UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Corso di Laurea Magistrale in Ingegneria Informatica

Robotics projects

Lecturers:

Ch.mo Prof.
Pasquale CHIACCHIO
pchiacchio@unisa.it
Ch.mo Prof.
Enrico FERRENTINO
eferrentino@unisa.it

Authors:

Davide RISI
Mat. 0622702013
d.risi2@studenti.unisa.it

Giovanni INTONTI
Mat. 0622701994
g.intonti@studenti.unisa.it

Adolfo PAGANO
Mat. 0622702010
a.pagano93@studenti.unisa.it

Guglielmo BARONE
Mat. 0622701885
g.barone36@studenti.unisa.it

ANNO ACCADEMICO 2023/2024

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

The aim of the project work is to provide a ROS-based planning system for the Fanuc M20iA/35M robot, which has no support for the ROS2 environment. To be more specific, the project includes the description of the robot, thus definition of the model in a URDF file, then integration with the MoveIt! 2 framework for trajectory planning, a dynamic simulation through the Ignition Gazebo environment, control through the independent joint control technique and task space velocity control, and finally MSE-based analysis of the trajectory execution of the robot. All of these features will be handled in multiple Work Packages (WPs), which aim at satisfying the different functional and non-functional requirements given by the client. In each of the sections of the following report a description of the functionalities delivered for the respective WP will be provided, including all the logical steps required for reaching a specific output in the development pipeline.

CHAPTER 2

WP1-SYSTEM SETUP FOR PLANNING WITH MOVEIT

In this chapter all the procedures for the generation of the URDF file and the setup for planning will be described, starting from the kinematics description and the application of the Denavit-Hartenberg convention, up until the usage of the Setup Assistant from the MoveIt! 2 framework.

2.1 Kinematic Description of the Fanuc M20iA/35M

The first step in building a ROS-compatible robot description consists in analyzing the robot's effective kinematics, thus understanding the number of degrees of freedom, the action space, and how to describe each joint of the robot. According to the given datasheets, the Fanuc M20iA/35M (which will simply called "Fanuc" for the rest of the report) robot has 6 degrees of freedom established by its 6 original joints. Logically speaking, we grouped the 6 joints into 2 categories:

- the "arm" joints, which are the first 3 joints starting from the bottom of the robot that define the spatial sphere in which the robot can move;
- the "hand" joints, which are the last 3 joints in the kinematic chain that allow multi-directional orientation of the end effector that will be mounted on the last joint.

All the "arm" and "hand" joints are revolute joints, which means their movement can be described through relative rotations of specific amounts of radians. In order to allow the robot to have access to spatial areas outside the sphere defined by the "arm" joints an additional joint is included at the base of the Fanuc robot: a prismatic joint represented by a slider. With the addition of the slider the robot now has 7 degrees of freedom. Given all these informations, it is necessary to define a kinematics description. Given that each joint connects just two links, it is reasonable to consider the problem of describing kinematics between

consecutive links first and then to obtain the complete transformation matrix from the base to the end effector.

This is the main concept behind the Denavit-Hartenberg (DH) convention, which is a method to describe the kinematics of a robot. The rest of the paragraph will deal with how the parameter table was obtained.

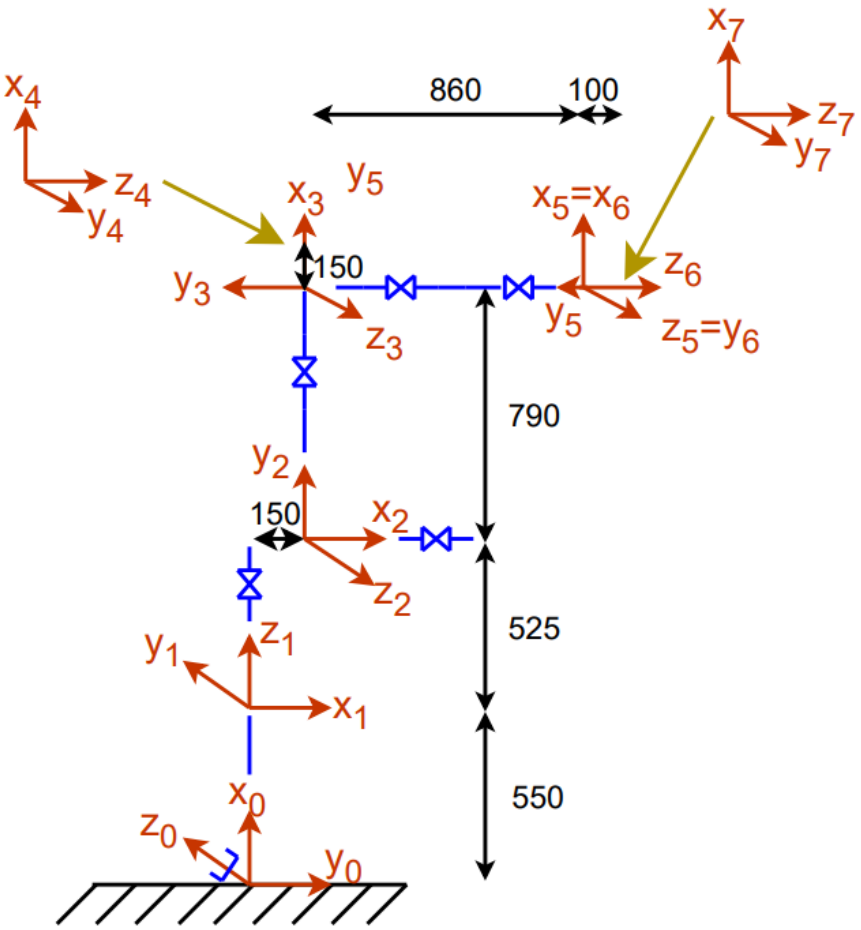


Figure 2.1: Fanuc robot kinematics description using DH convention.

Let Σ^i (with i being the i-th joint axis which connects the $i-1$-th link to the i-th one) be the i-th coordinate frame, with $i \in 1, \ldots, 7$.

The first coordinate frame Σ^0 will be related to the slider, whose movement direction is already defined, thus the z-axis on the first joint of the kinematic chain z_0 will follow said direction. In order to complete the coordinate frame, the origin point O_0 must be defined, which is usually done by considering the common normal between z_i and z_{i-1}, but since there is no previous joint, the x_0 axis can be arbitrarily defined, thus allowing to place O_0 right at the base of the prismatic joint represented by the slider. In order to make calculations of the a_1 parameter easier, the direction of x_0 is defined according to the following joint axis z_1. Finally, by placing y_0 in a way that completes Σ^0 as a right-hand frame.

The coordinate frame Σ^1 is related to the first revolute joint, thus the joint axis z_1 is placed along

the joint axis of the first revolute joint. Given that the z_0 axis intersect the z_1 axis, the origin point O_1 can be identified as the intersection point (an offset of 550 mm from the intersection point is actually included to represent the height at which the base of the robot is mounted to the slider). Now, for the same reason, the direction of the x_1 axis is arbitrary, thus it is defined pointing to the right. Finally, the coordinate frame Σ^1 is completed by placing y_1 in a way that completes a right-hand frame.

Regarding the coordinate frame Σ^2, its joint axis z_2 is placed alongside the joint axis of the second revolute joint. The origin point O_2 can be identified thanks to the intersection between z_2 and the only common normal between z_1 and z_2. Thus, it is possible to identify the x_2 axis which will follow the direction of the common normal. Finally the y_2 axis is placed in a way that completes a right-hand frame.

Regarding the Σ^3 frame, the z_3 axis can be placed on the joint axis of he third revolute joint. This time there are infinite common normals between z_2 and z_3, thus the origin point O_3 for frame Σ^3 is placed right at the base of the third joint, along the link of the robot. It is also possible to define the direction of the x_3 axis with positive sense in the direction of the new joint (so upward). Finally the y_3 is placed in a way the completes a right-hand frame.

Regarding the Σ^4 frame, which deals with the fourth revolute joint, it is possible to fix z_4 as the joint axis, while the origin point can be obtained as the intersection between axis z_4 and the common normal between z_3 and z_4 (this time z_3 is perpendicular to z_4), which means that the sense of axis x_4 is arbitrary, thus its positive sense is chosen to be upwards. The y_4 is chosen as completing a right-hand frame.

Regarding the Σ^5 frame, the z_5 axis can be defined as the joint axis of the fifth revolute joint. Once again the origin point O_5 is given by the intersection between the common normal between z_4 and z_5 axis, thus it can be placed right at the base of the joint. Also in this case the x_5 was choice arbitrarily, and upward to match the previous frames. Finally the y_5 axis is placed in a way that completes a right-hand frame.

By following a similar procedure the 2 final frames Σ^6 and Σ^7 can be obtained. To be more specific, frame Σ^7 is the one related to the end effector.
The output of the analysis of the robot kinematics is Figure 2.1 describing the coordinate frames for each joint. After applying the DH convention, the parameter table 2.1 is obtained according to the DH rules, (the a and d are expressed in meters, while the α and θ parameters are expressed in radians).
Given this description of the robot kinematics, it is possible to calculate the transformation matrix between each joint and the next one, thus obtaining the complete transformation matrix from the base to the end effector. This will be useful in the next steps of the project, in particular for the implementation of the forward and inverse kinematics algorithms, and were also used to obtain the right translation and rotations

Link	$\mathbf{a_i}$	$\mathbf{d_i}$	$\alpha_\mathbf{i}$	$\theta_\mathbf{i}$
$0 \rightarrow 1$	0.55	q_1	$\frac{\pi}{2}$	$\frac{\pi}{2}$
$1 \rightarrow 2$	0.15	0.525	$\frac{\pi}{2}$	q_2
$2 \rightarrow 3$	0.79	0	0	$q_3 + \frac{\pi}{2}$
$3 \rightarrow 4$	0.15	0	$\frac{\pi}{2}$	q_4
$4 \rightarrow 5$	0	0.86	$-\frac{\pi}{2}$	q_5
$5 \rightarrow 6$	0	0	$\frac{\pi}{2}$	q_6
$6 \rightarrow 7$	0	0.1	0	q_7

Table 2.1: Denavit-Hartenberg Parameter Table

matrices for the URDF file, but also for make the right translations and rotations to inertia matrices and center of mass of each link, as it will be explained in the next paragraphs.

2.2 Building the URDF

After understanding the Fanuc robot's geometry and how to work with the coordinate frames of each of its joints, it is finally possible to work on the robot's description in terms of URDF (Unified Robot Description Format) file.

2.2.1 RPY angles evaluation

Generally speaking, the mapping of the parameters is immediate in terms of x,y and z displacement, however the URDF convention for angles is different from the DH convention, which means that there is the need to transform the angles from one convention to the other. In particular, the URDF convention uses the RPY (Roll-Pitch-Yaw) convention, so, starting from the inverse formula of the rotation matrix:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.1}$$

taking into account the terms of the matrix, the RPY angles can be obtained as described in [1]:

$$\begin{aligned} \phi &= \text{Atan2}(r_{21}, r_{11}) \\ \theta &= \text{Atan2}(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}) \\ \psi &= \text{Atan2}(r_{32}, r_{33}) \end{aligned} \tag{2.2}$$

In order to quickly tackle the calculations of these angles for each joint, a Python script named `rotation_to_rpy.py` was produced which, starting from an input rotation matrix, evaluates the RPY angles, it is included in the *supporting_scripts* folder of the project.

2.2.2 Working on the URDF file

Right after the computation of the RPY angles, the URDF file is created. A new ROS2 package is created with the name *acg_resources_fanuc_m20ia_35m_description*, according to the Automation Control Group's naming convention, and inside it a new folder named *urdf* is created, which will contain the URDF file. In the URDF it is possible to define the robot's links and joints, and the relations between them. The URDF file is created by using the *xacro* package, which allows to define macros and to include other files in the main file.

The main file is the *fanuc_m20ia_35m.xacro* file, which includes the *inertial.xacro* used to provide the inertial parameters of each link that will be described in the next section. In order to define the robot link's meshes, there is no need to create a new xacro file, but the visual and collision meshes are defined directly in the main file. The provided meshes had a rotation and translation according to the coordinate frame of the first joint of the robot, and since the URDF convention requires the meshes to be defined in their respective link's frame, the meshes are rotated and translated again in order to match that. This is valid for all the meshes except for the "drilling_tool" and the "slider_tool", that are external to the robot, and for the "fanuc_m20ia_35m_link_6" that have a different coordinate frame from the other links, in particular, all the meshes were rotated according to their link's frame.

The meshes are defined in the *meshes* folder of the relative package. Additionally a *config* folder is created, which contains the *inertial.yaml* file, which contains the inertial parameters of each link, and a RViz configuration file, which is used to visualize the robot in RViz with the useful plugins such as the *TF plugin*, which allows to visualize the coordinate frames of the robot.

All the information about links and joints were obtained from the Fanuc's datasheet provided, but other resources were consulted to better undestand the joint's limits and the inertial parameters of the robot. To be more specific, here is how each piece of the robot is described in the URDF:

2.2.3 Links

- **world**, which is the world's frame and is fixed in the environment (it is not a link of the robot);
- **slider_tool**, which represent the slider that moves along the rail;
- **fanuc_m20ia_35m_link_0**, which is the base of the robot;
- **fanuc_m20ia_35m_link_1**, which is the first link of the robot;
- **fanuc_m20ia_35m_link_2**, which is the second link of the robot;
- **fanuc_m20ia_35m_link_3**, which is the third link of the robot;
- **fanuc_m20ia_35m_link_4**, which is the fourth link of the robot;
- **fanuc_m20ia_35m_link_5**, which is the fifth link of the robot;
- **fanuc_m20ia_35m_link_6**, which is the sixth link of the robot;

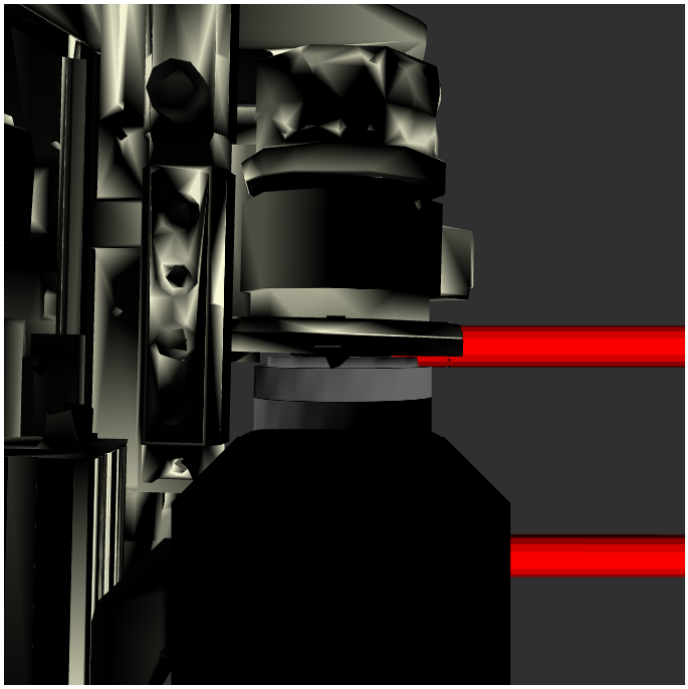


Figure 2.2: Drilling tool mesh in `Rviz`.

- **drilling_tool**, which is the drilling tool of the end effector of the robot.
- **fastening_tool**, which is the fastening tool of the end effector of the robot.

Each link is defined by a `urdf:link` tag, which contains the link's name, the link's mesh and the link's inertial parameters. To position the link's mesh in the right position, the `urdf:origin` tag is used, which allows to define the link's mesh position and orientation with respect to the link's frame. To perform this task, the previous Denavit Hartenberg information are used, moving and rotating both visual and collision meshes in accord to the DH convention. This is not valid, as previously stated, for the external tools and in particular for the "drilling_tool" where the collision mesh was manually rotated to remove a wrong degree of rotation that was present in the original mesh (of about 1.5 degree) as shown in figure 2.2.

A note to this is that, to correctly visualize the meshes in Gazebo, without making the program crash, an import/export process into `Blender` was computed, so the meshes were again rotated to match the `Blender` orientation.

2.2.4 Joints

To introduce the joints of the robot, the DH convention previously described in 2.1 is referred to. In particular the `origin` tag of each joint is obtained using the table 2.1 and the script mentioned before for the angles, the `axis` tag is obtained by the DH convention, so all the joints will have the same axis that is the z one, and the limits tag is obtained by the datasheet of the robot [2] ,[3]. Additional attention needs to be given to the `dynamics` tag, which is used to define the joint's friction and damping values. The values inserted in this tag were obtained by looking at a similar robot on which parameters identification was performed: [4].

- **joint_world_to_slider**, which is the joint between the world and the slider, is a prismatic joint with a limit of -2.1 m and 2.1 m and, since no information of limit effort was provided, it is set to 10000, because this joint is not the bottleneck of the robot, the max velocity is set to 9.6 m/s;
- **fanuc_m20ia_35m_joint_slider_to_link_0**, which is the fixed joint between the slider and the base of the robot;
- **fanuc_m20ia_35m_joint1**, which is the first revolute joint of the robot, has a limit of -3.22 rad and 3.22 rad and a limit effort of 10000, because also in this case no information about the limit effort was provided, the max velocity is set to 3.14 rad/s;
- **fanuc_m20ia_35m_joint2**, which is the second revolute joint of the robot, has a limit of -2.79 rad and 1.74 rad and a limit effort of 10000 , the max velocity is set to 3.14 rad/s;
- **fanuc_m20ia_35m_joint3**, which is the third revolute joint of the robot, has a limit of -4.81 rad and 3.22 rad and a limit effort of 10000, the max velocity is set to 3.49 rad/s;
- **fanuc_m20ia_35m_joint4**, which is the fourth revolute joint of the robot, has a limit of -3.49 rad and 3.49 rad and a limit effort of 110, the max velocity is set to 6.11 rad/s;
- **fanuc_m20ia_35m_joint5**, which is the fifth revolute joint of the robot, has a limit of -2.44 rad and 2.44 rad and a limit effort of 110, the max velocity is set to 6.11 rad/s;
- **fanuc_m20ia_35m_joint6**, which is the sixth revolute joint of the robot, has a limit of -7.88 rad and 7.88 rad and a limit effort of 110, the max velocity is set to 6.98 rad/s;
- **fanuc_m20ia_35m_joint_link6_to_drilling_tool**, which is the fixed joint between the sixth link and the drilling tool;
- **fanuc_m20ia_35m_joint_link6_to_fastening_tool**, which is the fixed joint between the sixth link and the fastening tool.

2.2.5 Inertial parameters

After defining the links and joints of the robot, the inertial parameters of each link are defined. In this section it will be described how these parameters are obtained and what tools are used to obtain them.

2.2.5.1 Meshes cleaning

Since no information about the inertial parameters of the robot were provided, it is necessary to obtain them through other means. By looking at ROS2 and Gazebo documentation, it is possible to understand that the inertial parameters of each link can be obtained by the meshes as explained in [5]. In order to perform this task, a useful tool is *MeshLab*, which is a software for the editing and processing of 3D meshes. An empirical check was performed on the SmartSix robot, which is a 6 DOF robot arm, in order to understand

if the procedure described in Gazebo documentation is correct. After obtaining the inertial parameters of the SmartSix robot, they were compared with the ones contained in the URDF file of the robot from the University of Salerno repository, and the results were comparable, and in some cases were exactly the same. Giving that the procedure is correct, it is possible to obtain the inertial parameters of the Fanuc robot. The first step is to import the meshes of the robot in *MeshLab*, and then to perform a cleaning procedure on the meshes, which is necessary:

- **Remove Duplicate Faces**, which removes all the faces that are duplicated;
- **Remove Duplicate Vertices**, which removes all the vertices that are duplicated;
- **Remove Zero Area Faces**, which removes all the faces that have zero area;
- **Remove non Manifold Edges**, which removes all the edges that are not manifold;
- **Close Holes**, which closes all the holes in the mesh.

All this operations are performed by selecting the filters from the *Filters* menu. This procedure is necessary because the software uses the volume as a parameter to evaluate the inertial parameters, and if the mesh is not clean, the volume will be wrong. This procedure was a bit different for the ”drilling_tool” and the ”slider_tool” and also for two links of the robot, the ”fanuc_m20ia_35m_link_2” and the ”fanuc_m20ia_35m_link_3”, because the cleaning tool was not able to remove all the non manifold edges, so the ”Close Holes” filter was not able to close all the holes. For this reason, some other steps were necessary. For the two external tools:

- **Convex Hull**, which creates a solid mesh that approximates the original mesh;
- **Close Holes**, which closes all the holes in the mesh.

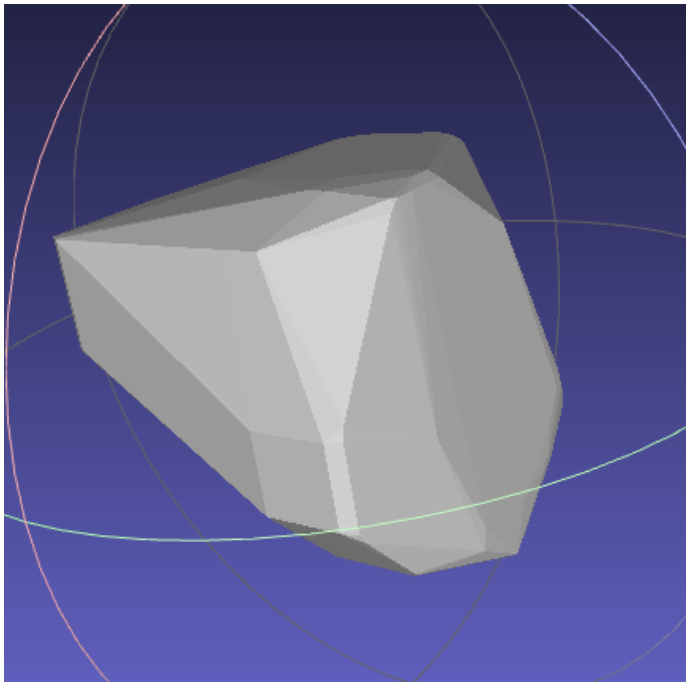


Figure 2.3: Convex hull filters for slider tool

For the two links:

- **Remove Connected Components**, which removes all the connected components of the mesh that can create problems to the cleaning procedure;
- **Close Holes**, which closes all the holes in the mesh;

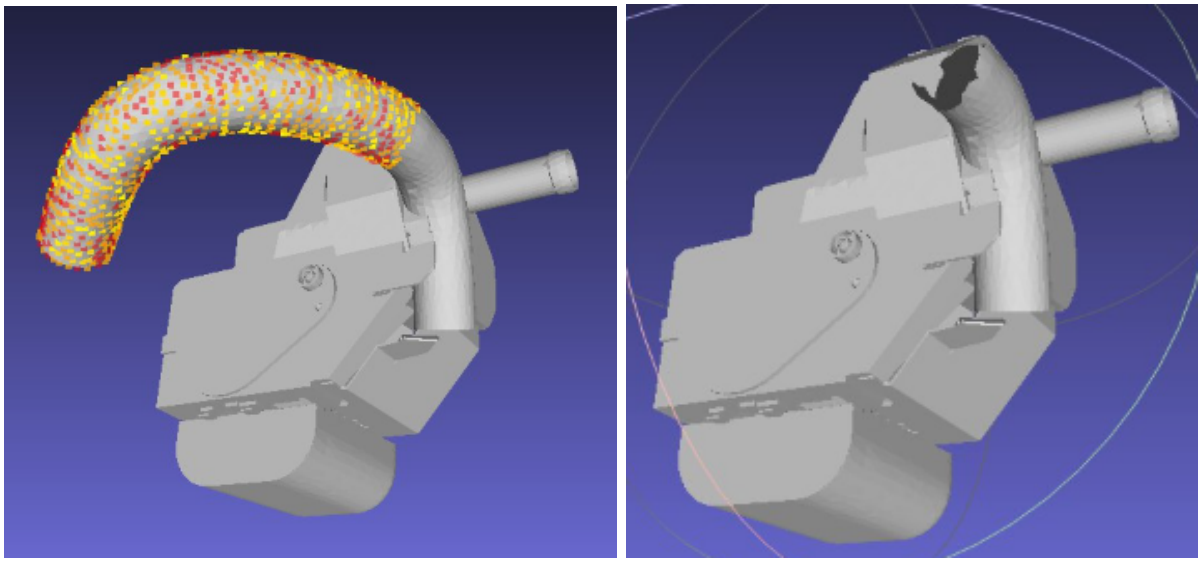


Figure 2.4: Remove first Connected Components for fanuc_m20ia_35m_link_3

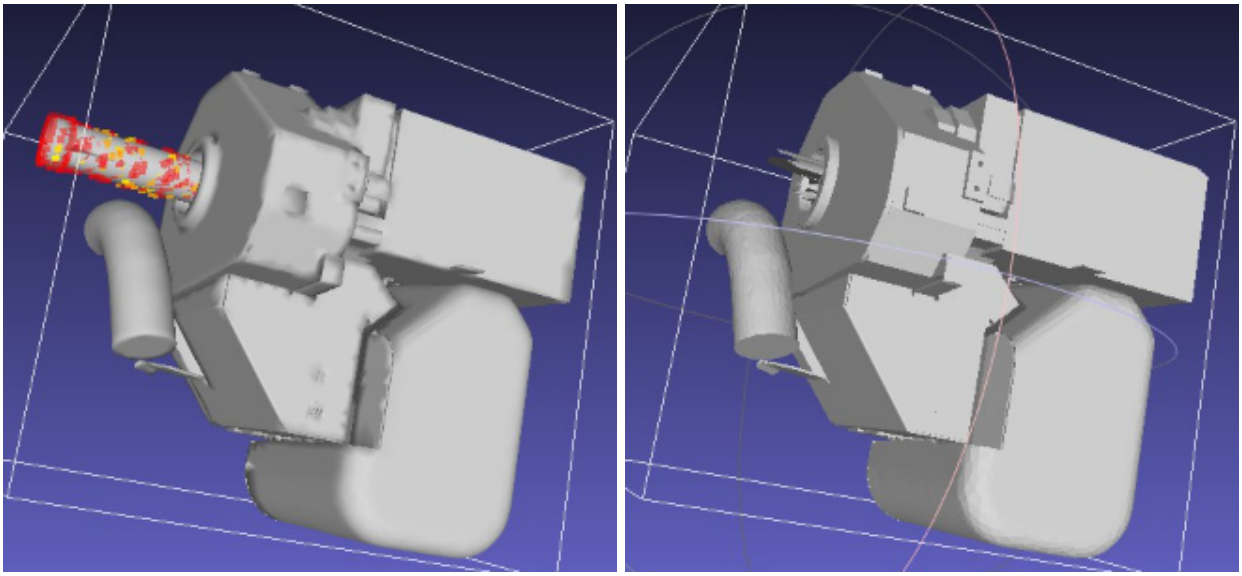


Figure 2.5: Remove second Connected Components for fanuc_m20ia_35m_link_3

2.2.5.2 Center of mass and inertia matrix

After performing this procedure, the volume of each mesh is evaluated by selecting the *Compute Geometric Measures* filter from the *Filters* menu and based on the volume of each mesh the mass of each link is calculated, assuming that all the links are made of the same material and knowing that the total mass of the robot is 252 kg. For the external tools, the mass was obtained by measuring the volume of the meshes in the simulation, and making the same assumption of the other links, which lead to defining a mass

of 20 kg for the end effector and 50 kg for the slider.
After obtaining the mass of each link, it is possible to obtain the center of mass of each link and also the inertia matrix, but their coordinates was referred, as mentioned before, at the frame of the first link of the robot, so to actually be inserted in the URDF, it must be taken into account the DH convention to apply the correct transformations that are required. In order to accomplish this transformation task, a `Python` script named `script_inertia_matrix_and_center_of_mass.py` is produced which takes the center of mass coordinates and the inertia matrix of each link and transforms them according to the relative omogeneous transformation matrix of each link calculated with the DH convention. This script is included in the *supporting_scripts* folder of the project.
At the end of this process the inertial parameters of each link are obtained, and they are inserted in the *inertial.yaml* file. The last think to mention is that the `slider_tool` mesh was elevated from the ground of $0.002m$ to avoid collision with the Gazebo floor.

2.3 Building and Running the Package

Finished the URDF description of the robot, it is necessary to build the package and launch it in order to visualize the robot in RViz. Before doing so, it is necessary to create a *package.xml* file, which is the manifest file for the package, and a *CMakeLists.txt* file, which is the file that contains the instructions for building the package. This will be done for all the packages that will be created in the following chapters, so it is not necessary to go into details about it. The only thing that is important to note is that every time a new package is created, a README file is created as well, which contains the instructions for building and running the package. All these information are contained in each package, so it is raccomanded to read them before building and running the packages. For this package, a launch file in Python is created, which is the file that will be used to launch the robot visualization in RViz, it simply contains the instructions to launch the RViz environment and the robot visualization.
It is important to note that the file launches the following nodes:

- **joint_state_publisher_node**, that simulates the sensors of the robot. It is a GUI that allows to modify the joint values of the robot, so that it is possible to see the robot in different configurations, as shown in Figure 2.7;
- **robot_state_publisher_node**, has the role of transform broadcaster, it publishes the intermediate transformations matrix for each couple of links, It computes forward kinematics in real time;
- **rviz_node**, it is the actual visualization tool, it reads the URDF and the joint states and visualizes the robot in the environment.

Finally, in order to build and use the package, it is necessary to run the following commands in the root of the workspace:

```
colcon build
```

```
source install/setup.bash
```

```
ros2 launch acg_resources_fanuc_m20ia_35m_description display.launch.py
```

From now on, every time it is said that a package is built, it is meant that the commands above are executed in the root of the workspace, just the last one will change depending on the package that is being built and will be specified.

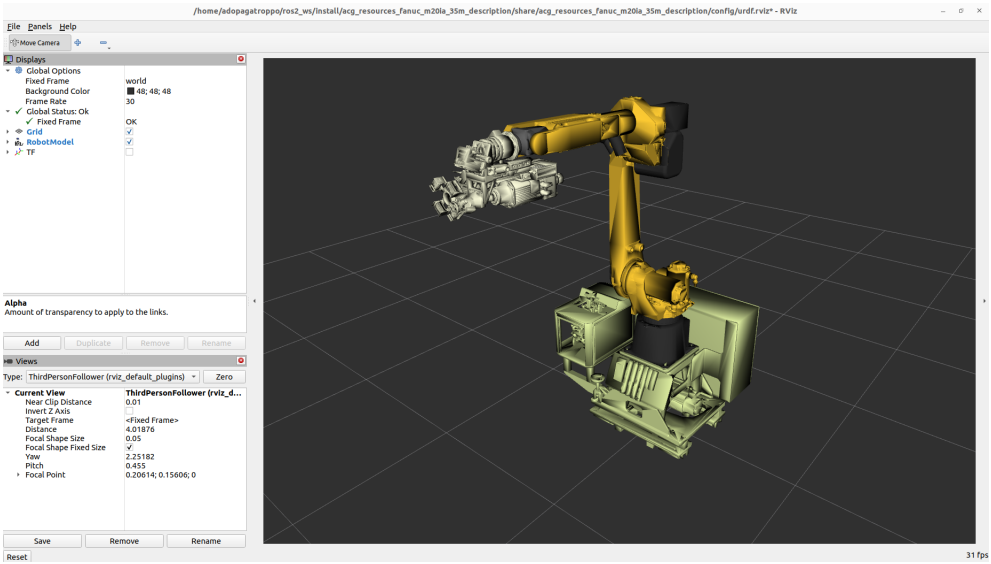


Figure 2.6: Fanuc robot visualization in RViz upon execution of the demo launch file.

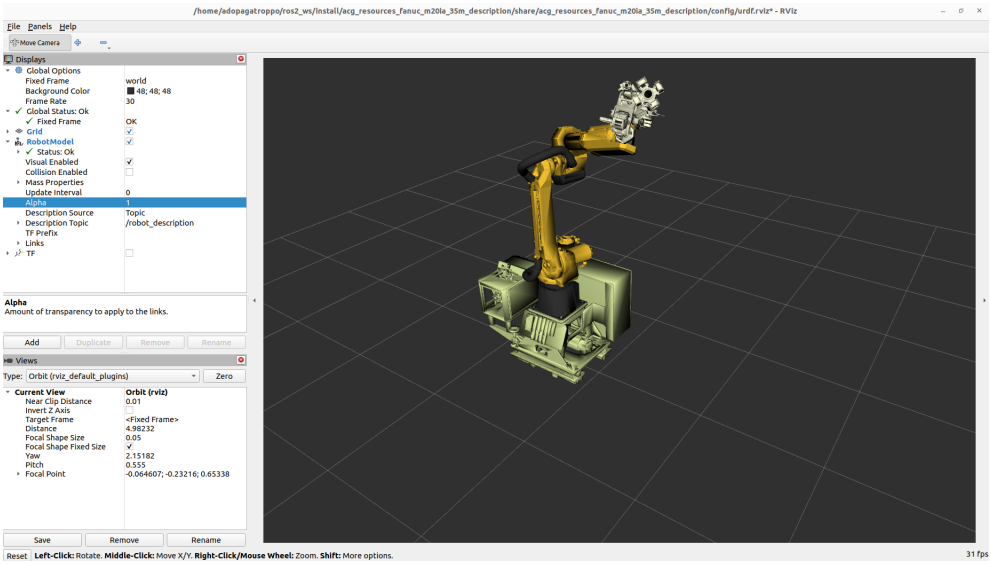


Figure 2.7: Fanuc robot visualization in RViz upon execution of the demo launch file, with joint values modified through joint_state_publisher_gui.

It is also important to note that through the RViz visualization the coordinate frames as defined in the URDF can be seen (there are only a few differences in terms of frames with what was done through the DH convention, but that is simply because those frames are necessary for the robot to be attached to the slider and for the end effector to allow for double usage, as a drilling tool and a fastening tool), as shown in Figure

2.8. This frames were manually positioned using the `Rviz` interface, with the help of the view functionality, so that was possible to fix the frames in the right position.

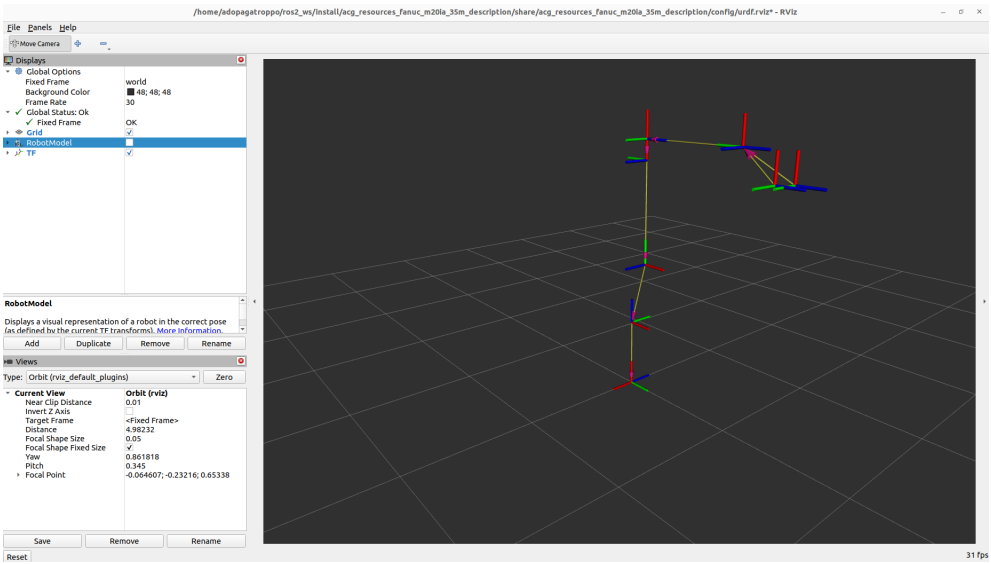


Figure 2.8: Fanuc coordinate frames visualization in RViz.

2.4 Robot Semantic Package

2.4.1 Running the Setup Assistant

After working on the robot's kinematics description, it is necessary to define fundamental elements for the upcoming WPs, as for planning purposes it is fundamental to define the kinematic solver to use, the kinematic chains and some planning configurations. In order to obtain the semantic information regarding the robot, MoveIt!2's Setup Assistant is used. To be more specific, after running

```
ros2 launch moveit_setup_assistant setup_assistant.launch.py
```

it is possible to access the software's features to prepare the MoveIt configuration package for the Fanuc robot starting from the URDF file. After clicking on *Generate Collision Matrix*, the collisions between the different links can be disabled. To be more specific:

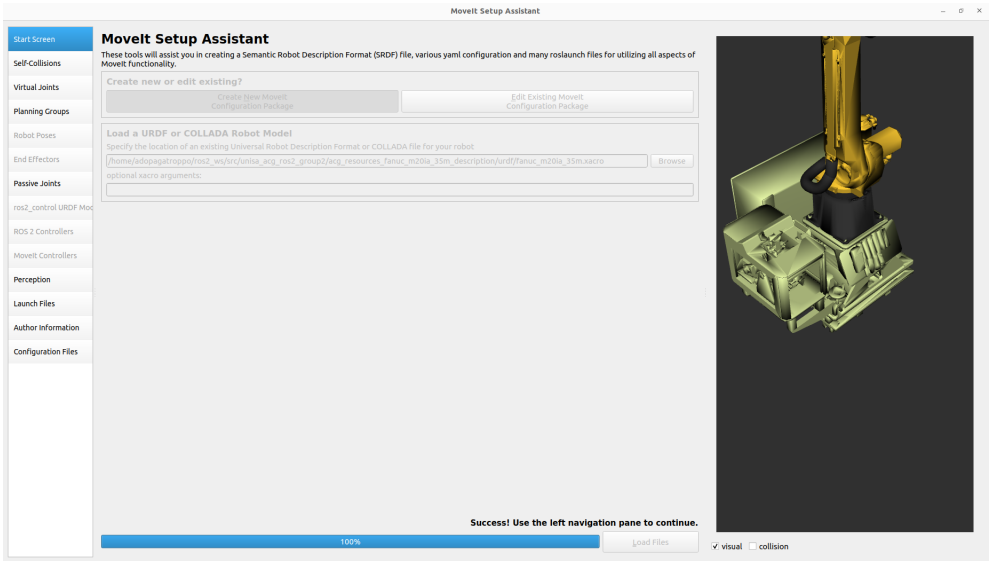


Figure 2.9: Setup Assistant's GUI after loading the Fanuc robot model.

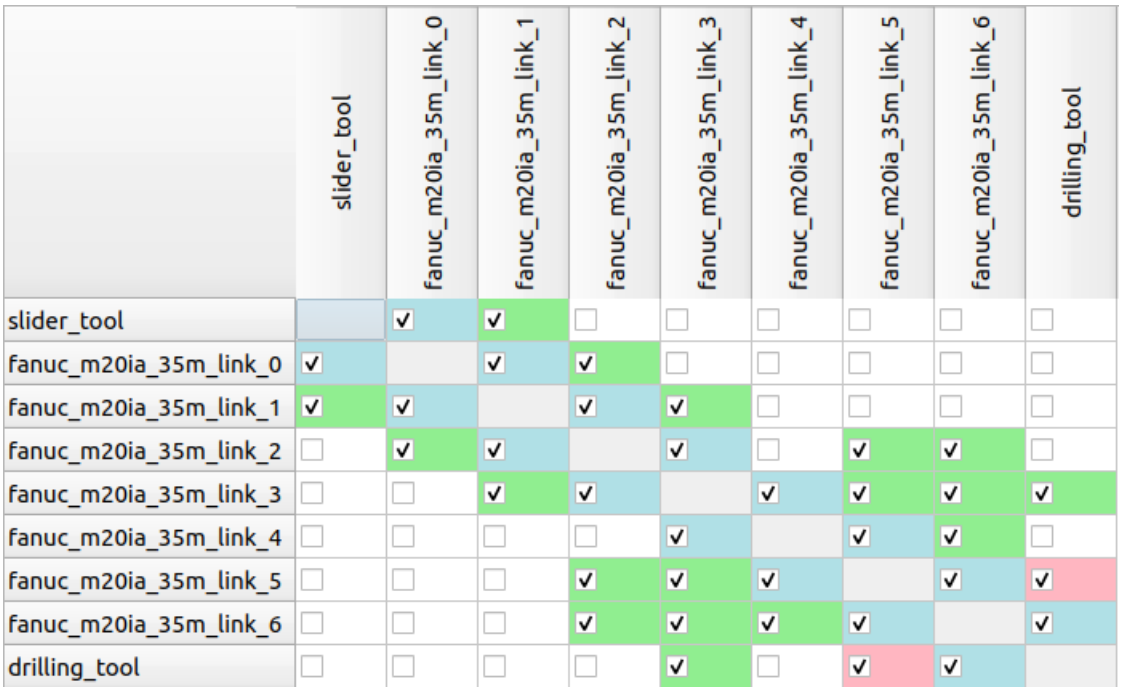


Figure 2.10: Collision Matrix GUI.

Where in sky blue are the links that are adjacent to each other, so they always collide and must be disabled, while in green are the links that never collides because of the structure of the robot, so they can be disabled too. In pink are the links that are not adjacent to each other, but they always collide, so they must be disabled too. This is the case of the `drilling_tool` and the `fanuc_m20ia_35m_link_5` because the end effector tool is fixed to the `fanuc_m20ia_35m_link_6` and, given that this link is tiny, the `drilling_tool` and the `fanuc_m20ia_35m_link_5` are strictly adjacent to each other. Finally the white cells represent links where the collision must be enabled. Please note that in this case just the `drilling_tool` appears in the list, because just one collision mesh was used for both the two tools of the end effector, but the same reasoning applies for the `fastening_tool` too. This collision disabling step must be taken in order

to help the kinematic solver that will be used to deal with the high computational weight introduced by the calculations that take into account collisions for each link, as disabling collisions, especially those that do not affect the correctness of the behaviour of the robot, lightens the calculations.
Next, virtual joints must be specified. Since all the needed elements to link the robot to the environment have been already defined in the URDF file (this will also help with the dynamic simulation in Ignition Gazebo), no virtual joints are added.
For the next step, planning groups must be defined. Here two planning groups are defined, as the end effector has two tools that can be used, thus they can both require planning, the choice to not define two other planning groups for the robot without the slider is driven by the fact that the robot without the slider is greatly restricted in its movements due to how cumbersome the end effector tool is and how limited the set of possibly executable parabolas would be.

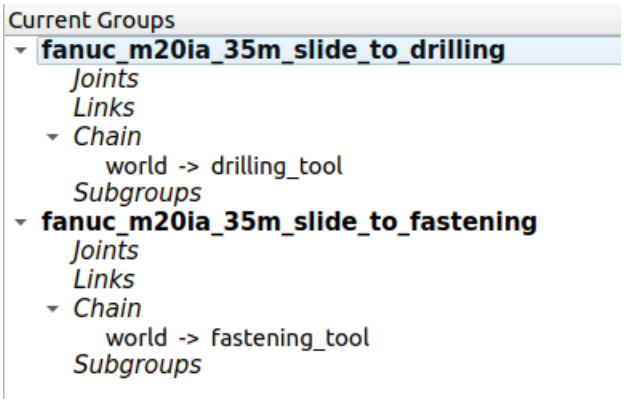


Figure 2.11: Planning Groups GUI.

