eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Master Thesis

Master's degree in Computational Engineering and Intelligent Systems

Towards bridging the sim to real gap for robotic manipulation through the development of a realistic simulation using Unity, MuJoCo and ROS2

Jon Ander Ruiz

Advisors
Ander Iriondo
Igor Rodriguez
Elena Lazkano

September 12, 2023

Acknowledgement
Agradecimientos

Before digging into the project, I would like to take a moment to thank the involvement of all the people that, in one way or another, helped in the materialisation of this master's project.

First, I would like to thank the help provided by my supervisors Ander, Elena and Igor, for your feedback, assistance and patience, and for guiding me through the project. Moreover, I would like to thank the unit of Tekniker for the aid in technical questions, and for the opportunity to carry out the project with them.

Following, I would like to thank to all the people that helped me outside the academic and professional field. To my parents, Imanol and Laura, and to my partner Noelia, for all the support and motivation, and to all my family and friends who, in different ways, have shown interest for the project.

Many thanks to all of you.

Antes de proceder con el trabajo, me gustaría agradecer la participación de toda la gente que ha ayudado de una manera o de otra en la realización de este proyecto de fin de máster.

Primero, me gustaría agradecer la colaboración de mis tutores, Ander, Elena e Igor. Gracias por vuestro feedback, ideas y paciencia durante el trabajo y por haberme guiado durante el mismo. Asimismo, gracias a la unidad de robótica de Tekniker por la ayuda en cuestiones técnicas y por la oportunidad de realizar el trabajo allí.

Por otra parte, fuera del ámbito académico y profesional, me gustaría agradecer por el apoyo y la motivación constantes a mis padres Imanol y Laura, y a mi pareja Noelia, y, en general, a toda la familia y amigos que, de una u otra manera, se han interesado por el trabajo.

Muchas gracias a todos.

Abstract

The dissimilar behaviour that occurs in a simulation and in the real world with seemingly the same controllers and physical features is a very real problem that current researchers are trying to minimise. This problem is known as the "Reality Gap" problem. Since the first use of simulations the problem has been present, as it must be remembered that there is currently not such thing as a perfect representation of the world. There are numerous factors that must be taken into account and be modelled in order to have the perfect simulation, and multiple that cannot be. Many current approaches focus on reducing this gap either by modifying the parameters of the simulation to fit as good as possible to the real world, or by generating more robust controllers that can adapt to some amount of discrepancies. Either way, even the quantification of the gap can be complex, and it should be adapted or limited to the task in hand.

In this master thesis, the MuJoCo physics engine has been used to develop a simulation in Unity. In order to achieve the integration in the ROS2 (Robot Operating System) ecosystem, most precisely with MoveIt2, the proper software has been developed and implemented as well. Then, in order to quantify the fitness of the simulation, the interactions of a robotic arm and gripper with an aluminium piece in a manipulation task has been studied, both in simulation and in reality. These analyses have been portrayed in a metric, which has the objective of comparing the behaviour of said piece in simulation and reality and quantify the dissimilarities.

The simulation parameters provided in this work show promising results, accurately representing a very close behaviour in simulation compared to reality. The metric also provides a distance-like coefficient as a result, thus opening the gates to parameter optimisation techniques in order to further reduce the Reality Gap.

KEYWORDS: Unity, MuJoCo, ROS2, manipulation, simulation, reality gap.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Simulators are extensively used in the fields of industrial robotics and robotics research in general. Their usefulness is indeed indisputable; they can be used to conduct experiments without the economical cost nor the hazards or even the time cost of turning up a real robot. In simulations, we can replicate nearly every physical aspect of the real world. This includes; modelling forces such as gravity acting on the robot, and objects in the environment, defining material properties like friction, and even creating new forces to simulate wind. We can also manipulate the environment or create new scenes without the need of buying new equipment in an easy way, adding new objects and new robots. Another very important aspect of the simulations is that time can also be modelled. Time modelling is particularly valuable when training artificial intelligence (AI) behaviours, for instance using reinforcement learning, we could speed up time in order to train them faster. In addition, with a simulated system, which is a simulated robot in an environment, we can modify the robot's sensors to match specific problem constraints and experiment with various configurations to determine the optimal sensor setup. Later, when the sensor configuration fulfils the problems' constraints, the configuration can be set in the real robot.

Robotic simulation has two main aspects:

1. **The simulation environment** is the visual tool where we can see the scene and the robot being tested. It should provide an easy-to-use user interface (UI) to add new objects or, overall, modify the scene as well as offer a realistic rendering to see in detail the models of the objects in the scene. Furthermore, simulation environments may offer specific additional features. For instance, CoppeliaSim integrates path planning capabilities, while MuJoCo focuses on adding support for inverse kinematics [1].

2. **The physics engine** is probably the most important part of a simulation. Its main objective is to replicate as reliable as possible the physical attributes of the real world. It can go from the physics of collisions to even the density of a fluid the robot must work with. Certainly, the very nature of the problem or the environment will mark which attributes are important and which not, and thus which simulator fits better to the problem.

One of the most common problems in current simulators is the inability to realistically simulate object contacts, which is crucial in scenarios involving robot manipulations. Examples of bad contact simulation go from not being able to simulate correctly the grasp of a piece (compared to the real world) due to excessive slipperiness, to not being able to pick the object due to excessive trembling of the piece on contact.

In this work, a simulator using Unity and MuJoCo, integrated within the ROS2 ecosystem, is presented. This combination of state-of-the-art technologies allows for precise contact simulation with MuJoCo and the utilisation of existing algorithms and controllers through Unity and ROS2, most precisely, MoveIt2.

Additionally, a metric is also presented to measure the fidelity of the simulation compared to the real world. Using different tools and techniques, this metric evaluates how an object behaves when picked up in both simulation and the real world, enabling a comparison to identify discrepancies. These discrepancies, often referred to as the "reality gap", will be thoroughly examined to understand their underlying causes. Where feasible, efforts will be made to minimise these differences.

The reality gap phenomenon is an inherent challenge in simulations, representing the divergence between the simulated environment and the actual physical world. It can occur because of a variety of reasons, ranging from inexactness in the definition of the model's physical characteristics to a bad configuration of the environment's material characteristics, like friction.

The document is organised as follows: In chapter 2 the state of the art is reviewed, where we explore the criteria that other researchers have taken into account in order to select a simulator, outline our criteria, examine other State-of-The-Art applications and the simulators they employ, introduce the simulator we have chosen, explore relevant works in the field of reality gap, and we discuss the metrics researchers have used in their works concerning grasping stability. In chapter 3 a bit of theory about the robotic manipulation, kinematics, both forward and inverse kinematics, path planning, controllers and MoveIt2, the software used to plan trajectories and control the arm, is reviewed. Following, in chapter 4, we explain how we developed the simulation where the experiments are conducted, detailing both the components of the real world and their simulated counterparts. In chapter 5 the metrics of evaluation are explained, and the experiments are described in detail. In chapter 6 the results obtained in the experiments are reviewed, and finally in chapter 7 we will discuss the conclusions and future work.

1.1 Setup

The setup used in this research work is based on the workspace of the UR10 robot in Tekniker, as seen in Figure 1.1. The UR10 is a collaborative robotic arm, or "cobot". With its 1300mm of reach, 10kg of payload and 6 degrees of freedom, it carries out pick and place tasks with remarkable flexibility. This robot is coupled with a Robotiq 2f 85 gripper, a parallel finger gripper with a 85mm wide stroke. Its innovative design enables parallel gripping as well as encompassing grip mode. The gripper can be spotted in Figure 1.2. It also offers force and torque sensors, for monitoring stable grasps and securing objects to prevent slipping.

The URDF (Unified Robotics Description Format) of the UR10 has been obtained from

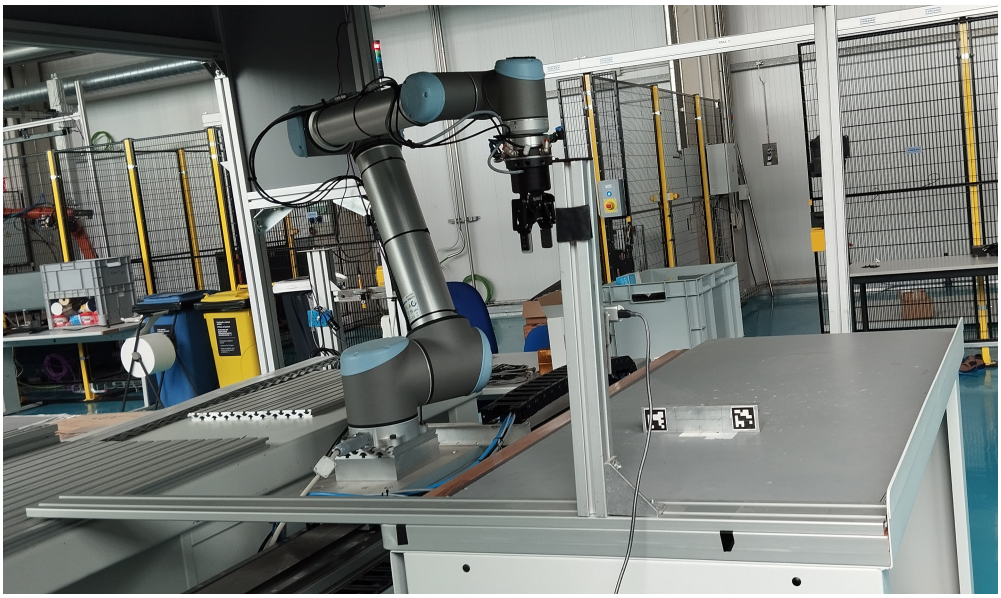


Figure 1.1: Universal Robots' UR10 collaborative robot arm and setup. In the figure, on the tracks the arm can be seen. Moreover, in the table, the aluminium piece, from now on grasping object, and the Realsense D435 camera can be seen. The objective is to model this setup in Unity.

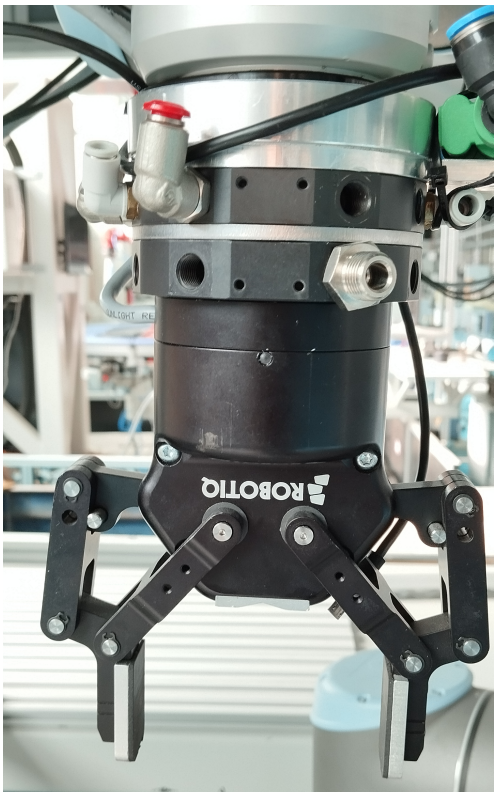


Figure 1.2: Figure showcasing the Robotiq 2f-85. 85 stands for the millimetres between the two pads. This is the gripper used in the project.

the *Universal_Robots_ROS2_Description* [1], and the corresponding model of the Robotiq gripper from *robotiq_2finger_grippers* [2]

As additional equipment, a Realsense D435 camera has been used for ArUco markers detection. These markers are later introduced in section 4.1.

Regarding the system, ROS2 Humble has been used, built from source. ROS2 is a collection of packages, functionalities and interfaces that ease the procedure of programming and commanding robots. Due to its modular structure, many capabilities can be implemented seamlessly in the ROS2 ecosystem. Such is the case of MoveIt2 [2] [3], a package that includes the latest developments in manipulation and motion planning.

Furthermore, Unity and MuJoCo 2.3.2 have been used for simulation development. The simulation is be developed in Ubuntu 20.04, and the tests using the real robot will be carried out in Ubuntu 22.04. The specifications of the computer are an Intel Core i7 10700 2.9GHz x 16 processor and 16 Gb of RAM.

1.2 Proposal

Past experiences showed us that neither Gazebo [4], the default ROS simulator, nor Unity with its default physics engine, PhysX, have shown as great potential in contact rich scenarios as MuJoCo, where high precision is also required.

Thus, our idea is to create a simulation based on the MuJoCo engine, within the Unity environment integrating ROS2, ideally being able to plan trajectories in MoveIt2 and seeing the results in the simulation, where the arm can interact with its environment. Having done that, we can then measure the fitness of the simulation using different techniques to evaluate the quality of a grasp.

1.3 Goals

Two are the goals of the project. The first goal is, to develop a realistic simulated environment for grasping integrated in the ROS ecosystem. To that end, the MuJoCo physics engine and the Unity engine are used. In order to complete this goal, the real world work environment must be modelled with high fidelity, together with the arm and the gripper. In the matter of modelling the arm, default robot's URDF has been used and adapted to fit the MuJoCo engine with modules from its own. These modules try to define as best as possible the physical characteristics of the object in question, such as, friction, mass, force... Then, the MuJoCo engine in Unity is integrated using its Plug-in and finally control the robot in Unity using ROS2 through MoveIt2.

The second goal, is to study the gap between reality and simulation, most precisely the differences on the behaviour of the different objects when they have been picked by a two-finger gripper. The differences will come to light when we attempt the same experiment both in the real world and the simulation. Hence, it is essential to have a good ground truth and metrics specifically fit for the experiments. These metrics will attempt to measure as best as possible all the appreciable differences between the real system and the simulated one. As mentioned, examples of this have been reviewed in the State of The Art [5, 6, 7].

[1] Link to the UR description repository - Humble branch.
[2] Link to the robotiq gripper description repository.

Cameras and computer vision techniques can be used to recognise displacements or imperfections during the grasping process, calculating for instance the inclination of the grasped object. This could be done in both the simulated environment and the real system and measuring the differences.

