figure 7.1. Details regading the connection between stepper motors and controllers

7.2. Moveo Robot Simulation

As stated in the section 6.6, by executing the command moveo.launch using roslaunch, the Moveo model is correctly generated in the Gazebo simulator, just as has been shown in Fig. 6.4. Then everything is ready for the Moveo Moveit package to be launched by executing the file moveo_planning_execution.launch. When it is done the Moveit-Rviz visual interface will start, allowing us to perform basic motions with Moveo using the Moveit motion planning engine Fig. 7.3. To check that everything is working properly the shell from which the Moveit package has been launched can be watched and the message *Trajectory execution is managing controllers* will be plotted.

Moreover, Moveit not only provides a GUI for performing motion plans but it also can be managed from Python scripts. There are various functionalities that can be employed in different set ups, everything is gathered in the Moveit Move Groups Python Interface documentation [65]. The most basic functionalities to move the robot are, firstly, to move the end effector to a certain pose and secondly, the set the joint of the robot in a certain position. For our case, since there is no end-effector defined yet in the model, the second choice is the most useful for us to set the robotic arm in its initial position once the model is generated. Taking into account that Fig. 7.3.a) is the position where the model is generated, what means that all the joints are equal to zero, an initial position similar to Fig. 7.2 is achieved setting the joint at the following values:

Base joint = 0, Shoulder Joint = $\pi/4$, Elbow Joint = $\pi/16$, Torsional wrist joint = $-\pi/8$, (radians).

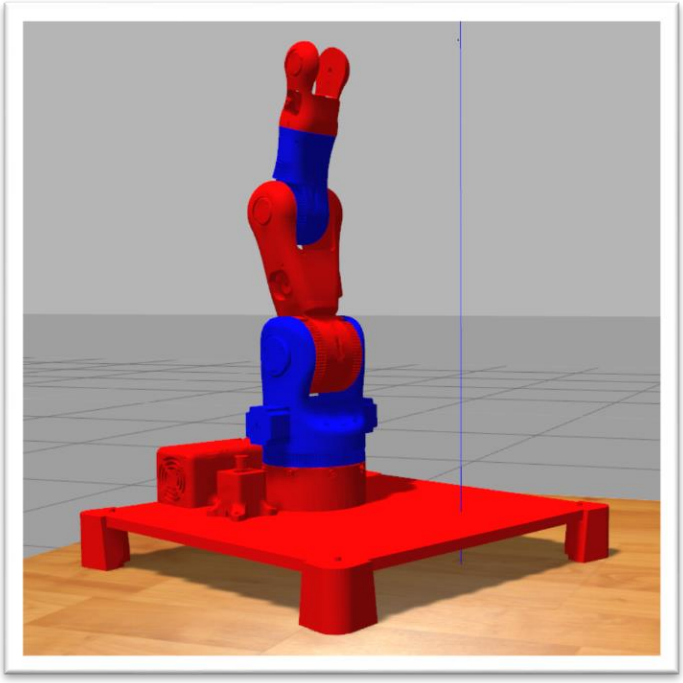


Figure 7.2: Moveo model in gazebo after performing the motion to set the initial pose. Command sent by python script.

