BACHELOR

Robotics in VR

Connecting Unity and Simulink through ROS

Vissers, Carolina M.E.

Award date:
2023

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Mechanical Engineering

4WC10 Bachelors Final Project - CST

2022-2023 quartiles 1 & 2

Robotics in VR

Connecting Unity and Simulink through ROS

Eindhoven, 30th January 2023

Author:
Vissers, Carolina 1415557

Supervisors:
Prof. Dr. Elena Torta TU/e

Contents

List of Abbreviations

AMBER AdvanceMents in BEhavioral autonomy for medical Robots

API Aplication Programming Interface

AR Augmented Reality

CAVE Cave Automatic Virtual Environment

DH Denavit–Hartenberg

EE end effector

FK Forward Kinematics

GUI Graphical User Interface

HMD Head Mounted Device

IK Inverse Kinematics

IP Internet Protocol

LH left handed

MAPE Mean Absolute Percent Errors

MQTT MQ Telemetry Transport

RH right handed

ROS Robot Operating System

SRC Soft Robotics Control-unit

TCP Transmission Control Protocol

UI User Interface

URDF Unified Robotics Description Format

VR Virtual Reality

WSL Windows Subsystem for Linux

XR Extended Reality

1 Introduction

Virtual Reality (VR) is an increasingly popular and accessible means of visualization and offers increased immersion when compared to 2D video or 3D video [1]. Therefore, it becomes attractive to consider using VR in order to influence the development process of robots by requesting user feedback before the robot is fully finished. This can reduce development time and cut back on machine time for hardware tests, thus reducing expenses. It can also be influential by removing early prototyping and transport costs when presenting the robot to a stakeholder. The purpose of this BEP is to investigate implementation methods in which the motion of a robot can be displayed to a user in VR. This assignment was performed in collaboration with Philips for the AdvanceMents in BEhavioral autonomy for medical Robots (AMBER) project.

The robot was provided by Philips IGT. This robot is a medical robot and is capable of performing nonmedical movements autonomously. An example of this is approaching a patient autonomously and waiting for the operator (i.e., doctor, nurse, etc.) to further fine tune the placement of the robot. Placing the robot in VR allows testing of the robot's autonomous movements and receiving feedback on the robot from the operators without requiring a physically present robot.

Therefore, there is the intention of using VR as a way to receive user feedback about the robot's autonomous movements. However, there is a question on if a robot modeled in VR can be considered equivalent to a real robot. Additionally, it needs to be investigated if existing control structures are capable of interfacing with Unity. These control structures exist in Simulink for previously developed physical hardware control, so Unity must be able to receive input from Simulink. The requirements for this BEP are as follows:

1. The robots used will be the Franka Emika Panda robot and the Philips IGT robot
2. The robot shall be simulated in Unity and be visible to the user via a VR Head Mounted Device (HMD).
3. The simulated robot shall move based on joint positions it receives.
4. Simulink will generate the joint positions and communicate them to Unity.
5. The manner of Unity to Simulink and Simulink to Unity communication shall be investigated and implemented.
6. Throughout the project, alignment with the stakeholders from Philips IGT and various master students adjacent to the project will be maintained.
7. Support for future implementation of robots in VR will be given in the form of a wiki and C# scripts that interface with any robot in Unity.

Due to the development stage of the Philips IGT robot, this robot is confidential and cannot be made public. In order to stimulate usage of the research and create more VR simulated robots, this project has been created in a repository for the Franka Emika Panda robot. This report will primarily focus on the Panda robot and not the Philips IGT robot. Additionally, to further ease implementation of future robots in VR, universal C# scripts for Unity have been created that allow for control from Simulink to any robot within Unity. An identical private repository has been created for the Philips IGT robot. The Panda repository contains a folder for the associated Matlab and Simulink files, the Unity Projects for the VR and the non-VR 3D version of the Panda robot, and a folder containing the universal C# scripts for further development of other robots [2][1]. A wiki has been created detailing the implementation steps of an arbitrary robot in VR [3][2].

[1]Franka Emika Panda Repository: `https://gitlab.tue.nl/et_projects/cv-robot-unity-vr`
[2]Wiki for robot creation: `https://gitlab.tue.nl/et_projects/cv-robot-unity-vr/-/wikis/home`

2 Perception of a VR Robot

In the goal to receive user and stakeholder feedback via VR, it is essential that the final product of the physical robot is perceived identically to the VR version. Otherwise, the VR based feedback may even negatively influence design of the final product. Therefore, a literature review and a discussion with experts in the field were done to gain an overview of existing methods and results for using VR for robot design. Based on these outcomes, a research proposal is created for a future study.

2.1 Literature Review

Much of the research done on VR vs reality and robotics uses social robots [1, 4, 5], training environments [6], and industrial robots [7, 8]. In the 2000s and early 2010s much research was done on perception of (social) robots in reality vs via a (2D) computer screen [9, 10, 11]. This has since expanded into research on different forms of 3D VR including 3D videos on 2D screens, HMD, or a Cave Automatic Virtual Environment (CAVE) system, where stereoscopic glasses and projector screens are used to create a realistic sensation to the user [7, 8, 12]. The articles that use social robots focus on human-robot interaction, whereas articles focusing on industrial robots focus on human-robot collaboration. This distinction is necessary as with industrial robots the same levels of empathy, personification, and interpersonal connection are not needed. However, trust, perceived control, and reliability of the robot when collaborating are still very useful factors.

One study [1] comparing the perception of a humanoid social robot compared the same robot in real life, in VR with a HMD, via a 2D screen, and via a 3D video on a 2D screen found that immersive VR via a HMD and physically present robot has no significant difference in perceived immediacy, whereas with 3D and 2D video there was a marked decrease.

Kamide et al. [5] looked at a humanoid robot as it autonomously approached a person from different directions and the preferred amount of personal space the user had. Users could indicate when the robot encroached on their personal space via a button press. It compared a real robot via a simulated robot in VR via a CAVE system. There were no significant differences in preferred distance in VR vs reality. Participant's height in this study also did not play an effect. Six different dimensions of perceiving the robot were also studied. The robot's perceived ability to perform useful tasks and its potential to communicate were rated lower in VR than in reality. The perception of a robot being under human control (and the idea that the robot would not start to act on its own) was higher in VR. This difference is attempted to be explained by Kamide et al. "Real robots require skilled operators to control them and resolve any mechanical problems that may occur. The general population is not likely to possess the skill set to deal with such difficulties. While the CAVE system used with the VR robot also requires a skilled operator, the participants simply observed the 3D image of the VR robot. In other words, the participants could easily observe the mechanical system (e.g., motors, electric cables) inherent to the real robot, but not the equipment supporting the VR robot." The perceptions of vulnerability, clumsiness of motion, and objective hardness were not different in VR when compared to the physically present robot.

Another study [7] compared a cooperative robot on an assembly line in VR with reality. It also tracked how these things changed when the robot was far away vs when the robot was close by. The study used user evaluations, but also physiological responses via heart rate and skin conductivity. It found that operators were more relaxed in the virtual simulation, likely due to the fact that they could not see the robot in their peripheral vision or sense it via sounds as much as in the real version. They likely perceived less threat from the VR robot and thus took the simulation less serious when compared to the physically present robot. This is backed up by the physiological data that saw a marked increase in skin conductivity when working with the real robot. This was

not seen in the VR setting.

2.2 Discussion

VR is thus a highly interactive and immersive alternative to reality. However, there are some key differences, especially as Kamide et al. [5] found that the robot was perceived as being more under human control. This was with the reasoning that the real robot still has exposed cables and motors, but the VR version does not. This suggests that the VR version should be as similar as possible to the real robot in order to not skew the perception of controllability. Controllability is perhaps the most important factor when comparing a medical robot to its VR version, as it comes so close to a person on the table and works autonomously.

Another concern comes from the cooperative robot on an assembly line, as the experiment results showed that in VR the immediacy of danger was not as high as in real life. This could potentially be improved by adding realistic sounds to the VR experience. This is important, as the closeness of the medical robot in VR should induce the same psychological and physiological responses as the real robot.

Applying these concerns to a real study on perception in VR vs reality can be done with the Franka Emika Panda robot. It can be considered as human-robot collaboration instead of human-robot interaction, as the Panda robot is not a humanoid robot. This can give the idea that the Panda arm is working to achieve a task, similar to a medical robot.

In a meeting on Thursday, the tenth of November 2022 with Dr. Irene Kuling, Dr. Femke van Beek, and Dr. Elena Torta possibilities for an experimental setup for this study were discussed. Kuling and van Beek both have recent and extensive experience in the field of robots in VR, especially around the question of perception. A number of new concerns arose, and numerous research questions were proposed. One of these concerns was of the fact that depth perception in VR is more difficult, so it may be harder to distinguish between velocities or accurately describe distances in VR. Also, the level of immersion can be immediately broken within the VR environment, as the user realistically knows that the robot cannot actually touch or hurt them or any simulated persons.

Many of the prior experiments fully, or mainly, relied on participant questionnaire feedback. However, care should be taken to generate tangible data that is comparable and not only based on user opinion, as the questionnaire questions can be leading and questionnaire data has an abundance of noise, so many participants are required. Testing the perception of the Panda robot should also be taken in a medical context. Therefore, similar to a medial robot, the Panda robot should approach a simulated patient autonomously. Then the experiment participant should attempt to give accurate judgment on a specific aspect of the robot. Options for this included comparing two different velocity profiles, judging distance between the robot end effector (EE) and the simulated patient, asking participants to state a preference for a specific approach path from the robot, or asking participants to stop the robot when they found it came too close to the patient via a button press. One thing that should also be noted that when changing the velocity of the robot, as the EE trajectory changes and/or the time to the final point changes. Keeping both constant is impossible, so care should be taken that if velocity based questions or tests are preformed, that participants rely on their perception of the robot's EE velocity, not the perception of time.

2.3 Research Proposal

For the research of operator perception of an autonomously approaching medical robot in reality and in VR, there are numerous research questions to be considered. This proposal considers the question of distance based perception. This is key as the operator must be comfortable with and confident in the autonomous robot that there will be no errors in its operation that are dangerous to

themselves, the patient, and anyone else present. Therefore, the research questions to be addressed are as follows:

1. What are the differences, if any, in operator perception of the distance between the robot and the (simulated) patient equal in VR and in reality?
2. What distance is considered too close to the (simulated) patient, and is there a difference between this in VR and in reality?
3. What is the preference of approach path of the robot towards the patient, and are they the same in VR and in reality?

The methodology to perform the experiment can be broken up into two smaller tests in order to acquire data on the three research questions. For all tests, a simulated patient will be present next to the Franka Emika Panda robot. In order to ensure that there are minimal differences between the reality situation and the VR situation, the VR room will be modeled after the room where the real robot is. The simulated patient in reality will place their hand on a table near the robot. A cloth will hand in front of the patient, so only their hand is visible. This is important as in the VR situation only a hand will be visible as well. This is to reduce bias between the visual differences of a person in reality vs a VR simulated person. Simulating only a hand in VR that looks realistic is more feasible than a full human upper body and head. During the experiment, the Franka Emika Panda robot hand will approach the patient's hand while it remains unmoving in both the reality and the VR scenario. The person in reality playing the role of the patient will practice keeping their hand still while the robot approaches to ensure this requirement is met. The participant in the experiment will be seated at a table close by the robot and the patient, but not in the moving path of the robot.

Participants who are selected for the experiment will be screened to ensure they have no prior experience working with the Franka Emika Panda robot in order to reduce bias of familiarity with the robot. Similarly, half the participants will experience the reality scenario first, and the other half will experience the VR scenario first. This experiment with be within-subject design where all participants are exposed to all conditions.

The first test relates to the first research question of perception of distance. In this test, the robot will approach the patient's hand and stop at some distance away from the hand. Then the participant will be asked to estimate the distance remaining between the hand of the robot and the hand of the patient, in order to acquire data on the accuracy and differences of depth perception in VR vs reality. This is done in the reality scenario by having the participants place two marker blocks on the table in front of them at the distance between the robot hand and the patient hand. Then the observer of the experiment will manually measure the distance between the two marker blocks. In the VR scenario, the participants will place their handheld controllers on the table in front of them at the same perceived distance. The observer will then request Unity to store the distance between the two handheld controllers. This distance can later be translated into measurable units. There will be ten different approach paths and final distance stopped combinations shown. For both scenarios, they will be the same ten combinations, but will be shown to the participant in a randomized order. These approach paths must still be developed in the future.

The independent variables from the first test are the VR vs reality scenario and the ten different approach paths. This results in twenty different test sets per participant. The dependent variables are the participant-perceived distance between the patient's hand and the robot's EE. The data from the first test will be analyzed in order to see if there is a statistically significant difference between the perceived distance of the robot hand to the patient hand. The perceived distances will also be compared to the actual distances in order to conclude whether the participants could accurately determine this distance. This depth perception test will be useful in concluding if VR based feedback can be considered similar to the feedback that would be given if the robot was seen

in reality instead.

The second test deals with the other two research questions. In order to experimentally determine the distance that crosses the perceived safe and comfortable distance to the simulated patient's hand, multiple approach paths at similar EE velocities will be tested in groups of three. Three sets of different EE velocities will be tested. During this, the desired end point will be at the top of the patient's hand. Participants will be given a button in the reality scenario and a VR hand controller in the VR scenario and will be told that when they press the button or the trigger on the hand controller, the robot will stop. They will be told that they should stop the robot when they feel the robot is too close to the patient's hand. It will be up to the participant to determine what this means to them. After each set of three different approach paths, participants will be asked to rate the preferred approach method. Then the next velocity profile will be used with the same approach paths and the experiment is repeated. For each separate approach to the patient, the time between the start that the robot moves and the time the participant indicates the robot is too close to the patient will be measured. This time can then be used to extrapolate how close the robot is to the hand.

The independent variables from the second test are the VR vs reality scenario, the three velocity sets, and the three approach paths per velocity set. This results in eighteen different test sets per participant. The dependent variables are the participant-perceived safe distance to the patient's hand that is extrapolated from the time the robot spent moving, and the participant's preferred approach path per EE velocity group. The data from the second test is useful in comparing what the participant experiences as too close to the patient in the VR scenario vs the reality scenario. It will also be useful to determine if the distance left before reaching the patient's hand correlates to the preferred approach path, and how this may change dependent on the velocity profile. Seeing if the data is statistically different between the two different scenarios or not will allow for evaluation of the perception of a robot in VR vs in reality.

With the total number of test sets per participant reaching thirty-eight separate tests spread over both proposed tests, this is a reasonable number of experiments to conduct. Each participant would spend approximately one hour to one hour and fifteen minutes in the experiment. This number is achieved by calculating one and a half minutes per test and allowing for setup of the real Panda robot, the VR environment, setting the participant up with the VR headset, and administrative tasks. One and a half minutes per test is a reasonable assumption, as simple movements with the Panda arm only take at most fifteen seconds without becoming extremely slow. This allots over a minute for the participant to answer relevant questions, which is reasonable.

3 ROS

The goal of this project is to communicate the desired movement of a robot from Simulink to Unity. In order to so this, a communication network must be setup. ROS is ideal for this as it has existing Matlab/Simulink and Unity support that can be directly applied to this project.

The Robot Operating System (ROS) is an open source operating system for building robotic applications on a software level. It is comprised of different software libraries and tools to allow for language independent robotic development that aims to allow for easy collaboration, research, code sharing, and code reuse. Via packing the ROS processes into packages or stacks, these robot developments can easily be shared for other open source purposes. ROS also focuses on scalability and ease of testing via built in testing frameworks. It is not a real-time framework, but can be integrated with real-time code. As it is free and open source, ROS can be used for any purpose by anyone. It also is allowed to be used in commercial projects [13].

