POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master of Science Degree in
Mechatronic Engineering

Master Thesis

Robotic arm pick-and-place tasks

Implementation and comparison of approaches with and without machine learning (deep reinforcement learning) techniques

Supervisor
prof. Marcello Chiaberge
Co-supervisors:
Enrico Sutera
Vittorio Mazzia
Francesco Salvetti

Student
Alessandro Aiello
id: 256673

Academic year 2019-2020

Abstract

A robotic arm is nothing more than a mechatronic structure inspired by the conformation of the human arm and capable of performing various tasks. The ability to perform tasks, in relation to the "hand" that is given, makes robotic arms very versatile; they can be programmed to perform any task a human arm can perform, however grasping objects is certainly the most interesting and requested.

Pick-and-place in general is one of the most complete tasks that can be required of a robotic arm; although it may seem a trivial and immediate gesture for a man, for a robot it turns out to be a very complex task, it is not a coincidence that its complexity makes it a task well suited for studies and research.

The intrinsic multidisciplinary nature of the field of robotics makes it suitable for the integration of techniques and knowledge from all parts of engineering and science, this leads to the introduction of one of the most profitable applications in this sector: the artificial intelligence (AI). The discussion focused on the concept of machine learning (ML): branch at the base of AI, union of several disciplines such as computer science and mathematics among all, which allows an entity to learn autonomously.

The techniques for pick-and-place tasks have changed a lot over the decades, the aim of this thesis is to use a unified presentation of two approaches to deal with these tasks, first using the state of the art of the legacy of a so-called "classical methodology" (i.e. not including modern AI techniques), and secondly, instead, the same problem is addressed by treating the state of the art of the latest ML techniques for these types of actions.

As regards the first approach, ROS framework with the MoveIt platform, Gazebo simulator and RViz visualizer were chosen, in order to implement the code and functionalities necessary both to perform a complete simulation of the arm and to allow integration on a real machine. Concerning the second approach instead, the reinforcement learning (RL) paradigm was exploited, specifically deep RL algorithms have been used because they are considered the most suitable and efficient based on research in the field; for this purpose, Gym (with MuJoCo simulator), Stable Baselines and RL Baselines Zoo toolkits were used to work with the "DDPG + HER" algorithm, which was in turn implemented through TensorFlow.

The robotic arm used is that of the Fetch Robotics' Fetch robot.

The author has chosen to organize the work on the basis of a rather widespread definition of the term "robotics", that is "intelligent connection between perception and action"; therefore, with this meaning, it is necessary that the robot perceives, reasons and acts. Precisely in this vision, the study and the two approaches refer to the part of the so-called "intelligent connection", specifically to the implementation of the reasoning entity: the motion planning system.

The results of the thesis work are the implementation of the two robotic programming approaches described above and the comparison of their peculiar characteristics.

List of Illustrations

Figures

Tables

Contents

Thesis Structure

The document is divided into parts.

1 **Introduction**: in order to have a basis on which to structure the considerations and topics covered in the following parts, a general introduction on the main concepts of robotics (with specific considerations on manipulators) is made, defining separately the components that allow the robot to perceive and act, as those that act as a connection between the two, in particular the motion planning system; the part ends with other information and knowledge common to both approaches.

2 **I - Approach Without Machine Learning**: with reference to the "thinking" system (the motion planning one) of the machine, detailed analyzes are carried out on the approaches considered standard, the state of the art is described, the practical work carried out and finally conclusions are given on what has been done.

3 **II - Machine Learning Approach**: the topic is treated with a scheme similar to that described in the previous part but from the point of view of a machine learning approach, in addition there is also a chapter for a recap on the main notions useful for the purposes of the thesis, regarding the theory on deep reinforcement learning.

4 **Conclusions**: short concluding part to dedicate a specific space to the comparison between the two approaches previously described.

5 **Appendices**: used to avoid overloading the reading with extra material and to insert specific or technical information that does not concern a specific part.

Below it is possible to find a brief description of the appendices present.

A Hardware and Software Setup: a register of the system configuration and software (versions and installation way) used.

B Acronyms: the Table B.1 with the acronyms used in the document sorted alphabetically

In addition, the first three parts show an introductory summary in order to give a general presentation, listing the chapters and describing them briefly for a better overview.

Introduction

Summary

In the literature there are various functional schemes and models that describe the overall system of a robot (or specifically a manipulator), but later a personal analysis of the actors involved on the expected needs for the introduction of the work will be proposed.

A robotic system can be seen as an "intelligent connection between perception and action"[18], and it is precisely on this principle that the author wants to structure the work; in this part the subsystems that allow the machine to perceive and act will be introduced, but even more attention will be given to those that "intelligently" connect them.

The first part intends to introduce the reader to the methodology chosen for the entire thesis, showing the entire system with which deal from a broader point of view (to give a general knowledge of what is "behind" a robot), but highlighting the section of this whole system that is the protagonist of the work carried out, which as will be repeated later is the motion planning system.

This because following the state of the art and the evolution of robotics in the field of programming applied to the latter, the direction taken is that of a modular work protracted at the high level of the tasks themselves and not at the lower levels of the subsystems that in practice allow their realization. For this reason, as will be clarified in detail below, the comparison takes place at the level of pure task programming and therefore at the level of motion planning.

The above is discussed in the first two chapters, while the third and last chapter shows other information and knowledge common to both approaches, for example introducing the manipulator chosen or explaing in detail the main characteristics of pick-and-place (and of tools with which it is carried out), characteristics of the task and not of the approach used to achieve it.

Chapter 1

Intelligent Connection

In this first chapter we will start from the components that provide the "intelligence" to the system, allowing to connect the elements of perception and actuation.

Precisely for the reason described above it was decided to use a top-down approach, starting from what is the objective required by the programming of a robotic arm, that is, the completion of a task, this can only be done by making the robot perform a movement, using a generic term. Consequently, using an essential description, the purpose, when treating manipulators, turns out to be really the "simple" execution of a desired movement, all the complexity results in how this movement is carried out.

Before continuing, however, in this introductory chapter it is necessary to make an essential assumption: controllers are not devices in which there is actual reasoning, they do nothing but give a signal to make the system follow it; consequently this document will conventionally consider the system designed to implement the "intelligent" connection between perception and action, as divided into two subsystems, that of real reasoning (i.e. planning) and that of controlling.

Precisely for this reason, as already mentioned previously, the entire work and consequently the approaches studied, are all to be understood at the conceptually highest level of the robotic system, namely that of the "heart" of the machine: the *motion planning*.

1.1 Motion Planning

In order to carry out the desired movement it is essential to introduce the concept of motion planning. This term can be explained by breaking it down into two parts: *trajectory planning* and *collision avoidance*.

It should be noted that this subdivision is fictitious and is introduced only for simplicity of explanation, in reality the two subsystems work together and can be considered a single entity.

1.1.1 Trajectory Planning

The trajectory planning process is responsible for generating the reference inputs to the motion control system which will in turn make the manipulator execute the required trajectory. Trajectory means not only the path but also the speed and accelerations with which it is carried out.

The trajectory can be of a *point-to-point* type or a *sequence of points*, clearly the second case is the more complex one and is nothing more than a more complete generalization of the first, the trajectory thus described serves to define time constraints necessary for the completion of more complex tasks, or to appropriately avoid collisions.

1.1.2 Collision Avoidance

Collision avoidance, on the other hand, refers to the subsystem which, by means of information on the external environment and on the structure of the manipulator, allows to verify and identify collisions between the two and between the last one and itself, providing the information necessary to avoid them.

1.2 Motion (and Force) Control

As mentioned earlier when talking about trajectory planning, there is a system that takes care of taking the information on the trajectory as input and generating a signal that, through appropriate components called *actuators*, allows the robot to make a movement; this system is the *control system*.

The motion control is carried out by means of special devices known as controllers, which take as input the reference given by the motion planning system and provide in output, through an appropriate control law, the electrical command signal necessary to drive the actuators that physically produce movement. The control technique used is the *closed loop* one (Fig. 1.1), this because the system output signal is fed back to the controller input allowing a more stable and dynamic control.

Depending on the task required, the control can be carried out in various ways, based on what has been said previously regarding the trajectory, the need for a control system not only of the position, velocity or acceleration but also of force (or torque) is evident.

1.3 Model

This section does not introduce an “intelligent element” unlike the previous ones, but clearly, in order for the above to work, it is necessary to introduce a mathematical model, which models the manipulator in order to be able to describe and predict its static and dynamic behavior. To do this, we start from the mechanics of the robotic arm: from a structural point of view, a robotic arm is nothing more than a chain of alternated *links* (rigid bodies) and *joints* (articulations).

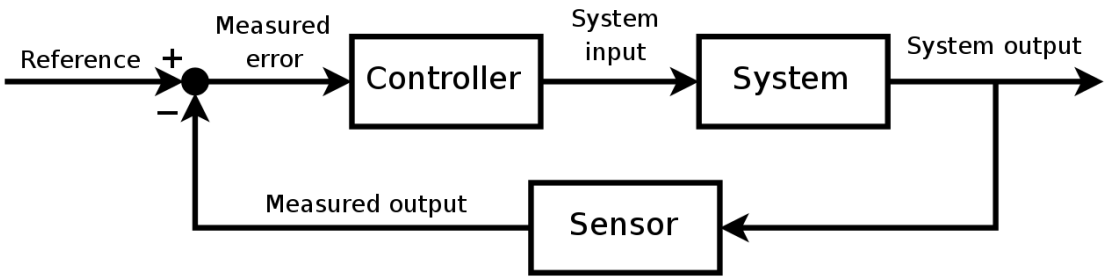


Figure 1.1: Closed loop (feedback) control block scheme
[credits: wikipedia.org by Orzetto]

The mathematical models of the manipulators allow us to relate the movement of the *end effector* (the last link of the arm) with that of the joints, specifically *kinematics equations* relate their positions, while *differential kinematics equations* do so for the speed; these equations therefore do not take into account the forces that cause the movement but are useful (the latter specifically) to describe the relationship between the forces applied to the end effector and the torques to the joints in a (static) equilibrium condition. To relate the accelerations, instead, *dynamic equations* are used (which also take into account the physical characteristics of the manipulator), which unlike the previous ones describe the relationship between the forces on the end effector and the torques to the joints in precisely dynamic conditions.

Since this thesis is not a treatise on robotic arms, the concepts of statics and dynamics will not be explored in depth, however it is necessary to know for the purposes of the analysis that above all the latter plays a fundamental role, as it is useful both for the design of control algorithms and for carrying out simulations.

The three equations mentioned can be used in direct or inverse form, the latter being the most important form, as for the first two specifically it allows, given a desired trajectory for the end effector, to obtain the constraints on the positions and speed to the joints to get it; for the last equation, on the other hand, it allows to obtain the torque required for the joint in order to satisfy constraints regarding accelerations in the trajectory. Consequently, the inverse forms are also used for the control in order to respect the respective constraints required on the trajectory.

In the design phase, the mathematical models of the arm are already defined, because it is through these models (the static and dynamic ones mentioned previously) that, based on the purpose of the arm, its parameters and physical characteristics such as materials for the links and for joints (friction, etc.), mass of components, actuators to be used, etc. are determined.

1.4 Processing System

Even this system, as was the case in the previous section, does not identify an entity that is part of “intelligence” from a conceptual point of view, but rather shows how to

implement the above in practice. The entity in question on which the due considerations will be made is the processing system: that is the computers (to use a generic term) that perform both the normal computational operations of the software, for motion planning in general but also other specific algorithms for its correct functioning (such as those for vision for example), and those of the control system.

To this end, every type of computer can obviously be used, from workstations to main frames, from PCs to smartphones, based on the purpose, however, it is necessary to find the best trade off between performances, costs, computer size, required energy, and take into account of other factors such as dissipated heat, weight (in any type of robotic system, and manipulators are no exception, in variable percentages and according to needs, the processing system can be distributed partly on the machine or partly distant from it), etc.

Given the technological development since the advent of the first manipulators, one of the main choices on which the calculation system has fallen has been that of *embedded systems*.

The types of embedded systems used are mainly two (without analyzing their advantages and disadvantages) (Fig. 1.2): *microcontroller* (MCU - Micro Controller Unit) and *system on chip* (SoC), both derive from the microprocessor, but while the former refers to a single monolithic device that integrates CPU (or in this case microcontroller unit), RAM, I/O peripherals and other components depending on the purpose (Arduino for example is a hardware platform equipped with a microcontroller), the second refers to a board on which the necessary components are discreetly soldered and only some of them are integrated into the CPU (in this case: actual microprocessor) (Raspberry PI for example is a single board computer whose microprocessor is mounted on a system-on-chip).

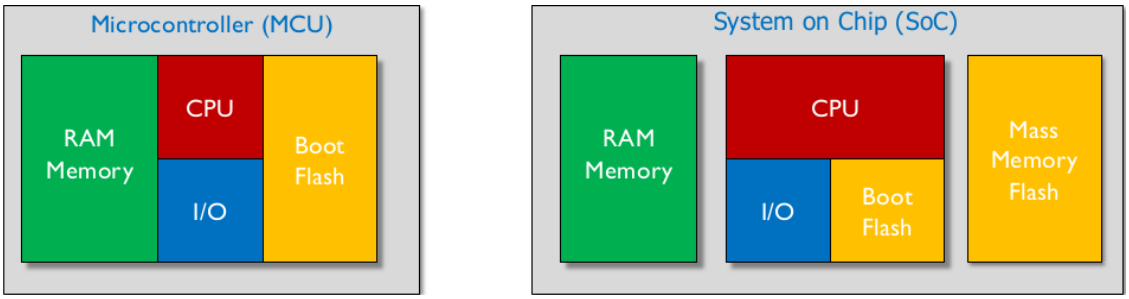


Figure 1.2: Schematic comparison between microcontroller and system-on-chip [source: slides of PoliTo Model-Based Software Design course by Massimo Violante]

So as far as the control system is concerned, microcontrollers are the cheapest device and have sufficient resources to implement an ad hoc controller, for these devices special OSs (Operating Systems) have also been developed to meet the need for particularly reliable real-time control, their small size and the reduced amount of energy required allow direct integration on the machine. It is important to specify that the above applies to almost all fields of applications except for industrial robotics, in which,

given the needs, the most successful device so far remains the PLC (Programmable Logic Controller).

Initially they were programmed with low-level languages, nowadays a controller can be comfortably designed with special software such as MATLAB, also capable of converting the project into high-level code for testing and direct integration.

As for everything else, i.e. the thinking part on which all the other algorithms run, microcontrollers do not lend themselves to the purpose and it is necessary to opt for systems with more performing microprocessors. Historically every type of computer has been used for this purpose, with the first technologies we found ourselves in situations where the computer was even larger than the robot to which it was connected; currently with the great technological development both in terms of computer size and computational performances, but also with regard to the power of wireless and non-wireless communications, the possible approaches are practically infinite.

Currently with regard to this aspect of the processing system, the embedded systems of the type of system on chip lend themselves well to this purpose, being equipped with operating systems comparable or even the same as those present on personal computers, they allow to have in all respects a real on-board computer mounted on the machine.

Chapter 2

Perception and Action

In this chapter, instead, the second part of the statement "intelligent connection between perception and action" will be treated, so the sensors (with particular attention to the vision system) that deal with perception and the actuators that perform the actions will be introduced.

2.1 Sensors

In addition to the actuators, there are other components that play a fundamental role in the success of the movement: the sensors.

According to the type of control, the output signal that is reported to the input can be the same position, velocity or acceleration of the joints obtained through classic sensors, or the respective data referring to the end effector obtained through kinematic algorithms or, according to the applications, always via sensors.

In the second case, always respecting application constraints, it is also possible to use *cameras* which, also operating thanks to sensors (interchangeably called vision sensors or image sensors), provide information (visual feedback) on the actual action performed by the actuation system in response to the command given by controllers; this topic will later be resumed by talking about the *vision system*, specifically *visual servoing*.

In all cases, however, a sensor known as *transducer* is further required which adequately converts the output signal so that it can be reused at the controller input.

Sensors that can be *proprioceptive* (on joints): position, speed, acceleration/torque to name a few, or *exteroceptive*: force and tactile (e.g. useful for grasping), proximity, range, vision, etc., are of importance primary in the field of robotics and automation in general, so much so that despite the hundreds of typologies already conceived, research is still very active in the field.

2.2 Vision

As will be explained later, in the work it will be assumed to be aware of the pose and shape of the object and therefore the whole phase concerning obtaining this information is not considered, however it is appropriate to summarize the way in which it is obtained, process which passes through the use of cameras (and therefore vision sensors).

Although this topic would concern sensors in a certain sense, the author has chosen to dedicate a special section for greater completeness, and to be able to introduce and justify some machine learning techniques that have applications in this area and that will be treated in the specific Part II.

Although this system can be considered of secondary importance, as in fact not essential, unlike the others, for an operation that can be defined as sufficient of a robotic system, it certainly completes its functionality by showing one of the aspects that most distinguish the simple "stupid" automation from actual robotics.

While the *computer vision* describes a more scientific than engineering concept, terms that will be presented after refer to specific technical applications of the generic field of it.

2.2.1 Computer Vision

Computer vision (CV), with which term refers precisely to vision systems, is a field that groups several subdomains such as object recognition, 3D pose estimation, motion estimation, visual servoing, etc.; it deals with obtaining information from 2D or 3D images and videos in the most generic possible conception.

OpenCV is currently the main library for the implementation of CV algorithms.

2.2.2 Machine Vision

Machine vision (MV) exploits CV paradigms to obtain information from images, in order to implement specific applications in the industrial field for inspection, process control, and robot guidance, and refers to both software and hardware technologies.

2.2.3 Robot Vision

Robot vision (RV), although as a similar term and sometimes usable interchangeably with MV, refers to the use of images to output not so much information as actual actions, concerns the specific field of robotics and visual servoing is an example of it.

Visual Servoing

Visual Servoing (VS, also known as vision-based robot control) is responsible for controlling the movement of a machine through a feedback system based on "visual" information (visual feedback), it is the same concept already introduced when talking about sensors, the feedback can take place through the information on the actual

action carried out by the actuation system compared to the command given by the controllers.

The VS can be implemented through different approaches based on the arrangement between the robot end effector (hand) and the camera, the following are the two main ones:[3]

- Eye-to-hand (or end-point open-loop control): fixed camera in the environment that observe target and hand;
- Eye-in-hand (or end-point closed-loop control): the camera, attached to the hand, observe the relative position of the target.

ViSP (Visual Servoing Platform) is the best library for visual tracking and visual servoing that can also exploit OpenCV.

2.3 Actuators

The term actuators usually refers to motors as components that conceptually act directly on the joints, it should however be noted that these are part of a broader system called *actuating system*, which although always referring to the joints is composed of other elements also not directly connected to them, but essential for the functioning of the overall system.

There is no unambiguous definition on what specifically this system may contain beyond the motors, generally some standard components are *power supplies*, *power amplifiers* and *transmission systems*.

Motors are referred to with the term *servomotors*, which indicates actuators with certain characteristics that make them suitable for use in a closed-loop control system; they can be electric, hydraulic or pneumatic but the most used are undoubtedly the electric ones.

The discussion will be carried out at a higher level than the actuators so no mention will be made about the electric drives and the topic will not be further treated.

Chapter 3

Common Info to Both Approaches

In this chapter information and knowledge, concerning the task required specifically and in practice, common to both approaches, will be collected.

3.1 Pick-and-Place

Since this discussion is focused on pick-and-place, a brief specific introduction will also be reserved for this topic.

3.1.1 Grippers

Grippers are none other than the end effectors used for pick-and-place tasks and specifically for grasping (other end effectors can be different types of tools depending on the task to be performed); they can be classified into four classes:[13]

- impactive: jaws or fingers (two-fingers or multi-fingered grippers) which physically grasp by direct impact, a mechanical force is directly applied on two (or more) opposite points on the surface of the object;
- astrictive: attractive forces applied only on one point or on the surface of the object (for example by vacuum, magnetoadhesion or electroadhesion);
- ingressive: pins or needles are used to physically penetrate, deforme or permeate the surface of the object;
- contigutive: after direct contact adhesion force takes place (for example with glue, freezing or surface tension).