Noteworthily, other sensors like tactile or visuo-tactile sensors can also be used. Placed on the tips of the gripper's fingers, these sensors can provide force information, enabling the identification of the minimum force required to securely grasp an object and prevent slippage. These sensors also offer the ability to measure displacement by analysing the images from the pads.

CHAPTER 2

State of The Art

State-of-the-art studies have shown different metrics in order to choose the software that best fits the problem or its characteristics [1, 8, 9, 10, 11]. Most precisely, in the work of *de Melo et al.* [8], they reference the attributes or quality components that Jakob Nielsen defined in his book *"Usability engineering"* [12]. This quality attributes are staples in usability engineering, and they refer to the basic or essential components a system should have [13]:

1. *Learnability:* Meaning that the system should be easy to learn in order to start doing work as soon as possible.
2. *Efficiency:* Once the user has learned how the system works, it should achieve a high level of productivity
3. *Memorability:* How easy a user can re-establish productivity after a period of not using the system.
4. *Low error rate:* Referring to the errors the users make, their number, their severity and the easiness to fix them.
5. *Satisfaction:* The subjective opinion of the user about the system.

Letting usability engineering aside, *Collins et al.* review the different characteristics of most popular simulators, such as MuJoCo, NVIDIA Isaac and Gazebo [1]. They put their attention on calculation features like the capability for path planning, availability of inverse kinematics or the option to simulate suction. They conclude that realistic physical simulation is a very important aspect in the field of intelligent robotics. It is also mentioned that in the future, the principal lines of investigation of simulation will lie in increasing the stability, speed and improving visual fidelity. In [9], researchers focused on the machine learning (ML) capabilities of simulators. They took into account the following criteria: Whether the simulator can run in headless mode (without a user interface), its support for machine learning, its compatibility with ROS2, and its open-source status. MuJoCo, which is of special interest in the context of this project, has full headless support and it

is open source. However, it lacks ROS2 support, and the machine learning support is not integrated. After carrying out experiments to assess the quality of the simulators (though they did not experiment in MuJoCo), they declare that current robot simulation is not as precise as required to develop a digital twin. The research discussed in [10] aimed to assess the behaviour of various physics engines, including MuJoCo (utilising both the RK and Euler solvers), Bullet, DART and ODE. They conducted experiments of predictable nature and compared to the ground truth, acquired from an analytical solution obtained from applying classical mechanics. These experiments consisted of rolling a cube downhill and on a flat ground in different directions. More precisely, they modelled each simulator's parameters to match the analytical results, and then observed the simulator outcomes. They emphasise the importance of good parameter modelling and conclude that, in the case of the first experiment, MuJoCo using the Euler solver obtained a good balance between matches in the number of rolls and least rotation axis deviation. Conversely, in the second experiment, DART and ODE showed more regular and repeating patterns. *Erez et al.* [11], the creators of MuJoCo [14], compared their engine to others, specifically PhysX [15], Bullet [16], Havok [17] and ODE [18]. In their comparison, they concluded that each engine is good for the task it was designed for. While those other engines were designed for gaming, and thus performed well in those environments, MuJoCo was the best simulator in regard to accuracy and speed on constrained systems in the field of robotics among the compared simulators.

We aim to measure the fidelity of a simulated manipulation scenario. Thus, the appropriate simulation tool must fulfil some metric criteria.

1. It should provide precise and realistic simulations of contacts and grasping.
2. The visuals or renders of the simulator should be realistic.
3. It should be compatible with ROS2.
4. There should have already defined controllers for the arm and the gripper.
5. The simulator should allow the user to load URDF files.
6. We assess positively the ability to simulate suction.

This review also aims to understand researchers' experiments and the criteria they consider when selecting a simulator.

In [19], the popular Unity engine is used in the context of Human Robot Interaction (HRI). Unity has animation tools that allow non experts to visually program robots. The authors presented The Robot Engine (TRE), a new way of animating robots and to control how robots interact with humans in a very easy way. Unity has NVIDIA PhysX integrated underneath [20, 21].

Simulators are also used to train AI agents, especially when the cost of doing it in reality is too high. In this case, a sim-to-real approach is usually followed, so a simulation that reflects the reality accurately is needed. For instance, *Rajeswaran et al.* explored the options of training dexterous multi-fingered hands with Deep Reinforcement Learning (DRL) [22]. In order to do such training, and with contact accuracy in mind, MuJoCo was used, arguing that the contact stability it provides, makes it apt for this task. Similarly,

Lowrey et al. studied the use and benefits of RL using MuJoCo in simulations. They came to the conclusion that, when compared to the real hardware (HW), the simulation was, generally, accurate enough [23]. Though in some cases, the simulated sensor readings differ slightly from the real sensor readings. *Andrychowicz et al.* also used RL to learn dexterous vision-based in-hand manipulation for the Shadow Dexterous Hand [24]. They utilised MuJoCo as the simulator for training these policies and Unity for rendering the environment due to Unity's excellent visual capabilities. This choice was driven by the need for high realism to properly train the visual pose estimator. Following with the use researchers are giving to these simulators, *Lou et al.* developed a novel Collision-Aware Reachability Predictor (CARP) for systems with 6 degrees of freedom, in order to achieve better grasping positions in challenging or difficult environments [25]. For this task, the simulator they used is CoppeliaSim with the Bullet physics engine. They experimented both in simulated and real systems, achieving a promising average grasping rate of 78.78% in simulation and an 80.65% in real experiments. Also, most notably, the results obtained in the real system and in simulation are quite similar. Further applications of CoppeliaSim include the work of *Bogaerts et al.*, in which a platform of robotic system verification in Virtual Reality is created [26]. This work isn't very related to our scope, but it's interesting to know what other uses are developers giving to the simulators. Finally, *Chen et al.* investigated a not so well explored side of bin picking: picking objects from a deep basket or container, also known as deep bin picking [27]. For the purpose of investigating this task, they developed a custom simulation using the PyBullet engine. Among the conclusions they make, it is noteworthy their argue about the improvement of the speed of the simulation. They show that the speed of the simulations can be improved ten-fold approximately by selecting simulation parameters, among other things.

It can be seen in the analysed applications that simulators have walked a long way since their first versions. They have become accurate enough and are being used to train learning based algorithms, as well as conduct experiments to study the viability of different perception algorithms. These statements are backed by the fact that there are emerging a good number of RL toolkits and environments. In the same manner as with the applications, a variety of those toolkits have been taken into account. *Urakami et al.* developed a door opening focused environment called DoorGym [28]. Their objective was to train robot arms to open doors using RL and domain randomisation (DR), a popular RL technique that can also be used to reduce the sim-to-real gap [29]. DoorGym is implemented in MuJoCo, due to the API's capability to create elaborated physical simulations. Following with RL and MuJoCo, we can find JORLDY [30], an open source and customisable RL framework, whose main characteristic is that it supports plenty of RL algorithms and environments. Gym-Gazebo2 [31] offers an enhanced version of gym-gazebo RL toolkit that complies with OpenAI Gym. As its name suggest, it uses Gazebo and ROS2.

Taking all of that into account, and considering that no simulation is perfect and that none is strictly better than others, a combination of Unity [1] and MuJoCo [2] has been chosen for this work. To model the physical aspect of the simulation MuJoCo is used, being referred to simulate very precisely the contact dynamics in the current literature [1, 14, 32], and it meets the most important criteria regarding the physical simulation aspects:

[1]Unity Engine: Link to the page
[2]MuJoCo Physics Engine: Link to the page

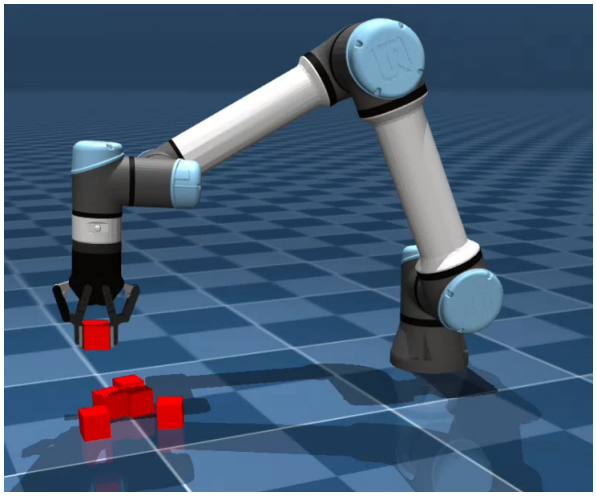


Figure 2.1: Example of UR5 arm with Robotiq 2F85 gripper attached in MuJoCo. In this figure, it can be appreciated the visual quality of OpenGL. The resolution is good enough, but the textures can be improved.

- Mainly focused on computational speed and contact stability.
- Capable of loading URDFs.
- Support the modelling of "suction" or something more approximate to a magnetic force[3].

Regarding the other aspects, MuJoCo can use OpenGL by its own, as it can be seen in the example scene in Figure 2.1. Yet, OpenGL does not offer the same visual quality as Unity. Moreover, it is also worth noting that we could use the fact that the integration of MuJoCo in Unity is achieved thanks to the plug-in[4] provided by DeepMind.

In terms of controllers, there are libraries namely ZeroSim [33] that allow the execution of trajectories using MoveIt![5] [2]. Furthermore, as a bonus characteristic, Kumar and Todorov developed MuJoCo HAPTIX [32], a VR system that allows the control of a simulated hand using a CyberGlove [34] that can interact with the simulated environment.

Concerning our subjective opinion of the MuJoCo engine, we have tested a few "toy experiments" and the first impressions were good. The modelling in their native robot description file, known as MJCF, is intuitive if you have a little experience with URDFs and the modification of basic physical constraints of objects such as friction is easy. More complex scripts can be coded in Python with their native Python bindings[6] supported since version 2.1.2.

The most relevant and current literature regarding the reality gap has been reviewed. Studies have pointed out [35, 36, 37] that, in the current literature there are mainly 2 ways of reducing this gap, either build more accurate and precise simulations or build/train controllers resistant to noise [38]. *Hwangbo et al.* also added that normally both approaches

[3] The developers of MuJoCo considered the petition of adding the "suction" feature and added it.
[4] https://mujoco.readthedocs.io/en/latest/unity.html
[5] Ioan A. Sucan and Sachin Chitta, "MoveIt", [Online] Available at moveit.ros.org.
[6] Link to their Python binding page

are carried out simultaneously [35]. Even so, in the current literature a good number of examples try to improve the simulation by optimising the parameters. A noteworthy instance of this can be seen in [6], where they optimised the parameters of the simulations using differential evolution (DE). They concluded that the optimisation of the algorithms helped greatly in the bridging of the reality gap. Another illustrative case can be read in [39]. In that work, they converted a biomechanical model from OpenSim to MuJoCo as they searched for higher efficiency in the simulation. They also optimised some parameters using a python implementation of CMA-ES (Covariance Matrix Adaptation Evolution Strategy) [40], a numerical optimisation technique used in continuous search spaces. In [37] they argue that a way to narrow the reality gap is to improve the fidelity of the model in the simulation. They do this by creating an accurate URDF, by disassembling their real robot and measuring all its physical characteristics, in addition to developing a more realistic actuator model.

A very good and contemporary example of the learning of more robust controllers against noise in simulation can be found in the domains of RL or DL, using techniques like **domain randomisation** [41]. Several simulation systems already support this capability, namely the aforementioned DoorGym [28] and Robosuite [42]. Morevover, *James et al.* established a novel way of reducing the reality gap with Randomized-to-Canonical Adaptation Networks (RCAN) [43]. Their approach only takes randomised simulation data and then translates that data to get their non-randomised version. This produces better results than only using domain randomisation. Regarding their tests, they trained a vision-based closed-loop grasping agent with RL and their results show that, after some joint fine-tuning in the real robot with 5000 real grasps, they achieved a 91% of success rate. As it can be seen, domain randomisation is of critical importance for learning based algorithms, but for our case, as we do not use AI, this technique is let out of the scope of the project.

In respect of the evaluation of the simulators, several studies have measured the quality by direct comparison with some kind of ground truth. For example, in [5] *Collins et al.* compared directly the performance of simulators with recorded, accurate real data and measured the error. The error measured was the Euclidean distance between the joint state in the simulation and in reality, and also the difference in distance between a displaced cube and the pitch of the cube. In [6], they used a collection of tasks, available at [44], and this time their metrics consisted of the measure of Euclidean distance error, the inner product of unit quaternion error (measure of the cumulative rotational error for the arm), pose error, velocity measures (mean, max and error), acceleration measures (mean, max and error), motor torque measures (mean, max and error), contact force and contact moment analysis (max and error), moving time and finally translational and rotational distribution comparison. In the same manner, the tool Live Tests for Robotics (LT4R) developed by *Fabry et al.* allows creating a state-based model of expected behaviour to measure the discrepancies and to write unit tests, a test that ensures the simulation is working as it should be [45].

With respect to our current goal, we will need to measure and quantify the differences between the simulation of the object grasping behaviour and its real counterpart. We found several references related to the simulation of object picking in the literature. *Kolamuri et al.* developed an algorithm that tracks the motion of a grasped object in order to detect rotations and potential failures in the grasping using gel pads, vision based tactile sensors [7]. Similarly, in [46] they detect and measure incipient slip with gel tactile sensors. In

[47], the authors describe that the use of tactile sensors can increase the grasping quality by translating and understanding the information these sensors provide. In their work, they analyse if a system can find the features to determine if a grasp will be successful or not. By seeing this works, it is conspicuous that the use of this tactile sensors provide great data [48], they can be used to see the movement of the object inside the grasp, assess the quality of the grasp or even collect geometric data to reconstruct the geometry form of an object [49]. It is important to have in mind that tactile sensors are not the only available resource to assess the grasping of an object. Camera based techniques are normally used to locate the object and grasp it [50, 51, 52], but note that they also can be used to evaluate the quality of the grasp or the object's stability after it has been picked. Another alternative to QR codes are the always popular ArUco markers [53]. These common markers are widely used to develop augmented reality (AR) applications or locate robots. Since their creation in 2014, their use has grown notably within the robotics community. We can find several uses in flying robots and drones [54, 55]. Most notably, the ArUco markers can also be used to measure rotation and position of an object between the grasp of a gripper, as seen in the aforementioned work of *Kolamuri et al.*, and set the ground truth with the readings of the real world [7].