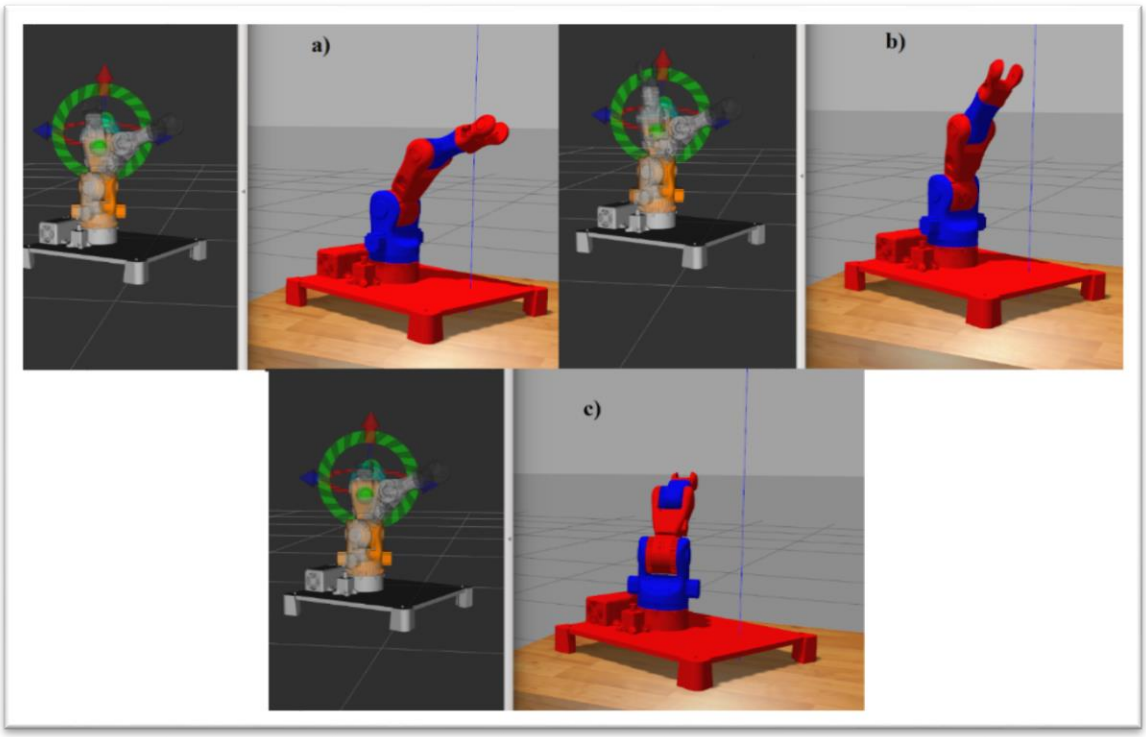


Figure 7.3: Executing motion through Moveit. a) Initial pose b) Intermediate pose c) Final pose

Even though creating a series of files for defining a robot model, simulating it and generating the corresponding controllers, by following a proper documentation, seems easy, it has to be carried out carefully. ROS is a fragile and not very informative system when it comes to handling errors. Moreover, as open source and state of the art software, there are continuous updates to the libraries and the methods get improved constantly so the documentation online can be confusing sometimes.

For our concrete case, such a simple task as building an interface between the Gazebo simulator and Moveit has been a hard one. The main problem is that the Moveit API was not able to find the corresponding controllers of the Gazebo simulation so the Action Server for passing the joint states was never created. Mainly it had to do with the namespaces, of how the controllers are defined in the YAML parameters folder but also in the Launch files. Finally, we have provided a working structure that can be improved easily when the new parts for the end effector are designed.

7.3. Fetch Training Simulation using Openai-ROS

Two different HER implementation algorithms are tested for the training of the Fetch Robot, the algorithm provided by OpenAI in the Baselines package [61] and the algorithm provided by a new fork of the Baselines package named Stable-Baselines.

7.3.1. OpenAI Baselines

After setting up the simulation, section 6.7, the training script is launched setting DDPG as an off-policy training algorithm and the first episode of training begins. It can be seen in the simulator how the robot performs the motion to set the initial pose of the manipulator, Fig. 7.4, given as a parameters of the function `get_params_()` to read, `self.init_pos = {'joint0': 0.0, 'joint1': 0.0, 'joint2': 0.0, 'joint3': -1.5, 'joint4': 0.0,'joint5': 1.5, 'joint6': 0.0,}`.