It is anticipated that the discussion, as will be shown in Part I of the work, foresees the use of grippers belonging to the first class, namely impactive gripper, and specifically two-fingered gripper will be used.

3.1.2 Task Phases

Then 4 phases will be shown in which the task can be divided: pre-grasp manipulation (preparation), grasp acquisition (pick or grasping), post-grasp transport (holding) and goal (place or releasing); in brackets are indicated names from the point of view of the grasp.

Based on the dynamism of the environment, the pick-and-place can be statically predefined by the motion planning system a priori, or be adapted in real-time on the basis of information obtained from appropriate sensors.

Pre-Grasp Manipulation

The gripper of the manipulator from an initial pose (by pose means position and orientation) is placed in a suitable pose to prepare to pick up the object, the pose depends on the pose and shape of the object, the type of gripper, the presence of any obstacles or even the need to respect any constraints.

The variables just described lead to the definition of appropriate points of contact between the gripper and the object, which on the basis of what has been said can be classified as follows[13]:

- point contact
- line contact
- surface contact
- spherical contact
- two-line contact

Grasp Acquisition (Pick)

It is actually the phase in which the gripper engages the target, it is the most important phase, as well as the most delicate and complicated one; for example, with regard to force control, its importance in this phase should be emphasized. In general, real objects are not rigid non-deformable bodies, so necessary attention must be paid to controlling the joints in order to preserve the structure of the object to be gripped without damaging it.

With reference to the type of gripper treated, this phase involves the sufficient opening of the gripper fingers to be able to grasp, and their closure to be able to consider the object grasped.

In the literature, two main requirements for the containment of the object in the grasp have been conventionally identified and defined: *form closure* (or form-fit) and *force closure* (or force-fit)[17].

The first occurs when the closure is due to the respective geometries of the gripper and object, in this case a casing condition occurs for the object which is immovable; this closure reduces the force required for grasping because, as mentioned, it is guaranteed

more by the geometry than by the force applied. As an example, it is possible to imagine to grasp an apple using the whole hand with the fingers that wrap the apple to form a cage that holds it totally firm.

The second case refers to the situation in which the forces applied, through friction, allow to maintain the grip on the object. In this case, to get an idea of the concept, you can imagine holding a pen with a firm grip with two fingers.

Clearly the first case implies the second but not vice versa.

In the discussion we will use a two-finger gripper that grasp in the most general way, i.e. in force closure and not in form closure.

Post-Grasp Transport

Basically the grasping task is not an end in itself but has the purpose of varying the pose of an object from the initial one to a final one and then finally accomplish a goal.

This phase does not require additional explanation as it is a simple movement of the gripper (with the object gripped) from the point of view of the action to be performed.

Goal (Place)

By goal we mean the achievement of the goal, which for a task of this type involves the controlled placement of the object to be released, bringing the gripper into an "open" pose, and the possible repositioning of the arm in a predetermined pose.

In other cases, i.e. in cases where the task is not only the pick-and-place, but the object to be taken is considered a tool with which the arm must perform an action, we can mean an action goal more or less complex based on what is precisely the ultimate goal of the task.

3.2 Fetch Robot Manipulator

The manipulator chosen is that of the Fetch Mobile Manipulator[20] (Fig. 3.1) from Fetch Robotics, having 7 DOF (Degrees Of Freedom) and a two-fingered parallel gripper (with compatibility with external grippers).

The technical specifications of the arm and gripper of the Fetch robot are summarized in the Table 3.1

Given the characteristics of the Fetch robot, the author of this thesis considers it as the best robotic system for research and service robotics, the manufacturing company has invested in ROS support as the main factor and it enjoys excellent integration with MoveIt (the ROS library used in this work), as well as possessing valid documentation, furthermore, its functions, of which the arm is only a part, make this same work suitable for various integrations with future projects.

Figure 3.1: Fetch Robotics Fetch Mobile Manipulator [credits: fetchrobotics.com]

Table 3.1: Fetch robot arm and gripper specifications [source: fetchrobotics.com]

7-DOF arm	
Payload (at full extension)	6 kg (13.23 lbs), 5 kg (11.02 lbs) with gripper installed
Arm length to gripper mount	940.5 mm (37.25 in)
Max end effector speed	1.0 m/s (2.23 mph)
Gripper	
Max grasp force	245 N
Max gripper opening	100 mm (3.94 in)
Wrist max pickup	6 kg (13.27 lbs)
Grip weight	1kg (2.20 lbs)

Part I

Approach Without Machine Learning

Summary

In this second part a "standard" approach (so without considering modern machine learning techniques) in carrying out robotic tasks will be dealt with, and specifically, as regards the chapters dealing with the work performed, pick-and-place tasks performed by manipulators.

A first introductory chapter, reconfirming in detail the concepts seen previously, shows the evolution of programming techniques to explain the more modern one to which development in the robotics field is leading.

The second chapter shows the current state of the art by describing the main frameworks and softwares that have widely established themselves; furthermore, through the latter, it highlights the main points of the workflow that has become standardized in the sector and which will later be shown in the work carried out.

The work done is described in chapter three, which explains not only the workflow carried out by the author but also the relevant functioning of the entities shown in the previous chapter.

In the final chapter the results of the work completed will be shown and due considerations will be made, while a special section is reserved for ideas and advice on possible future work and/or improvements.

Chapter 4

Introduction

Starting from the model and thus obtaining a manipulator that works on paper, it is necessary to take care of its programming in order to implement its reasoning system, i.e. the one that through the information obtained from the appropriate entities, given an assigned task, is capable of providing the information necessary for its completion to all the entities underlying: the motion planning system.

As already said, this thesis will show the difference between two approaches from the point of view of robot programming, where programming refers precisely to the reasoning part and therefore to all the code that decides the actions and how they must be carried out (not so much from the point of view of the components that actually perform them), to this end we can define three types of programming techniques that have marked the history of robotics; later they will be briefly described (without going into detailed comparisons on the advantages and disadvantages of each) in order to show the changes from the first methodologies, focusing specifically on the evolution of the latest techniques[18]:

- teaching by showing (demonstration)
- robot-oriented programming
- object-oriented programming

4.1 Teaching by Showing/Demonstration

Having the arm physically available, an operator guides its movements according to the task to be performed, in this way the sensors read the data on the joints during the movement and they can be used several times to perform the movement again independently (without the presence of the operator) to perform the given task repeatedly.

The disadvantages of this technique, which lends itself only to the programming of a "stupid" automaton, capable only of repeating the same movements that it has been taught, are evident, bringing this methodology to be outclassed and supplanted by the two following; this at least up to the most recent research, where (as will be

mentioned in more detail in the introduction on the Part III) with machine learning (ML) techniques it has been possible to discover new potentialities of this technology making it interesting and competitive for many purposes (leading to define it, in a more appropriate way, *learning by showing/demonstration*).

Furthermore, in modern times, both with but also without the use of ML techniques, new approaches have been developed that involve the use of *simulations* (through special simulators) and *gestures* in order to program the movements of the machine in a more efficient way that what was done before.

It should be emphasized with this approach that part of the concept of motion planning fails, it is a limit situation from simple pure automation to true robotics, apart from reading from sensors, saving these data and sending them to the control system when necessary, all the little computational load resides in the microcontrollers, which perform their inevitable essential function of supplying the command signal to the actuators.

4.2 Robot-Oriented Programming

This is the first real form of *programming* introduced (before the "programmer" was nothing more than an operator, now the programmer must have computer skills and knowledges in the field of programming languages), both from the conceptual point of view of the approach and from what regards the programming languages actually used, this methodology tries to tackle the problem at ever higher levels in order to abstract the hardware as much as possible.

In this context, a real writing of functional software takes place, where in several programs every function required by the machine is coded in addition to the control ones alone. However, knowledge of the hardware is partly necessary, programming is generally entrusted to the arm manufacturers themselves or to other companies that tackle the problem on behalf of the former, based on the same hardware documentation provided by the manufacturers.

In most cases, ad hoc languages are developed, often proprietary, with the consequent writing of non-free code suitable to work only for that specific arm or for the arms of that given manufacturer.

Simulators began to spread, software also often written by the same manufacturers companies (for internal purposes), which emulate real-world physics through appropriate mathematical models, and which through models of the arm can test its functioning without actually having the physical manipulator.

4.3 Object-Oriented Programming

Abstraction is one of the basic concepts of object-oriented programming, well-structured languages of this type allow you to write programs and real environments (*frameworks*) that give the programmer the ability to work in the most efficient way directly at the *task level* and no longer at the level of single actions or elementary movements; the

purpose is to minimize the necessary knowledge of hardware, favoring the reuse of code and developing modular libraries that allow the code not to be dependent on it.

This clearly describes a new programming paradigm, but does not exclude the previous approaches, in which the writing of proprietary software continues to be the first choice for many companies.

Not so much object-oriented programming, but the evolution of the world of robotics itself which occurred at the same time, has led to the affirmation and spread of a large number of simulators, both proprietary but above all open source, dynamic and versatile, which lend themselves to use for various robotic systems to support research and design.

Chapter 5

State of the Art

In this specific field of discussion, the state of the art does not translate into theoretical research work but rather into the practical development of well-structured platforms, frameworks and various softwares; consequently the focus is placed on actual programming.

Based of the above, the main protagonists of the ecosystem of modern robotics will be presented below.

5.1 ROS

ROS (Robot Operating System)[15] is an open source collection of frameworks for robots development and programming, where the term "framework" refers to a structure that represents a support architecture for the design and implementation of software, this structure takes shape in libraries and tools.

ROS provides an infinite and ever-growing quantity of packages to program robots from every point of view, making an absolutely non-exhaustive example, one of the most important is the *ros_control*[5] set of packages that allow interfacing with lower level controllers.

Some of the most important aspects, as well as among the main ones that have led to the affirmation of ROS, are the excellent documentation it enjoys, the frequent work of maintaining and updating the official libraries, and not least the active community, which in primis translates into forums and support spaces for users.

RViz and rqt are two of the main general purpose packages widely used in any advanced ROS-based project.

5.1.1 RViz

RViz[9] is a 3D visualization tool with GUI (Graphic User Interface) natively supported by ROS, it not only allows to view and analyze in detail any model of robotic system or environment in which the latter is inserted, but also allows to acquire and reproduce

the data obtained from any sensor for which a specific module has been implemented in ROS.

It is also highly modular as it allows the integration of any functionality simply by adding the appropriate plugin, for example among these there is also one of MoveIt (Sec. 5.3), allowing operations and commands directly from the plugin interface, making RViz a software with functionality that go far beyond just visualization.

5.1.2 rqt

rqt is a simple and lightweight ROS framework implemented in both Python and C++ which acts as a graphical wrapper for a multitude of plugins; its purpose is to facilitate as much as possible the development of complex projects by providing specific functionalities (implemented through the appropriate plugins) through a GUI, and performing functions that sometimes would be much more complicated and complex through classic terminal commands.

5.2 Gazebo

Gazebo[10] is an open source 3D robotics simulator natively supported by ROS, it is probably the most popular robotics simulator together with MuJoCo (discussed later in the next part of this thesis) and certainly the most widespread and used of the open source ones.

It is a complete simulator equipped with any necessary functionality, just to name a few: it has its own format (the SDF - Simulation Description Format) for the description of robot models or environments, for which a large number of them are freely available; it has a wide variety of plugins available and also allows the use of custom developed ad hoc; it is equipped with several efficient physics engines according to your needs.

In summary, Gazebo allows to all intents and purposes to simulate every aspect of the real world in detail, from gravity to wind as regards the environments, from the dynamics of the controllers to the behavior of the sensors (taking into account the noises) as regards the robots, and from the distribution of inertias to friction for any model in general, and these are just a few examples.

Such as for ROS, also Gazebo enjoys the characteristics of open source products, having at its disposal a rich community and efficient support forums, as well as an increasingly active development.

5.3 MoveIt

The state of the art regarding robot programming and specifically the implementation of motion planning features is definitely MoveIt[6, 4]: “Easy-to-use open source robotics manipulation platform for developing commercial applications, prototyping designs, and benchmarking algorithms.” [source: moveit.ros.org].

Although born for manipulators, now MoveIt is used by big brands such as Google, NASA, Microsoft (just to name a few) for an ever-increasing number of robotic systems, becoming a de facto standard. It consists of a series of ROS libraries and implements functions perfectly integrated with the rest of the framework, it uses RViz visualization software and Gazebo simulation software, the same ones natively supported by ROS; it also has modules for the definition of tasks and a module entirely dedicated to grasping.

MoveIt essentially allows you to implement every functionality described above, from trajectory planning to collision avoidance and sensor functionality (for 3D perception), it also implements functionality for a further abstraction of the controllers, passing from the one already provided by ROS (ros_control) by virtualizing a control to high level based not on strength or position but on the trajectory itself.

Chapter 6

Work

Before starting it is necessary to make an assumption: as already mentioned, for simplicity and to focus on the main purpose of the work, it is assumed of being aware of the pose and shape of the object and therefore not consider the whole phase concerning obtaining this information; furthermore, since the shape of the object is relevant for grasping, it will be limited to objects of known geometric shape (parallelepipeds, cylinders, etc.), specifically a cube was used.

The main focus of all the work is on MoveIt, consequently with Fig. 6.1 and Fig. 6.2 (on page 39) an introduction is made on its main concepts regarding its system architecture and the central component of the whole framework: the *move_group* node; in them you can find some of the notions discussed previously, and in the following sections the main elements will be explained.

6.1 Moveit Robot Configuration Package

The use of a robot via MoveIt takes place by means of a configuration package that follows the convention *<robot_name>_moveit_config* in the name.

Regarding the robotic arms with which MoveIt has already officially been used, on the official website it is possible to have an updated list, however it should be emphasized that the framework has a package called *Setup Assistant* which allows, through a practical GUI, to generate the standard configuration package for any robotic arm. This is possible starting from the simple URDF (Unified Robot Description Format: an XML file that describe all the elements of a robotic system in ROS) of the same or from an eventual *xacro* (XML Macros: an XML macro language used to construct more readable and modular URDF files) file which will be parsed automatically.

Later it is described the structure of this package to briefly analyze its components in order to better understand the functionality of this environment; specifically, in a standard version, there are two folders inside the package:

- *config* folder
- *launch* folder

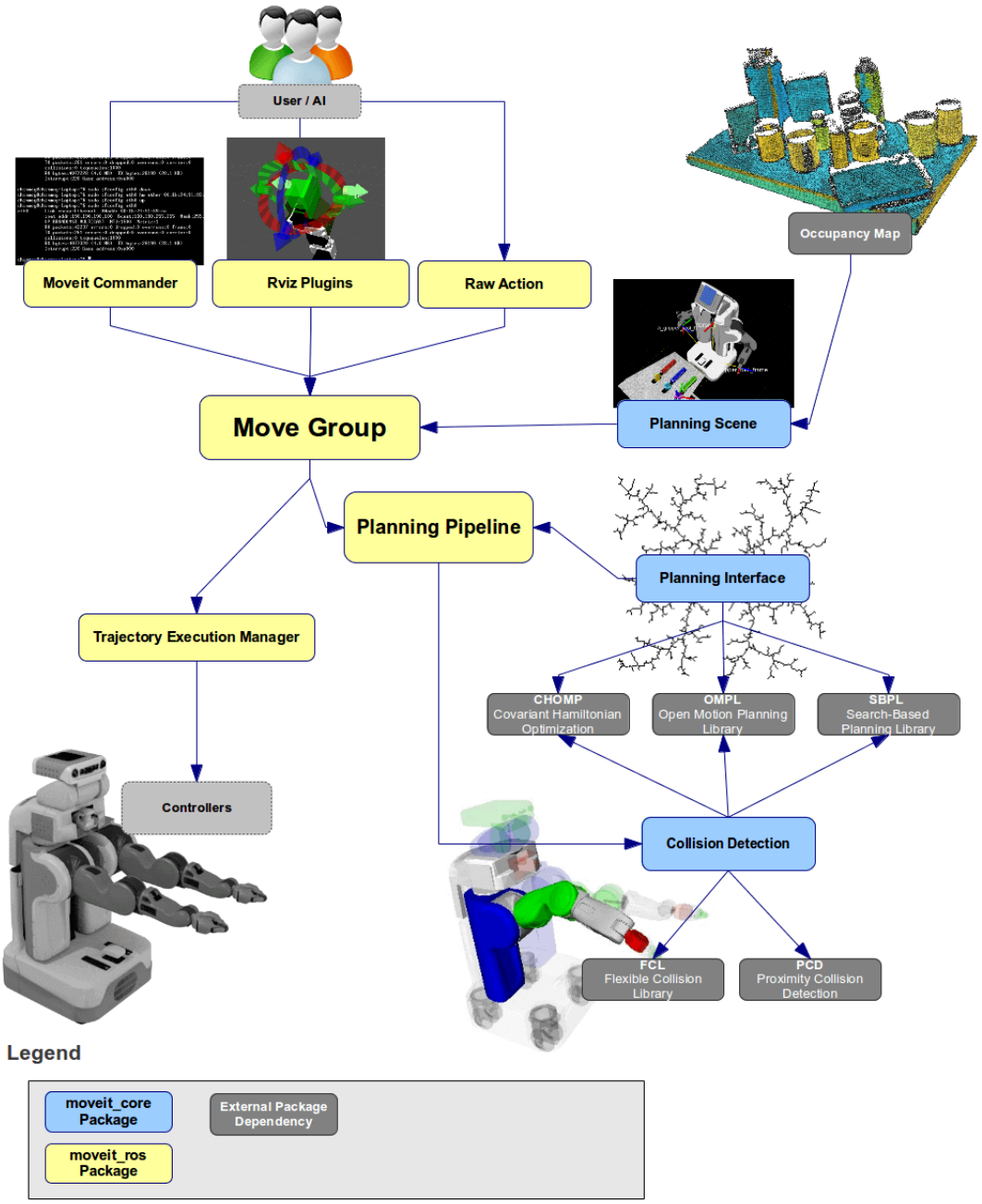


Figure 6.1: High level system architecture diagram of MoveIt [credits: moveit.ros.org]

6.1.1 Configuration Folder

This folder contains the actual robot configuration *.yaml* files, the only exception is the SRDF (Semantic Robot Description Format) file which complements the information provided by the URDF by adding tags for semantic (and functional for application

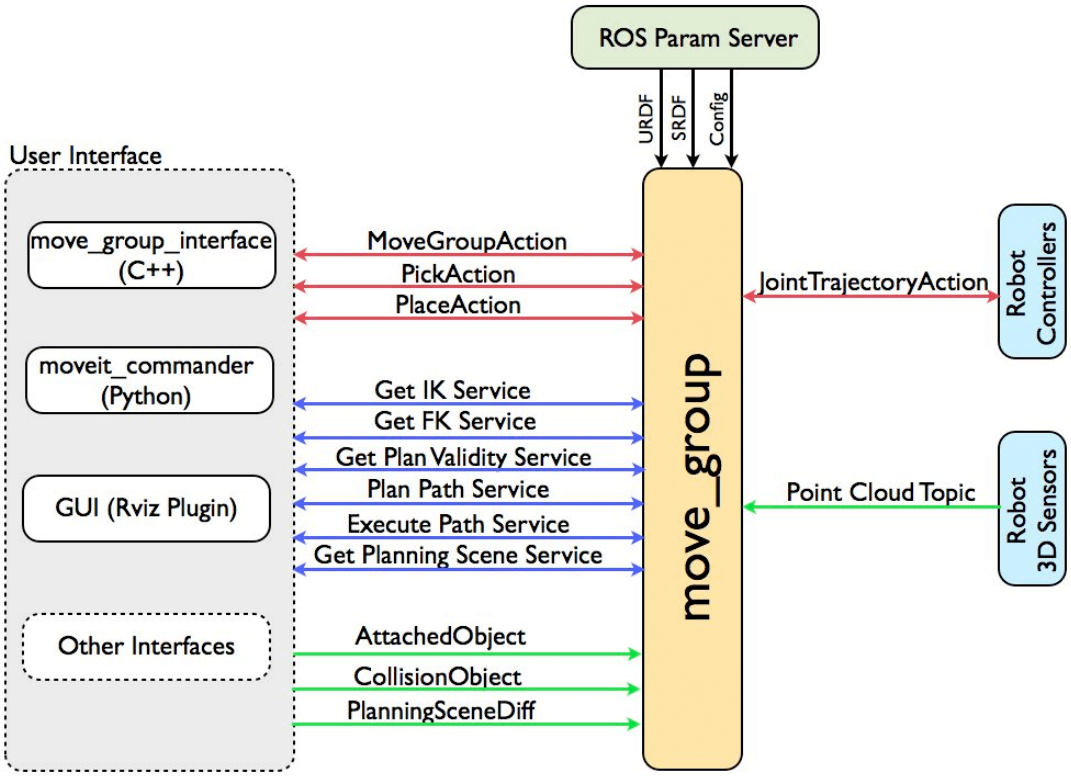


Figure 6.2: High level system architecture diagram of the MoveIt *move_group* node [credits: moveit.ros.org]

purposes) definitions besides purely physical ones.