Finally, we are also interested in a full integration of Unity and ROS2, most notably the use of arm controllers available in ROS2 to control the robot in simulation. As for this matter, there are several ways to establish communications between ROS2 and Unity, based on messages and services. One notable solution has been developed by the Unity Technologies team [56], facilitating the exchange of messages and services using *ROS TCP connection*. However, it has been considered that this may not provide the level of precision required for arm control. In the case of Gazebo, a staple in robotic simulations and directly integrated within the ROS ecosystem, the so called Gazebo-Ros-Control can be found. This plug-in enables the use of the controllers in the Gazebo simulated world.

CHAPTER 3

Manipulation Basics

Robots have been designed and engineered, chiefly, to complete dangerous, monotonous and non-rewarding tasks. Manipulators are no exception. Even if the attention put on this project may drift towards a more investigation-oriented overview, the reality is far beyond that. As time goes by, more and more industrial brands are adopting the use of these robots for pick and place tasks in applications ranging from assembly lines to packing zones.

This document will not go as far as to explain industrial systems nor applications but, even so, a theoretical base on manipulation and robotics is recommended, mostly to understand certain topics or technicalities. This chapter aims to provide some basics insights of manipulation to achieve a better understanding about the report.

Pick and place tasks consist of picking an object at a certain point in space and transporting that object to another location in space, normally placing it there. Although it sounds easy, the robot must carry out countless calculations. Let's put an easy example. We have a generic robotic arm with 6 Degrees of Freedom (referred to as DoF in the literature). To its right we have a table with a known item, and to its left an empty box where the object has to be placed.

For this task 5 steps can be identified:

1. Obtain the position of the grasping point.
2. Plan a valid trajectory to the object.
3. Pick the object.
4. Plan a trajectory to the bin.
5. Place the item inside the bin.

In industry, some robots may offer the capability of planning this kind of complex task. Though, in this particular case, using the UR10 robot, the ROS2 [57] infrastructure will be exploited.

In the first step, the robot must calculate the grasping point. Usually, a grasping point is known as a point of the object where the robot should pick the object from. To achieve

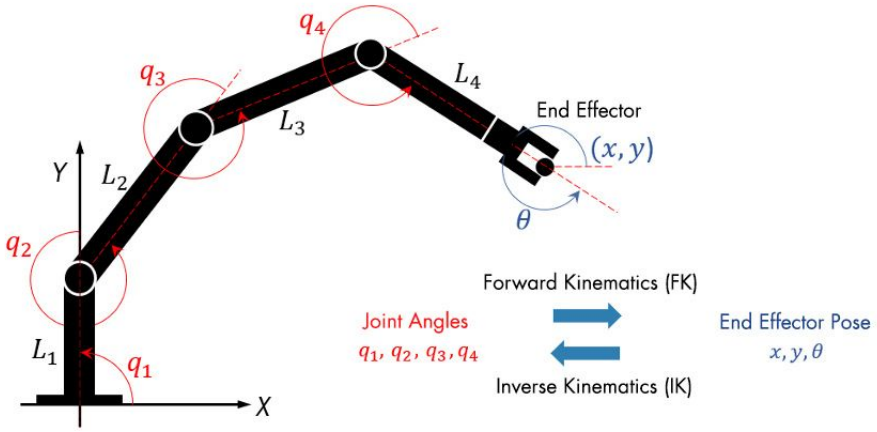


Figure 3.1: Inverse and Forward Kinematics. Knowing the position of the End Effector we can calculate the angles of all the other joints, also known as joint state, and knowing the joint state we can obtain the pose of the end effector. Image taken from MathWorks.

this, either the robot knows already the position and orientation of the object, i.e. that the object is in an already defined pose, or it identifies the position and orientation at runtime. For this last case, the robots need to perceive the environment, let's say, with a camera and computer vision techniques. Once the object is identified, and the grasping point obtained, the robot should plan a trajectory from the current state to the destination point. Here lies, perhaps, the most complex task. This is where inverse kinematics is calculated. On the one hand, Inverse kinematics (IK), is the calculation of joint states to move the End Effector[1], EE, to the desired point. On the other, Forward Kinematics (FK) calculate the position and speed of the End Effector once the joint states and velocities are known. This is the reverse process of the IK. A quite clarifying image, can be observed in Figure 3.1.

With the objective of reducing the complexity of IK, and making this calculation as easy and accessible as possible, MoveIt2 has been solidly established within the ROS2 community. This framework offers the state-of-the-art planners and IK solvers to lighten the manipulation tasks. Given this advantages, MoveIt2 has been a go-to tool in this project.

Once the trajectory has been successfully planned and executed, the object must be picked. For this part, MoveIt2 can also be used, as it offers the capability of managing multiple interfaces (one for the arm and another one for the gripper, for example). After the object has been secured, a new trajectory must be planned to the bin. This is done in the same fashion as before.

Finally, the task of placing the object inside the bin remains. The example of putting the object inside the bin was no coincidence, as in order to put the object inside the bin, a path within a constrained environment must be planned. MoveIt2 also supports this capability, if the constraints are defined correctly, of course.

Focusing on the path planning step, an arm controller is required to perform this task. A controller, as its name suggest, governs the actuators or motors. In essence, when a user specifies a command for velocity, force, or position, the controller instructs the actuators to exert the required force on the joints to achieve the desired outcome. MoveIt2 also

[1] In some cases, the point used to plan a trajectory is set between the pads of the gripper.

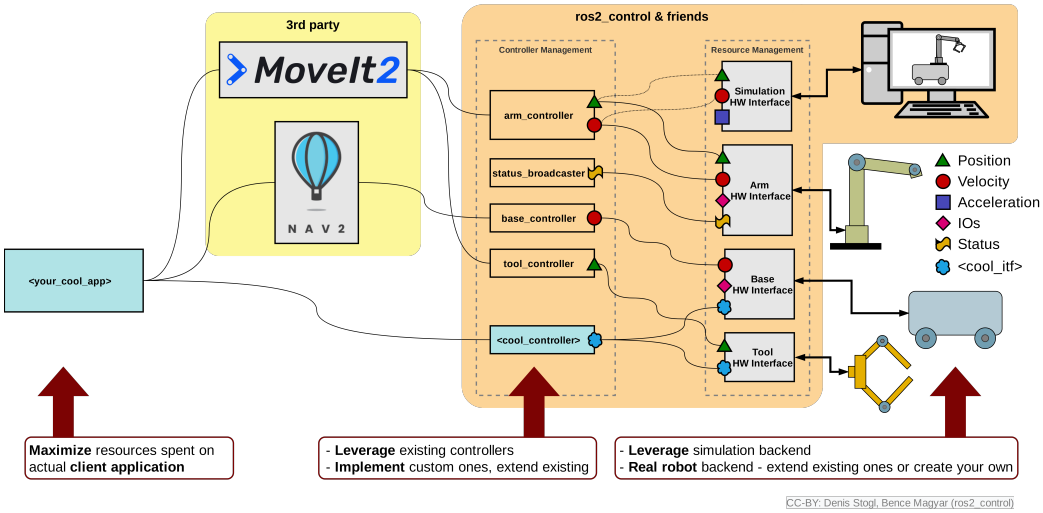


Figure 3.2: Architecture of ROS2 Control connected with MoveIt2. Here, we can clearly visualise the modular design ROS2 offers. Image obtained from the official ros-control page.

supports the capability of executing trajectories with ROS2 controllers. The infrastructure ROS2 Control, which is the architecture that implements the control functionalities, can be observed in Figure 3.2.

For this report, the design of a controller was out of the scope, and it should be recalled that, following the main idea of the project, the integration of the simulation within the ROS2 ecosystem would facilitate all the controllers already available.

CHAPTER 4

Developing a Simulated Environment for Realistic Manipulation Operations

Saying that the simulation is one of the most important part of development and validation of systems, controllers, environments and designs is not an overstatement. Simulations offer the highest possible control in all kind of variables that may affect the environment. From the time itself to any kind of forces exerted by the different elements in the system, all can be controlled and modelled in simulation. However, it must be admitted that as for today, there is no so called "the perfect simulation". There are always little miscalculations that affect the simulation ranging from little errors in the model of the robot to greater errors in the calculations of forces, or, generally, simulating something that can not happen in reality. Finally, not every interaction of physical robots with the environment are known or can be modelled. Neither the uncertainties intrinsic to physical sensors.

This chapter aims to illustrate the process followed to create the realistic simulation for grasping. Here the reader will found the different steps followed and proposed to develop the simulation; the Unity environment, the integration of MuJoCo in Unity, the integration of the simulation within ROS2 and offered functionalities, and finally, some executions of the main pipeline.

4.1 Measuring the Reality Gap with ArUco markers

In order to measure the reality gap, two poses of the grasping object are taken, using a camera to monitor the piece. The first pose is taken at the beginning of the monitoring process, while the second pose is taken at the end of the process. Then, the difference between the two poses is calculated, taking the displacement and rotation data from that operation. This process is done both in simulation and reality, and the difference in the results between them are calculated, thus quantifying the reality gap.

To estimate the grasping object's position in the coordinates of the camera, ArUco markers are used, one on each side of the grasping object. ArUco markers are square

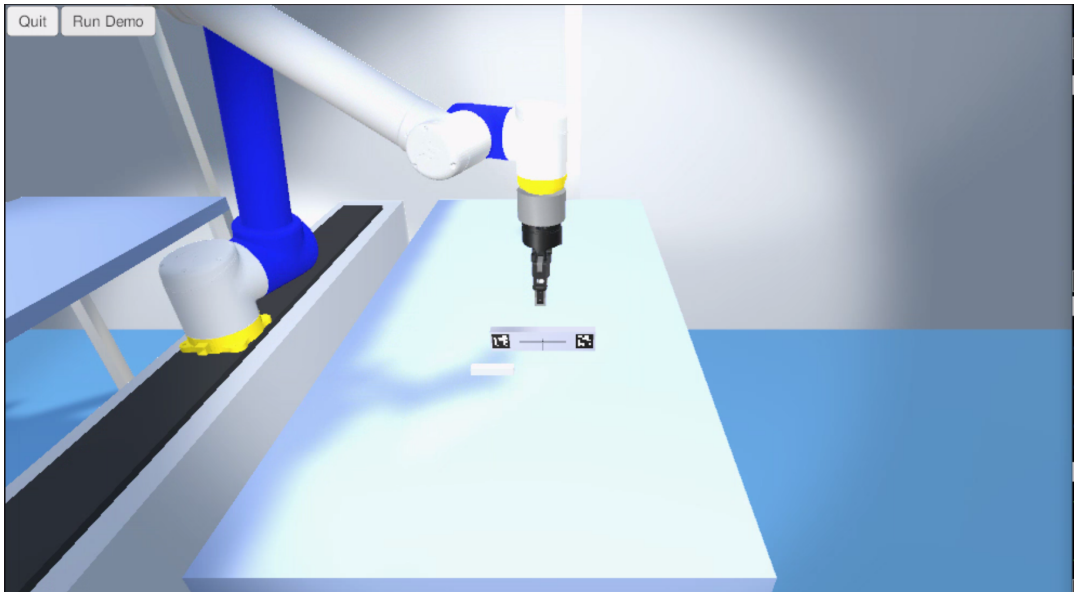


Figure 4.1: The environment in Unity seen from one of the cameras' perspective. The arm, table, grasping object with the ArUco markers and the buttons in the top left corner can be seen.

markers with black border and a predefined white pattern that are most commonly used for pose estimation in computer vision. The cv2 library, used in this project, offers the capability of creating and detecting these markers.

4.2 The Unity environment

The environment of the robot features two tables, a track where the robot stands (which is not functional, because that was out of the scope of the project) the UR robot itself (explained later in section 4.4), a housing, the aluminium grasping object (explored in section 4.4.3) and an array of cameras monitoring the setup. For all these elements, custom materials that somewhat resemble the visual aspects of the real setup have been developed, but most importantly, the dimensions and location of each element has been carefully measured and modelled accordingly. Given that in the current simulation there are no complex computer vision algorithms that benefit from the visual fidelity of the environment, the resources and effort have been directed into developing a realistic setup regarding dimension and location, as the comparison between the real behaviour and physical phenomena with the same conditions, both in the real system and in the simulated one is desired.

The Realsense D435 camera has been modelled as close to reality as possible, in terms of specifications such as camera intrinsics and functionalities, as well as providing distance to each pixel in the images recorded by this simulated camera.

Finally, a simple User Interface with buttons have been developed. These buttons access the functionalities (run the demo and quit) at runtime. Figure 4.1 shows the final version of the environment, seen from one of the cameras' view.

4.3 MuJoCo in Unity

MuJoCo, standing for Multi-Joint dynamics with Contact, is an advanced physics simulator that has the objective of easing the simulation of complex and contact rich scenarios. The interest on this physics engine by the robotics community lies in that manipulation tasks are usually contact rich scenarios. Initially developed by Roboti LLC, it was acquired and made open source by DeepMind in October 2021.

From a more technical perspective, MuJoCo is a C/C++ library with a runtime simulation module operating on low level data structures, pre-allocated by the built-in XML parser and compiler, to optimise performance.

As mentioned, MuJoCo offers an easy to install Unity plug-in. This plug-in allows the Unity editor and runtime to use MuJoCo physics. The way that MuJoCo integrates its physics in Unity is by creating an instance of the scene and storing all the information of the MuJoCo objects. These objects are typical Unity "Game Objects" but with special MuJoCo components that make them interact with one another and with the scene in a special way. Then, several global parameters and flags can be changed in a special script.

It must be noted that the MuJoCo plug-in offers a vast amount of possibilities, too many to name them all, to model scenes. Some aspects of the simulation are not even explicitly set, and it is let to MuJoCo to set them with the default options.

4.4 Modelling the robot

The modelling of the robot is a critical point. In order to have a realistic simulation, not only the environment but the robot *per se* must be faithfully modelled; from the dimensions of each piece to the actuators and motors, all must relate as accurately as possible to the real system.