For both of them the kinematic solver specified is the default one, which is the KDL kinematic solver. This is the default one of ROS2 and it is chosen because it is the most used one and it is the most reliable one. It is an iterative solver based on gradient descent, that, at each iteration, tries to minimize the difference between the desired pose and the current pose of the robot. If the robot is redundant, as in this case, the solver will run a different algorithm that can be described by the following equation:

$$q(k+1) = q(k) + \beta J^{+} e(k) + \alpha(I - J^{+}J)\nabla g(k) \tag{2.3}$$

where $e(k)$ is the error mentioned before, $q(k)$ is the current joint position, J^{+} is the pseudo-inverse of the Jacobian matrix, $\nabla g(k)$ is the gradient of the cost function, α and β are the learning rates, and I is the identity matrix. Normally, in the default setup of KDL, the learning rates and the cost function are set to default values, in particular the cost function is set to

$$\begin{aligned} &min_{\dot{q}} \;\; g(q) = \tfrac{1}{2}\dot{q}^{T} W \dot{q} \\ &s.t.\; v = J\dot{q} \end{aligned} \tag{2.4}$$

so it tries to minimize the quad norm of the joint velocities. After defining the planning groups and their kinematics solvers, relevant robot poses can be defined. In the case of the Fanuc robot, 2 poses have been designed:

- a home position for the *fanuc_m20ia_35m_slide_to_drilling* group called *fanuc_m20ia_35m_home_slide_to_drilling*, which has the values on all joints set to 0;
- a home position for the *fanuc_m20ia_35m_slide_to_fastening* group called *fanuc_m20ia_35m_home_slide_to_fastening*, which has the values on all joints set to 0.

These home positions have been designed for the sole purpose of giving the Fanuc robot a pose to start planning with and a pose to return to after executing the given trajectory.
Now end effectors must be declared. Since the end effector has 2 working modes (as a fastening tool and as a drilling tool), 2 end effectors will be defined:

- *drilling_end_effector*, with parent link *fanuc_m20ia_35m_link_6* and belonging to the *fanuc_m20ia_35m_slide_to_drilling* planning group;
- *fastening_end_effector*, with parent link *fanuc_m20ia_35m_link_6* and belonging to the *fanuc_m20ia_35m_slide_to_fastening* planning group.

After defining the end effectors, passive joints must be specified, however, since no joints among those of the Fanuc robot and that are not fixed require no actuation, thus they must not be taken into account for planning, no joints are specified here.
Regarding the ros2_control URDF Modification step, all joints undergo a modification procedure to allow for the creation of a fanuc_m20ia_35m.ros2_control.xacro file which will define the command and state interfaces of each joint that will be taken into account while working with the ros2_control framework. To be more specific, every joint is given a position command interface (to allow for position control) and velocity and position state interfaces.

Figure 2.12: Command Interfaces GUI.

Of course this is also customizable, so it is possible to add other interfaces, like the effort command interface, or to remove the position command interface and add the velocity command interface and so on. The choice to add the position command interface lies in the fact that, at the first stage of the project, the robot will be controlled in position, so it is necessary to have the position command interface to actually control the robot.
After defining the interfaces of the joints, the ros2_control controllers that will be used are defined, and in the case of the Fanuc robot a JointTrajectoryController named *fanuc_m20ia_35m_slide_to_end_effector_controller* is defined. The JointTrajectoryController class is chosen to allow for the execution of joint-space trajectories in the joint space with position control.
Regarding the MoveIt controllers definition step, the controllers to be used by the MoveIt Controller Manager are automatically generated, with the same names as the ros2_control controller, specifying

FollowJointTrajectory as the controller type.

Since there are no infos about Fanuc's sensors, no 3D sensors are specified in the next step.

Finally, all the suggested launch files are selected and generated, then the package folder is created.

After the entire development procedure, it is possible to observe the *acg_resources_fanuc_m20ia_35m_moveit_config* configuration package in its entirety, with all the required dependencies as defined in the *CMakeLists.txt* and *package.xml* files automatically generated by the MoveIt! Setup Assistant. This means that it also possible to build the project and run:

```
ros2 launch acg_resources_fanuc_m20ia_35m_moveit_config demo.launch.py
```

Note that the *demo.launch.py* launch file was automatically generated by the Setup Assistant. After running the demo launch file, the content of Figure 2.13 is shown and it can be used to interact with the robot, plan movements and execute them with the help of the MoveIt framework.

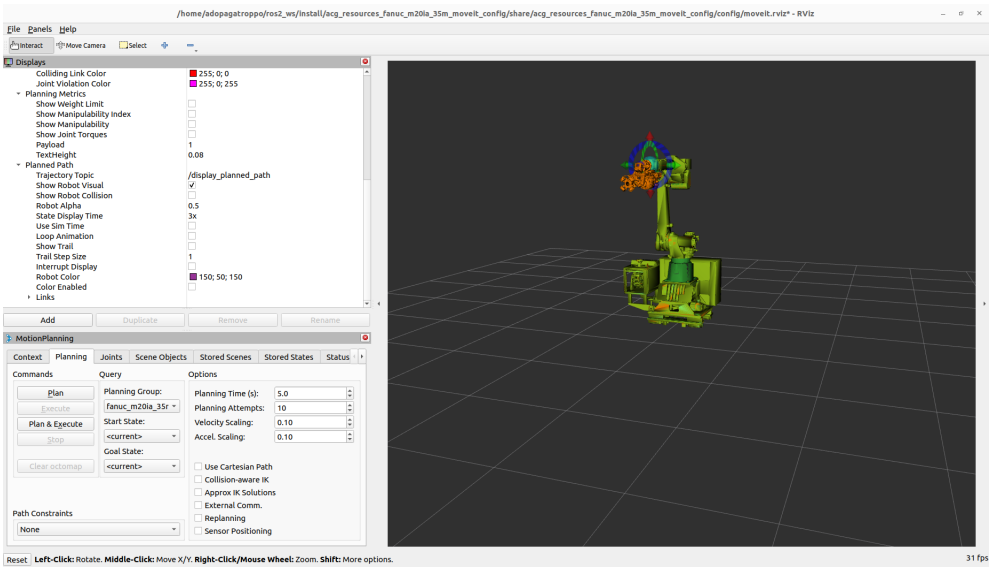


Figure 2.13: Fanuc visualization with interactive gizmo after running the configuration package demo launch file.

With the help of an interactive gizmo, it is possible to move the robot's end effector and plan and execute movements. Then it is possible to visualize the planned points of the trajectory using the `Trajectory Slider` of the `MotionPlanning` tab.

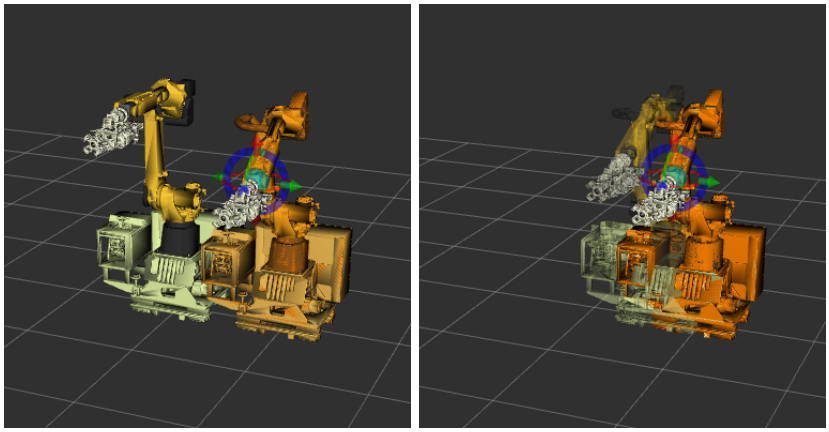


Figure 2.14: MotionPlanning plugin to plan and execute a trajectory

Note that, in this case, no dynamic simulation is shown, as the robot is not yet connected to the Gazebo environment and does not have an effort command interface, but this will be done in the next WPs.

CHAPTER 3

WP2-PLANNING SYSTEM

In this chapter the planning system as required is developed, including a trajectory generation module, the trajectory planning for the Fanuc robot, and the trajectory execution simulation in RViz. Note that sometimes the term ”trajectory” is used to refer to the path without the time component, and sometimes it is used to refer to the path with the time component. The context should make it clear which one is being referred to, but to avoid confusion, every time the term ”trajectory” is used without taking into account the planning phase or the execution phase, it refers to the path without the time component.

3.1 Trajectory Generation module

Before starting to develop the trajectory generation module, it is necessary to define the architecture of the module itself. In particular, the trajectory generation module is composed by three main parts:

- A MATLAB module that is used to generate the trajectory and to store it in a bagfile
- A trajectory server that is used to call the MATLAB module and to check if the trajectory generation was successful or not
- A trajectory planning node that is used to plan the trajectory for the Fanuc robot and to execute it in simulation

This architecture is represented in the following figure:

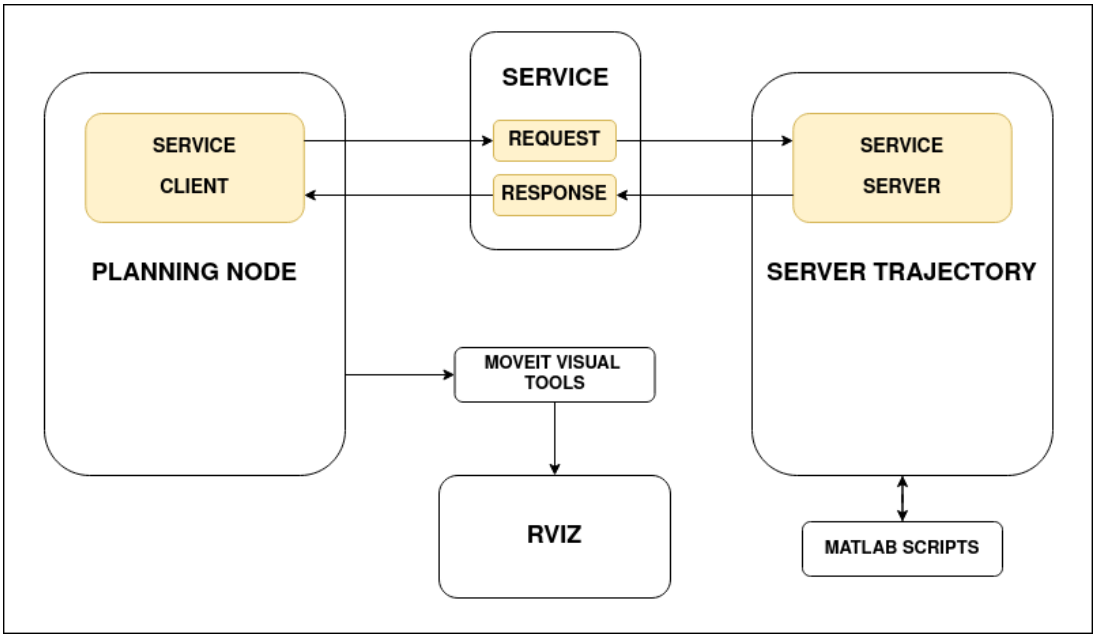


Figure 3.1: Architecture of the trajectory generation module

Note that the MATLAB module is not a ROS node, so it is not possible to call it directly from the trajectory server as a command. This design was necessary doing that because the desired behaviour was to call a single launch file that starts all the trajectory generation module and, by making the MATLAB module a ROS node, that would not have been possible, because it was necessary to start the MATLAB module from the MATLAB environment. This choice reduces the modularity of the system, but it was necessary to grant the desired behaviour.

3.1.1 Tackling the Mathematical Challenge

The goal of the project is to make the Fanuc robot follow a given trajectory, which in this case is a parabolic path with length $\Lambda \geq 3.14m$. While designing a parabolic path with specific length seems easy at first, mathematically speaking a difficult problem must be solved. To be more specific, given the length Λ of the parabolic path from one (x_i, y_i) point to another (x_f, y_f), a linear integral of the following type must be solved (the indefinite version is given for generality purposes):

$$\Lambda = \int \sqrt{1 + (2ax + b)^2} dx = \frac{\sqrt{1 + (2ax + b)^2}(2ax + b) + \sinh^{-1}(2ax + b)}{4a} + constant \tag{3.1}$$

Given the generic formula of a 2-dimensional parabola, which is $y = ax^2 + bx + c$, with a, b and c being the degrees of freedom of the parabola, it is relevant for the problem of trajectory generation to find out the specific values for the coefficients given 2 points and the plane on which the parabola will be generated. For the purpose of the project, it is assumed that the user who interacts with the system is willing to give the coordinates for the points through which the parabola will pass (which will be concretely done through a graphical interface which will help with establishing these coordinates), so for the rest of the paragraph the points will be referred to as $p_i = (x_i, y_i)$ and $p_f = (x_f, y_f)$. Given the points, the problem of finding

out the coefficients of the parabolic path translates to a system of 3 equations with the unknown variables being the a, b and c coefficients:

$$\begin{cases} y_i = ax_i^2 + bx_i + c \\ y_f = ax_f^2 + bx_f + c \\ \Lambda = \frac{\sqrt{1+(2ax_f+b)^2}(2ax_f+b)+\sinh^{-1}(2ax_f+b)}{4a} - \frac{\sqrt{1+(2ax_i+b)^2}(2ax_i+b)+\sinh^{-1}(2ax_i+b)}{4a} \end{cases} \tag{3.2}$$

Note that given two points and a plane, infinite parabolas can be generated, so there are some degrees of freedom that must be removed from the problem. The first degree of freedom is the nature of the parabola, which can be simmetric or not, and this is removed by imposing the condition that the parabola must be simmetric with respect to the middle point between the two given points, which is $p_m = (\frac{x_i+x_f}{2}, \frac{y_i+y_f}{2})$, or more specifically simmetric with respect to the line that passes through p_m and is perpendicular to the line that passes through p_i and p_f. The second degree of freedom is the concavity of the parabola that can be chosen by the user through the graphical interface. More information about the graphical interface and the approach chosen will be given in the next section. Solving this equation system allows for the definition of the coefficients and the completion of the formula of the parabolic path that will traverse the user-desired points. This type of problem can be tackled by equation system solvers that exist among the libraries of numerous programming languages, however the Python library for symbolic calculus SymPy fails in finding a solution to this specific system (even by making use of the equivalence $sinh^{-1}(x) = ln(x+\sqrt{x^2+1})$), thus MATLAB is used in place of Python (no other softwares or programming languages are taken into account as MATLAB outperforms the rest in terms of symbolic calculus and it has a built-in compatibility with the ROS2 environment).

3.1.2 The MATLAB module

After working out the mathematical definition of the problem, it is possible to start working on a script (or a series of scripts) that can deal with it while also adding the required features for the trajectory generation. A list of the scripts that are used to generate the trajectory is made below, with a brief description of their content and their role in the trajectory generation process. The main focus will be on design choices and how different problems are tackled.

3.1.2.1 Trajectory_script.m

This is the main script that is used to generate the trajectory, given the name of the bagfile in which the trajectory will be saved it initializes some parameters, fixed for the project, and then it calls the *generate_path.m* script, which is the one that actually generates the trajectory. This file also contains logic to check whether the trajectory generation was successful or not, so it returns a boolean value to the server that calls it, that will be discussed in 3.1.3.

3.1.2.2 Generate_path.m

Given the parameters initialized in the previously mentioned script, this script is the one that actually manage the trajectory generation by calling all the other scripts. In order to do so, it calls the *GUI_functions.m* script, which is the one that allows the user to define the trajectory through a graphical interface, and then it calls the *readDataFromFile.m* script, which is the one that reads the file that contains the infos about the trajectory that the user defined through the GUI. After this, the script calls the *parabola.m* script, which is the one that actually solves the equation system 3.2 and returns the points that define the parabolic path. Another script is called that is used to plot the points on a 3-dimensional environment and ask the user whether the parabolic path is the one that they wanted or if it needs to be inverted in terms of concavity: *plot3D_function_from_points.m*. When the parabola is defined, the scripts do some checks on the trajectory to see if it is feasible or not, in terms of the action space of the robot. To understand what the action space of the robot is, it is necessary to explain what assumptions are made about the robot's movement. In particular, after defining the robot description package, with the help of RViz and the MoveIt! plugin, it is possible to see that the robot has a limited action space, which is the space in which the robot can move without colliding with itself and with the environment, and limited by the joint limits. This space was "mapped" by adding to the scene spheres that represent the maximum distance that the robot can reach. In particular, with the help of joint state publishers, it is possible to see what's the maximum distance that the robot can reach with the end effector, without considering the slider:

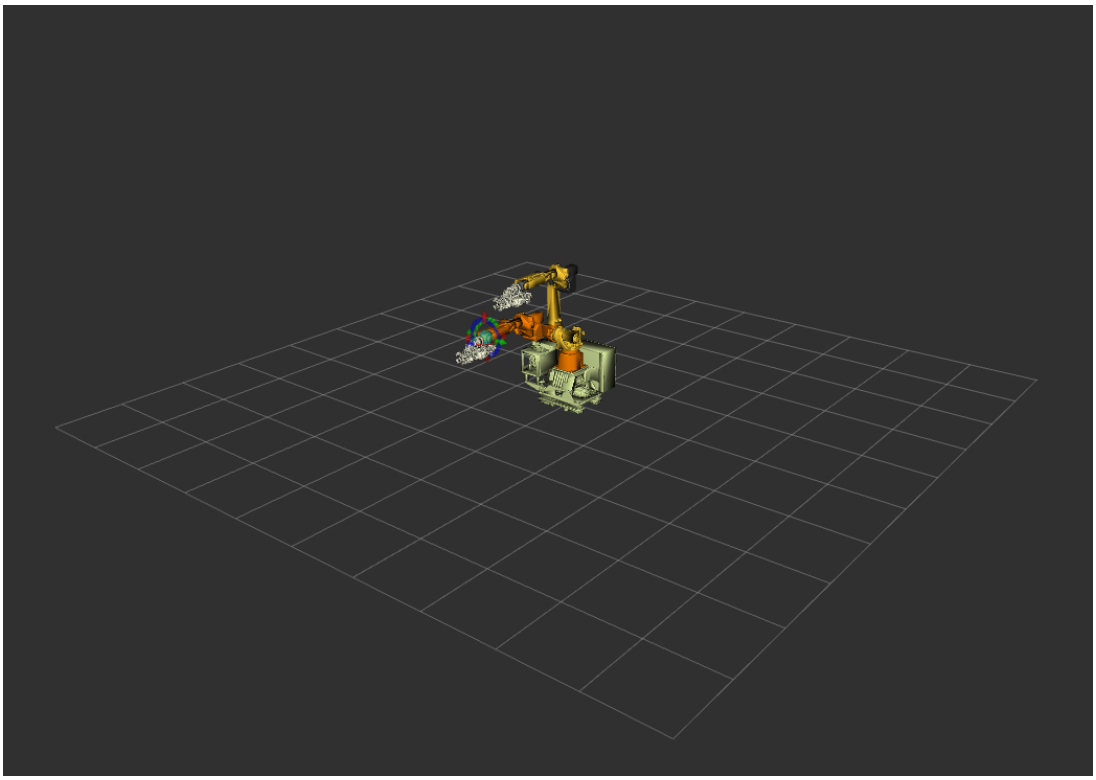


Figure 3.2: Action space of the robot without sliding

After defining the action space of the robot, it is possible to add to the scene a sphere that represent the maximum distance that the robot can reach:

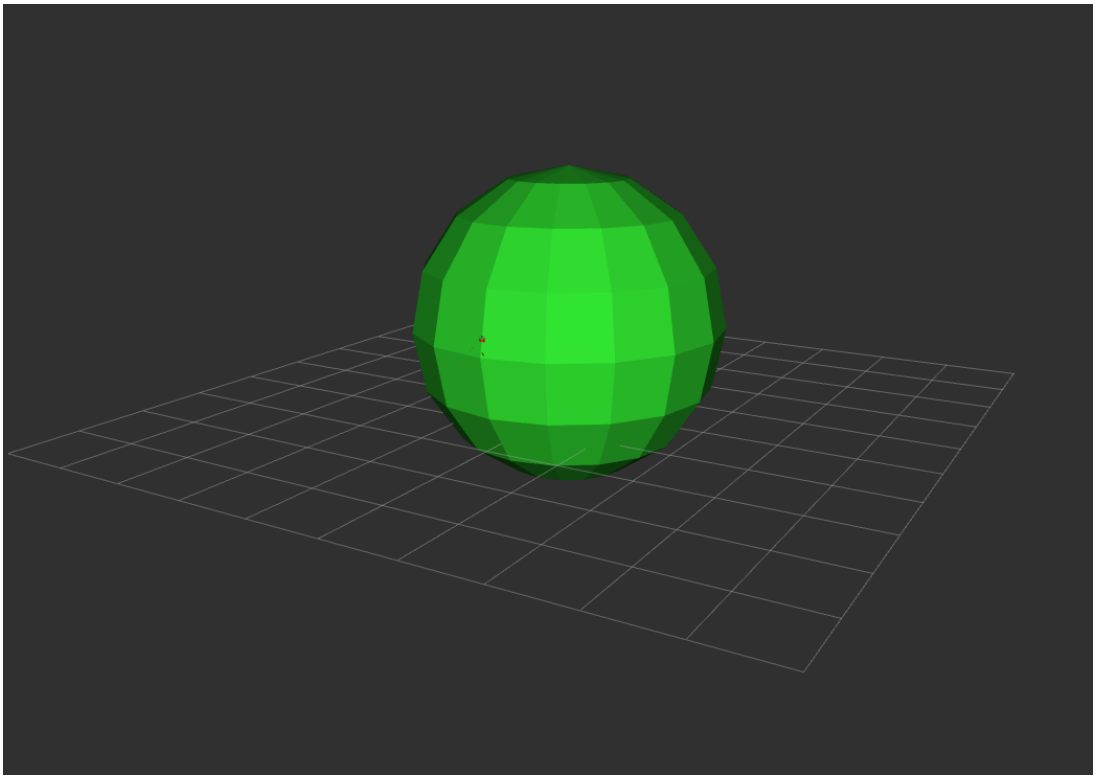


Figure 3.3: Sphere that represents the maximum distance that the robot can reach without the slider

Taking the slider into account, it is possible to see that the action space of the robot is extended, but it is still limited in terms of two directions, so it can be represented by a group of spheres that starting from the previous sphere, moves in the direction of the slider:

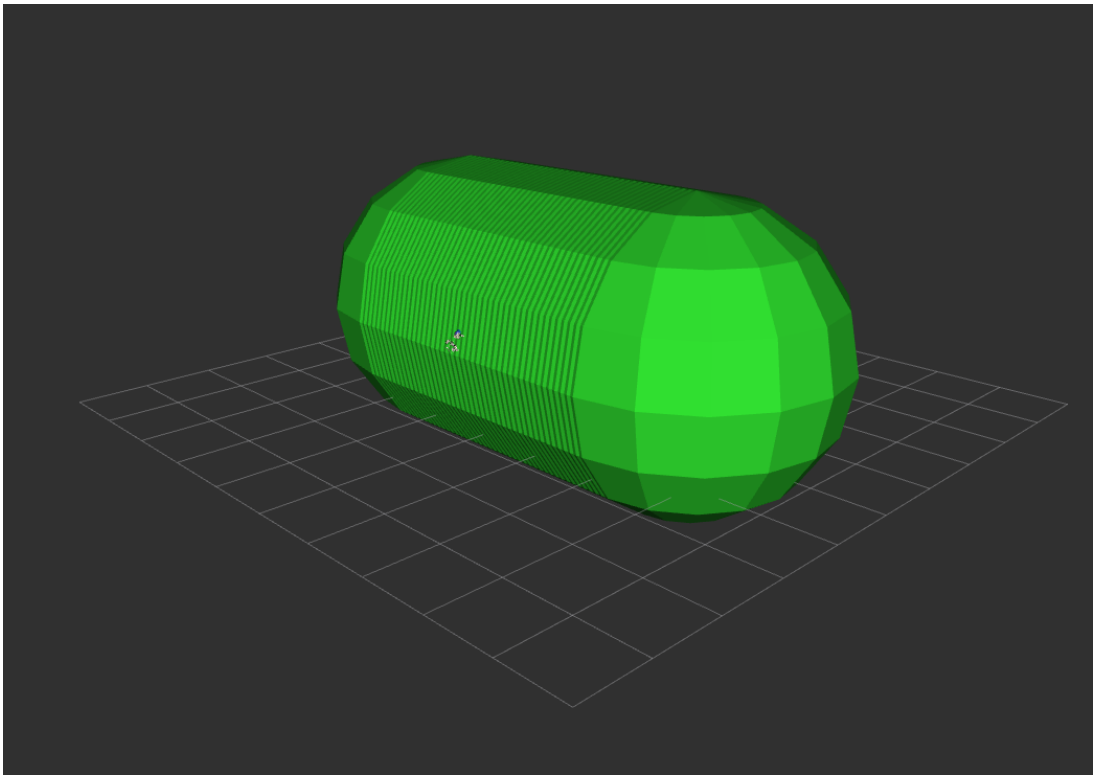


Figure 3.4: Spheres that represents the maximum distance that the robot can reach

Based on this representation, a simplification is made, which is that the action space of the robot is a

cylinder with a radius of 2.1 meters and a height of 8.4 meters centred in the point and rotated of 90 degrees around the y axis:

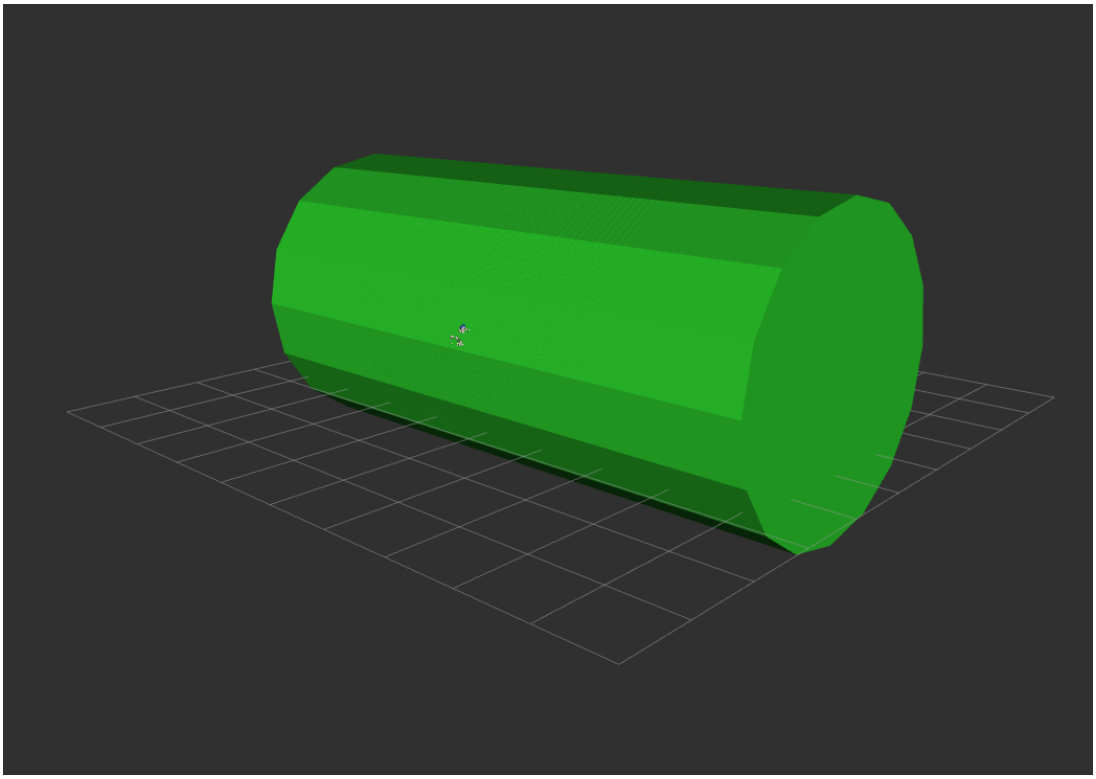


Figure 3.5: Cilynder that represents the action space of the robot

So, after defining the action space of the robot, the script checks if the trajectory is feasible or not by checking if the points that define the parabolic path are inside the action space of the robot. it is important to note that this check is useful to avoid problems with the trajectory execution, but it is still possible that the user defined parabola is not feasible in terms of agility of the robot, so it is still possible that the robot will not be able to follow the trajectory. This is because there is a difference between the reachable space of the robot and the task-specific reachable workspace. The first one refers to the area in three dimensional space that a robot can effectively access with its end effector, taking into account the limits of its joints and environmental constraints. It represents all the positions that the robot can physically reach with its arm, considering factors such as joint lengths and obstacles in the path. The other one, instead, refers to the subset of the reachable workspace where the robot can successfully perform a specific task. This may include additional constraints such as precision requirements, end effector orientation, etc... Given this, one assumption is made, that will be exploited in the next chapters, which is that the robot real work space should be in front of the robot, so that it is capable of reaching all the points that are in front of it. This assumption is made because the robot is used to perform tasks that are in front of it, so it is not necessary to define a trajectory that goes behind the robot, for example. This assumption is useful to simplify the trajectory generation process, but it is important to note that this is not limiting the robot in terms of the tasks that it can perform. Other information about this assumption will be given in the next chapters when the planning and execution of the trajectory will be discussed. Finally the script store the parabola points

in a bagfile, with the name defined in the parameters received.

3.1.2.3 GUI_functions.m

This script is the one that allows the user to define the trajectory through a graphical interface. Note that Matlab environment is not the best one to develop a graphical interface, but it is the only one that, as said before, was capable of solving the equation system 3.2, so the interface appears to be a bit ”rough” and not very fast to use. That's because, to plot the plane, for example, it is necessary to plot a lot of points, and at each iteration the script must plot all the points rotated again and again, so it is not very fast. A tip for this GUI is that, instead of using slider as real sliders, it is better to click on the slider position in order to reduce the number of plots that MATLAB computes, because it is very slow. Despite this, the interface is still usable and it is possible to define the trajectory without problems. It appears as follows:

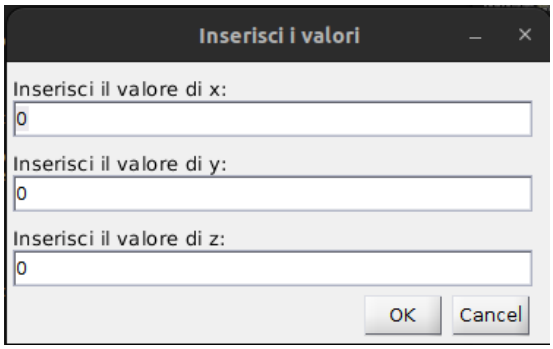


Figure 3.6: MATLAB script GUI: choice of the first point

A common initial point is: $x_i = 0.0064$, $y_i = 1.4056$, $z_i = 1.80093$ that are the coordinates of the drilling tool in the starting configuration, and the same for the fastening tool: $x_i = 0.12044$, $y_i = 1.3476$, $z_i = 1.8134$ and refers to the world frame. At the start the user can select the first point of the parabola by inserting the coordinates in the text boxes and then pressing the ”OK” button. After that, a 3D plot of the plane is shown, with the first point plotted in red and the final one in green (the final point is not defined yet, so it is plotted in the same position of the first one):

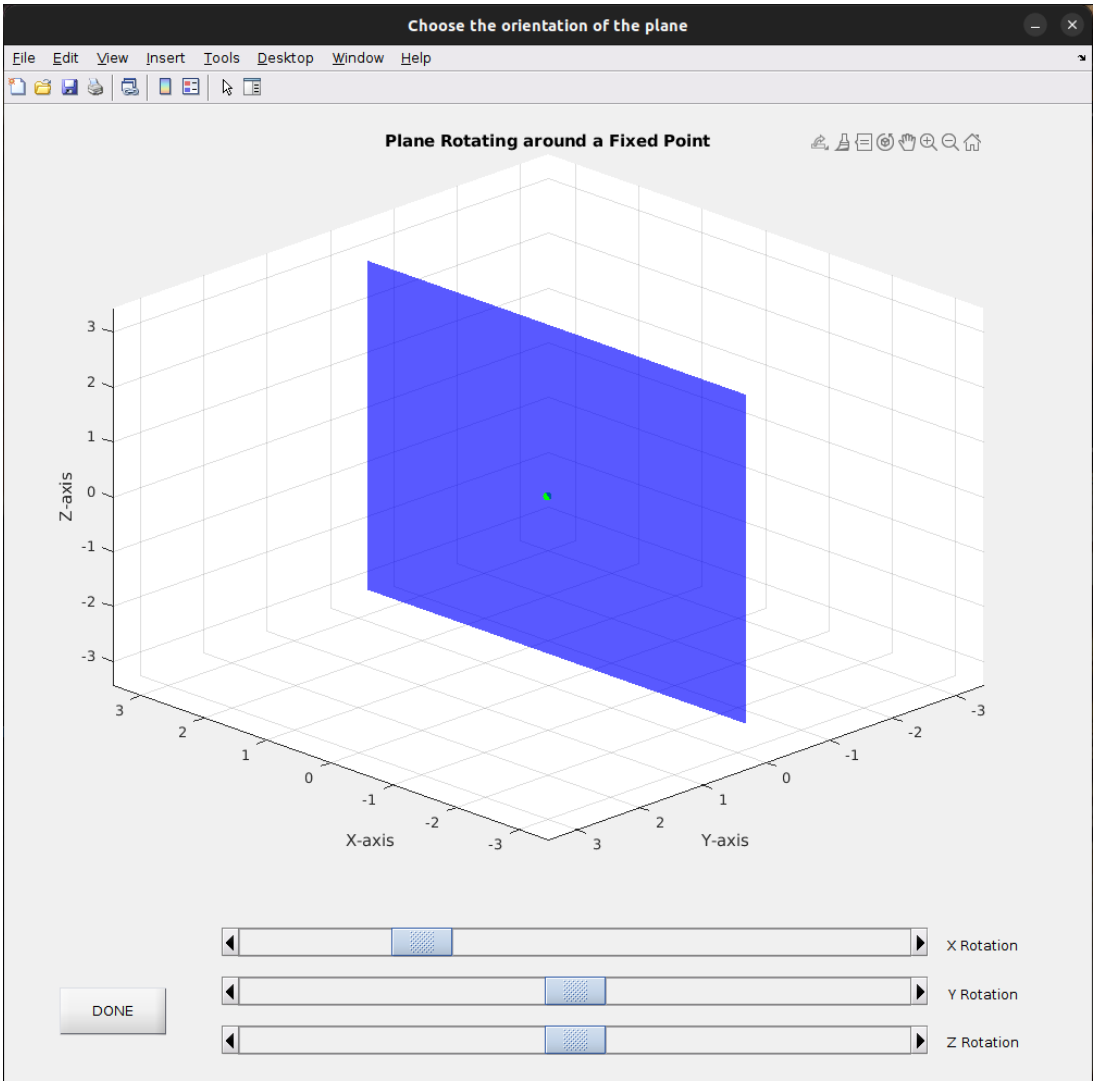


Figure 3.7: MATLAB script GUI: choice of the plane orientation

It is important to note that the axis of the 3D space matches the world frame in Rviz. Note that the plane that appears is concordant with the ortogonal plane to the end effector of the robot at the starting position made up of the x and y axis of the end effector frame, or, referring to the world frame, made up of the x and z axis as seen in figure. The user has to imagine that the end effector is pointing in the direction of the positive y axis, and based on this, rotate the plane in the desired position. A useful tip is to visualize the mapping between this two systems in figure 3.8.

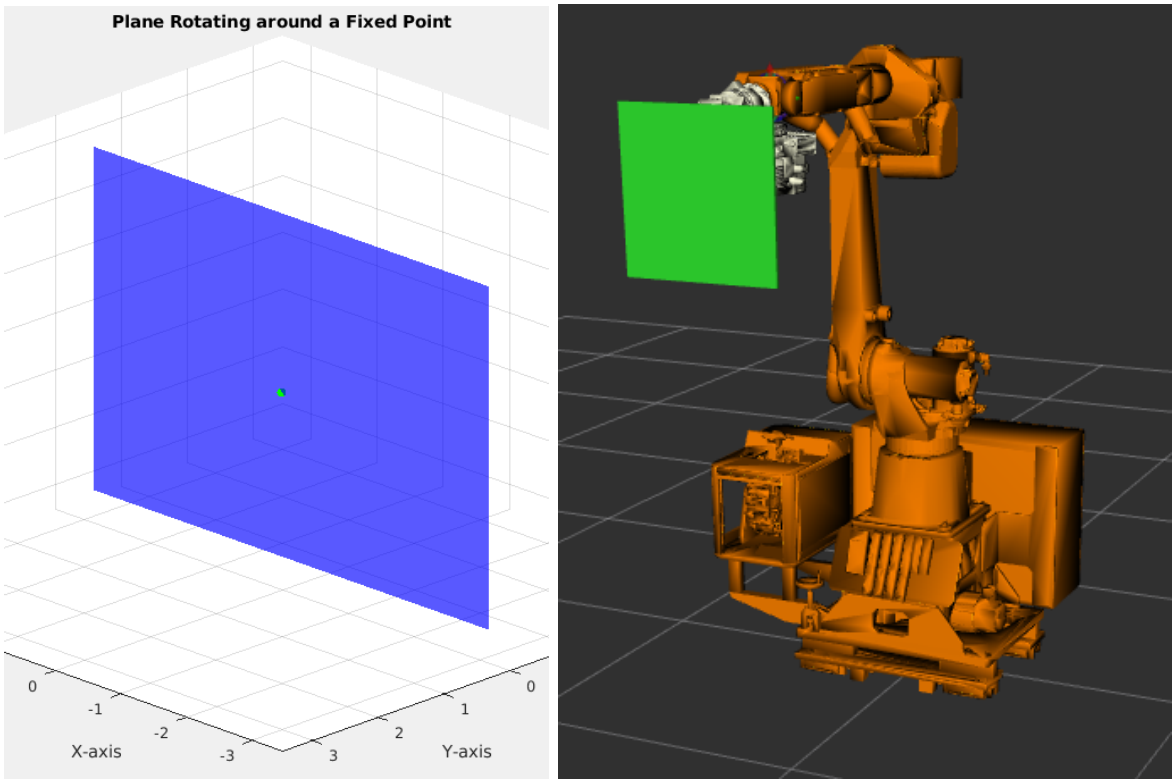


Figure 3.8: MATLAB script GUI: compare between Rviz and MATLAB visualizations

The comparison figure shows the mapping between what the user sees in the MATLAB-generated GUI and the robot's orientation and position: the blue plane in the MATLAB GUI corresponds to the green plane displayed in RViz, thus the user must take into account this mapping while designing the desired trajectory. If the starting point chosen by the user at the first step of the trajectory generation is different than the default one, the user must imagine that the green plane in RViz is moved elsewhere in the 3D environment, and same goes for its orientation. So the user can rotate the plane by using the relative sliders.