Below are presented the configuration files in question referring to them on the basis of the entity they configure.

Controllers

It lists the controllers and specifies their configurations for the robot, usually two sets, one for the arm and the other for the hand (gripper); as explained above, working MoveIt with trajectories, the first set describes controllers that perform actions of the *FollowJointTrajectory* type, while the second generally presents controls with actions of the *GripperCommand* type.

Fake Controllers

It lists fictitious trajectory controllers and specifies their configurations, these controllers are used to carry out tests and display the behavior of the robot on RViz. However, since they are not real, they cannot be used neither for an implementation on the actual robot nor in simulation with Gazebo, this is because both the simulator

and the arm model also emulate the controls by treating the entire system as a real system.

They are listed here for completeness but for the reasons explained they are not be used.

Joint Limits

For each joint it describes any possible constraints on minimum and maximum velocities and accelerations.

Kinematics

Configure the kinematics for the robot, specifically it indicates the *solver* (the plugin) to be used for the execution of the inverse kinematics algorithms and defines the parameters, such as resolution and time constraints.

There are three solver plugins currently available:

- KDL (Kinematics and Dynamics Library) Kinematics Plugin
- LMA (Levenberg-Marquardt) Kinematics Plugin
- Trac-IK Kinematics Plugin

The latter is not yet fully integrated into the MoveIt system and requires separate installation, of the first two the first has been selected as it is best supported, default plugin used for MoveIt and indicated in the case of arms with more than 6 DOF.

Planner

It configure the motion planning planner.

MoveIt is equipped with various types of planners for motion planning:

- OMPL (Open Motion Planning Library)
- CHOMP (Covariant Hamiltonian Optimization for Motion Planning)
- STOMP (Stochastic Trajectory Optimization for Motion Planning)
- SBPL (Search-Based Planning Library) (full integration work in progress)

We will use the first with its default configuration, as it is the default in MoveIt, complete, and which is suitable for most situations.

Sensors

Included for completeness but not used: vision sensor configuration file that uses Microsoft well-known Kinect as sensor.

6.1.2 Launch Folder

This folder contains the essential launch files and some examples for starting the essential features of the package.

Launch files are XML format files that, using the roslaunch tool, allow you to start the master node (necessary for ROS 1 to work) and multiple nodes at the same time, they can also manage the passing of arguments and the loading of parameters, possibly also by calling additional launch files and thus generating a launch-tree.

Demo

It acts as a root for the tree of launch files that are called up in succession to start all the necessary features (implemented in ROS nodes) in a modular manner and upload all the necessary parameters to the ROS server. Examples of launch files it starts are *move_group.launch* and *moveit_rviz.launch*.

Move Group

Main launch of the entire MoveIt system, starts the *move_group* node and, by recalling the appropriate subsequent launch files, all the functionality (planning, trajectory execution, and so controllers, and sensors) necessary for it.

Planning Context

It loads all generic configuration utility files such as URDF, SRDF and those on joint limits and kinematics.

RViz

It starts the RViz software node by loading the *moveit.rviz* configuration file in which to set the minimum basic settings for use with moveit.

Planning Pipeline

Two launch files: the first one is of service, with the only purpose of separately calling up the launch file of the planning pipeline of the specific planner (in this case OMPL) in a modular way; the second one loads all the parameters proper of the planner useful for planning and the respective configuration file.

Trajectory

Load parameters such as trajectory constraints and call the *controller manager* (or the *fake controller manager*) launch file to load the configuration file of the controllers based on the one chosen.

By default via the argument *fake_execution* in the demo launch file it starts the fake controllers, but for this project this behavior is changed starting the launch file that loads the configuration file for the real ones.

Controller Manager

It loads the controller configuration file to the parameter server.

Fake Controller Manager

Unused: it loads the fake controller configuration file to the parameter server.

Sensor Manager

As for the respective configuration file, this is also inserted only for completeness; it starts and loads the sensor functionality by means of two launch files in a modular way.

6.2 Fetch Pick-and-Place Package

fetch_pick_place is the package that implements all the entities needed for the specific task execution. It will be treated on the basis of the same hierarchical organization used for the MoveIt configuration package, so for this purpose the folders contained in it are listed below:

- launch
- models
- scripts
- src

Each of them will be explained below with attention to the functionality it provides.

It is only anticipated that the actual program, or rather, the programs that actually perform the task, are contained in the scripts and src folders, this because the completeness and versatility of MoveIt provide two interfaces (three if we also consider the RViz plug-in) to control the robot: *move_group_interface* in C++ and *moveit_commander* in Python; consequently the author, to show the approach in its entirety and evaluate the different possibilities, has chosen to implement the task in both programming languages.

In practice, the use of this package involves the launch through the launch-tree of all the functionalities useful for the simulation (or possibly even the actual integration): Gazebo, MoveIt, etcetera; while on another terminal the program (Python or C++) is executed for the actual implementation and execution of the task.

6.2.1 Launch Folder

In this folder the launch files useful for the project are inserted, they are specifically two: the first acts as the root of the launch-tree calling the launchers *move_group* and *rviz* (if required) mentioned above, it also starts the second launch file present.

The latter is useful in order to simulate and control the real robot, it starts Gazebo and all the models (robot and environment) necessary for the simulation, prepares the robot for the execution of the task and loads the configuration file of the controllers to the purpose of interfacing with the simulated/real ones.

Regarding this controller configuration file, a clarification must be made, it is not the same one that refers to the controllers for MoveIt mentioned in the previous section, but it's the default configuration file used by *ros_control*, and that it was written to interface with real controllers, regardless of the use of MoveIt, on the basis of the interfaces included in the URDF for the purpose of integration with Gazebo.

What has been said about the interfaces included in the URDF is generally valid in the vast majority of cases but it is not an absolute rule, in fact in the specific case of the Fetch robot the URDF does not present any adaptation for the use of the model in Gazebo, and the role performed by these components is implemented through an appropriate script that configures the interfaces and takes care of making the simulation consistent with a real execution.

6.2.2 Models Folder

This folder simply contains the models useful for simulating the task, the standard conventionally provides that it refers only to the models of the environment and therefore does not contain the robot model which, when defined in the form of URDF, is present in packages marked with syntaxes like *<robot>_description* provided in ROS support repositories by manufacturers.

The standard format for models in Gazebo is the SDF, an XML type format that already natively supports the complete modeling for the simulator in question without the need for adaptations as for the URDF (for example via the *<gazebo>* tag), in fact unlike the latter which is capable of describing in detail only entities at the "robot level", the SDF is much broader and allows description of models at the "world level".

In the package in question the model SDF represents a parallelepiped that acts as a table and a cubic block (with 5 cm long edges) placed above it (Fig. 6.3), both are then placed in front of the robot when the simulation starts and then when all models are loaded; in writing the file all the physical characteristics of the objects have been implemented to simulate the task making it possible in a realistic way.

6.2.3 Scripts Folder

This is one of the two folders that contain the heart of the task execution, conventionally programs written in scripting languages that do not need to be compiled are placed in the scripts folders, for example in ROS projects in fact nodes and general codes written in Python are present here.

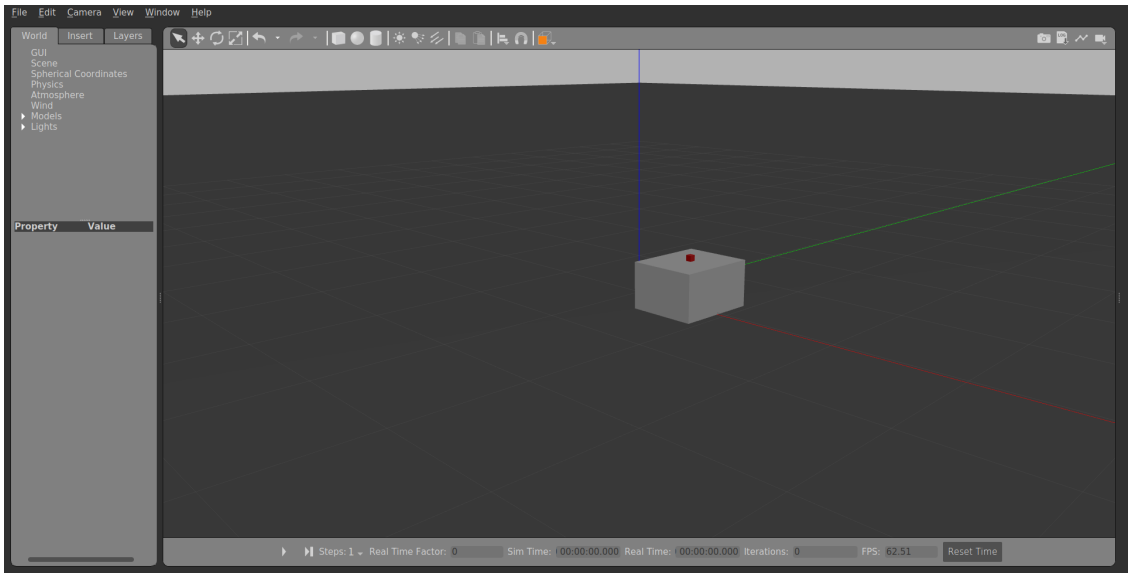


Figure 6.3: Gazebo standard starting window with environment model spawned

In the case of this specific package, the Python script *py_pickplace.py* implements the task using the interface (*moveit_commander*) of the same programming language to interface to the manipulator through the modules provided by MoveIt for the language itself.

6.2.4 Source Folder

For this folder exactly what has been said for the scripts folder is valid (but with the *move_group_interface*), with the only note that it is conventionally used for the sources, and therefore for the programs that must be compiled, such as in this case for the C++ code.

An interesting fact is that for this implementation of the task the author has chosen to use, for demonstration purposes, particular definitions of ROS messages introduced by MoveIt, namely *Grasp msg* and *PlaceLocation msg*. It codes inside, in a single block, all the characteristics of the grasping and the placing, and allows the completion of the task through these two single message which contain all the necessary information.

Chapter 7

Conlusions

7.1 Results

The simulation tests show that the arm is able to carry out the task correctly, grasping the object, performing a post-grasp transport and then repositioning it in the same previous point. To be exact this is what happens with the task implemented in Python, as far as C++ is concerned, a bug with the *PlaceLocation* message for the placing phase does not allow to complete the last phase and therefore the task in its entirety; this is a MoveIt bug that the developers are aware of and for which a fix is planned.

The various phases of the pick-and-place task are shown on Figure 7.1: starting from the initial position of the arm (a) up to a possible post-place retreat (f) following the goal achievement. Furthermore, as a last action, after moving away from the object the arm is programmed to return to its initial position.

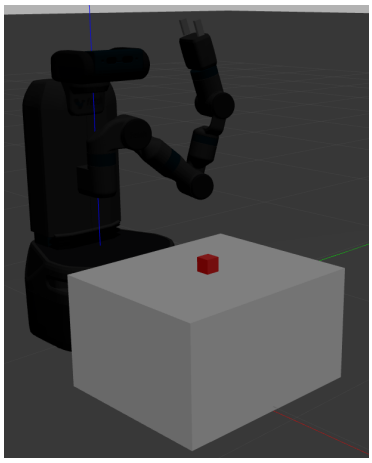
The anomalous color of the robot textures is due to the fact that the manufacturer of the arm have made available COLLADA (COLLAborative Design Activity, an interchange file in XML format for 3D softwares) files generated for a previous version of Gazebo.

In conclusion, through the rqt tool, it is possible to view the graphs of the joint trajectories, both as regards the *FollowJointTrajectory actions* produced by the motion planner and with which the controllers are controlled via the *FollowJointTrajectory interfaces*, but also all the trajectories in general planned by the *move_group node*, consequently, also the gripper actions of type *GripperCommand* for gripper control.

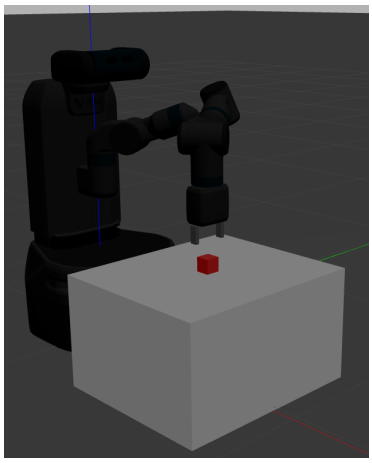
By way of example, the plots of the trajectories of the gripper joints have been produced, having a more representative reading, so a qualitative analysis is made.

In figure 7.2 on pages 47 and 48 it is possible to see the four graphs representing the trajectories of both fingers in both the actions they perform (picking and placing), the abscissas show the time scaled to the duration of the action, the ordinates instead the units of measurement based on the curve that is taken into account in the graph, and therefore for position, velocity, acceleration and effort they are respectively meters, meters per second, meters per second square and newton.

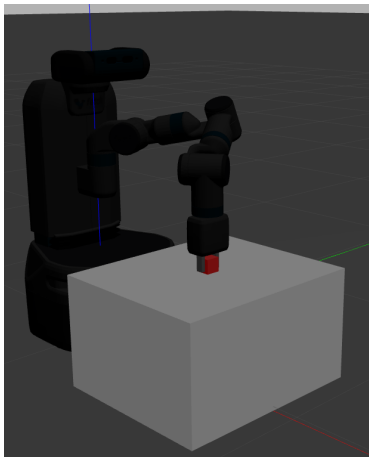
To understand the graphs it is necessary to keep in mind that the versor with