In order to simulate the UR10 robot arm, two models are required. On the one hand, a set of files, including the URDF is necessary to interact with the ROS2 ecosystem, mainly with MoveIt2. However, MuJoCo is not integrated in ROS and thus, it reads its MJCF version of the URDF model.

As a way of familiarising with the intuition and the logic behind the modelling, two main resources have been consulted: the collection of robot models in the Menagerie [58] and the official documentation, most precisely the modelling and the XML reference sections.

4.4.1 UR10 in ROS2

The main URDF description model has been taken from the public GitHub repository of Universal Robots. Nonetheless, some modifications have been made in order to make the model as close as possible to the real robot in the lab, to modify the behaviour when it is used alongside MoveIt, and to enable the use of the Unity ROS2 control interface. This interface enables the control of the arm with MoveIt2, in ROS2. More information about this is presented in section 4.5.3. Here the main changes introduced to the robot are enumerated:

1. **Adding a virtual point in the gripper for object picking:** The changes in the description of the robot involve creating a new, point, called virtual link. This virtual

link is positioned between the pads of the Robotiq gripper to improve trajectory accuracy during object manipulation. The offset for this virtual link is determined by measuring the distance between the midpoint of the gripper's pads and the gripper's base, taking into account the gripper's unique design with additional articulations (see Figure 4.2). This virtual link must be defined as the tip of the robot arm (last link in the kinematic chain) SRDF[1]

2. **Limiting arm's joint movement:** The joint movement has been limited by modifying the corresponding YAML[2] file. This is done in order to avoid strange or not so efficient trajectories. For example, let's assume that we have our UR10 in rest pose and the objective point just in front of the robot, slightly to the left (see Figure 4.3). The logical approach would be to move the base to the left a bit, and then to approach the pose without moving the base. If the joints are not limited in the respective YAML file, the planner might get a path that makes the base move to the right, make a 350^{o} turn and then arrive to the point, instead of a more efficient trajectory. Usually, in pick and place scenarios, the time is a factor to take into account, and losing time having the robot make big rotations is not a good thing.

3. **Adding the robot tag:** To control the robot using the simulation and ROS2, the corresponding robot tag on the URDF has also required to be modified.

4. **Add gripper to arm and define controller for the gripper:** Add another virtual joint to attach or associate the last link of the arm and the gripper as end effector. This last modification entrails the integration of a new controller in the corresponding ROS2 controllers file. Otherwise, the gripper won't move.

4.4.2 UR10 in MuJoCo

On the other hand, the MuJoCo robot model is obtained through the MuJoCo compiler, that has the capability to parse the UR10's URDF file into MJCF file[3]. This tool is offered by MuJoCo, and its use is as simple as loading the URDF file into the simulation. MuJoCo internally will create the MJCF version of this model, and then, the XML can be saved by clicking in the Save XML option inside the simulation.

Though useful, the parser does not convert all the characteristics from URDF to MJCF, and consequently the robot description has to be completed by hand. All the physical parameters of the joints (such as damping) as well as the actuators had to be modelled. The main changes introduced are:

1. **Modelling the arm:** It must be noted that the arm must be defined in MuJoCo in the same way it is defined in ROS2. Otherwise, control accuracy can be lost when using ROS2 controllers. Discrepancies for instance in the definition of a target part can lead to inadequate results when the trajectory is executed, most notably when the execution involves constrained space.

[1] The SRDF (Semantic Robot Description Format), is the format used in MoveIt to read along the URDF and that extends the information provided by the latter.

[2] The joint_limits.yaml stores the information about the limits of each joint. Modifying that information will limit the joints when creating a trajectory.

[3] Robot description format based on XML and with .XML extension

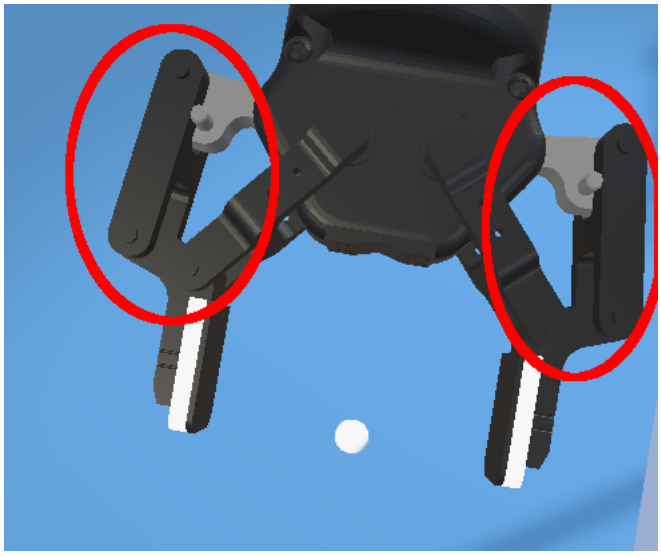


Figure 4.2: The figure shows a white point marking an approximation of the point used to plan trajectories. The red ellipses mark the articulations that enable the adaptation of the fingers to an uneven surface.

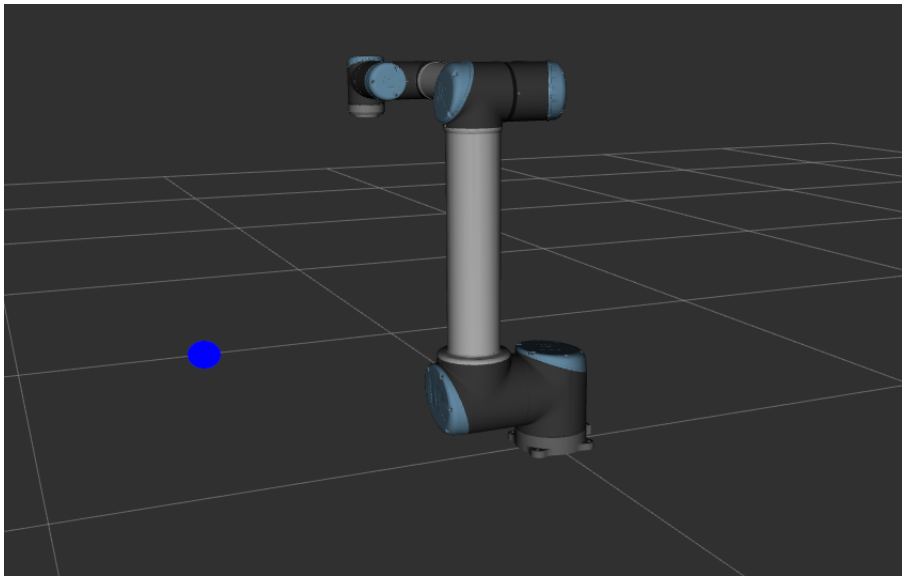


Figure 4.3: In this figure, the UR10 robot can be seen in rest pose, and in front of it marked in blue the objective point.

2. **Adding the end effector:** After modelling the arm and ensuring it was correctly defined, the end effector was added, in this case, the Robotiq 2f85 gripper. The MuJoCo model for this gripper can be found in the often aforementioned model menagerie. The menagerie's gripper is derived from the public URDF[4] and thus, no significant physical differences are to be expected from the gripper defined in the URDF. Therefore, a model that has the UR10 derived from the URDF and the gripper from the menagerie was developed.

3. **Modelling the MuJoCo parameters:** MuJoCo offers a vast amount of modifiable parameters, leading to model a lot of physical aspects with great accuracy. However, the modelling can be overwhelming. For this case, the parameters have been set following the parameters found in other robots of the menagerie, searching the documentation of the UR10, and also experimentally.

   Regarding the UR10 arm parameters, they have been set collecting the required information from its URDF file and from the UR5 model[5]. Table 4.1 summarises the collection of the parameters modified.

   It is also of special interest the definition of the gripper's pads' parameters, as the pads will collide with the grasping object. Table 4.2 collects the parameters and values for the pads. The parameters were already defined in the menagerie's model, and only a few modifications were made. It also can be seen that there are two different friction values. This is because the pads have been divided into two parts, in order to create more contact points, and they have different friction values.

4. **MuJoCo scene:** After the arm has been modelled, the MuJoCo scene has been loaded into Unity, simplifying the mesh files and converting them to STL format, as there was an already known problem when trying to load them directly into Unity. When the robot was successfully loaded into Unity, we ensured the robot was oriented in the same manner as the real robot is.

After several tests, a little discrepancy was observed; the robot was not behaving correctly when executing MoveIt trajectories. Specifically, it was seen that the fourth joint, the "wrist 2" joint, was shaking whenever it moved, in some cases even making the grasped object fall of the gripper. After some research it was found that, MuJoCo by default represents the environment in its ideal conditions, and thus does not consider noise by default. It was detected[6] that the firmware of the real arm may have a short delay compared to the simulated one, and therefore making the arm tremble. After adding a little delay with a specific MuJoCo parameter, it was observed that the performance improved.

4.4.3 Modelling the grasping object

The grasping object has been defined as an aluminium block of 22.5cm long, 5cm tall and 2cm wide. In order to model it, several MuJoCo parameters have been modified experimentally. The collection of parameters, as well as a brief explanation of them, is presented in Table 4.3. Finally, the ArUco markers were added in Unity.

[4] https://github.com/ros-industrial/robotiq/tree/kinetic-devel/robotiq_2f_85_gripper_visualization
[5] The UR5 uses parameters defined here
[6] Link to the issue where we found an answer

Parameter	Description
KP	It refers to the position feedback gain.
Damping	Joint specific parameter. It defines the damping applied to all degrees of freedom created by the joint.
Armature	Joint specific parameter. It defines the armature inertia to all degrees of freedom created by the joint.
forcerange	Actuator specific parameter. Defines the range of force applied by the actuator.
ctrlrange	Actuator specific parameter. Defines the range of movement of the actuator.
Range	Joint specific parameter. Defines the range of movement of the joint.
Solimplimit	Joint specific parameter. Sets the constraint on the solver for simulating joint limits.
Solreflimit	Joint specific parameter. Sets the constraint on the solver for simulating dry friction.
Solimp	Geometry specific parameter. 4.2.
Solref	Geometry specific parameter. See table 4.2.
Contype and conaffinity	Geometry specific parameters. Used to filter collisions. Two geometries collide if the *contype* of one of them is compatible with the *conaffinity* of the other.

Table 4.1: Collection of modified parameters for the UR10 robot arm.

It is worth highlighting the importance of establishing a good parameter baseline in the whole simulation, since this will make scaling the simulation easier when adding more objects and materials.

4.4.4 Modelling the physical aspects of the environment

In MuJoCo, there exists a set of global parameters that modify the overall behaviour of the environment. These parameters can be employed for various purposes, such as specifying environmental viscosity, adjusting gravity, introducing new magnetic forces, and controlling the direction of wind. Furthermore, these parameters also define the computational solvers, such as Newton or PGS, to be used in the simulation, integrators such as Euler and RK4, the contact cones and more. They also feature several flags that allow to override other parameters that have been set individually. Additionally, these parameters can enable or disable the use of the "Noslip" solver, which is used to prevent the grasped object from falling.

In MuJoCo, all these options and parameters can be set and modified in the MJCF file. However, in Unity, it is a good *praxis* to create a new GameObject and set inside this the Global settings script, then filling and modifying all the parameters the user deem necessary.

Table 4.4 summarises the collection of global parameters which have been selected by researching and by experimental tests.

Parameter	Value	Description
Density	1000	Determines the density of the piece and is used to calculate the mass taking into account the dimensions of the object.
Priority	1	Defines the priority on which the properties of the colliding geometries are going to be combined
Con Dim	3	Determines the dimension of the contact. A value of 3 indicates regular frictional contact
Sol Mix	1	Defines the weight of the parameters whenever averaging is required. It interacts with the priority attribute
Sol Ref	(0.004, 1)	Constraint solver parameter. On contact between two geometries, the one from the geometry with higher priority is used. If they have the same, the weighted average is used. This parameter has two forms, depending on the sign of the coefficients. If both coefficients have the minus (-) sign, then the parameter adopts the *(-stiffness, -damping)* form. If not, it adopts the *(timeconst, dampratio)* form. The first form is especially useful for defining bouncy geometries, like a rubber ball.
Sol Imp	(0.999, 0.999, 0.001, 0.5, 2)	Constraint solver parameter. On contact between two geometries, the one from the geometry with higher priority is used. If they have the same, the weighted average is used. The five coefficients are used to parameterise the impedance function.
Contype	1	Used to determine with which geometries does this geometry collide. Interacts with Conaffinity.
Conaffinity	1	Used to determine with which geometries does this geometry collide. Interacts with Contype.
Friction	(0.6, 0.005, 0.0001) and (0.7, 0.005, 0.0001)	There are five friction coefficients: two tangential, one torsional and two rolling. In MuJoCo, for dynamically generated contacts, three are used: tangential, torsional and rolling.
Shape type	Box. The dimensions have been modified to model the pad's form.	Defines the shape of the geometry.

Table 4.2: Collection of modified parameters of the gripper's pads.

Parameter	Value	Description
Density	2710	Determines the density of the grasping object, and it is used to calculate the mass taking into account the dimensions of the object.
Priority	3	Defines the priority on which the properties of the colliding geometries are going to be combined
Con Dim	3	Determines the dimension of the contact. A value of 3 indicates regular frictional contact
Sol Mix	1	Defines the weight of the parameters whenever averaging is required. It interacts with the priority attribute
Sol Ref	(0.02, 1)	Constraint solver parameter. On contact between two geometries, the one from the geometry with higher priority is used. If they have the same, the weighted average is used. This parameter has two forms, depending on the sign of the coefficients. If both coefficients have the minus (-) sign, then the parameter adopts the *(-stiffness, -damping)* form. If not, it adopts the *(timeconst, dampratio)* form. The first form is especially useful for defining bouncy geometries, like a rubber ball.
Sol Imp	(0.999, 0.999, 0.001, 0.5, 2)	Constraint solver parameter. On contact between two geometries, the one from the geometry with higher priority is used. If they have the same, the weighted average is used. The five coefficients are used to parameterise the impedance function.
Contype	1	Determines with which geometries does this geometry collide. Interacts with Conaffinity
Conaffinity	1	Determines with which geometries does this geometry collide. Interacts with Contype
Friction	(0.61, 0.005, 0.0001)	There are five friction coefficients: two tangential, one torsional and two rolling. In MuJoCo, for dynamically generated contacts, three are used: tangential, torsional and rolling.
Shape type	Box. The dimensions have been modified to model the aluminium grasping object's form.	Defines the shape of the geometry.