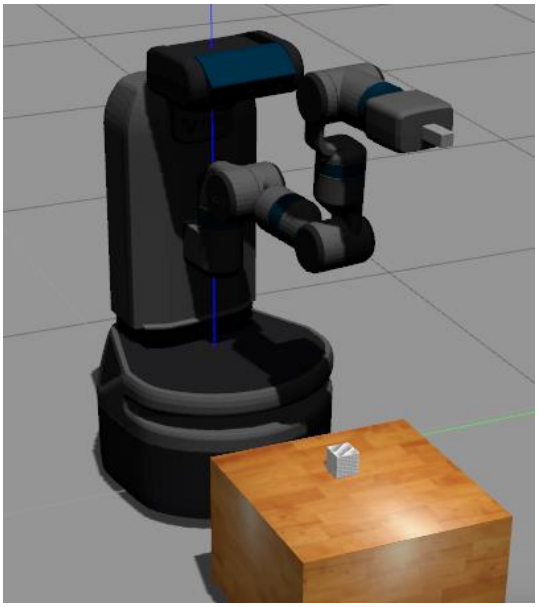


Figure 7.4. Fetch robot set to its initial position at the beginning of the episode.

The shell from where the training script has been executed prompts all the parameters and information related to the training. After the training starts and the first exploration motion is proposed before the algorithm crashes due to an Attribute error associated with the function `venv.reset()`, where the attribute reset is not recognized, in the RolloutWorker.

Trying to modify or refine the code has proven to be useless. The source of the error seems to be attributed to a mismatch between the current version and the version of the HER baselines implemented in train.py. This mismatch makes the RolloutWorker function to take as an argument the function that makes the environment instead of an environment itself. However, any modification to the code leads to errors, the errors found have been already reported in [66].

Even though we have not been able to properly train the Fetch robot unsing the HER OpenAI baselines algorithm, the Fetch environment seems to be correctly generated. When the training starts, the Fetch robot properly executes the motion to set the initial pose and the first step of the training is executed, an action is performed and the observation is received. All those behaviours are performed without error which gives us the hint that the environments structure that set up the learning problem is well formed and fully functional.

Although the training has not been possible due to the learning algorithm issue; we should pay attention a warning message plotted in the shell where the training script is launched:

```
                        *** Warning ***
You are running HER with just a single MPI worker. This will work,
but the experiments that we report in Plappert et al. (2018,
https://arxiv.org/abs/1802.09464) were obtained with --num_cpu 19. This
makes a significant difference and if you are looking to reproduce
those results, be aware of this. Please also refer to
https://github.com/openai/baselines/issues/314 for further details.
```

This means that even though we were able to make the algorithm work, a high accuracy performance would have been hard to reproduce using only one MPI for the training.

7.3.2. Stable-Baselines

Following the same methodology as with baselines, we created the training script by following the documentation [67], allocate it in the corresponding folder inside the Stable-Baselines package and execute it from a virtual python environment when the whole simulation is set up.

Firstly, the algorithm is tested using the DDPG for the approximator networks, however an error is found as well, therefore the SAC method is tested.

This method introduces more sophisticated behaviours and concepts that have not been reviewed in this project. When the training script is executed the following error is found:

```
File"/home/mario/catkin_ws/stable-
baselines/stable_baselines/common/base_class.py", line 1006, in
replay_buffer_add
self.replay_buffer.add(obs_t, action, reward, obs_tp1,
float(done),**kwargs)
TypeError: add() missing 1 required positional argument: 'info'
```

In this case the algorithm crashes but seems to be due to an environment dependent method. It is known that the HER algorithm uses the replay buffer to store episode information such as the trajectory and reward but in this case the function that stores the information in the replay buffer also ask for an `info` variable, referenciated in the code as a dictionary variable with extra values used to compute the reward when using HER.

So, in search of a quick solution, knowing that there is no function in our environment to compute the variable `info`, a slight modification is implemented in the code of the script `base_class.py`, so an empty info variable is filled to the corresponding function. This modification is performed as a preliminary solution.

With the modified version of the Stable-Baselines HER algorithm and using SAC as off-policy learning method, the training of Fetch robot does not hold due to errors right away, and the first episode of the learning takes place, Fig 7.5 Even though it seems to be working properly during the first episode of the training, sporadic errors still occur, related to the replay wrapper, when a step of the episode is performed:

```
ValueError: Cannot feed value of shape (1, 13) for Tensor
'input/input/Ob:0', which has shape '(?, 9)'
```

The second episode of the training is never reached.