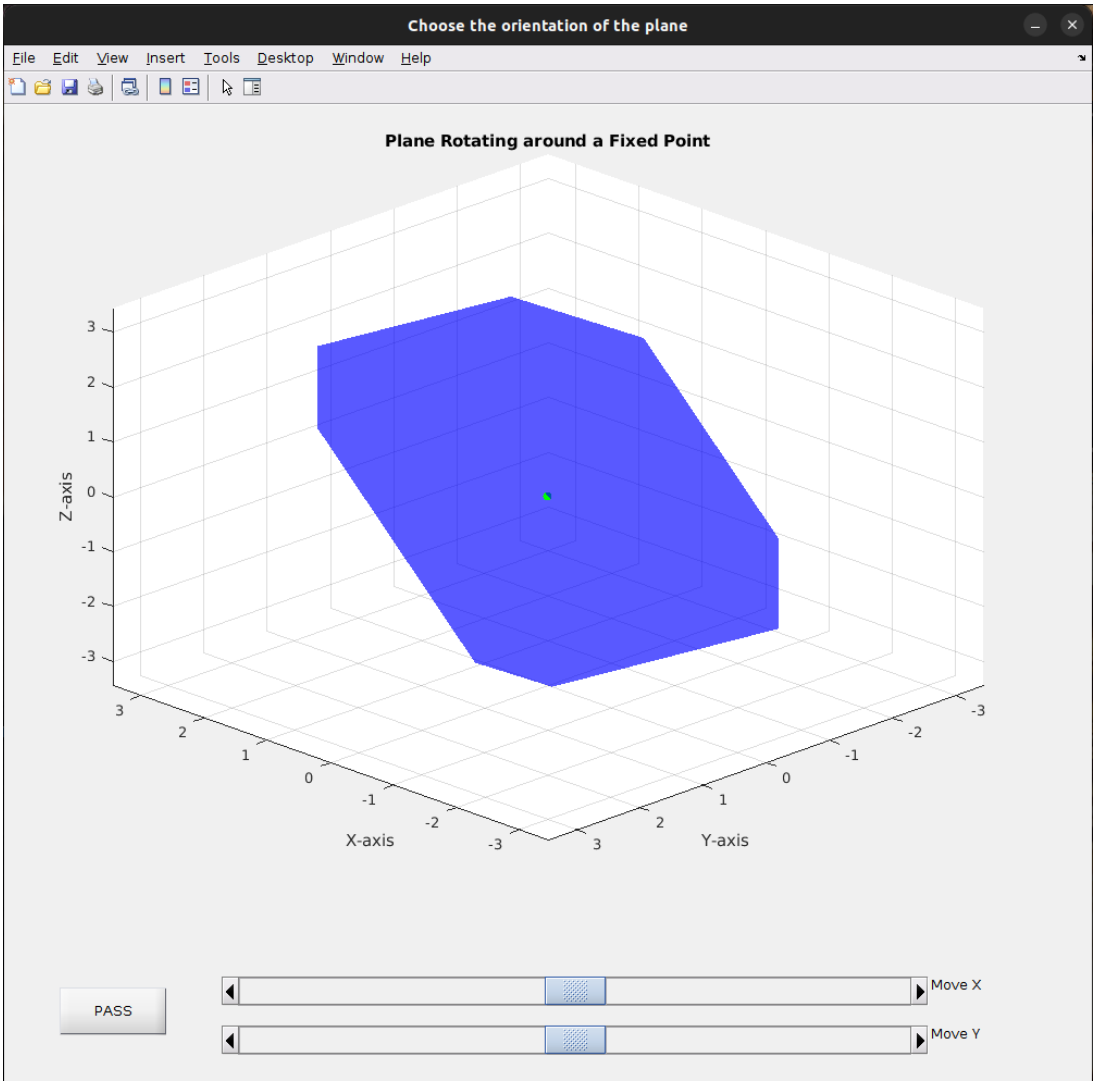


Figure 3.9: MATLAB script GUI: selecting the plane orientation

This mechanism is not very intuitive, but it is the only one that was possible to implement in the MATLAB environment. After that, the user can click on the ”DONE” button to confirm the orientation of the plane and to define the final point of the parabola:

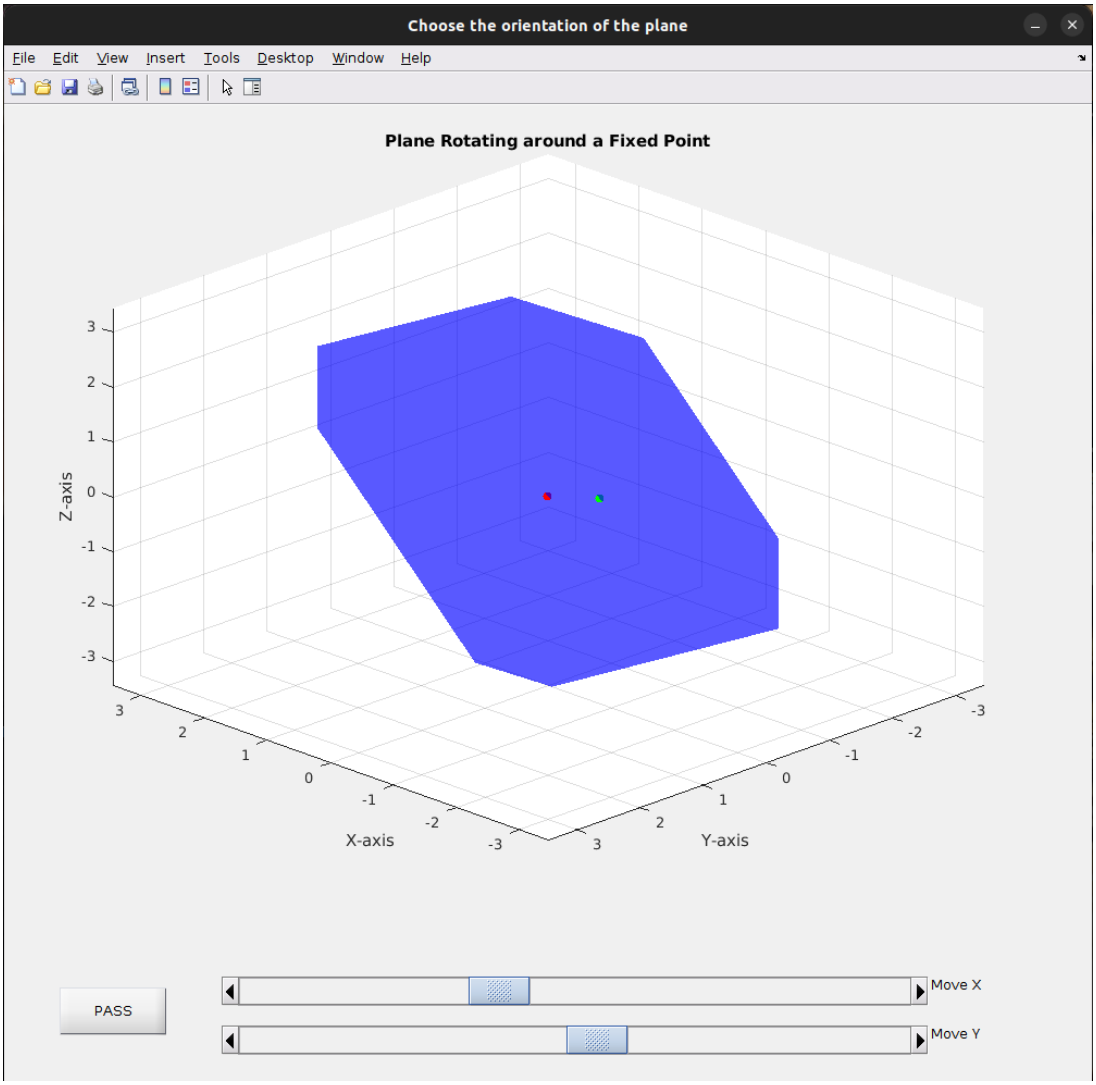


Figure 3.10: MATLAB script GUI: choice of the last point

After choosing the final point, the user can click on the ”PASS” button to confirm the parabola and to visualize it in the 3D plot:

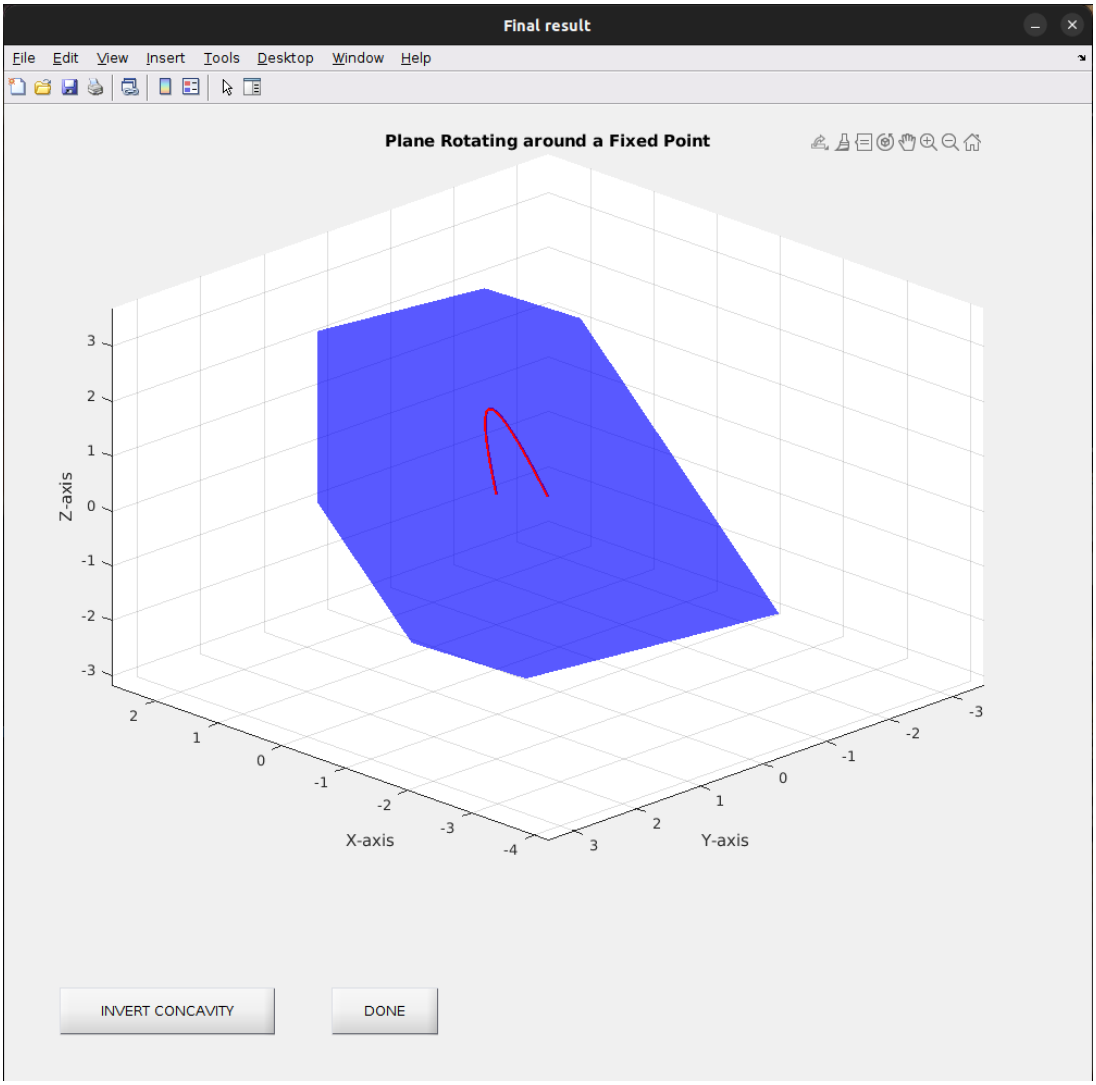


Figure 3.11: MATLAB script GUI: visualization of the parabola

Finally the user can confirm the parabola, or change the concavity of it and the script saves all the infos in a file that will be read by the next script

3.1.2.4 readDataFromFile.m

This script is the one that reads the file that contains the infos about the trajectory that the user defined through the GUI, and it returns the following values:

- the starting position of the trajectory
- the final position of the trajectory
- the rotation matrix that identifies the rotation of the plane from the base coordinate frame of the Fanuc robot
- the translation vector to define the origin point of the coordinate frame of the new coordinate frame

3.1.2.5 parabola.m

This is the script that actually solves the equation system 3.2 and returns the points that define the parabolic path. Before solving the equation system, this steps are made: first the 3D points are rotated

to be in the xy plane. This is done by multiplying the points by the rotation matrix that is the inverse of the rotation matrix that defines the orientation of the plane. After that, the points are newly rotated to be simmetric with respect to the y axis. This is done by calculating the angle between the first point and the last point and then rotating the points by this angle. After that, the equation is solved and the equation of the parabola is defined. Finally the points are computed and rotated back to the original position in the 3D space. Eventually checks are made to see if the parabola is feasible or not, in terms of length. The script returns the points that define the parabolic path.

3.1.2.6 plot3D_function_from_points.m

This last script is a capillary script used from the GUI script to plot the points on a 3-dimensional environment and ask the user whether the parabolic path is the one that they wanted or if it needs to be inverted in terms of concavity. It is not necessary to explain it in detail, because it was already explained in 3.1.2.3.

3.1.3 The Trajectory Server

Although the MATLAB module is very rich in terms of functions for the resolution of the mathematical task at hand and for the definition of useful data for the ROS-based architecture for trajectory, planning and execution, a communication element for the MATLAB module is still needed as explained before. This is the main reason why the `server_trajectory` server is implemented. This server is a ROS service server that is used to call the MATLAB module and to check if the trajectory generation was successful or not. So a new package is created, called `trajectory_server`, that contains the logic for the server itself. It is implemented as a ROS service server, so it is necessary to define the service message that it uses. The service message is defined in the `srv` folder of a new package called `trajectory_interfaces`. The need for a new package is due to the fact that the type of the message is not a standard one, so it is not available in the ROS distribution. The message is defined as follows:

- **trajectory_infos** is a string that contains the name of the bagfile in which the trajectory will be placed;
- **result** is a string that contains the result of the trajectory generation, that can be and error message or a success message;
- **success** is a boolean that is `true` if the trajectory generation was successful, `false` otherwise.

as specified in the srv file `Set_Trajectory_Start.srv`. The main logic of the node is starting the server and waiting for a request. When the request arrives, the server calls the MATLAB module and checks if the trajectory generation was successful or not. If it was successful, the server returns a success message, otherwise it returns an error message. The way the server calls the MATLAB module is with a shell script that is called from the server itself. The shell script calls the MATLAB module before checking if MATLAB is installed on the system. For that reason, it is necessary to install MATLAB on the system.

3.2 Trajectory Planning

3.2.1 Planning node

After working out the mathematical problems and solving them through a trajectory generation module in MATLAB, a ROS node that deals with the trajectory planning for the Fanuc robot is needed. For the planning node to work, the entire package must be defined. The package created is called `fanuc_m20ia_35m_planning_demo` and it will communicate with the previously defined `server_trajectory` server. Before going into the details of the planning node, it is necessary to introduce the new configuration of RViz that is been added to the `fanuc_m20ia_35m_moveit_config` package, which is called `planning_rviz.rviz`. This configuration is used to visualize the trajectory that is planned by the planning node, and it contains some useful plugin such as the `MarkerArray` plugin, `PlanningScene` plugin and `Trajectory` one. Also the `RvizVisualToolsGui` plugin is used to let the user interact with the graphical interface of RViz and with the planning node itself. Furthermore the `TF` plugin is used to visualize the frames of the robot and the environment, in particular to let the user understand the position of the robot and the environment in the space with respect to the world frame. The main code of the planning node can be divided into four parts, that will be explained in the following sections.

3.2.1.1 Initialization

The entire planning node is agnostic to the real robot, so, despite the fact that's the name suggests that it is a node for the Fanuc robot, it can be used for any robot that is compatible with the MoveIt framework. This is done by parametrize the node itself. In fact, when the node is launched, it is necessary to specify some parameters, that are the following:

- **joint_model_group_name**, which is the name of the group of joints that are used for the planning;
- **base_link**, which is the name of the link that `moveit_visual_tools` will use as the riferiment frame;
- **end_effector_link**, which is the name of the link that is considered as the end effector of the robot;
- **continuos_planning**, which is an integer value that specifies whether the planning is continuous or not (if it is, the node will keep planning until the user stops it manually);

After checking that the parameters are correctly set, the node initializes a node that will be used to communicate with the Rviz environment, and with the `server_trajectory` server. Some checks are carried out to make sure that the information about the robot are not inconsistent, so a function named `check_validity` is called and eventually the node is shut down if the checks fail. Otherwise the node initializes the `move_group_interface` and the `moveit_visual_tools` objects, which will be used to plan and execute the trajectories, along with the `planning_scene_interface` object, which will be used to add and remove objects from the planning scene. In this case there are no objects to add to the planning scene, but they will be added in the next chapter. After this some useful variables are initialized, and some closure

functions are defined, that will be used to handle the graphical interface of RViz and the service response. Note that, making the same node handle the communication with the server and the graphical interface of RViz is made to assure that if the node is waiting for the user to interact with the graphical interface, it will not be able to send and receive a response from the server, and viceversa.

3.2.1.2 Planning of the trajectory

After the initialization, the node starts the planning loop, which will go on until the planning fails or the user stops the node, this is valid only if the planning is continuous, otherwise the node will shut down after the first planning. At each iteration of the loop, the node prompts the user to proceed through the graphical interface of RViz (everytime "the user proceeds through the graphical interface of RViz", it is intended that the user must click on the "Next" button of the graphical interface) and if the user proceeds, the node checks if the trajectory server is ready to receive a request, and if it is, it generates a conventional name for the bagfile, made of the current date and time, and then it sends a request to the server. If the server is not ready, the node returns an error and restarts the loop. After the request is made the MATLAB script is launched, and the user can specify what kind of parabola they want to generate through the graphical interface, as explained in the 3.1.2.3 section. If the parabola was generated correctly, the node reads the bagfile that contains the trajectory points, and then it prompts the user to let the robot get t othe first point of the trajectory, and if the user proceeds, the node plans the trajectory through the `computePointToPoint` function. This function checks if the last point of the trajectory computed is the same as the desired pose, by doing forward kinematics. This is done because of some problems that were encountered during the development of the node, in particular the fact that the planner sometimes finds a solution that is not the final pose of the trajectory, but is a pose that is "near" to the final pose, according to the documentation of the MoveIt framework as explained in the 3.2.1.3 section. Note that the planning pipelines that are used and the choices that are made are explained in the 3.2.1.3 section. If the planning is successful, the node draws the trajectory on the RViz environment and then it prompts the user to let the robot execute the trajectory, and if the user proceeds, the node executes the trajectory.

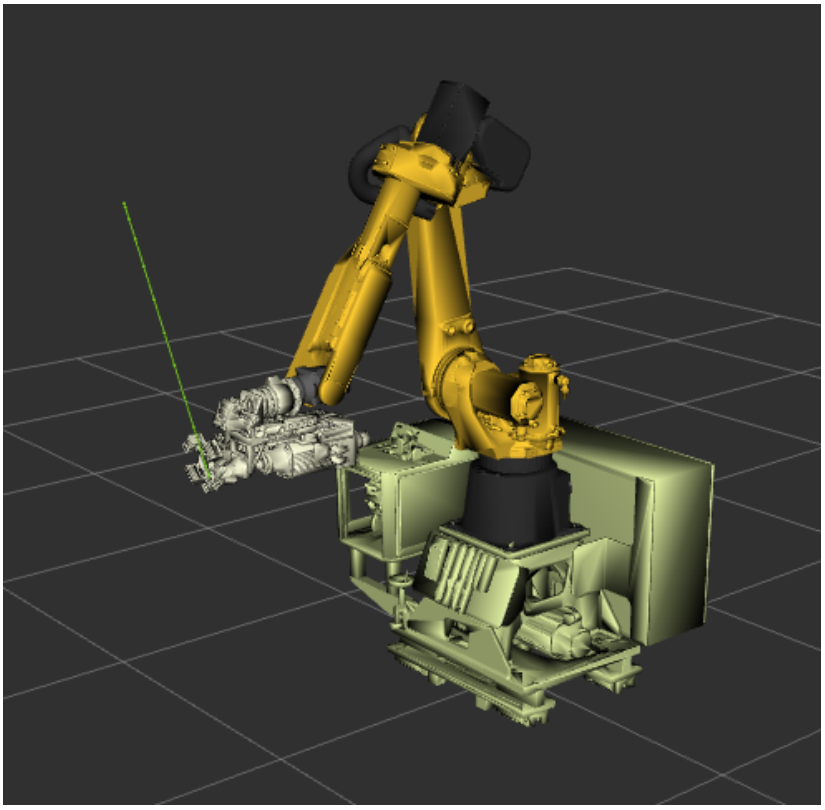


Figure 3.12: Point to point executed by the planning node to reach the first point of the trajectory

After the robot reaches the first point of the trajectory, the node prompts the user to visualize the parabolic path created by the MATLAB script in red color.

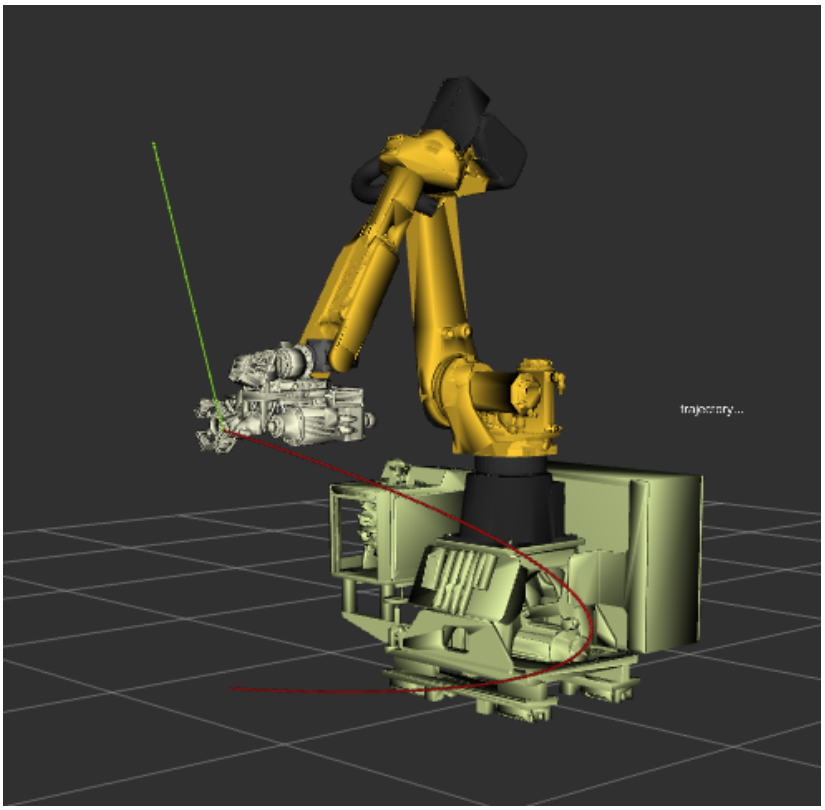


Figure 3.13: Bagfile parabola generated by the planning node

Then the node prompts the user to let the node plan the parabolic path through the

`computeCartesianPath` function, and if the planning is successful, so that the fraction of the path that is computed is 100%, the node prompts the user to let the robot execute the trajectory. Node that in order to plan a feasible trajectory, the `jump_threshold` parameter of the `computeCartesianPath` function is set to 100.0 and the `eef_step` parameter is set to 0.01, so that the planner is able to plan a trajectory that is feasible in terms of agility of the robot and also in terms of next configurations in joint space. In fact the `jump_threshold` parameter is used to specify the maximum distance, in the joint space, between consecutive points in the resulting path. Some tests were carried out to find the best value for this parameter, and it was found that extreme values, such as 200.0, or 0.0 (which means that the planner will not check for the distance between consecutive points) are not good, because the planned trajectory includes some points where, in the middle, the robot is reconfiguring itself, modifying the configuration of the joints in a way that is not feasible in terms of agility of the robot, or that is not feasible in terms of collision avoidance. In fact between two points no collision is checked by the planner, so if the robot is reconfiguring itself in a way that is not feasible in terms of collision avoidance, the planned trajectory will include some points where the robot is in collision with itself. Note that, also the 100.0 value is a large value, but it was chosen because it is difficult to find an upper bound for the distance between consecutive points in the joint space and a conservative value was chosen.

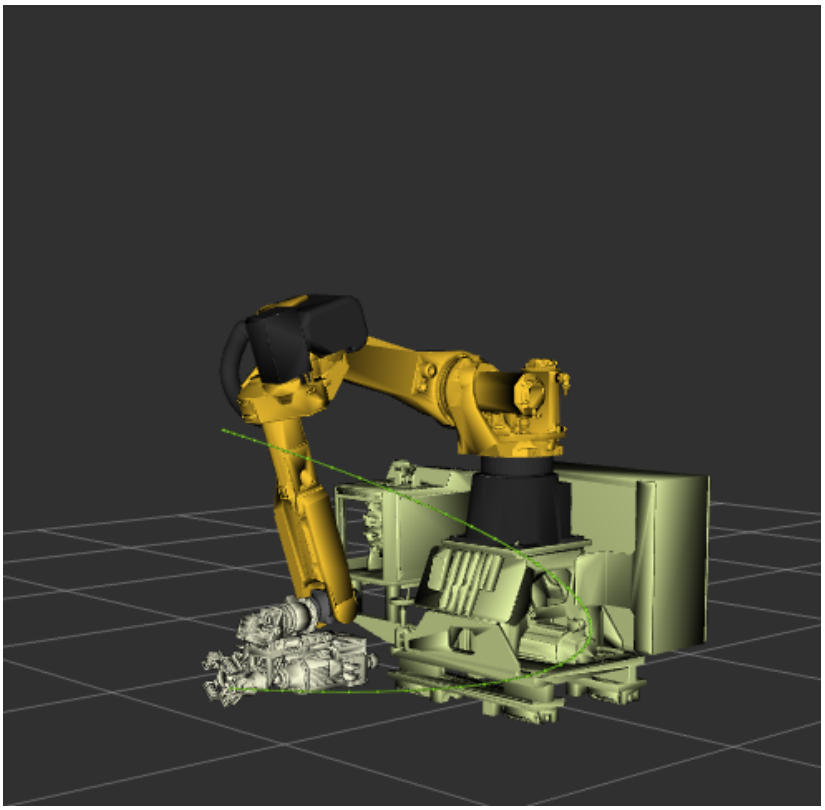


Figure 3.14: Real planned parabola executed by the planning node

After the execution of the trajectory the node prompts the user to let the robot plan the trajectory to return to the home position, and if the planning is successful, the node executes the trajectory. Note that in this case the home position is specified as the initial position of the robot when the planning node is launched. It is important to point that there is another way to return to the home position, and it is used

by the planning node, if the planning fails. In this case the node uses the same parabola that was used to reach the first point of the trajectory, but in this case the parabola is used in the opposite way, so that the robot is able to return to the first point of it, and then plan the trajectory to return to the home position with the `computePointToPoint` function. This is done because it was thought that the parabola that was executed, in the first phase of the trajectory, was feasible, so it can be used to return to the first point of the trajectory, and then plan the trajectory to return to the home position. This is done by letting the user know that an inverse path is been planned, and let him execute it in two steps, one for reaching the initial point of the trajectory and the other one to reach the initial position. The choice to make this planning and execution in two ways come from two main factors: the first one is that planning the reverse trajectory plus the point to point trajectory to return to the home position in a single step, assumes that the robot is able to perfectly reach the intermediate point between this two planning phases, and this can be not always true, due to errors in the execution. The second one is that if the planning is capable to plan the first phase of the trajectory and not the second one, the robot should end in a better configuration despite the one in which it was after the execution of the parabola. These are just assumptions, because it's assumed, that if a linear path from the initial position to the first point of the trajectory is computed, also a reversed one is feasible.

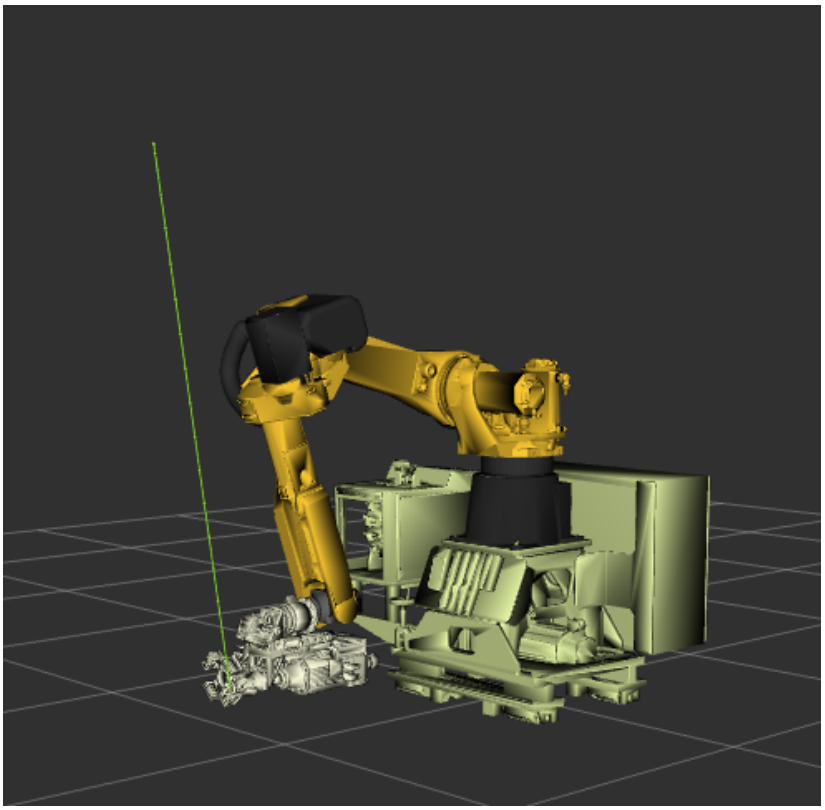


Figure 3.15: Point to point executed by the planning nod to return to the initial position

Finally the node restarts the loop, if specified so, and another trajectory is planned.

3.2.1.3 Planning pipeline choice

In the creation of the planning node, `ompl` was chosen as the default planning pipeline, because it is the one that is more reliable and that is able to plan the trajectory in the shortest time, in the default configuration. The problem with this pipeline is that, often, it is not able to plan a trajectory that is feasible in terms of agility of the robot, in fact even if the planning is successful the robot ends in a bad configuration, and the next trajectory planning fails. Another issue that was encountered is that, sometimes, the planner find a solution that is not the final pose of the trajectory, but is a pose that is "near" to the final pose, according to the documentation of the MoveIt framework. This behaviour was investigated on, and it seems that the problem is related to the fact that sometimes the planner find an approximate solution but the error code that is returned is still SUCCESSFUL, as explained in [6], but the main problem is that this "near" solution is not very near, and the robot ends in a very bad configuration. To solve this problem, a first approach was to use the `computePointToPoint` to iterate the planning for a maximum established number of times, but this approach was not very efficient, because the planning was very slow. The chosen solution was to use the `pilz_industrial_motion_planner` pipeline, which is a more efficient pipeline and it is able to plan the trajectory in a shorter time, and it is able to plan a trajectory that is feasible in terms of agility of the robot, because it can plan linear trajectories, which are more feasible than randomized ones. A shortcoming of this pipeline is that it is not able to plan complicated trajectories, but it was chosen as the default one for compute the point to point trajectory, because it is more reliable than the `ompl` one, and it is able to plan the trajectory in a shorter time. Furthermore, it was thought that, given that the robot is on a slider, that is mounted on a linear guide, the work space of the robot will be limited, in front of the robot. If a piece should be drilled, the piece should be placed in front of the robot. Given this, it was preferred to use a linear pipeline, such as `pilz_industrial_motion_planner` to reach all the possible points, in front of the robot, in a linear way and in the shortest time. Some experiments that were carried out show that the `ompl` planner often plans a trajectory that is not feasible in terms of agility of the robot, despite the fact that the planning is succesful, and, even if some points are reachable with the `ompl` and not with `pilz_industrial_motion_planner`, are points outside the "work space" of the robot, as said before, so they are not useful. By that it's possible to say that the `pilz_industrial_motion_planner` is the best choice for the purpose of the project. Another shortcoming of the `ompl` planner is that some linear trajectories are planned as an entire reconfiguration of the robot, making fast movements that are not feasible on the real controllers. Finally, it's important to point out that some parabolas also could be rejected by the planning pipeline, because the constraints of length of 3.14 meters, is a constraint that sometimes is not feasible in terms of keeping the end effector orientation constant, and the planner is not able to plan the trajectory.

3.2.1.4 Execution of the trajectory

After each planning phase the node prompts the user to let the robot execute the trajectory, and if the user proceeds, the node executes the trajectory. The execution of the trajectory is carried out through the `move_group_interface` object, and the `moveit_visual_tools` object is used to visualize the trajectory in

the RViz environment. This behaviour is valid only in this version of the planning node, because in the next chapter the real controllers will be used to execute the trajectory, and the need of store information about the execution of the trajectory will be needed. At the end of this phase, some considerations about the planning are made, in particular how the seven degrees of freedom of the robot are used to plan the trajectory. In this case the kinematic solver that was used is the KDL one in its default configuration, whcih optimizes the quad norm of the velocity. No changes were made to the configuration of the solver, despite some tests were carried out to find different configurations that could be more efficient, but the default one was the best one for the purpose of the project [7]. This is true because, since the robot is redundant, it makes no sense to consume more energy to move joints in a way that is not useful for the trajectory execution. Furthemore, given that no secondary tasks are specified, like joint limits avoidance, or obstacle avoidance, the default configuration of the solver is a good choice. During the node definition process, furthermore, also other kinematics solvers were tested, such as the `pick_IK` one to try to solve problems related to the previous section. The experiment was carried out because this type of solver is able to take as input parameters such as avoiding joints limits and also minimal displacement as explained in [8]. Finally note also that no constraints were added to the parabola definition in the MATLAB script, other than checks for the action space, so sometimes the planning fails because the robot is not able to reach the desired pose, but it was decided to not add constraints in this sense, because it was difficult to undestand what kind of constraints were needed, and even if the constraints were added, some feasible parabola could not be planned. That decision was made because even if a planning fails, the user can always try again, and the planning will be successful if the user is able to find a feasible parabola.

3.2.2 Integration with the Trajectory Generation module and the configuration package

Finally, after defining the trajectory generation module, the planning node and all the required communication interfaces, the integration of all these modules can be done. As described in 3.2.1, it is the *fanuc_m20ia_35m_planning_demo* package that contains the workflow that coordinates the different modules, which means launching the executable of the planning node is fundamental, however the `acg_resources_fanuc_m20ia_35m_moveit_config` package and the `server_trajectory` must be taken into account too, as their demos must be launched simultaneously too. To be more specific, all components must be launched in the same folder, that is the one containing the `fanuc_m20ia_35m_planning_demo` package, so that the saving of bagfiles and the calls to system scripts are successful.

To guarantee the correct execution of the planning phase, two different launch files were created: the *alone_start.launch.py* and the *demo.launch.py*. The first one is used to start the planning node alone, so the user must manually start the `trajectory_server` and the `acg_resources_fanuc_m20ia_35m_moveit_config` demo with planning configuration. To avoid the need of starting multiple terminals by hand, the *demo.launch.py* was created, which starts all the required nodes in the correct order, so that the planning phase can be executed without any manual intervention,

more informations and details about the launch files and parameters are given in the `README.md` file of the `fanuc_m20ia_35m_planning_demo` package. When this launch file is run, three terminals are opened, one for each node, and at the end of the planning phase, the user can terminate the execution by pressing the `Ctrl + C` keys on the process they want to stop.

3.2.3 Secondary choices and considerations

At the end of this chapter, the trajectory planning for the Fanuc robot is developed. It is important to point out that the choices made in the development of the trajectory planning are not the only possible ones. In particular, the trajectory planning could be done in a different way, for example using the `ompl` planning pipeline, to allow the robot to reach more complex poses, but at the same time, accepting more trajectory failures. Another possible choice that was investigated on was to use the `Parallel Planning` pipeline, that allows to plan the trajectory for the robot in parallel with different planners, and to choose the best trajectory [9]. This route was not taken because of the complexity that it would have added to the project, and because the desired behaviour was to have a simple trajectory planning that was able to reach the desired poses with a high success rate. Furthemore, another choice that was ”imposed” by the lack of the information was the choice to use no acceleration limits in the planning. This choice was made because the acceleration limits of the Fanuc robot were not known, and it was not possible to find them in the documentation. The only useful information was about similar robots, but it was not possible to know if the acceleration limits were the same. This choice could be changed in the future, if the acceleration limits of the Fanuc robot are be known.

CHAPTER 4

WP3-SIMULATION, EXECUTION AND ANALYSIS

In this chapter the first dynamic simulation of the robot is produced alongside an independent joint space controller designed and tuned to follow the trajectories coming from the planning module and an analysis module to evaluate the quality of a trajectory execution. To do so the system was extended in terms of robot description and configuration, but also in terms of planning and control. By choosing an independent joint controller, it is assumed that all the non-linearities of the robot and the mutual influence of the joints are not taken into account, but considered as a disturbance. This is a simplification that allows to have a first approach to the control of the robot, and to have a first evaluation of the quality of the trajectories. Each joint controller must guarantee disturbance rejection, tracking and stability.

4.1 Design of the robot controller

4.1.1 Choosing the controller

First of all it is necessary to choose the controller that will be used to control the robot. In this case, the choice was made taking into account the requirements given. In fact it was requested to visualize, in a simulation tool called `Gazebo`, the real dynamic of the robot, and to control it in a joint space. To do so the `fanuc_m20ia_35m.ros2_control.xacro` file has been modified in order to provide robot joints with effort configuration. Otherwise, with velocity or position configuration, the simulation would have not been realistic, but it would have been like writing in the integrator of the robot the desired joint position or velocity. After this modification, the other request made by requirements is that the controller had to be able to receive an entire trajectory offline and then to execute it. This is a typical task of a *JointTrajectoryController* in ROS2, in particular the *JointTrajectoryController* with effort command interface is used. This kind of controller is easy to setup, it is done by a configuration file automatically

generated by the `moveit_setup_assistant` tool in the previous chapter called `ros2_controllers.yaml`. This file is used to configure the controller and to specify the joints that it has to control. Initially the parameters of the controller are set to the default values, then they are tuned in the 4.1.3.1 section. Note that the Gazebo simulation have an update rate of 1000 Hz, so the controller has to be able to work at this frequency and to do so it is necessary to set the `update_rate` parameter to 1000 Hz in the `ros2_controllers.yaml` file.

4.1.2 Configuration of the envirorment: ros2_control

As said in the previous section, the Ignition Gazebo environment is the one that will be used to simulate the Fanuc, and to do so, the `fanuc_m20ia_35m.ros2_control.xacro` file must be edited to include the `IgnitionROS2ControlPlugin` tag. Meanwhile, the original values for all joints in terms of effort, position and velocity are all set to 0 to allow any dynamic simulation to start with a robot only affected by gravity and additional dynamic elements, other than a controller, if active.
Through the dynamic simulation of the robot a tedious problem in the robot model was encountered, which included self-collisions of the robot components with eachother and in turn a self collapse of the robot. This problem led to an additional modification to the *fanuc_m20ia_35m.urdf.xacro* file by adding the following tag:

```
<gazebo>
  <self_collide>1</self_collide>
</gazebo>
```

Regarding the integration with the Ignition Gazebo environment, the *moveit_gazebo_ros2_control_demo.launch.py* launch file was produced to load the Fanuc robot in it and work with it. The launch file not only spawns the controllers, the controller manager and the RViz node, but it also launches Ignition Gazebo and spawns a node that exclusively focuses on putting the Fanuc robot inside the environment (the `ignition_spawn_entity` node). Also, delays in the node spawns are introduced to avoid crashes due to timeouts in loading the robot model and loading the controllers while the controller manager has yet to start.
Upon launching the nodes the `fanuc_m20ia_35m_slide_to_end_effector_controller` controller is activated, which deals with the gravity effects on the Fanuc robot, thus allowing it to stay in its home position at the start of the simulation.

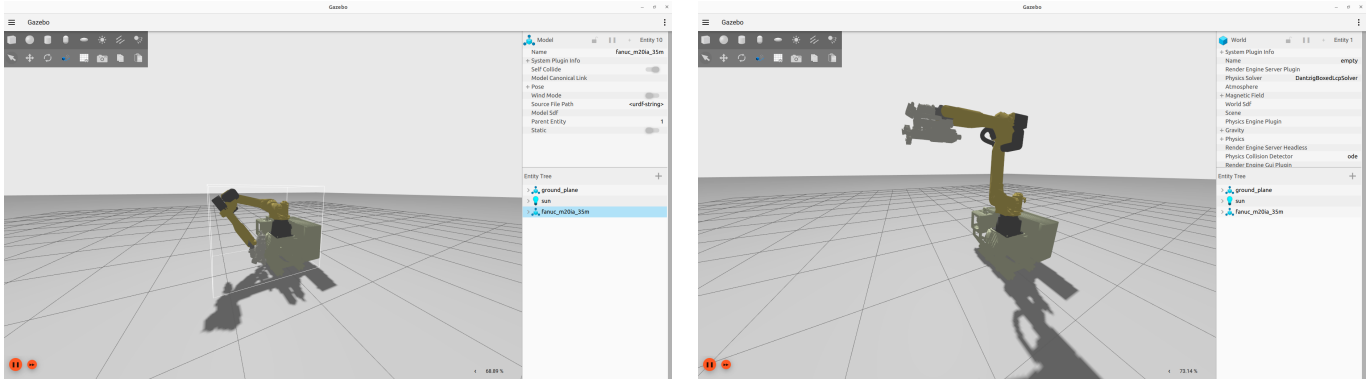


Figure 4.1: Fanuc robot inside the Ignition Gazebo environment, respectively without and with an active controller.

The controller that allows for the Fanuc to stay up is the same that is used to help it with following trajectories, so its parameters in terms of gains are the same that will be seen in 4.6.
In order to work both with the planning module and the Ignition Gazebo component, a new version of `fanuc_m20ia_35m_planning_demo`'s *`demo.launch.py`* must be taken into account, as it adds the launch of the Ignition Gazebo environment and the spawner node for the Fanuc robot inside the environment. In this case, the planning of the trajectory has a consequence on the Ignition Gazebo environment, as the Fanuc robot inside of it follows the trajectory the robot displayed inside RViz, so it is just used as a visualizator and to let the user interact with the planning. This trajectory tracking occurs thanks to the `rviz` and `gz_ros2_control` interacting with eachother through the `joint_states` topic, as while the RViz display of the Fanuc robot executes the trajectory, the joint states are also followed by the dynamic simulation step by step.

4.1.3 Changing the planning module

With the necessity of storing information about the trajectory executed, some changes in the `fanuc_m20ia_35m_planning_demo` package are necessary. In particular, the planning node should be able to store the trajectory in a bag file. This is done by adding a new node called `action_node.cpp` that has the role, as the name suggests, to be an action client that sends the trajectory to the controller and receive, in the feedback callback, the desired joint positions and the actual ones, throughout all the trajectory. This node is the same that is used as service client to comunicate with the MATLAB parabola generator and the same that is used to wait interaction by the user on the RViz environment as described in the 4.2.

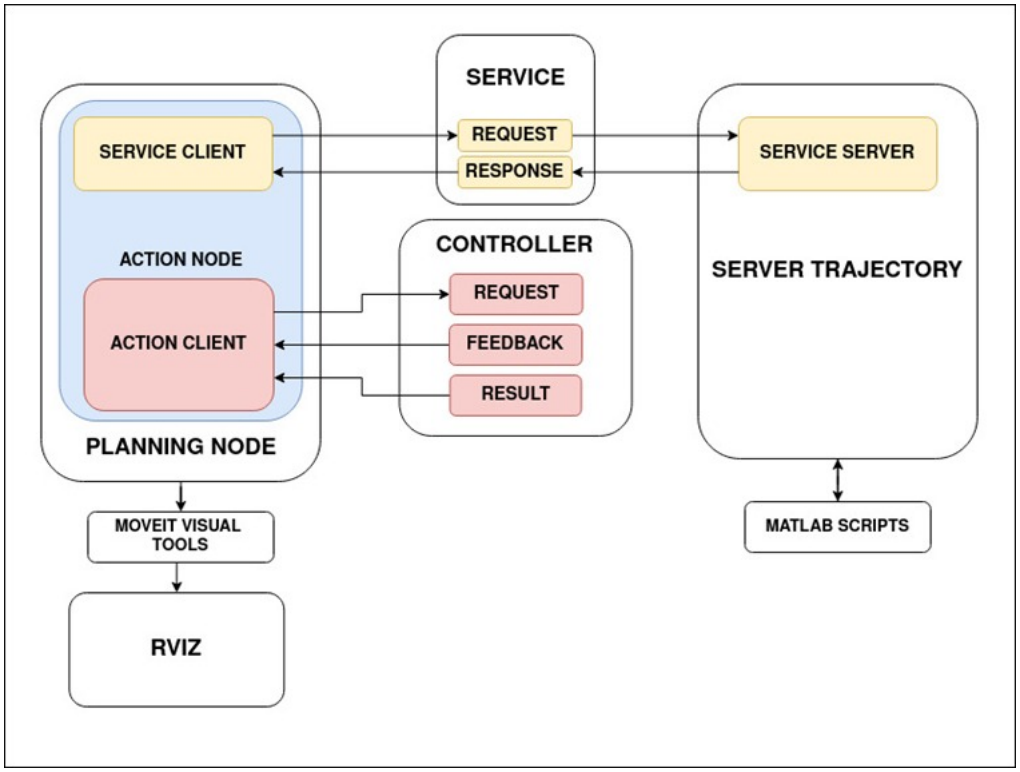


Figure 4.2: Sequence diagram of the planning module.

In particular, given that just the trajectory executed has to be monitored and stored, the `action_node.cpp` is equipped with some useful functions to manage the name of the bag file, when to store a trajectory or not, and all the logic to manage with errors and exceptions. To syncronize the main node with this one, to wait the completation of the trajectory goal, a `future` object is used, and the `get()` method is called to block the execution of the main node until the trajectory is executed. This `future` object is instantiated on a `promise` object that is set in the feedback callback of the action client.
This change is necessary to allow the `fanuc_m20ia_35m_planning_demo` package to be able to store the trajectory in a bag file, and then to be able to analyze it, but in reality there is another choice that was proposed but not considered, as it was less extendable and less flexible. This choice was to continue to use the `move_group_interface` to send the trajectory to the controller, and then to build a secondary node as a LyfeCycle node that is it activated when the parabola trajectory is executed, that listens to the `feedback` topic, and then store the trajectory in a bag file, after the controller is deactivated. This option was not considered because in the next chapter another change in the `fanuc_m20ia_35m_planning_demo` package is necessary, so it was better to have a unique node that is able to manage all the logic of the package, and then to be able to extend it in the future. Another change that was made is to add a new parameter in the `fanuc_m20ia_35m_planning_demo` package, called `controller_action` used to specify the action name of the controller, in this case `fanuc_m20ia_35m_slide_to_end_effector_controller/follow_joint_trajectory`.

4.1.3.1 Tuning PID parameters

In order to accomplish the task of independent joint control a PID-based control for each joint is developed as no model of the robot was provided. An additional strategy consists in a step of system

identification, but this eludes the scope of the project. For the PID controller in question, it is possible to specify for each joint the K_P, K_I and K_D gains of the PID. Note that a MIMO (*Multiple Inputs Multiple Outputs*) PID is considered since for the structure of the robot, the choice of the PID parameters of a joint can influence the behavior of the adjacent joint (probably all the open kinematic chain). The strategy used initially in our work it was proposed by Ziegler-Nichols, which starts with the identification of the proportional gain K_u for each joint and, subsequently, the oscillation period T_u of the output. It is important to note that our outputs are the joint positions: a ROS/ROS2 plugin named PlotJuggler ([10]) is considered. Using this tool, it is possible to appreciate easly the trend of each output. So, for each joint the oscillation period T_u was obtained and the parameters K_P, K_I and K_D were derived according to these relations:

$$K_P = 0.6K_u \tag{4.1}$$

$$K_I = \frac{1.2K_u}{T_u} \tag{4.2}$$

$$K_D = \frac{3K_uT_u}{40} \tag{4.3}$$

Obviously, these equations are valid only if a PID (*Proportional Integral Derivative*) controller is considered. The use of the PlotJuggler tool was fundamental in this phase as it allows for the choice of a topic to be analyzed and, after identifying the quantities to be analyzed (in the case in question, the positions of each joint), obtaining an online monitoring. In this regard, the *observed* topic was `/joint_states` and the position of each joint (`joint_world_to_slider` and from `fanuc_m20ia_35m_joint1` to `fanuc_m20ia_35m_joint6`). Fig. 4.3 shows the positions of the joints during a simulation conducted as an example. In addition, in order to simplify this monitoring activity for tuning the PID gains, a configuration file has been created for the aforementioned tool, called *configuration_plotjuggler.xml* which allows the easy import of all joint positions within the graph to pay attention to.