Table 4.3: Collection of modified parameters for the grasping object. The values have been defined both following the intuition seen in the MuJoCo documentation and experimentally trying different combinations.

Parameter	Value	Description
Imp Ratio	100000	Determines how hard it is for a geometry to start moving whenever it has collided with another one. In other words, how hard is to break the idle state and start moving or sliping.
Integrator	Euler	Determines the numerical integrator used for the calculations of the collision. It can be either *Euler*, *RK4*, *implicit* or *implicitfast*.
Cone	Elliptic	Determines the collision cone. It can be *elliptic* or *pyramidal*. Pyramidal tend to be faster, but elliptic gets better and more realistic results.
Solver	Newton	Selects the algorithm for the constraint solver. The available algorithms are *PGS*, *CG* and *Newton*.
Iterations	100	Maximum number of iterations for the constraint solver.
Tolerance	1e-08	This parameter sets the threshold for setting the early stop of the constraint solver.
No Slip Iterations	MAX	Defines the iterations of the No Slip Solver, a solver that prevents slipping.
No Slip tolerance	1e-06	Threshold for stopping the No Slip algorithm before it reaches its maximum number of iterations.
Mpr Iterations	50	Number of maximum iterations for the MPR algorithm, which is used for the calculation of convex mesh collisions.
Mpr Tolerance	1e-06	Tolerance for early stopping of the MPR algorithm.

Table 4.4: Collection of the global parameters. The values have been defined both following the intuition seen in the MuJoCo documentation and experimentally trying different combinations.

4.5 Integration within ROS2

As stated many times before, the integration of the simulation within the ROS ecosystem is paramount. For that purpose, a new control interface for Unity has been developed, namely, Unity ROS2 Control.

Before we dig further into this matter, a quick note must be made. Throughout this section, a differentiation is made between "The Unity part" and "The ROS part". This choice of words is used to differentiate in an intuitive manner Unity and ROS2, but it must be recalled that in order to communicate with ROS2, Unity also creates ROS2 nodes. Summarising, Unity and ROS are differentiated, but not forgetting that to establish a communication between both, Unity also uses ROS nodes that send information via TCP to the simulation.

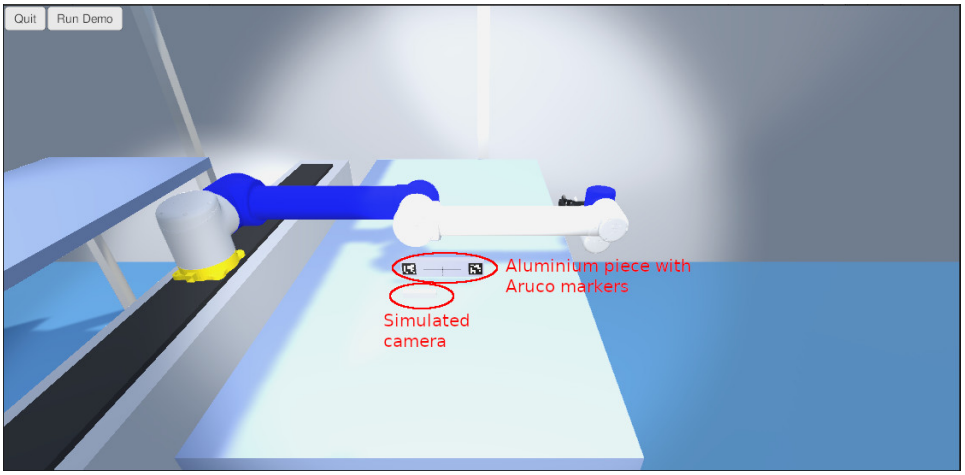


Figure 4.4: User interface of the simulation. On the scene, the working table. In the middle, the robot mounted on a track is observed. Recall that this track is non-functional in the scope of this project. In front, the grasping object with the ArUco markers can be found. Furthermore, the floating white piece represents the Realsense camera, and all the visual information of the simulation is collected from that point of view. In the top left corner, the utility buttons are found. The "Quit" button ends the simulation, and the "Run Demo" starts the pick operative.

4.5.1 ROS2 Message publishing in Unity

In order to publish to topics of the ROS2 infrastructure, Unity Robotics Hub has been used. This package, offers an entry point to the ROS architecture by providing a ROS2 node that communicates with Unity via TCP connection.

In the context of this project, the message and service communication capability is used for two main tasks:

1. Sending the object's position to MoveIt2.

2. Feeding ROS2 with images to simulate a video stream, and to detect the ArUco markers on those images.

For the first task, *geometry_msgs/Pose.msg* message type from the ROS2 geometry messages group has been used. The structure is formed by the tridimensional coordinates, represented in ROS2 by the **Point** message type, and the orientation, in **Quaternion** form.

In order to get the pose of the object, the coordinates have been obtained in Unity by consulting the Transform attribute of the object (since vision-based object and grasping detection is out of scope on this project). Though, this pose had to be adapted, as the coordinate system in Unity differs to the one in ROS. In Figure 4.5 both coordinate systems can be seen. The orientation of the object is also taken, partially, from Unity. The Y axis of the object is consulted in Unity, but it is modified to always be picked perpendicularly. Nonetheless, the simulation in this project has been modelled to pick the object that way, but in future versions of the simulation, it should also support the capability of picking the object from other orientations. The grasping pose greatly depends on the object's morphology, and thus picking it up from above it is not always the optimal solution.

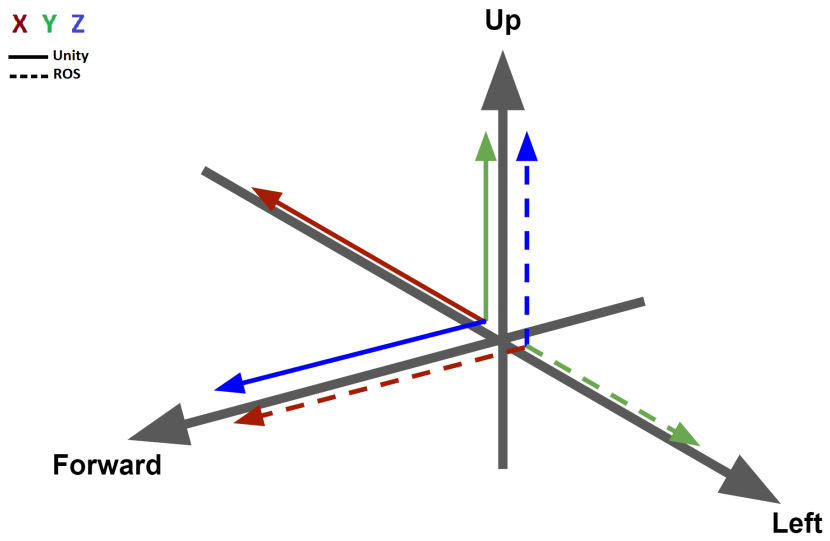


Figure 4.5: Coordinate systems, both in Unity and in ROS. Unity has the Z axis pointing forward, X pointing right and Y pointing upwards, whereas in ROS, X is forward, -Y is right and Z is upwards. Adding to this, the rotation in Unity is clockwise and in ROS it is counter-clockwise. Image taken from Siemens' ROS-Sharp wiki.

Note that this message is published by a message publisher node whenever the "Run demo" Unity button of the simulation UI is clicked (see Figure 4.4), and it publishes the point of the object but with an offset of 10cm in the object's Y axis. This was a deliberate decision in order to approach the object perpendicularly. Pushing this button will start the whole pick operative.

The second task requires images to simulate a video stream. Those images are published at 3-4Hz by a node in the Unity part constantly. The message frequency limit was established manually, because a greater frequency would put the computer under heavy load and affect the whole simulation without providing notable benefits.

In this case, *sensor_msgs/Image.msg* message type is used, from the sensor messages group. The message is formed by a **Header**, the **height** and **width** of the image, the **encoding**, a flag that determines if it is **bigendian**, the **step** and finally, the **data** of the image. The image is sent in greyscale from the Unity part, generally with an empty header, except by the last image the simulation sends. More information about this topic will be discussed in section 4.6.

To publish this gray image, a Unity shader was used, found in this GitHub repository. The shader transforms distance into greyscale. This way, knowing the gray colour and the scale, the distance to certain pixel can be easily calculated (see Figure 4.6). The scale is 256 at a maximum distance of 4 meters. This means that any object at 4 meters would have a gray value of 256, while an object further from this point will not be rendered into the image. It is known that large distances are measured with higher uncertainty. Thus, limiting the maximum distance to 4 meters allows a better discretisation within the gray scale.

Note that, when the end simulation button is pressed, Unity will publish one last image with a modified header. This header will notify the node in the "ROS Part" that no more

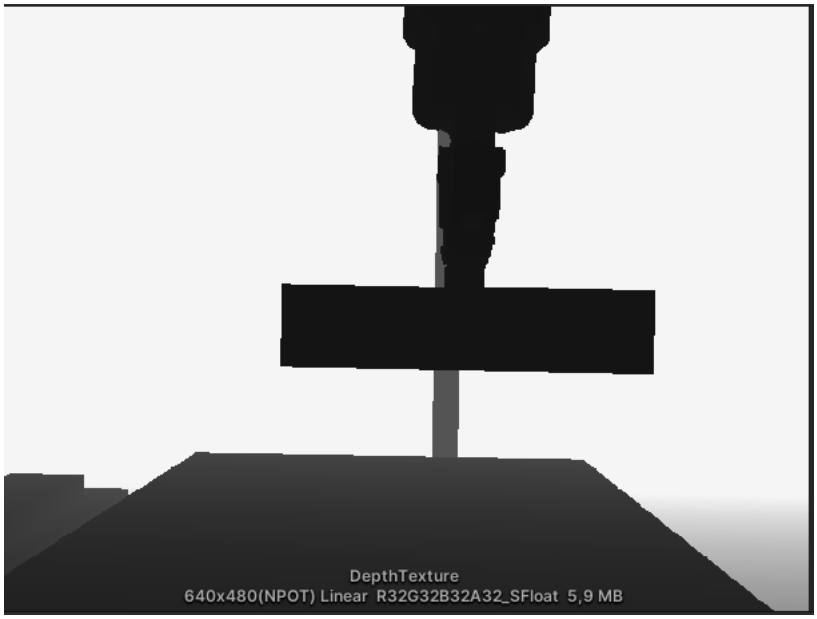


Figure 4.6: Image in greyscale. This RGBA image is generated by the shader. This shader generates the gray scale on this last channel. A script in Unity then publishes the whole image and a subscriber node reads it, dividing it into colour and gray scale.

images are going to be sent.

4.5.2 Message Subscription in ROS2

With respect to the "ROS Part", it consists of two nodes that subscribe to messages (a pose message and image messages) sent from the "Unity part"; one processes the position while the other processes the image.

Concerning the pose message part, two main applications that can deal with this type of information have been developed in the context of this project. One simply lifts the object, namely *unity_mover* and the other one lifts and shakes the object, called *unity_shaker*. The initial behaviour of both of them is the same. When the position is published, the respective ROS2 node, already subscribed to the topic, reads the content and obtains the pose. In the callback of the node, first, it creates all the environmental constraints that represent the work environment. It sets a ceiling and a table in order to avoid planning a path that would collide with them. After setting the constraints, it stores the pose (remember that the pose already has an offset of 10cm) in a variable. Then, it plans and executes the trajectory to that point. The script is able to execute the trajectory in Unity thanks to the Unity-ROS2-Control node. More information about that will be explained in subsection 4.5.3. After reaching that "pre-grasp" position, the script creates a point 10cm below the current point, which is the grasping point without the offset. When the "grasp" position is achieved, the script makes the robot close the gripper, grasping the object. Once the object is grasped, it is lifted 15cm. Here *unity_mover* ends. In the case of *unity_shaker*, after lifting, it shakes the object, tilting it 30 degrees to two opposing sides. It is worth noting that, the lift and the whole pick operative are done at 10% of the maximum speed of the arm and the shake part is done at maximum speed. This is done in order to give maximum stability while lifting and maximising any possible movement of the grasping object while shaking it.

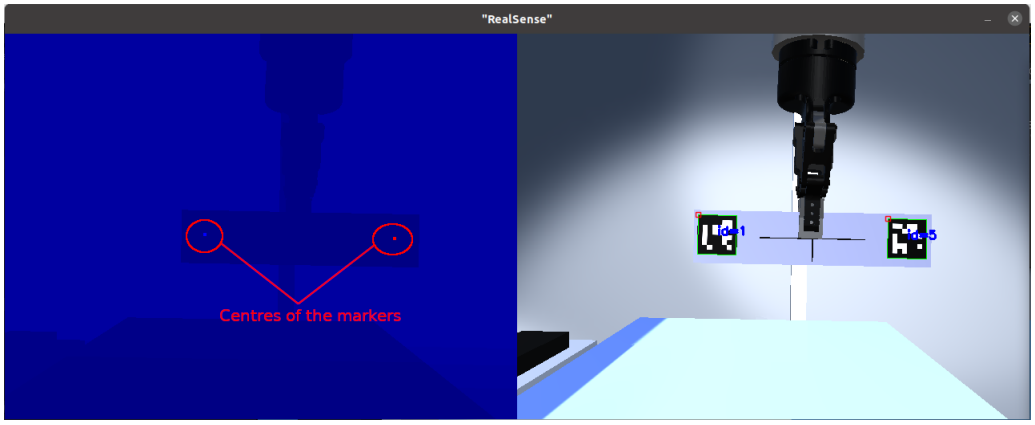


Figure 4.7: Visualisation of the images obtained from simulation. On the left we can see the depth image. In that image, it is also visible how the centre of the ArUco markers have been colourised in order to visualise that they are being correctly detected. On the right, it can be seen the colourised image.