3.1 Alternatives to ROS

In order to determine that ROS is the optimal solution for this project, various other options were considered. This includes a Transmission Control Protocol (TCP)/Internet Protocol (IP) connection, MQ Telemetry Transport (MQTT), and a direct connection via Matlab and Unity via scripts. The best option would allow for ease of implementation, ease of universality (i.e., working on many different robots, not just one specific option), and would preferably be free and open source.

Caasenbrood et al. [14] created a Soft Robotics Control-unit (SRC) development platform to communicate with Matlab, Simulink, and Unity with a pneumatic soft robot. In order to apply this to a hard robot that uses actuators in joints instead of actuators throughout the robot like in soft robotics, the TCP/IP protocol and would have to be adjusted. This was determined to be a potential option, but would require much new research and setup to develop the platform, thus violating the ease of implementation requirement.

Another option is to use MQTT (formerly known as MQ Telemetry Transport). MQTT is a useful messaging protocol used for the Internet of Things to publish and subscribe data designed for TCP/IP sockets [15]. Matlab has a MQTT Aplication Programming Interface (API) that works over ThingSpeak channels. These are free to use when staying under a certain limit of devices and channels [16]. It would be possible to use MQTT to communicate with Unity, but no highly supported solution for this exists. This option also runs into ease of implementation issues on the side of Unity, but much less so than the TCP/IP issues from the SRC platform. Also, the use of ThingSpeak channels violate the preference for a free solution may be violated.

The final investigated alternative is to create a direct TCP/IP connection between Unity and Matlab via a TCP client that publishes and subscribes data to the various listeners and publishers. Both Matlab and C# have support for this option, so it can be implemented [17]. This solution is similar to the SRC, but does not require the additional external hardware and can be hosted on the computer running the Simulink and Unity applications. This is likely the best of the three options mentioned in the subsection, but it may run into issues with universality when listening to the data sent within Unity and Matlab. It also requires the creation of data formats, whereas ROS already has preexisting data formatting in the form of messages. For these reasons, ROS was taken to be the preferred solution.

3.2 Setup

In order to run ROS, an Ubuntu distro on Linux is required [18]. The computers that were used during this project all have Windows 10 installed. In order to overcome this, the option to use Windows Subsystem for Linux (WSL) is used instead. This allows for GNU/Linux distros to be run within terminals, and also allows for Graphical User Interface (GUI) applications to be run on later Windows install versions [19]. Specifically, for this project, WSL2 was used to run Ubuntu 20.04 with ROS Noetic Ninjemys installed.

Within the Ubuntu distro, a catkin workspace was created and the ROS-TCP-Endpoint package was added. The ROS-TCP-Endpoint package allows for communication with the ROS-TCP-Connector in Unity, as discussed in subsection 4.2.

In order to start up a ROS connection, a master must be started via Code A.1. Then, a node must be started to connect with Unity. The ROS-TCP-Endpoint package can be easily used to set up the node required, as seen in Code A.2. Later connection with the Simulink node is handled internally by Simulink. Further information on setting up WSL, Ubuntu, and ROS are found in the wiki [3].

3.3 ROS Concepts

Within ROS there are varying levels of organization aimed at allowing for effective communication to and from the robot. This hierarchy starts with the master, where the nodes are hosted. Within the master, nodes can be started via a client library in order to perform computations. This client library is a collection of code that allows for the ROS concepts to be programmable in a specific language. In case of the ROS-TCP-Endpoint package, this is Python.

Within a node, or between various nodes, messages can be sent. Messages are data structures that contain any number of data types. These messages are sent via topics. Topics are names that are used to identify the data that is sent over it. It uses publishers and subscribers to add or receive information from the topic. A node can subscribe to any topic, and a node can publish to any topic, as long as it sends the correct data structures to the topic. A node can publish and subscribe to various topics.

3.4 ROS Structure

For the panda arm, ROS has been used to include two nodes and four topics communicating between these nodes. The nodes, seen in Figure 1 as diamonds, are registered to the ROS Master, seen as a circle. Both a node for Unity and a node for Simulink are registered within the ROS Master. Both nodes are automatically generated by their respective applications when running the simulations. The Simulink node shuts itself down once the Simulink is no longer running. This behavior is part of the ROS Toolbox from Matlab and allows for easy transition between various Simulink files. The Unity node stays online until the ROS Master shuts down.

The topics, seen in Figure 1 as rectangles, each have one publisher and one subscriber. Which node publishes or subscribes to which topic can be seen via the blue and red arrows respectively. Simulink provides the majority of information and so publishes to three of the topics, while Unity publishes to one of the topics.

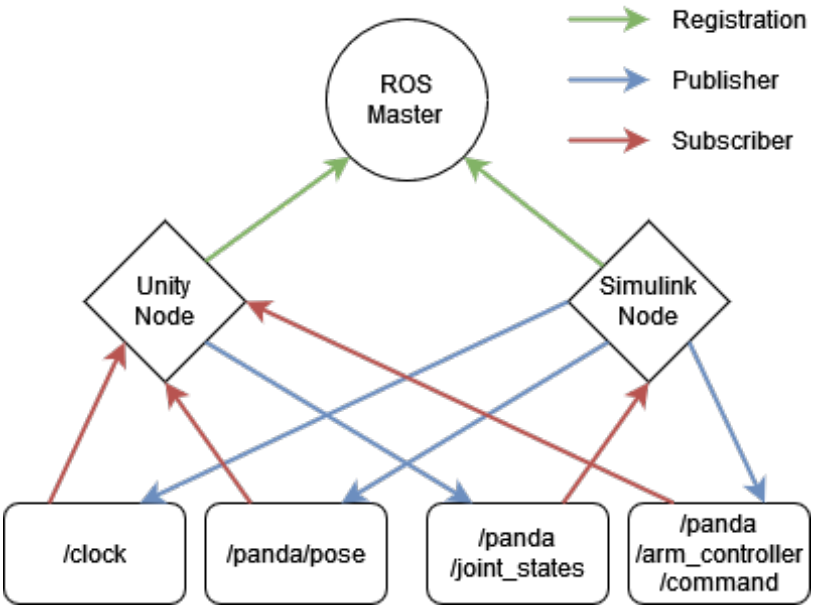


Figure 1: ROS Structure for Panda Arm

3.5 Topics

As topics themselves send messages, i.e. data, from the publisher to the subscriber, there are many ways this data can be formatted. ROS has many standardized message types, and both Matlab and C# support their use. For each topic, various different message types are used.

The topic /clock uses the rosgraph_msgs/Clock.msg message type. It is used for communicating the currently simulated time [20]. Simulink publishes the current simulation time to this topic and the Unity node subscribes to it. This allows for the simulation to be run with the same time. If this message would not be used, the wall clock time would be used. Using simulation time instead avoids time discrepancy issues in case the Simulink simulation runs slower than real time. Simulink publishes to /clock with the time in floor(seconds) and in floor(nanoseconds) [21].

The topic /panda/pose is a topic that is optional. It is currently used in the Simulink files to visualize a user defined desired EE location and, if applicable, a desired end-effector rotation. The pose message type uses the geometry_msgs/Pose.msg which is made up of the geometry_msgs/Point.msg type which uses XYZ float64 points to send a position and the geometry_msgs/Quaternion.msg which sends float64 values for a xyzw quaternion [22] to send an orientation. However, due to the fact that this ROS topic communicates between a right handed (RH), z-up coordinate system in Matlab and a left handed (LH) y-up coordinate system, this translation is prone to errors, especially with different robot models following different conventions. The change between a RH and LH coordinate system can be seen in Figure 2. Therefore, the point sent is in Cartesian format of RH, Z-up, and the orientation is not based on a quaternion but uses Euler angles to a xyz orientation also in RH, Z-up coordinate system. Later, this point is translated to an LH, Y-up system as discussed in subsection 4.4 within the Unity scripts found in the Panda repository [2]. This is the only part where the change in coordinate systems become a consideration. As it is also optional to display, a design choice could be made not to display the desired EE position in Unity at all.

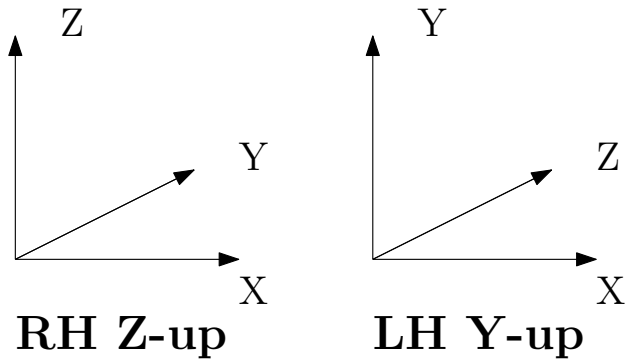


Figure 2: Difference between RH z-up and LH y-up coordinate systems

The topic /panda/joint_states is the only topic where the Unity node publishes the data and the Simulink node subscribes to it. Here the sensor_msgs/JointState.msg type is used. Within this message, a Header submessage, a sting list for the link names, and float64 lists for joint positions, velocities, and efforts are recorded. Effort means the force on a prismatic joint or the torque on a revolute joint. The header message includes a sequence id that increases with every message published by one, as well as a time stamp and a string of frame id are sent [23]. The joint state topic publishes the achieved joint position and the joint velocity from the simulated robot in Unity. It also fills in the header messages and the list of joint names. This data is used by the subscriber in Simulink in order to log the data and then later perform validation checks.

Finally, the topic /panda/arm_controller/command is similar to the /panda/joint_states as they use similar message types. Here, the trajectory_msgs/JointTrajectory.msg type is used. Within this message, a Header submessage and a JointTrajectoryPoint submessage, of message type trajectory_msgs/JointTrajectoryPoint.msg, are used. Also, a list of strings of joint names is sent. The header message includes a sequence id that increases with every message published by one, as well as a time stamp and a string of frame id are sent. The JointTrajectoryPoint submessage uses float64 lists of joint positions, velocities, accelerations, and effort. Also, a duration time from start can be sent [24]. The use of this topic is to communicate the desired joint position from Simulink to Unity. The option for various different lists of joint trajectory points allows for robots to be manipulated based on joint position, velocity, acceleration, or force/torque. However, Unity Articulation Bodies do not allow for joint acceleration control, which are the components that enable the Unity physics engine to move the joints to the specified positions. This is further discussed in subsection 4.5.

4 Unity

Unity is a development platform for 2D and 3D games that has expanded into digital twins, film and animations, architecture, and many more industry applications. It can be used for free for small users and has paid solutions for enterprises and businesses.

For this project, a 3D project is used, which can be expanded to use VR. The packages used will be discussed, as well as general robot setup with the implemented scripts. There are some issues with the Unity physics where the robot joints do not reach the desired positions. These are analyzed and solutions to this problem are proposed. Afterward, the VR implementation is discussed.

4.1 Introduction to Unity

The game engine Unity incorporates many different items in it in order to allow for a very wide range of options for the user to create a game, simulation, or whatever the goal of the project it is. Game engines incorporate 3D models, textures, audio, scripts, animation, 2D sprites, and many more objects that can be used to create the desired experience. Unity has the goal of being real time, meaning that images are rendered in real time as the scene changes. This is opposed to offline rendering, which may take hours or days. Offline rendering may result in a higher quality image, but with decent graphics support in the computer, Unity is able to achieve real time rendering at a high quality level.

When working within Unity, the program can have a lot of new features and a different layout than what one may be used to. In Figure 3, the layout of the Unity editor for the 3D project of the Panda Robot can be seen.

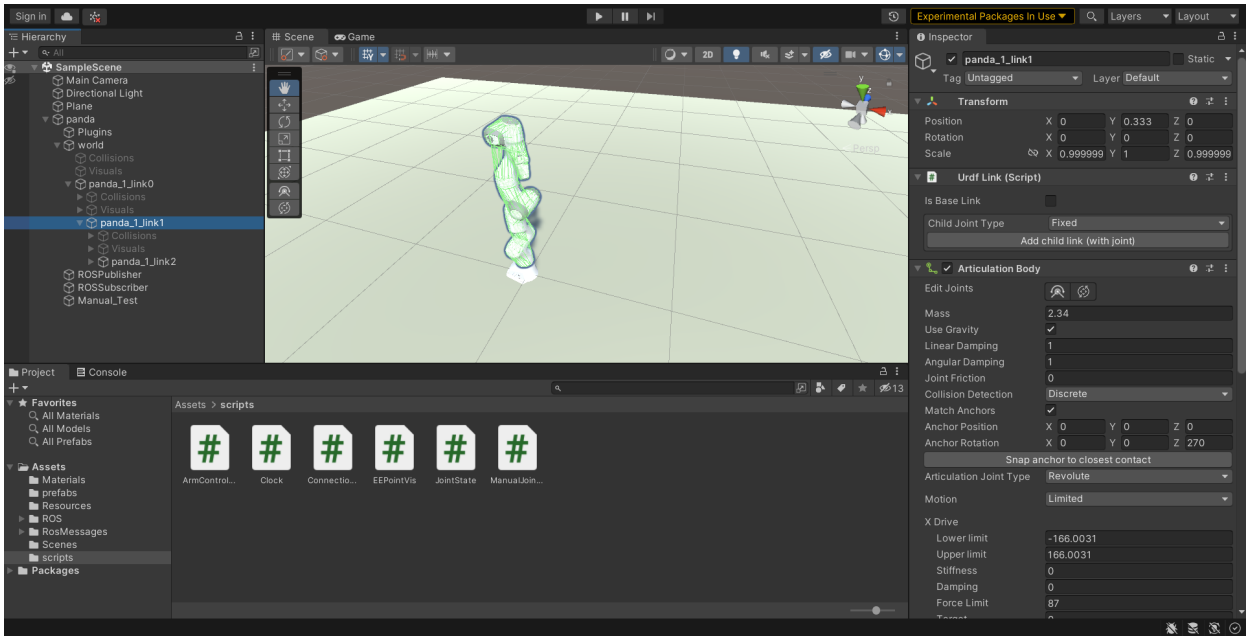


Figure 3: Layout of the Unity Project 2_Unity_3D

In the center is the currently open Scene, which is what the user of the published project would see when interacting with the program. On the left is the Hierarchy, in which all the GameObjects in the Scene are included. GameObjects can have visual aspects included, like the Plane GameObject, or can be empty and thus invisible, like the ROSPublisher GameObject. The Main Camera and the Directional Light are defaults when creating a new project. The camera allows for the user

to view the scene, and the light creates shadows. A plane has been added to give the illusion of a floor. Then comes the Panda robot. The robot has many children, including the links that it incorporates, and the visuals that are associated with these links. The Panda also has three empty GameObject children, as further explained in subsection 4.4.

As can be seen, The Panda_1_link1 is selected in the Hierarchy. The opens the GameObject in the Inspector pandel, seen on the right-hand side of the Unity Editor. All GameObjects have a Transform section, where its position, rotation, and scale can be seen. Another important aspect of this GameObject is the Articulation Body, which gives the physical properties of the robot link and associated joint.

At the bottom of the Unity Editor, the Project tab can be seen. This is a sort of file overview of all the files in the project. This holds all the Assets and Packages. The packages are further described in the next section. The Assets are all the parts that are most relevant for the project. As can be seen, the scripts folder is selected and all six scripts for this project can be seen. The other folders are materials, which hold all the information for visual materials, like the color of the floor plane and colors of the prefabs. The Prefabs folder holds Game Objects that are used regularly and need to be spawned in or out of the game at any time. Here, the EE visualization block is stored, as discussed later. The Resources and RosMessages folder hold generated scripts and items stemming from packages. These scripts are generated by the ROS-TCP-Connector package The ROS folder holds the Unified Robotics Description Format (URDF) and visualization files for the Panda arm. Finally, the Scenes folder holds the different Scenes in a project. This project only has one scene, but there can be many more created. Then all that needs to be added is the relevant items back into the Hierarchy, but the Assets will be usable across all Scenes.