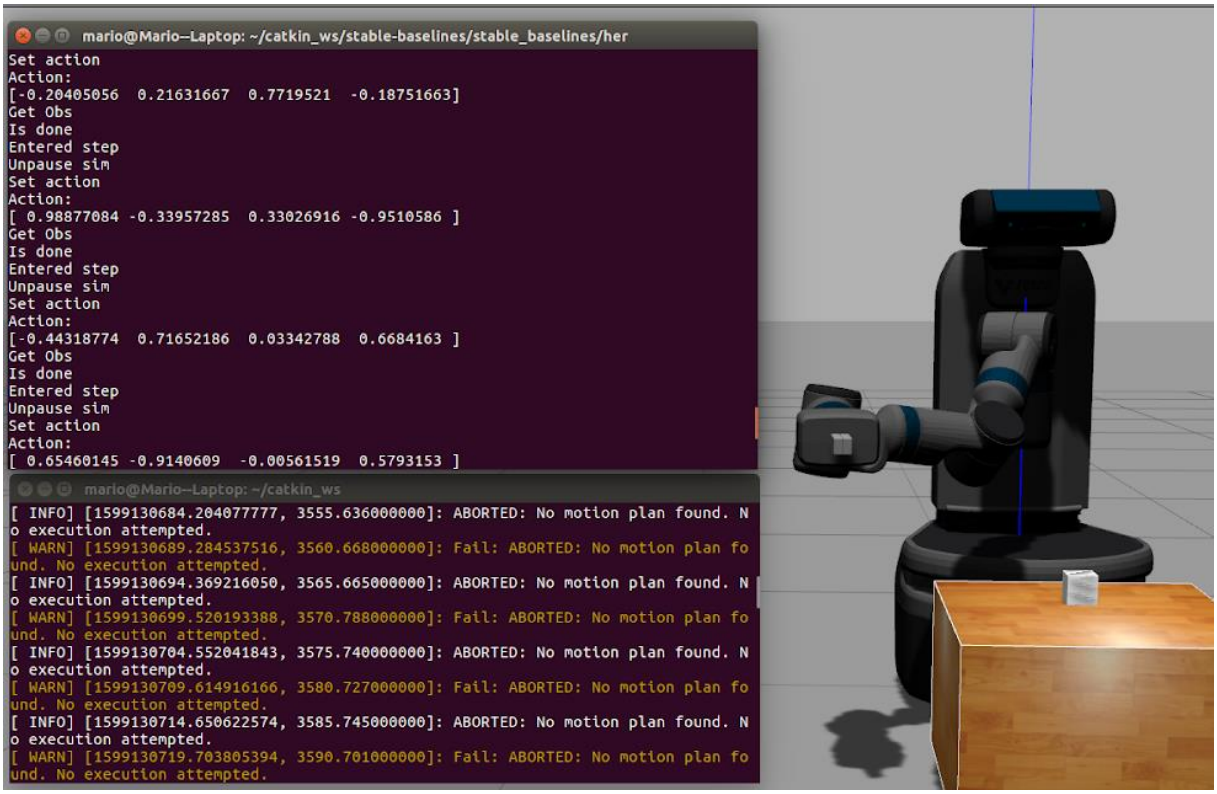


Figure 7.5. First episode of the Fetch training using Stable-Baselines algorithm

We must bear in mind that our custom Fetch environment has been built to work with a certain version of OpenAI-Baselines. Therefore, there might be the need of performing certain improvements to to the environments in order to make it useful with Stable-Baselines,

where new functionalities might be needed since new off-policy training algorithm has been implemented, such as a function to compute the variable info needed by the SAC algorithm. Regarding the error found during the training with Stable-baselines HER-SAC algorithm, it is a value error that must be investigated in order to elucidate if it is an algorithm related issue or an environment one.

OpenAI Gym and OpenaAI baselines are packages in development stage and that suffer with continuous updates for bug fixing, and most important that are designed to work with MuJoCo. Thus there is the chance that the base structure of the environment has been modified to work better with the training algorithms, putting our environment structure outdated.