On the other hand, the image acquisition node, is in charge of processing the greyscale images from the Unity image node; it checks for two conditions: it waits a small time step to avoid any possible error in the very first readings, and then it checks that both ArUco markers are being detected. When the two conditions are met, and while both markers are in the camera's view, it starts storing the information in Pandas[7] dataframes. The information obtained from the camera is as follows:

- The pixel coordinates of each marker (X, Y) are collected.
- The coordinates of each marker in camera coordinates. The markers' 3D coordinates are obtained by converting the selected pixels into point clouds, using the camera intrinsic parameters.
- The rotation in all axes with respect to the horizontal plane. This information can be obtained from the pose of each marker in camera coordinates.
- Translation in all axes. This can be calculated knowing the position of the markers in camera coordinates.

As stated previously, the last image is sent with a special header that lets this node know that no more images are going to be published. When this image is read, the dataframes are then saved as CSV files, and a set of figures that visualise the collected data are created. At the end, a table is generated that shows the difference between the first and the last captured readings. This is specially useful for a sequential comparison of the grasping object's behaviour in both simulation and in reality. More information about this will be explained in section 5.1.

In Figure 4.7 it can be noticed how the images have been represented, mimicking the display shown in the real camera.

[7] Pandas is a flexible data analysis library available for Python.

4.5.3 Unity ROS2 Control

This plugin is an updated version of the already existing Unity-ROS-Control for ROS1, which, at the same time, is an adaptation of Gazebo-ROS-Control. This previous version was created for ROS1, and it was not designed to be used in conjunction with MuJoCo. The adaptation, made in the context of this project, not only updates the driver to be used with ROS2, but it also enables the control of robot arms with MuJoCo elements. It must be mentioned that, this plugin is also based on Gazebo-ROS2-Control.

The plugin is divided into three parts for the sake of clarity:

- The Unity interface.
- The ROS2 interface.
- The plugin.

The two interfaces are connected via TCP connection, sharing a model structure that reflects the state of the robot in Unity and is used to be commanded by the ROS controllers. The model is built when the plugin is started, representing the same structure as the robot in the simulation. For this, both the robot in the simulation and the URDF should be the same, as the commands given in ROS2 should translate to commands on Unity.

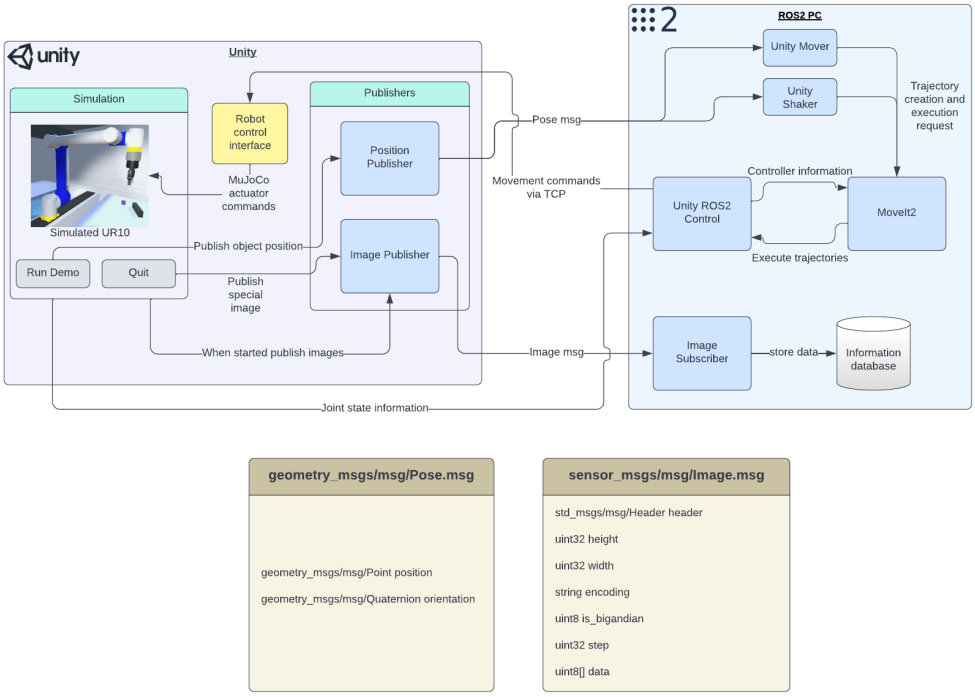
The Unity Interface, represents the current state of each joint of the robot and parses the commands given by the controllers to the MuJoCo actuators, which are elements in the Unity simulation. It is divided into two parts: "read simulation" and "write simulation". In the read part, the plugin reads the state of each joint in the simulation and fills the mentioned robot model structure. This information is then passed via socket to the ROS2 interface. Finally, in the write part, the commands given by the controllers are read and parsed in order to use them with the actuators of each joint.

The ROS2 Interface is divided into two parts as well: read and write. The read part obtains the information sent by Unity and updates the status of the structures, while the write part sends this information to Unity. This read and write form a loop in the server, reading and writing on the simulation constantly. This interface is initialised as a ROS2 node. When the node is started, it creates an instance of *unity_ros2_control* which is implemented as a library, enabling the access to the functions that load the model structure and update the status. The instance of *unity_ros2_control*, at the same time, also creates an instance of the controller manager[8] in form of a node, in order to manage the load and unload of different controllers, such as *joint_trajectory_controller*.

Finally, when the information from Unity is read, the read part also calls the update function of the plugin furthermore calling the update function of the controller, thus updating the status of the controllers.

The plugin. The plugin is accessed by calling the update function, which will also call the read, update and write functions of the controller manager. The controller manager offers control over the lifecycle of the controllers and manages access to the previously mentioned ROS2 interface. The plugin instantiates a *ros2_control* controller architecture

[8] More information about the controller manager can be acquired in the ROS2 control documentation page.



geometry_msgs/msg/Pose.msg

geometry_msgs/msg/Point position

geometry_msgs/msg/Quaternion orientation

sensor_msgs/msg/Image.msg

std_msgs/msg/Header header

uint32 height

uint32 width

string encoding

uint8 is_bigandian

uint32 step

uint8[] data

Figure 4.8: Architecture of the simulation and used messages. The communication in the architecture is divided in Unity and ROS2 Nodes outside Unity. The Unity part consists of the simulation environment, the robot control interface and the publishers. These are connected via TCP or messages with the other ROS2 nodes. This consists of several nodes that receive the information, subscribing to the topics, treat and process this information, and create movement commands or store the information, depending on the node. The ROS2 nodes are highlighted in blue.

4.6 Architecture

In order to better understand the global picture of the simulation, the structure of the architecture is shown in Figure 4.8. As it can be seen, there is a clear division between the Unity and the ROS2 parts. The Unity ecosystem is formed by:

1. **The simulation**. This would be the "game" in terms of Unity. It encloses the robot and environment models, and the UI buttons used to publish the robot's position or end the simulation.

2. **The robot control interface**. This interface is the script corresponding to the Unity part of Unity-ROS2-Control. It serves to communicate the robot state of the robot to ROS2 and to parse the MoveIt commands into actuator commands. The communication of this script with ROS2 is made using the TCP communication protocol.

3. **Publishers**. These publishers are Unity scripts that create ROS2 publishers, using the functionalities provided by Unity robotics hub, and send the pose and image messages in their respective topics. The structure of the messages can also be seen in 4.8.

On the other hand, the ROS2 part is composed by:

1. ***Unity Mover* and *Unity Shaker***. These are mutually exclusive, and they are the scripts that once they read the pose of the grasping object, they plan and execute trajectories in conjunction with MoveIt2. This execution is performed in the robot model of the simulation.

2. **MoveIt2**. It obtains the robot state as well as the objective point passed by the *unity mover* and *unity shaker* scripts to later plan and execute a trajectory between the robot's current state and the objective point.

3. **Unity ROS2 Control**. It receives and publishes the information about the robot's current state. This node also implements the initialisation of the controller manager, thus enabling and disabling the use of the controllers. Finally, it also sends motion commands to Unity.

4. **Image Subscriber**. This node obtains the images sent by Unity and searches for the ArUco markers on them. Then, it stores the marker's information, as well as some image files, into the PC. Note that, Unity will publish images from the start of the simulation, but the images will be only processed when this node is subscribed to the image topic.

CHAPTER 5

Empirical Assessment

The main scope of this project has been the simulation of manipulation operations. In order to achieve flexible and realistic results, several state-of-the-art technologies have been integrated, such as the MuJoCo physics engine, ROS2 and Unity. To measure the quality of the developed simulation environment, the assessed level of realism must be somehow quantified. I.e. the fidelity between the simulated system and the real one, namely the reality-gap, must be measured.

Due to the nature of the task, the grasped object is chosen as the reference to calculate the dissimilarities between the simulation and the real system.

The chapter has been divided into three parts, the first one being designing a metric that would measure the distance between the simulation and the reality. This is explained in section 5.1. The development of the experiments to fill this metric is covered in section 5.2, divided in two parts: the procedure in simulation, described in section 5.2.1 and the procedure in the real robot, reviewed in 5.2.2.

5.1 Metric

In order to establish a quality control and thoroughly validate the simulation, a metric that comprises the displacement and rotation in all of the grasping object's axes has been defined. This metric aims to set a score based comparison between the real behaviour and the simulated behaviour of the robot (see Table 5.1). The axes have been defined following the consensus of Unity, I.e. the left-hand Cartesian coordinate system (see Figure 5.1).

Both the displacement and the rotation of the grasping object in each of the three axis are calculated by measuring the differences between its initial state and the state after the monitoring ends. Then the error is depicted as the differences among the simulation values and the ones obtained with the real robot. This allows to assign a score to each experiment by following the score guide. This guide takes into account the error between the simulation and reality. Regarding rotation values, it has been considered that a difference of 90º in the X axis and any difference above 1º in the other axes are the worst case scenario. The displacement values follow the dimensions of the grasping object, i.e. the value of displacement in the Z axis has been set to a maximum of 225mm as this is the

Metric	Score Guide
Rotation in X (Degree)	0 if angle >= 90 1 if angle <= 1
Rotation in Y (Degree)	0 if angle > 1 1 if angle <= 1
Rotation in Z (Degree)	0 if angle > 1 1 if angle <= 1
Displacement in X (mm)	0 if displacement > 1 1 if displacement = 0
Displacement in Y (mm)	0 if displacement >= 50 1 if displacement <= 1
Displacement in Z (mm)	0 if displacement >= 225 1 if displacement <= 1

Table 5.1: Metric for the experiments. The metric compares the results in simulation and in reality, and it assigns a score based on the difference of both. The final score is the mean of all the scores. In any case, if the grasping object drops in simulation and not in reality and vice versa, the final score will be zero.

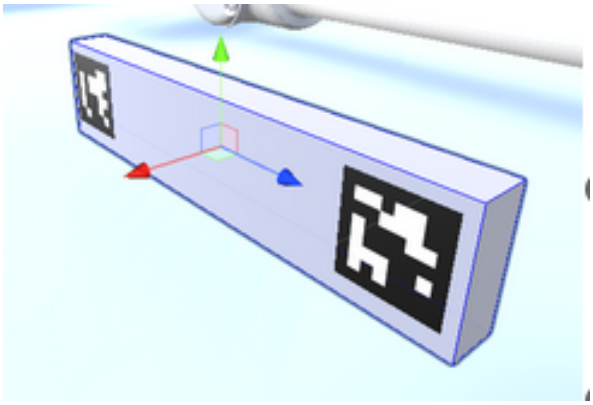


Figure 5.1: The coordinate system of the grasping object, shown in Unity. The X axis is represented by the red arrow, the green represents the Y axis and the blue the Z axis.

dimension of the grasping object in the X axis. This amount of displacement depicts the worst case scenario. We set minimum values to 1° in rotation and 1mm in displacement as we considered that said differences are acceptable.

The maximum rotation value in the axis X (90°) follows the logic that, if the object is horizontal while grasped, in case of the simulation does not model the friction correctly, the maximum amount of rotation the grasping object could do is 90°, getting vertical. The maximum rotation allowed in the other two axes is 1. This value has been set arguing that the grasping object should not rotate on those axes while is grasped. The minimum value for the rotation in all axes is 1, as it was thought that an error between 0 and 1 could be attributed to errors in the readings of the markers. In the case of displacement, the values in the X axis, which are 0mm at minimum at 1mm at maximum, follow the intuition that the grasping object should not move in that axis. Regarding the Y and Z axes, the displacement values are set following the dimensions of the grasping object; i.e. if the object would be picked from the very bottom, the maximum displacement this could make is its height.

It is also notable that the metric allows to calculate a weighted score, as seen in equation 5.1.

$$S_{total} = \frac{W_1 * S_{rx} + W_2 * S_{ry} + W_3 * S_{rz} + W_4 * S_{dx} + W_5 * S_{dy} + W_6 * S_{dz}}{6} \quad (5.1)$$

Where:

- S_{total}: is the final score.
- S_{rx}: is the partial score of rotation in axis X.
- S_{ry}: is the partial score of rotation in axis Y.
- S_{rz}: is the partial score of rotation in axis Z.
- S_{dx}: is the partial score of displacement or movement in axis X.
- S_{dy}: is the partial score of displacement or movement in axis Y.
- S_{dz}: is the partial score of displacement or movement in axis Z.
- W_{1-6}: are the weights assigned to each partial score. By default, they equal 1.0. The sum of all the weights must be 6.

This weighted score approach could result extremely useful if, depending on the object's characteristics, the precision in rotation or movement in certain axis is more important than in others. For example, if a water filled glass is being picked, the rotation in the object's X axis should have more weight than the rotation in the glass' Y axis, as the object rotation in the X axis would mean that the liquid inside is spilled.

5.2 Experiment Design and Procedure Development

Five experiments have been designed in order to measure the reality gap through the aforementioned score metric:

1. **Experiment #1:** Pick the grasping object by the centre and lift it 25 cm. This experiment aims to evaluate the fitness of the simulation in the simplest form, checking if the grasping object slips from the grasp, and in that case, how much.
2. **Experiments #2-3:** Pick the grasping object from a lateral, with 4.5 cm offset from the centre, and lift it 25 cm. This test aims to pick the object from a not-so "comfortable" pose, from where the grasping object could be rotated by gravity alone. This experiment is done for both left and right sides.
3. **Experiment #4:** Pick the grasping object from the centre and shake the object. The shakes should make the slip of the object more probable than in the first experiment.
4. **Experiment #5:** Pick the grasping object and make the object collide with an obstacle, set 20cm above the object and 9.9 cm from the centre to the side. The distance to the side is set to 9.9cm, as this is the distance where the real obstacle has been set.