At the very top of the Unity Editor, in the center, there is a play button that will run the Scene. This enables one of the most important features of Unity, where the project is run in Play Mode and goes through the scripts, allows for user input and interaction, and whatever other options are enabled within Unity. The Panda robot will only follow ROS commands and move autonomously when Play Mode is on. Finally, the Panda robot referred to throughout the report can be seen in Figure 4, where the initial start position can be seen in Play Mode. This start position is taken from the URDF file. However, it should be noted this start position is not a physically possible joint orientation, as the robot partially clips though itself. Unity does not have issues with this, as there are no physical colliders between various parts of the robot. This must however be taken into consideration in section 5 within the Matlab/Simulink files.

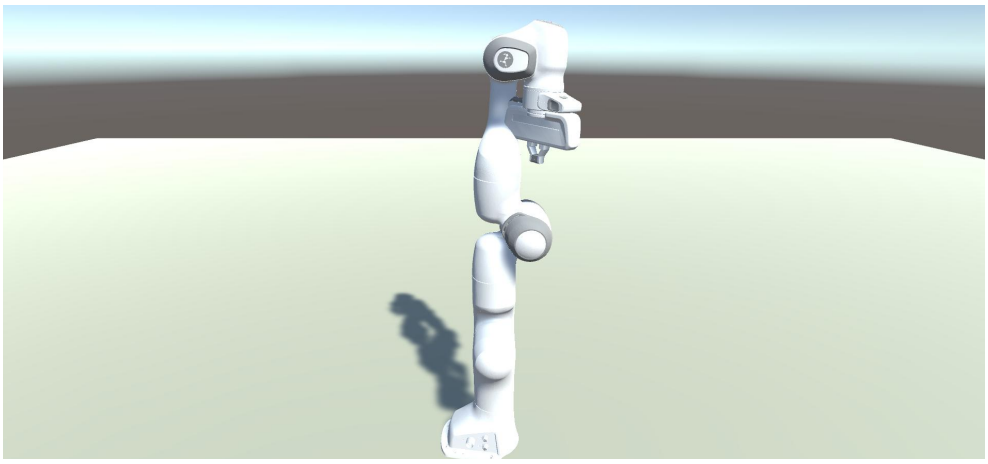


Figure 4: Panda Robot in Unity

4.2 Packages

The basic setup for a 3D Unity project allows for an extensive possibility of development options. However, there are many additional packages that can be used to extend the possibilities of Unity, both official and unofficial. Four of these packages must be added to the Unity project for the robot to be implemented into VR and allow for full control via ROS.

The first of these packages is the URDF Importer from Unity Technologies. It takes a robot URDF file and implements the robot based on the URDF and associated STL files as articulation bodies into the Unity project. The URDF file, short for Unified Robotics Description Format file, determines the geometry, visual aspects, and the joint limitations and configuration of the robot. As mentioned in subsection 3.5, Unity uses an LH, y-up coordinate system instead of the industry standard in robotics to use RH, z-up. The URDF Importer corrects for this in the articulation axis for revolute joints by applying the correct transformation to the joint [25]. This package allows for robots with the correct file structure to be easily imported into a Unity project.

In order to allow for ROS communication with Unity, the Unity side of the ROS-TCP-Endpoint as discussed in subsection 3.2 must be enabled. This is done by adding the ROS-TCP-Connector package from Unity Technologies to the Unity project. This package enables ROS communication and message generation in C# [26].

The third package is tied very closely to the ROS-TCP-Connector and is called the Visualizations package. It allows for visualizations of the received and sent ROS messages [26]. While not strictly necessary for the working of the project, it is very useful to visualize if the ROS connection is working without having to look at the ROS terminal in the WSL setup.

The final package, the Extended Reality (XR) Interaction Toolkit, is required only if VR is wanted. For this project, both 3D and VR versions of the Unity project are created. The only difference between them is the addition and integration of this package. The toolkit allows for the creation of VR and Augmented Reality (AR) projects that work cross-platform (i.e., via Open XR or Oculus) and allows for user input for User Interface (UI) interactions from the VR, or XR, hardware components [27]. This project makes use of the OpenXR, specifically for the HTC Vive setup. Thus, via this package, the packages for OpenXR and the XR Plugin Management are also used.

4.3 Robot Model

Once the packages listed in subsection 4.2 are added, and the URDF is imported, the robot must still be further setup in order to function. At this stage, the robot does not have its joints properly connected, and the joints will not rotate or move when different joint positions are given. Unity and the URDF importer as set up in such a way that all links appear in the hierarchy window as GameObjects as children of each other as specified in the URDF file. All joints are modeled as attached to its parent link. The links and joints are modeled via the Unity component reference Articulation Body.

Articulation Bodies are bodies that are part of a physics articulation that allows for accurate and precise control of joints in a system. These cannot be adjusted via transforms of the system, but must instead be adjusted solely by forces. This has the benefit of ensuring that unwanted or accidental transforms to a scene are not applied to the robot. The articulation body allows for accurate and easy control of the system [28]. The URDF Importer has placed a controller script at the root of the robot hierarchy. This allows for setting of the stiffness, force limit, as well as limits on speed, torque, and acceleration. This is required because part of the Articulation Body is the Articulation Drive. Articulation Drive applies a force to the joint, taking into account the joint upper and lower limits of movement and the joint force limit. Furthermore, the first link of the robot, panda_1_link0 in the case of the Panda robot, must be set as immovable and to turn off using gravity. This keeps the base of the robot in one place while the rest of the robot can move.

The URDF importer has already assigned what type of joint it is (i.e. prismatic or revolute), given joint limits, and set the anchors and parent anchors.

4.4 C# Scripts

There are six C# scripts added in the Unity environment. Four of the scripts correspond to the three topic subscribers and one topic publisher of the Unity node from Figure 1. The other two scripts correspond to removing Matlab/Simulink and ROS from the system and just running the simulation via Unity. These are all added to empty GameObjects in the Unity hierarchy as a child of the robot root with names of ROSSubscriber, ROSPublisher, and Manual_Test respectively. These divisions are purely for organizational purposes. In reality, they could all be added to the same GameObject, but for readability three separate GameObjects are implemented. The ArmController.cs, Clock.cs, and JointState.cs scripts were provided courtesy of Bart van den Dool [21]. The ArmController and JointState script have been heavily edited from their original state, but the Clock script has very few edits, most of which are comments.

4.4.1 ROSSubscriber

The ROSSubscriper GameObject has three script components added. ArmController.cs, Clock.cs, and EEPointVis.cs correspond to the /panda/arm_controller/command, /clock, and /panda/-pose topics from subsection 3.5 respectively. One thing to note is that the scripts are made to be mainly universal and generate the topics with the robot root name from Unity, so in the case of the another robot, these topics become /<robot_name>/arm_controller/command and /<robot_name>/pose instead. Due to ROS convention, the /clock topic does not refer to the robot name, as it is just a clock and should operate independent of the robot.

First, ArmController.cs processes the incoming desired joint positions. It has a boolean noJoint-Physics which, when True, removes the Unity physics from the joints by disabling the Articulation Bodies in the list of links. The reason for this, and its further consequences, is discussed further in subsection 4.5. For now, it is assumed that the joint physics are enabled. Within the script component in Unity, a list of Links is implemented, to which all non-fixed link GameObjects are added. The script iterates over these links and creates a list of joints, which can then be further manipulated. A ROS Subscriber to the /<robot>/arm_controller/command topic is started. For every incoming message, a list of desired joint positions is created. Then, within a fixed update that runs at the frequency of the base physics system [29], the list of joints is updated to the list of positions. This is done by checking if the ArticulationJointType is prismatic or revolute, and then assigning the position to the target of the xDrive. In the case of a revolute joint, the position is first converted from radians to degrees, as Unity xDrive has units in degrees, whereas the Simulink joint positions generated from the Inverse Kinematics (IK) are in radians.

Clock.cs is used to remove the autoSimulation if desired from the system. The physics engine is by default set to update every 0.02 seconds. The /clock topic allows for syncing of the Matlab/Simulink clock to the Unity physics engine. The boolean autoSimulate, when set to false, allows for the physics engine to simulate based on the Simulink time step instead of on the Unity time step [29]. This is beneficial if the Simulink cannot run in real time, as then the physics will still behave as if it were real time. There is a failsafe built-in for time steps that are too large, as otherwise the physics simulation will become distorted. In such a case, the robot will appear to pause, before resuming to achieve the next desired joint configuration.

The final script on this GameObject is EEPointVis.cs which allows for visualization of the desired end point of the robot. As stated before in subsection 3.5, /<robot>/pose is not a strictly necessary topic, and thus this script is not a strictly necessary script either. However, for visual confirmation of accuracy of the movement of the robot, it is a useful tool. The script has a spawnStartPrefab boolean which also instantiates a block at the start position of the robot when true. The script

subscribes to the pose topic and assigns the pose message to position and rotation variables. Due to the translation from RH z-up to LH y-up coordinate systems, this must also be taken into account. This is the only script that is not usable with every new robot without change, as discussed in section 6. When the y value for the position is no longer the default value of 0, the EE prefab is instantiated at the desired position and rotation.

4.4.2 ROSPublisher

The ROSPublisher Gameobject holds all script objects that have a publisher in them. In this case, only JointState.cs is a publisher to the /<robot>/joint_states topic. This script publishes the achieved joint states of the robot. It also has capabilities of publishing the joint states if the joint physics is turned off. When joint physics is enabled, the ROS message type of JointStateMsg must be filled in. The same list of links from ArmController.cs is used in order to loop over the links and gather their names, positions, and velocities. The effort is currently set to 0.0, but may be able to be gathered in the via jointForce. However, since force is not a focus of this report, this has been omitted. The positions and velocities are gathered from the link's Articulation Body. In case of a revolute joint, the position and velocity are already defined as radians and radians per second, which is unintuitive as defining the position or velocity must be done in degrees, as mentioned in subsubsection 4.4.1.

In the case that joint physics is turned off, the Articulation Bodies have been disabled. Therefore, the joint positions and velocities must be obtained in another manner. Unfortunately, this is where the universality breaks again, as each robot is different, and has a different series of joints, that rotate or translate in different ways. The URDF importer working with the Articulation bodies and its corresponding XDrive compensated for this. However, now the script must compensate for this manually. Based on which direction the joint rotates, the Inspector rotation is obtained for the revolute joints. The inspector position is used for prismatic joints. However, some joints, like the joint associated with link 6 from the Panda robot still are given incorrect in this manner as the inspector rotation is calculated based on -180 to 180 instead of 0 to 360 degrees. Therefore, an extra step to compensate for this must be added. Ensuring that the joint positions are accurately published to the topic is a requirement in order to test for accuracy. Velocity may be able to be obtained similarly by taking the link's transform and requesting the velocity. However, since this report does not concern itself with velocity control, only position control, this is not done for now. Instead, 0.0 is sent for the velocity as a placeholder.

4.4.3 Manual_Test

Removing ROS and Matlab/Simulink from the system is done by shutting off the ROS connection and instead using a csv file that has the desired joint positions for each time step. This csv file is created beforehand by running the Simulink and recording the data into a csv file through Matlab. Removing ROS and the Simulink is preferable for a variety of situations, but not always the ideal option. ROS does introduce a slight delay in the system, as will be later discussed in section 7, so removing ROS can remove this delay. Also, for ease of use, continuously switching between Unity and Simulink to restart the system for user testing introduces an extra step. The option to only run Unity is thus desirable if the same robot path needs to be run consecutively numerous times, or if Simulink cannot simulate in real time due to computational power constraints and if the /clock topic cannot compensate for this. For continuously or frequently changing desired robot paths, removing ROS and Simulink is not the ideal solution, as then the extra step of creating a csv file for each new robot path is introduced.

The two scripts relating to this are called ConnectionCheck.cs and ManualJointState.cs. They are both in the Unity hierarchy as components of the Manual_Test GameObject. ConnectionCheck.cs has a boolean called setManual that controls whether ROS is used for the whole system. When

it is true, the ROS is disabled. There is also a boolean manualNoJointPhysics which, when true, removes joint physics from the system. A separate assets folder called csv_files stores the desired joint positions in a csv file called data. A StreamReader is used to read the csv file and store the data in a list of lists. Every list within the main list holds the joint position for each joint and the time point this joint should be input to the robot. Then the simulation time is tracked, and the joints are updated when the simulation time reaches the desired time point. As Fixed Update is used, the updates are every 0.02 seconds by default. The desired joint positions are sent to the joints in the same way as the ArmController.cs described in subsubsection 4.4.1.

ManualJointState.cs records the joint states reached in a separate csv file called logged_test_data. Again, a list of lists is used to save the achieved joint position and the time stamp it is recorded for every inner list. This is running in an Update loop, which creates a new list for every frame from Unity. The joint positions are acquired in the same was as the JointState.cs from subsubsection 4.4.2. Then, once the desired joint states from ConnectionCheck.cs are all executed, the logged data is written to the csv file via a StreamWriter.

4.5 Unity Physics

The Articulation Drive force described in subsection 4.3 is controlled by Equation 1. Here the stiffness, positions, damping, and velocity all play a role in the physics of the joint. As velocity is not taken into account or used with the Simulink scripts, damping can be set to 0 to fully ignore velocity targets. Stiffness should be set high enough so accurate tracking can be enabled based on the current joint position and the target joint position.

$$\text{F} = \text{stiffness} * (\text{currentPosition} - \text{targetPosition}) - \text{damping} * (\text{currentVelocity} - \text{targetVelocity}) \quad [29] \quad (1)$$

Theoretically, this means that as stiffness approaches infinity (assuming the force limit is also approaching infinity), the joint's target position should become the joint's current position at the next time step. However, this is not the case, and can be attributed to Unity's physics engine. The exact deviations are discussed and analyzed in section 7, but an up to 5% error can still be observed in some joints. This could be fairly significant, and a solution must be found. Therefore, the option to omit joint physics was introduced.

This is done by disabling the articulation bodies and transforming the joint rotations instead. Then the local Euler angles of the revolute links and the local position of the prismatic links must be transformed. This is done based on the geometry of each robot joint, and so is unique for each robot configuration. Similarly to reporting the joint states as discussed in subsubsection 4.4.2, this must be configured manually in order to compensate for the RH to LH coordinate systems.

Manual configuration of the robot without joint physics can best be done by observing the joint articulation body movement, as the xDrive frequently, but not always, negates the target in the position or Euler angle movements. Copying these patterns into the manual system gives a fairly accurate movement. In the case of the Panda robot, the 6th link deviated from this pattern and had to be compensated for. This is similar to what was seen in the joint state publishing from subsubsection 4.4.2. It is possible that if the robot adheres to a Denavit–Hartenberg (DH) convention that this DH table could be used to configure the robot, but this was not looked into during this project.

4.6 Virtual Reality

For this project the HTC Vive Pro 2 headset with 2 base stations, 2 controllers, and required connection equipment were used to test the VR Unity projects. The Vive Pro headset requires the

use of SteamVR to operate, and can thus be used in conjunction with Steam and Vive published games, platforms, and projects like training via simulations. SteamVR also connects to Unity when Play Mode is entered. Play Mode allows for use, manipulation, and running of the Unity scene within the Unity editor. This results in being able to experience the robot in VR, and for easy bug testing and fixing. The Panda robot can be seen in VR in Figure 5 where the robot is halfway though the movement to reach the desired end position.