Nevertheless, OpenAI-ROS interface is such a flexible structure that, with the corresponding adaptations, should be able to work properly and allow robot training once the optimal algorithm is found.

8. Chapter 8: Conclusions and Future works

The main future goal of controlling the Moveo Robot with reinforcement learning tecniques will be naturally fulfilled once the three experiments that takes place in this project are further developed and the issues found in each of them solved.

8.1. Real Moveo Robot

Since the work in University facilities has not been able to be carried out, future works regarding the real Moveo are close to what is proposed in section 4.8.1.

To ensure well functioning locomotion the pulleys must be glued to the transmission motor shafts of the motor, then the rest of the controllers TB6560 should be assembled following the setup of the already assembled controller of the base motor, alternatively, the controllers can be updated to pin drivers such as DRV8825. Also the drivers for the shoulder articulation motors should be tested to ensure that the drivers tolerate enough electric current for the motor locomotion.

Next the interface between the controllers and ROS should be built taking as reference [49]. Moreover, as has been said already, an application oriented end-effector should be designed in order to allow the use of Moveo in medical researches on surgery.

Furthermore, there is the chance that to be fully functional for research in medical applications the Moveo robotic arm would need certain improvements such as computer vision device in order to implement more sophisticated and accurate behaviours.

8.2. Moveo Simulation

Regarding the Moveo model in Gazebo, it should be updated as soon as the alternative end-effector is designed or any other modification is performed in the real Moveo robot. Once the new features are updated in the model, the only thing to do is to perform the corresponding modification to the controllers and generate a new Moveit package for the update version of Moveo.

8.3. Fetch training

There are two essential things to do as the next step to continue with this research. Firstly, investigate the training algorithm source of the error and discern if it is environment or baseline package related. Secondly, a proper hardware setup and software adaptation must be found in order to set the correct `--num_cpu` to be able to reproduce the result from [10].

Moreover, attending to the Fig.7.5, it can be seen that the actions taken by the policy are almost never performed due to the fact that no motion plan is found, so the actions taken by the policy are indicating not reachable points. Therefore, the environments can optimize the training by filtering the action proposed and avoiding feeding action that will lead to motion planning abortions.

Furthermore, once there is the guarantee that OpenaAI-ROS interface works, in order to be able to deploy a useful trained policy into the real robot, Domain Randomization [33] algorithms should be explored in order to bridge properly the gap between the real world and the simulation.

9. List of figures

10. References

[1] Z. Lu, A. Chauhan, F. Silva, and L. S. Lopes, "A brief survey of commercial robotic arms for research on manipulation," in *Proceedings - 2012 IEEE Symposium on Robotics and Applications, ISRA 2012*, 2012, pp. 986–991.

[2] "robots.ros.org." [Online]. Available: https://robots.ros.org/. [Accessed: 21-Aug-2020].

[3] W. P. Xu, W. Li, and L. G. Liu, "Skeleton-sectional structural analysis for 3D printing," *J. Comput. Sci. Technol.*, vol. 31, no. 3, pp. 439–449, 2016.

[4] "BCN3D MOVEO - A fully Open Source 3D printed robot arm - BCN3D Technologies." [Online]. Available: https://www.bcn3d.com/bcn3d-moveo-the-future-of-learning/. [Accessed: 21-Aug-2020].

[5] C. Christensen, "A Robotics Framework for Simulation and Control of a Robotic Arm for Use in Higher Education Recommended Citation 'A Robotics Framework for Simulation and Control of a Robotic Arm for Use in Higher Education,'" 2017.

[6] "Zortrax Robotic Arm by Zortrax – Zortrax Library." [Online]. Available: https://library.zortrax.com/project/zortrax-robotic-arm/. [Accessed: 21-Aug-2020].

[7] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[8] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation."

[9] V. R. Konda and J. N. Tsitsiklis, "Actor-Critic Algorithms."

[10] M. Andrychowicz *et al.*, "Hindsight Experience Replay."

[11] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[12] J.-A. M. Assael, N. Wahlström, T. B. Schön, and M. P. Deisenroth, "Data-Efficient Learning of Feedback Policies from Image Pixels using Deep Dynamical Models," Oct. 2015.