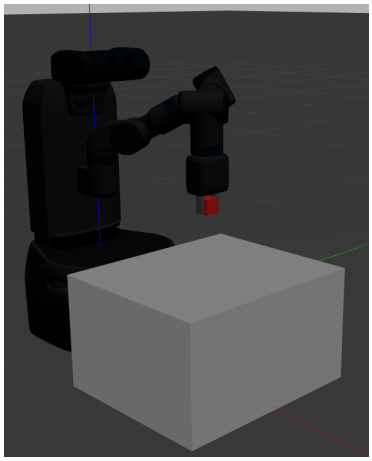
(a) *Initial position*



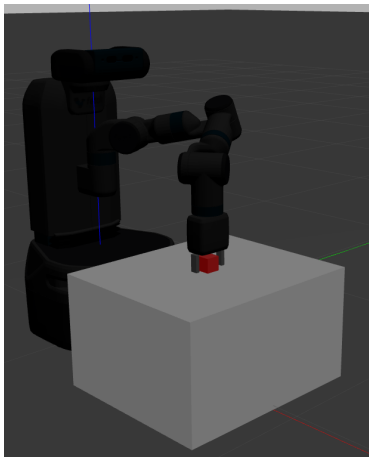
(b) *Pre-grasp manipulation*



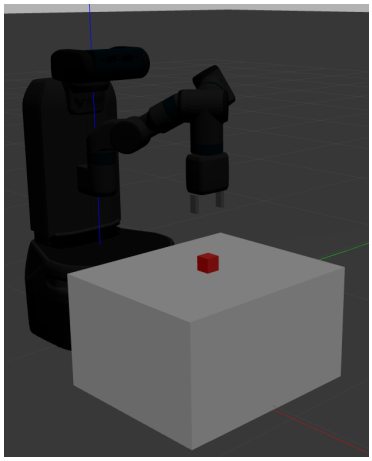
(c) *Pick*



(d) *Post-grasp transport*

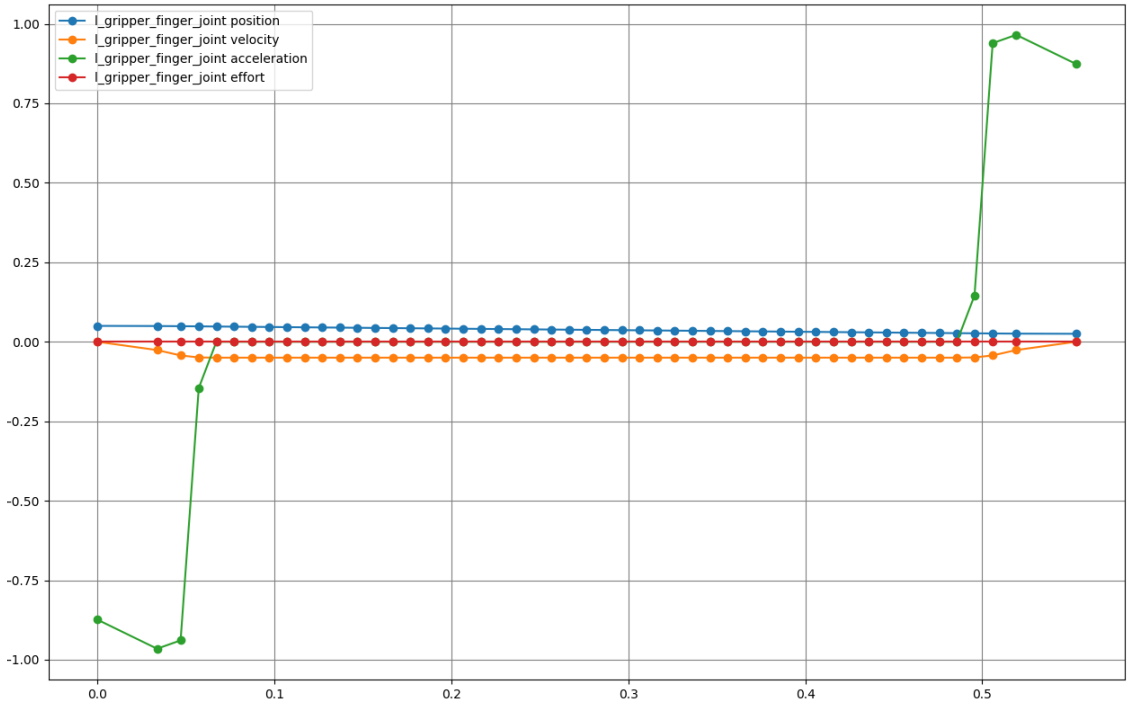


(e) *Place*

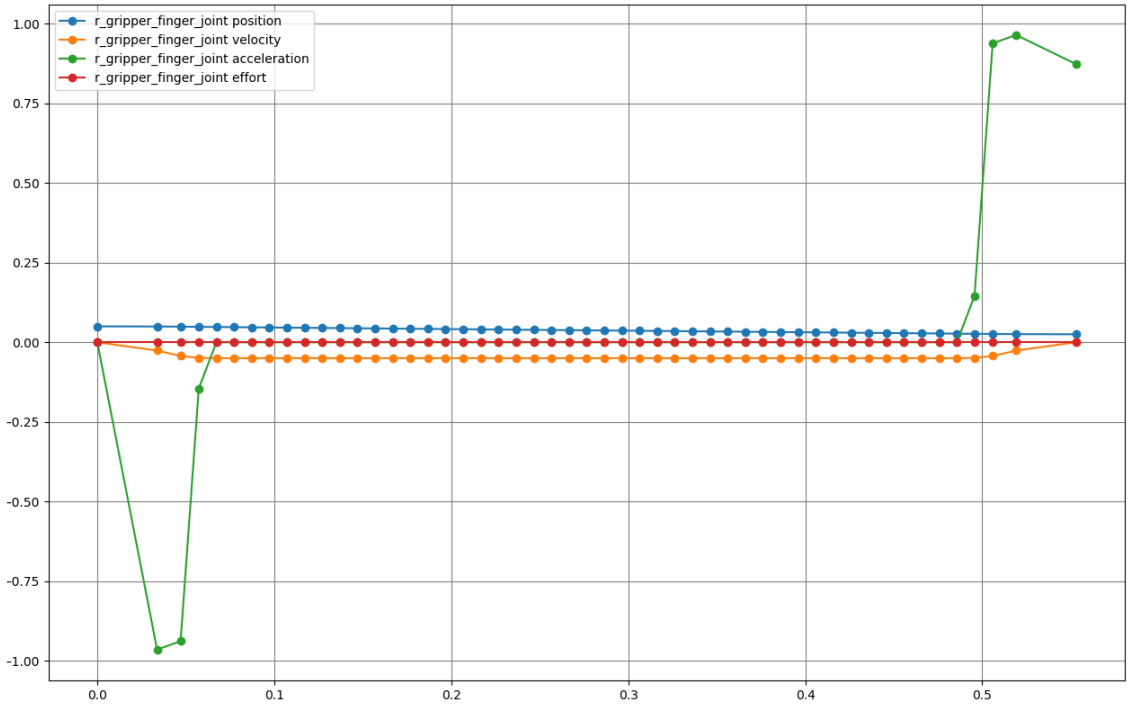


(f) *Post-place retreat*

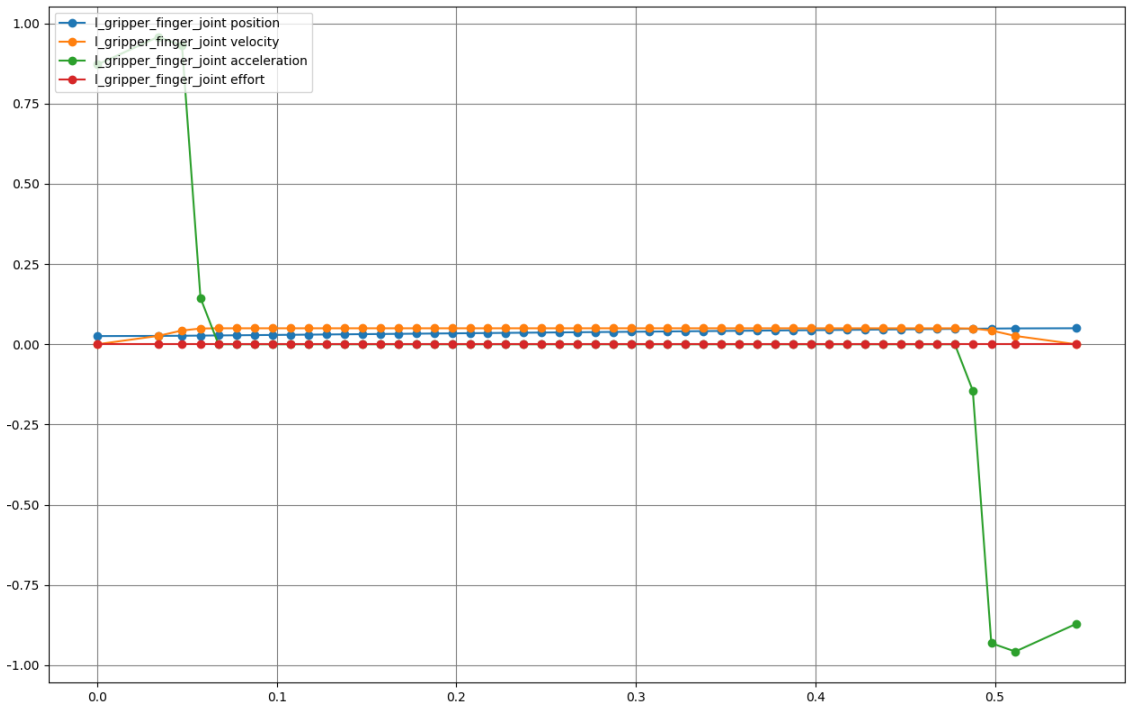
Figure 7.1: Pick-and-place phases in Gazebo simulation



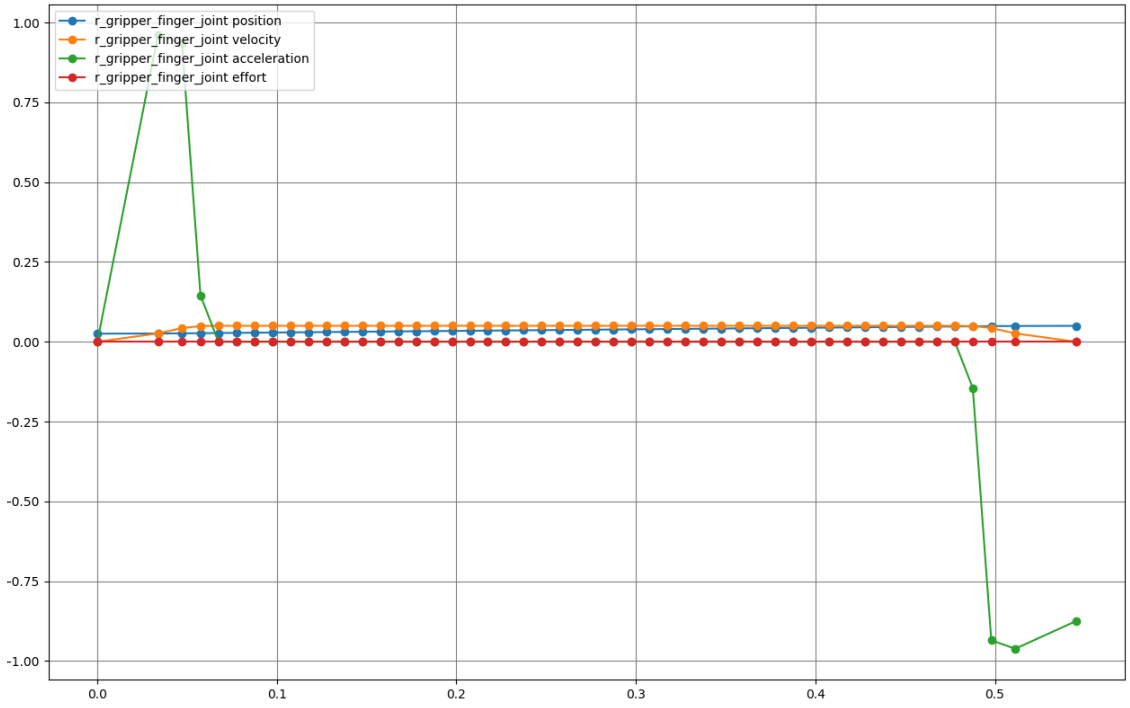
(a) *Trajectory of the gripper left finger joint during picking*



(b) *Trajectory of the gripper right finger joint during picking*



(c) *Trajectory of the gripper left finger joint during placing*



(d) *Trajectory of the gripper left finger joint during placing*

Figure 7.2: Trajectories for left and right fingers joints during picking and placing ($\mathtt{x} \rightarrow$ s ; $\mathtt{y} \rightarrow$ m, $\frac{\text{m}}{\text{s}}$, $\frac{\text{m}}{\text{s}^2}$, N)

respect to which the quantities are defined has a direction that goes from the center of the gripper (position taken by the fingers when the latter is closed) to the outside (position taken by the fingers when it is open instead).

Being a planning of the exclusive trajectory, and issuing a command in order to control only the latter, the effort is always null for the entire duration of the actions in both types.

The most appreciable variation in these images is that of the acceleration, which, in the picking phase, in (a) and (b), rapidly decreases from a zero value to a modulus peak of almost 1 $\frac{m}{s^2}$, to produce a coherent sign velocity which causes the value of the joint position to decrease (joint approaching the center of the gripper); the acceleration immediately returns to zero to make the finger move at a constant speed, until it is about to reach the correct position to lock the object, almost a tenth of a second before it reaches the expected position, the acceleration increases with the opposite sign to that of the velocity causing a deceleration and consequently the stop of the finger.

Once the desired position has been reached, the action is completed and the movement stops without the necessary return to zero of the acceleration.

In the placing phase, in (c) and (d), the same process takes place but in reverse, as the fingers move in opposite directions due to the opposite directions of speed and acceleration.

In Figure 7.3, on the other hand, the focus is on the position and velocity parameters that were difficult to analyze in detail from the previous graphs, and the left finger was chosen as example.

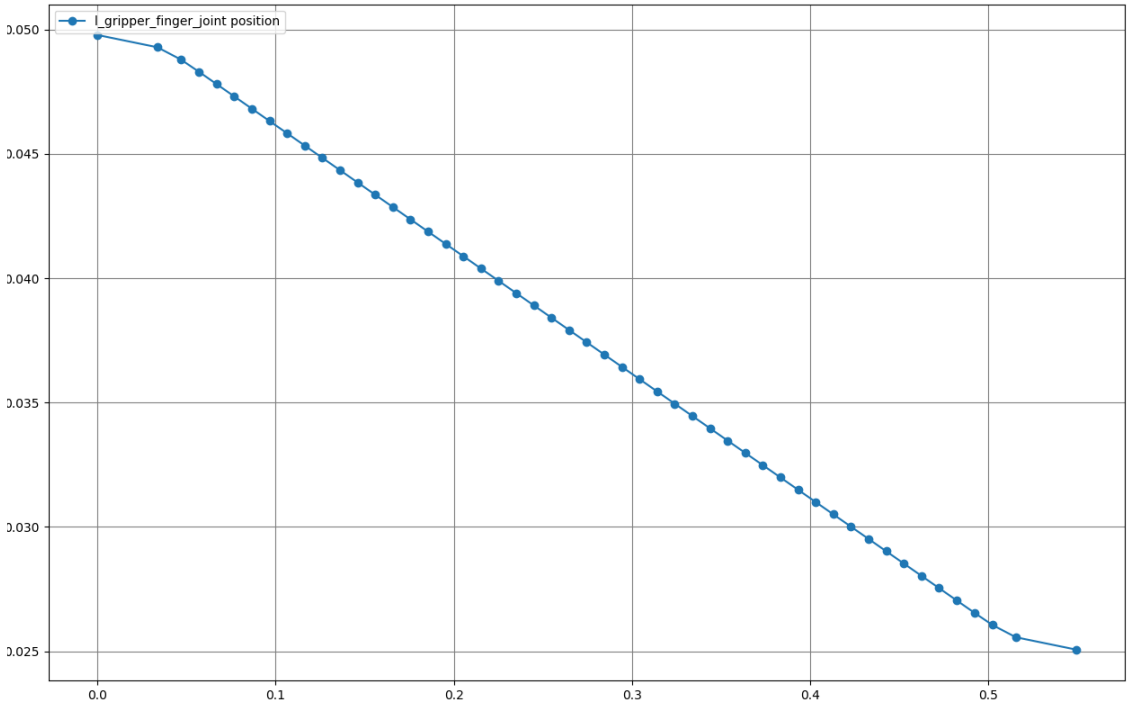
In this figure it is possible to verify what has been said above in terms of positions and velocity, and observe the classic linear and trapezoidal trend of the two, respectively, characterized by similar impulsive acceleration behaviors like those seen above.

Furthermore, in these graphs it is easier to verify the actual movement of the fingers starting from the maximum opening position, both 5 cm from the center (therefore total opening of the gripper equal to 10 cm), and approaching up to a distance of 2.5 cm from the center, which added together give the actual size of the cube grasped, ie 5 cm. The speed instead reaches the constant peak of modulus equal to 0.05 $\frac{m}{s}$, that is 5 $\frac{cm}{s}$; in fact the action is completed in about half a second.

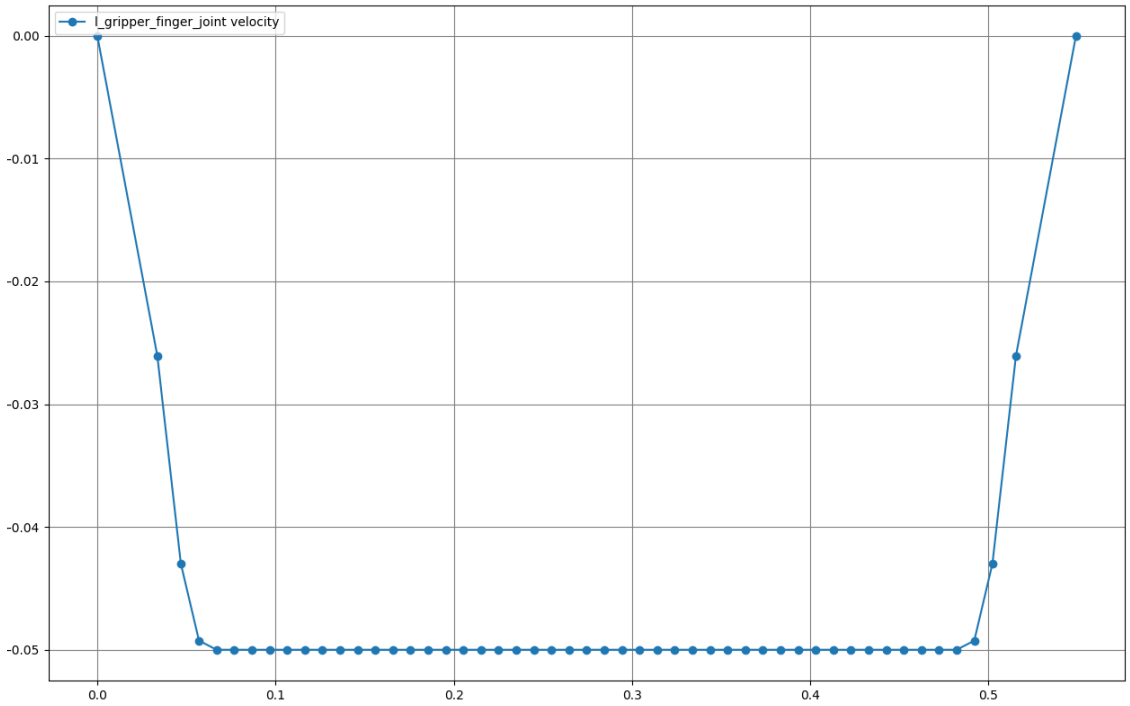
7.2 Future works

This section lists ideas, proposals or advice relating to the approach described in this part of the script, for future works of various kinds, whether they are additions, modifications or improvements.

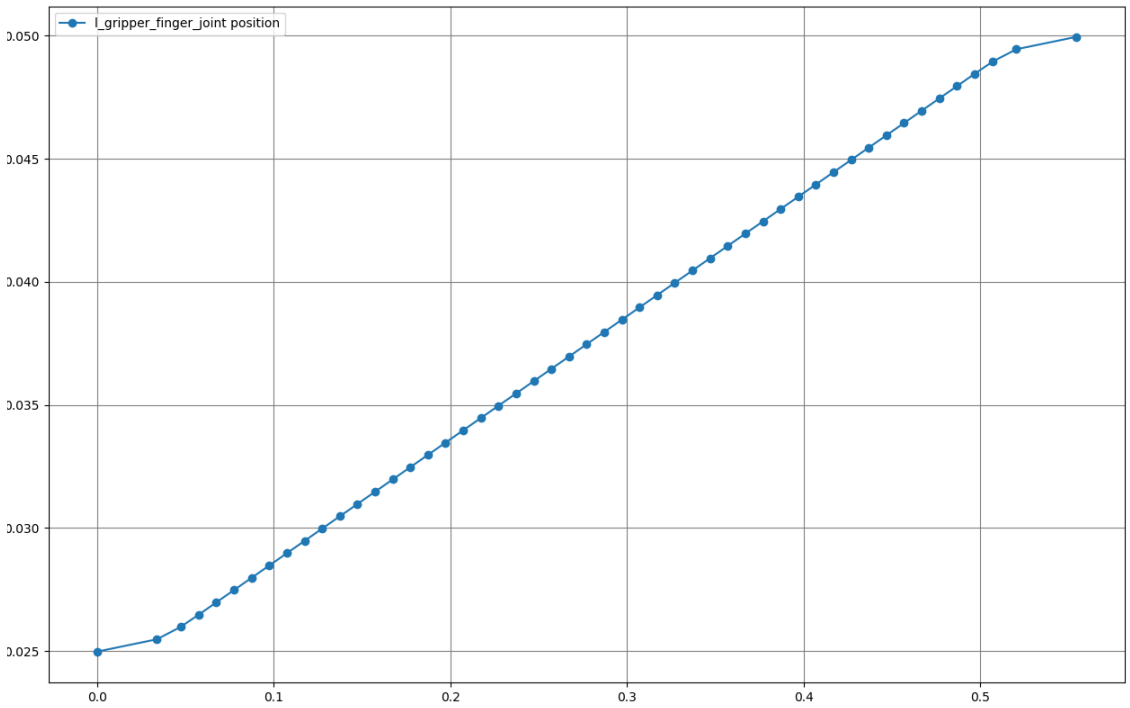
- Wait for the fix of the bug with the *PlaceLocation msg* in the place phase of the pick-and-place pipeline in order to verify the complete task with the C++ interface, or possibly help the developers in the resolution.
- Test the implementation of the task with the *Task Constructor* library of the development version of MoveIt (not the stable one).



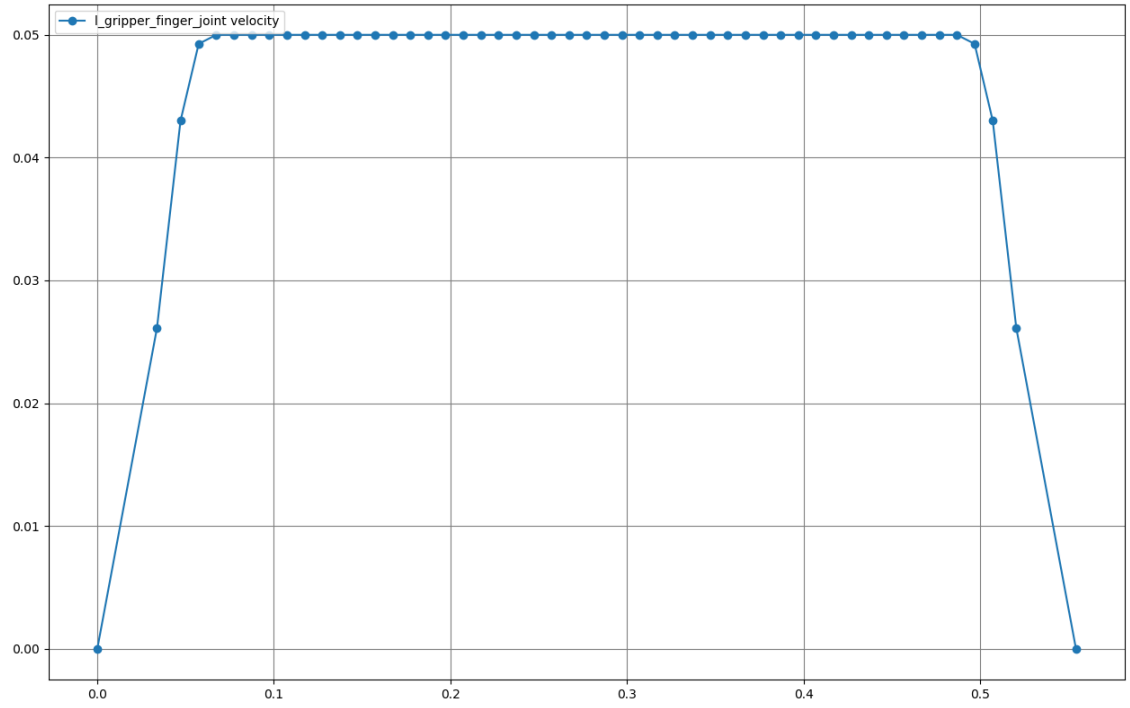
(a) *Position for the gripper left finger during picking*



(b) *Velocity for the gripper left finger during picking*



(c) *Position for the gripper left finger during placing*



(d) *Velocity for the gripper left finger during placing*

Figure 7.3: Positions and velocities curves for left finger during picking and placing ($\mathtt{x} \rightarrow \mathrm{s}$; $\mathtt{y} \rightarrow \mathrm{m}$, $\frac{\mathrm{m}}{\mathrm{s}}$)

- Test the implementation of the task through the *Grasps* library of the development version of MoveIt (not the stable one).
- Integration of sensor modules for RV in order to automatically estimate the pose, shape and size of the object; consistent updating of the code to take advantage of this data and not have to enter it manually.
- Use of the entire robotic system (the entire Fetch Mobile Manipulator) for the execution of more complex pick-and-place tasks that involve moving the arm base.
- Implementation of real-time functionality for task execution in environments with more complex dynamics, such as pick-and-place of moving objects.