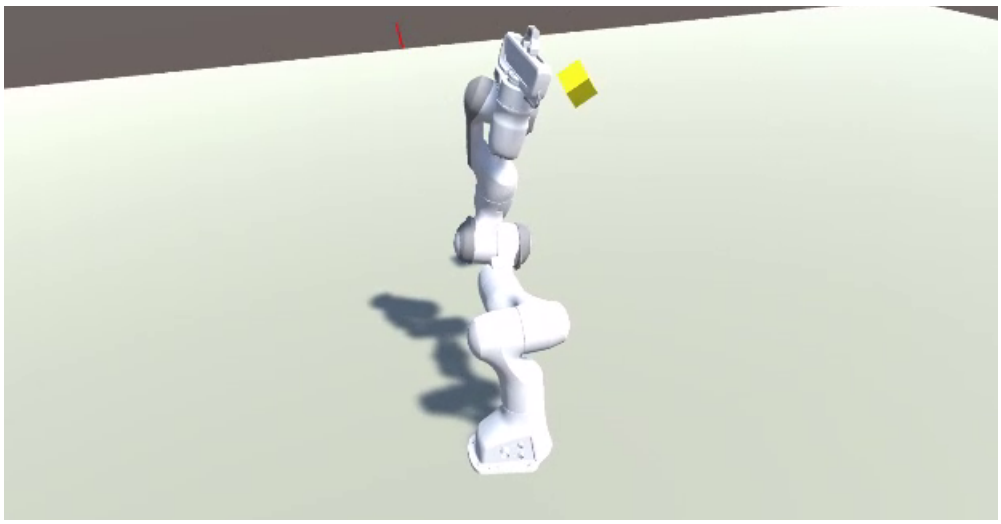


Figure 5: Panda Robot in VR halfway though movement

During this project, two separate HTC Vive Pro 2 full kits were set up to work with a laptop, one at Philips IGT, and one for Control Systems Technology at TU/e. Also, a portable VR ready laptop was set up to be able to run the entire Matlab, Simulink, WSL, Unity, VR headset system. Setting up the Vive Pro requires using the required drivers, software, etc. provided in the instruction manual. One troubleshooting point when setting up the whole system is to ensure that the proper firewall settings are enabled. While the whole system is on one laptop, the VR headset and WSL are considered as outside connections, and so public network connections should be allowed.

4.7 Philips IGT robot in Unity

An identical solution is implemented for the Philips IGT robot, with the main exception being that a different URDF file for this specific robot is used. Also, the coordinate axis translations from the Philips robot into the Unity LH y-up coordinate axis are different, so the EE visualization Unity script is adjusted as well. The no joint physics parts of the arm controller and joint state script have been adjusted for this specific robot. A VR version is made as well.

Due to the confidentiality of the Philips IGT robot, no further data about the Unity implementation can be given in this report. The repository for this robot is separate from the Panda repository [2] and has been supplied to Philips.

5 Simulink

Matlab and Simulink are used to calculate the desired robot joint states and send it to Unity via ROS. Matlab/Simulink are chosen for this due to the prior existence of various joint state scripts. Due to the universality of the ROS and Unity setup, any prior existing controller or joint state calculator can be easily transformed to be compatible with the ROS and Unity setup. For the chosen panda robot, various joint state setups are discussed and built up to gain a full understanding of the possibilities. First, a simple joint setup is explained, and then a more in depth look is taken at achieving a desired EE point. All simulations done in this section are with joint physics enabled.

5.1 Simple Joint Movement

Initially, the idea is to simply move the joints in the Panda robot. This can be achieved by sending desired joint locations to the /panda/arm_controller/command topic via Simulink. Figure 6 shows the associated Simulink file, called a_SingleInput.slx in the Panda repository [2]. Simulink ad Blank ROS message formats, which takes the trajectory_msgs/JointTrajectory.msg and trajectory_msgs/JointTrajectoryPoint.msg message types and uses them to populate the joint message with the joint positions given. As the Panda arm has 7 joints, all of which are revolute, there are 7 final joint positions supplied. These values are currently arbitrary, and can be set to any value within the joint limits. These joint limits can be found in the URDF file and in the Unity Editor under each link's Articulation Body.

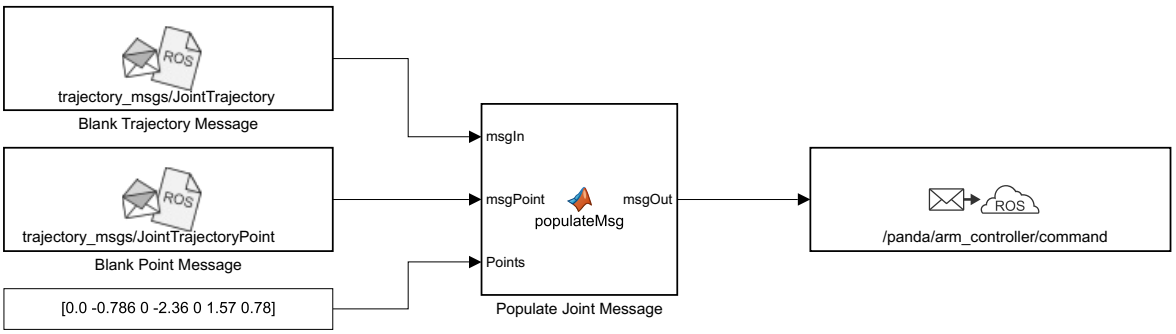
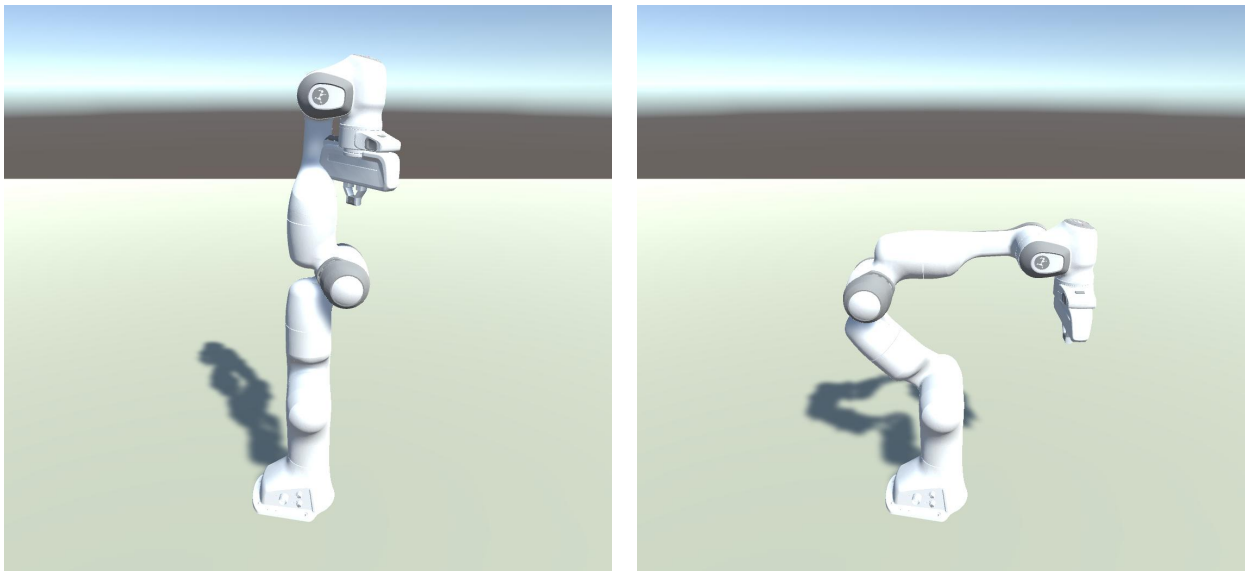


Figure 6: Single Input Simulink

The Populate Joint Message Matlab function code van be found in Code A.3, where the joint names are manually assigned and converted into integer numbers, as the ROS messages that Simulink publishes cannot handle strings. Then the desired joints are assigned into the Points.Positions part of the message [30].

Finally, the joint message is published to the /panda/arm_controller/command. When the Simulink is run, Unity shows the results in Figure 7a and Figure 7b where the robot is in its initial position and the final position respectively. For now, confirmation that the desired joints are reached is done only visually and within the Unity Editor. This is addressed more in depth in the next Simulink file in order to allow for confirmation of reaching the desired joint state of the system.

Just looking at the robot movement is not very accurate, especially as the robot will move over time, and not just instantaneously. Therefore, it is desired to log the data sent via the /panda/-joint_states to Matlab in order to analyze it. Also, it is required to send joint data over time, and reach desired joint configurations non-instantaneously, all of which are not met with the previous Simulink file. Figure 8 is the b_OneJointOverTime.slx file which does implement these requirements [2]. The blank joint messages, population message, and arm controller publisher all remain.



(a) Initial Position (b) Final Position

Figure 7: Moving to one joint configuration

The position points are no longer a constant block, but now are determined by the Polynomial Trajectory block. This block uses the simulation time, an array of waypoints, and a list of time points as input. Six time points and six joint angle waypoints are given, and the trajectory block computes smaller points to avoid instantaneous movements with extreme forces. The Simulation time is also populated into the /clock topic. Finally, in order to confirm accurate movement of the robot, a subscriber to /panda/joint_states combines the achieved joint states and the desired joint configurations and puts it into a scope that also saves to the Matlab Workspace.

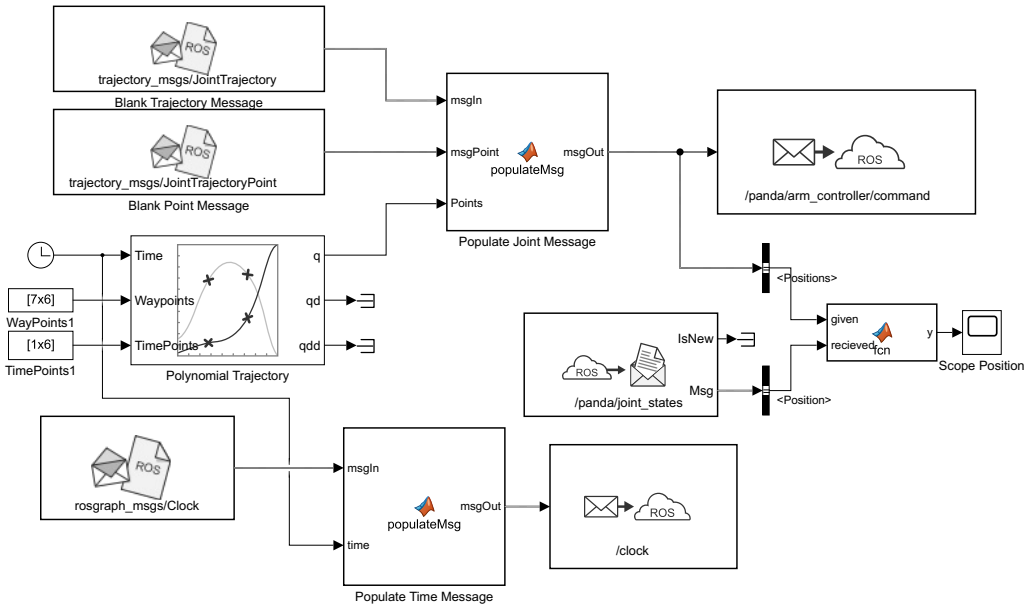


Figure 8: Moving One Joint Over Time Simulink

For this example, only joint one was rotated, with a new waypoint being desired to be achieved once every two seconds, for the following angles: 0.25, 1, 2.75, 1.5, -2, -2.45 radians. The other six joints are kept at 0 radians. This can be seen in Figure 9 where the dashed lines are the desired joint angles, and the solid lines are the achieved joint states. Each separate line color corresponds

to the same joint. As can be seen, the singular joint tracks the desired line accurately, even if there is a miniscule amount of time delay between the two lines, as can be most clearly seen around 7 seconds. The other joints all stay at the desired 0 radians.

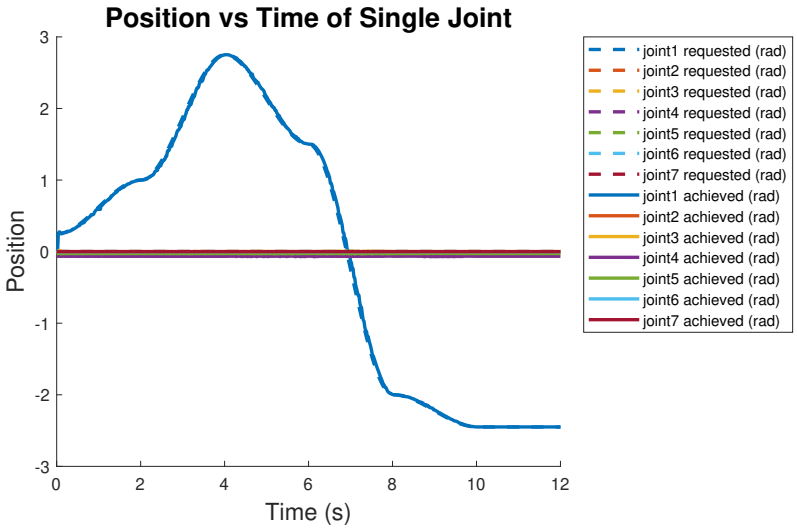


Figure 9: Joint 1 moving over time

5.2 Inverse Kinematics

Now that it is clear how to reach a desired joint configuration, the next goal is to reach a desired point in space with the EE. This point can be given in Cartesian RH z-up coordinates and then achieved through various means. In order to translate from a desired EE point to the corresponding joint configuration, the robot's IK must be used. Knowing it is possible to translate from Cartesian to joint space, two approaches can be taken. The first option is that the starting point and the end point are translated to joint space, and then the two corresponding joint angles are put through the polynomial trajectory block. The second option is that the starting and end points in Cartesian coordinates can be fed through the polynomial trajectory, and then to take the IK of each output point. The first option will result in a much more wildly moving robot, but be computationally much lighter. The second option will result in the EE traveling in a straighter line, but will be very computationally heavy, as many more IK computations must be made.

5.2.1 IK for start and end position

Initially, option one will be attempted in order to avoid heavy computations. This corresponds to the a_IK_model_joints.slx file in the Panda repository [2]. For this file, the init_robot.m is required. The desired final EE point must be input into the Matlab script and ran. The original joints from Unity are all set at 0, and the IK will be computed for the final EE point. For this report all desired final end positions are at 0.5 m, 0.375 m, 0.75 m for X, Y, and Z respectively. For now, the final orientation of the EE is not relevant, and thus not taken into account in the IK calculations.

Only once the associated Matlab file has been run can the Simulink file shown in Figure 10 compile. Here there are many similarities with the previous Simulink file, but with some key differences. Manually adjustable values are marked in green. For now, the polynomial trajectory will achieve the final joint space from zero to five seconds. Also, the desired final EE point is sent via the /panda/pose topic to Unity.