[13] S. Levine and V. Koltun, "Guided Policy Search," 2013.

[14] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-End Training of Deep Visuomotor Policies," 2016.

[15] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *Int. J. Rob. Res.*, vol. 37, no. 4–5, pp. 421–436, Apr. 2018.

[16] R. Featherstone and R. Featherstone, "Dynamics of Rigid Body Systems," in *Rigid Body Dynamics Algorithms*, Springer US, 2008, pp. 39–64.

[17] A. Gupta, C. Eppner, S. Levine, and P. Abbeel, "Learning Dexterous Manipulation for a Soft Robotic Hand from Human Demonstration," *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2016-November, pp. 3786–3793, Mar. 2016.

[18] G. Gilardi and I. Sharf, "Literature survey of contact dynamics modelling."

[19] Z. Pan, C. Park, and D. Manocha, "Robot Motion Planning for Pouring Liquids," 2016.

[20] "Abaqus Analysis User's Manual Abaqus 6.11 Analysis User's Manual."

[21] P. Christiano *et al.*, "Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model," Oct. 2016.
[22] B. Mettler, M. B. Tischler, and T. Kanade, "System Identification of Small-Size Unmanned Helicopter Dynamics 1," 1999.
[23] T. Ingebretsen, "System Identification of Unmanned Aerial Vehicles," Institutt for fysikk, 2012.
[24] E. Tzeng *et al.*, "Adapting Deep Visuomotor Representations with Weak Pairwise Constraints," Springer, Cham, 2020, pp. 688–703.
[25] Y. Ganin *et al.*, "Domain-Adversarial Training of Neural Networks," 2016.
[26] M. Wulfmeier, I. Posner, and P. Abbeel, "Mutual Alignment Transfer Learning," Jul. 2017.
[27] A. A. Rusu, M. Večerík, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-Real Robot Learning from Pixels with Progressive Nets," PMLR, Oct. 2017.
[28] B. C. Stadie, P. Abbeel, and I. Sutskever, "Third-Person Imitation Learning," Mar. 2017.
[29] T. Le Paine *et al.*, "One-Shot High-Fidelity Imitation: Training Large-Scale Deep Nets with RL," Oct. 2018.
[30] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *IEEE International Conference on Intelligent Robots and Systems*, 2017, vol. 2017-September, pp. 23–30.
[31] X. Bin Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2018, pp. 3803–3810.
[32] O. M. Andrychowicz *et al.*, "Learning dexterous in-hand manipulation," *Int. J. Rob. Res.*, vol. 39, no. 1, pp. 3–20, Jan. 2020.
[33] J. Tobin *et al.*, "Domain Randomization and Generative Models for Robotic Grasping," in *IEEE International Conference on Intelligent Robots and Systems*, 2018, pp. 3482–3489.
[34] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An evaluation platform for general agents," *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, Jun. 2013.
[35] Y. Duan, X. Chen, C. X. B. Edu, J. Schulman, P. Abbeel, and P. B. Edu, "Benchmarking Deep Reinforcement Learning for Continuous Control," 2016.
[36] G. Brockman *et al.*, "OpenAI Gym," Jun. 2016.
[37] "OpenAI." [Online]. Available: https://openai.com/. [Accessed: 21-Aug-2020].
[38] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, "Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo," Aug. 2016.
[39] M. Quigley, K. Conley, B. Gerkey, … J. F.-I. workshop on, and undefined 2009, "ROS: an open-source Robot Operating System," *willowgarage.com*.
[40] "openai_ros - ROS Wiki." [Online]. Available: http://wiki.ros.org/openai_ros. [Accessed: 21-Aug-2020].
[41] N. Gonzalez Lopez *et al.*, "gym-gazebo2, a toolkit for reinforcement learning using ROS 2 and Gazebo," 2019.
[42] W. Qian *et al.*, "Manipulation task simulation using ROS and Gazebo," in *2014 IEEE*