Part II

Machine Learning Approach

Summary

In this third part of the script, a non-standard approach will be considered, one that makes use of more modern techniques such as machine learning or specifically reinforcement learning.

The subdivision into chapters is structured with a general introduction to chapter 8, a summary of the useful theory in chapter 9, a detailed description of the state of the art in chapter 10 and final conclusions in chapter 11.

The introduction shows and describes the main paradigms of machine learning, their main applications and implementations on the basis of current techniques, and exposes an overview on other modern methodologies that are beyond the scope of this thesis.

The chapter on theory presents the basic concepts and knowledge for understanding the work of the thesis, starting logically first from reinforcement learning in general and finally showing the transition to deep reinforcement learning techniques.

The tenth chapter talks about the state of the art in the field of deep reinforcement learning applied to pick-and-place tasks, presenting theoretical and practical research works.

Finally, a conclusion is presented in the last chapter in order to show, as in the previous part, results and final conclusions, followed also in this case by ideas and advice for future developments.

Chapter 8

Introduction

This part of the thesis will deal with the topic of AI (Artificial Intelligence) applied to robots, specifically to manipulators.

It should be emphasized right away that the field of AI, and specifically of ML, is a sector in strong and continuous growth, the literature is not always clear in terms and conventions, especially as regards a clear and schematic classification of paradigms, approaches, algorithms, models or methodologies. In this discussion the author avoids purely semantic disquisitions in order to present rigorous classifications that in fact do not exist, but he focuses on the main key concepts, also for this reason it is important to remember that there are no written and precise rules and that often the distinctions among the various entities of this area are more nuanced than one might think.

The most common idea at the moment is to conventionally divide the approaches to ML on the basis of three main paradigms on which algorithms and methodologies are based (Fig. 8.1):

- Supervised learning
- Unsupervised learning
- Reinforcement larning

At the same time, an exhaustive collection of approaches to ML will be presented for a better understanding of the background.

Other modern applications of ML (which however will not be dealt with) besides those mentioned below can be found in the BMI (Brain-Machine Interface) which exploit EEG (electroencephalogram), HMI (Human-Machine Interface) in general that uses EMG (electromyography) or ENG (electroneurography), or in the control through gesture, which in turn can be used to give a simple command or to teach through a *learning by demonstration* approach. Regarding the learning-by-demonstration (or learning-by-showing), as mentioned in 4.1, through the ML it receives a considerable enhancement, for example there is an entire class of reinforcement learning methodologies known as IRL (Inverse RL) that has precisely this among its fields of application.

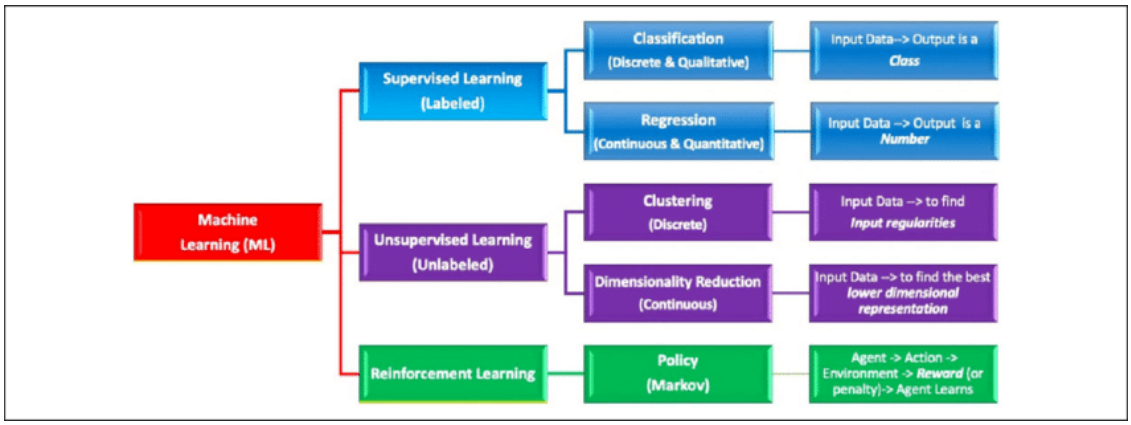


Figure 8.1: Machine learning paradigms scheme [credits: Rashidi et al. (2019)]

8.1 Supervised Learning

The entity that has to learn is presented with an input data set and an output data set, the task is to learn the mapping function (i.e. the relationship) between the two by building the mathematical model that binds them, in this way by providing a new input never seen before, the algorithm must be able to autonomously return the corresponding output.

The main application is the implementation of classification and regression algorithms.

The entity in question is the model and takes different names based on the one that implements the algorithm, examples of models that have relevance in the field of robotics are SVMs (Support Vector Machines), ANNs (Artificial Neural Networks) or more simply NNs (Neural Netowrks) and also *decision trees* and *random forests* (models composed of several decision trees).

NNs are certainly the most used model in the field of ML as a very powerful and versatile tool, lending themselves to multiple applications, they are present in every ML paradigm and advanced uses can lead to the implementation of even more complex models that make use of them, such as in EML (Extreme Machine Learning, i.e. feed-forward NNs in which the node parameters are also set), SOMs (Self-Organized Maps) which are widely used in the field of CV, or CMACs (Cerebellar Model Arithmetic Controllers) which also with the use of simple NNs can be exploited for kinematics and inverse dynamics algorithms.

An interesting application with SOM-type algorithms allows the system to "learn" (even if it is an intelligent management and organization of data rather than real learning) through movements in the workspace, so that given a point it knows autonomously what joint movements to make to achieve it.

It is intuitive how a NN lends itself well to the control (as well as the identification) of systems, a network does nothing but, given input-output pairs, generate a model that relates them, a controller does the same thing, given a reference input it must

ensure that the output of the overall system is consistent with that input.

Concerning NNs, a class of them introduces a subfield of ML known as deep learning (DL), for DL we mean that sector of ML that makes use of particular NN models called DNN (Deep Neural Network); this term basically refers to multilevel NN, this class includes for example RNN (Recurrent Neural Network) and CNN (Convolutional Neural Network). A recent alternative approach to the always more common DNNs makes use of structures known as *deep forest*, a model obtained from the use of decision trees (not to be confused with random forest).

8.2 Unsupervised Learning

The entity that has to learn is presented with a set of inputs without any label, learning consists in finding a structure in these inputs based on common characteristics in order to be able to analyze subsequent inputs.

The main application is the implementation of clustering algorithms in the first place, but also of association and dimensionality reduction.

It is the type of learning that sees the greatest application in the CV.

8.3 Reinforcement Learning

Reinforcement learning is the paradigm that comes closest to the concept of true AI, in this case the entity (generally known as agent, or in some specific areas, not surprisingly, also "controller") interacts with a dynamic environment on which it must learn to perform a task.

The basic principle is the following (Fig. 8.2): the agent acquires the state of the environment, on the basis of the latter and his experience it evaluates the action to be taken to achieve its goal, according to the result of its action on the environment it receives a reward in order to let it understand if that action on that state was positive or not for the achievement of its goal, until the goal is reached the agent resumes acquiring the system state (modified each time by the last action of the agent itself) and repeats the cycle until its behavior progressively improves and learn how to reach the goal.

RL applications range from every sector: navigation (for example self-driving vehicles), control, computer vision, games, chemistry, medicine, etc., and new applications are thought up continuously every day.

It seems clear that a multidisciplinary field such as robotics is one of the sectors that has most benefited from this technology, in this context the agent can be practically anything: the controller, a model that learns to perform inverse kinematics/dynamics operations, or the robot itself which as a complete system exchanges data (for example through exteroceptive sensors) with the external environment, carrying out actions on it in order to complete a given task (for example the pick-and-place).

The algorithms for the implementation of an agent in the RL are various and their choice depends on the needs, but also in this case the NNs lend themselves as a powerful

tool for the implementation of some of them, in a general view they are useful for example to map states-values or states/actions-values, learn to predict how valuable an action will be for the achievement of the goal or recognize a state when the input is visual.

Also in this case, the use of deep learning leads to a class of reinforcement learning known as deep reinforcement learning (DRL).

The RL (but specifically the DRL) is the paradigm selected by the author for completeness, research interest and usefulness for the purpose of the thesis.

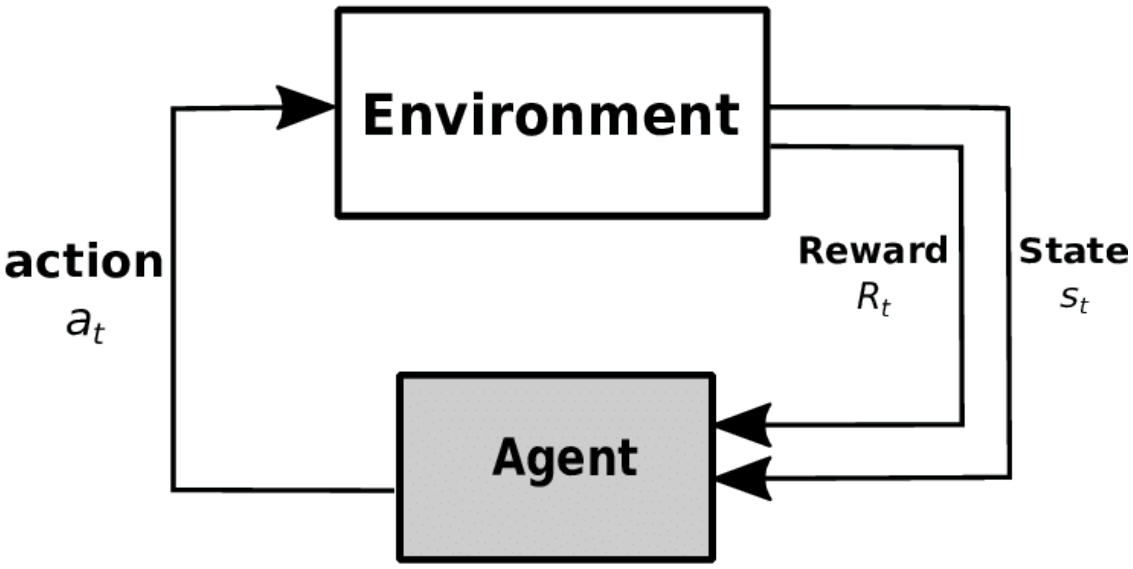


Figure 8.2: Reinforcement learning general block diagram
[credits: Amiri et al. (2018)]

Chapter 9

Deep Reinforcement Learning Theory

The chapter in question does not aim to replace a theoretical book on RL or specifically on DRL, however it wants to present itself as a tool to better understand the concepts covered in the thesis, defining the necessary basis for those who do not have them, or acting from recap for those who already have knowledge on the subject.

The theory discussed concerns RL learning in general, while DRL specifically will be treated in the final section.

For a better understanding it is also specified that in this context the term "method" is equivalent and interchangeable with that of "algorithm", consequently the different types of methods presented from here to the end of the chapter, refer to the same different types of algorithms.

9.1 Key Elements

In this section the key elements (Fig. 9.1) of a RL problem will be presented starting from the two basic ones: *agent* and *environment*; although we can intuitively think that it is more logical to start the discussion from the concept of agent (already mentioned in the introductory chapter) as the main element, that of environment will be presented first, having some characteristics that make an explanation of the agent itself clearer.

In the various subsections, hints will be given to different methods that describe different approaches to a RL problem, they will not be explored in this section but will have references to the next one (9.2) in which they will be treated in more detail in view of an adequate classification.

9.1.1 Environment

The environment is all that is outside the agent, the element with which the latter interacts and in which it must achieve is goal.

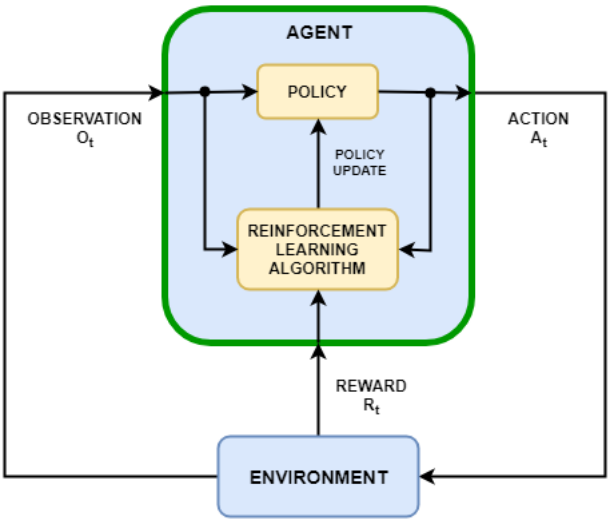


Figure 9.1: Key elements of a Reinforcement Learning problem
[credits: mathworks.com]

In a general view it represents the implementation of the RL problem that one wants to face: it defines the goal both indirectly through *reward signals* only and directly in the cases of *Multigoal RL* (goal-based environments); it defines a *state space* for its description and an *action space* for interactions with it; according to the problem it implements it can also be described by a mathematical model.

States Spaces

In the literature it is sometimes possible to find distinctions between state and *observation*, if we want to make this subdivision, state means the set of data that describe the complete configuration of the entire system environment, while observation refers only to the parameters of the state that indicate characteristics useful for the purposes of the problem and observable by the agent, in fact there may be factors relevant to the problem but which are not observable by the agent, they are not included in the state and are considered *noise*. However, both the observation and state concepts refer to instantaneous information about the environment.

Subsequently, the common choice to conventionally refer to observation with the term "state" will be made.

The set of possible states defines a state space; on the basis of the complexity of the problem, and consequently the complexity of the environment, this space can be more or less large (potentially infinite) as regards its size, and be defined *discrete* or *continuous* as regards the variation of the states themselves.

The size of the states space (together with that of the actions) determines the methodology for dealing with the problem: *tabular solution methods* or *approximate solution methods* (Subsec. 9.2.1).

Actions Spaces

Although the actions are interactions performed by the agent on the environment, they are defined on the basis of the problem and consequently it is the environment that defines them.

For example, in a "chessboard" environment that implements the problem of a chess game, it is the environment itself that defines the possible actions for the agent, the latter cannot act through actions that are not foreseen and accepted by the environment (such as actions that can range from a simple illegal move to any other imaginable action that has nothing to do with chess).

As regards the concept of actions space, the same considerations made for that of states apply; also concerning its size, that one of the actions space (as mentioned for the state space) determines the methodology for dealing with the problem (Subsec. 9.2.1).

Reward Signal

Through this signal the environment defines the goal (required by the problem) of the agent. Whenever the agent carries out an action on the environment, the latter returns him a reward through which the agent himself understands whether a given action in a given state was positive or negative for the purpose of achieving its goal.

Based on what has been said, the reward is a function of both state and action, as it evaluates a given action in relation to a given state.

Model

The model is a mathematical representation of the environment that describes its dinamics, and behavior in general, based on how accurate it is; the knowledge or not of the model depends on the problem being treated, it therefore represents more information that is not always possessed, the more complex the problem is, the more articulated the environment will be, and consequently even more difficult its mathematical modeling.

The presence of a model greatly facilitates the learning of an agent and therefore the resolution of the problem, it can be used to obtain precise predictions (and not approximate estimates), for example on the next state of the environment given the current state and the action performed on it.

Two classes of methods have been introduced with their respective algorithms, to manage cases in which the model is present (even generated by the agent itself) or not: *model-based methods* and *model-free methods* (Subsec. 9.2.4).

9.1.2 Agent

The agent, as already summarized in the introductory chapter, is the entity that acts over time on an environment in order to achieve a goal. The agent initially does not have any information on how to perform its task, therefore it needs a learning phase

(training) in which, based on the experience that matures over time, it can learn how to do it.

For this purpose the agent acts according to two different behaviors, both of which are necessary for complete learning: *exploitation* and *exploration*; the first behavior is necessary to exploit the most of what it has already learned, but consequently also the second is essential to learn new things without being "tied" only to what it already knows; it goes without saying that it is necessary to obtain the best trade-off between the two behaviors.

The agent observes the state of the environment and takes an action chosen on the basis of rules defined by a *policy*; in turn these rules are (or rather, can be) perfected thanks to functions that evaluate them: the *value functions*.

Policy

It defines how the agent should behave, mapping the observed states of the environment with the action to be taken in them. In a nutshell, it is a rule by which the agent selects the actions according to the states, it can be *deterministic* if it assigns a precise action to each state, or *stochastic* if given a state/action pair it returns the probability of carrying out the given action in the given state.

Each RL problem has one or more (if they are qualitatively equivalent) better solutions than all the others, this in practical terms translates into the concept of *optimal policy*, which refers to the policy (or to more policies if there are more than one) which maximizes the agent overall reward.

The different use of policies has led to the distinction of two types of methods (or algorithms), those *on-policy* and those *off-policy* (Subsec. 9.2.3), in addition the different implementation of synergistic approaches between policy and value function allow a further methodological classification between *value-based methods* or *policy-based methods* (Subsec. 9.2.2).

Value Function

A further element that helps the agent to learn is the value function, its usefulness is to be discovered in the agent intention to maximize his overall *return*, in fact, while the reward represents an immediate feedback, return refers to a long-term reward. On the basis of what has been said, a function that predicts this expected return is useful, making an estimate of the "quality" of a state or a state/action pair.

It is called *state-value function* if it estimates the value of states or *action-value function* if it estimates the values of state/action pairs. The value function is not an absolute function, but is defined with respect to a given policy, it is estimated from experience and allows to evaluate which policy is better than others.

With reference to what has been said for the policy, for each RL problem there is one and only one *optimal value function* that refers to the optimal policy (or to the entire set if there are more than one).

Various methodologies revolve around the concept of value function to approach RL problems. The first and the greatest classification is that, already mentioned,

between *tabular solution methods* and *approximate solution methods* (Subsec. 9.2.1); moreover, as mentioned talking about the policy, the value function is also linked to a classification based on its use (or possible use) with the policy itself (Subsec. 9.2.2).

9.2 Methods Classification

This section is essential to frame, in a more possible general vision, the entire background on RL, as it shows all the classification terms of each method that deals with RL problems.

According to the problem to be faced, there are an ever increasing variety of algorithms, they can also differ greatly from each other.