This results in all joints moving in a polynomial fashion to the desired end point. This can be seen in Figure 11a where all the joint positions achieved closely follow that of the desired joint

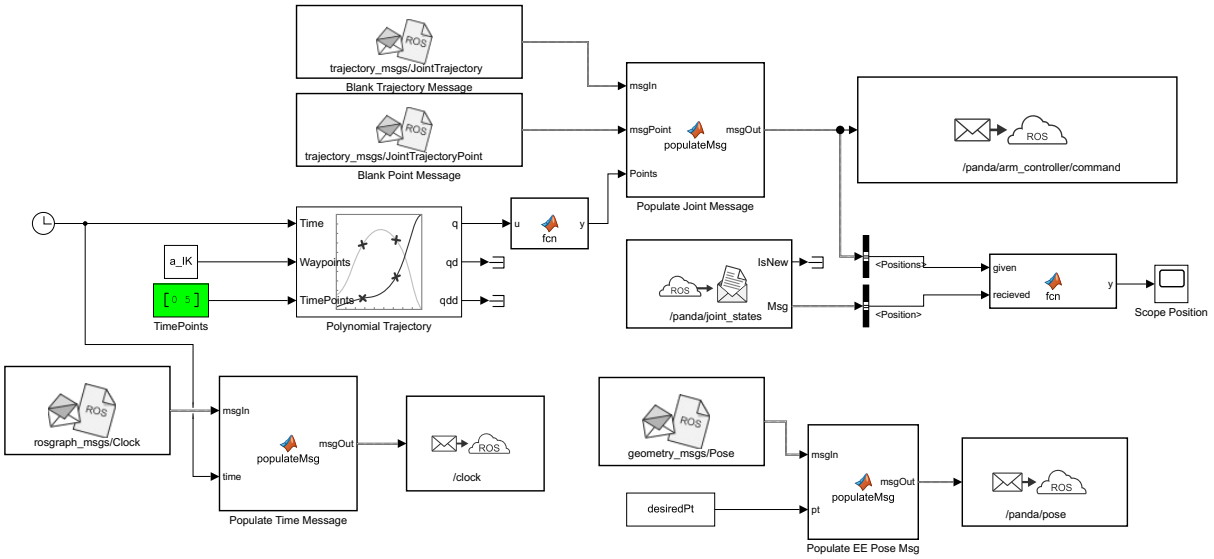
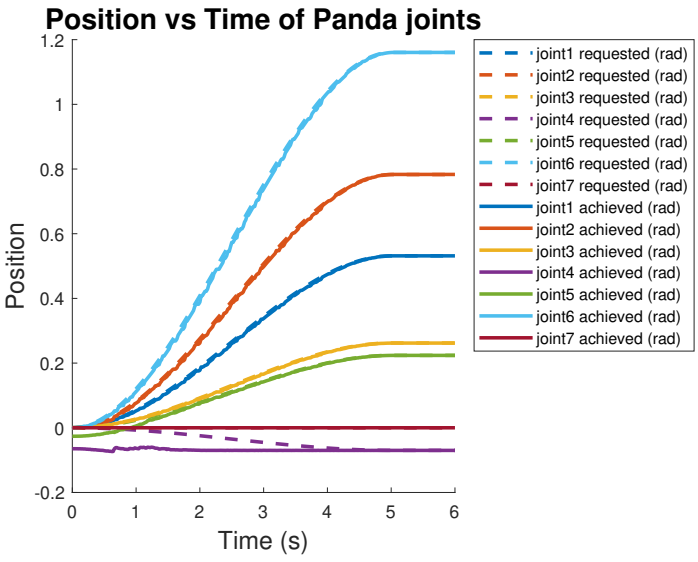
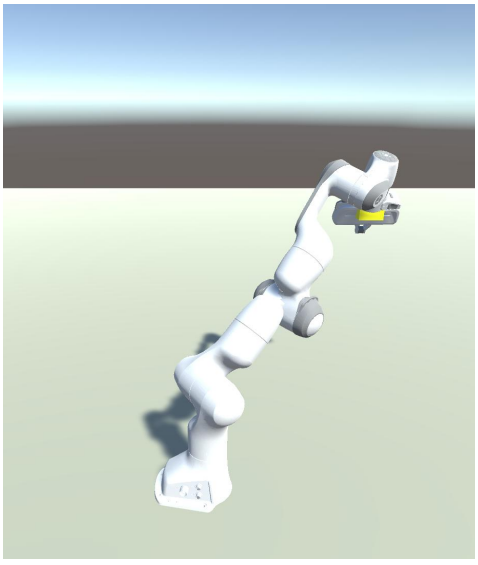


Figure 10: Robot over joint angles Simulink

configurations. Only joint 4 exhibits errors at the start of the simulation, likely due to joint physics issues. However, since the final desired joint position is reached, this is not considered an issue. Furthermore, it is a rarely seen issue in the Panda robot that there is such a clear deviation. The pose topic has been utilized and the EE visualization prefab has been spawned in Unity, as seen by the yellow block in Figure 11b. The final position of the robot is reached.



(a) Joint positions over time



(b) Final position

Figure 11: Polynomial trajectory on joint angles

However, the main downside of this application is that while the joints move in a controlled manner, the EE is moving sporadically while the joints are slowly moved.

5.2.2 IK over all points for position

The second option is implemented now in order to enforce a straighter EE path. This Simulink can be seen in Figure 12, and is called b_IK_model_cartesian.slx in the Panda repository [2]. Only the first section of init_robot.m needs to be run in order to give the IK block a reference to a rigid

body tree imported from the URDF file. In this Simulink file the desired final EE point is defined, which gives more ”changeable” green constant blocks. The initial EE position is in the red block. These two point sets are fed into the polynomial trajectory. This Cartesian translation vector is translated into the homogenous transformation matrix with the Matlab function `trvec2tform`. This is then fed to the IK block which takes inputs of the 4 by 4 homogenous transform, the pose tolerances weights, and the initial guess for the solution. The weights are set to a six element vector of three zeros and three ones. This is to enforce the EE position but leave the EE orientation as not important. The initial guess for the solution is set to the previous time step's solution, as it is assumes that the next solution is likely close to the new solution. This cuts down on computational time by suggesting that the initial guess is close to the final solution. In order to further reduce computational time, the solver parameters have been set to code generation using the Levenberg-Marquardt algorithm [31, 32]. This IK solution sent to the /panda/arm_controller/command topic.

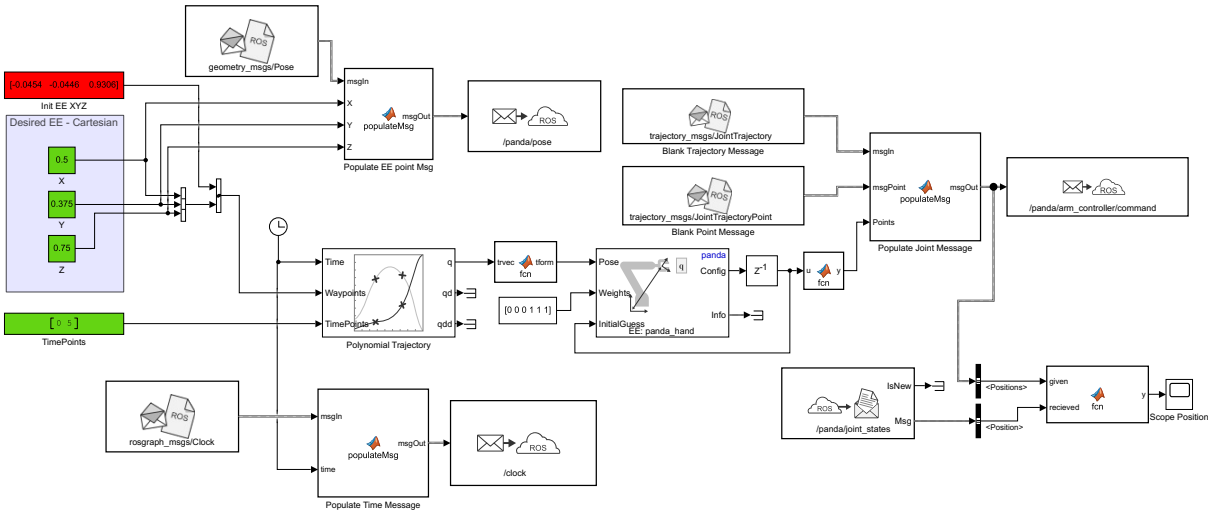
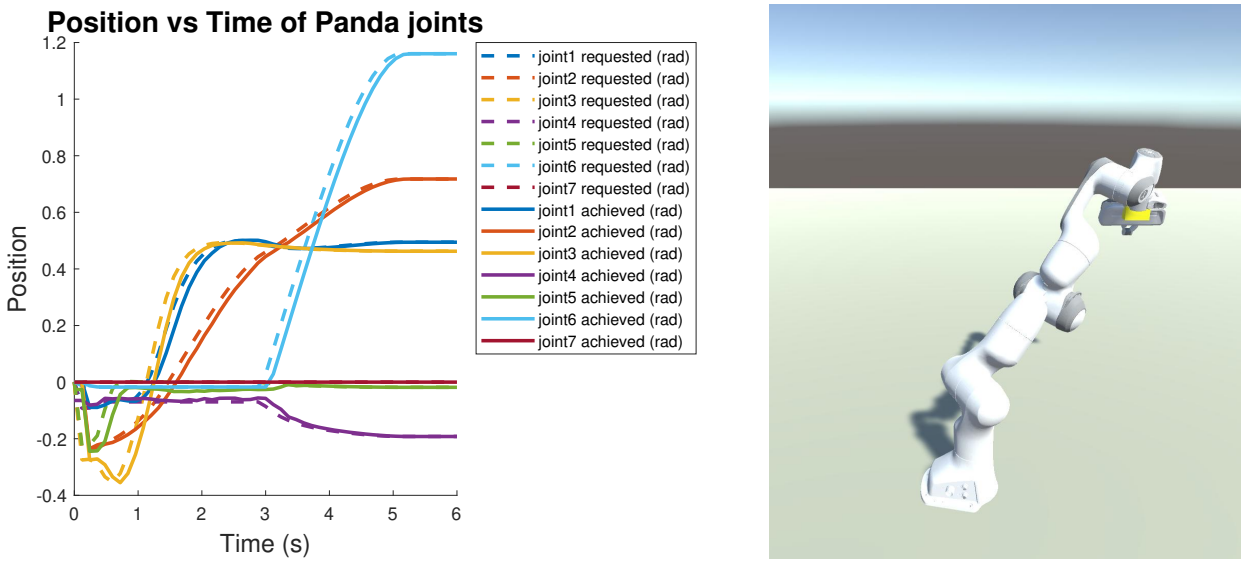


Figure 12: Robot over Cartesian position Simulink

With this application, the joint positions in Figure 13a were reached. Here there is a slight delay in moving the joints, which is especially clear in joints that move at a higher velocity, but the final desired position is reached. The final EE point is the same when compared to the first application, but due to the use of the previous IK solution, the final joint configuration is different, especially in joint three. Figure 13b shows this robot view.

Sometimes solving for many joint configurations via the IK block is still too slow to run in real time, and then Simulink slows down the execution time. In such a case, it is beneficial to remove the live generation of the points and also remove the ROS time. This allows the robot to be simulated in real time instead of waiting for calculations to be completed in Simulink and sent via ROS. To do this, the setManual boolean in Unity can be set to true as discussed in subsubsection 4.4.3 and the data can be read from a csv file. Generating this csv file is can be done by running the desired Simulink file and then running manual_csvGeneration.m. This will create the desired CSV file in the 3D Unity version. The file path can be easily updated to the VR version. Another option is to turn off autoSimulation and allow the /clock topic to Enforce the Simulink time into the Unity physics engine. Then the Unity project can be put into play mode and the desired joint positions will be achieved and then recorded as well.



(a) Joint positions over time (b) Final position

Figure 13: Polynomial trajectory on Cartesian points

5.2.3 IK with EE orientation

Not only is the end goal to reach a specified final EE position, but also to reach a specified EE orientation. By setting the weights array to all ones, the IK will weigh the desired EE position and orientation equally. Also, more user inputs for desired orientation are required. This can be seen in Figure 14 and is from c_IK_and_EE_rotation.slx in the Panda repository [2].

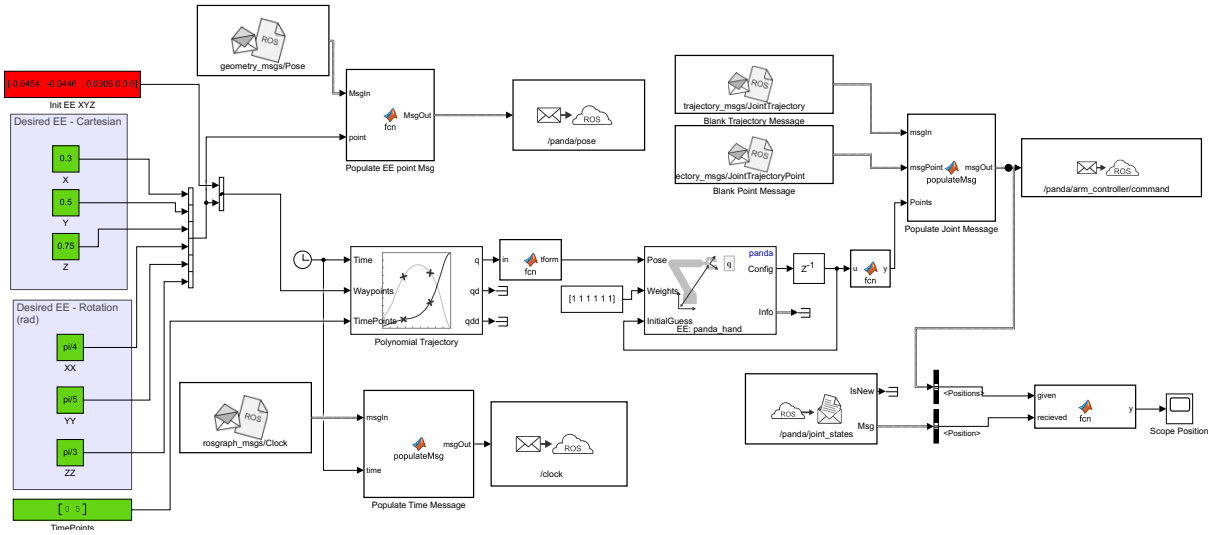
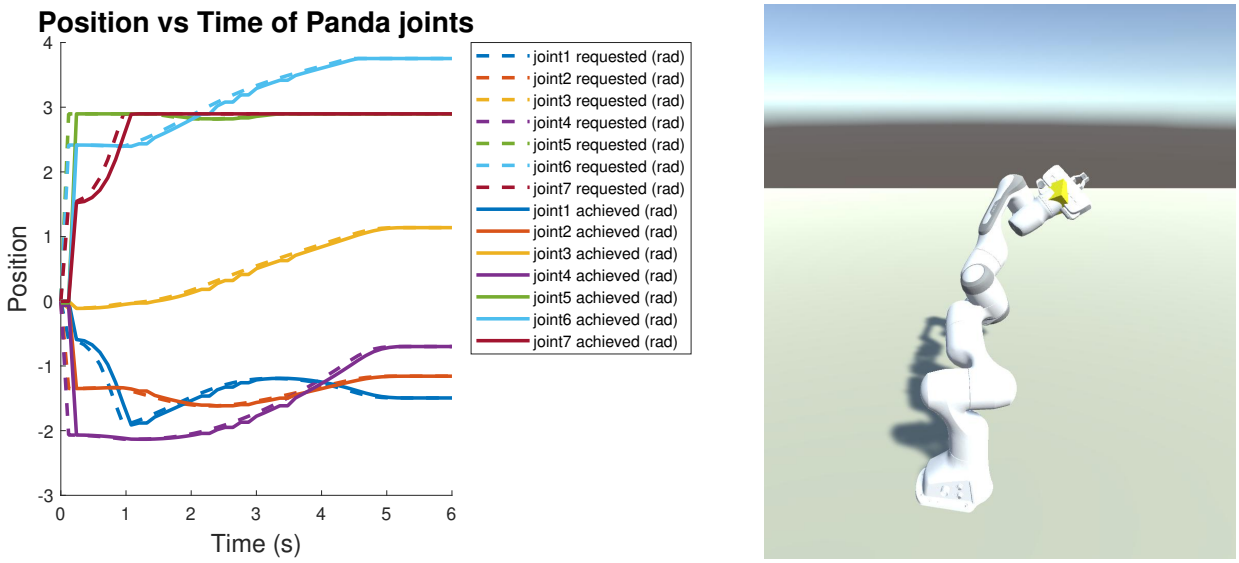


Figure 14: Robot over Cartesian position with EE rotation Simulink

Again, the joint positions over time and the final position can be seen in Figure 15. Specifically, in Figure 15a, the achieved joint positions can be seen to be folloing closely along with the desired joint positions, but in Figure 15b, the yellow desired point block is rotated differently than the panda EE. This suggests that the IK solution was unable to find a solution for the desired EE position and orientation of 0.3 m, 0.5 m, 0.75 m at pi/4 rad, pi/5 rad, pi/3 rad. However, it must be confirmed if this is a Unity visualization issue or an IK solution issue. Furthermore, it should also be confirmed that the desired EE path is followed and reached, and that it is not just a visualization coincidence.