International Conference on Robotics and Biomimetics, IEEE ROBIO 2014, 2014, pp. 2594–2598.
[43] S. Chitta, "Moveit!: An introduction," *Stud. Comput. Intell.*, vol. 625, pp. 3–27, Feb. 2016.
[44] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *IEEE International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033.
[45] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004, vol. 3, pp. 2149–2154.
[46] "GitHub - BCN3D/BCN3D-Moveo: Open Source 3D Printed Robotic Arm for educational purposes." [Online]. Available: https://github.com/BCN3D/BCN3D-Moveo. [Accessed: 23-Aug-2020].
[47] "Build a Giant 3D Printed Robot Arm : 83 Steps (with Pictures) - Instructables." [Online]. Available: https://www.instructables.com/id/Build-a-Giant-3D-Printed-Robot-Arm/. [Accessed: 23-Aug-2020].
[48] "3D Hubs | On-demand Manufacturing: Quotes in Seconds, Parts in Days." [Online]. Available: https://www.3dhubs.com/. [Accessed: 23-Aug-2020].
[49] "Moveo with ROS — Jesse Weisberg." [Online]. Available: https://www.jesseweisberg.com/moveo-with-ros. [Accessed: 25-Aug-2020].
[50] "xacro - ROS Wiki." [Online]. Available: http://wiki.ros.org/xacro. [Accessed: 25-Aug-2020].
[51] "Basic ROS MoveIt! and Gazebo Integration · AS4SR/general_info Wiki · GitHub." [Online]. Available: https://github.com/AS4SR/general_info/wiki/Basic-ROS-MoveIt!-and-Gazebo-Integration. [Accessed: 30-Aug-2020].
[52] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh, "YAML: A tool for hardware design visualization and capture," in *Proceedings of the International Symposium on System Synthesis*, 2000, vol. 2000-January, pp. 9–14.
[53] C. J. C. H. Watkins and P. Dayan, "Q-Learning," 1992.
[54] T. Degris, M. White, and R. S. Sutton, "Off-Policy Actor-Critic," 2012.
[55] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee, "Incremental Natural Actor-Critic Algorithms."
[56] D. Silver, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms."
[57] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal Value Function Approximators."
[58] "Learn Robotics from Zero; Robotics & ROS Online Courses | The Construct." [Online]. Available: https://www.theconstructsim.com/. [Accessed: 22-Aug-2020].
[59] "Bitbucket." [Online]. Available: https://bitbucket.org/theconstructcore/. [Accessed: 24-Aug-2020].
[60] "marioacera/openai_ros_Robotic_Arm: Final Thesis Project." [Online]. Available: https://github.com/marioacera/openai_ros_Robotic_Arm. [Accessed: 08-Sep-2020].
[61] "GitHub - openai/baselines: OpenAI Baselines: high-quality implementations of reinforcement learning algorithms." [Online]. Available:

https://github.com/openai/baselines. [Accessed: 30-Aug-2020].
[62] "GitHub - hill-a/stable-baselines: A fork of OpenAI Baselines, implementations of reinforcement learning algorithms." [Online]. Available: https://github.com/hill-a/stable-baselines. [Accessed: 30-Aug-2020].
[63] "GitHub - jesseweisberg/moveo_ros: ROS packages and Arduino scripts that can be used to control the BCN3D Moveo robotic arm in simulation and real-life." [Online]. Available: https://github.com/jesseweisberg/moveo_ros. [Accessed: 02-Sep-2020].
[64] "MoveIt! Setup Assistant — moveit_tutorials Kinetic documentation." [Online]. Available: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html. [Accessed: 30-Aug-2020].
[65] "Move Group Python Interface — moveit_tutorials Kinetic documentation." [Online]. Available: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/move_group_python_interface/move_group_python_interface_tutorial.html. [Accessed: 07-Sep-2020].
[66] "Her.RolloutWorker not finding env.reset method · Issue #1083 · openai/baselines." [Online]. Available: https://github.com/openai/baselines/issues/1083. [Accessed: 09-Sep-2020].
[67] "Welcome to Stable Baselines docs! - RL Baselines Made Easy — Stable Baselines 2.10.2a0 documentation." [Online]. Available: https://stable-baselines.readthedocs.io/en/master/. [Accessed: 04-Sep-2020].