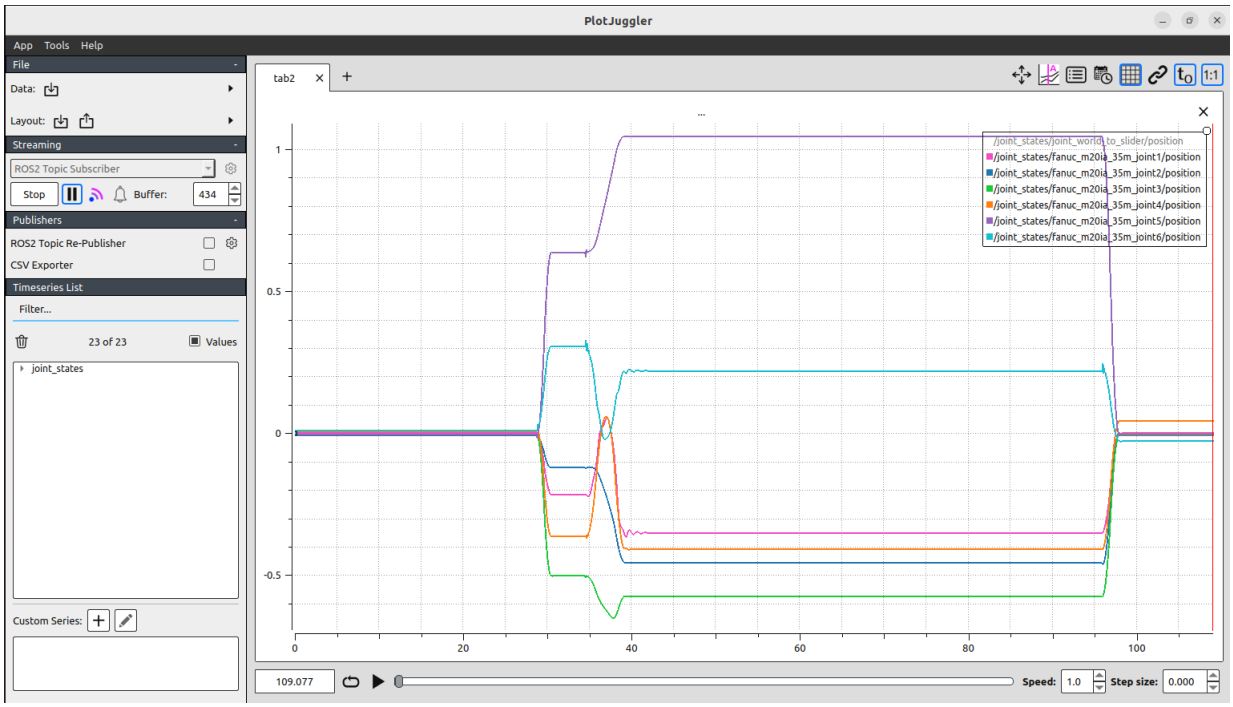


Figure 4.3: Trends of joint positions in PlotJuggler

At the start of the Ziegler-Nichols procedure, the gains K_u in Tab. 4.1 have been considered. These values were obtained by simulating several times in Gazebo and monitoring in PlotJuggler whether oscillations occurred on the positions of the joint variables.

Joint	K_u
joint_world_to_slider	917000.0
fanuc_m20ia_35m_joint1	944000.0
fanuc_m20ia_35m_joint2	800000.0
fanuc_m20ia_35m_joint3	4000000.0
fanuc_m20ia_35m_joint4	398400.0
fanuc_m20ia_35m_joint5	767000.0
fanuc_m20ia_35m_joint6	45340.0

Table 4.1: Ziegler-Nichols tuning: values of K_u

Joint	$T_u[ms]$
joint_world_to_slider	118
fanuc_m20ia_35m_joint1	30
fanuc_m20ia_35m_joint2	72
fanuc_m20ia_35m_joint3	11
fanuc_m20ia_35m_joint4	11
fanuc_m20ia_35m_joint5	32
fanuc_m20ia_35m_joint6	12

Table 4.2: Ziegler-Nichols tuning: values of T_u

When starting Gazebo, an oscillating behavior can be seen, as highlighted in Fig. 4.4 4.5 4.6 4.7 4.8 4.9 4.10 4.11. In a second step, the oscillation period T_u (Tab. 4.2) was obtained.

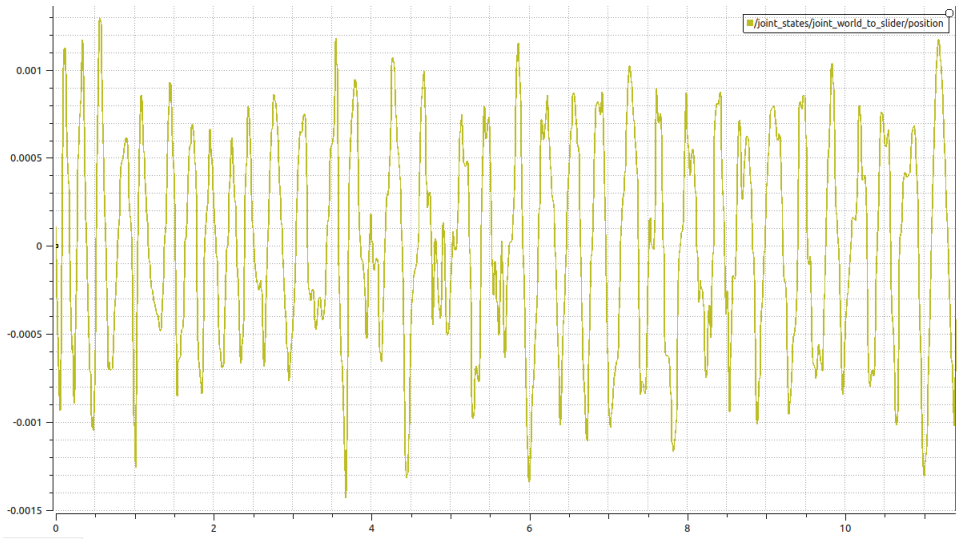


Figure 4.4: Situation when is applied only K_u in PlotJuggler for `joint_world_to_slider`

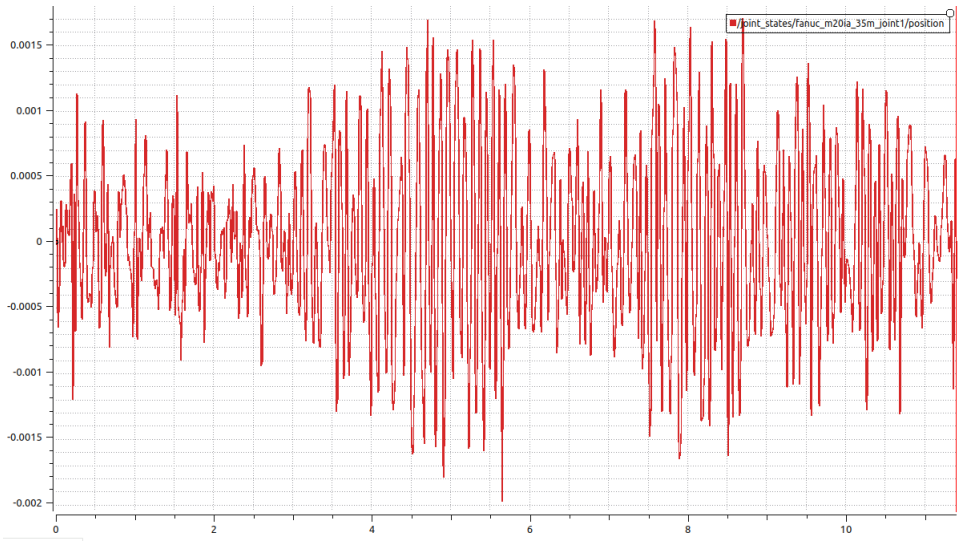


Figure 4.5: Situation when is applied only K_u in PlotJuggler for `fanuc_m20ia_35m_joint1`

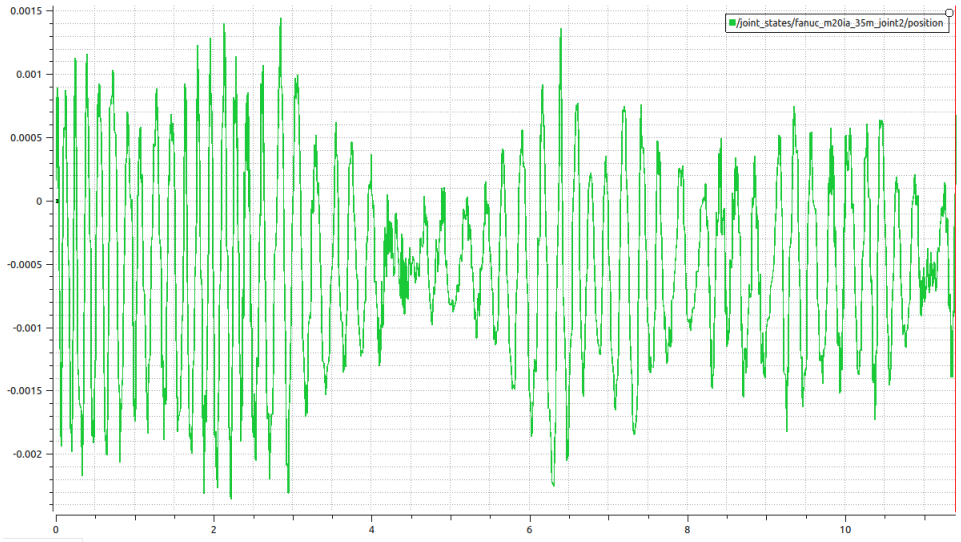


Figure 4.6: Situation when is applied only K_u in PlotJuggler for `fanuc_m20ia_35m_joint2`

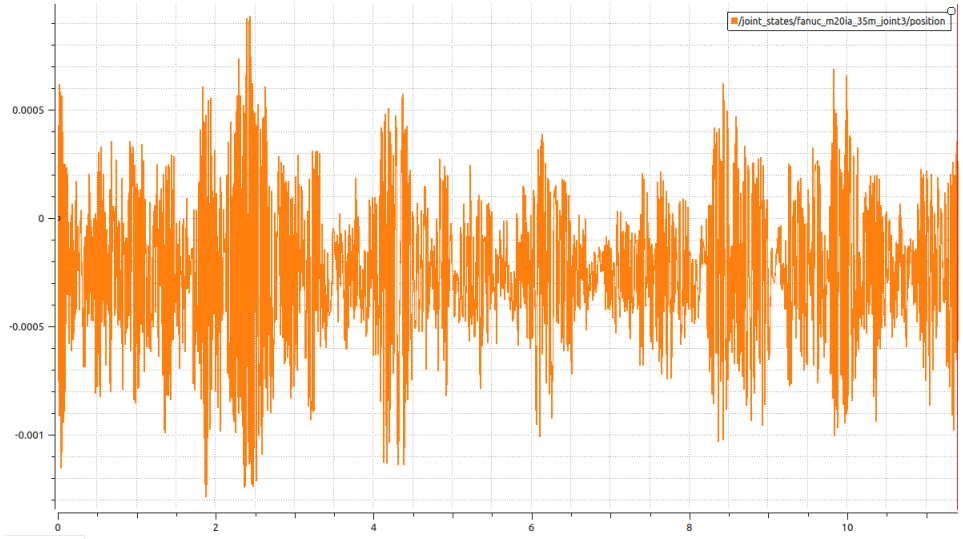


Figure 4.7: Situation when is applied only K_u in PlotJuggler for `fanuc_m20ia_35m_joint3`

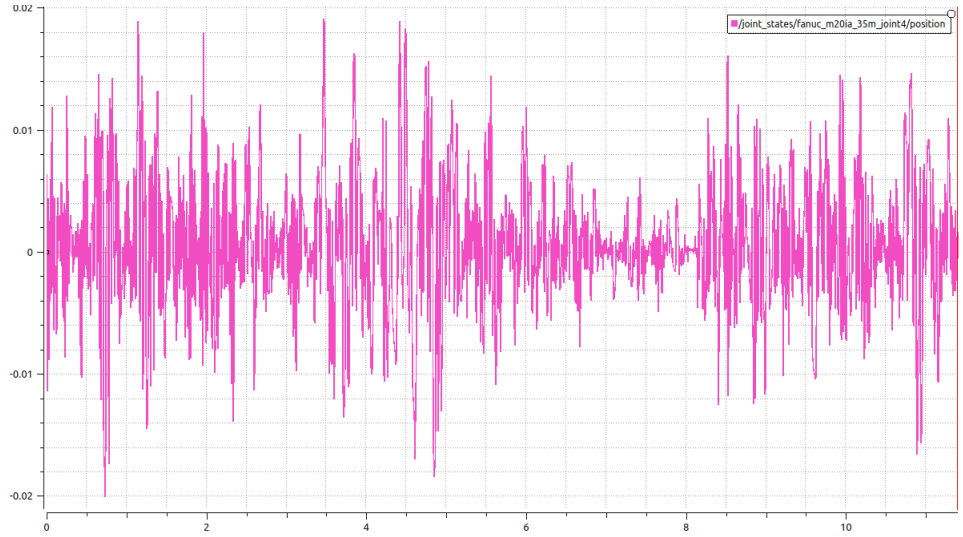


Figure 4.8: Situation when is applied only K_u in PlotJuggler for `fanuc_m20ia_35m_joint4`

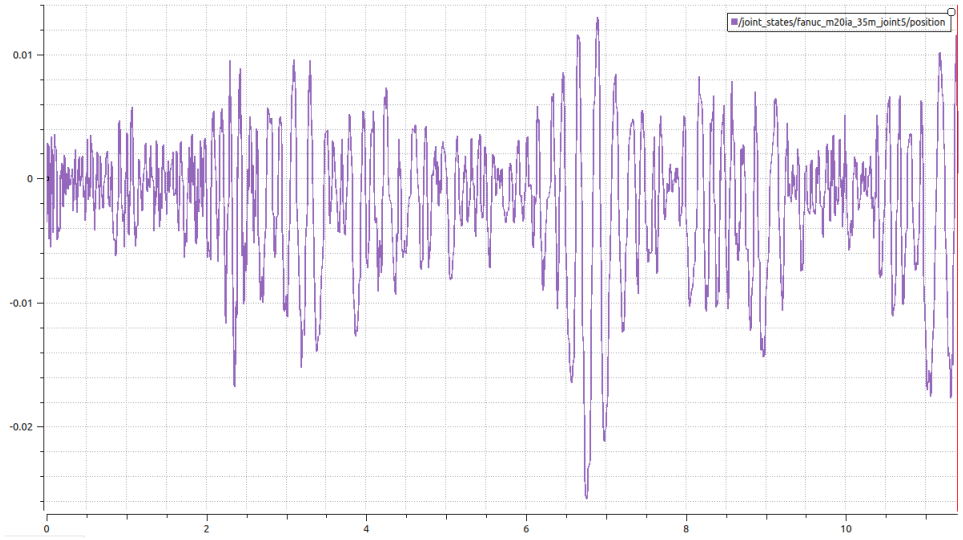


Figure 4.9: Situation when is applied only K_u in PlotJuggler for `fanuc_m20ia_35m_joint5`

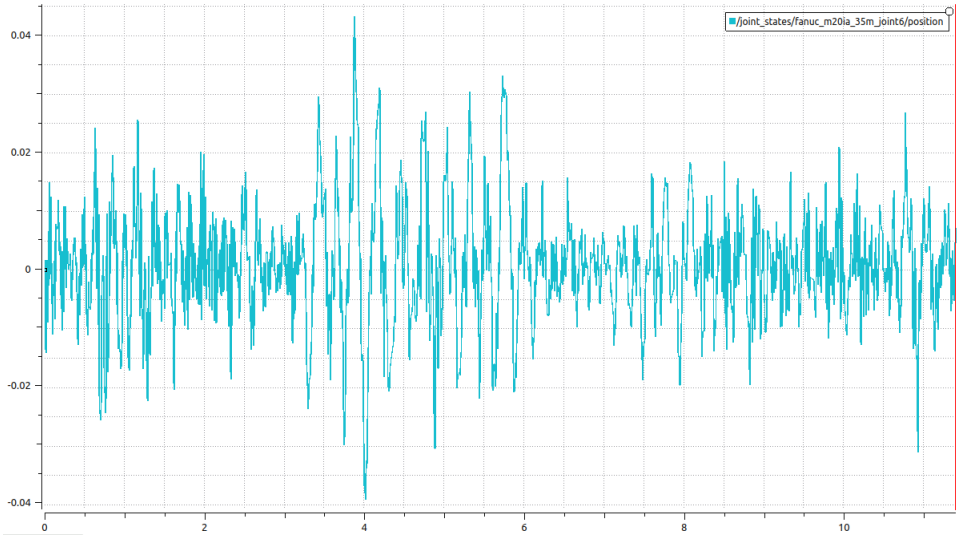
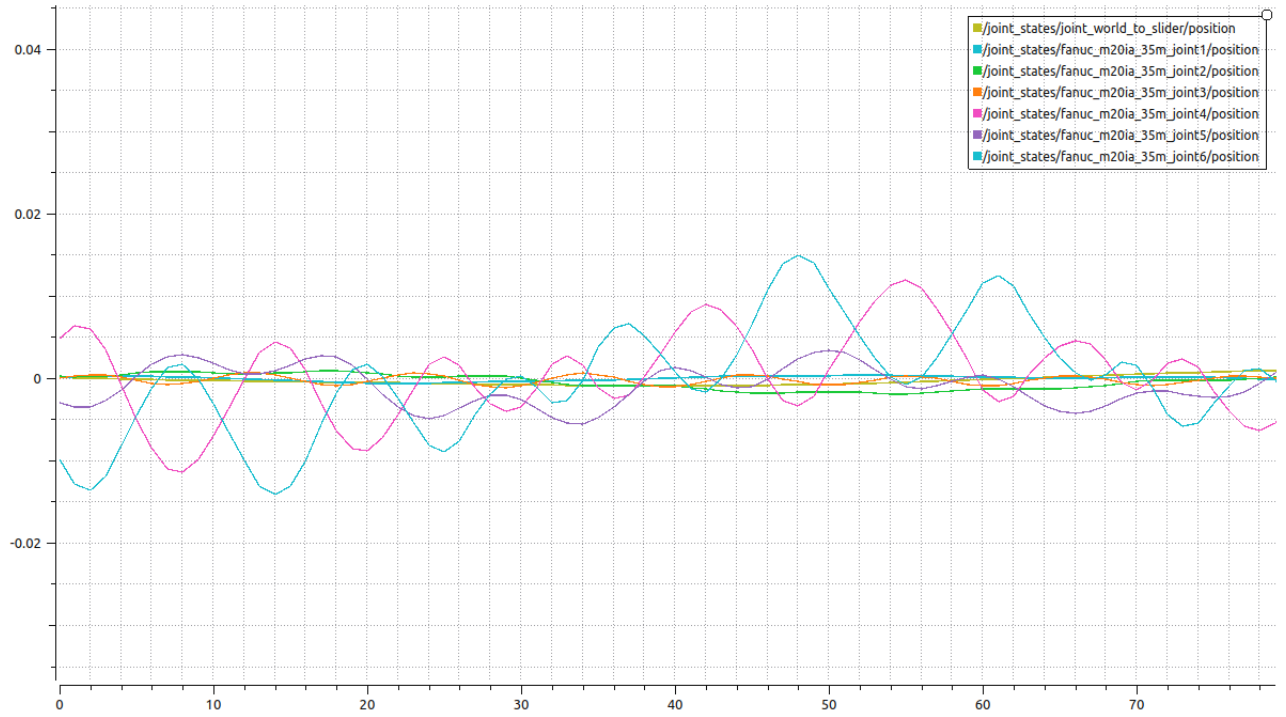
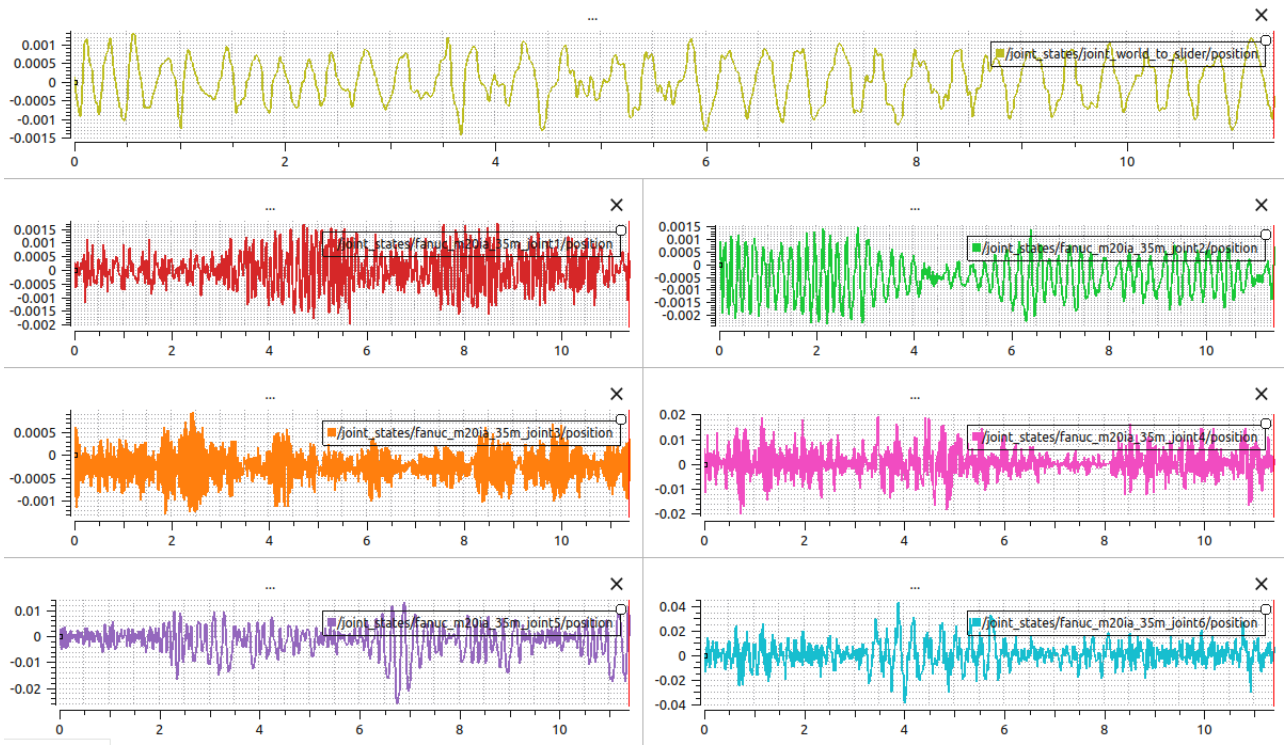


Figure 4.10: Situation when is applied only K_u in PlotJuggler for `fanuc_m20ia_35m_joint6`



(a) Overlapped view



(b) Separeted view

Figure 4.11: Situation when is applied only K_u in PlotJuggler (summary)

Since the project deals with a MIMO system, the oscillation periods are difficult to find. However, by analyzing the response of each joint in PlotJuggler an estimate of the periods was obtained for each of the joints. In particular, the periods shown in Tab. 4.2 were obtained by saving the simulation results offline in a .CSV file called *before_tuning_csv_dataframe_final.csv* (and related statistics in *before_tuning_csv_statistics_final.csv*). All the csv files are available in the *plotjuggler_configuration* folder.

Then, by importing the dataframe into PlotJuggler (and choosing the option shown in Fig. 4.12) the response for each joint was analyzed, as shown in Fig. 4.13 4.14 4.15 4.16 4.17 4.18 4.19 4.20 where a zoom was performed local on the response waveform (in Tab. 4.3 contains the time references of the simulation from which the period was extracted).

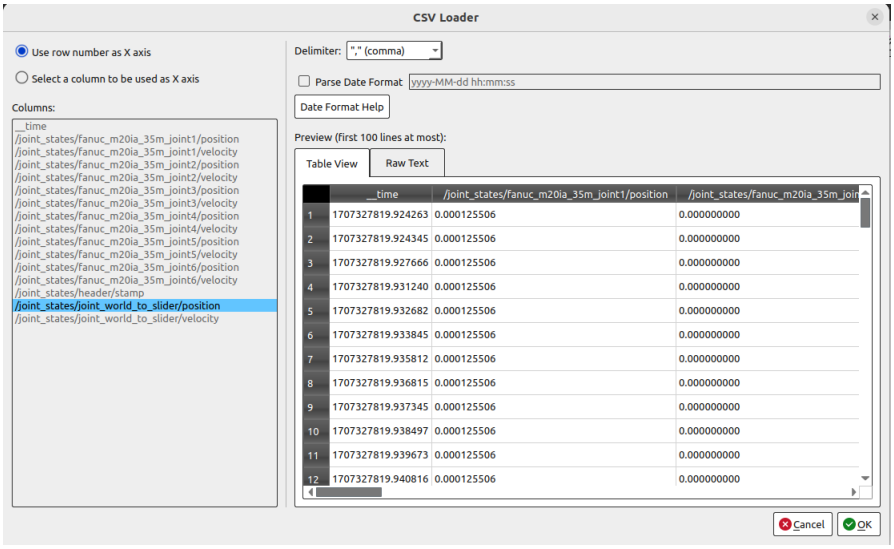


Figure 4.12: Settings used to import CSV files in PlotJuggler

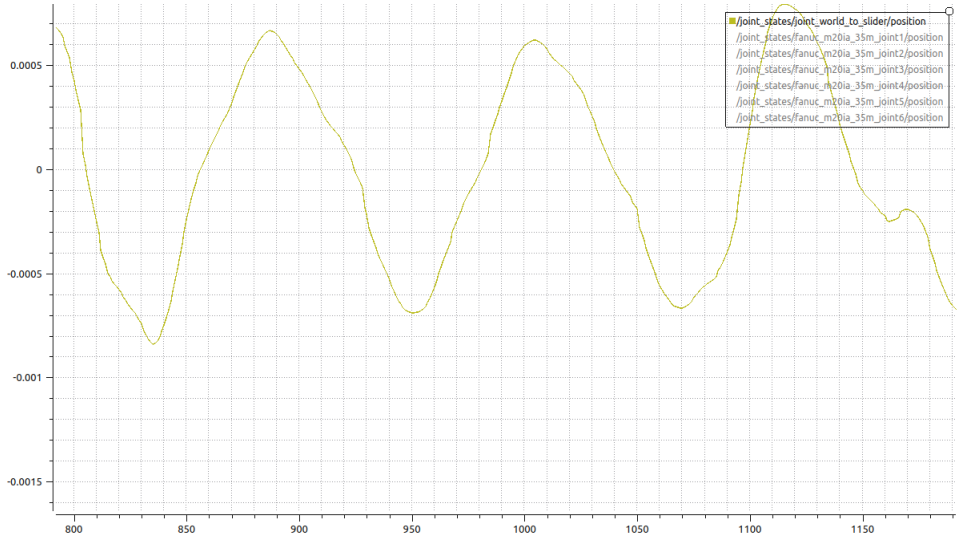


Figure 4.13: Evaluation of T_u in PlotJuggler for `joint_world_to_slider`

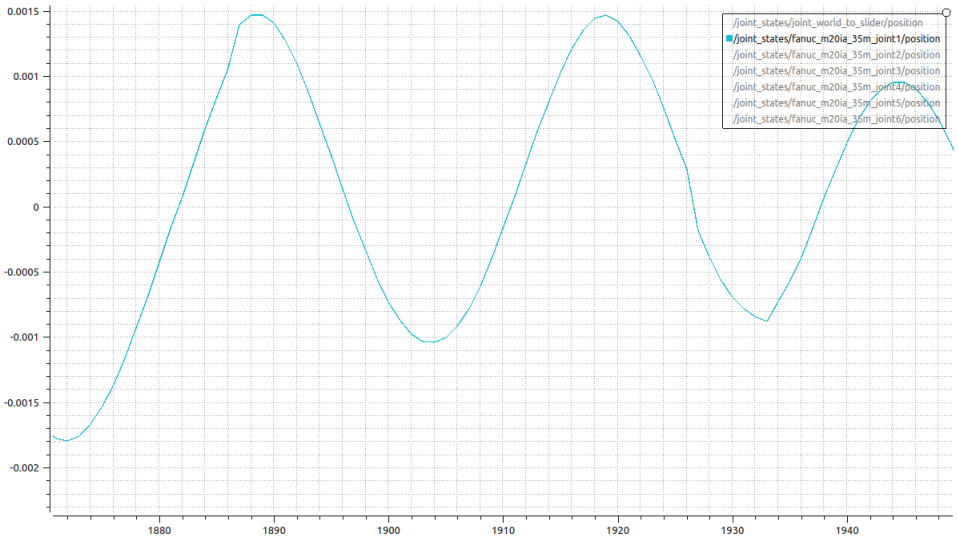


Figure 4.14: Evaluation of T_u in PlotJuggler for `fanuc_m20ia_35m_joint1`

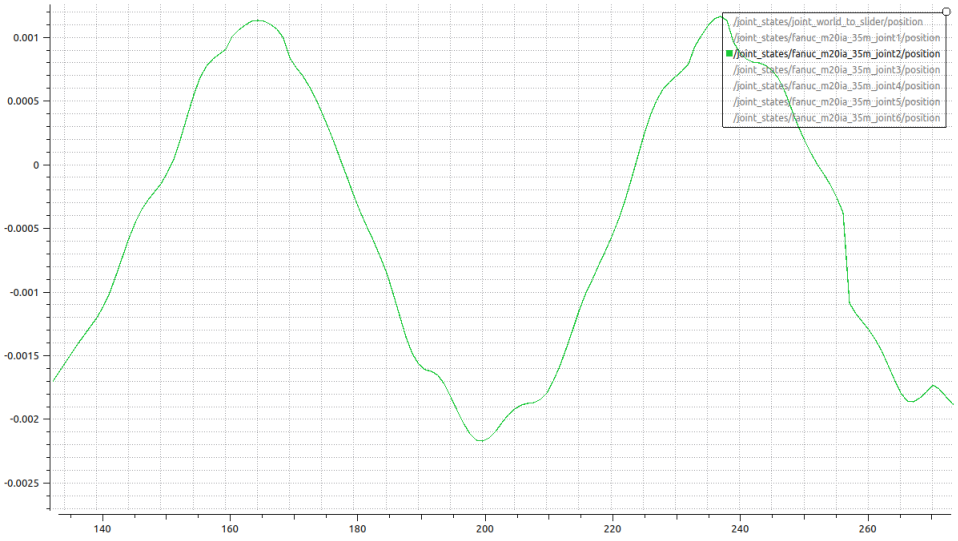


Figure 4.15: Evaluation of T_u in PlotJuggler for `fanuc_m20ia_35m_joint2`

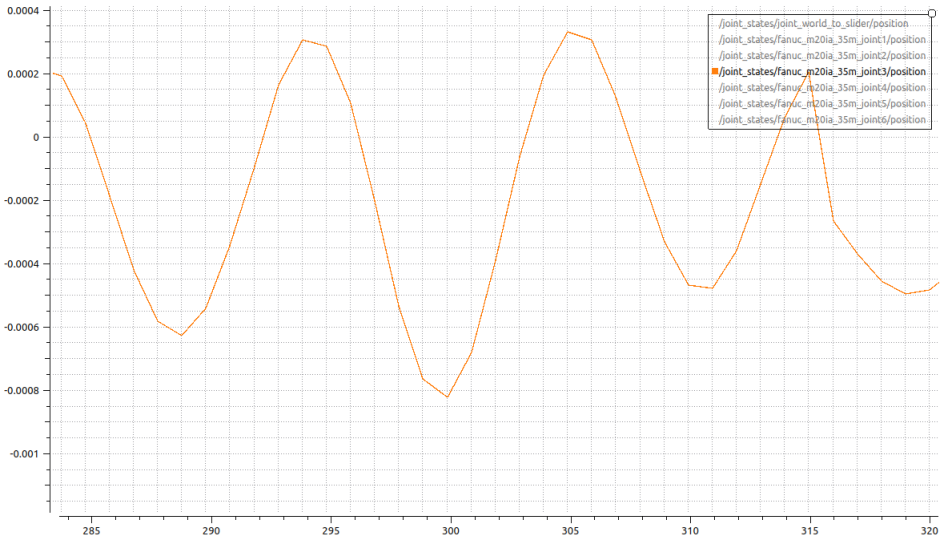


Figure 4.16: Evaluation of T_u in PlotJuggler for `fanuc_m20ia_35m_joint3`

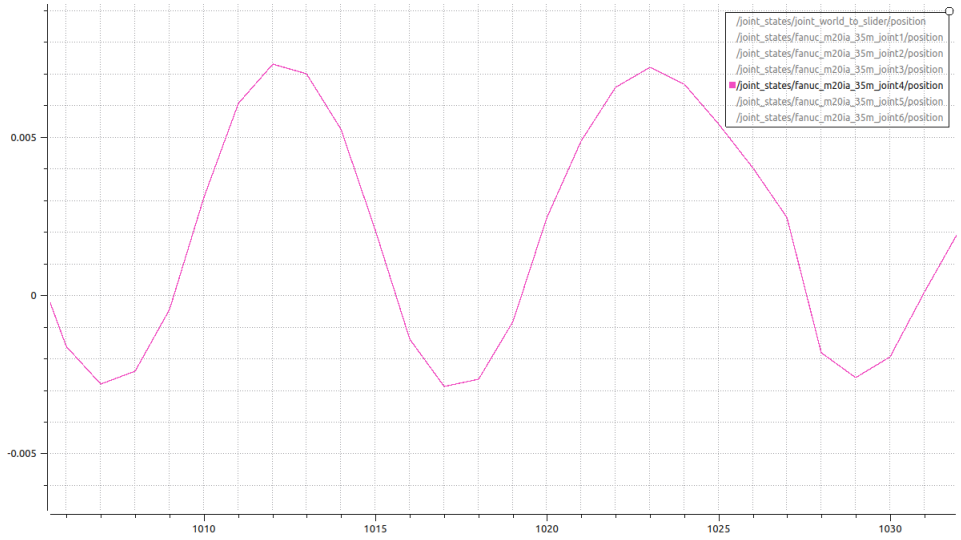


Figure 4.17: Evaluation of T_u in PlotJuggler for `fanuc_m20ia_35m_joint4`

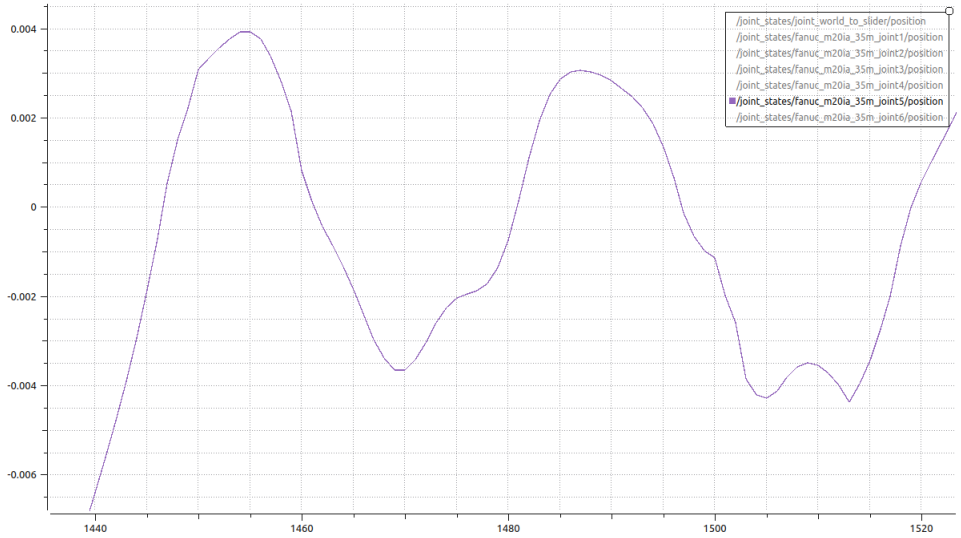


Figure 4.18: Evaluation of T_u in PlotJuggler for `fanuc_m20ia_35m_joint5`

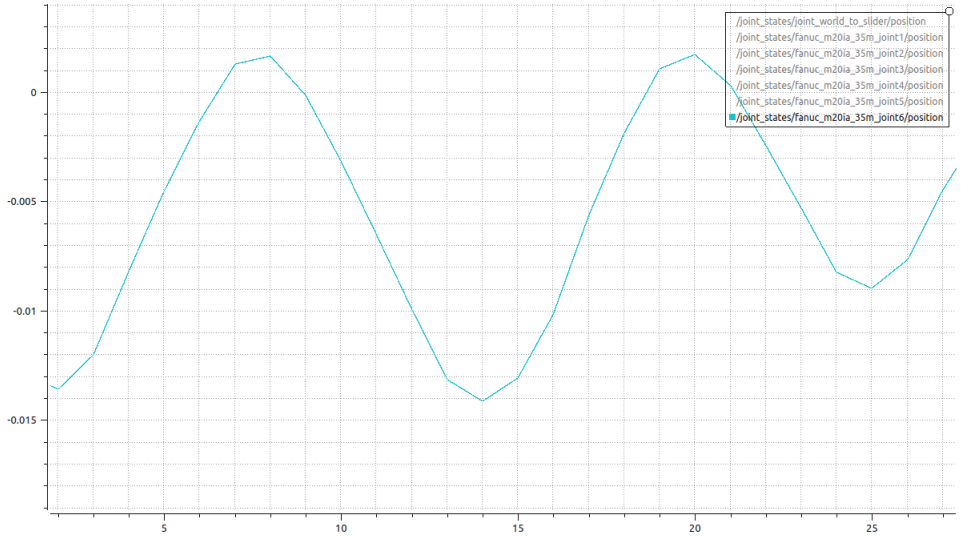
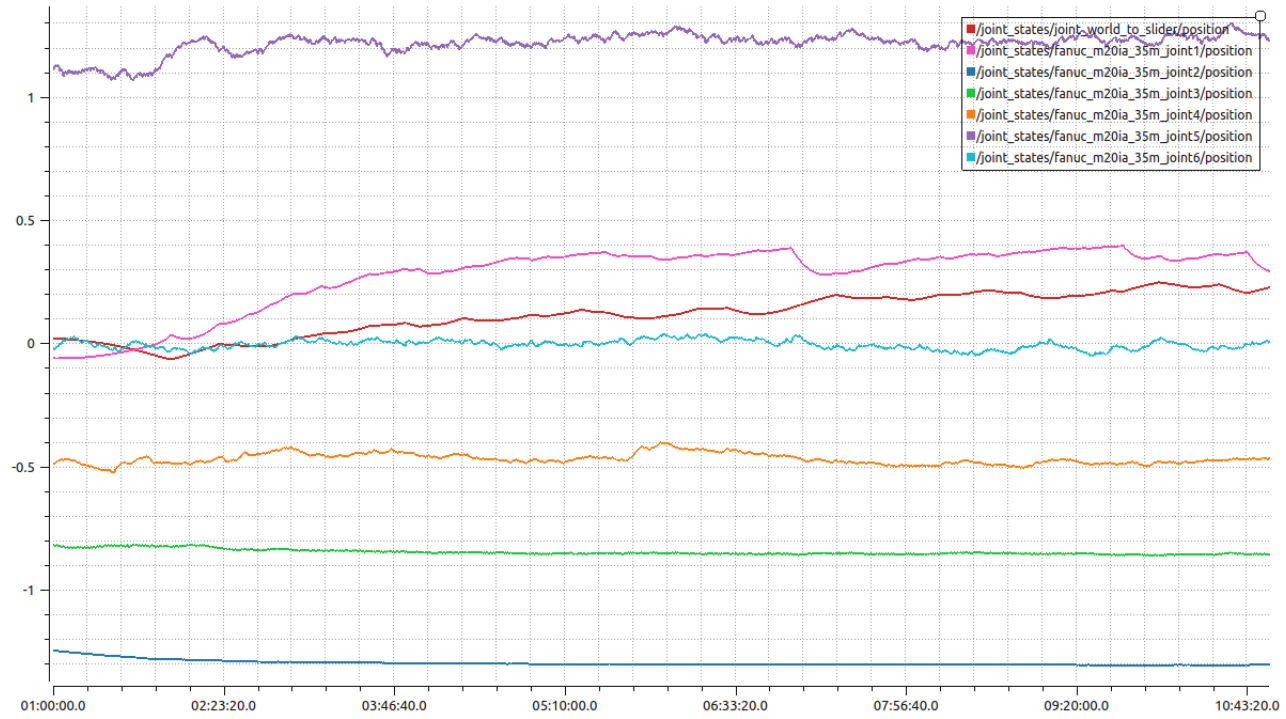
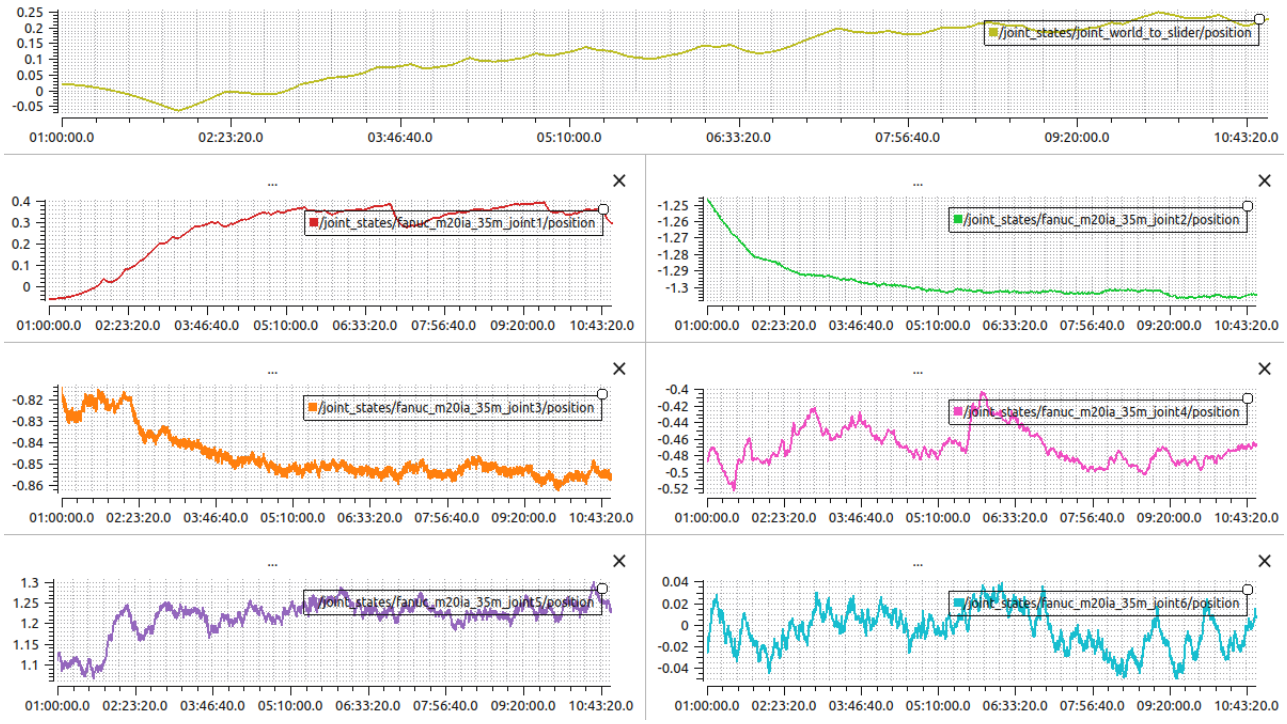


Figure 4.19: Evaluation of T_u in PlotJuggler for `fanuc_m20ia_35m_joint6`



(a) Overlapped view



(b) Separeted view

Figure 4.20: Evaluation of T_u in PlotJuggler (summary)

Joint name	Absolute time		Timestamp	
	From	To	From	To
joint_world_to_slider	887	1005	01:14:47	01:16:45
fanuc_m20ia_35m_joint1	1889	1919	01:31:29	01:31:59
fanuc_m20ia_35m_joint2	165	237	01:02:45	01:03:57
fanuc_m20ia_35m_joint3	294	305	01:04:54	01:05:05
fanuc_m20ia_35m_joint4	1012	1023	01:16:52	01:17:03
fanuc_m20ia_35m_joint5	1455	1487	01:24:15	01:24:47
fanuc_m20ia_35m_joint6	8	20	01:00:08	01:00:20

Table 4.3: Extraction of T_u value thanks to PlotJuggler

Joint	PID parameters		
	K_P	K_I	K_D
joint_world_to_slider	550200.0	9325.42	8115450.0
fanuc_m20ia_35m_joint1	566400.0	37760.0	2124000.0
fanuc_m20ia_35m_joint2	480000.0	13333.33	4320000.0
fanuc_m20ia_35m_joint3	2400000.0	436363.63	3300000.0
fanuc_m20ia_35m_joint4	239040.0	43461.82	328680.0
fanuc_m20ia_35m_joint5	460200.0	28762.5	1840800.0
fanuc_m20ia_35m_joint6	27204.0	4534.0	40806.0

Table 4.4: PID parameters used after Ziegler-Nichols

Finally, the gains (Tab. 4.4) were derived and were added in *ros2_controller_effort.yaml*. Unfortunately, due to the mutual coupling of the PID parameters, the detection of the oscillation periods cannot be obtained analytically as the oscillations in the terminal joints of the chain are strongly influenced by the oscillations of the joints at the base of our system. As it can be see in Gazebo there is an abnormal behavior (Fig. 4.21).