(a) Joint positions over time (b) Final position

Figure 15: Enabling desired End Effector rotation

5.2.4 FK confirmation

Forward Kinematics (FK) are applied to the joint state achieved in Unity in order to translate back from a joint configuration into Cartesian position data and Euler angles. The GetTransform Simulink block was attempted to be used with the provided URDF file and the joint state information, but this was prone to errors including negation of positions, rotating coordinate axis halfway through a simulation, and various other unexplainable errors rendering this option unusable. This issue is further explained in Appendix B. Therefore, a FK script was made that is dependent on the Franka Emika Panda robot's DH parameters [33]. This FK script can be found in Code A.4. This is applied in d_FK_confirm.slx in the Panda repository [2] as seen in Figure 16.

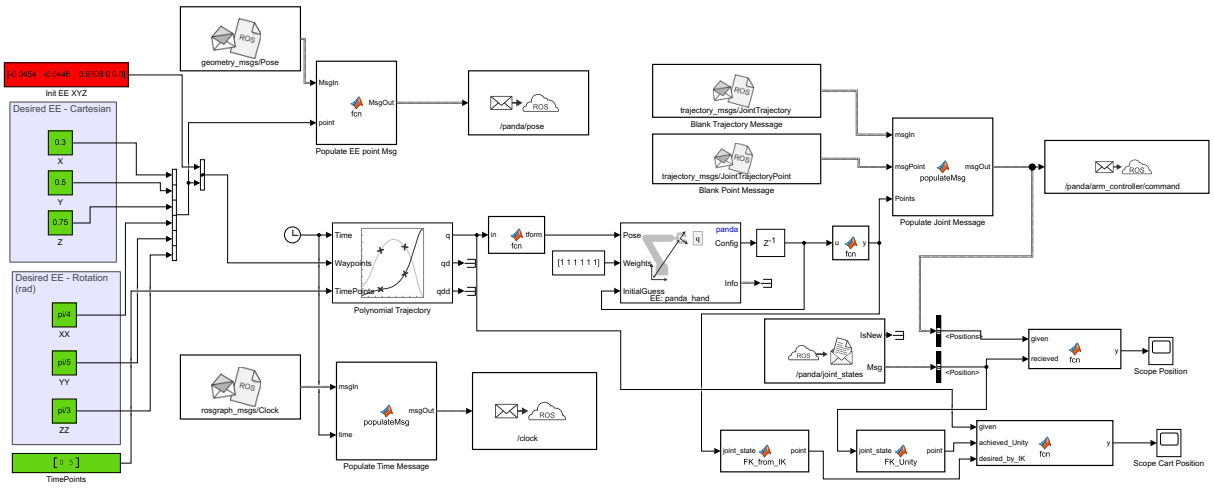
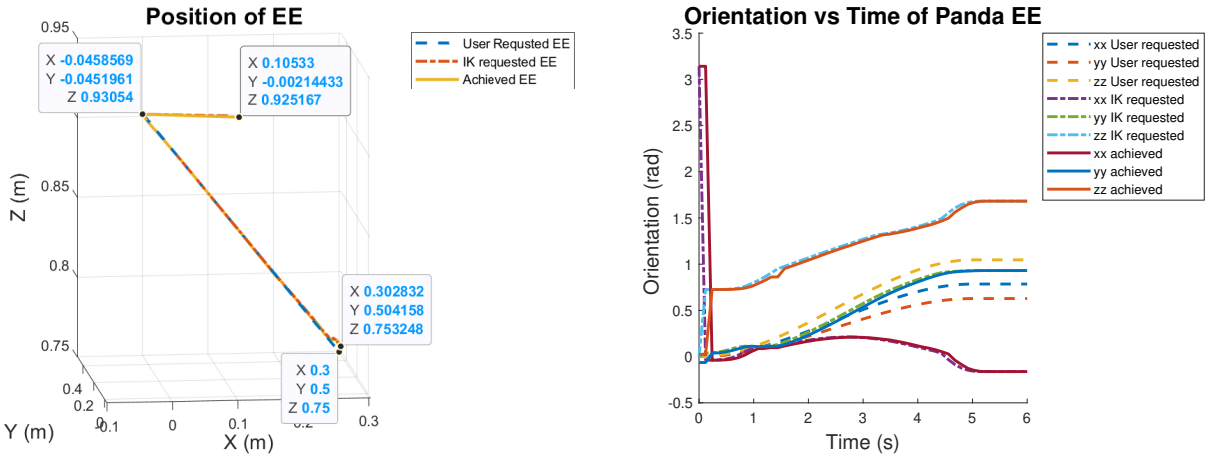


Figure 16: Robot over Cartesian position with EE rotation and FK confirmation in Simulink

Running the identical desired end point and orientation from the previous simulation results in Figure 15 again. With the FK data now the EE path can be visualized and the orientation over time as well. This can be seen in Figure 17. In the plot of the EE from Figure 17a the path the EE travels in the Simulation closely follows that of the goal path. At the start, with the higher Z point, the simulation starts with the Unity robot being in its original URDF configuration from

Figure 4. As this position is technically not reachable by the robot, the IK will never solve for it. This is the reason the robot immediately updates to the desired start position. Then the robot travels closely along the line that is desired, before deviating slightly at the end when trying to reach the 0.3 m, 0.5 m, 0.75 goal point. The reason for this deviation is due to the IK solution deviating from the user requested path. As this is an IK problem, and the Panda robot does follow the IK solution, this is considered acceptable. A similar issue follows in Figure 17b, where the User requested orientation is very different from the IK solution orientation. Again, the Unity robot follows the IK solution quite accurately.



(a) EE path in Cartesian coordinates

(b) EE orientation over time

Figure 17: Confirmation of reached position with forward kinematics

Why the IK solution varies drastically from the user requested solution is unknown. It is possible that the requested position and orientation are simply beyond the robot's workspace. However, when testing this via trial and error, more accurate IK results were not achieved. This is left as out of scope for the given projects.

5.3 Philips IGT robot with Simulink

In order to confirm that the Philips IGT robot implementation in Unity works, similar Simulink files were made in order to reach a desired point and orientation with the EE. The Philips IGT robot has three joints. One of these is a prismatic joint, which was previously not seen in the Panda robot. The other two joints are revolute joints. Due to the configuration of these joints, only the X, Y, and zz positions and orientations can be adjusted. The results of these tests can be seen in Appendix C, where the desired end point and rotation is reached.

5.4 Existing Simulink Implementation

The Unity scene can of course also be used to simulate any joint path that is generated by a Simulink, including controllers. In case of a setup that purely tracks the position of the joints and only influences the position of the robot, even the no joint physics setup in Unity can be used. In case of a velocity controlled robot, velocity can be sent and received from Unity only when joint physics is used via the Articulation Bodies. All that is required in order to enable this is to ensure that the damping constant from Equation 1 is no longer set to zero, and the Articulation Body will attempt to both adhere to the target velocity and the target position. If pure velocity control is desired, stiffness can be set to zero. These options were not rigorously tested, but can be fully enabled in the future.

All that is required to make any Simulink joint position or velocity file connected to Unity is to add the message population scripts and the publishers for the /<robot>/arm_controller/command

and /clock topics to the file. If feedback control is used, the /<robot>/joint_state topic can be subscribed to in order to enable the feedback. This will allow for testing of the robot in order to ensure it does not violate any joint constraints, or other constraints to the motors like acceleration or force. The Unity based robot may not physically react completely identical to the same robot in reality due to different joint friction, damping, ect. However, it is a good first assessment of the controller or joint path.

6 Universality

In order to ensure that future projects can be easily implemented into Unity, the C# scripts have been made to be as universal as possible. This allows for implementation of any robot in VR without needing to change the C# scripts in Unity. Only the URDF file would need to be imported and the VR implementation added. Unfortunately, removing the joint physics is not robot independent. Therefore, the script files from the Unity projects were edited to remove all references to the option that allows for no joint physics. Also, as end point visualization is also not universal and also not required for a working robot, this script has been omitted as well.

Setting the joint to the desired position or rotation is thus achieved by using the Articulation Drive and updating the xDrive's target and target velocity. This can be seen in Code 1. In the case that the joint is revolute, the desired position is converted into degrees. This iterates over the entire list of joints in order to update all joints at every FixedUpdate.

Code 1: Set Joint

```
for (int i = 0; i < joints.Count; i++) {
    ArticulationDrive drive = joints[i].xDrive;
    if (joints[i].jointType == ArticulationJointType.PrismaticJoint) {
       // prismatic
        drive.target = (float) positions[i];
        drive.targetVelocity = (float) velocities[i];
        joints[i].xDrive = drive;
    } else { // revolute
        drive.target = (float) positions[i] * Mathf.Rad2Deg;
        drive.targetVelocity = (float) velocities[i] * Mathf.Rad2Deg;
        joints[i].xDrive = drive;
    } }
```

Similarly, the achieved joint state must be exported. As seen in Code 2 the list of links is iterated over and the joint for each link is acquired. Then the joint position and velocity are documented, along with the name of the link. Note that the joint position and the joint velocity are already radians and radians per second for a revolute joint. Therefore, no distinction on the joint type is required.

Code 2: Send Joint State

```
for (int i = 0; i < links.Length; i++) {
    ArticulationBody joint = links[i].GetComponent<ArticulationBody>();
    name[i] = links[i].name;
    position[i] = joint.jointPosition[0];
    velocity[i] = joint.jointVelocity[0];
    effort[i] = 0.0; }
```

As these code snippets are universal for any robot of any amount of prismatic and/or revolute joints, no changes need to be made for any script. This is ideal for future development as it allows a fairly accurate use of a virtual robot with relative ease. Should higher degrees of accuracy be desired, then the no joint physics code will need to be adjusted to retrieve the link's transform in position and rotation coordinates instead of simply retrieving the xDrive values.

7 Results

The Unity projects for both the 3D and VR visualizations of the Franka Emika Panda robot and the Philips robot have been completed. Simulink and Matlab scripts in order to reach a desired end point and rotation have been implemented as well. These can be found in the public Panda Repository from the Robot Unity VR Gitlab [2]. For ease of future robotics in VR implementation, a documentation wiki is also included [3]. In section 5, it was shown that the Panda robot followed the joint configurations that were specified in the /panda/arm_controller/command topic closely. However, as mentioned in section 4, there are many settings that can be adjusted in Unity in order to give a more accurate visualization of the desired joint coordinates. These are turning the joint physics off, enforcing synchronization of the Unity clock to the Simulink clock, and removing the ROS connection as a whole and using a csv data file instead. These options are not all possible at the same time, for example the ROS connection is required in order to sync the clock times. Thus, eight different use cases were devised to retrieve data on the accuracy of the Unity robot. These can be seen in the following list.

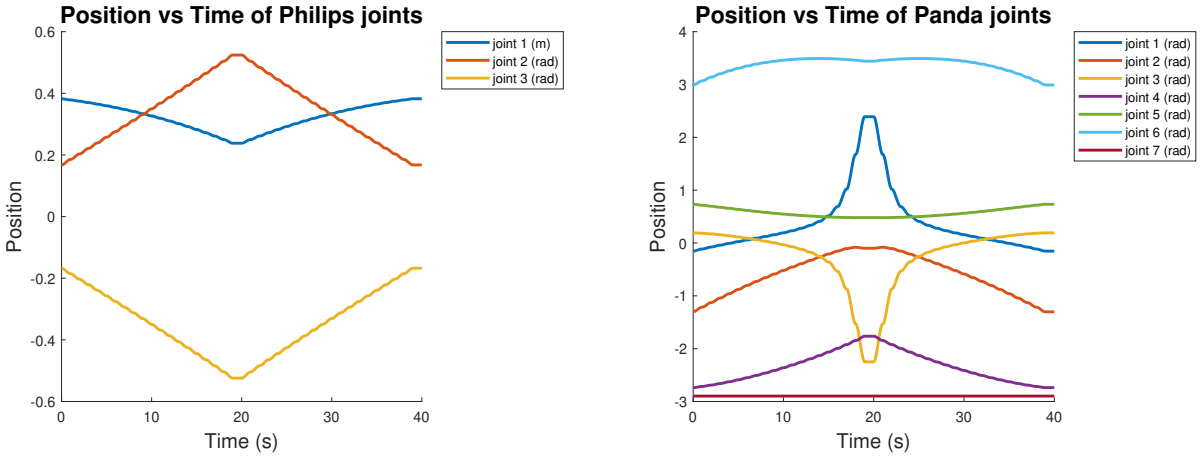
1. **Base case:** The standard operating procedure for the Unity robot. Has a ROS connection via WSL, uses Articulation Body physics, and uses the physics autoSimulation.
2. **No ROS:** Removes the ROS connection from the base case and reads data from a csv file. Done by having setManual = true.
3. **ROS via Linux:** ROS runs on a separate Linux computer, and Unity runs on a dedicated VR computer. The Simulink runs on a third Windows computer.
4. **No joint physics:** Turns off the Articulation Body xDrives and enforces joint movements via transforms. Done by noJointPhysics = true.
5. **Infinite stiffness:** Sets the stiffness value from Equation 1 and the Force Limit values to infinity.
6. **No ROS, No physics:** Combines item 2 and item 4. Done by having setManual = true and manualNoJointPhysics = true.
7. **AutoSimulate off:** Enforces the Simulink clock via the /clock topic. Done by having autoSimulate = false.
8. **AutoSimulate Off, No Physics:** Combines item 4 and item 7. Done by having autoSimulate = false and noJointPhysics = true.

The base case was used in section 5, but the other options must be confirmed as well for both the Panda and Philips robots. Therefore, the joints were measured over 40 seconds in a symmetrical movement over time, as can be seen in Figure 18. The Panda test was supplied by v. d. Dool, and the Philips test is a variation of this [21]. For each use case, five trial runs were used in order to gather data for the joint errors.

In order to calculate the error percentages, the Mean Absolute Percent Errors (MAPE) is the average of absolute percentage errors. It is calculated via the use of Equation 2. Here, R_i is the requested joint value and A_i is the achieved joint value at data point i. N is the number of data points i received [34]. In order to calculate the MAPE for use cases without ROS that use csv files, the csv data points must be interpolated via `interp1` in order to interpolate that data set of A to the equal length.

$$\frac{1}{N}\sum_{i=1}^{N}\left|\frac{R_i - A_i}{R_i}\right| \tag{2}$$

For the Philips robot, all eight use cases were tested in order to confirm that they work. Each use case was tested five times. This data can be seen in Figure 20, where the average of five MAPE was taken, and the standard deviation is also displayed. Note that for all use cases involving the no



(a) Philips Robot

(b) Panda Robot

Figure 18: Trial testing paths

joint physics option, the MAPE is significantly lower for joints 2 and 3, the revolute joints. Joint 1 also has a lower MAPE with no joint physics, except where the ROS connection was removed as well. Also, where the ROS connection is removed, the standard deviation is also much smaller for all joints. As discussed in subsection 4.5, the infinite stiffness does not improve the error of the robot either. For the Philips robot, the desired use case seems to be 4, no joint physics, in order to keep the errors low. Use case 8, autosimulate off, no joint physics could also be used in order to further reduce the standard deviation of the robot's movement.