Those experiments have been conducted both in reality and simulation. For each experiment, 10 iterations have been made in order to obtain a good amount of variability in the possible results. In total, information of 100 iterations has been collected, 50 in simulation and 50 in reality.

5.2.1 Procedure in simulation

In order to validate and test several aspects of the simulation, a validation process has been conducted. The pipeline consist of the following procedure:

1. Initialise simulation and controllers.
2. Make the arm pick the grasping object - conduct the experiment.
3. Obtain the readings of the grasping object from ArUco markers.

To **initialise the simulation and controllers**, a group of ROS2 nodes must be initialised. These nodes are: the TCP endpoint connector from Unity robotics hub, used to establish a connection between the ROS2 and the Unity endpoint; the Unity-ROS2-Control main node, that starts the aforementioned ROS2 plugin to access the ROS2 Control architecture and creates the model structure; the controller spawner nodes, used to load the desired controllers; the *unity_mover* or *unity_shaker* nodes; and finally, the image subscriber node. It must be noted that this last node is not always initialised at the beginning. It could be initialised later on, depending on the operative; either if we want to test the behaviour when the grasping object collides with an object.

Secondly, the picking process is executed in order to **make the arm pick an object**. The simulation offers a button that when pressed publishes the position and orientation of the object that is to be picked. When the simulation, thanks to the endpoint connector, publishes the position and orientation of the robot, the *unity_mover* or *unity_shaker* takes action, depending on the node that has been initialised between the two of them. These nodes, first build the environmental constraints, and then plan and execute a pick operative. First the system plans a trajectory to the object's position plus an offset of 10cm in the Z axis, as explained before, this is done in order to approach the object directly from above. When the arm reaches such position, it lowers the gripper to pick the object at the desired orientation. The moment it is at reach, it picks the object, closing the gripper, and lifts the object 25cm above the table. When this happens, and as mentioned before, in the case of *unity_shaker* it shakes the object 30 degrees front and back.

Finally, we **obtain the readings of the grasping object from ArUco markers**. To do so, the image subscriber node must be initialised. As previously stated in section 4.5.2, the initialisation of this node varies depending on the experiment. If the experiment is the collision experiment (experiment #5), the node is started from the very beginning. For the rest of the experiments, the image subscriber node is started later, when the grasping object has been lifted. Either way, the image acquisition node receives images with 3-4Hz frequency. When both markers are being detected in the image, this node stores the information of the markers.

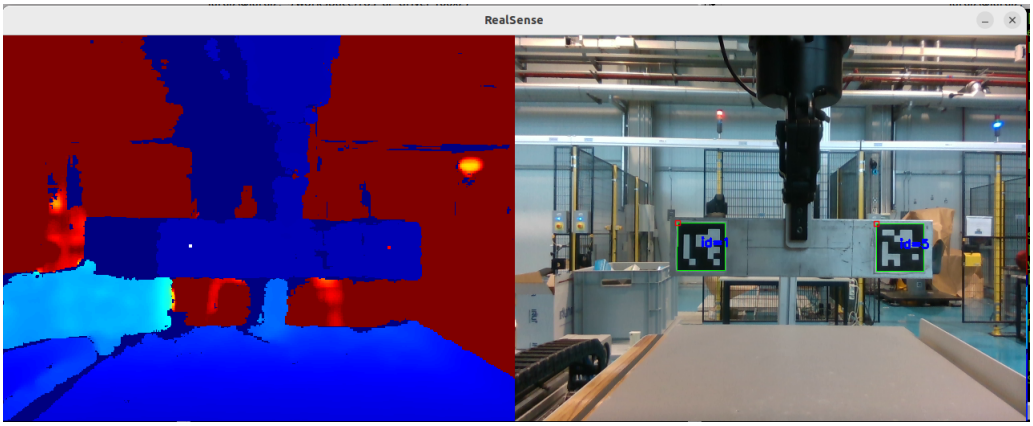


Figure 5.2: Point of view of the camera. The right side shows the RGB image, with the markers identified by the green perimeter and the top left identified by the red square. On the left, the distance image can be seen. On that, the marker's centres are marked in white and red.

5.2.2 Procedure in real world

The procedure followed in reality is the same as the procedure followed in simulation. Furthermore, the scripts of the simulation have been taken out as starting points, with little modifications, to carry out the experiments. There are two main differences between the scripts of the simulation and the scripts used with the real robot:

- For the experiments in the real setup, a Realsense D435 was available, and thus the official Realsense library for Python, *pyrealsense2*, could be used. An example of the point of view of the camera can be seen in Figure 5.2.
- The control of the gripper differs between simulation and the real environment. In simulation, the gripper is managed through MoveIt2, while in the real environment, a straightforward Python interface based on ROS2 services has been developed and employed. To use the interface in the scripts, the opening and closing MoveIt2 commands in the Unity scripts were replaced by service calls. Whenever an open/close service call is received, the commands are then passed to the gripper via MODBUS connection, ased on the information provided both in the manual and in this blog.

It is also noteworthy that in this case, the object's position is sent with the offset defined in the script instead of in the published object point like in simulation. This is only an implementation difference and does not affect these experiments

CHAPTER 6

Results

6.1 Experiment #1

Metric	Simulation	Reality	Error	Score
Rotation x (Degree)	0.000406	0.003880	3.474395e-03	1.0
Rotation y (Degree)	0.0	0.103902	1.039020e-01	1.0
Rotation z (Degree)	0.0	0.437708	4.377088e-01	1.0
Movement x (mm)	0.0	0.000249	2.499967e-04	0.999750
Movement y (mm)	0.0	9.293854e-06	9.293854e-06	1.0
Movement z (mm)	5.459082e-05	0.000198	1.434325e-04	1.0

Table 6.1: Metric filled with the data obtained from the iterations of experiment number one. The data consists of the mean of the iterations, both in simulation and reality.

Table 6.1 shows the results obtained from the first experiment, which consisted of picking the object and holding it for fifteen seconds more or less.

As it can be observed, the results are quite good. The final score (S_{total}), calculated as the mean of the results, achieves 0.999958 out of 1, indicating a very good result.

This experiment was the simplest one, meaning that it picks the object right in the centre and does not move nor shake the object, thus no movement was expected.

One interesting aspect of the results obtained in reality, that also remain in all the experiments, is that there is no zero reading even though the grasping object did not move. These phenomena is attributed to the noise of the Realsense. The simulated camera, on the other hand, does indeed show zero readings. Though, sometimes it also shows non-zero readings that could be attributed to both movement, and errors in the readings of ArUco markers.

During the experiment, no fall was produced, neither in simulation nor in the real world. The grasping object was picked correctly, and no movement was seen.

The video[1] showcasing the experiment in simulation and in reality is presented here.

6.2 Experiment #2

Metric	Simulation	Reality	Error	Score
Rotation x (Degree)	0.0	0.005185	5.185108e-03	1.0
Rotation y (Degree)	0.0	0.143813	1.438135e-01	1.0
Rotation z (Degree)	0.0	0.612239	6.122395e-01	1.0
Movement x (mm)	0.0	0.000349	3.499984e-04	0.999650
Movement y (mm)	0.0	1.402422e-05	1.402422e-05	1.0
Movement z (mm)	0.0	0.000239	2.395638e-04	1.0

Table 6.2: Metric filled with the data obtained from the iterations of experiment number two. The data consists of the mean of the iterations, both in simulation and reality.

Table 6.2 shows the results of the second experiment, which consisted of picking the grasping object by the left side and lifting it, for fifteen seconds.

The main fear in this experiment was that the grasping object would roll or rotate in simulation, while with the real robot it would not rotate. However, no rotation occurred, neither in simulation nor in reality.

Finally, the averaged result is 0.999941 out of 1, indicating again a great result.

The video of this experiment is available here.

6.3 Experiment #3

Table 6.3 shows the results of the third experiment, which has been essentially the same as the second experiment, but this time on the right side of the grasping object. In the same manner as before, the grasping object did not rotate, neither in simulation nor in reality.

The final score is 0.999916 out of 1.

The video of this experiment can be seen here.

6.4 Experiment #4

Table 6.4 shows the results of the fourth experiment, which consisted of picking the object and then shaking it, thus improving the probabilities of slipping or rotation. Compared

[1] If there is any problem regarding the videos, contact jaruiz.res@gmail.com

Metric	Simulation	Reality	Error	Score
Rotation x (Degree)	0.072503	0.048808	2.369534e-02	1.0
Rotation y (Degree)	0.0	0.083244	8.324434e-02	1.0
Rotation z (Degree)	0.0	0.433778	4.337781e-01	1.0
Movement x (mm)	0.0	0.000499	4.999995e-04	0.999500
Movement y (mm)	0.000163	0.000121	-4.209360e-05	1.0
Movement z (mm)	0.0	0.000129	1.297380e-04	1.0

Table 6.3: Metric filled with the data obtained from the iterations of experiment number three. The data consists of the mean of the iterations, both in simulation and reality.

Metric	Simulation	Reality	Error	Score
Rotation x (Degree)	0.025693	0.027583	1.889492e-03	1.0
Rotation y (Degree)	0.0	0.124534	1.245341e-01	1.0
Rotation z (Degree)	0.0	0.527753	5.277538e-01	1.0
Movement x (mm)	0.0	0.000399	3.999948e-04	0.999600
Movement y (mm)	0.001146	7.045771-05	-1.075949e-03	1.0
Movement z (mm)	0.000709	0.000175	-5.341685e-04	1.0

Table 6.4: Metric filled with the data obtained from the iterations of experiment number four. The data consists of the mean of the iterations, both in simulation and reality.

to other tests, it is visible that the results show more variation specially in simulation, meaning that there are more non-zero values. This could be attributed to the fact that it may have moved a little in the shaking and that it sometimes the markers may have not been identified correctly.

The final score for the fourth experiment is 0.999933 out of 1.

The video showing this experiment both in simulation and reality can be seen here.

6.5 Experiment #5

Table 6.5 shows the results of the fifth experiment. Recall that this experiment consisted of making the grasping object collide with an obstacle. This experiment was the most interesting and the one with more uncertainty, as it involves interaction with another object.

Metric	Simulation	Reality	Error	Score
Rotation x (Degree)	30.451138	33.619538	3.168399	0.975635
Rotation y (Degree)	0.0	1.105891	1.105891	0.0
Rotation z (Degree)	0.0	0.710522	0.710522	1.0
Movement x (mm)	0.0	0.005900	0.005900	0.994100
Movement y (mm)	0.175236	0.191545	0.016308	1.0
Movement z (mm)	0.022764	0.040893	0.018129	1.0

Table 6.5: Metric filled with the data obtained from the iterations of experiment number five. The data consists of the mean of the iterations, both in simulation and reality.

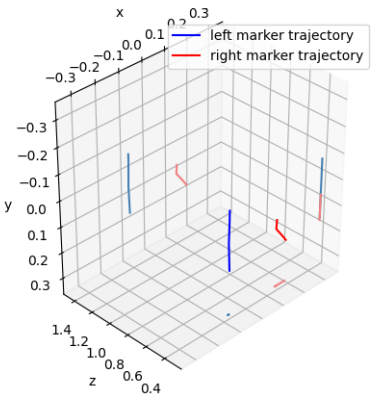
The results are very interesting. The grasping object did not fall in any iteration, nor in reality nor in simulation. Interestingly enough, the grasping object did rotate more in reality than in simulation, which may result counter-intuitive, as at the very first moments of the project, without any type of parameter modelling, the contacts were slippery.

The scores obtained are very good, and show that the behaviour can be modelled in a quite precise way, even when colliding or interacting with other objects in the environment.

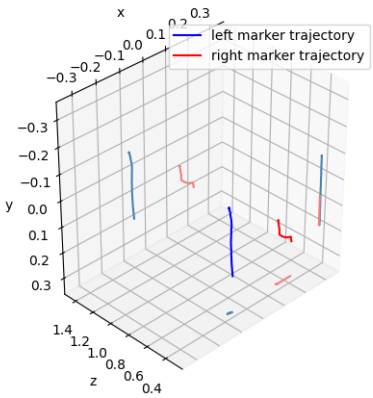
The final score of the fifth experiment is 0.828289 out of 1, which is not bad, even though it is the lowest obtained in all the experiments. This fact was expected, as it was the only one that interacted with other elements of the environment. The score may view itself lowered because of the 0.0 scored in the Rotation y metric. If we lowed the value of that metric using the weighted approach, giving a weight of 1.5 to the Rotation in X and reducing to 0.5 the Rotation in y, the final score would be better; 0.909592.

The video regarding this experiment can be seen here.

As mentioned, in addition to the metric, the information processing script also outputs figures portraying the trajectory of the markers in camera coordinates. Figure 6.1 shows an example of this, captured in the first iteration of the fifth experiment in simulation. In the figure, it can also be observed a little discrepancy in the trajectory. Even if the end point is nearly the same, the trajectory seems to differ a little. This could be because of a combination of various factors, such as the obstacle in the real world trembling a little when the contact was made, the camera noise, and different frequency in image acquisition. It is also noteworthy that in further iterations of the same experiments in the real world, such as seen in Figure 6.2, the trajectory seems to be closer to the one in simulation.



(a) Simulated trajectories of first iteration in experiment five.



(b) Real trajectories of first iteration in experiment five.

Figure 6.1: Tridimensional representation of the ArUco marker's trajectories in camera coordinates. This has been taken from the first iteration of the fifth experiment, both in simulation and in the real.