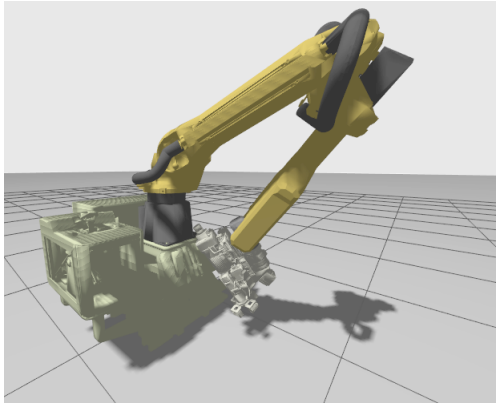
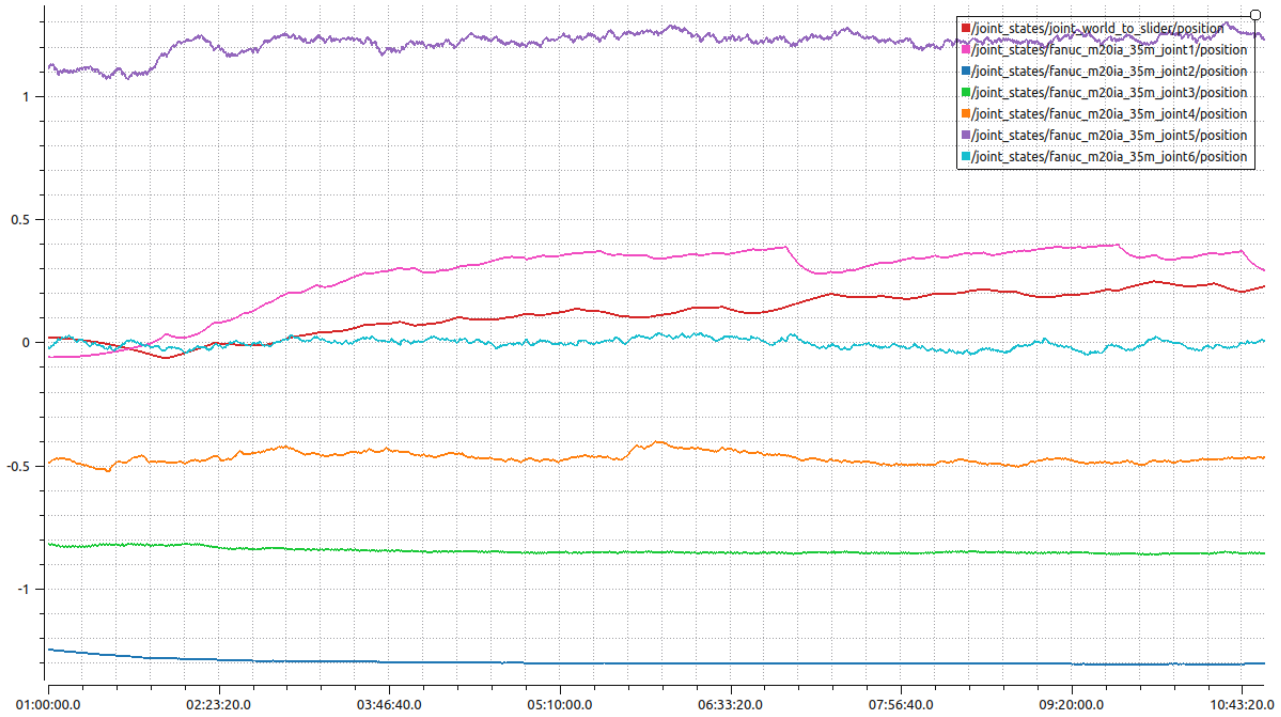
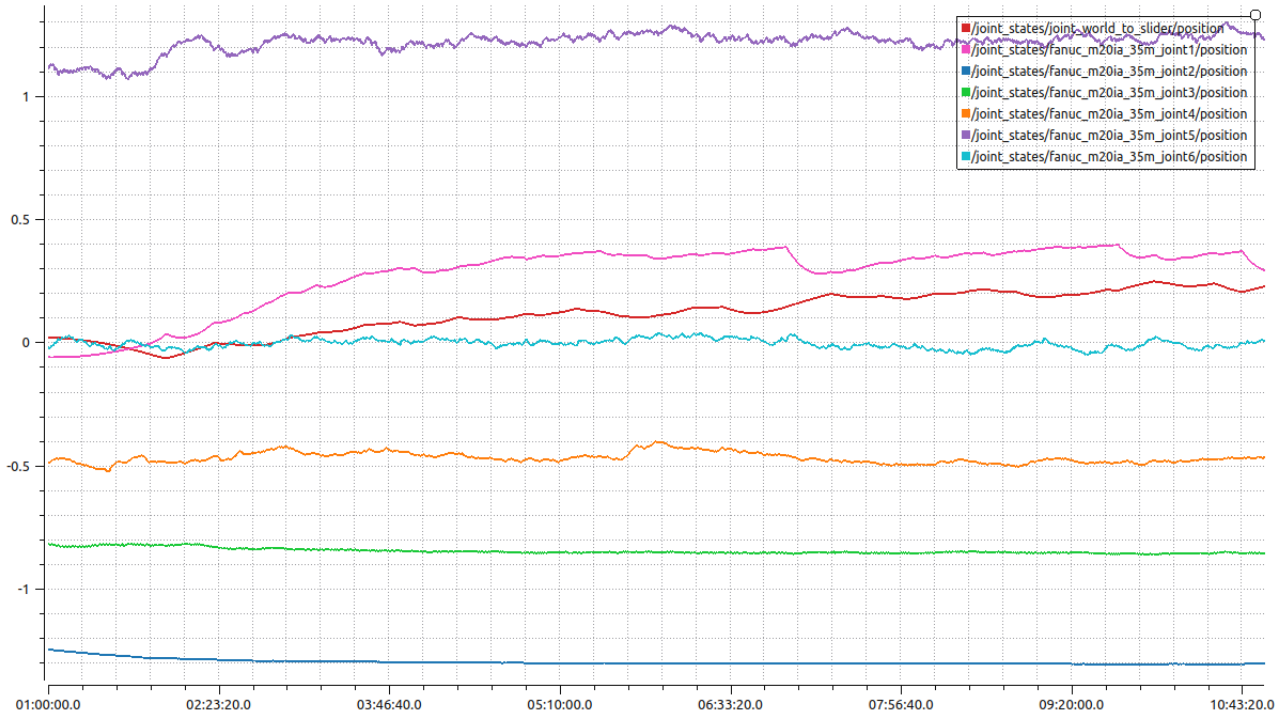


Figure 4.21: Undesidered behavior after Ziegler-Nichols in Gazebo

The consequences of these incorrect gains can als be highlighted using PlotJuggler (Fig. 4.22). Also for this case, the simulations (both dataframe and statistics) were saved in CSV format as follows: *after_tuning_csv_statistics_final.csv* and *after_tuning_csv_datarange_final.csv*.



(a) Overlapped view



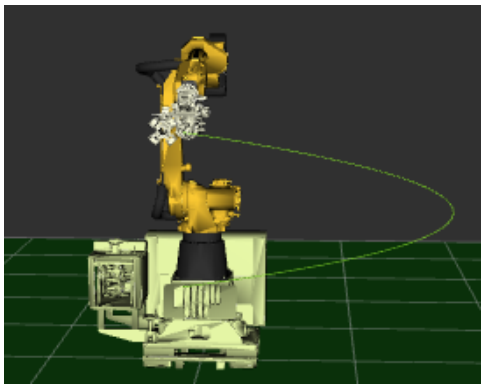
(b) Separeted view

Figure 4.22: Undesidered behavior after Ziegler-Nichols in PlotJuggler (summary)

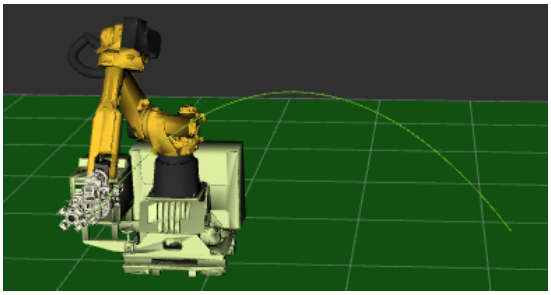
Parameters	Rise time			Steady state error	Stability
K_P	Decrease	Increase	Small change	Decrease	Decrease
K_I	Decrease	Increase	Increase	Eliminate	Decrease
K_D	Minor changes	Decrease	Decrease	No effect	Improve if K_D is small

Table 4.5: Effects of K_P,K_I and K_D on output response

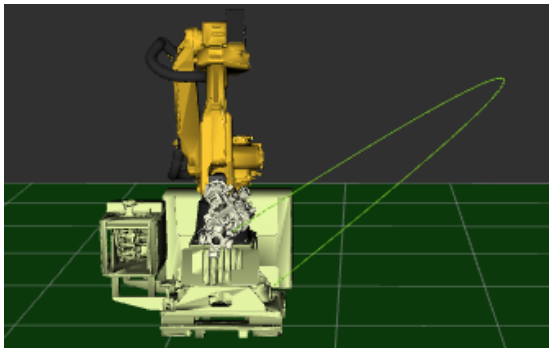
To resolve this problem, following Tab. 4.5 the obtained gains were modified. In particular, to remedy such unwanted behaviors, an empirical approach to calibration was followed, based on a trial and error philosophy starting from existent previously calculated gains. In this regard, 4 reference parabolas (Fig. 4.23) were considered, described obviously within a bag file and which satisfy the path length requirements.



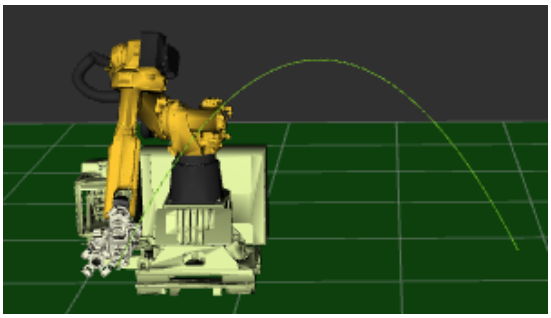
(a) *basic.db3*



(b) *horizontal_low.db3*



(c) *yrotateplane.db3*



(d) *zrotateplane.db3*

Figure 4.23: Four parabolas considered during PID tuning phase in RViz

After the tuning, the obtained performances were evaluated in terms of settling time, overshoot and obviously verifying that there were no persistent oscillations at steady state. Furthermore, to take into account the physics of the robot, a settling time near the 1 second of each joint position was considered as metric.

Once the set of optimal gains has been identified, they were further verified using the motion planner in RViz: in addition to verifying the correct execution of parabolas, generic trajectories in the joint space were planned and the results were again compared in PlotJuggler.

The optimal gains obtained are shown in Tab. 4.6. It is important to point out that the values

obtained each time were updated for one of the 4 reference trajectories considered and then verified again. Furthermore, the sizing of the gains for each joint always took into account the mass and inertia of each joint, in particular the impact it has on the other joints in our system. Also for this case, the simulations (both dataframe and statistics) were saved in CSV format as follows: *after_corrections_csv_statistics_final.csv* and *after_corrections_csv_datarange_final.csv*. The results of the correct parameters are shown in Fig. 4.25.

Joint	**PID parameters**		
	K_P	K_I	K_D
`joint_world_to_slider`	**190000.0**	**8000.0**	**5000.0**
`fanuc_m20ia_35m_joint1`	**350000.0**	**20000.0**	**10000.0**
`fanuc_m20ia_35m_joint2`	**300000.0**	**8000.0**	**10000.0**
`fanuc_m20ia_35m_joint3`	**800000.0**	**10000.0**	**10000.0**
`fanuc_m20ia_35m_joint4`	**5000.0**	**200.0**	**100.0**
`fanuc_m20ia_35m_joint5`	**100000.0**	**5000.0**	**150.0**
`fanuc_m20ia_35m_joint6`	**2000.0**	**150.0**	**70.0**

Table 4.6: PID parameters used after the simulations

Note that, in order to modify dinamically the PID values, a useful tool provided by the `ros2_rqt` environment was used, which is `rqt_reconfigure` that allows to modify the parameters of generic nodes of the `ROS2` environment. In particular, the `rqt_reconfigure` tool was used to modify the PID parameters of the `ros2_controller_effort.yaml` file:

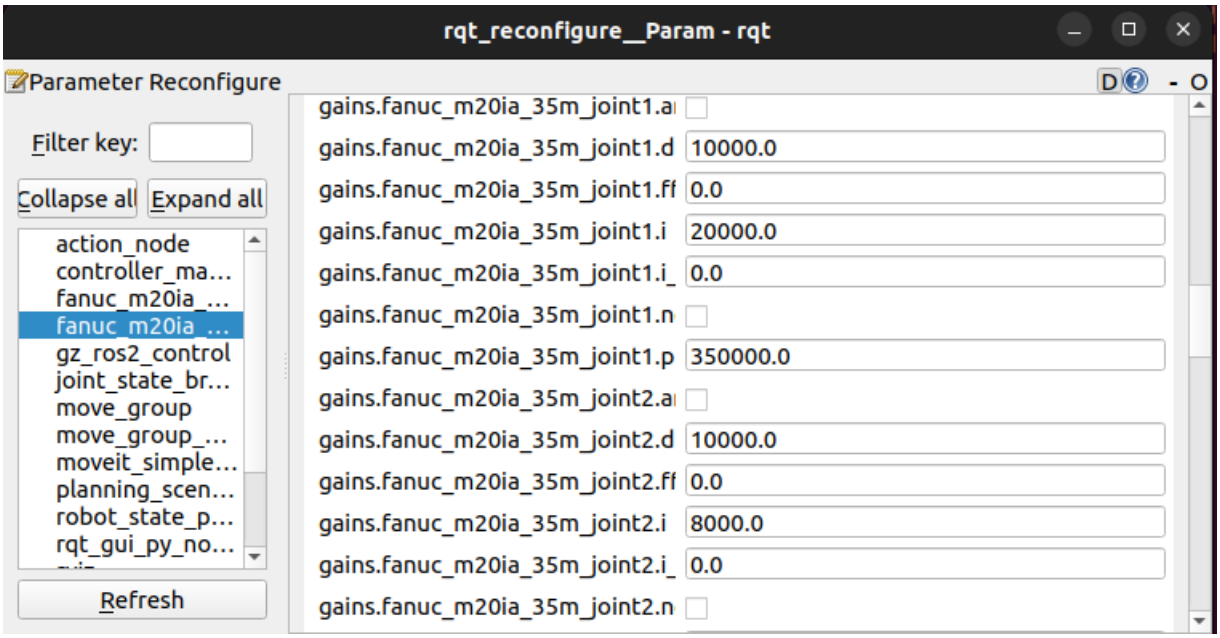
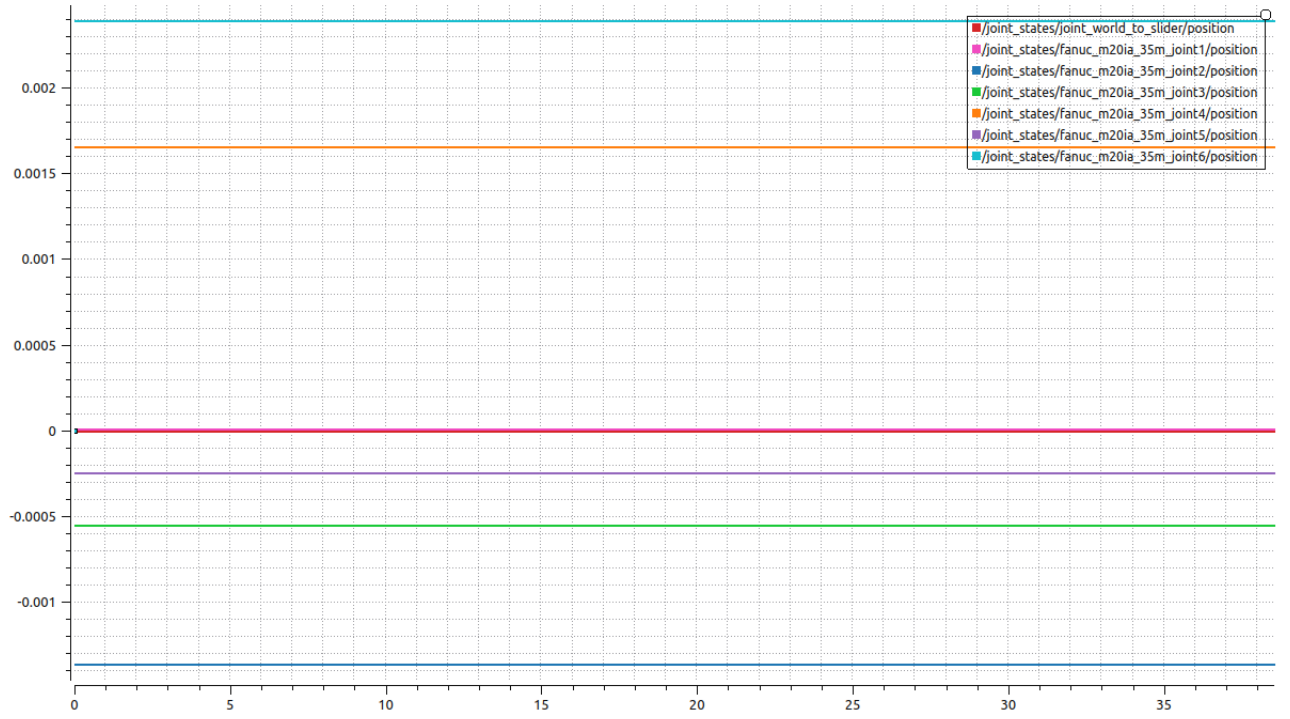
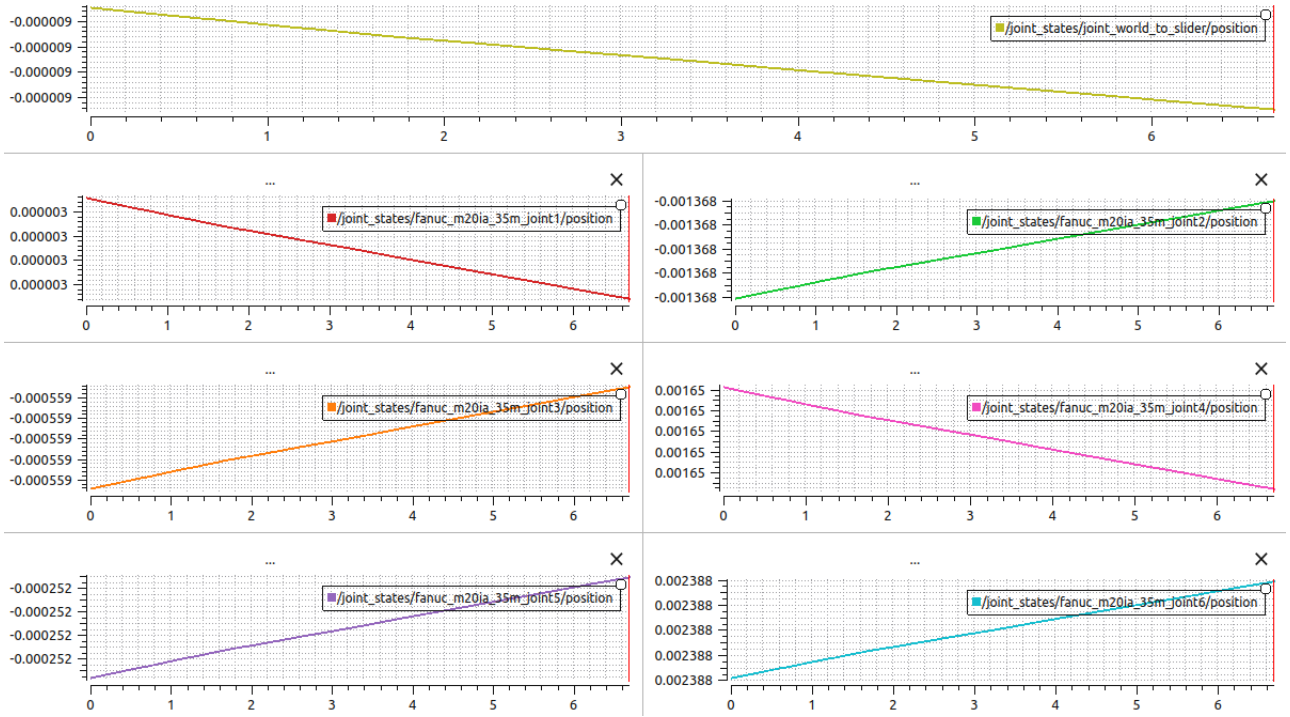


Figure 4.24: Tool used to modify the PID parameters



(a) Overlapped view



(b) Separeted view

Figure 4.25: Running a new simulation the start position is finally correct. This can be confirmed in PlotJuggler (summary)

At this point, after having updated the *ros2_controller_effort.yaml* file again, it is possible to simulate the 4 trajectories considered. These example trajectories were generated offline and saved in bag files. At the end of the simulations the CSV file of the simulation obtained is saved following this format: `csv_<datarange/statistics>_<name_trajectory>`.

4.1.4 Simulations in Gazebo

After tuning the PID parameters, some simulations in Gazebo were performed. In particular, the 4 previously mentioned parabolas were simulated alongside other 4 that will be dealt with in 4.2.1, and multiple checks on the performances of the robot in terms of settling time, overshoot and steady state error were done. These simulations were also used to validate and tune the PID parameters, and to verify that the robot was able to execute the parabolas well. The need for using Ignition Gazebo lies exactly in the fact that it allows for dynamic simulations that take into account real factors and forces that can affect the robot during its movements. Although RViz is involved in the execution of the simulations and the integration of the Ignition Gazebo component in the environment, the trajectories that can be seen in RViz are only the planned (or "desired") ones, as the true dynamics of the robot are dealt with in Ignition Gazebo and the conditions of the robot there are then reflected in the RViz display of the robot.

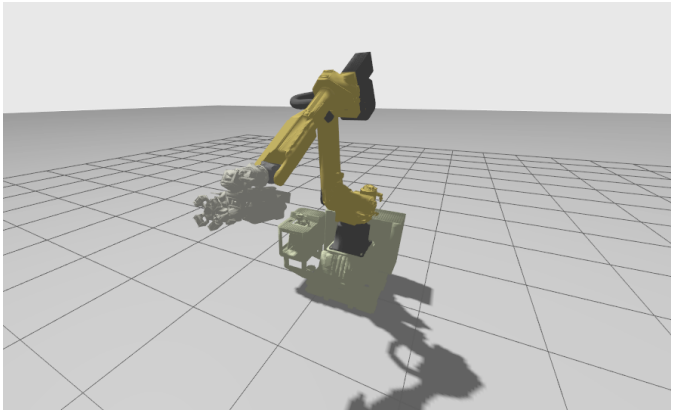


Figure 4.26: Fanuc robot inside the Ignition Gazebo environment, executing the *tight_left* trajectory.

Let's start with *basic.db3*. The results obtained are shown in Fig. 4.27 4.28 4.29 4.30 4.31 4.32 4.33 4.34.

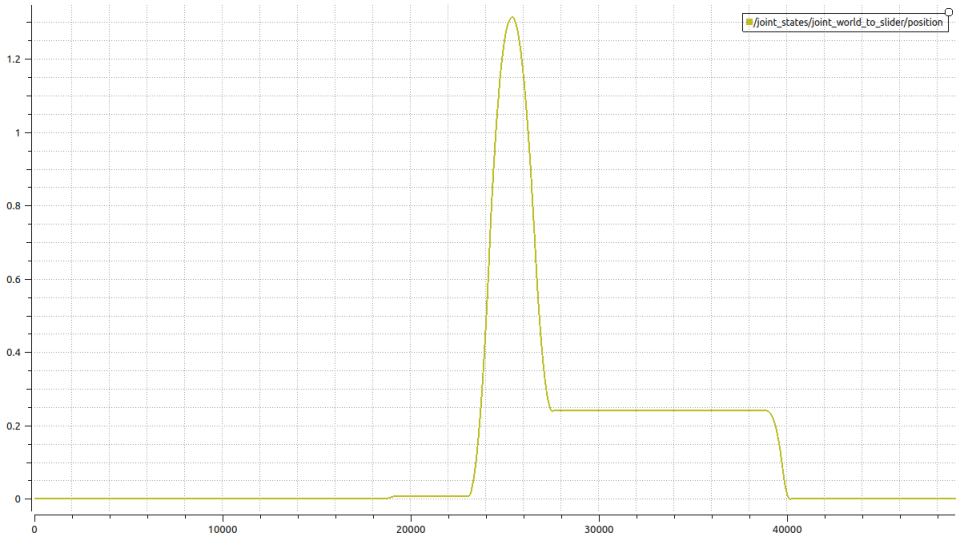


Figure 4.27: *basic.db3*: Gazebo simulation results in PlotJuggler for `joint_world_to_slider`

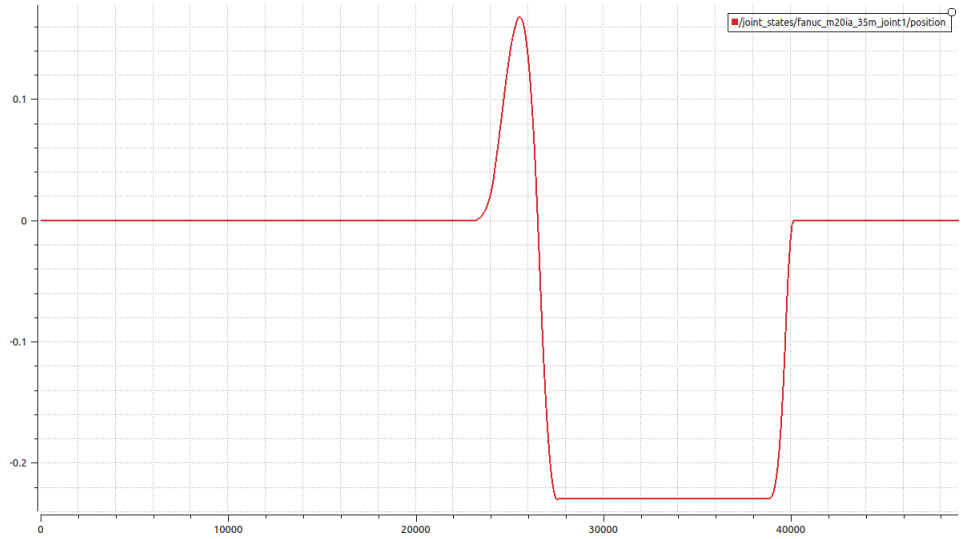


Figure 4.28: *basic.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint1`

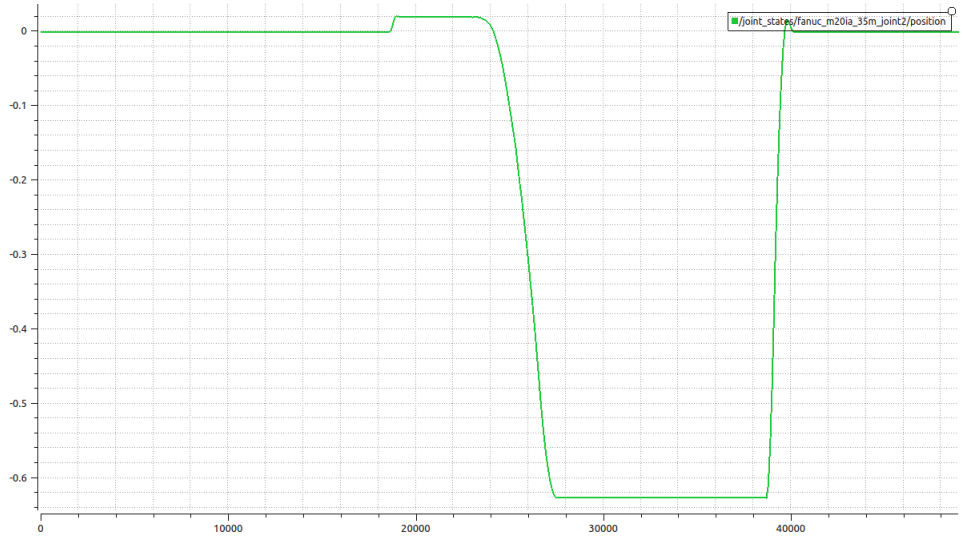


Figure 4.29: *basic.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint2`

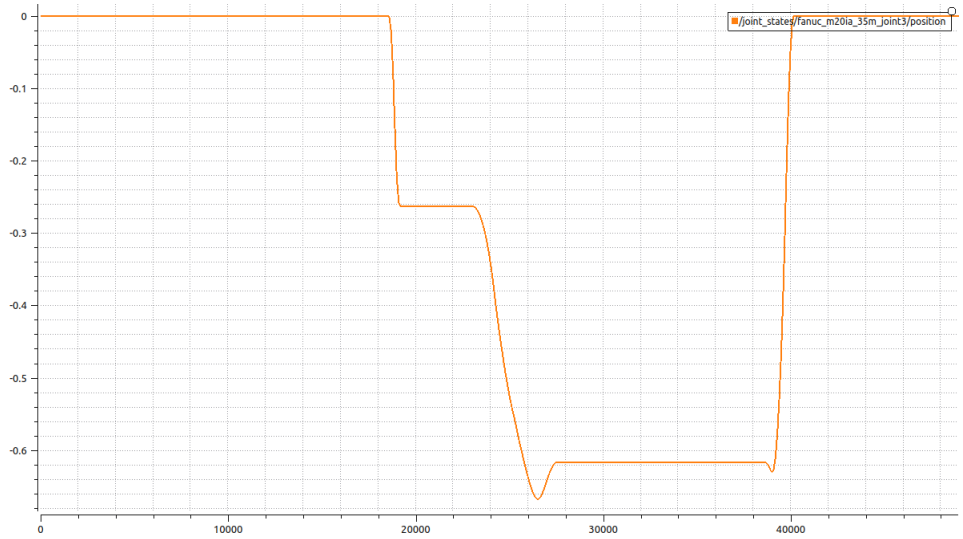


Figure 4.30: *basic.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint3`

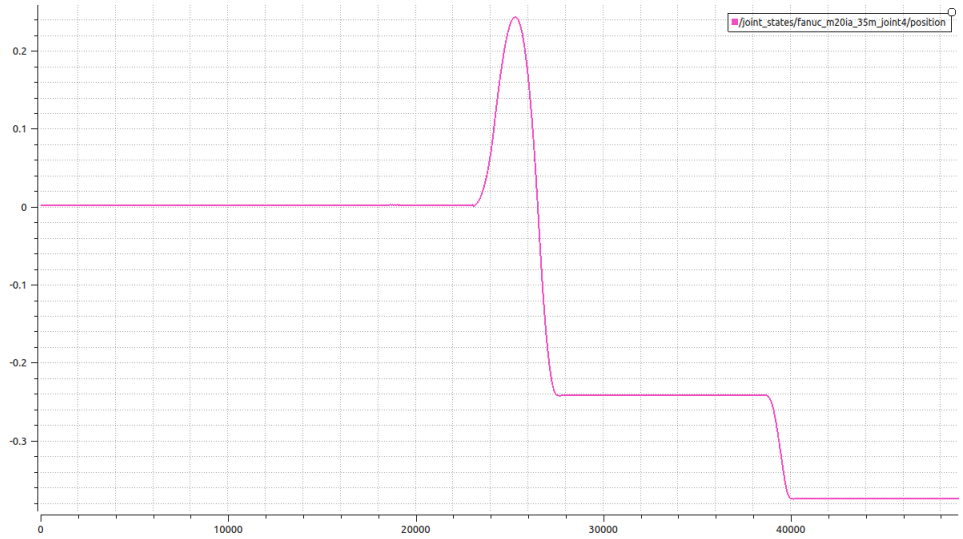


Figure 4.31: *basic.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint4`

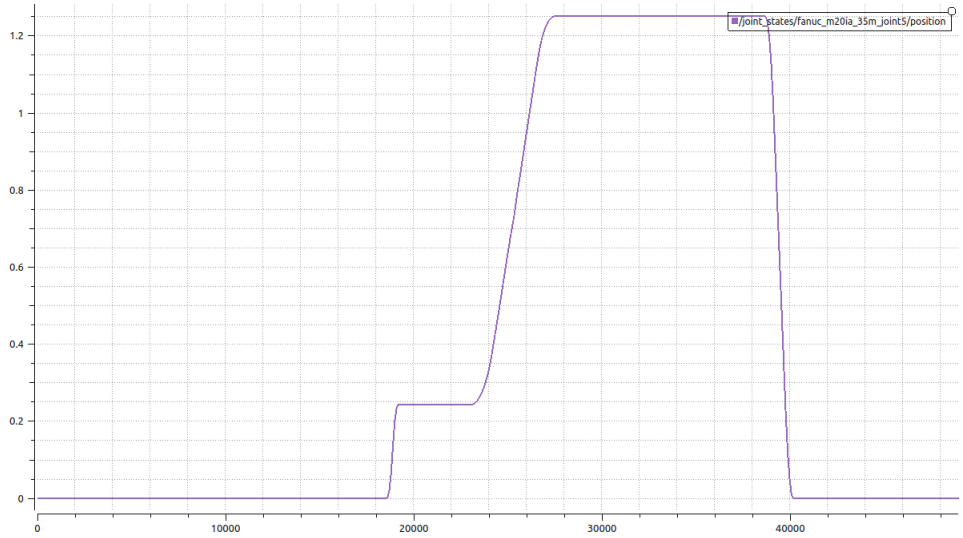


Figure 4.32: *basic.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint5`

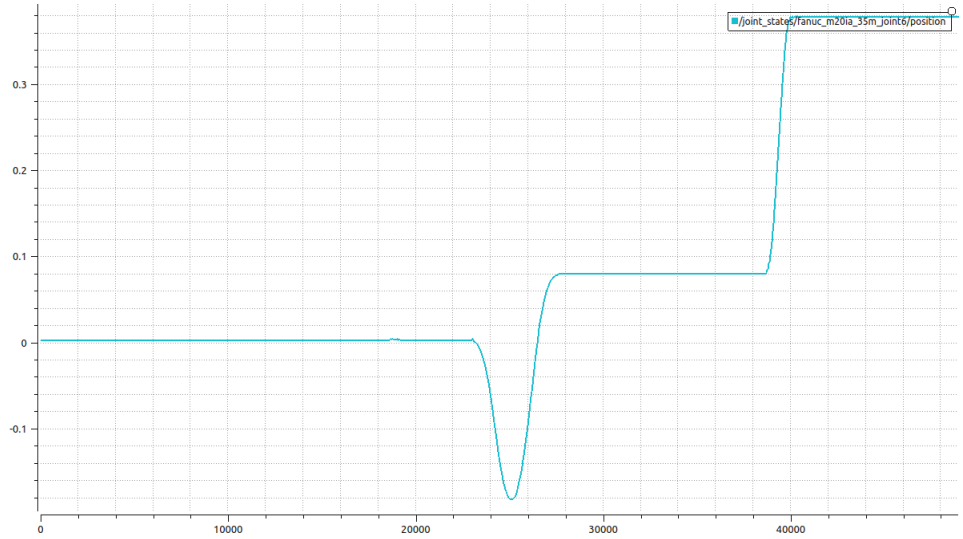
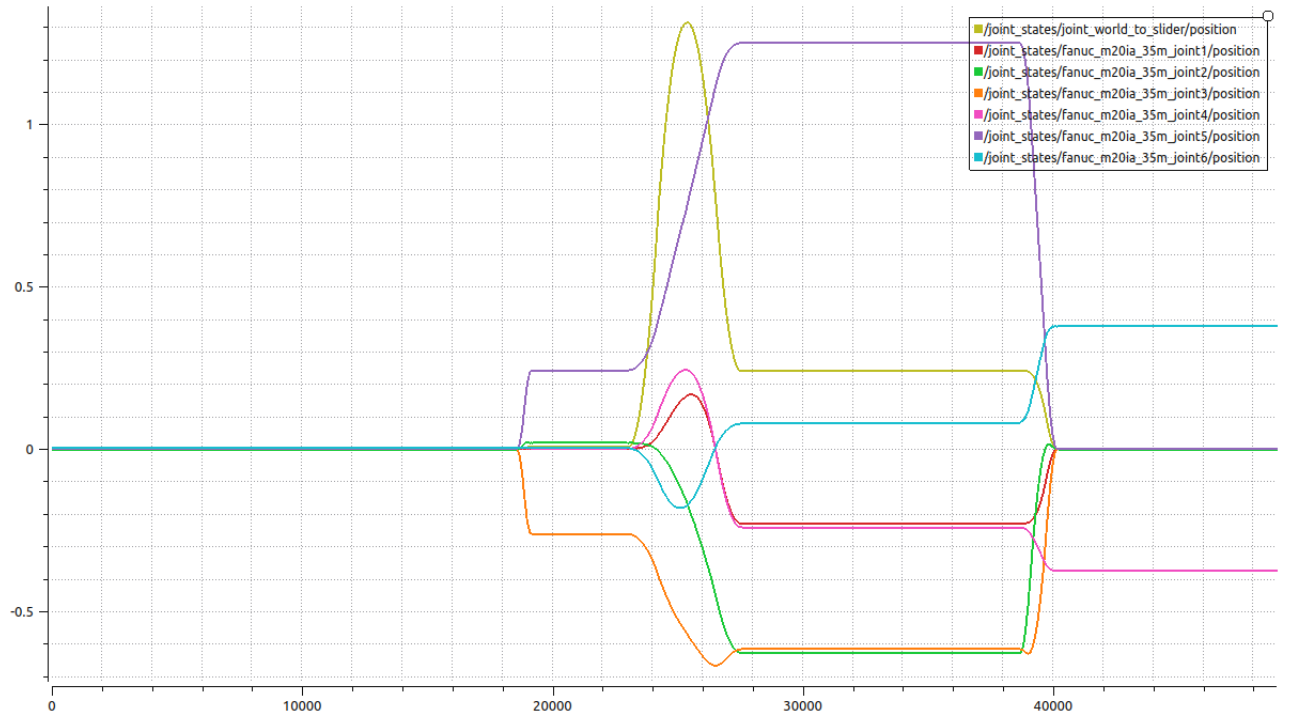
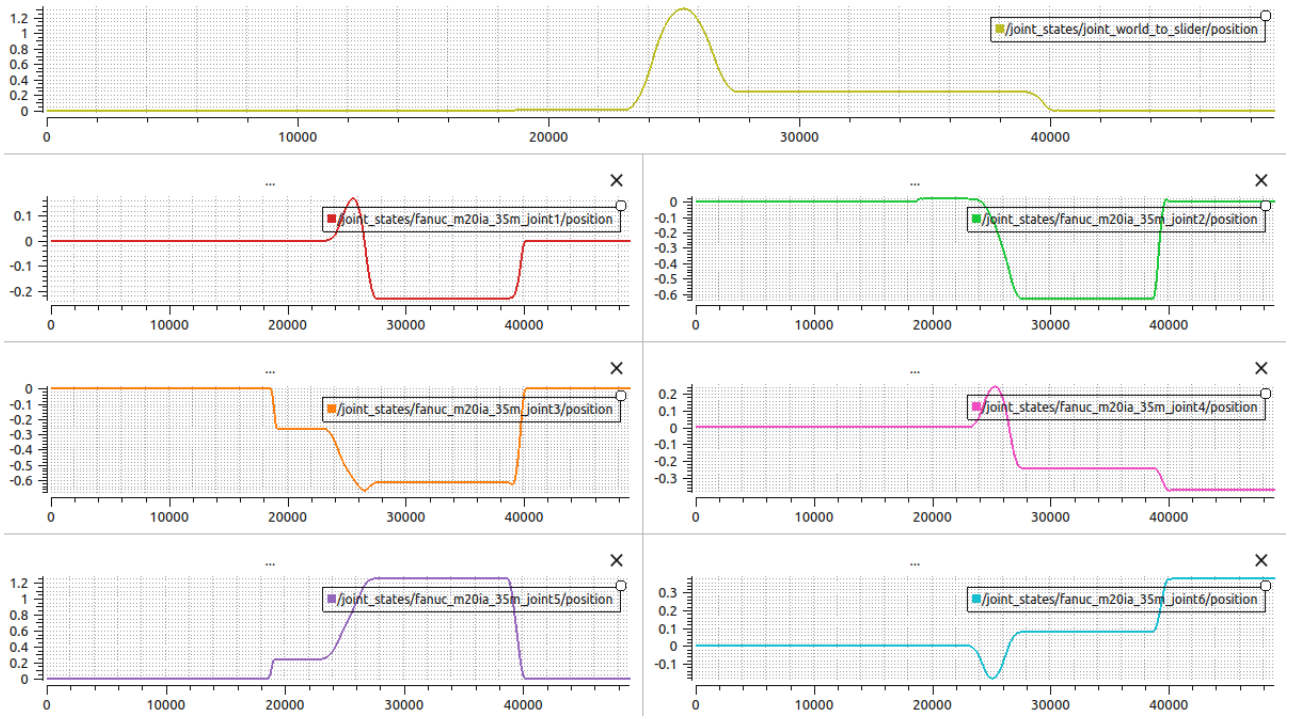


Figure 4.33: *basic.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint6`



(a) Overlapped view



(b) Separeted view

Figure 4.34: *basic.db3*: Gazebo simulation results in PlotJuggler (summary)

Let's move on to *horizontal_low.db3*. The results obtained are shown in Fig. 4.35 4.36 4.37 4.38 4.39 4.40 4.41 4.42.