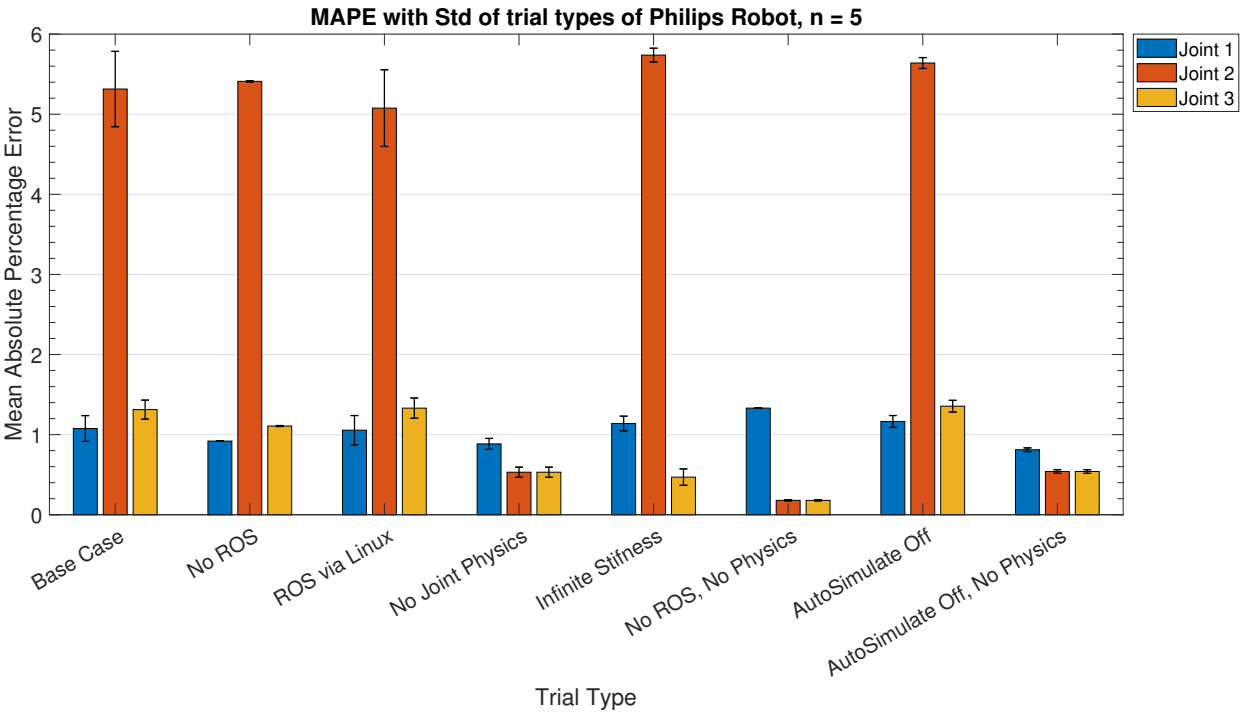


Figure 19: Mean Absolute Percent Error of Philips robot joints per Use Case

Similarly, five of the eight use cases were run for the Panda robot. The ROS via Linux test is not done as it is similar to the Base case, and the AutoSimulate tests are not done due to the fact that the Simulink runs in real time as no IK calculations are done in Simulink for this test.

The remaining test cases are seen in Figure 20, where there are surprisingly different results than

the Philips robot use case applications. While once again removing ROS also reduces the standard deviations, the MAPE for joints 1 and 3 drastically increase as well. This could be partially explained by the fact that MAPE artificially increases errors to infinity as the reference data approaches zero. As joints 1 and 3 both cross zero radians from Figure 18b, this could be an explanation. However, then this artifact would also be expected in the other use cases. The base case also has a factor 10 lower error in the Panda robot than the Philips robot, which would suggest it to be a reasonable use case for this robot. Using infinite stiffness also further improves this robot. Therefore, the current recommendation is to enable use cases 1 or 5 in order to achieve the best results for the Panda robot.

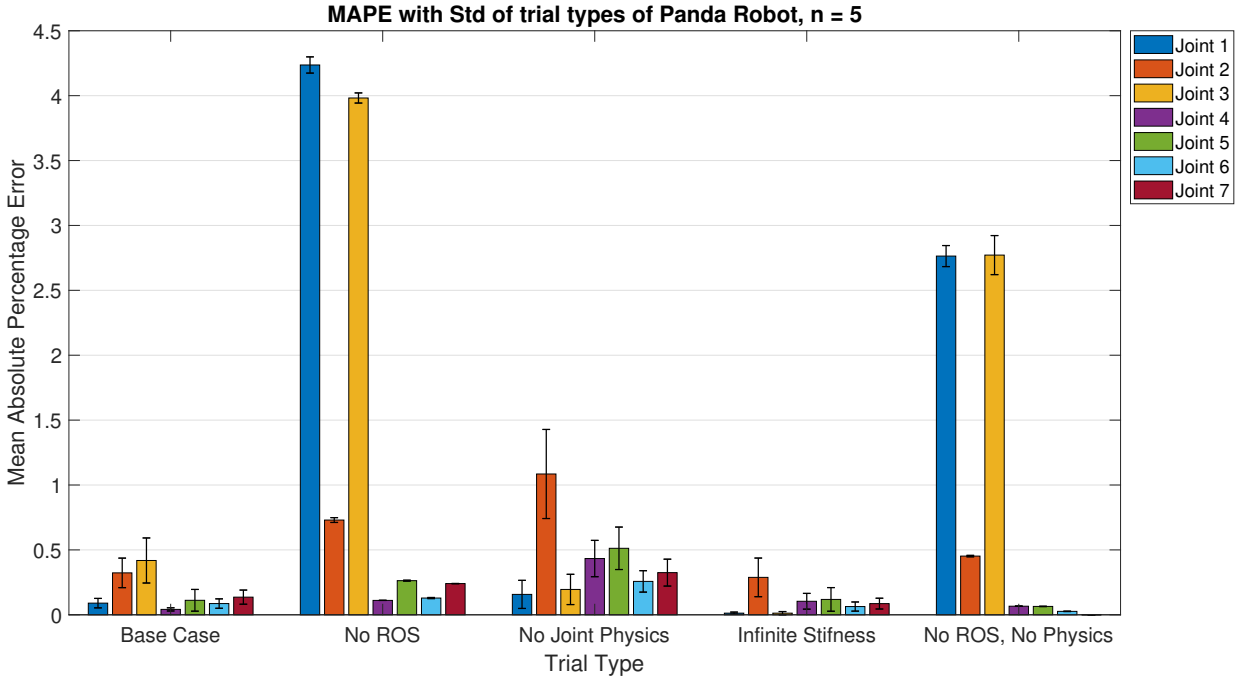


Figure 20: Mean Absolute Percent Error of Panda robot joints per Use Case

The results from both robots suggest that the ROS connection while Simulink runs introduces the higher standard deviation. This is likely due to the Simulink running delay, as the standard deviation also decreased when syncing the Unity clock to the Simulink clock with AutoSimulate being off. Also, it appears that the robot configuration heavily influences the magnitude of errors in the base case.

Further research must be done in order to trace the origin of the higher error results in the Panda use cases where the ROS connection is removed. Also, the error in the Panda robot with no joint physics are unexpected as this should in theory be exactly the joint placements the Simulink request without further physics engines introducing an error. However, as this is not the case, further investigated is this use case is to be used in the future. Finally, these results on the accuracy of the robots suggest that for each robot that is developed in Unity a similar analysis should be done to determine if the base case is sufficiently accurate, or if further steps need to be taken in order to enforce the lowest error possible.

8 Conclusion

This report is meant to be a precursor to further research on robot perception in VR. This research has been proposed to be based off of users taking the role of the operator and determining how well perception of a collaborative robot in VR and in reality match up. In order to facilitate this research, the Panda robot has been implemented in VR. Philips IGT started this project with the request to implement their robot in VR as well in order to support current master thesis projects, which has also been completed.

Both robot's accuracy within various use cases has been tested and confirmed to be accurate for the usability within one percent of the desired joint states. Repositories containing basic Simulink files for the running of the Unity projects, as well as 3D and VR versions of both robots' projects, have been created. Furthermore, the public Panda repository also contains universal Unity scripts to further stimulate creation of VR projects within Unity [2].

In order to further stimulate research and development of robots in Unity, not only have universal C# scripts been created, but within the public panda repository, a wiki has been created to give detailed information about the creation of a 3D or VR Unity project, implementing ROS, and using Simulink in order to control the movement of the robot. This wiki includes references to the created items in the repository, but also includes references to other repositories and sources on the development of such projects. This is especially in reference to specialized Unity packages that are useful in importing a robot to Unity and connecting to ROS, both of which may be completely new to people starting such a project [3].

The desired project goals have been reached in this report. However, there are still various instances of improvement available for future research. One of these is understanding the source of errors in the various use cases from section 7, where the results from the Philips IGT robot and the results from the Panda robot were wildly different. Understanding where these discrepancies originate from and potentially further suppressing them will lead to a better understanding of error sources in other robots that may be implemented in VR in the future. Also, it is valuable in order to be able to give a universal recommendation for a preference of a specific use case from section 7.

Another instance of improvement would be to create a universal strategy in order to remove the joint physics, instead of making the C# code dependent on the robot configuration. This would again aid in making this project more universally applicable to other robots. Finally, the last point of further study would include the errors seen in translating from Cartesian coordinates and rotations into the IK of the robot, especially around the orientation of the EE. This also holds for the errors seen in the GetTransform Simulink block that have been compensated for with a robot dependent FK script. Being able to simply use the GetTransform Simulink block in the future would be valuable for the universality of similar projects.

Implementing these improvements of the project is preferable, but is not a requirement to having a working final product. As is, this project is a functioning product that can be improved in the future. The next steps include creating the specific robot paths required in order to implement the research proposal and start work on the experiments.

References

[1] M. Mara, J. P. Stein, M. E. Latoschik, B. Lugrin, C. Schreiner, R. Hostettler, and M. Appel, "User Responses to a Humanoid Robot Observed in Real Life, Virtual Reality, 3D and 2D," *Frontiers in Psychology*, vol. 12, 4 2021.

[2] C. Vissers, "Robot Unity VR Repository," 2023. [Online]. Available: https://gitlab.tue.nl/et_projects/cv-robot-unity-vr

[3] ——, "Robot Unity VR Wiki," 2023. [Online]. Available: https://gitlab.tue.nl/et_projects/cv-robot-unity-vr/-/wikis/home

[4] O. Sadka, J. Giron, D. Friedman, O. Zuckerman, and H. Erel, "Virtual-reality as a simulation tool for non-humanoid social robots," in *Conference on Human Factors in Computing Systems - Proceedings*. Association for Computing Machinery, 4 2020.

[5] H. Kamide, Y. Mae, T. Takubo, K. Ohara, and T. Arai, "Direct comparison of psychological evaluation between virtual and real humanoids: Personal space and subjective impressions," *International Journal of Human Computer Studies*, vol. 72, no. 5, pp. 451–459, 5 2014.

[6] E. Matsas, G. C. Vosniakos, and D. Batras, "Effectiveness and acceptability of a virtual environment for assessing human–robot collaboration in manufacturing," *International Journal of Advanced Manufacturing Technology*, vol. 92, no. 9-12, pp. 3903–3917, 10 2017.

[7] V. Weistroffer, A. Paljic, P. Fuchs, O. Hugues, J. P. Chodacki, P. Ligot, and A. Morais, "Assessing the acceptability of human-robot co-presence on assembly lines: A comparison between actual situations and their virtual reality counterparts," in *IEEE RO-MAN 2014 - 23rd IEEE International Symposium on Robot and Human Interactive Communication: Human-Robot Co-Existence: Adaptive Interfaces and Systems for Daily Life, Therapy, Assistance and Socially Engaging Interactions*. Edinburgh: Institute of Electrical and Electronics Engineers Inc., 8 2014, pp. 377–384.

[8] K. Inoue, S. Nonaka, Y. Ujiie, T. Takubo, and T. Arai, "Comparison of human psychology for real and virtual mobile manipulators," in *Proceedings - IEEE International Workshop on Robot and Human Interactive Communication*, vol. 2005, 2005, pp. 73–78.

[9] W. A. Bainbridge, J. W. Hart, E. S. Kim, and B. Scassellati, "The benefits of interactions with physically present robots over video-displayed agents," *International Journal of Social Robotics*, vol. 3, no. 1, pp. 41–52, 2011.

[10] S. Woods, M. Walters, K. L. Koay, and K. Dautenhahn, "Comparing human robot interaction scenarios using live and video based methods: Towards a novel methodological approach," in *International Workshop on Advanced Motion Control, AMC*, vol. 2006, 2006, pp. 750–755.

[11] S. H. Seo, D. Geiskkovitch, M. Nakane, C. King, and J. E. Young, "Poor Thing! Would You Feel Sorry for a Simulated Robot?: A comparison of empathy toward a physical and a simulated robot," in *ACM/IEEE International Conference on Human-Robot Interaction*, vol. 2015-March. IEEE Computer Society, 3 2015, pp. 125–132.

[12] A. Ayala García, I. Galván Bobadilla, G. Arroyo Figueroa, M. Pérez Ramírez, and J. Muñoz Román, "Virtual reality training system for maintenance and operation of high-voltage overhead power lines," *Virtual Reality*, vol. 20, no. 1, pp. 27–40, 3 2016.

[13] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, 5 2009. [Online]. Available: http://stair.stanford.edu

[14] B. J. Caasenbrood, F. E. Van Beek, H. K. Chu, and I. A. Kuling, "A Desktop-sized Platform for Real-time Control Applications of Pneumatic Soft Robots," in *2022 IEEE 5th International Conference on Soft Robotics, RoboSoft 2022*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 217–223.

[15] Oasis Open, "MQTT 5 Specification," Tech. Rep., 3 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf

[16] The MathWorks Inc., "MQTT Basics." [Online]. Available: https://nl.mathworks.com/help/thingspeak/mqtt-basics.html

[17] ——, "TCP/IP Communication." [Online]. Available: https://nl.mathworks.com/help/matlab/tcpip-communication.html

[18] Open Source Robotics Foundation, "Documentation - ROS Wiki." [Online]. Available: http://wiki.ros.org/

[19] Microsoft, "Windows Subsystem for Linux Documentation," 6 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/

[20] ROS Documentation, "rosgraph_msgs/Clock Documentation," 2 2022. [Online]. Available: http://docs.ros.org/en/melodic/api/rosgraph_msgs/html/msg/Clock.html

[21] B. v. d. Dool, "Panda Simulink Twin," Eindhoven, 2022.

[22] ROS Documentation, "geometry_msgs Msg/Srv Documentation," 3 2022. [Online]. Available: http://docs.ros.org/en/noetic/api/geometry_msgs/html/index-msg.html

[23] ——, "sensor_msgs/JointState Documentation," 2 2022. [Online]. Available: http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/JointState.html

[24] ——, "trajectory_msgs Msg/Srv Documentation," 2 2022. [Online]. Available: http://docs.ros.org/en/melodic/api/trajectory_msgs/html/index-msg.html

[25] Unity Technologies, "Unity-Technologies/URDF-Importer: URDF importer," 2 2022. [Online]. Available: https://github.com/Unity-Technologies/URDF-Importer

[26] ——, "Unity-Technologies/ROS-TCP-Connector," 2 2022. [Online]. Available: https://github.com/Unity-Technologies/ROS-TCP-Connector

[27] ——, "XR Interaction Toolkit 2.2.0," 10 2022. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.2/manual/index.html

[28] ——, "Articulation Body component reference," 12 2022. [Online]. Available: https://docs.unity3d.com/Manual/class-ArticulationBody.html

[29] ——, "Unity Scripting API," 12 2022. [Online]. Available: https://docs.unity3d.com/ScriptReference/index.html

[30] J. Vissers, "Bachelor Final Project A Matlab/Simulink-ROS architecture for dual arm control," Eindhoven University of Technology, Eindhoven, Tech. Rep., 6 2022.

[31] T. Sugihara, "Solvability-unconcerned inverse kinematics by the levenberg-marquardt method," *IEEE Transactions on Robotics*, vol. 27, no. 5, 10 2011.

[32] The Mathworks Inc, "Inverse Kinematics." [Online]. Available: https://nl.mathworks.com/help/robotics/ref/inversekinematics.html

[33] Franka Control GmbH, "Robot and interface specifications," 2017. [Online]. Available: https://frankaemika.github.io/docs/control_parameters.html#denavithartenberg-parameters

[34] S. Kim and H. Kim, "A new metric of absolute percentage error for intermittent demand forecasts," *International Journal of Forecasting*, vol. 32, no. 3, pp. 669–679, 7 2016.

A Appendix: Code Snippets

This Appendix includes the various code snippets mentioned in the main text that are not directly required in the main text. For full code implementation, see the public Panda repository [2].