Subsequently, the methods present in the same subsection are obviously mutually exclusive, however for each subsection each algorithm will always present one of the two methods shown; the different methods used in different combinations for the implementation of an algorithm, define the overall approach that has been chosen to use for the given problem.

9.2.1 Tabular or Approximate Solution

This is the largest classification of approaches to RL, it directly depends on the implementation of the approximate value functions (approximated because of complex mathematical and computational tractability), and therefore indirectly on the dimension of the states and actions spaces, since as mentioned previously, these latter define the complexity of the problem, and depending on their size it is possible to represent the value functions in simple data structures (tables) or or it may be necessary to approximation further by means of more complex structures (parameterization).

The concept of approximation is however relative, and this depends on the fact that regardless of the method used, the approximations are equally necessary (whether they are for value functions, for policies or even for the models themselves), even if a complete model of the environment is available, other constraints such as computational or memory constraints in any case force an approach based on approximations, so often the optimal solution cannot be found but must be approximated.

Tabular Solution Methods

These are methods indicated in cases where spaces of actions and states are contained (tendentially but not necessarily discretes), in these favorable conditions the value functions can be represented by means of simple data structures such as tables, or even vectors.

This approach is used to tackle the simplest problems, tend to be useful for research purposes, applications with agents in exclusively simulated environments or for agents implemented in playful environments, it is therefore not very suitable for facing real problems.

Due to the simplicity of the problem, these methods are the only ones that in theory can lead to obtaining the optimal solution, that is, the optimal policy achieved by means of the optimal value function; however, as already explained, technical constraints can present an obstacle also in this case.

Approximate Solution Methods

The methods described here are suitable for problems with large spaces of states and actions (generally but not necessarily continues); the value functions are more complex and their representation is not possible through simple data structures, consequently their parameterized representation is necessary.

These methods are used to deal with the most relevant types of problems and above all with real applications, as it is difficult in the real world to face environments with a small number of states and available actions, but the combinatorial space of them tends to be very vast; on the other hand, with these problems it is not possible to find the optimal solution, so we must aim to obtain the best possible one.

The key concept in this type of approach is that of *generalization* (by virtue of the enormous quantity of different data that cannot be treated individually) which, in combination with the known RL concepts, allows the implementation of techniques suitable for the most complex problems. The concept of generalization is implemented by introducing another paradigm of ML, namely supervised learning: through models (for example NN) and knowledge belonging to this sector that allow to implement approximations of functions in an adequate manner.

9.2.2 Value-Based or Policy-Based

The difference in these two methods lies in the use of the value function and in the different implementation and management of the policy, in the event that an approach that does not use the first is chosen.

Value-Based Methods

The algorithms belonging to this category are those that necessarily require the value function, through it they estimate the value of the actions in order to improve the policy; without the value function the policy (for the most part deterministic) would not even exist.

This approach is the one used in the tabular solution methods, but it is also present in the less complex algorithms belonging to the approximate solution methods.

The *action-value method* is the most widespread of this class, working with value functions of state/action pairs.

Policy-Based Methods

This type of methods allows the agent to learn a parameterized (mostly stochastic) policy that selects actions without consulting a value function, the latter can be used

to learn the parameters of the policy but does not play an essential role in the choice of actions.

This approach is used exclusively with approximate solution methods.

The *policy gradient methods* are the most widespread policy-based methods and base their operation on the concept of policy gradient (PG), used to define the "direction" in which to vary the parameters of the policy in order to improve it, as the gradient of a function shows how the function varies as its variables change. In turn they are divided into *stochastic policy gradient methods* and *deterministic policy gradient methods* (9.3.3).

9.2.3 On-Policy or Off-Policy

These methods concern the implementation approach for obtaining the trade-off between exploitation and exploration.

On-Policy Methods

This type of algorithm evaluates and improves the policy used to make decisions, thus learning the values of the actions not for the optimal policy but for one close to the optimal that always allows at least a minimum of exploration.

Off-Policy Methods

Algorithms of this type evaluate and improve a policy different from the one used to make decisions, consequently they present two policies: one that learns and tends to the optimal one (deterministic), and another one used to make decisions and therefore allow exploration (stochastic).

9.2.4 Model-Based or Model-Free

A subdivision, based on the use or not of a mathematical model (Subsec. 9.1.1) that describes the environment, distinguishes two different methods.

The possibility of modeling the environment depends on the complexity of the environment itself, and therefore on that of the RL problem.

With a model, state values alone are usually sufficient to determine a policy, while without a model it is also necessary to estimate action values.

Model-Based Methods

Model-based methods refer to problems in which modeling the environment is possible, and consequently, to the use of algorithms that either make use of this information if it is available, or generate the model themselves (the agent learns the model such as it does with the policy); they are complex algorithm that use a *planning* approach.

One of the advantages of this approach is the reduced dependence on the environment, which, in the case of a complete and accurate model, can even become superfluous; another advantage is the versatility of the agent, small changes on the required goal result in small changes of the model, and an agent trained on the previous model needs less retraining to be efficient on the new one.

Having an accurate model of the environment, which describes all the dynamics, is certainly an advantage to obtain the optimal solution, but it must nevertheless be remembered that it is not sufficient in the case of problems with a large number of data, due to computational and memroy constraints.

Model-Free Methods

These methods deal with complex problems in which the environment cannot be modeled or, if it were, the approach would not be convenient anyway; they tend to be characterized by a *trial-and-error* approach, precisely for this reason, although the absence of the model represents something less, algorithms of this type are simpler.

9.3 Deep Learning in Reinforcement Learning

This final section of the chapter deals with the conceptual leap between traditional RL and DRL: first it shows a fundamental element of ML in general, the ANN; then it is understood how this element evolves to cope with more complex problems, developing a new sector of ML, the DL; analyzing the improvements that the DL brings in the unsupervised and supervised paradigms, the latter is finally introduced as a support to the RL.

In summary, the DRL arises from the need to exploit the supervised paradigm together with the traditional approaches to RL to tackle complex problems through the approach described by the approximate solution methods, specifically the modeling complexities require an additional effort to the supervised paradigm, requiring more than the its standard approach, than that of the DL.

9.3.1 Artificial Neural Networks

Artificial neural networks (which will later be referred to simply as NN) are nothing more than generic computational models based on the real biological neural networks of the brain in the animal kingdom, they model entities called *neurons* which have the purpose of emulating in a greatly simplified manner the tasks of biological neurons.

The operating principle of a NN is very simple (the complexity lies in the adaptation of this principle to each application case): the network is divided into layers composed of several neurons, the first is called *input layer*, and consists of the first level of neurons that takes the data as input, then there is an intermediate *hidden layer* and finally an *output layer* represented by the last level of neurons.

Each neuron of each level can be connected to one or more neurons of the next level (obviously with the exception of those present on the output layer), and in turn by one

or more neurons of the previous level (except those of the input layer); a network in which each neuron is connected with each neuron of the previous level and each neuron of the next level is called *fully connected network*.

A neuron can have multiple inputs but the output is always only one, even if it is eventually sent to multiple neurons; each connection has an assigned *weight*, through a function called *propagation function*, each neuron calculates its output as the weighted sum of each of its inputs (based on the weight of the connection) plus a *bias* term.

However, as described so far, the neural network is only capable of modeling linear behaviors, as expressed as combinations of linear functions (the propagation functions), for this reason each neuron also has an *activation function*, to which the result of the propagation function passes before it is output. Activation functions in short are none other than non-linear functions belonging to certain classes of functions that allow the network to learn non-linear relations.

On the basis of what has been said, the intrinsic nature of function approximator of a NN is evident, and this is one of the basic principles of supervised learning, as well as the basic principle on which RL approximate solution methods are based.

Unlike traditional programming, in which inputs and functions are possessed in order to obtain outputs, machine learning is an approach that aims to identify mathematical models, consequently, as regards supervised learning, network inputs and outputs are known but not the "function", ie the mathematical model that binds them, the purpose of the network is in fact, following a training phase, to return a mapping function between inputs and outputs.

The training phase, in the supervised paradigm, having the purpose of identifying the model that binds inputs and outputs, translates into the learning of the network parameters, ie weights of connections and bias terms; starting from prefixed parameters, the network adapts them to each iteration on the samples, calculating the errors between the output of the samples and the output obtained and, in turn, tuning the parameters through back propagation processes so that they suit the optimal ones.

The clarifications on supervised learning, in the two previous paragraphs, are to be found in the fact that in unsupervised learning, as mentioned in the introductory chapter, the outputs are not given and network learning consists precisely in modeling the relationships between the inputs, rather than approximating a function that binds two distinct sets of data.

So far we have only talked about three-level networks, however NNs of this type are not capable of implement any complex mathematical model, for this purpose it is necessary to design multi-level structures, called multilayers NNs, with varying numbers of hidden layers. The discussion made up to this point also applies to this type of networks, however their management and their application field falls within that of deep learning; indeed, DNNs are used in the most complex supervised learning and unsupervised learning algorithms, but only the first ones have direct applications to RL, in this case DRL.

Regarding deep learning: so far it has been assumed that in the connections between neurons there are no loops, a network of this type is defined as feedforward neural network, however there are also networks in which this is not true and the output of a

neuron can become input for itself or for neurons of previous layers, in this case it is called recurrent neural network.

Figure 9.2 below shows a general fully connected feedforward deep neural network with three hidden layers.

Feedforward NNs are the main topic, as well as the tool, in the whole discussion on the DL applied to the policy gradient methods (Subsec. 9.3.3).

As mentioned in the introductory chapter, there is another type of DNN, known as CNN, unlike the standard NNs these are specifically inspired by the human's visual cortex, in fact, it is no coincidence that they see their main application field in image processing; an example of an algorithm that makes use of this powerful tool is discussed in Par. 9.3.2.

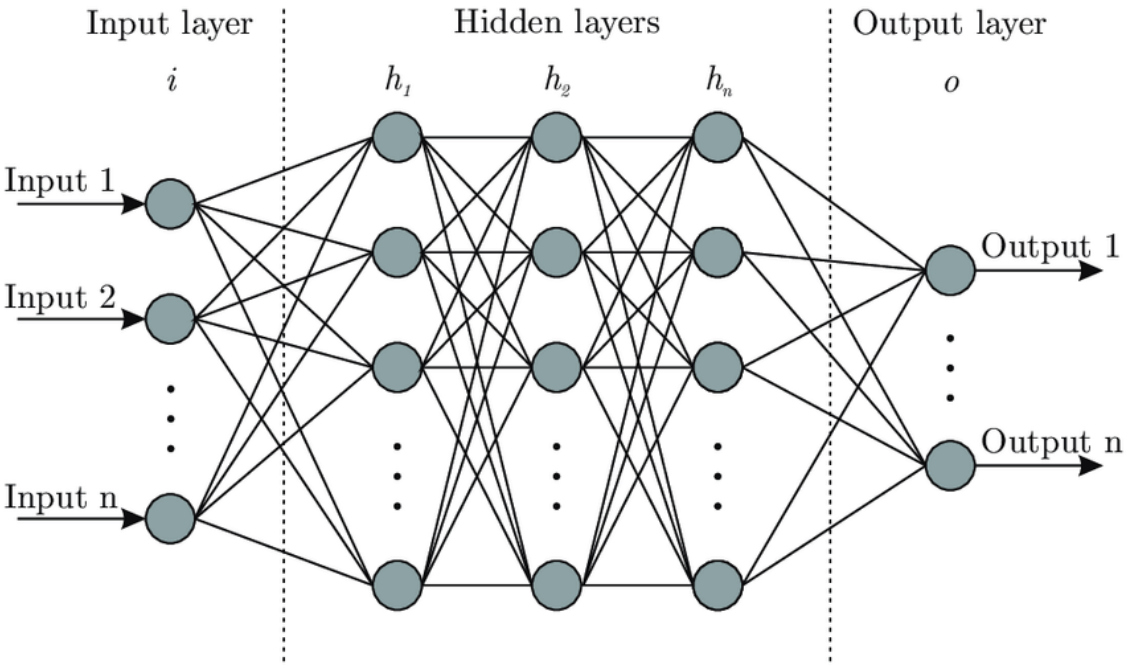


Figure 9.2: Generic fully connected feedforward deep neural network [source: Bre et al. (2017)]

9.3.2 Deep Learning in Action-Value Methods

Action-value methods have already been dealt with talking about value-based methods, and as already anticipated, they are used both with tabular solution and approximate solution.

In this part of the section on deep learning, an action-value method belonging the approximate solution methods, and which therefore exploits the supervised paradigm through a "deep" approach, is explored: the DQN (Deep Q-Network).

DQN The DQN is an off-policy algorithm that has established itself in the RL sector applied to the videogame field, in fact, the use of a CNN network to model the Q function (an approximate estimate of the action-value function) is due to the inputs purely visual it receives. With this algorithm it is faced the problem of using non-linear function approximators that cause instability, the latter is due to the fact that small changes in Q determine considerable changes in the policy; to solve this problem, the concept of *experience replay* is introduced for the first time, a technique that re-evaluates past actions in a random way rather than relying solely on the "best" ones at the moment.

9.3.3 Deep Learning in Policy Gradient Methods

This part of the section on deep learning initially deals with SPG (Stochastic PG) methods, which present stochastic policies and are also on-policy, giving some examples (the REINFORCE and the family of Actor-Critic methods); their evolution is then shown in DPG (Deterministic PG) methods, with deterministic policies and the characteristic of being on-policy, which are more suitable for continuous control (vanilla DPG and in conclusion the DDPG are treated).

Each of the algorithms mentioned has various variants with which they have been improved, but in this theoretical summary, wanting only to describe their basic characteristics, they are only described in their vanilla form, i.e. the original one.

Stochastic Policy Gradient

SPG methods are on-policy policy-based methods with stochastic policy, they are treated as a starting point for DPG algorithms, historically the first algorithm of this type is the REINFORCE; its evolution in the use of the value function has led to the Actor-Critic family of methods, "family" as there are more variants (A2C is the basic one) and some DPG algorithms such as DDPG also derive from this family of algorithms.

REINFORCE REINFORCE is an on-policy algorithm characterized by a single DNN used to learn the policy, its simplicity consists precisely in this, in fact it is the most "pure" example of a policy-based algorithm, as it exclusively exploits the concept of gradient to update the network and consequently the policy based on its experience, without making any form of estimation or prediction on the values of its acrions, in other words without using any value-function.

Actor-Critic Actor-Critic (A2C) methods learn both the approximation of the policy and that of the value function and to this end are implemented through two neural networks, one known as the actor that determines the learned (stochastic) policy and the other called critic that implements the learning of the value function (generally state-value). These methods are PG methods, consequently the critic is an element that improves performance but is not essential for learning, without the implementation

of the critic we would have the simpler REINFORCE algorithm, in fact conceptually the A2C is nothing more than a REINFORCE combined with DQN which has the characteristic of having a value estimator. Despite the validity of the A2C algorithms, they are not born to face continuous problems, but rather discrete problems, and it is for this reason that alternative methods such as DPG and DDPG have been studied.

Deterministic Policy Gradient

DPG methods unlike A2C and SPG in general, are a class of methods that, as the name implies, use a deterministic and non-stochastic PG, the main difference with the latter in fact consists in being off-policy algorithms with the actor that learn a deterministic policy. The main limitation of the vanilla DPG turns out to be the linear function approximator, in fact its improvement consists precisely in combining this standard method with the DQN, equipped with a non-linear function approximator based on a DNN, and obtaining the DDPG.

Deep Deterministic Policy Gradient DDPG[11] is one of the main examples of application of the DL to the RL, and therefore of the DRL, as well as the main one when dealing with complex tasks such as continuous control with huge states and actions spaces: it is an algorithm belonging to the classes of approximate solution, policy-based (specifically PG and, more, DPG), off-policy and model-free methods and derives from the A2C (Actor-Critic) family of methods as a union of two different algorithms, the DPG and the DQN. Unlike the A2C predecessors, this method has two other networks besides the actor and the critic, two target networks that converge to the first two in a delayed manner, increasing the stability of the algorithm, and also, like DQN, it uses experience replay.

Chapter 10

State of the Art

Beyond the state of the art described in the introduction, here we talk about the specific one for this work, and in this case, unlike the previous part, it describes an approach that is a combination of theoretical research and, in practice, implementation of algorithms, development of environments and toolkits as well as use of specific softwares.

10.1 Multigoal Reinforcement Learning

By multigoal RL we mean those frameworks that describe problems in which an agent is required to learn more goals, consequently, in addition to the state, the goal is also an input for the agent; the difference with single-goal case is evident.

An example, to make the difference easy to understand, may be to compare an agent who represents a robot who must learn how to travel a complex path made up of several obstacles by doing as much road as possible, in this case the goal is unique; using the work of this thesis as an example, a multigoal environment can be one in which a robotic arm has to pick-and-place an object in which, however, the position in which to place it can be any random, so the goal is not always the same, because it changes according to the desired position for the object.

10.1.1 Dense and Sparse Rewards

Although the first type of the previous example is very useful for research purposes, the second type is the one that more faithfully represents most of the real scenarios, in which an agent is required to reach more similar goals and in which each attempt returns an affirmative or negative result on the achievement.

What has just been said leads to distinguish two of the main types of reward signals: dense (or shaped) rewards and sparse rewards.

While dense rewards require rather complex reward engineering work, to describe a signal that numerically indicates how valid a given action was, in a given state, in order to reach the goal, the rewards that by nature apply well to multigoal contexts

and how the latter follow the logic of real scenarios (especially in robotic applications) are the sparse ones.

Sparse rewards are mostly non-positive or negative rewards depending on the case, where the value is "positive" (from a semantic point of view, not numerical) only if you are close enough to consider the task performed on the basis of a threshold pre-established; the most widespread example in the literature, also as regards practical utility, is that of binary sparse rewards (-1 if the goal has not been reached and 0 otherwise).

The type of reward, for what has been said, turns out to be a natural consequence of the type of problem to be faced, however sometimes the choice of a reward signal for a given problem can be forced for implementation reasons or as a workaround.

Dense rewards can often be complex to implement but they are easy to deal with, as the agent, after each action, through the reward has significant information on how well (or badly) it has acted, consequently they can be used as workarounds (but this is not always feasible and requires a shift in complexity in the problem) to circumvent the difficult treatability of problems that are naturally described by a sparse reward logic, this is because through them the agent has no direct information about the "quality" of its action, as it can only understand if the goal has been reached or not.

10.2 TensorFlow

From the implementation point of view, TensorFlow (TF)[12] is certainly the state of the art not only for RL but for ML in general; it is much more than a software library, but rather a complete platform (moreover open source) for ML, developed entirely by the Google Brain section of Google.

Through TF it is possible to implement, develop and train different ML models in order to build and deploy complete ML applications through a single tool.

Among the advantages of TF we find:

- be cross-platform (including the Android OS);
- hardware support for CPU, GPU and the TPU (Tensor Processing Unit) designed by Google specifically for TF;
- an active community;
- constant development and updates;
- availability of official APIs for various programming languages (Python primarily for completeness and ease of use, but also C++, Java, JavaScript, etc.) and third-party APIs (for many other languages);
- different specific application sectors such as mobile, IoT or manufacturing;
- being the main choice made in research.

TF also has the *agent* library, a very young library (still not widely used but destined to become the new standard) structured to favor the development of models and algorithms for RL

10.3 OpenAI

OpenAI, which boasts the well-known face of Elon Musk among the founders and investors, is the world's leading organization (moreover non-profit) on research and development in the field of artificial intelligence.

All the analysis on the approach to the task treated in this document through machine learning algorithms, as well as the comparison, objective of the thesis, have been made possible thanks to the work and research of this company that has revolutionized the state of the art in the field.

As for the state of the art referred to, there are two papers that have made it possible to solve complex problems that were previously unsolvable: Marcin Andrychowicz et al. *Hindsight Experience Replay*. 2018. arXiv: `1707.01495 [cs.LG]` and Matthias Plappert et al. *Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research*. 2018. arXiv: `1802.09464 [cs.LG]`.