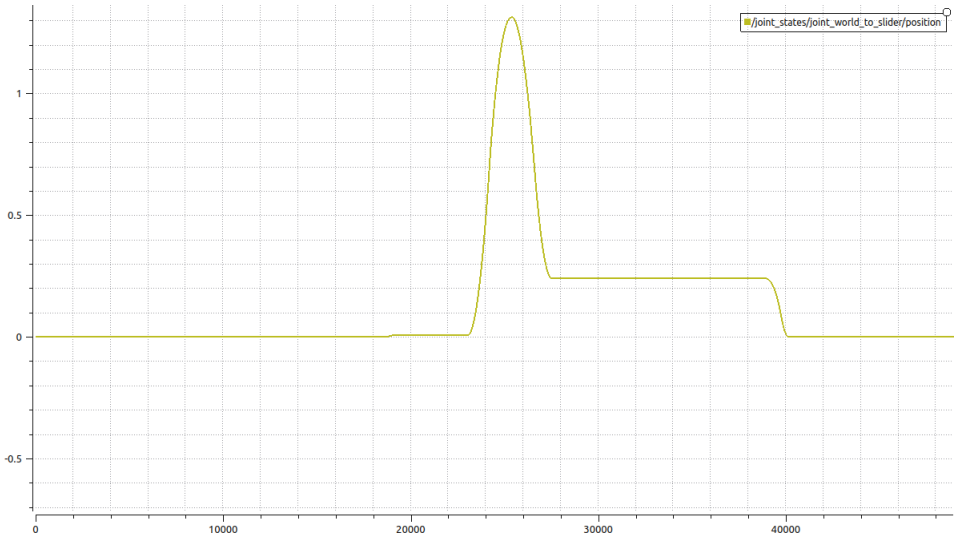


Figure 4.35: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler for `joint_world_to_slider`

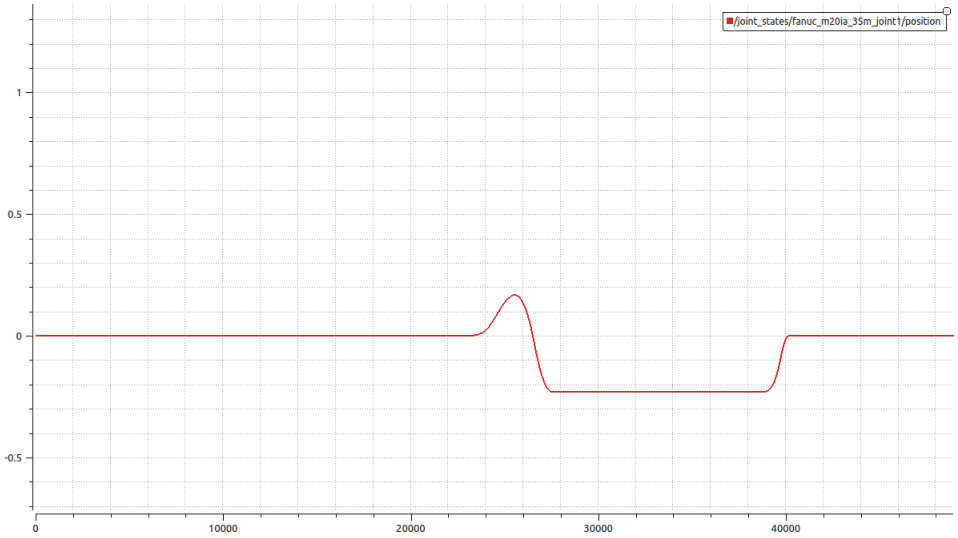


Figure 4.36: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint1`

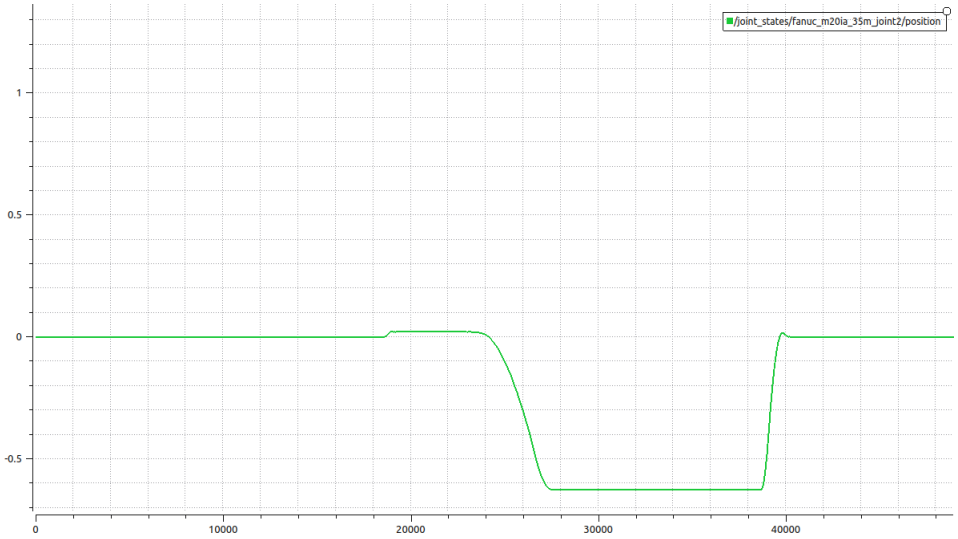


Figure 4.37: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint2`

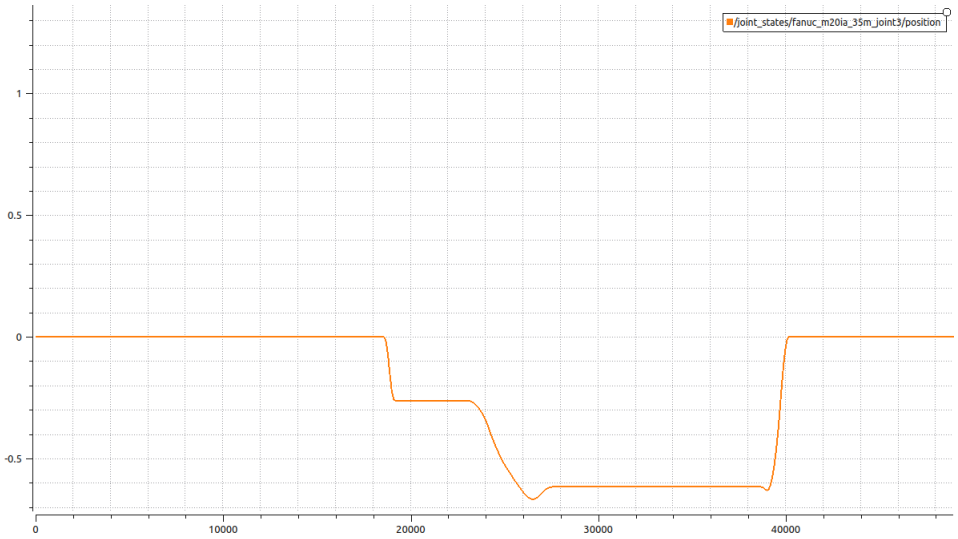


Figure 4.38: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint3`

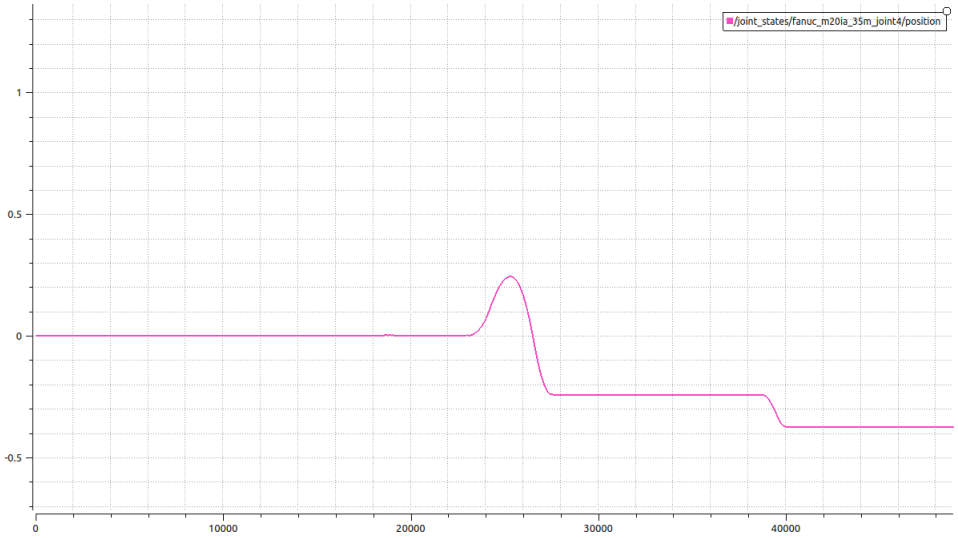


Figure 4.39: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint4`

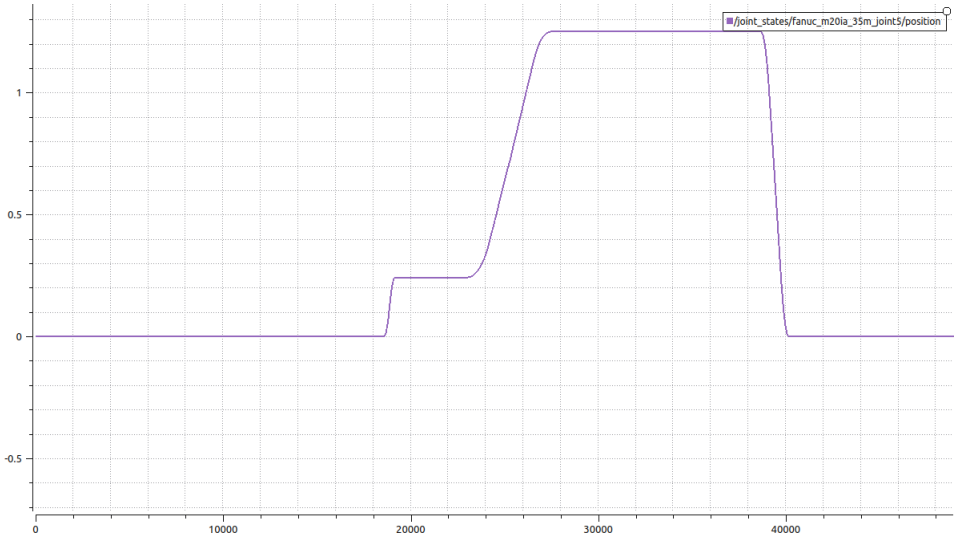


Figure 4.40: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint5`

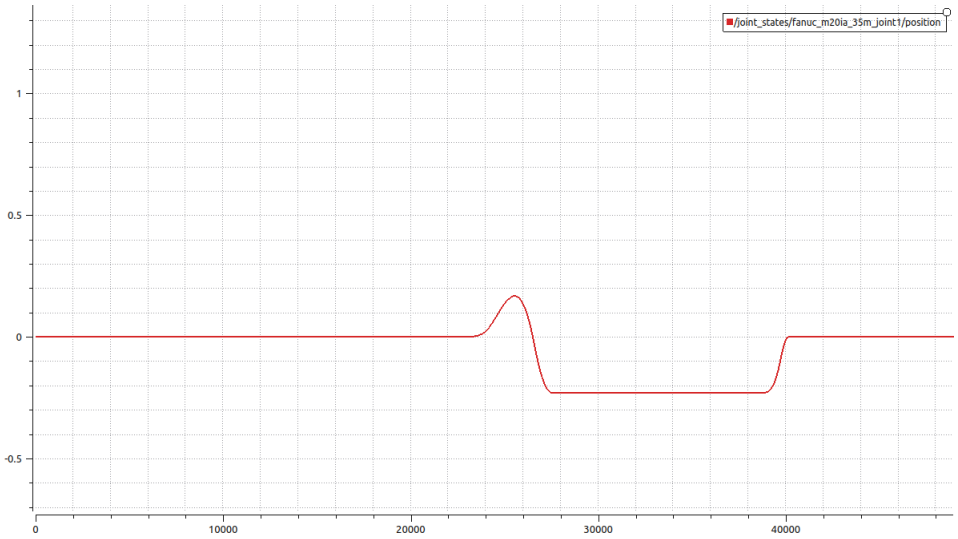
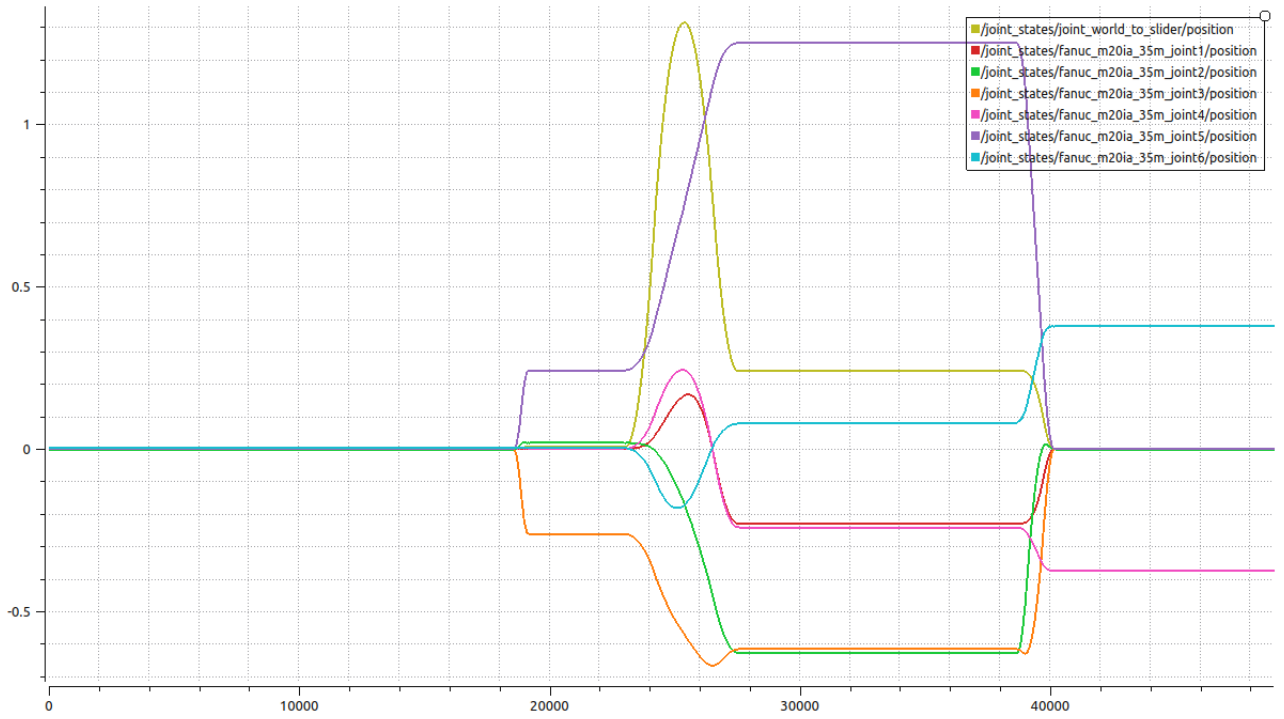
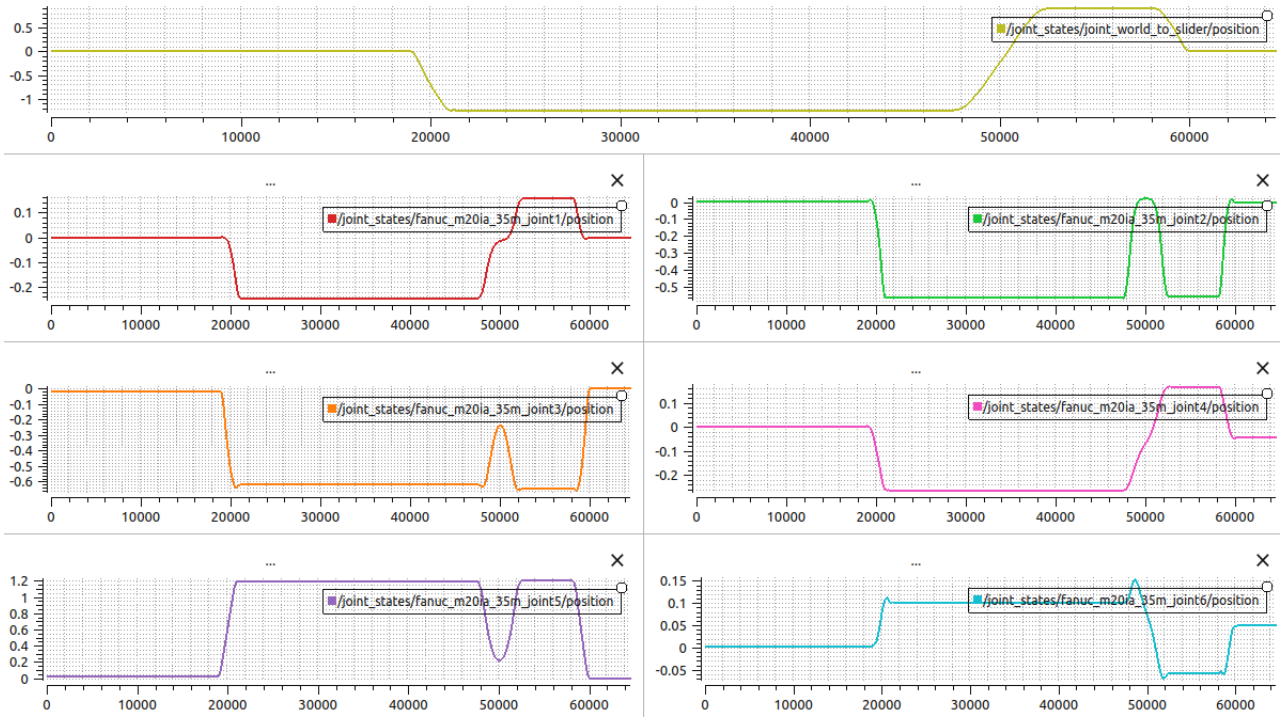


Figure 4.41: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint6`



(a) Overlapped view



(b) Separeted view

Figure 4.42: *horizontal_low.db3*: Gazebo simulation results in PlotJuggler (summary)

Let's move on to *yrotatedplane.db3*. The results obtained are shown in Fig. 4.43 4.44 4.45 4.46 4.47 4.48 4.49 4.50.

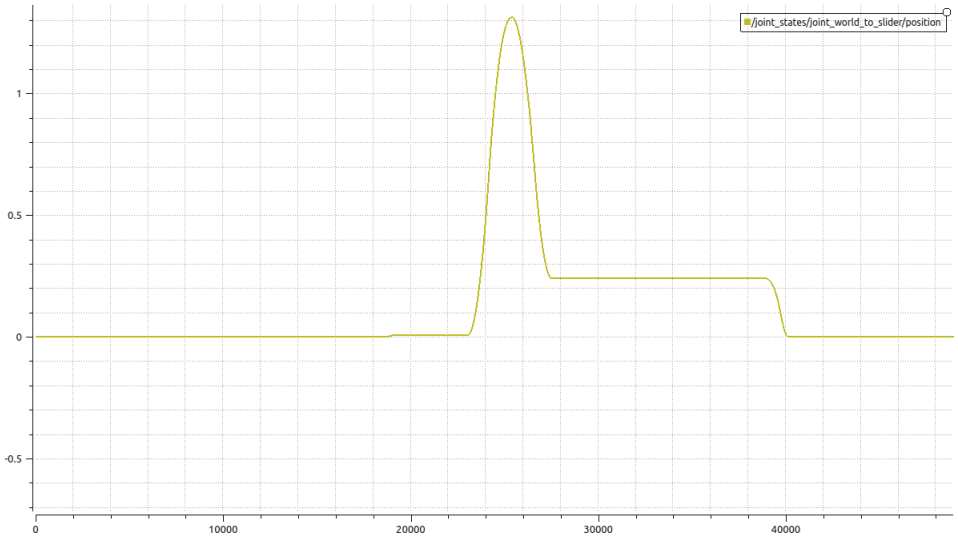


Figure 4.43: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `joint_world_to_slider`

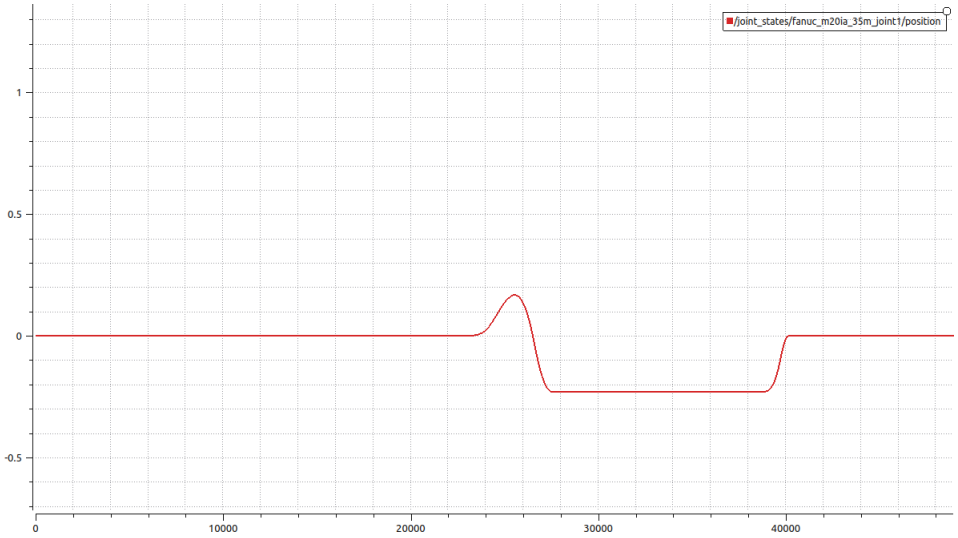


Figure 4.44: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint1`

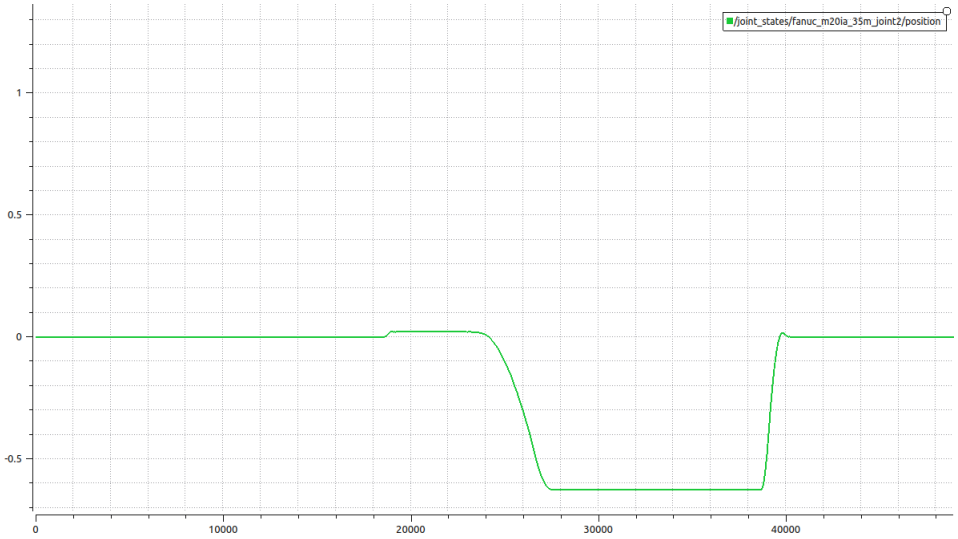


Figure 4.45: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint2`

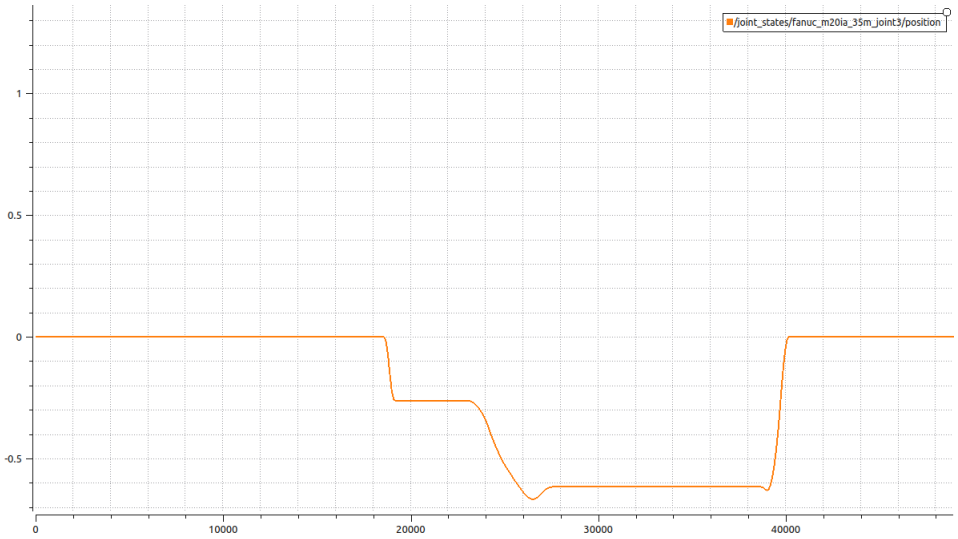


Figure 4.46: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint3`

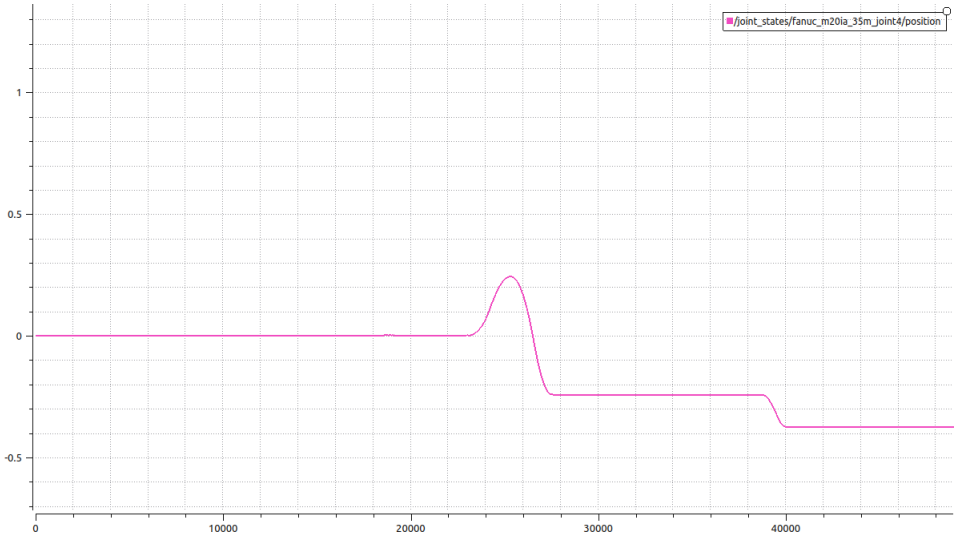


Figure 4.47: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint4`

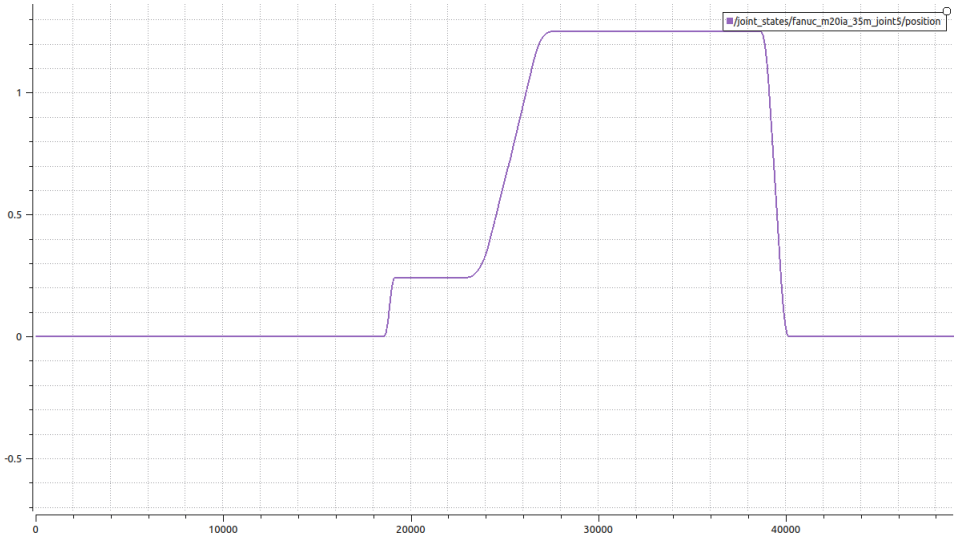


Figure 4.48: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint5`

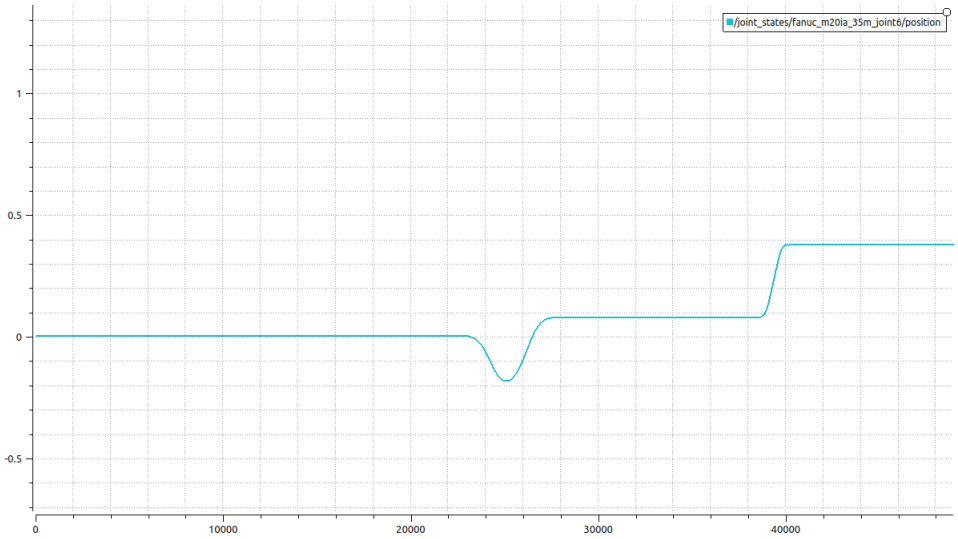
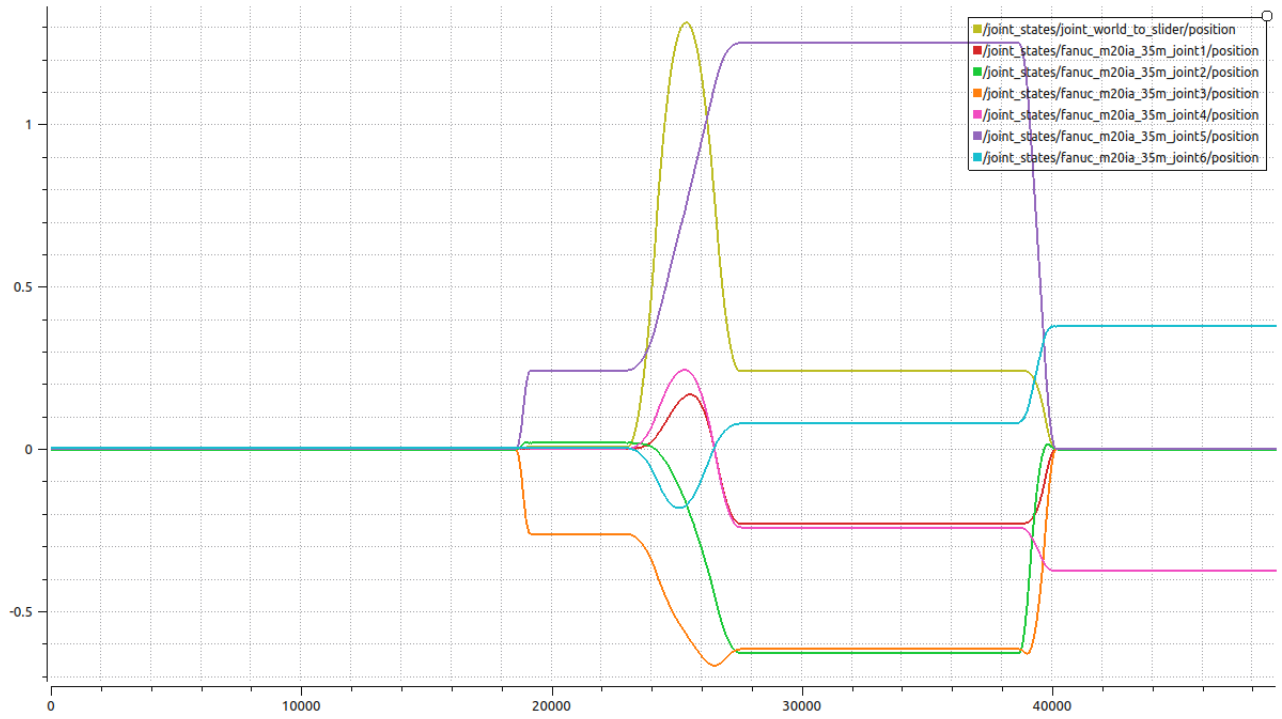
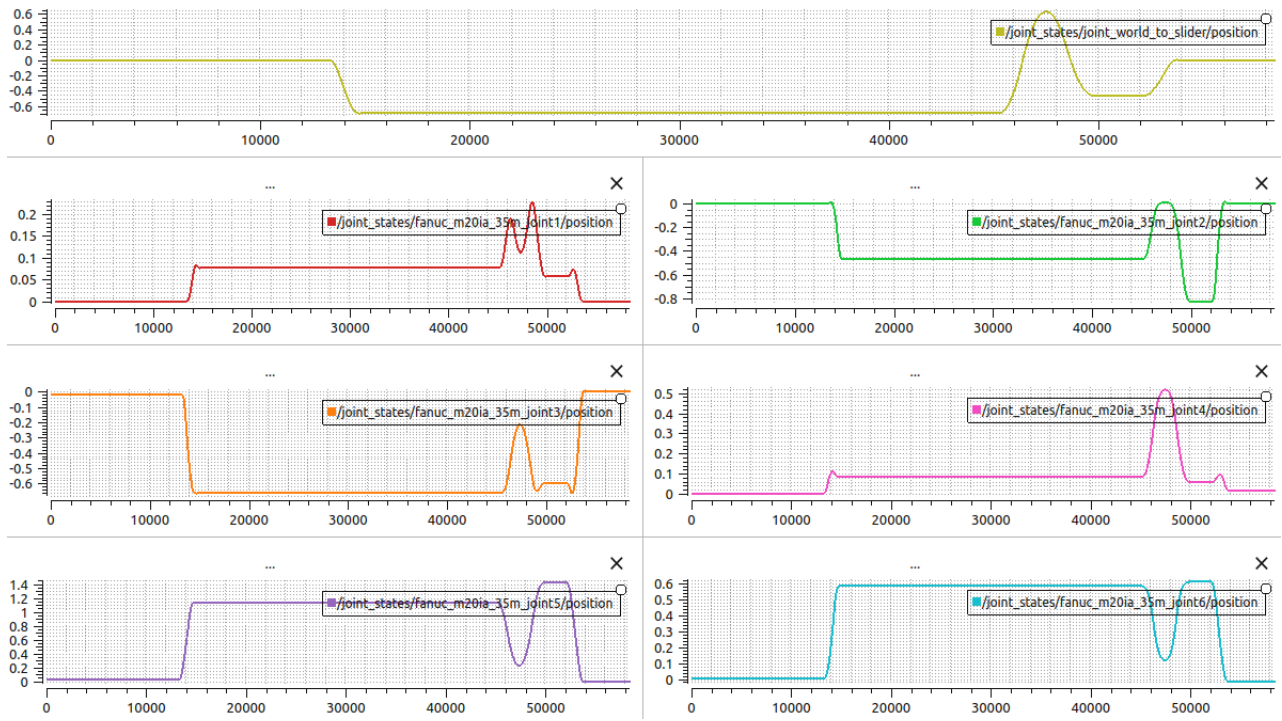


Figure 4.49: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint6`



(a) Overlapped view



(b) Separeted view

Figure 4.50: *yrotatedplane.db3*: Gazebo simulation results in PlotJuggler (summary)

Let's move on to *zrotatedplane.db3*. The results obtained are shown in Fig. 4.51 4.52 4.53 4.54 4.55 4.564.57 4.58.

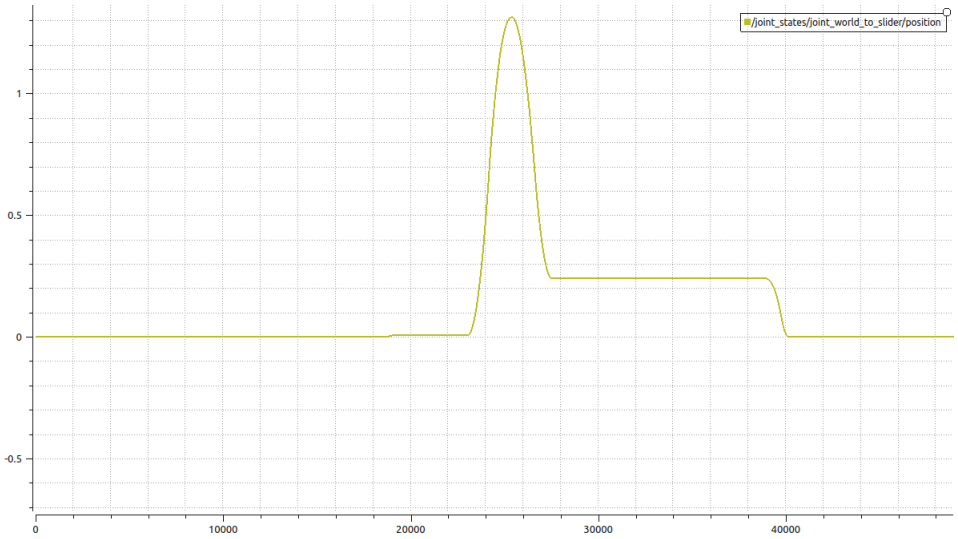


Figure 4.51: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `joint_world_to_slider`