11. Appendix

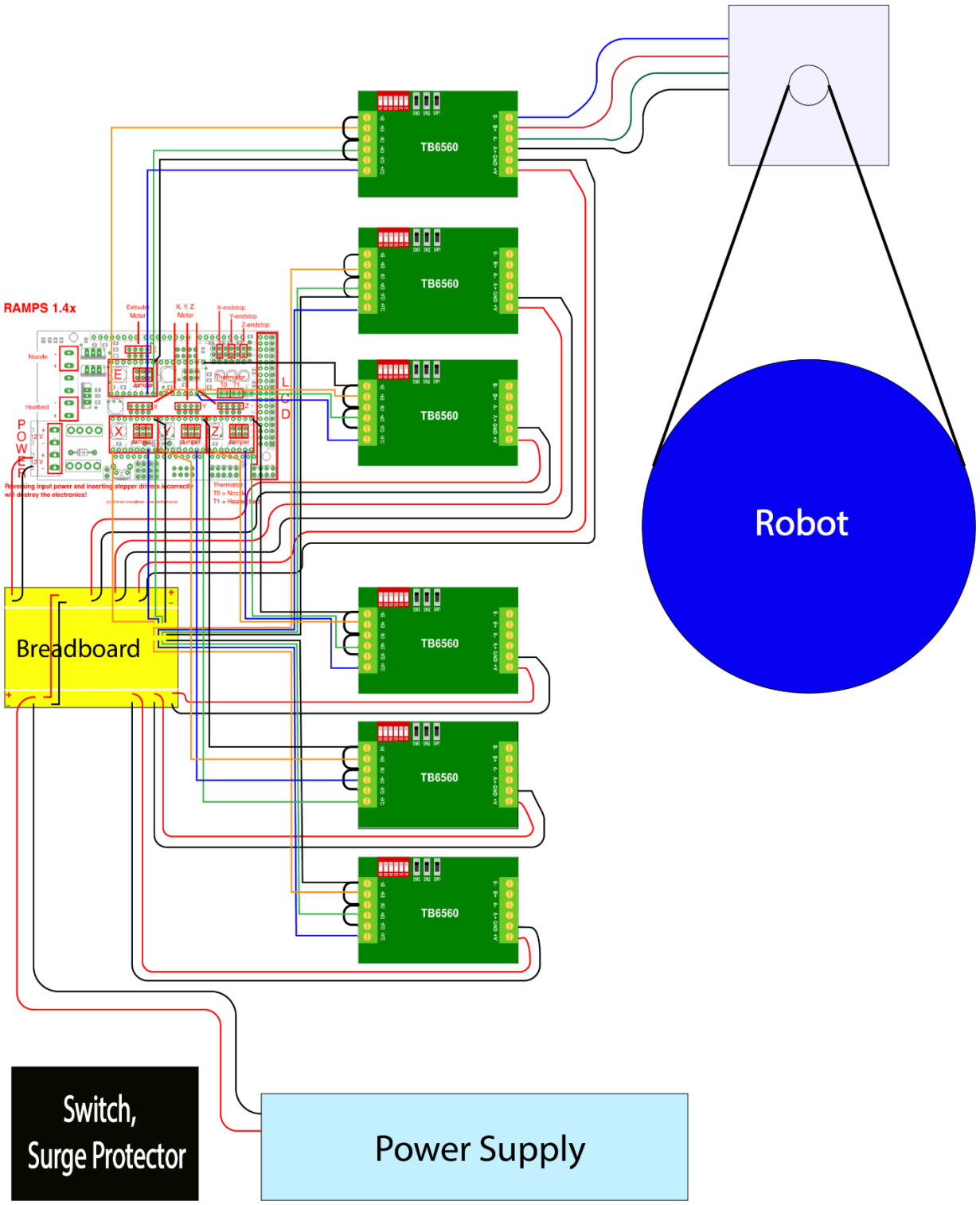


Figure 11.1: Singleline Schema for the controllers connections