In full "open style", as well as for publications, the company freely makes the fruits of its work available, specifically it has updated two of its previous products with the addition of what was developed and produced in the context of the aforementioned papers: the Gym toolkit (the main historical contribution of OpenAI) in which the new types of environments implemented were introduced, and the Baselines repository, in which the new developed algorithm was added in addition to the rest of the code used in these works.

10.3.1 Gym

Gym[2] is the first product with which OpenAI has established itself to the public in the field of RL research, it is a real gym that has the purpose of training agents to facilitate their development and comparison.

It consists of a library that, in addition to providing a variety of official or third-party environments, covering fields from videogames to robotics or from classic to continuous control, also provides support through classes, modules and functionalities, for development of own environments ad hoc for own needs.

10.3.2 Goal-Based Environments

In order to deal with multigoal problems (but also to more easily manage sparse rewards) OpenAI has released an update for the Gym toolkit with the aim of introducing support for goal-based environments[14], already including the eight environments developed by them for their latest research (four with the *Fetch Robot* platform and four with the *Shadow Robot Hand* platform).

The characteristic of these environments is that the concept of goal is already encoded within them, in order to provide this data to the agent in view of a multigoal problem; as for the reward, it follows the same principle of a classic binary reward described above.

The environment used in this thesis is the *FetchPickAndPlace* one, belonging to the 4 environments using the Fetch platform among the eight distributed by OpenAI; like the others it uses the physical simulator MuJoCo.

FetchPickAndPlace Environment

The *Fetch* environment in general presents the Fetch robot which was widely discussed in the introductory part, using its arm as a manipulator in order to carry out the operations required for the specific environment.

The rewards, as already explained, are of the sparse and binary type, specifically the goal is considered achieved, obtaining a reward equal to 0, when the object (or the end effector in the case of the *Reach* environment) reaches the goal with a tolerance of 5 cm. By goal we mean the desired position expressed in three dimensions.

On the other hand, regarding the PickAndPlace environment (Fig. 10.1) specifically, its own characteristics concern spaces of states and actions. The former include linear positions and velocities of the gripper relative to its state and in relation to the arm, and in the case in which the object is taken, also its Cartesian positions, rotations via Euler angles and linear and angular velocities absolute and relative to the gripper. The second ones are 4-dimensional: three parameters to specify the Cartesian position of the gripper (its rotation is also present in the code, but it is always kept fixed as it is superfluous for the achievement of the task) and the last one to control its opening and closing.

The goal of the environment is to bring a cube (with 5 cm long edges) from a random position to a desired one (indicated by a red dot), both obviously within the manipulator workspace.

10.3.3 Hindset Experience Replay

Hindset experience replay (HER)[1] is a technique presented in the paper of the same name through which, in combination with off-policy algorithms, it is possible for an agent to learn in the presence of sparse rewards, specifically binaries.

The basic principle of HER is to allow learning from errors and therefore from the early stages of training, phases in which traditional RL algorithms would not obtain any useful information. This is possible through a goal substitution process in which when the desired goal is not reached, it is considered to have reached a different goal, a fictitious one, from which to obtain a significant learning signal; through repeated attempts the agent will then be able to reach any different goal arbitrarily.

As said initially HER is designed to be used in a complementary way to other off-policy algorithms, so in this thesis we will consider the same combination used in OpenAI researches, that is the “DDPG+HER” which has proved to be the most efficient

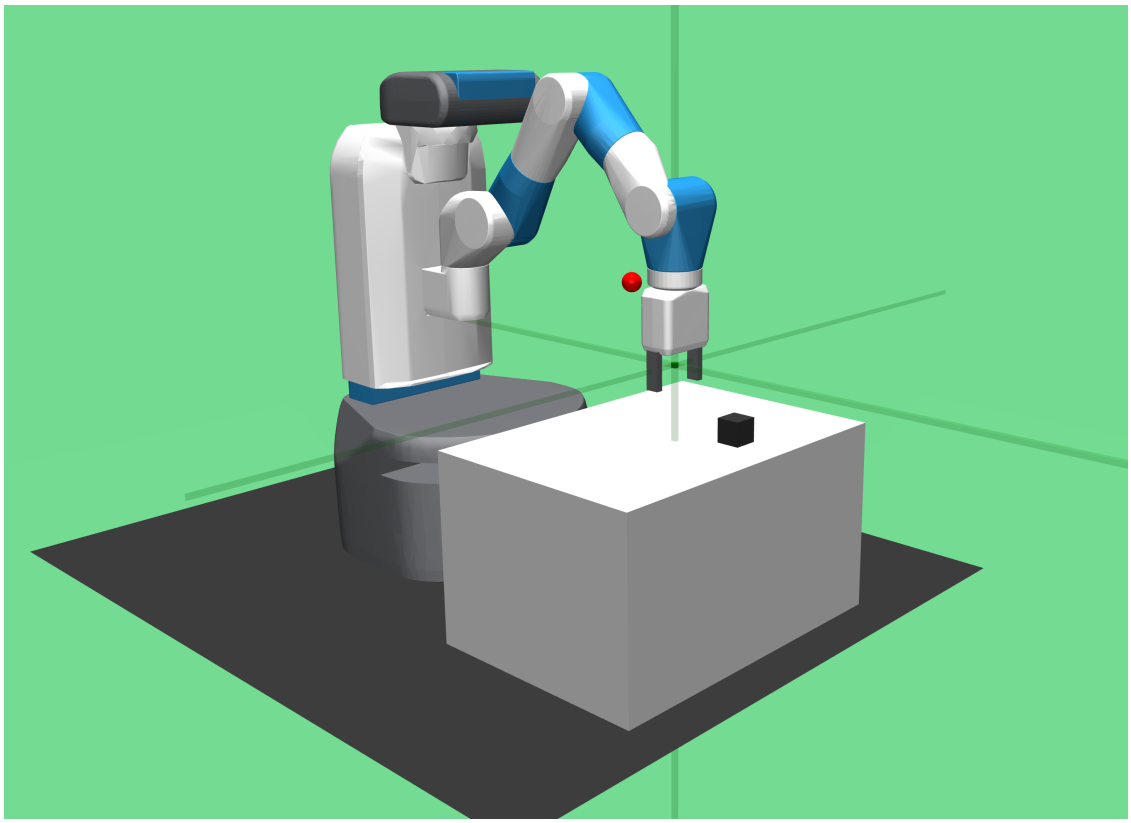


Figure 10.1: OpenAI Gym *FetchPickAndPlace* goal-based environment on MuJoCo [credits: openai.com]

by virtue of the excellent skills demonstrated by the DDPG algorithm in these types of tasks (continuous control).

10.3.4 Baselines

Baselines[7] is another contribution from OpenAI, also made available in an "open" way; it is a GitHub repository that serves as a collection of RL algorithm implementations and also provides support for managing agents.

Like Gym, it is updated following new research too, and as regards the two treated in this thesis, an implementation of the HER algorithm as well as a coherent DDPG implementation has been added to the collection for joint use.

Baselines fully supports TensorFlow from version 1.4 to 1.14, while support for version 2 is still under development, so some more modern implementations such as HER are not yet available in the latest version.

10.4 Stable Baselines and RL Baselines Zoo

Stable Baselines[8] is a fork of the Baselines project that presents some improvements and more frequent updates compared to the previous one, it is also compatible with the *RL Baselines Zoo*[16] toolkit which has the convenience of providing, in addition to pre-trained agents with hyperparameters already tuned, also more intuitive interfaces for managing agents themselves.

These two toolkits are used by the author of the thesis; as for the original project also these do not yet fully support TensorFlow 2, but versions from 1.8.0 to 1.14.0, so TensorFlow 1.14.0 was used.

10.5 MuJoCo

MuJoCo (**Mu**lti-**Jo**int dynamics with **Co**ntact)[19] is a physics engine that allows you to quickly and accurately model, simulate and visualize contact dynamics of multi-joint systems.

Born for the design of controllers and with a main focus on high performance, one of the fundamental characteristics of MuJoCo is its ability to manage very onerous operations from the computational point of view, especially in dynamic and complex systems with a high number of contacts, systems for which the simulation is optimized.

In fact, in addition to being a traditional simulator, it can be used for model-based calculations, its preferential use by the field of research and development for ML and specifically RL is due to this, it is not by chance that it is the simulator selected by OpenAI for the implementation of 3D environments designed for continuous control in the Gym toolkit.

MuJoCo is written in C and is cross-platform, furthermore the models are written in the MJCF format which is an efficient XML format, however URDF models used by ROS can also be loaded.

The main components are:

- simulation module
- XML parser and model compiler
- interactive OpenGL viewer

Chapter 11

Conlusions

11.1 Results

The agent execution is divided into *episodes*, an episode represents an attempt by the agent to perform the task, at the end of an episode, an *episode_reward* equal to 1 if the task has been completed, or 0 otherwise, is returned; this reward is a "utility" return (provided by RL Baselines Zoo) that refers to the episode and is not to be confused with the reward (0 or -1) that has been extensively described above and that refers to the return of each single action.

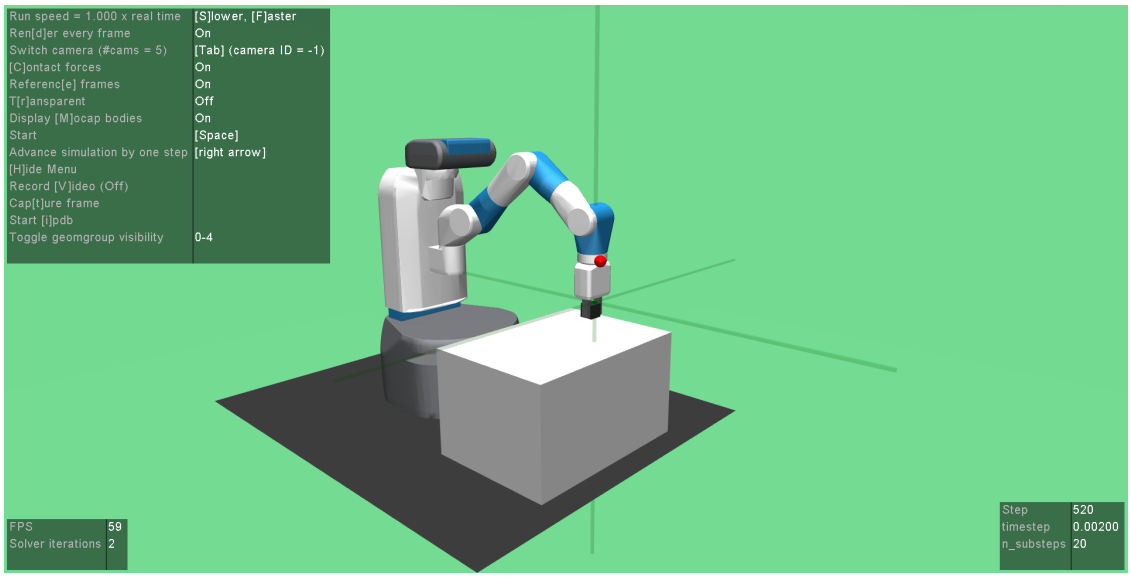
The episode in turn evolves into incremental *timesteps* and ends either when the agent reaches the goal (*episode_reward* = 1) or when the *max_episode_steps* value, set at 50 for the FetchPickAndPlace environment, is reached (*episode_reward* = 0).

Each environment timestep is divided into several MuJoCo steps, these steps that do not concern the environment itself as much as its simulation, are decided by a parameter called *n_substeps* (also visible in the simulator window, below on the right in Figure 11.1) and represent the sampling substeps that the MuJoCo solver executes at each timestep and consequently at each call of the *step function*.

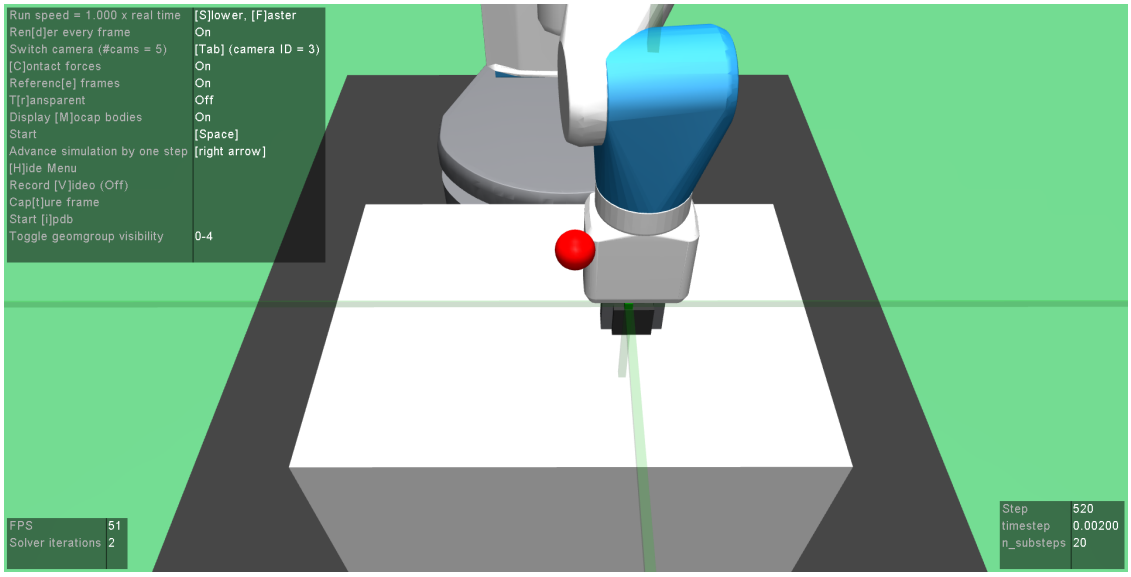
The step function is the function that in the Gym environments allows the interaction between agent and environment to proceed, consequently it is the function to be recalled so that in a given timestep the agent carries out his action and receives the reward and observation of the new state, passing in a later timestep of its execution.

The *n_substeps* is a parameter decided in the code that implements the class of the specific Gym environment and which in turn is passed to MuJoCo when the simulated environment is initialized, each substep has a duration expressed in seconds and defined by the XML model of the environment with the *timestep* variable (not to be confused with the timestep intended as a step, and not as a time, which has been mentioned previously), which has the same reference by the simulator and is also shown in the simulation window.

To clarify, this time interval refers to the simulator steps (the substeps) and not to the execution steps (the timesteps), and it is a parameter of MuJoCo that determines the accuracy and stability of the simulation.



(a) *Free camera view on entire environment*



(b) *Fixed camera view on table*

Figure 11.1: Agent execution on MuJoCo *FetchPickAndPlace* environment

For the PickAndPlace environment, *n_substeps* and *timestep* are respectively 20 and 0.002 s.

The simulator window shows the simulation, consequently the parameters shown inside must refer to the MuJoCo parameters, the third datum not yet treated, in the same box as the previous two in the Figure 11.1, is called *Step* and shows the step of the simulation that is currently being performed.

The execution proceeds in terms of timesteps but the solver samples in terms of substeps, consequently, at each action of the agent, the steps counter have an increment equal to *n_substeps* until the end of the episode; if the agent fails to reach the goal, being the episode fixed with a maximum length of 50 timesteps, the steps counter ends at an overall value of steps equal to $50 \cdot 20 = 1000$.

However, the steps counter in the simulation does not start from 0 to get to 1000 (always in case of not reaching the goal), but starts from 200 to get to 1200, this is because in the environment setup ten timesteps are dedicated (through as many calls of the step function) to bring the arm to a pre-fixed position, therefore ten calls of the step function require $10 \cdot 20 = 200$ simulation steps in total.

Regarding the timestep parameter, it indicates the actual simulation time, as a result, under the conditions of having lower processing times and coherent dynamics of actuators, simulation timing may be the timing of the task running on a physical robot, this parameter can be tuned in the testing phase to obtain simulation times perfectly consistent with the real ones; based on the simulation alone, the maximum time per episode is $50 \cdot 20 \cdot 0.002$ s = 2 s

All the other named parameters can also be modified and adapted for different needs.

Another parameter that concerns the RL Baselines Zoo tool is *n-timesteps*, which refers to the total timesteps of the entire simulation and in a certain sense determines the number of episodes of the same, in a certain sense because the episodes contain a different number of timesteps based on when the agent reaches the goal, in the case of episodes of maximum length, their number is equal to $\frac{n\text{-}timesteps}{max\_episode\_steps}$.

As mentioned at the beginning, the execution of an agent returns information on the episodes in form of a reward, furthermore, together with this data, for each concluded episode, its length is also obtained, represented by the number of timesteps passed within it; if the goal is not reached, the length is obviously 50.

In the tests it was chosen to leave all the parameters of the environment at their default settings, and to vary only *n-timesteps* to show the variation of the statistics on the behavior of the agent from a few episodes up to large numbers (Table 11.1).

Table 11.1: Behavioral analysis of the agent

No. of timesteps	No. of episodes	Success rate	Mean episode length (± SD)
50	3	100 %	12.67 (± 0.94)
100	4	75 %	22.00 (± 16.19)
200	13	92.31 %	15.15 (± 10.42)
500	30	93.33 %	16.53 (± 10.18)
1000	57	91.23 %	17.39 (± 12.43)
2000	122	93.44 %	16.28 (± 11.72)
5000	309	94.17 %	16.16 (± 11.01)

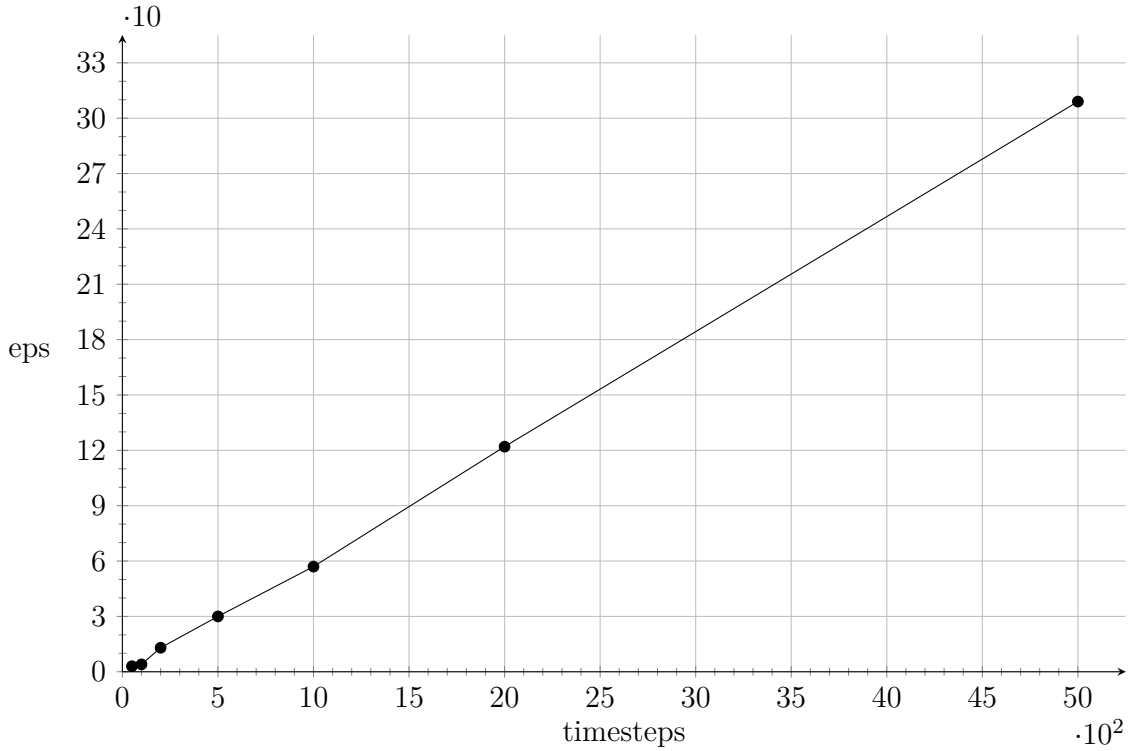
Later, instead, in the graphs in Figure 11.2, it is possible to see the curves that

describe the trend of the data extrapolated from Table 11.1 above, on the basis of the information resulting from the tests; it can be easily seen that, as expected, the number of episodes has a linear trend (except for the very first data of course) with the variation of the simulation timesteps, success rate and length of the episodes instead, after short fluctuations they settle to converge respectively at about 94 % and at 16.2.