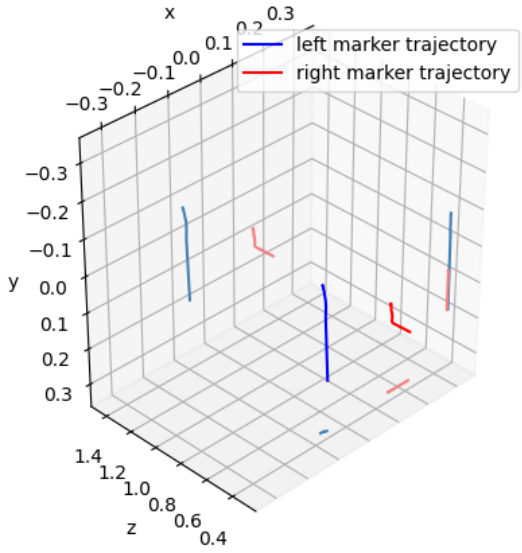


Figure 6.2: Trajectory of the markers in experiment 5, iteration 5 in the real world. Note that, compared to the trajectories seen in Figure 6.1, the trajectory is closer to Figure 6.1a than to Figure 6.1b.

CHAPTER 7

Conclusions and Further Work

Throughout this work it has been demonstrated the viability of developing a realistic simulation environment for manipulation, integrated within the ROS2 environment thanks to the development of Unity ROS2 Control and the node creation system of Unity Robotics Hub. This last technology shows great potential in terms of the implementation of custom messages and services. Unity ROS2 Control has shown great results when controlling the robot in simulation using MoveIt2, which is a great way to send control commands to the arm.

MuJoCo, thanks to its depth regarding the options and parameters available for modelling elements and the collisions between them, has shown great results. It must be noted that the modelling of parameters is indispensable in order to achieve a realistic simulation. Even though it has not been shown in this work, the default set of parameters of MuJoCo did not offer great results, even not being able to pick the grasping object at first instance. It has been demonstrated empirically that the modelling of several key parameters has been critical in order to achieve this level of realism. The set of parameters that have been identified as important in order to avoid slippages are the same as the ones presented in the documentation, which consist of using the Newton solver with elliptic friction cones and large value of impratio. In addition, further research and tests have proven that in order to model a bouncy object correctly, a good combination of the two coefficients in the *solref* parameter must be set, using the (-stiffness, -dampratio) mode.

The ArUco markers have proven to be a cheap, fast and easy to implement and in general terms a great technology to measure the reality gap. The Realsense Python library has also facilitated the tridimensional reconstruction of the marker's positions in camera coordinates.

The employed metric has shown great potential, most notably in the fifth experiment, the collision related one. The score system has shown its potential to intuitively measure the reality gap and portray them in a numerical scale.

It can be concluded that the development of a realistic simulation has been correctly achieved. The results are promising, with very high accuracy in the experiments, and most importantly, without any critical failures, such as pronounced vibration of the grasping object or a failed grasp (the grasping object falling). An initial set of parameters has been

modelled to work with the current configuration in a reliable way.

7.1 Further Work

Although we obtained promising results, some steps are needed in order to achieve a more general pick and place realistic simulation tool.

Regarding the metric, only the position of the grasped object has been considered for measuring the gap. We consider this approach a good initial hint. However, the differences in the arm trajectories or the trajectory of the object itself would give a wider perspective about the fidelity of the simulation. In addition, further experiments where collisions are produced between objects should be carried out, as it has been shown that it could be a very interesting approach to measure the reality gap.

With respect to the scale of the simulation, different grasping objects should be considered. This is not a so straightforward step, since the integration of uneven or non-convex pieces would require the use of software that decomposes the geometry into convex subgeometries that satisfies the shape of the piece. This can be seen in the *obj2mjcf* conversor, that takes an OBJ file and transforms to MJCF, but most importantly, when *V-HACD* is installed, it uses the package's utilities to obtain the decomposition of the geometry in convex geometries. The package should also be used in order to decompose the arm's collision geometries, with the objective of improving the accuracy.

Adding to that, the definition of the physical aspects of new grasping objects and materials, i.e. friction and solver parameters (*solref* and *solimp*) of the new grasping objects is not direct either, and would require further testing.

The simulation environment could also benefit from a supply of different grippers and arms. This would be notably harder to implement, as it would require changes within the arm's control interface in Unity. It should also be mentioned that the implementation of new grippers, for example a three finger gripper could result in new unexpected behaviours, as on this project only a two finger gripper has been tested.

Nevertheless, the next immediate step should be optimising the set of parameters, by taking profit of AI algorithms, namely genetic algorithms such as PyGAD. This action surely will help to reduce the sim to real gap.

Bibliography

[1] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. A review of physics simulators for robotic applications. *IEEE Access*, 9:51416–51431, 2021. See pages 1, 7, and 9.

[2] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study. *Journal of Software Engineering for Robotics, 5(1):3-16*, May 2014. See pages 4, 10.

[3] Michael Görner, Robert Haschke, Helge Ritter, and Jianwei Zhang. Moveit! task constructor for task-level motion planning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 190–196, 2019. See page 4.

[4] Gazebo simulator. https://gazebosim.org/home. See page 4.

[5] Jack Collins, David Howard, and Jurgen Leitner. Quantifying the reality gap in robotic manipulation tasks. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6706–6712, 2019. See pages 4, 11.

[6] Jack Collins, Ross Brown, Jurgen Leitner, and David Howard. Traversing the reality gap via simulator tuning. *arXiv preprint arXiv:2003.01369*, 2020. See pages 4, 11.

[7] Raj Kolamuri, Zilin Si, Yufan Zhang, Arpit Agarwal, and Wenzhen Yuan. Improving grasp stability with rotation measurement from tactile sensing. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6809–6816, 2021. See pages 4, 11, and 12.

[8] Mirella Santos Pessoa De Melo, José Gomes da Silva Neto, Pedro Jorge Lima Da Silva, João Marcelo Xavier Natario Teixeira, and Veronica Teichrieb. Analysis and comparison of robotics 3d simulators. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pages 242–251. IEEE, 2019. See page 7.

[9] Florent P Audonnet, Andrew Hamilton, and Gerardo Aragon-Camarasa. A systematic comparison of simulation software for robotic arm manipulation using ROS2. *22nd International Conference on Control, Automation and Systems (ICCAS 2022), BEXCO, Busan, Korea, pp. 755-762. ISBN 9788993215243 (doi: 10.23919/ICCAS55662.2022.10003832)*, 27 Nov-1 Dec 2022. See page 7.

[10] Se-Joon Chung and Nancy Pollard. Predictable behavior during contact simulation: a comparison of selected physics engines. *Computer Animation and Virtual Worlds*, 27(3-4):262–270, 2016. See pages 7, 8.

[11] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, Havok, MuJoCo, ODE and Physx. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4397–4404, 2015. See pages 7, 8.

[12] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994. See page 7.

[13] Usability 101: Introduction to usability. https://www.nngroup.com/articles/usability-101-introduction-to-usability/. Accessed: 2023-01-24. See page 7.

[14] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. See pages 8, 9.

[15] Physx simulator. https://developer.nvidia.com/physx-sdk. See page 8.

[16] Bullet real-time physics simulation. https://pybullet.org/wordpress/. Accessed: 2023-01-23. See page 8.

[17] Havok physics engine. "https://www.havok.com/". See page 8.

[18] Ode physics engine. "https://www.ode.org/". See page 8.

[19] Christoph Bartneck, Marius Soucy, Kevin Fleuret, and Eduardo B Sandoval. The robot engine—making the unity 3d game engine work for hri. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 431–437, 2015. See page 8.

[20] Unity documentation - physics. https://docs.unity3d.com/Manual/PhysicsSection.html. Accessed: 2023-01-27. See page 8.

[21] Nvidia physx. https://www.nvidia.com/en-us/drivers/physx/physx-9-19-0218-driver/. Accessed: 2023-01-27. See page 8.

[22] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations. In *Proceedings of Robotics: Science and Systems (RSS)*, 2018. See page 8.

[23] Kendall Lowrey, Svetoslav Kolev, Jeremy Dao, Aravind Rajeswaran, and Emanuel Todorov. Reinforcement learning for non-prehensile manipulation: Transfer from simulation to physical system. In *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 35–42, 2018. See page 9.

[24] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020. See page 9.

[25] Xibai Lou, Yang Yang, and Changhyun Choi. Collision-aware target-driven object grasping in constrained environments. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6364–6370, 2021. See page 9.

[26] Boris Bogaerts, Seppe Sels, Steve Vanlanduit, and Rudi Penne. Connecting the coppeliasim robotics simulator to virtual reality. *SoftwareX*, 11:100426, 2020. See page 9.

[27] Jeff Chen, Tori Fujinami, and Ethan Li. Deep bin picking with reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR*, volume 80, pages 1–8, 2018. See page 9.

[28] Yusuke Urakami, Alec Hodgkinson, Casey Carlin, Randall Leu, Luca Rigazio, and Pieter Abbeel. Doorgym: A scalable door opening environment and baseline agent. *arXiv preprint arXiv:1908.01887*, 2019. See pages 9, 11.

[29] Naijun Liu, Yinghao Cai, Tao Lu, Rui Wang, and Shuo Wang. Real–sim–real transfer for real-world robot control policy learning with deep reinforcement learning. *Applied Sciences*, 10(5):1555, 2020. See page 9.

[30] Kyushik Min, Hyunho Lee, Kwansu Shin, Taehak Lee, Hojoon Lee, Jinwon Choi, and Sungho Son. Jorldy: a fully customizable open source framework for reinforcement learning. *arXiv preprint arXiv:2204.04892*, 2022. See page 9.

[31] Nestor Gonzalez Lopez, Yue Leire Erro Nuin, Elias Barba Moral, Lander Usategui San Juan, Alejandro Solano Rueda, Víctor Mayoral Vilches, and Risto Kojcev. gym-gazebo2, a toolkit for reinforcement learning using ros 2 and gazebo. *arXiv preprint arXiv:1903.06278*, 2019. See page 9.

[32] Vikash Kumar and Emanuel Todorov. Mujoco haptix: A virtual reality system for hand manipulation. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 657–663. IEEE, 2015. See pages 9, 10.

[33] Zerosim. https://github.com/fsstudio-team/ZeroSimROSUnity. Accessed: 2023-01-27. See page 10.

[34] Cyberglove. http://www.cyberglovesystems.com/. Accessed: 2023-01-31. See page 10.

[35] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019. See pages 10, 11.

[36] Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Transactions on Evolutionary Computation*, 17(1):122–145, 2012. See page 10.

[37] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-Real: Learning Agile Locomotion For Quadruped Robots. *Robotics: Science and Systems (RSS)*, 2018. See pages 10, 11.

[38] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in Artificial Life: Third European Conference on Artificial Life Granada, Spain, June 4–6, 1995 Proceedings 3*, pages 704–720. Springer, 1995. See page 10.

[39] Aleksi Ikkala and Perttu Hämäläinen. Converting biomechanical models from opensim to mujoco. In *Converging Clinical and Engineering Research on Neurorehabilitation IV: Proceedings of the 5th International Conference on Neurorehabilitation (ICNR2020), October 13–16, 2020*, pages 277–281. Springer, 2022. See page 11.

[40] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11(1):1–18, 2003. See page 11.

[41] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30, 2017. See page 11.

[42] Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning. In *arXiv preprint arXiv:2009.12293*, 2020. See page 11.

[43] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 12627–12637, 2019. See page 11.

[44] Jack Collins, Jessie McVicar, David Wedlock, Ross Brown, David Howard, and Jürgen Leitner. Benchmarking simulated robotic manipulation through a real world dataset. *IEEE Robotics and Automation Letters*, 5(1):250–257, 2019. See page 11.

[45] Johan Fabry and Stephen Sinclair. Interactive visualizations for testing physics engines in robotics. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 106–110, 2016. See page 11.

[46] Wenzhen Yuan, Rui Li, Mandayam A Srinivasan, and Edward H Adelson. Measurement of shear and slip with a gelsight tactile sensor. In *2015 International Conference on Robotics and Automation (ICRA)*, pages 304–311. IEEE, 2015. See page 11.

[47] Deen Cockburn, Jean-Philippe Roberge, Alexis Maslyczyk, Vincent Duchaine, et al. Grasp stability assessment through unsupervised feature learning of tactile images. In *2017 International Conference on Robotics and Automation (ICRA)*, pages 2238–2244. IEEE, 2017. See page 12.

[48] Rocco A Romeo and Loredana Zollo. Methods and sensors for slip detection in robotics: A survey. *Ieee Access*, 8:73027–73050, 2020. See page 12.

[49] Siyuan Dong, Wenzhen Yuan, and Edward H Adelson. Improved gelsight tactile sensor for measuring geometry and slip. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 137–144, 2017. See page 12.

[50] Deirdre Quillen, Eric Jang, Ofir Nachum, Chelsea Finn, Julian Ibarz, and Sergey Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6284–6291, 2018. See page 12.

[51] Gert Kootstra, Mila Popović, Jimmy Alison Jørgensen, Danica Kragic, Henrik Gordon Petersen, and Norbert Krüger. Visgrab: A benchmark for vision-based grasping. *Paladyn*, 3:54–62, 2012. See page 12.

[52] Guoguang Du, Kai Wang, Shiguo Lian, and Kaiyong Zhao. Vision-based robotic grasping from object localization, object pose estimation to grasp estimation for parallel grippers: a review. *Artificial Intelligence Review*, 54(3):1677–1734, 2021. See page 12.

[53] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, and Manuel Jesús Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014. See page 12.

[54] Mohammad Fattahi Sani and Ghader Karimian. Automatic navigation and landing of an indoor ar. drone quadrotor using aruco marker and inertial sensors. In *2017 International Conference on Computer and Drone Applications (IConDA)*, pages 102–107, 2017. See page 12.

[55] Jan Bacik, Frantisek Durovsky, Pavol Fedor, and Daniela Perdukova. Autonomous flying with quadrocopter using fuzzy control and aruco markers. *Intelligent Service Robotics*, 10:185–194, 2017. See page 12.

[56] Unity Robotics. Unity robotics hub. https://github.com/Unity-Technologies/Unity-Robotics-Hub, 2022. See page 12.

[57] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. See page 13.

[58] MuJoCo Menagerie Contributors. MuJoCo Menagerie: A collection of high-quality simulation models for MuJoCo, September 2022. See page 19.