In order to start the ROS master and ROS node within the Ubuntu distro on WSL, Code A.1 and Code A.2 must be run in different terminals.

Code A.1: Master ROS

```
export ROS_HOSTNAME=<WSL IP>
roscore
```

Code A.2: Node ROS

```
export ROS_HOSTNAME=<WSL IP>
cd ./catkin_ws/
source devel/setup.bash
roslaunch ros_tcp_endpoint endpoint.launch tcp_ip:=<WSL IP> tcp_port:=<
    Unity Port #>
```

In order to create the joint message for the topic /panda/arm_controller/command, the Code A.3 is used in Simulink before publishign the message.

Code A.3: Populate Joint Message

```
function msgOut = populateMsg(msgIn,msgPoint,Points)

msgOut = msgIn;

% Assign Joint Names
jointNames = {'panda_1_joint1','panda_1_joint2','panda_1_joint3','
    panda_1_joint4','panda_1_joint5','panda_1_joint6','panda_1_joint7'
    };
numJoints = uint32(length(jointNames));
msgOut.JointNames_SL_Info.CurrentLength = numJoints;
for idx = 1:numJoints
    jName = jointNames{idx};
    lenName = uint32(length(jName));
    msgOut.JointNames(idx).Data(1:lenName) = uint8(jName);
    msgOut.JointNames(idx).Data_SL_Info.CurrentLength = lenName;
end

% Assign Points
msgOut.Points = msgPoint;
msgOut.Points.Positions(1:7) = Points(:,1);
msgOut.Points.Positions_SL_Info.CurrentLength = uint32(7);
msgOut.Points.Velocities(1:7) = zeros();
msgOut.Points.Velocities_SL_Info.CurrentLength = uint32(7);
msgOut.Points_SL_Info.CurrentLength = uint32(1);
```

In order to work around the GetTransform problem, as further explained in Appendix B, the FK

script for the Panda arm is made manually, as seen in Code A.4. This is required in order to confirm the desired EE trajectory is followed.

Code A.4: Forward Kinematics script

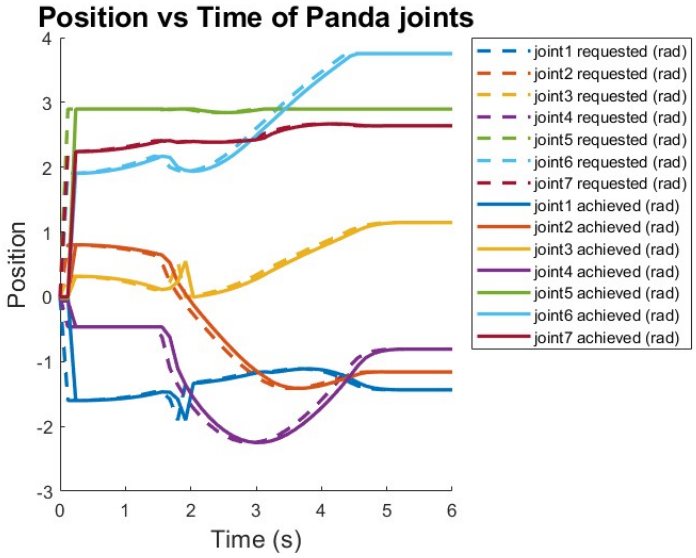
```
function point = FK(joint_state)
dh =    [0,       0,         0.333,    joint_state(1);
         -pi/2,   0,         0,        joint_state(2);
         pi/2,    0,         0.316,    joint_state(3);
         pi/2,    0.0825,    0,        joint_state(4);
         -pi/2,   -0.0825,   0.384,    joint_state(5);
         pi/2,    0,         0,        joint_state(6);
         pi/2,    0.088,     0.107,    joint_state(7)];


T_01 = getTF(1,dh);
T_12 = getTF(2,dh);
T_23 = getTF(3,dh);
T_34 = getTF(4,dh);
T_45 = getTF(5,dh);
T_56 = getTF(6,dh);
T_67 = getTF(7,dh);

T_07 = T_01*T_12*T_23*T_34*T_45*T_56*T_67;

point = [tform2trvec(T_07) tform2eul(T_07, 'XYZ')];
end
function TF = getTF(i, dh)
    alpha = dh(i,1);
    a = dh(i,2);
    d = dh(i,3);
    q = dh(i,4);

    TF =    [cos(q),              -sin(q),              0,              a;
             sin(q)*cos(alpha),   cos(q)*cos(alpha),   -sin(alpha),    -
                sin(alpha)*d;
             sin(q)*sin(alpha),   cos(q)*sin(alpha),   cos(alpha),     cos
                (alpha)*d;
             0,                   0,                   0,              1];

end
```

B Appendix: Get Transform Inaccuracy

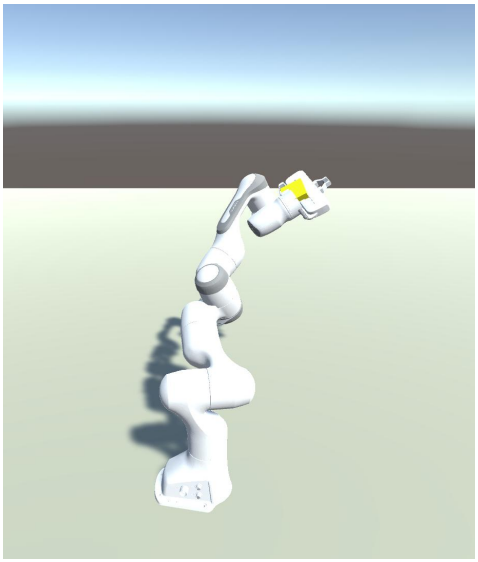
The FK are desired in order to confirm the EE is reaching the desired Cartesian position and orientations as specified by the polynomial trajectory. Without this, there is simply the assumption that the IK block is supplying the correct joint states, and that the Unity visualization is also correct. By being able to plot the EE's movement over time and through the 3D space, this assumption is no longer required and the project can be considered accurate.

The built in Simulink block is the GetTransform block that, when given a robot rigid body tree from a URDF file, the joint states reached by the robot, and the desired reference frames (i.e. the world reference frame to the EE frame), will output a homogeneous transformation matrix that can be further manipulated to quire the Cartesian points and the Euler angles. However, the GetTransform seems to output the incorrect results.

As can be seen in Figure B.1a the joints do reach the desired joint positions, and in unity the arm reaches the desired end point (the yellow block) in Figure B.1b.



(a) Joint positions over time with polynomial trajectory on Cartesian points



(b) Final Position

Figure B.1: Polynomial trajectory on Cartesian points

This data is retrieved from the Simulink file set up as seen in Figure B.2.

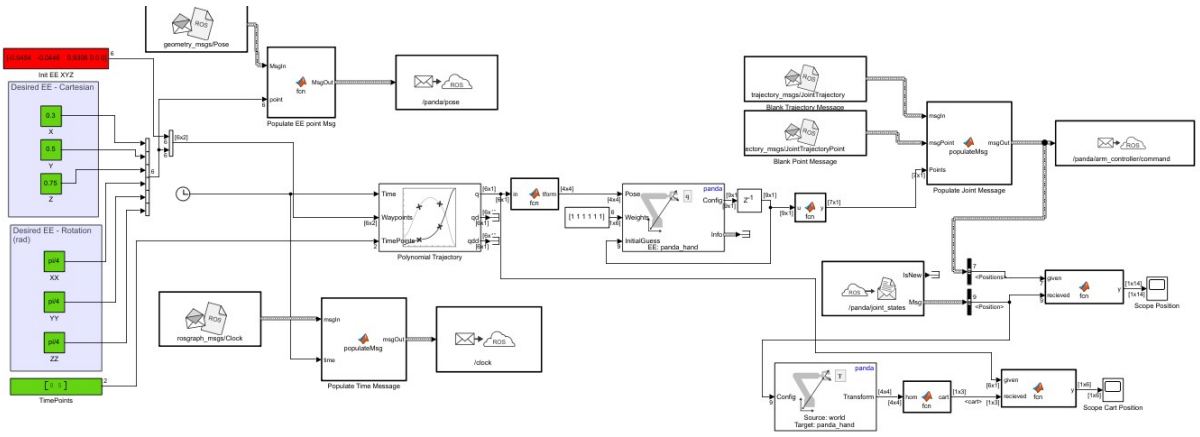


Figure B.2: Incorrect FK Simulink

However, when using the GetTransform Simulink block to take the joint positions and translate them back to Cartesian coordinates to once again confirm it, the GetTransform does not give accurate results, as seen in Figure B.3. The data points are the desired final position.

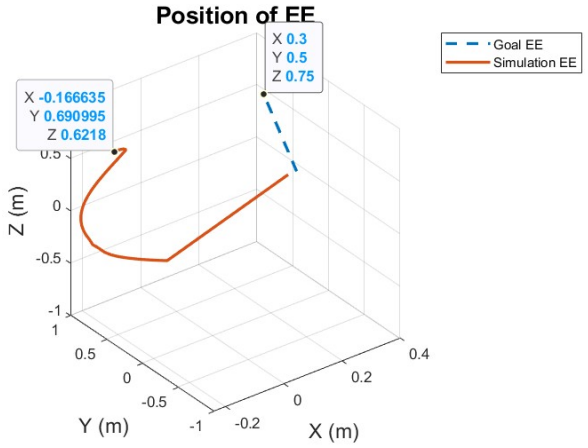


Figure B.3: Incorrect FK with Unity

Clearly there is something wrong with this. One theory was that perhaps that Unity or the ROS may be negatively influencing the joint states, so the output of the IK block was input directly into the GetTransform block, and the results are extremely similar, as seen in Figure B.4. Therefore, it is not a Unity issue, but a GetTransform issue instead. Logically, inputting the output of the Inverse Kinematics block into the GetTransform block would give result in the output of the GetTransform and the input of the Inverse Kinematics to be identical. However, this is not the case.

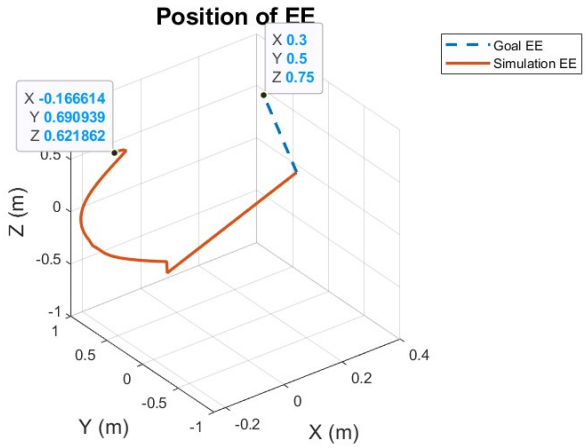


Figure B.4: Incorrect FK without Unity

By implementing the Panda robot's DH parameter's into an FK script, this issue is overcome, as seen in Figure 17. However, it violates the desired universality of this project. Using the GetTransform block would therefore be preferable in the future.

C Appendix: Philips IGT Robot Results

In order to confirm the Philips IGT robot works as well in Unity, similar tests to the Panda tests from section 5 are preformed. Figure C.1 shows that the joints move accurately to the desired point without orientation being defined.

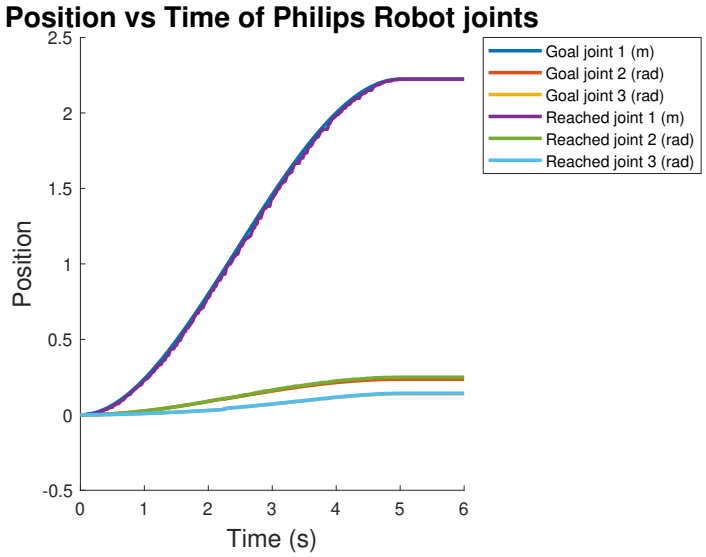


Figure C.1: Philips robot movement to point

Adding an orientation constraint to the position constraint ends in Figure C.2, where the Unity simulation also closely tracks the desired positions.

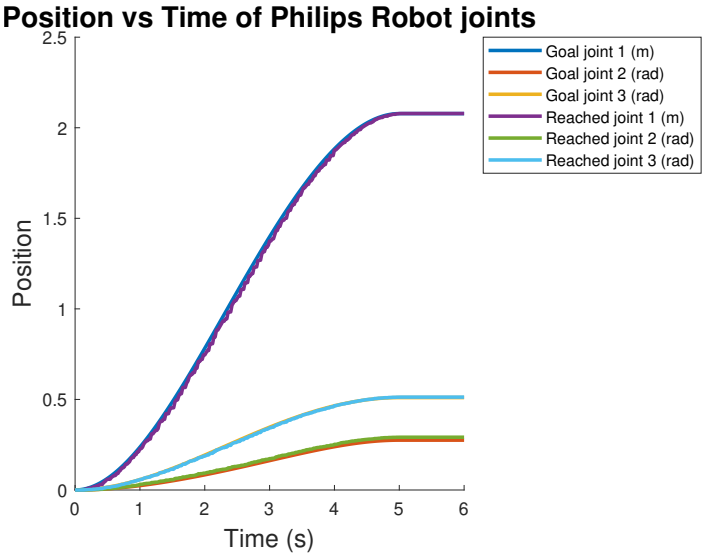
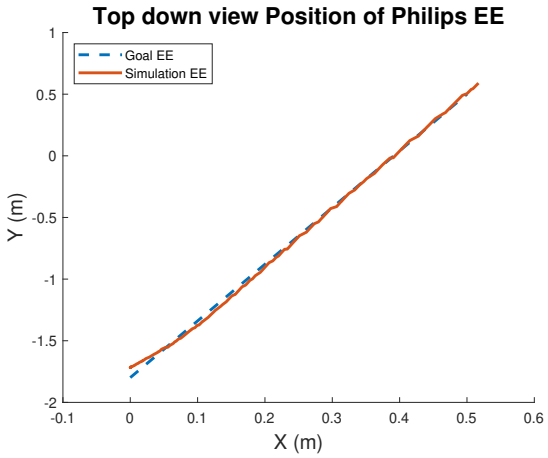
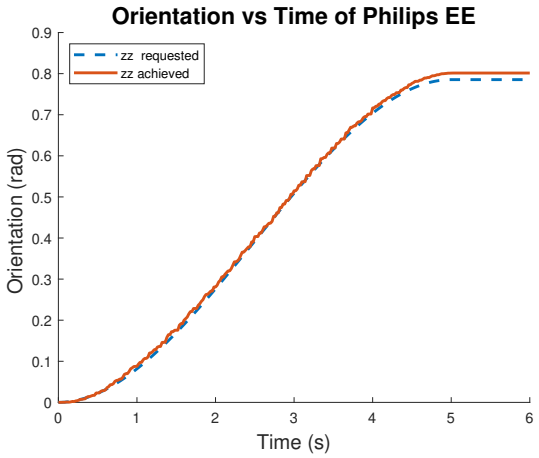


Figure C.2: Philips robot movement to point and rotation

A FK script was also implemented for these joints, and a top-down view of the EE path and the EE orientation over time of the EE were tracked in Figure C.3. This confirms that the Philips robot implementation is accurate.



(a) EE path in Cartesian coordinates



(b) EE orientation over time

Figure C.3: Confirmation of reached position with forward kinematics for the Philips robot