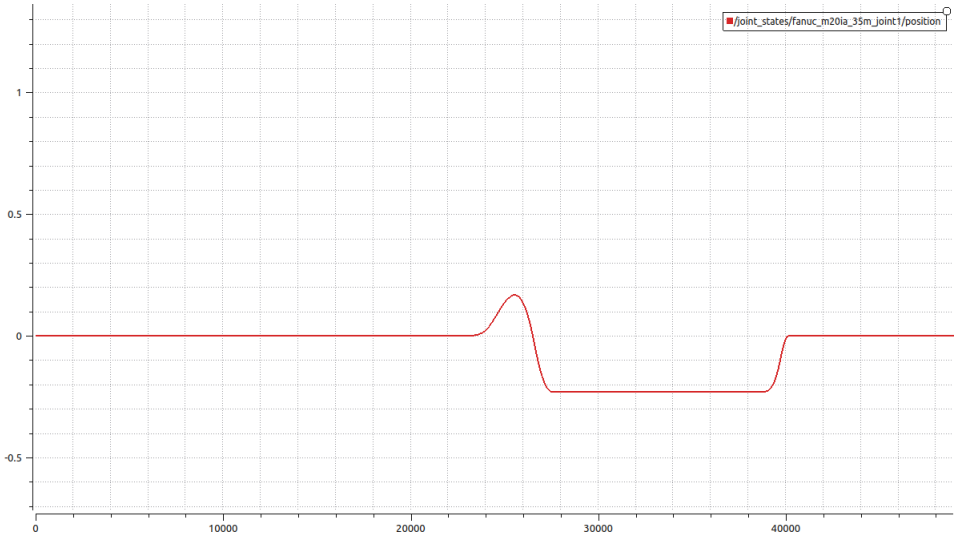


Figure 4.52: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint1`

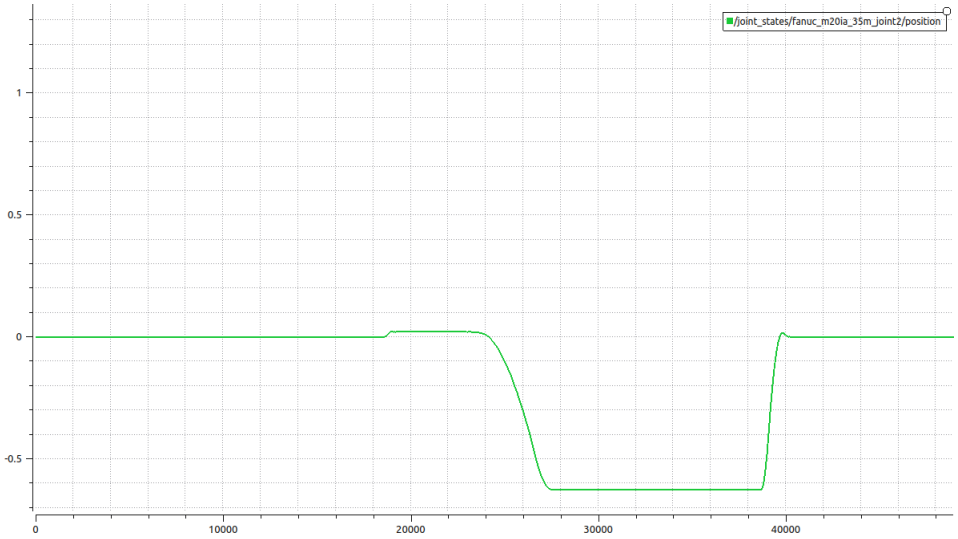


Figure 4.53: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint2`

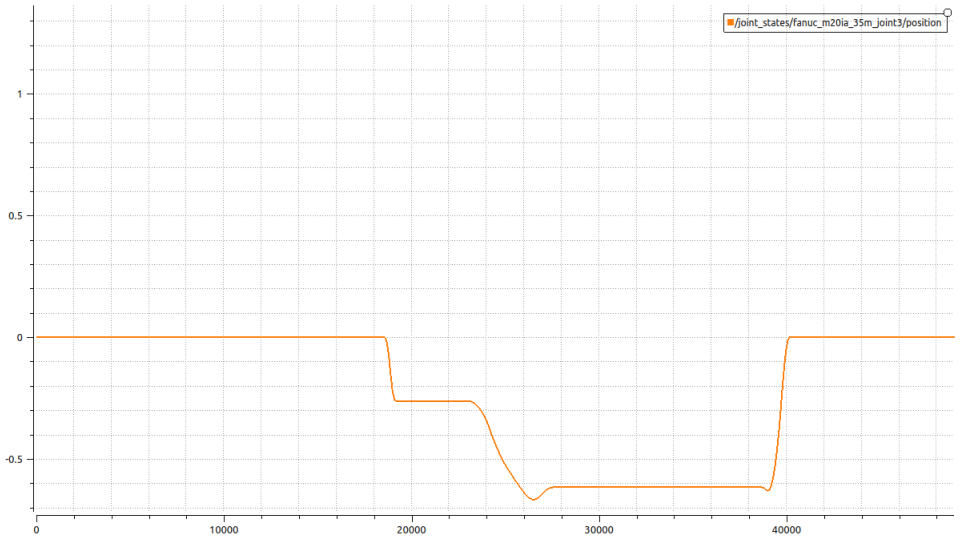


Figure 4.54: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint3`

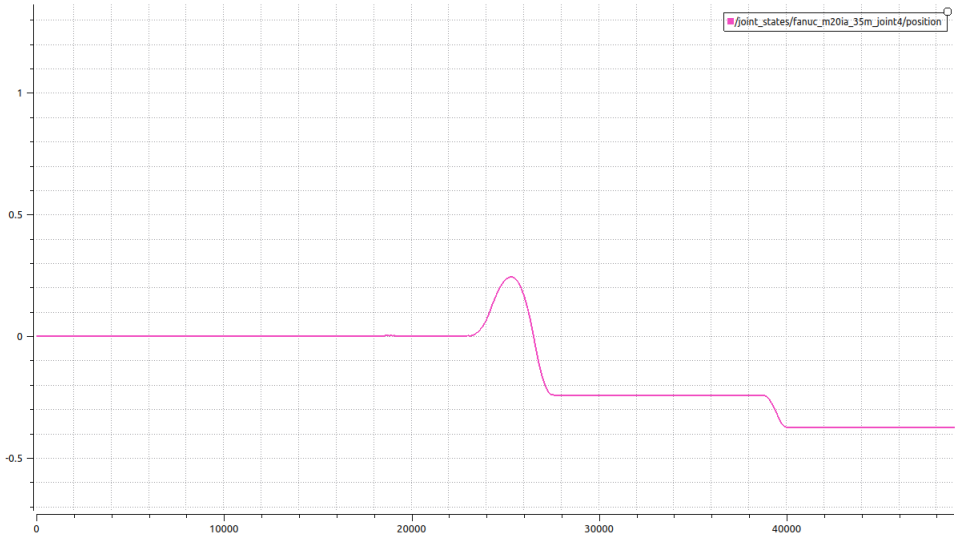


Figure 4.55: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint4`

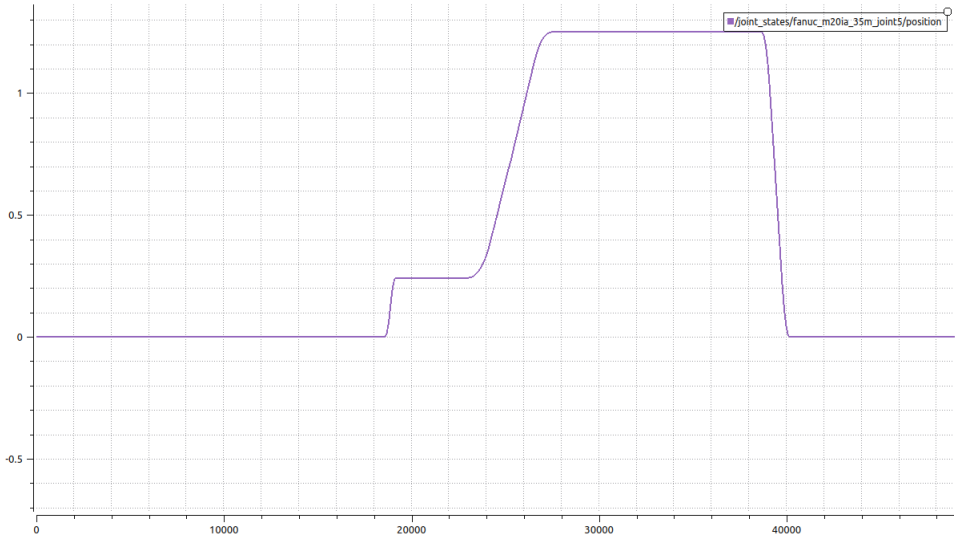


Figure 4.56: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint5`

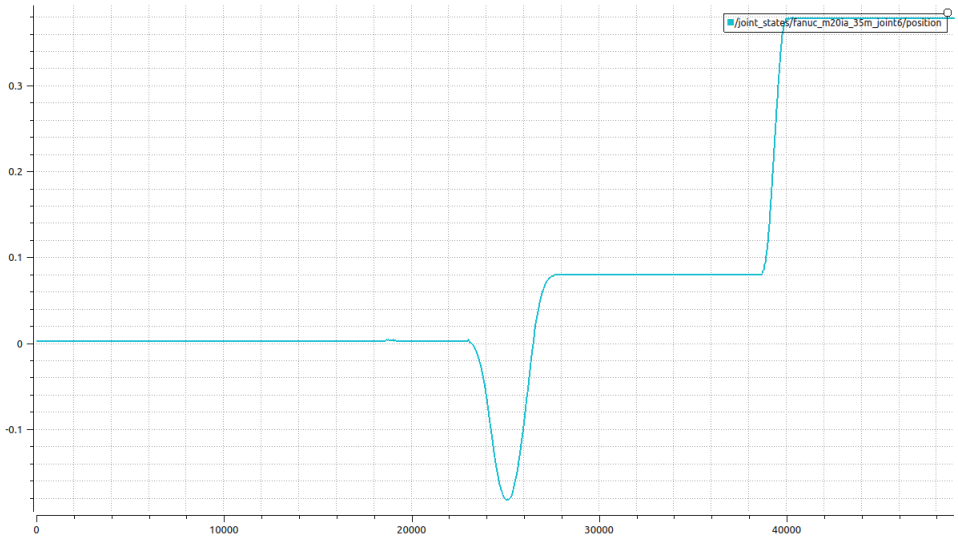
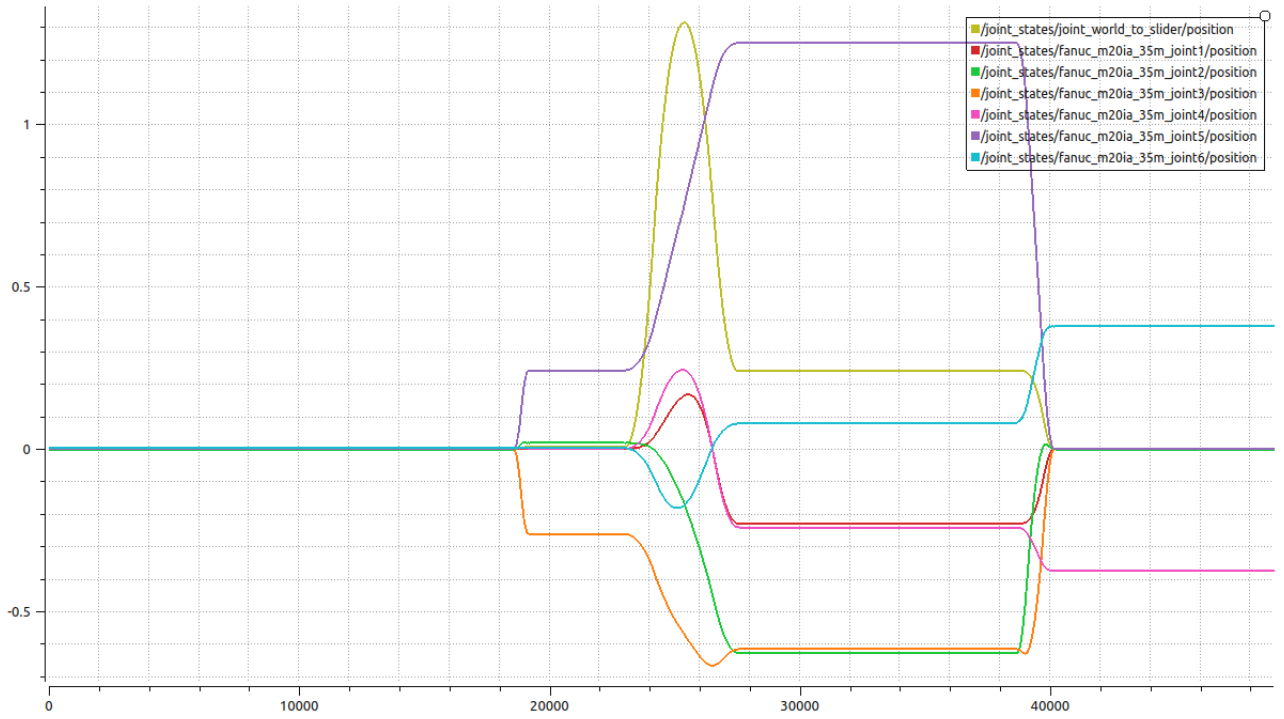
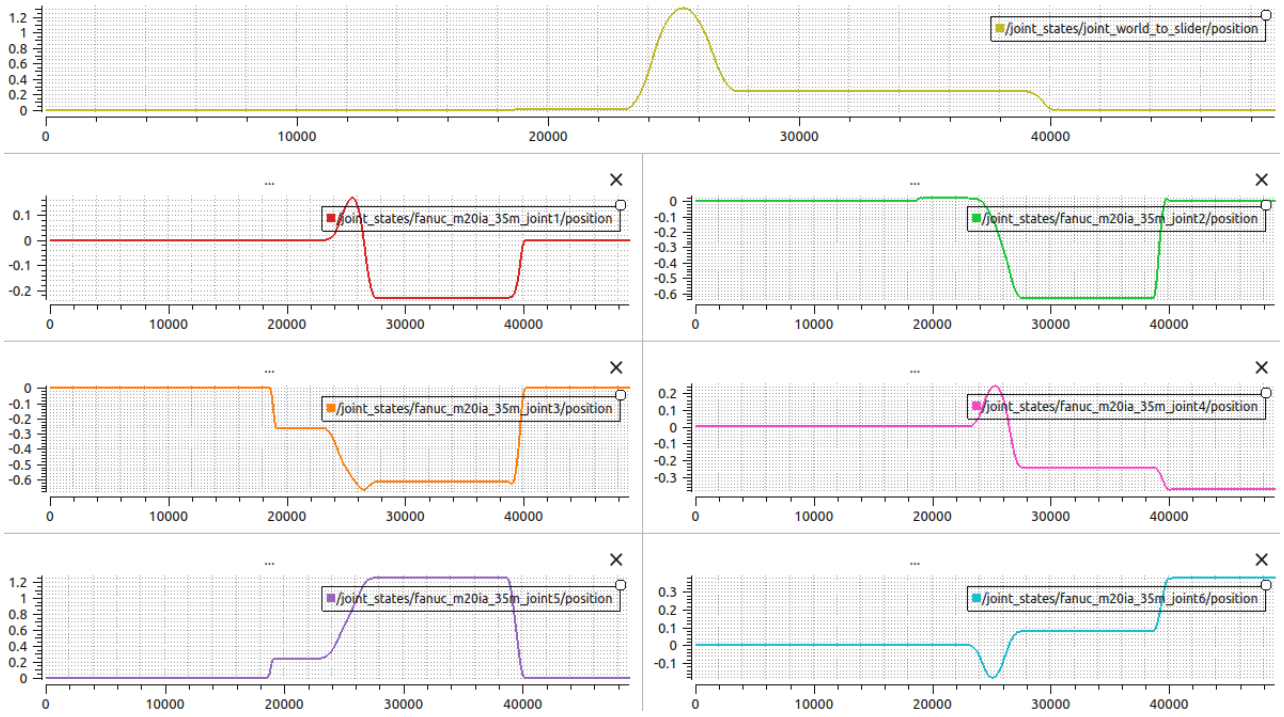


Figure 4.57: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler for `fanuc_m20ia_35m_joint6`



(a) Overlapped view



(b) Separeted view

Figure 4.58: *zrotatedplane.db3*: Gazebo simulation results in PlotJuggler (summary)

4.2 Data acquisition: design of MATLAB interface

Another developed feature was the possibility to analyze the performance of the robot in terms of trajectory execution. To do so, a MATLAB interface was developed to be able to load the bag files and to extract the joint positions and the joint velocities. This analysis tools are fundamental to understand the performance of the robot in terms of trajectory execution. The file is called `MSEgiunti.m`, the script

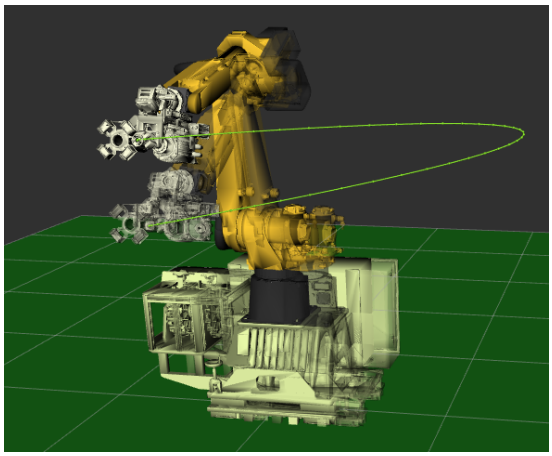
opens the two bagfiles, one for the real trajectory computed and one for the desired one, planned by the `fanuc_m20ia_35m_planning_demo` package and interpolated by the controller. After retrieve the values for each point, the module computes the Mean Square Error (MSE) for each joint and for each point of the trajectory. The MSE is a measure of the quality of the trajectory execution, and it is used to understand if the robot is able to execute the trajectory in a realistic way. After that, for each joint, two plots are made: the first one is the plot of the actual joint position and the desired one, and the second one is the plot of the MSE for each point of the trajectory.
The MSE of j-th joint is calculated as follows:

$$MSE_j = \frac{\sum_{i \in N} [p_{a_i} - p_{d_i}]^2}{N} \tag{4.4}$$

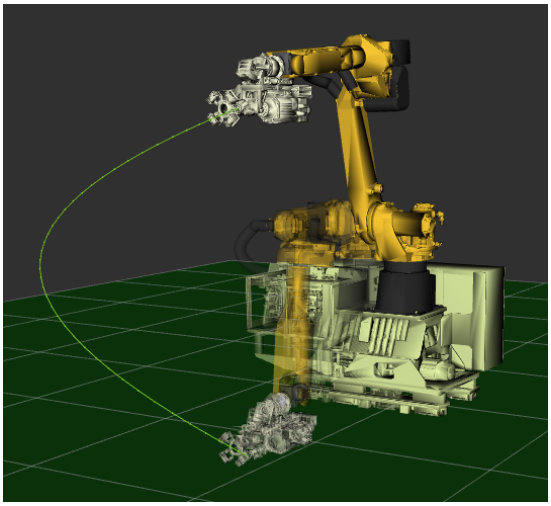
with N being the number of points in the trajectory, p_a is the vector of real values of joint j throughout the trajectory, and p_d is the vector of desired values of joint j for the trajectory.

4.2.1 Our case study

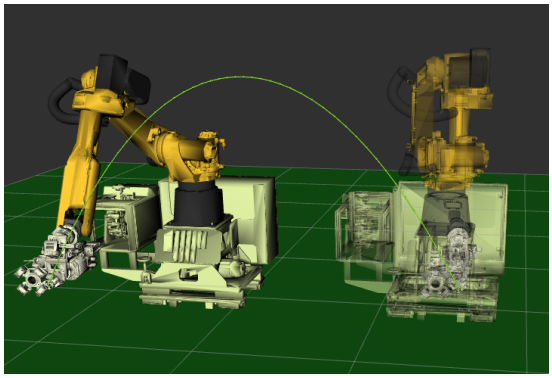
After tuning the parameters for the PID controller, the obtained controller was validated on trajectories different from the ones used for tuning, in order to test whether the obtained gains are good enough to allow the Fanuc robot to follow the given references with a contained error. Regarding the error, other than checking whether the trajectories are followed, a mathematical parameter to evaluate the goodness of the controller is found in the following chapter 4.2.2 but for now the focus is on the selected trajectories.



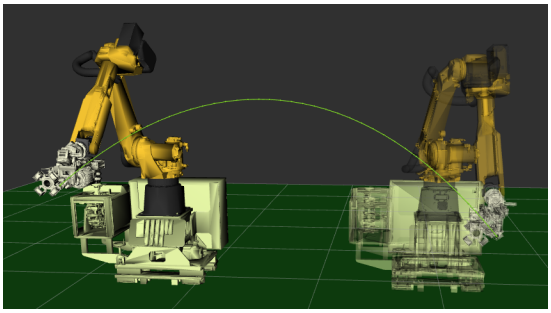
(a) *tight_left.db3*



(b) *low_ground.db3*



(c) *low2high.db3*



(d) *basicplane_lowoblique.db3*

Figure 4.59: Four parabolas considered after PID tuning phase in RViz

The trajectories shown in 4.2.1 have been designed to test whether the robot manages to execute trajectories in conditions that are much different than the ones used for tuning (without going outside the operating space of the robot), also the specified trajectories are such that the planner manages to easily find a path for them and the execution is possible. Many more trajectories were designed through the GUI generated by the MATLAB module, but in some cases the planner failed, as explained in 3.2.1.3 or the waypoints were such that the robot would get out of its operating space to go through them. To be more specific, they were chosen to:

- test the ability of the robot to execute low trajectories with the starting and ending point on a different height (trajectory *basicplane_lowoblique*);
- test the ability of the robot to execute low trajectories with high peaks (trajectory *low2high*);
- test the ability of the robot to execute trajectories that end on very low points with minimal error (trajectory *low_ground*);
- test the ability of the robot to execute tight trajectories with starting and ending points very close to eachother in terms of height (trajectory *tight_left*).

A small thing to note regarding the "tight" trajectories is that no setting in which the starting and ending point of the parabola are close in terms of width as the computed parabola would exceed the maximum height that the Fanuc robot can handle.

4.2.2 Metrics used: Mean Square Error (MSE)

In this paragraph the focus moves on the performances of the joint space controller expressed as MSE between the desired trajectory (the one produced by the planning module) and the real trajectory (the one given by the execution of the trajectory in the dynamic simulation through Ignition Gazebo).
The output of the analysis module in MATLAB consists of plots for each of the joints displaying the differences between the desired trajectory and the real one, while also displaying the evolution of the error throughout the execution of the trajectories. Although the plots are labeled as "Joint 1" to "Joint 7", they respectively refer to **joint_world_to_slider** all the way to **fanuc_m20ia_35m_joint6**, so:

- "Joint 1" refers to **joint_world_to_slider**;
- "Joint 2" refers to **fanuc_m20ia_35m_joint1**;
- "Joint 3" refers to **fanuc_m20ia_35m_joint2**;
- "Joint 4" refers to **fanuc_m20ia_35m_joint3**;
- "Joint 5" refers to **fanuc_m20ia_35m_joint4**;
- "Joint 6" refers to **fanuc_m20ia_35m_joint5**;
- "Joint 7" refers to **fanuc_m20ia_35m_joint6**.

The plots displaying the differences between the desired and real trajectories are all grouped in the following images.

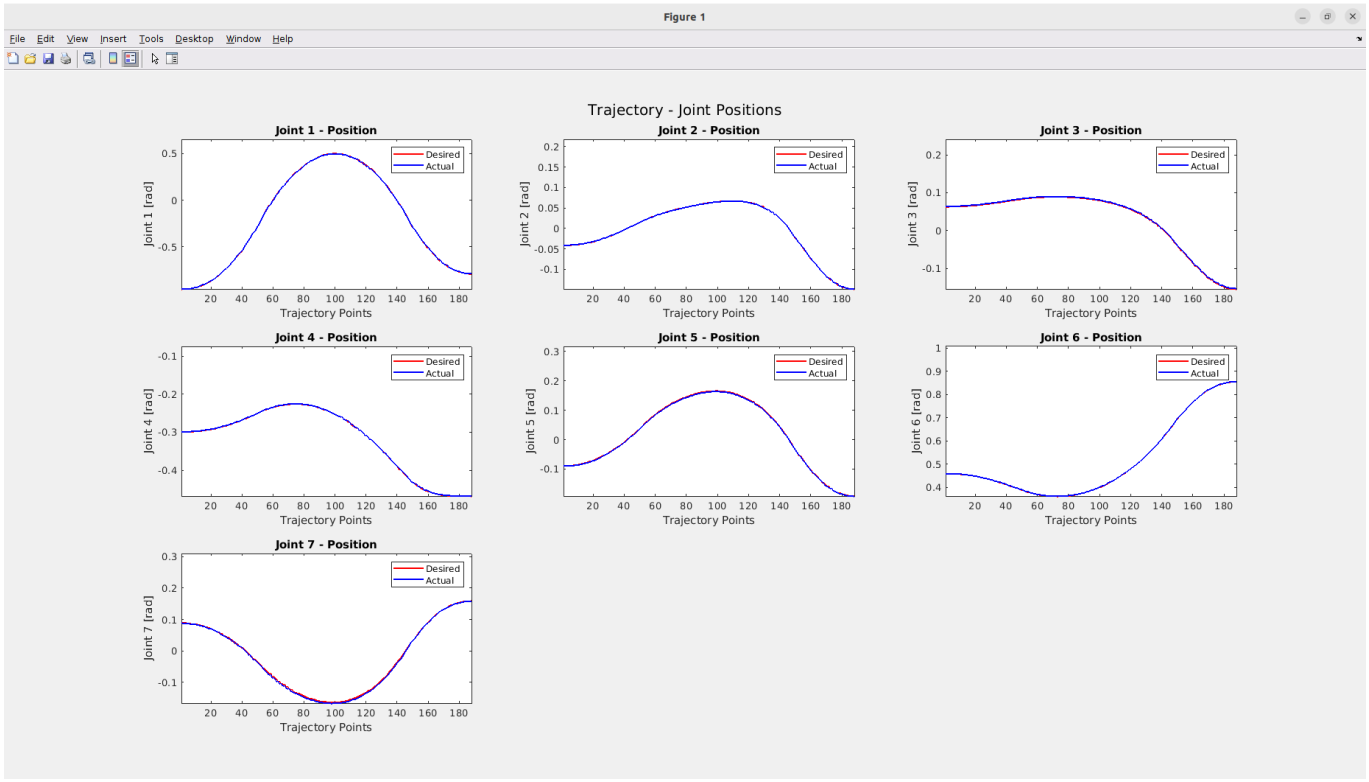


Figure 4.60: *tight_left.db3*

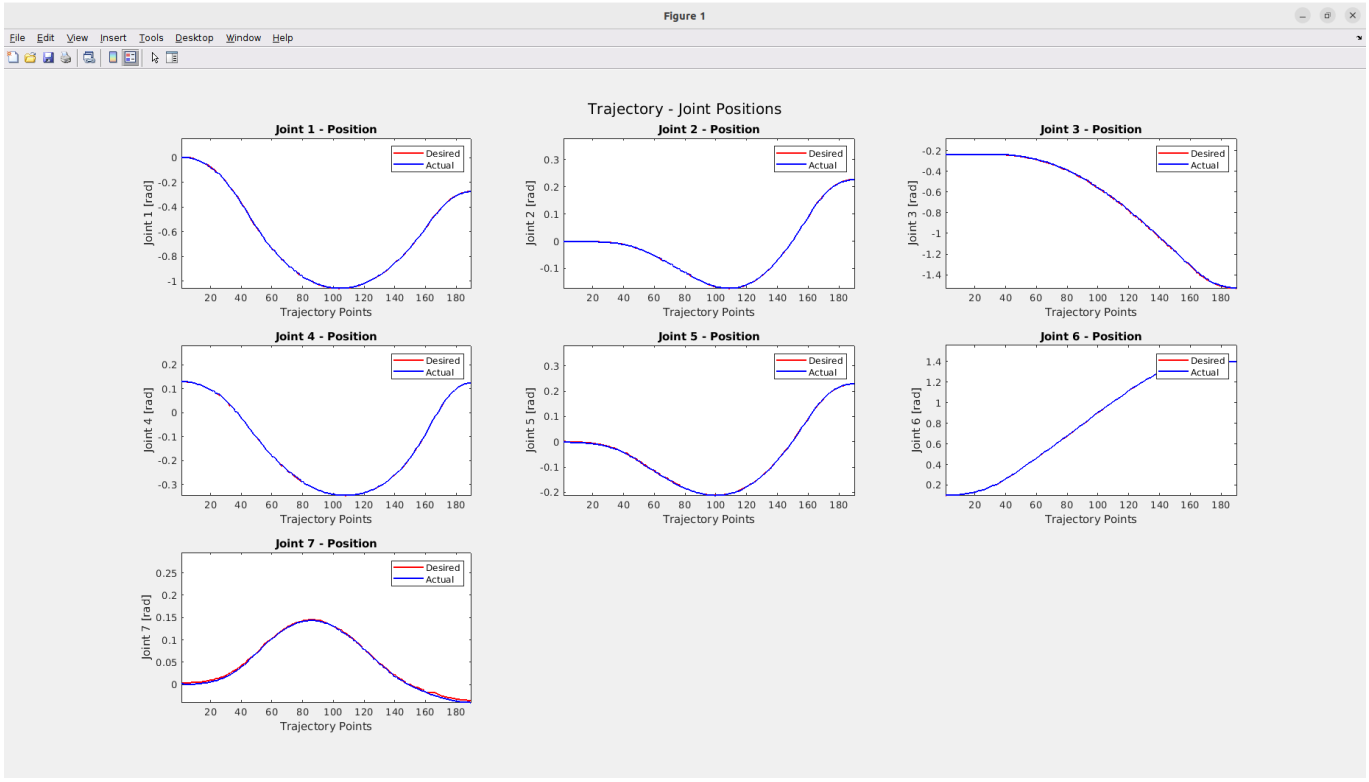


Figure 4.61: *low_ground.db3*

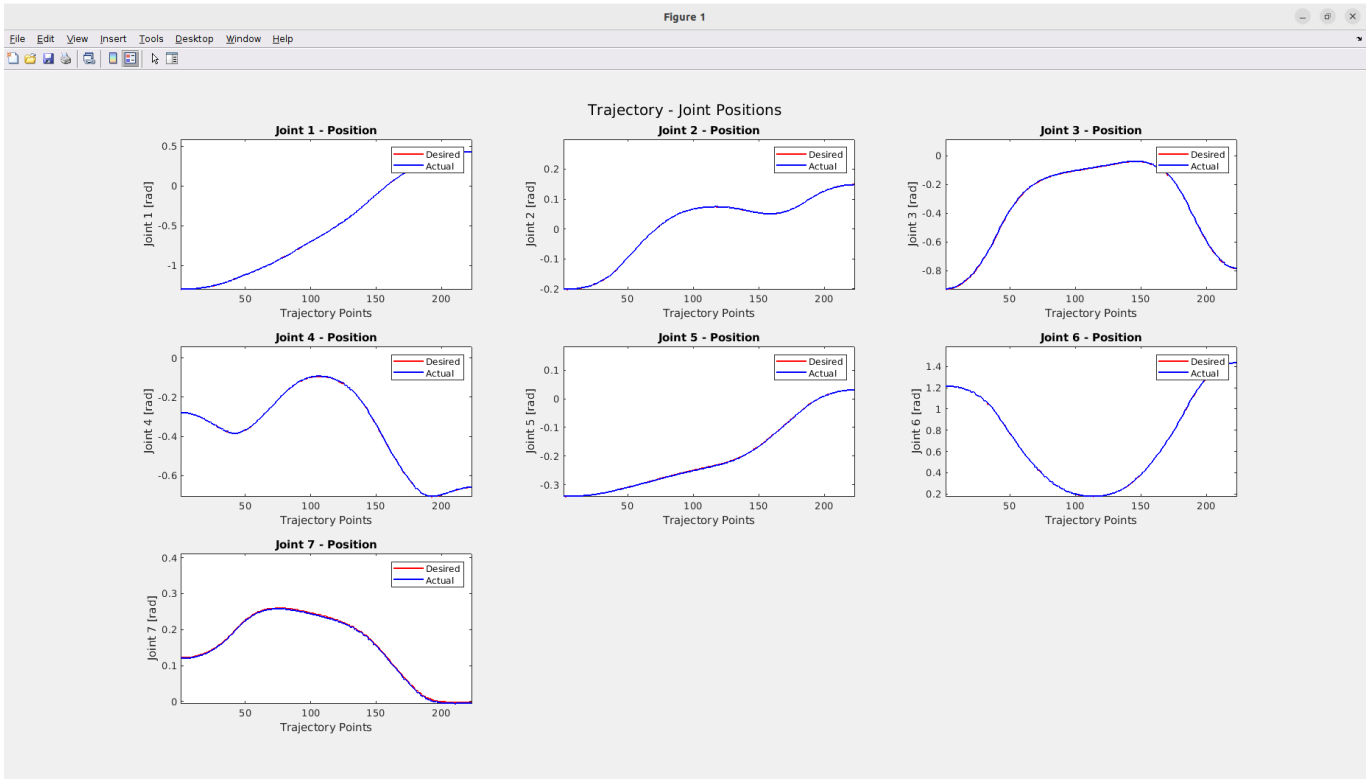


Figure 4.62: *low2high.db3*

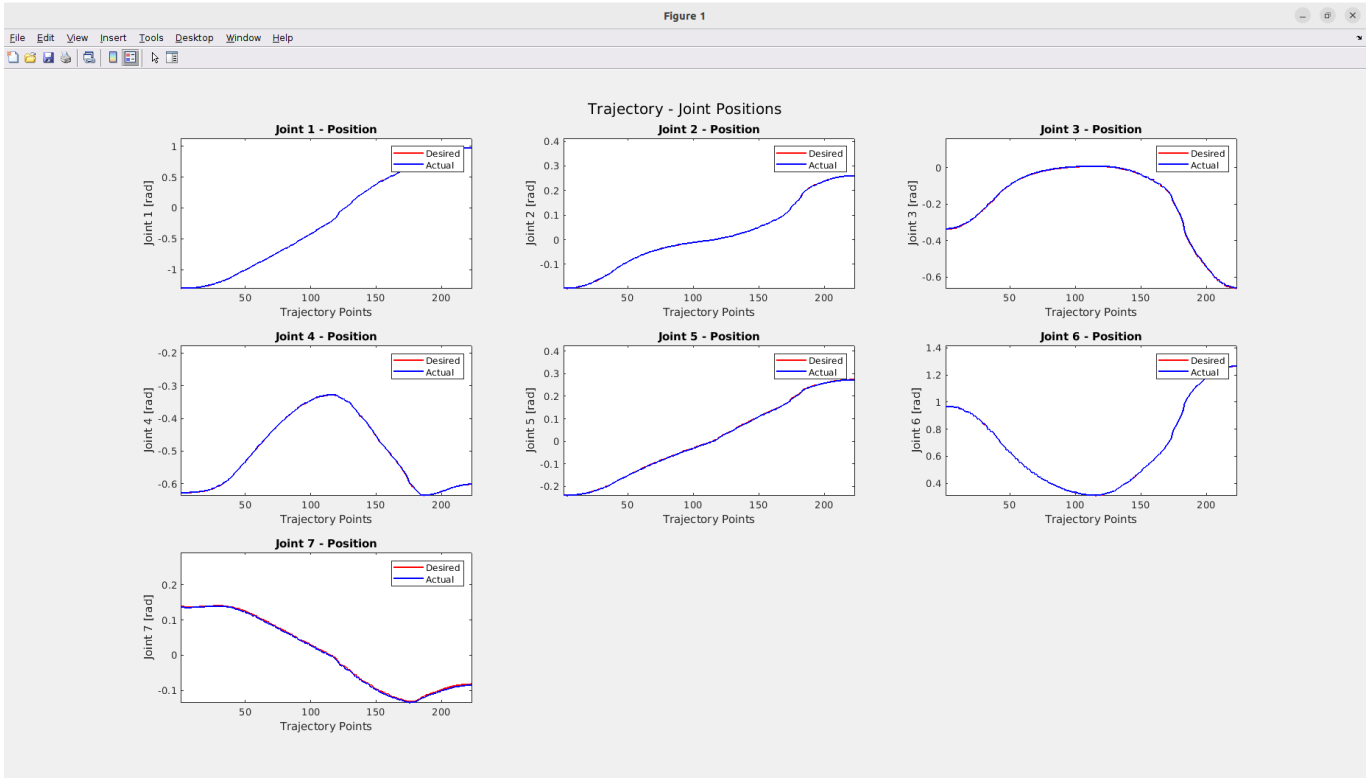


Figure 4.63: *basicplane_lowoblique.db3*

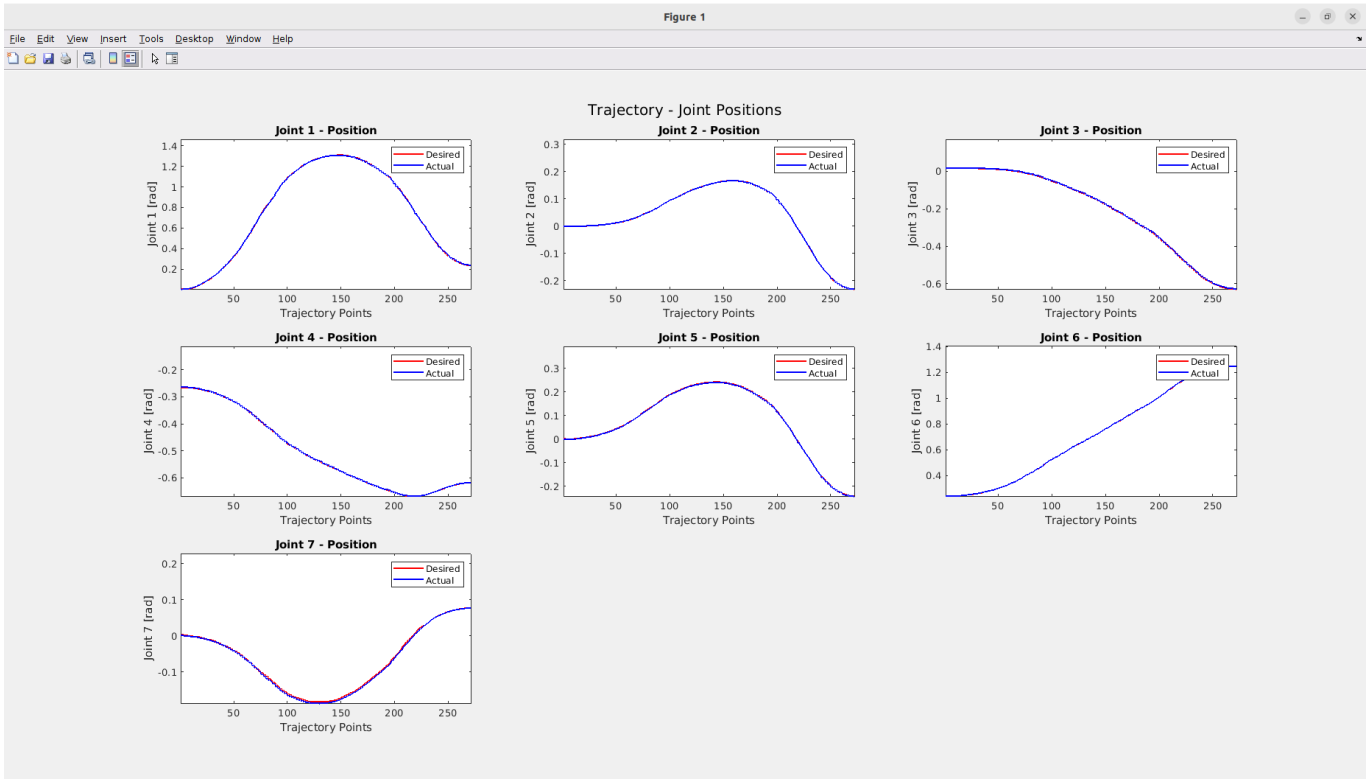


Figure 4.64: *basic.db3*

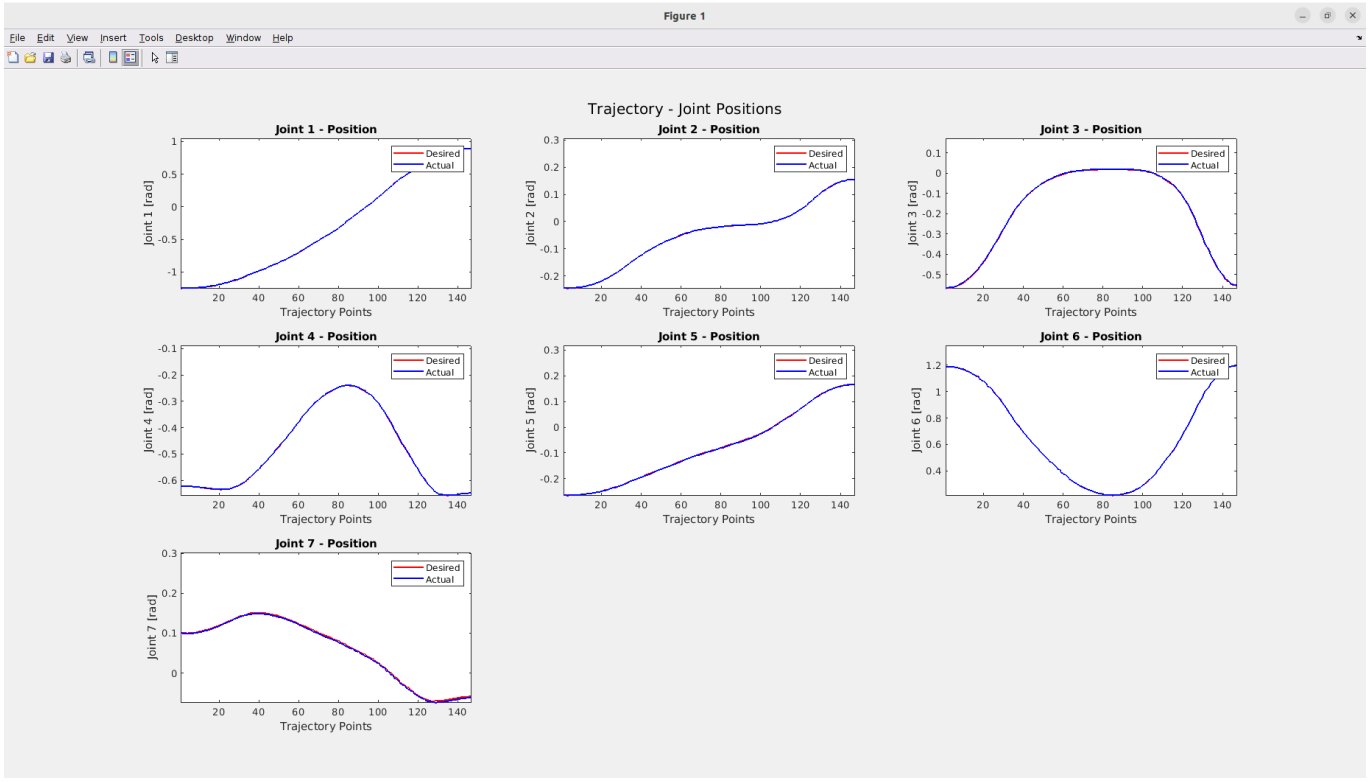


Figure 4.65: *horizontal_low.db3*

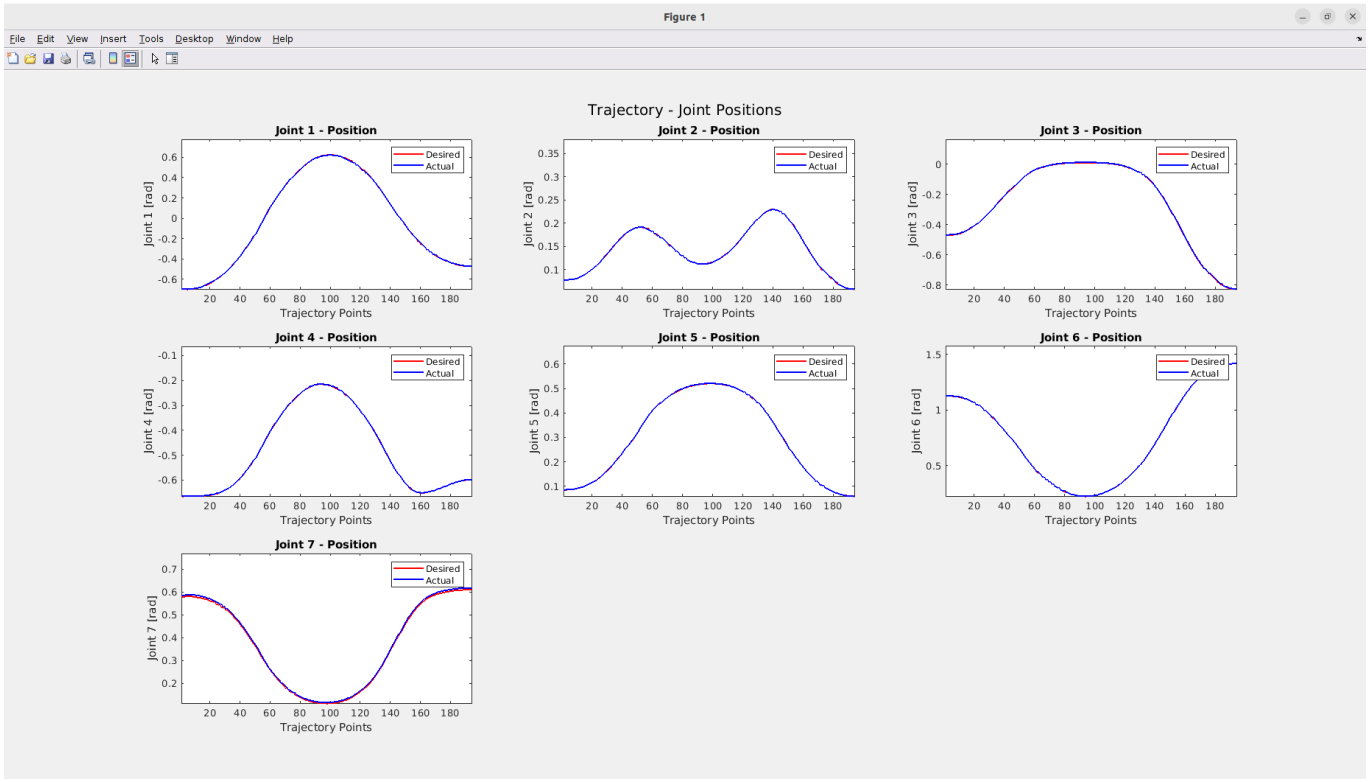


Figure 4.66: *yrotatedplane.db3*

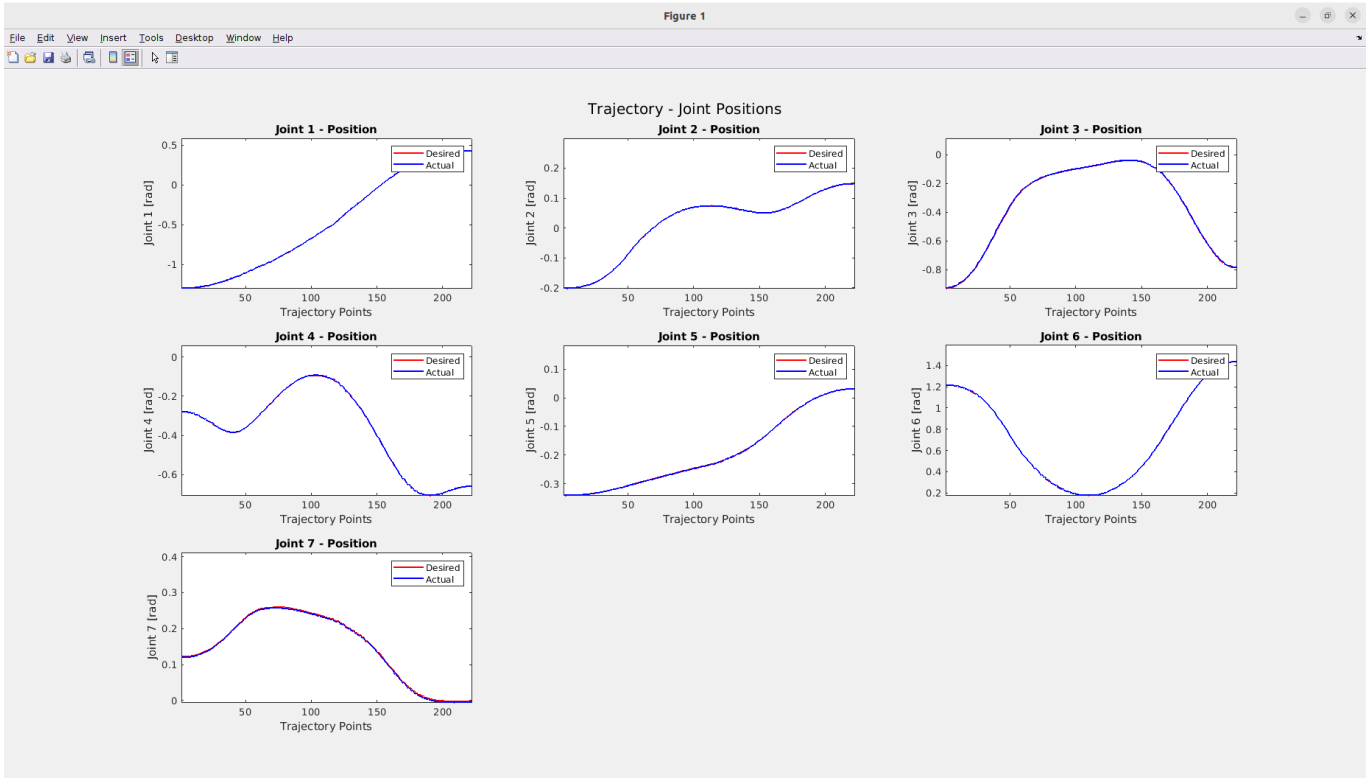


Figure 4.67: *zrotatedplane.db3*

For all the trajectories it is possible to appreciate how the desired trajectories are precisely followed in most cases by most of the joints, with the exception of ”Joint 7”, which is the joint directly linked to the

end effector. The joint ends up differing the most from the desired trajectories because it is the one whose behaviour is the most influenced by the elements that precede it in the kinematic chain. This behaviour cannot be avoided by changing the gains on the joint, as:

- regarding the derivative gain
  - by increasing it the oscillations would be too many to allow for the trajectory executions to end, thus slowing down the execution time by a lot
  - by decreasing it the integral gain would obtain dominance and the oscillations introduced by it would increase the error
- regarding the integrative gain
  - by increasing it any action taken on correcting the errors would cause oscillations on the end effector
  - by decreasing it the errors introduced by the kinematics would be much more difficult to handle
- regarding the proportional gain
  - by increasing it any error introduced by the kinematics is inflated, thus increasing the MSE
  - by decreasing it the joint is unable to follow the given trajectory

Luckily, although the behaviour on the last joint is suboptimal, the resulting MSE is still particularly low.