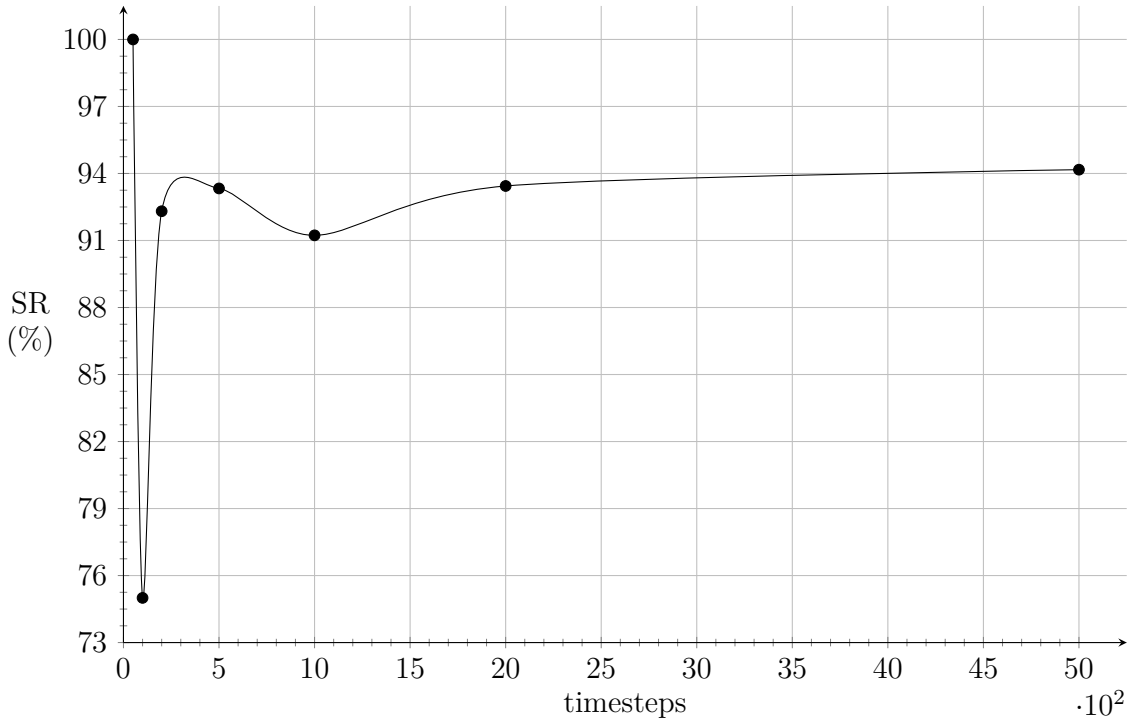
A final significant note on the results of the tests concerns the behavior that the agent learns, in fact it has no awareness of what a pick-and-place task is, but reasons only in terms of rewards, therefore in the learning phase, in a completely autonomous way, it realizes that it can make the object reach the desired position even by moving it in other ways other than a standard pick-and-place, for example by pushing it or almost throwing it at the target point.

This obviously happens only when the desired final position is at the level of the table and not in the air, furthermore the coefficients of friction and the flat surfaces of a cube do not make this a big problem as far as the final result is concerned; however, it is reasonable to consider changes in learning, or rather limitations in the granting of rewards, regarding the use of objects with curved surfaces such as cylinders or spheres.

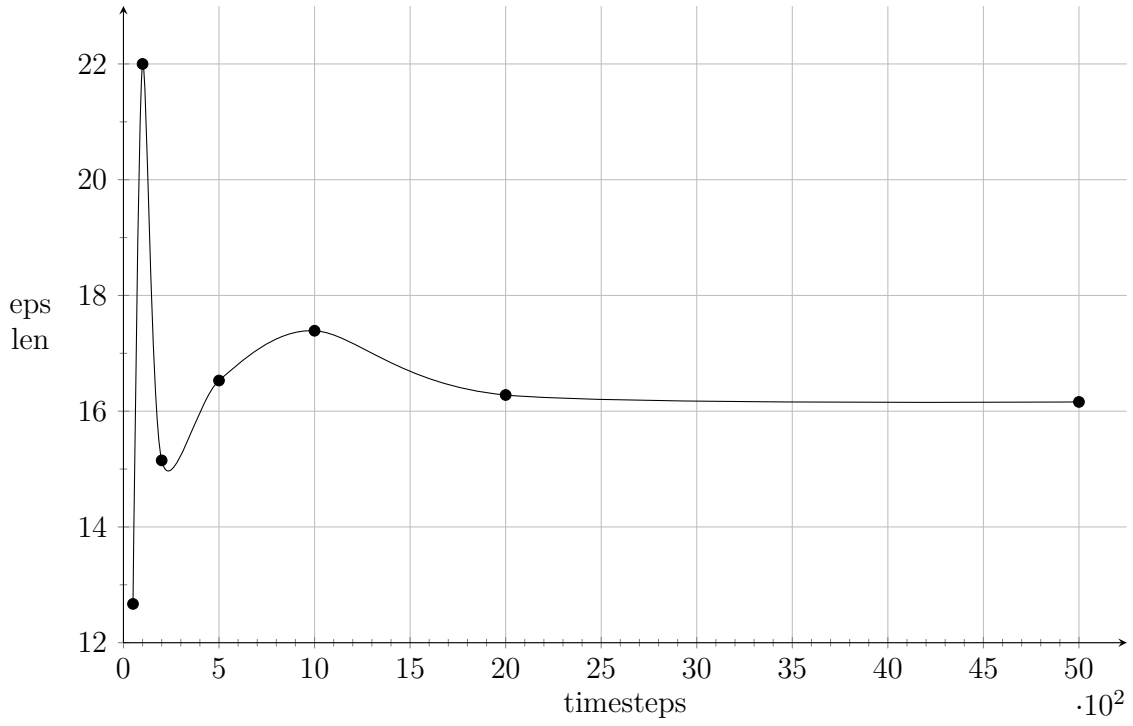
In any case, the one just mentioned is an excellent food for thought, showing the potential of artificial intelligence and the total logic and rationality of an entity programmed to achieve a specific goal, a goal that it could often achieve in a way that had not been predicted, or previously considered, by the programmers themselves.



(a) *Number of episodes performed as a function of the total number of timesteps*



(b) *Success rate as a percentage in function of the total number of timesteps*



(c) *Average length of episodes based on the total number of timesteps*

Figure 11.2: Curves extrapolated from the behavioral analysis of the agent

11.2 Future works

This section lists ideas, proposals or advice relating to the approach described in this part of the script, for future works of various kinds, whether they are additions, modifications or improvements.

- Satisfy the "Request for Research" in Plappert et al. (2018) useful for the improvement of pick-and-place tasks.
- Implement the code via the recent *TF-Agents* library for RL.
- Modify the environment so that the training presents, in addition to a random initial position of the object, also its random shape and size.
- Modify the environment so that the task is not completed as soon as the object reaches the predetermined position, but make it necessary to wait a margin of time to verify that the object has remained there and has not moved (this is useful in cases where the cube passes from the desired position without being left precisely there, and in the case of objects with a partially spherical surface).
- Make the agent more efficient in achieving the goal with tighter tolerance values.
- Carry out an estimate of the pose and shape of the object using CV methods equipped with ML techniques, for example with unsupervised learning algorithms using CNN, so as not to have to provide these data in a predetermined manner and thus making the arm more autonomous.
- Implement a "bridge" (valid and modern unlike the low quality offer currently available in the sector) between the Gym environment simulated with MuJoCo and a ROS environment simulated through Gazebo, which makes practical to integrate the code on physical systems.

Conclusions

Chapter 12

Comparison

All the treatment carried out so far, composed of the descriptions of the two approaches considered in this thesis, already expresses in itself all the implementation and executive differences that can distinguish them. However, later on, precise parameters of comparison on which to make any considerations, or to analyze in a more explicit way concepts already easily deducible from the text, are listed.

Finally, a personal conclusion are given on what emerged from the comparison.

In the following, for convenience of notation, the approach without ML is referred to as the approach *1* or first approach, and consequently the approach that instead makes use of ML is referred to as with the approach *2* or second approach.

Implementation and Integration Considerations

Before going into a more schematic list, the author want to highlight a significant note from the implementation point of view: the first approach makes use of a tool (MoveIt) which is already part of an environment (ROS) created ad hoc for programming, management and integration of robotic systems. By its nature the approach *1* allows to provide the robot, at the time of programming, with all the necessary information on the environment with which it will interact, thus making simulation superfluous in view of real application.

On the other hand, the second approach, to implement an agent and obtain a software that works, does not need as the first an ad hoc environment for a direct interaction with the robotic system. Consequently while the first approach implements, as a direct consequence, the problem from a complete robotic point of view (motion planning system, control system, actuating system and sensor system), the second does it only from the point of view of motion planning, and despite achieving its purpose, it needs a further step for a real implementation.

In fact, to implement the approach *2* on a real system, it remains necessary to switch between intermediary systems, such as ROS, but also proprietary software, or in any case any framework or application that allows interfacing to the machine at a lower level. Also with regard to the environment, the close dependence on it in this approach, as shown in this work, makes it equally necessary even in the execution

phase on the physical arm, as even without an actual rendering (graphic visualization of the simulation) there is always the need to implement an entity that exchanges data directly with the agent.

Adaptability

Using this as yardstick, the approach *2* is certainly better.

While an agent, if adequately trained, is able to dynamically adapt to new situations (specially if online learning is implemented, and therefore it continues also in the execution phase), an arm not equipped with RL algorithms will always be bound to its programming limits, and will be good at dealing with only situations for which it has been programmed and which have therefore been taken into consideration a priori.

Repeatability

As far as repeatability is concerned, there is no real winner as it is not a characteristic that determines the quality of the action (in the end, however, cases where this is not true are considered).

Analyzing this characteristic in the pure sense of the term, surely the arm programmed through the first approach is more likely to be able to execute a task several times in the same way. However, there are possibilities (albeit low) that the motion planner, as a result of its internal calculations, in the case of complex trajectories that require difficult elaborations to avoid collisions, for reasons of speed or time constraints, may select one solution rather than another when it finds a valid one even if others are available.

An arm programmed with RL will hardly perform the same action to perform the same task, even in this case the result depends on the complexity of the task, and certainly this can be taken for granted in the case of complex tasks such as pick-and-place. Furthermore, this statement is even more true in the case of agents who implement stochastic policies or who seek solutions through approximations, cases in which the stochastic variable is always present.

However, it should be added that while in the first case it is easy to insert constraints that prohibit certain practices and therefore force repeatability, in the second case this can be even impossible; so even if this feature is not important in standard cases, in situations in which there is a need to know as much as possible the actions of the system with the greatest possible determination, the first approach is certainly to be preferred.

Accuracy

For what regards accuracy, the best approach is always doubtful the first, this is because with standard programming it is possible to implement a task at any level of precision required, as long as the actuators can act consistently with those degrees of sensitivity.

Even using algorithms that foresee learning, it is possible to potentially reach any desired level of precision, however the computational complexity increases consistently

with the increase of the required accuracy, thus making a certain sensitivity in facing the task impossible in practice. This also, like everything, depends on the complexity of the task, elaborated tasks such as pick-and-place require high precision both in the phase of picking and in that of placing, therefore training the agent to perform this task with too high precision can be prohibitive.

Reliability

Reliability could be seen as a parameter dependent on repeatability and precision and closely related to safety, it could trivially be summed up with the question "Which of the two is more likely to go wrong?"; on the basis of what has been said, for what is the current state of the art, surely an arm programmed with the approach *1* is more reliable.

Autonomy

Autonomy is the ability to "make on its own"; two actions performed by the arm can be distinguished: planning and execution.

As far as planning is concerned, both approaches allow the arm, once programmed, to plan the movement autonomously; even if with the approach *2*, at the beginning, a training phase which may require continuous interventions is necessary, however, that phase can be considered as prior to the actual termination of programming.

As for the execution, in this case the second approach is better, in the analysis carried out in this thesis it may not be immediately visible, but in the case in which a more complex pick-and-place is implemented, a task that forces the arm to learn different positions with which to approach the object to be taken, and not the simpler ones that can be defined by normal Cartesian movements, it is possible also the situation in which, in turn, with the first approach it is not possible to carry out the task knowing only pose and form of the object, but it may be necessary for the programmer himself to code how the arm must approach the pick phase (and the same can apply to the place phase).

Responsiveness and Speed

Concerning the execution of more generic tasks, there is not much difference for the two approaches, but if it is a question of optimizing a specific task in detail, through the first approach there is certainly a possibility of intervening in a more fine manner at a lower level, possibility denied instead with the second one. Furthermore, it is a fact that no matter how valid an RL algorithm may be, an arm that has learned "a good way" to do something, cannot aspire to the quality of action of an arm that has already encoded "the best way" to do it.

Complexity

On the basis of what has been covered from the beginning of the thesis to now, it is evident that an approach of the second type is more complex than the first. This complexity does not refer only to the code and the problem to be faced from the logical point of view, but also a complexity from the computational point of view, since ML algorithms are much more expensive, especially in the learning phase, than those of the motion planner can be in the first approach.

Times/costs

It is chosen to put together the times and costs as it is a fact that the cost of a project increases in proportion to the time needed to complete it, consequently the approach *2* is to be considered more onerous both in terms of time that costs; the design of an optimal algorithm for solving a complex problem can require an indefinite amount of time, and despite this, the training factor should not be underestimated, as this phase can take several days of code execution.

Conclusion

Although artificial intelligence is a fascinating field, with infinite possibilities, and of indescribable importance for research, and although technology is continuously making great strides in this field, based on what has emerged, at the state of the art, undoubtedly an approach of the the first type, and which therefore does not rely on ML techniques, is still more robust and therefore to be preferred in real applications where reliability is a fundamental discriminant.

Appendices

Appendix A

Hardware and Software Setup

The hardware and software configuration with which the entire thesis project was carried out is described below, in order to reproduce as closely as possible the results described in the document. Furthermore, for each software, in addition to the version, the installation method is also indicated.

Notebook:

- CPU: Intel® Core™ i7-9750H CPU @ 2.60GHz (12 MB of cache, until 4,5 GHz, hexa-core) × 12
- RAM: 16 GB
- Graphic card: NVIDIA® GeForce® GTX 1660 Ti with 6 GB di GDDR6
- OS: Ubuntu 18.04.5 LTS 64-bit

Software:

- ROS Melodic (from binary)
- Gazebo 9.0.0 (gazebo_ros package from ROS)
- RViz 1.13.13 (from ROS)
- rqt 0.5.2 (from ROS)
- MoveIt 1 for ROS Melodic and Ubuntu 18.04 (from binary)
- Gym 0.17.2 (with pip3)
- MuJoCo Pro 1.50 (with pip3)
- Stable Baselines 2.10.0 (with pip3)
- RL Baselines Zoo (from repo)

- TensorFlow-GPU 1.14.0 (with pip3)

Extra tools used:

- Eclipse 2019-12 (from binary)
- git (from binary)
- python3-pip (from binary)
- virtualenv (with pip3)

Appendix B

Acronyms

On the next page (Table B.1) it is possible to view the alphabetical list of all acronyms used in this thesis.

Table B.1: Acronyms used in the document

Acronym	Meaning	Acronym	Meaning
A2C	Actor-Critic	**MJCF**	MuJoCo Format
AI	Artificial Inteligence	**ML**	Machine Learning
ANN	Artificial Neural Network	**MV**	Machine Vision
API	Application Programming Interface	**NN**	Neural Network
BMI	Brain-Machine Interface	**OS**	Operating System
CMAC	Cerebellar Model Arithmetic Controller	**PC**	Personal Computer
CNN	Convolutional Neural Network	**PG**	Policy Gradient
COLLADA	COLLAborative Design Activity	**PLC**	Programmable Logic Controller
CPU	Central Processing Unit	**RAM**	Random Access Memory
CV	Computer Vision	**RNN**	Recurrent Neural Network
DDPG	Deep Deterministic Policy Gradient	**RL**	Reinforcement Learning
DL	Deep Learning	**ROS**	Robot Operating System
DNN	Deep Neural Network	**RV**	Robot Vision
DOF	Degrees Of Freedom	**SD**	Standard Deviation
DPG	Deterministic Policy Gradient	**SDF**	Simulation Description Format
DQN	Deep Q-Network	**SoC**	System on Chip
DRL	Deep Reinforcement Learning	**SOM**	Self-Organized Map
EEG	ElectroEncephaloGraphy	**SPG**	Stochastic Policy Gradient
EMG	ElectroMyoGraphy	**SR**	Success Rate
EML	Extreme Machine Learning	**SRDF**	Semantic Robot Description Format
ENG	ElectroNeuroGraphy	**SVM**	Support Vector Machine
GPU	Graphics Processing Unit	**TF**	TensorFlow
GUI	Graphical User Interface	**TPU**	Tensor Processing Unit
HER	Hindsight Experience Replay	**URDF**	Unified Robot Description Format
HMI	Human-Machine Interface	**ViSP**	Visual Servoing Platform
IoT	Internet of Things	**VS**	Visual Servoing
IRL	Inverse Reinforcement Learning	**XACRO**	XML Macros
MCU	MicroController Unit	**XML**	eXtensible Markup Language

Bibliography

[1] Marcin Andrychowicz et al. *Hindsight Experience Replay*. 2018. arXiv: 1707.01495 [cs.LG].

[2] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[3] F. Chaumette and S. Hutchinson. "Visual servo control, Part II: Advanced approaches". In: *IEEE Robotics and Automation Magazine* 14.1 (Mar. 2007), pp. 109–118.

[4] Sachin Chitta, Ioan Sucan, and Steve Cousins. "Moveit![ROS topics]". In: *IEEE Robotics & Automation Magazine - IEEE ROBOT AUTOMAT* 19 (Mar. 2012), pp. 18–19. DOI: 10.1109/MRA.2011.2181749.

[5] Sachin Chitta et al. "ros_control: A generic and simple control framework for ROS". In: *The Journal of Open Source Software* 2 (Dec. 2017), p. 456. DOI: 10.21105/joss.00456.

[6] David Coleman et al. "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study". In: (Apr. 2014).

[7] Prafulla Dhariwal et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[8] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[9] HyeongRyeol Kam et al. "RViz: a toolkit for real domain data visualization". In: *Telecommunication Systems* 60 (Oct. 2015), pp. 1–9. DOI: 10.1007/s11235-015-0034-5.

[10] 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) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[11] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[12] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[13] G.J. Monkman et al. *Robot Grippers*. Wiley Interscience. Wiley, 2007. ISBN: 9783527406197. URL: https://books.google.it/books?id=EDpSAAAAMAAJ.

[14] Matthias Plappert et al. *Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research*. 2018. arXiv: 1802.09464 [cs.LG].

[15] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: vol. 3. Jan. 2009.

[16] Antonin Raffin. *RL Baselines Zoo*. https://github.com/araffin/rl-baselines-zoo. 2018.

[17] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. 2nd. Springer Publishing Company, Incorporated, 2016. ISBN: 3319325507.

[18] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 1846286417.

[19] E. Todorov, T. Erez, and Y. Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[20] M. Wise et al. "Fetch & Freight: Standard Platforms for Service Robot Applications". In: 2016.