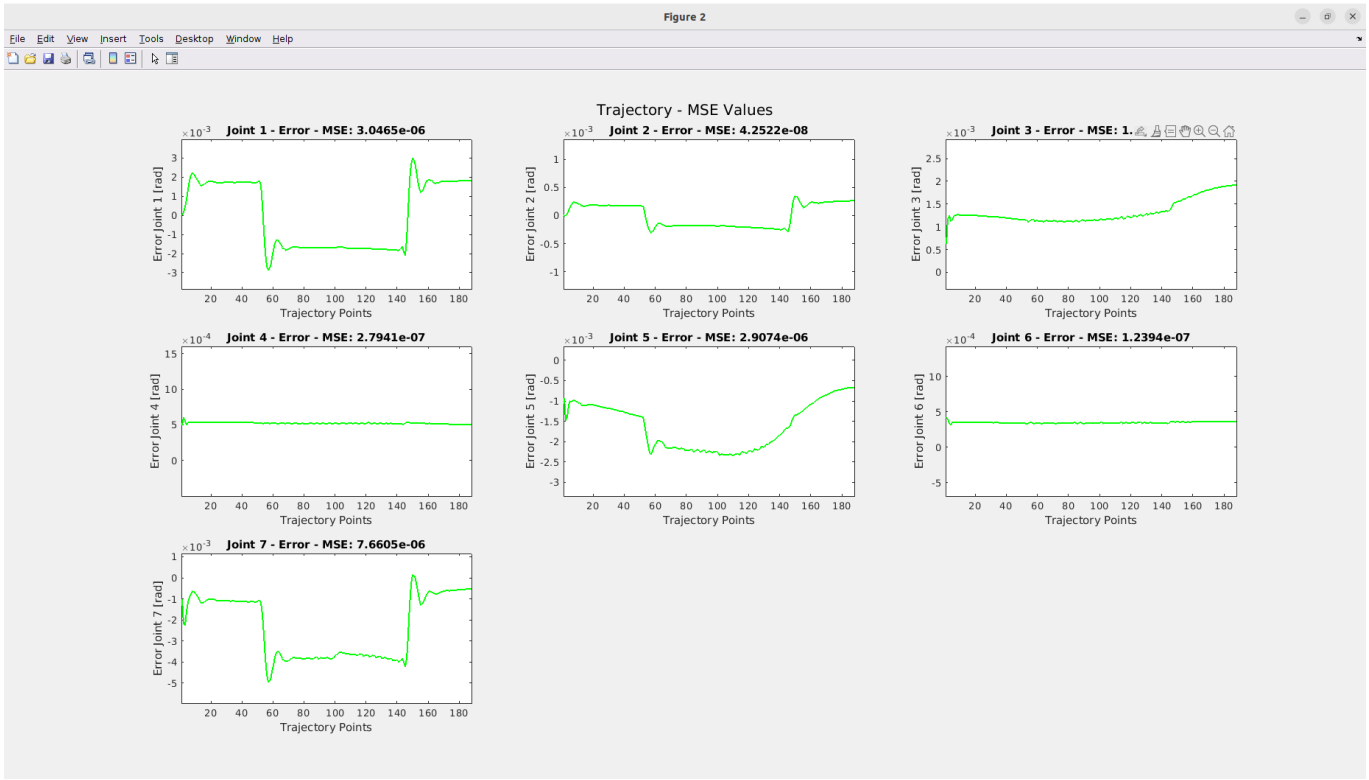


Figure 4.68: *tight_left.db3*

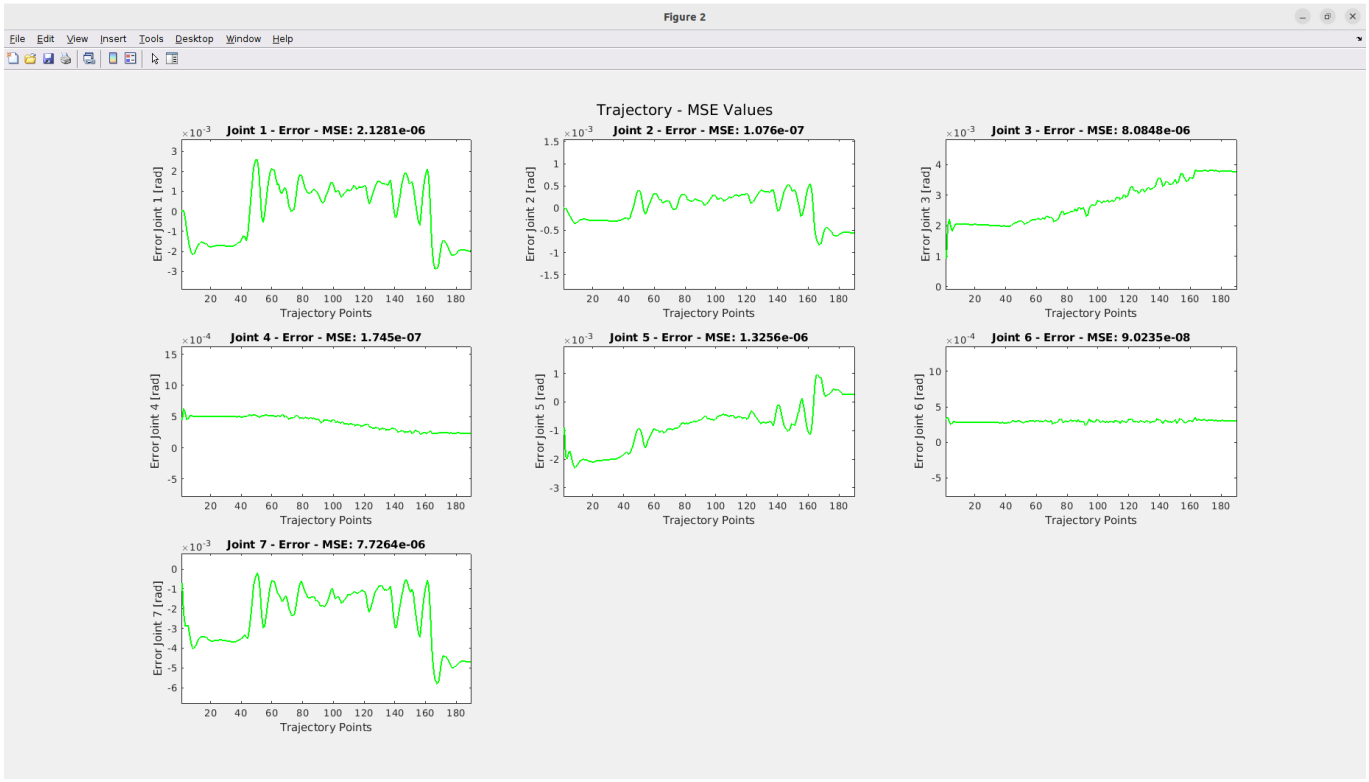


Figure 4.69: *low_ground.db3*

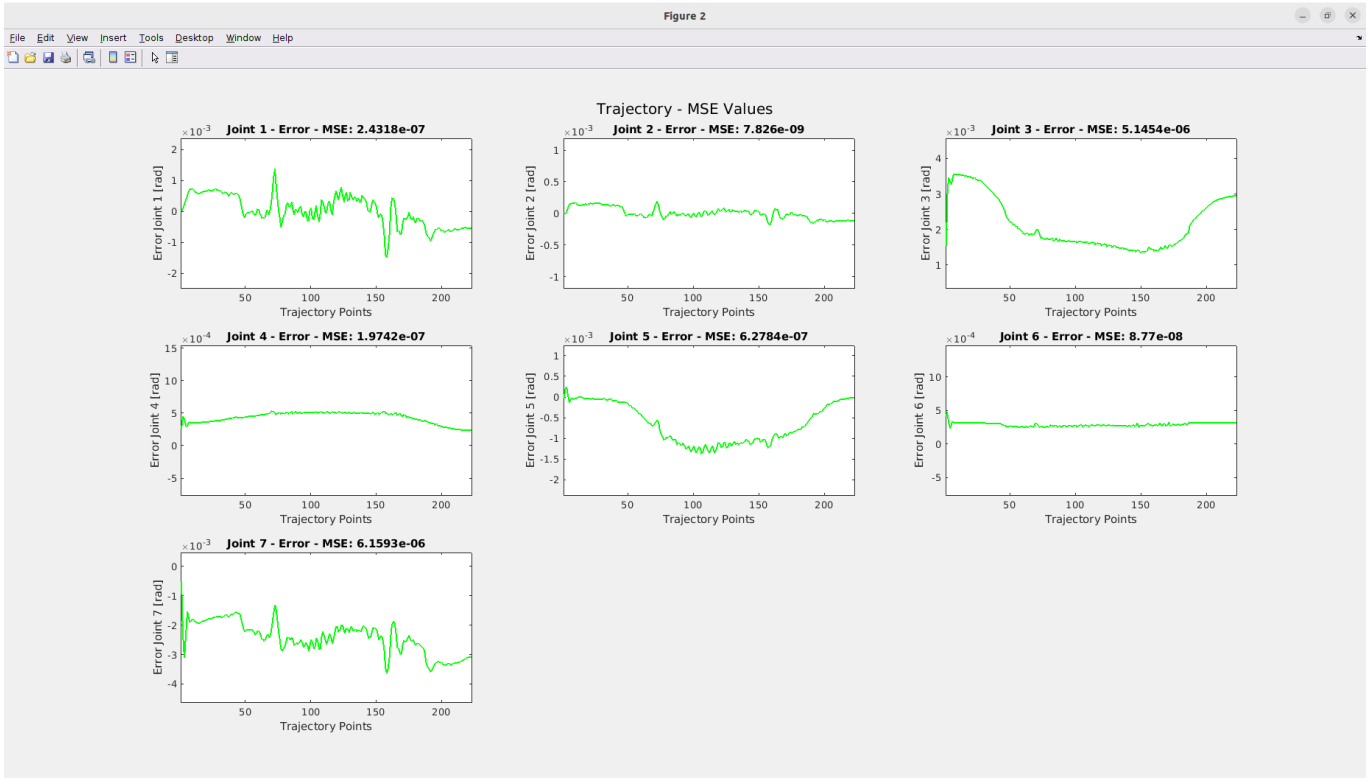


Figure 4.70: *low2high.db3*

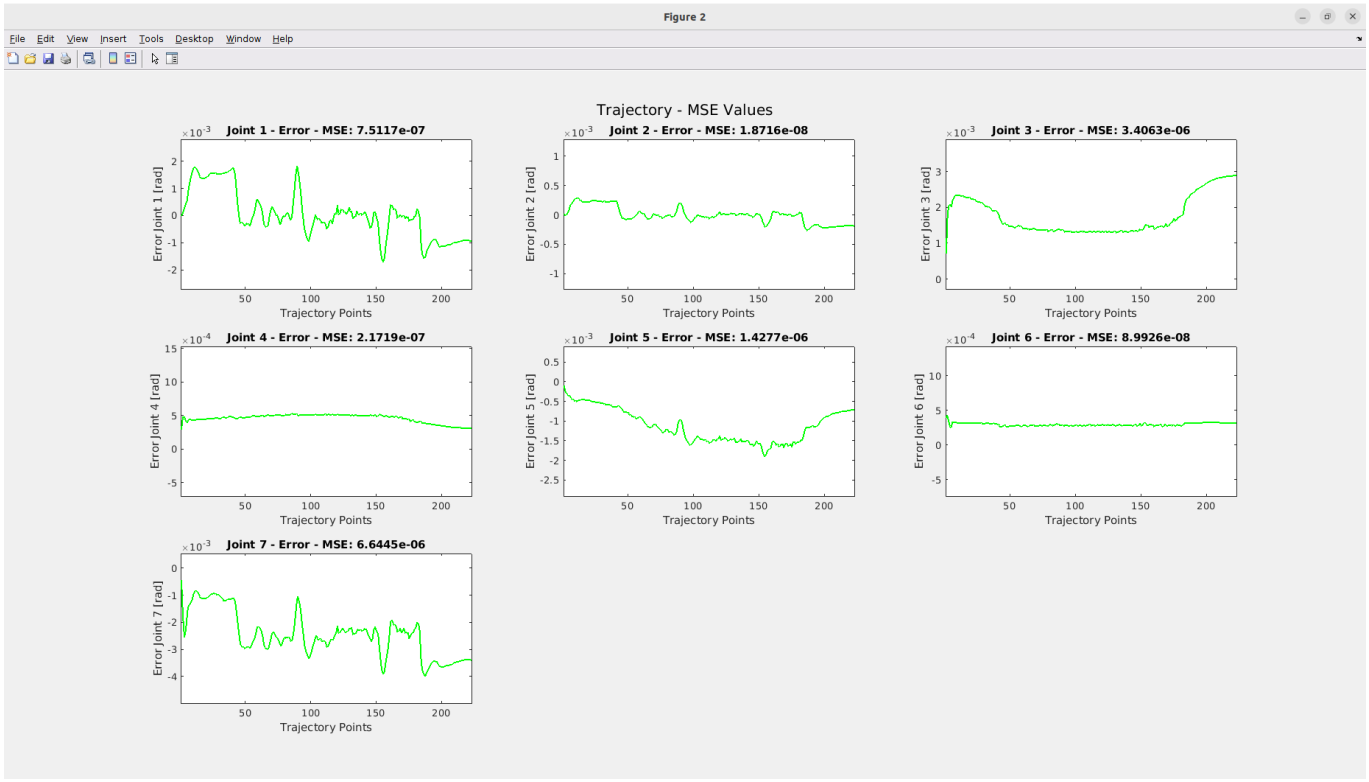


Figure 4.71: *basicplane_lowoblique.db3*

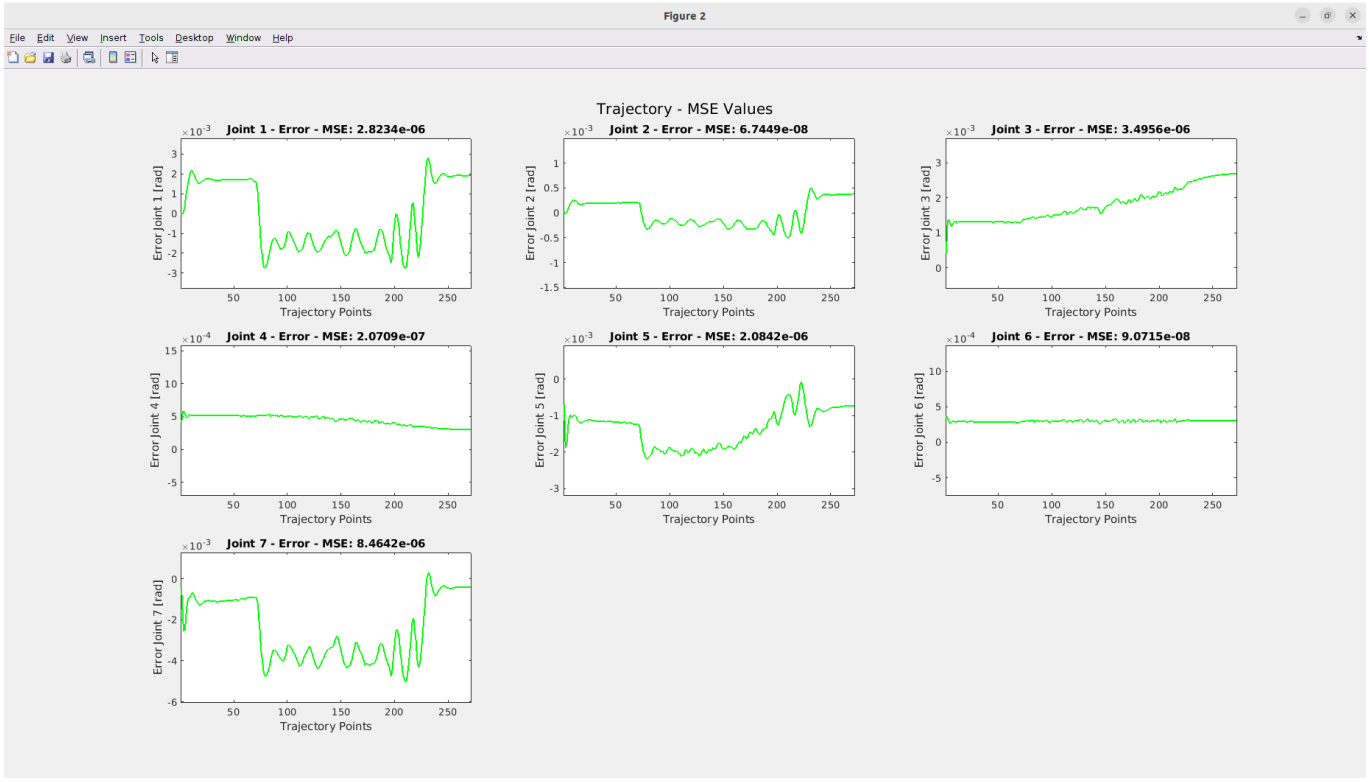


Figure 4.72: *basic.db3*

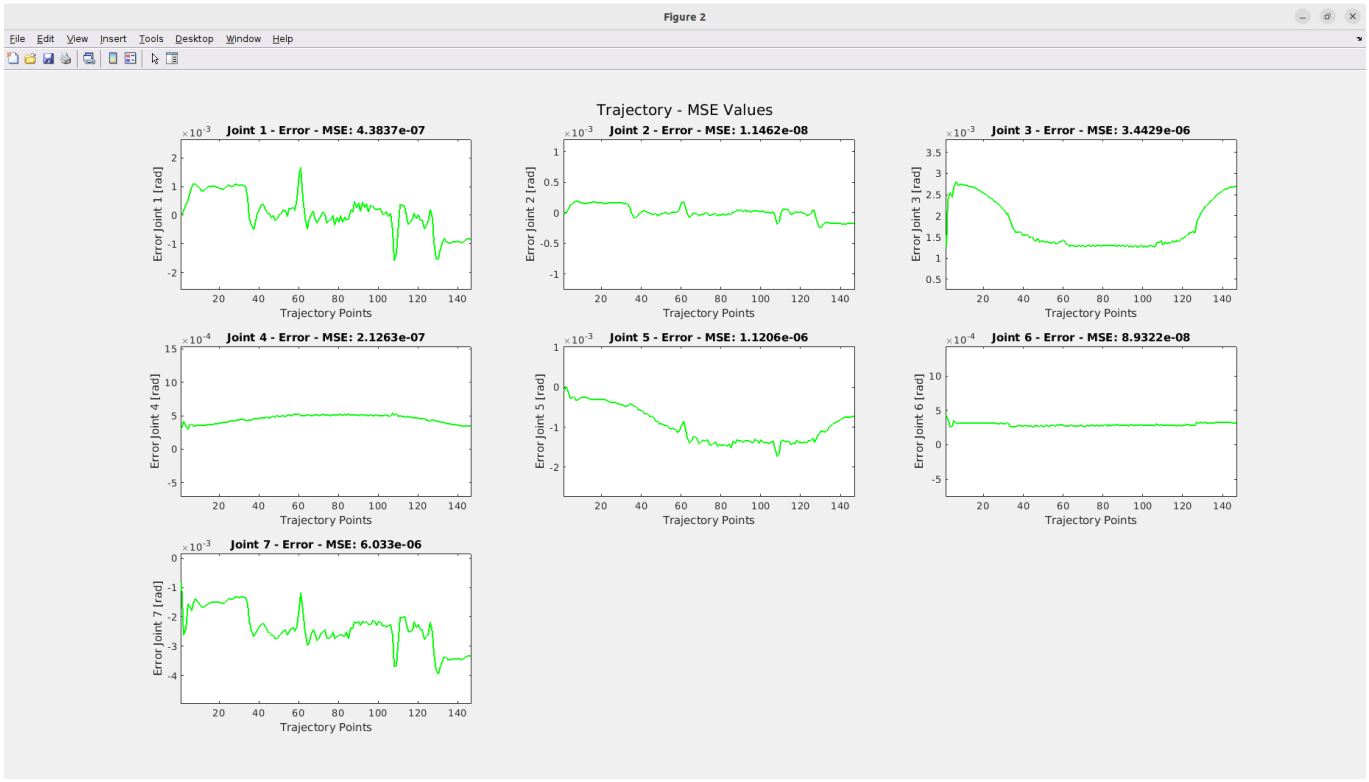


Figure 4.73: *horizontal_low.db3*

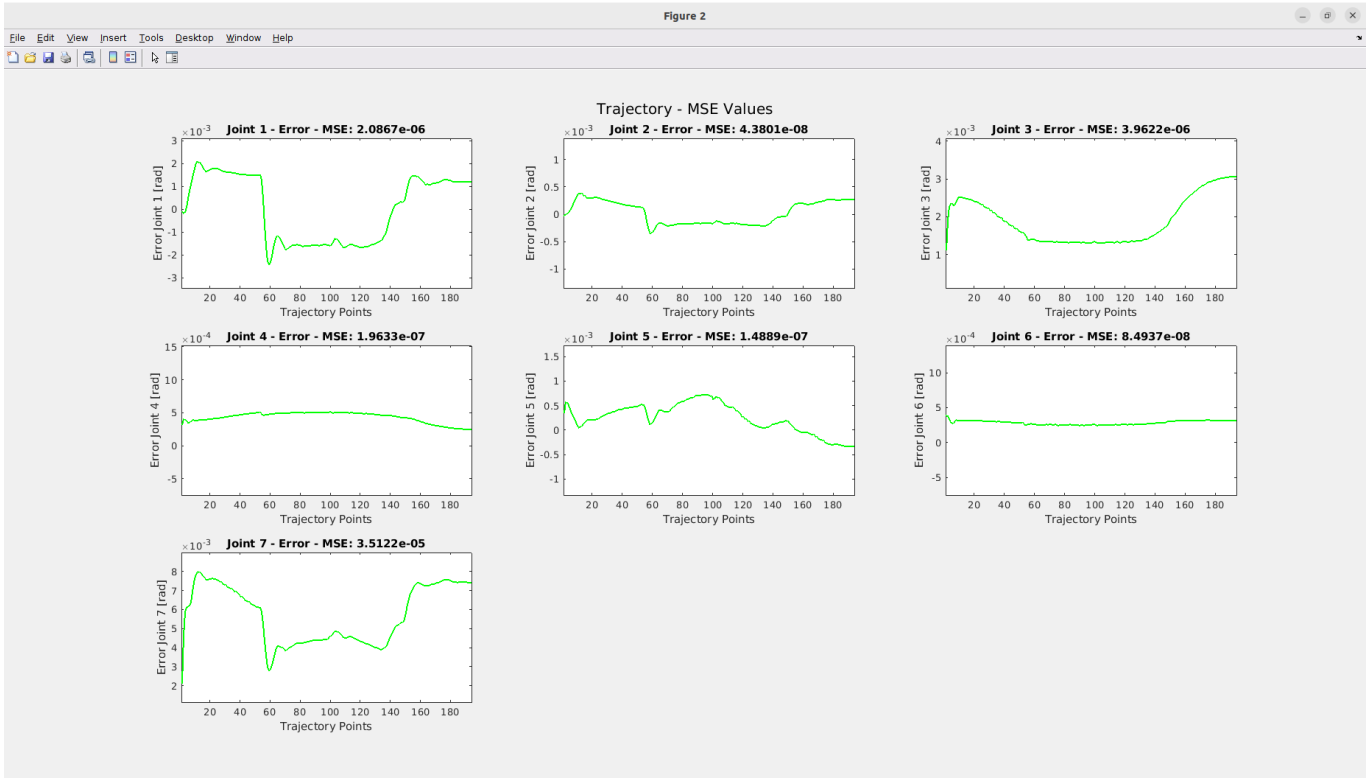


Figure 4.74: *yrotatedplane.db3*

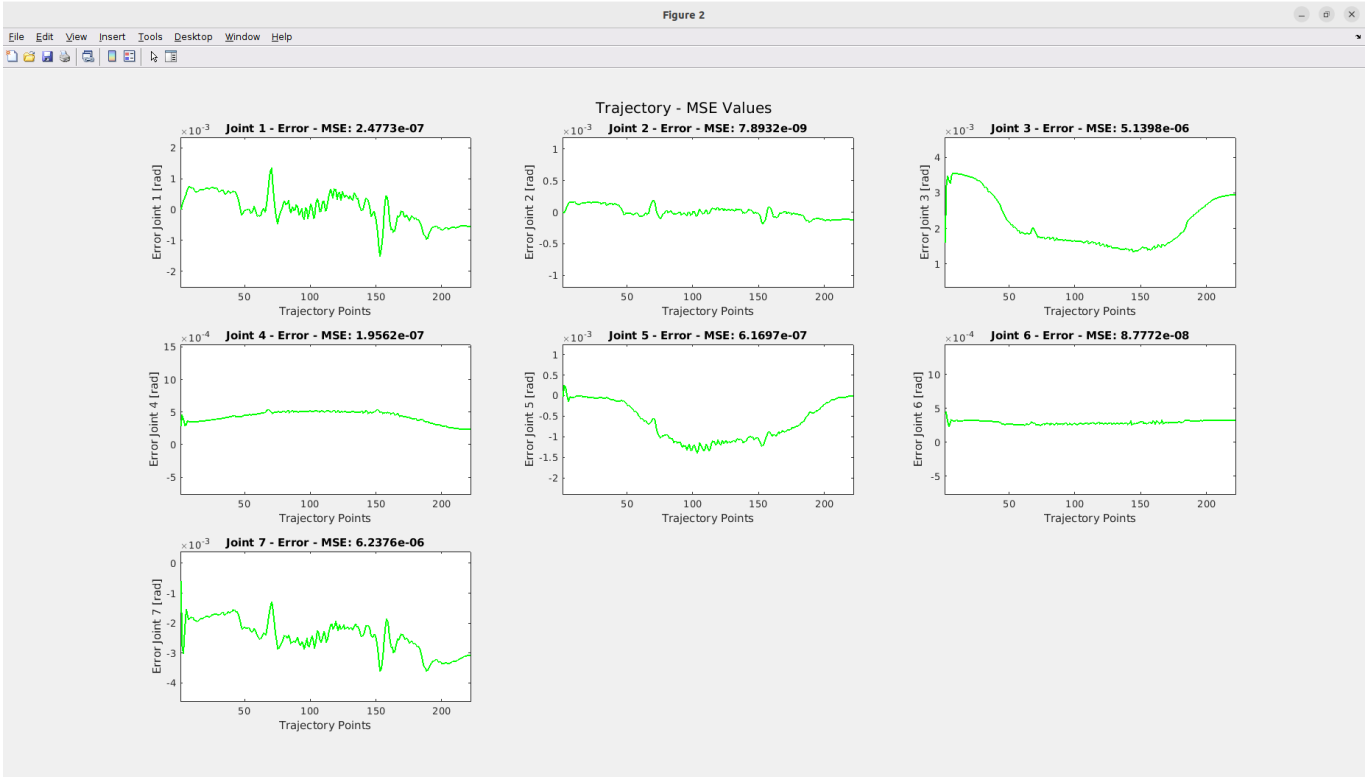


Figure 4.75: *zrotatedplane.db3*

As these plots show, the MSE values involved do not exceed very small ranges of values, as they all assume values between the 10^{-8} order and the 10^{-5} order of values, which is still a very low amount.

4.3 Final considerations

At the end of this chapter, we have developed a controller for the robot, and we have tuned the PID parameters to be able to execute the trajectory in a realistic way. We have also developed a MATLAB interface to be able to analyze the performance of the robot in terms of trajectory execution. This way the user can start controlling the robot and accomplish the task of trajectory execution, while also establish the goodness of said execution. Although control of the real robot is finally achieved, the technique is far from perfect as through system identification a higher precision in tuning the controller could have been reached, as of now the tuning has only been done based on estimates of values and an empirical analysis of the behaviour of the robot.

CHAPTER 5

WP4-CONTROL AND ANALYSIS IN THE TASK SPACE

In this chapter the content of the previous WPs is ultimately extended to allow for planning and control in the task space, while also including an analysis module for the execution of the trajectories through task space position or velocity control.

5.1 Extending the system

5.1.1 Planning module extension

In order to allow for task space control and analysis, first a way to prepare a task space reference set, thus giving the Fanuc robot trajectories defined in the task space, is required. This task can be accomplished by extending the planning module to allow for the creation of such trajectories.
First of all, a few things must be taken into account: no changes are needed for the `server_trajectory` server nor for the MATLAB components seen in 3.1.2, as the communication mechanisms between the planning node and the MATLAB module are the same as before, also the needed calculations for letting the user choose the points of the trajectory are the same. The only changes needed to allow for task space planning lie in the source code of the planning node: the content of the `fanuc_moveit_planning_node.cpp` file.
This time, the node includes a parameter that defines whether the planning should give the robot a task space trajectory or a joint space one: the parameter is called `work_space` and, if set to 0, requires the planning to produce a joint space trajectory, otherwise, if it is 1, it will give a task space trajectory as output. Also, upon executing the trajectory received from the trajectory generation module, if the `work_space` parameter is set to 1, a new function called `execute_cartesian` is called, which takes the same inputs as the `execute` function, but this time the given trajectory is a task space one. It is important to note that, when the group was developing the code, it was decided to add also the possibility to execute a stored trajectory, and to

do so a new parameter was added to the node, called `trajectory_name` that, if passed, will make the node execute the trajectory with the given name and then continue with the normal planning routine based on the `continuos_planning` parameter. This is not included in the previous version of the code, because was a change made after the first version of the code was developed, so it is directly included in the final version of the code. To manage a task space trajectory, the `fanuc_moveit_planning_node.cpp` file must be modified in different ways: the main changes of this node are the extension of the `action_node.cpp` file, which now includes the capability to call a task space controller, in particular, one that has as action a new type of messages that will be described in the next section. This is made by adding new callbacks for the task space trajectories, but also keeping the old ones for the joint space trajectories and the other useful functions mentioned in the previous chapter, to manage with bagfiles and initializes of node that are in common with the previous version of the node, changing only in the type of messages that are stored, in one case the joint space trajectories and in the other the task space ones. Another important change is the introduction of a new function called `fromJointTrajectoryToCartesianTrajectory` that is used to convert a joint space trajectory into a task space one by computing the forward kinematics of the robot and storing the results in a new message of type `CartesianTrajectory`, in this case also the velocities are calculated from the joint space's ones by using the Jacobian matrix of the robot as in:

$$\dot{x} = J(q)\dot{q} \tag{5.1}$$

where $\dot{x}$ is a vector with 3 linear velocities and 3 angular velocities, $J(q)$ is the geometric Jacobian matrix and $\dot{q}$ is the vector of the joint velocities.

5.1.2 Interface change and new launch file

The other changes needed to allow for task space planning lie in other files of related packages. First the *`fanuc_m20ia_35m.ros2_control.xacro`* was modified to include a new parameter that allows to switch from effort command interface to the velocity interface for all the joints, and, at the same time, switch also the configuration files of the controllers, in order to have two different configurations for the two different interfaces. The *`demo.launch.py`* file was also modified to include this new parameter in the starting of the Gazebo simulation, in order to achieve an automatically switch at run time. Another big aspect of the planning routine, is the comunication between the node and the actual controller. In this case, a new type of message was introduced, to allow for the communication between the two nodes, in particular the `action_cartesian_trajectory` package contains an action type of message that allows to specify a `CartesianTrajectory` message and to receive a feedback message that includes the desired and the actual poses of the end effector in the trajectory, as well as the velocities and the accelerations. Furthemore, an error feedback is also included. As a result there are some error codes that are defined. In the action client node, in reality, these codes are not checked, because the `rclcpp_action::ResultCode` default ones are preferred. Note that in this version of the code, no position command interfaces are provided, as they were used in the previous version of the code, when no type of controller was available.

5.2 Task space controller

After extending the planning module to adapt to the task space control, the actual controller is configured and integrated in the environment. As written in the requirements, the dynamic can be astracted, so in this case, as explained in the previous section, the controller is built to manage the velocity of the robot, so that it moves at the desired pose, so the joints will be commanded with the velocities that are needed to reach the desired pose. The other requirement for this controller is that it should be able to receive an entire trajectory offline and then execute it. This is possible assuming that the robot have built-in controllers on each joint that guarantees whatever velocity commanded in a faithful way and work with the robot as it is a single integrator system. Note also that the solution in the operative space does not need inverse kinematics, but to be able to measure where the robot is the forward kinematics is needed. This is valid in this case, because no laser or other sensors are used to measure the position of the robot.

5.2.1 Parameters

In order to achieve the task space control, a new controller must be built, which is capable of receiving a task space trajectory and executing it. To do so, a new ROS2 package must be created, because it is impossible to use a ros2_control default controller for this purpose. To achieve the creation of a new controller, the ROS2 documentation was followed, which explains how to create a new controller from scratch [11]. So a LyfeCycleNode was created, and then the methods of the controller were overridden to allow for the control of the robot, which will be explained in the next section. The parameters that are needed to configure the controller are the following:

- `controller_gain`, which defines the matrix of gains of the controller;
- `joint_model_group_name`, which specifies the planning group with which the controller works;
- `robot_description`, which links the controller to the controlled robot's URDF file;
- `frequency`, which is the frequency at which the controller works;
- `position_tolerance`, which specifies the maximum allowed error on the first point of the trajectory, referring to the position of the end effector of the robot;
- `position_tolerance`, which specifies the maximum allowed error on the first point of the trajectory, referring to the orientation of the end effector of the robot.

5.2.2 Methods

Following the ROS2 documentation, some methods must be overridden in order to allow the controller to work properly. The methods that are overridden are introduced and explained in the following, but for the must important one, which is the `update` one, the workflow will be explained in a separate section. The first one is the `on_init` method in which the variables are initialized. Two important methods are

the `command_interface_configuration` and the `state_interface_configuration` in which the command and state interfaces are configured. When the controller is configured, the `on_configure` method is called, in which the variables initialized in the `on_init` method are set to the values that are specified in the configuration file. In this method the robot model is loaded and an action server is initialized, the one that is used to receive the task space trajectories and to send feedback to the planning module. This action server will be described along with the update method, because they are strictly related. The `on_activate` and `on_deactivate` methods are used to handle the activation and deactivation of the controller along with the initialization of the command and state interfaces. In all of these methods the errors are handled and the controller is stopped if an error occurs.

5.2.3 `update` method and control law

In this section the `update` method of the controller is explained, along with the control law that is used to command the robot. The `update` method is the most important one, because it is the one that is called at the frequency specified in the configuration file for the controller manager, and it is the one that is used to command the robot. The control law that is used to command the robot is the following:

$$\dot{q} = J_A^+(q)(\dot{x}_d + Ke) + (I_n - J_A^+ J_A)\dot{q}_0 \tag{5.2}$$

This is the Closed Loop Inverse Kinematics (CLIK) in which e is defined as the error between the desired pose and the actual pose of the robot, J_A is the analytical Jacobian matrix of the robot, K is the gain matrix of the controller, specified in the parameters, $\dot{x}_d$ is the desired velocity of the end effector, $\dot{q}$ is the velocity of the joints, $\dot{q}_0$ is the velocity of the joints in the null space, I_n is the identity matrix of the dimension of the joints, and J_A^+ is the pseudo-inverse of the Jacobian matrix. In this project the main task was to execute a trajectory, so the null space was not used, because no secondary tasks were specified, furthemore the pseudoinverse is used instead of the inverse of the Jacobian matrix, because the robot has more than 6 joints, so the Jacobian matrix is not square. Given this, the control law can be simplified as follows:

$$\dot{q} = J_A^+(q)(\dot{x}_d + Ke) \tag{5.3}$$

Using this control law, the dynamic of the error is the following:

$$\dot{e} + Ke \tag{5.4}$$

so by choosing the gain matrix K it is possible to make the error converge to zero. In reality, according to [1] there is another way to compute the joint velocities, having the possibility to use unit quaternion to represent the orientation, and the control law can be written as follows and is computionally more efficient:

$$\dot{q} = J^{-1}(q) \begin{bmatrix} \dot{p}_d + K_p e_P \\ w_d + K_O e_O \end{bmatrix} \tag{5.5}$$

Where J^{-1} is the inverse of the geometric Jacobian matrix, $\dot{p}_d$ is the desired linear velocity, w_d is the desired angular velocity, K_p and K_O are the proportional gains for the position and orientation, and e_P and e_O are

the errors between the desired and actual position and orientation. Given that the Jacobian is not square, the pseudo-inverse is used instead of the inverse, so the control law can be written as follows:

$$\dot{q} = J^{+}(q) \begin{bmatrix} \dot{p}_d + K_p e_P \\ w_d + K_O e_O \end{bmatrix} \tag{5.6}$$

Note that the pseudoinverse of Jacobian is the Moore-Penrose pseudoinverse, and it is computed as follows:

$$J^{+} = J^T (J J^T)^{-1} \tag{5.7}$$

The e_p is easy to compute, because it is the difference between the desired and actual position, but the e_O is not so easy to compute, because it is the difference between the desired and actual orientation, and the orientation is represented by a quaternion. A generic formula to compute this is:

$$e_O = \Delta\epsilon = \eta_e(q)\epsilon_d - \eta_d\epsilon_e(q) - S(\epsilon_d)\epsilon_e(q) \tag{5.8}$$

Where the $_d$ and $_e$ subscripts refer to the desired and actual orientation, η is the scalar part of the quaternion, ϵ is the vector part of the quaternion, and S is the skew-symmetric matrix of the vector part of the quaternion defined generally as follows, given a vector x and its components x_1, x_2, x_3, and a vector y and its components y_1, y_2, y_3:

$$S(x) = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \tag{5.9}$$

So matching this in our case, the $S(\epsilon_d)$ can be computed as follows:

$$S(\epsilon_d) = \begin{bmatrix} 0 & -\epsilon_{d3} & \epsilon_{d2} \\ \epsilon_{d3} & 0 & -\epsilon_{d1} \\ -\epsilon_{d2} & \epsilon_{d1} & 0 \end{bmatrix} \tag{5.10}$$

where $\epsilon_{d1}, \epsilon_{d2}, \epsilon_{d3}$ are the components of the vector part of the desired quaternion. Finally, if it's considered that sometimes the pseudoinverse of the Jacobian could be singular, a damping factor can be added to the control law, so that the control law becomes:

$$\dot{q} = J_d^{+}(q) \begin{bmatrix} \dot{p}_d + K_p e_P \\ w_d + K_O e_O \end{bmatrix} \tag{5.11}$$

Where J_d is the damped pseudoinverse of the Jacobian computed as:

$$J_d^{+} = J^T (J J^T + \epsilon^2 I)^{-1} \tag{5.12}$$

where ϵ is the damping factor, and I is the identity matrix of the dimension of the joints. So starting from the kinematics model of the robot, having the desired pose and velocity, and having the actual pose and velocity, thanks to the retroaction, it is possible to compute the joint velocities that are needed to reach the desired pose.

In the `update` method, the control law is implemented as follows:

1. first, the current positions and velocities of all the joints are acquired and stored, while the pose of the end effector is evaluated by using the forward kinematics of the robot;

2. position and orientation tolerances are checked, also an additional control to check whether the starting point of the trajectory is too far from the desired starting point establishes whether the procedure should be aborted or not, as if it is too far the entire process is stopped;

3. given the desired quaternion and the actual quaternion, the error is computed as described above;

4. the pseudo-inverse of the jacobian is computed from the robot's kinematic state;

5. the desired linear and angular velocities are taken from the desired state attribute and put into a vector V;

6. the velocities to command the robot joints with are computed through the control law as $\dot{q} = J_d^+(KE + V)$ and the relative command interface are used to publish the velocities to the joints, this is equivalent to the formula written above, but in this case K is a vector including both K_p and K_O and the error E is a vector including both e_P and e_O;

7. the trajectory index is incremented until the trajectory is complete.

In order to allow the controller to interact with the rest of the ROS environment, it must work with the help of an action server, which is the `follow_cartesian_path` action server initialized with the `on_configure` method of the controller. In order for the controller to use this server, 3 methods needed to be overridden:

- `handle_goal`, which deals with handling the trajectories coming from the planning module, as it rejects them if they are not valid or if the controller is not running;

- `handle_cancel`, which makes the controller save the trajectory, if valid, upon accepting the goal coming from the environment, but first the trajectory undergoes an interpolation step, described in the next section;

- `handle_accepted`, which allows the server to manage a request of canceling the trajectory execution.

When the goal is accepted, the `update` method starts its execution, and the trajectory is executed (this is not exactly true, because in reality, the `update` method is called by the controller manager, so in order to discretize if there is a goal to execute, a boolean variable is used). While the main flow was described, the feedback is also managed. In particular at each step of the `update` method, if the trajectory is being executed, the feedback is sent as feedback of the action server, and it includes the actual pose and velocity of the end effector, the desired pose and velocity, and the error. It is important to note that the desired and actual poses and velocities are taken in two different ways, because the desired one is the one that is specified in the trajectory at the previous step, while the actual one is the one that is computed at the current step and the error is the difference between the two. Another important aspect deals with the orientation, as one quaternion is taken from the trajectory, and the other is computed from the actual pose,

so, sometimes, calling q the quaternion from the trajectory and q_c the quaternion from the actual pose, they may differ in sign, for example if $q = [0.5, -0.5, -0.5, -0.5]$ and $q_c = [-0.5, 0.5, 0.5, 0.5]$ they represent the same orientation. To handle this, before publishing the feedback, the two quaternions are compared, and if they differ in sign, the quaternion from the actual pose is multiplied by -1 to guarantee a realistic comparison in the MSE module 5.3.2.

5.2.4 Interpolation

The interpolation is a fundamental step in the execution of the trajectory, because the trajectory is not executed as it is, but it is interpolated in order to make the robot execute it at the desired frequency of the controller. The interpolation is done by the `handle_accept` method of the action server, and it is done in a linear way, so that the robot moves at the desired pose at the desired time. For each couple of points of the trajectory, the interpolation is done as follows:

1. the time distance between the two points is retrieved;
2. the number of samples between the two points is computed as the time distance multiplied by the update rate of the controller;
3. for each new sample, the pose and velocity are linear interpolated starting from the first point, based on the time distance and the number of samples. For the orientation the interpolation is done by using the `Slerp` method of the `Eigen` library, which is a spherical linear interpolation method, and it is used to interpolate the quaternions in a linear way, so that the robot moves at the desired orientation at the desired time;

The choice to use a linear interpolation instead of a more complex one is due to the fact that the linear interpolation is computionally less expensive, and it is enough to make the robot move at the desired pose at the desired time in our cases, where the trajectories are not so complex and the points are already well defined and close to each other. Furthermore it is important that the kind of interpolation that a controlloer should use is a linear-microinterpolation as explained in the Robotics course.

5.2.5 Tune of the controller

Given the structure and the behaviour of the controller, the controller gains must undergo a tuning procedure to make sure the designed controller succeeds in following the given trajectories. To be more specific, the gains deal with 6 parameters:

- the errors on the x, y and z coordinates of the positions of the trajectory to be followed;
- the errors on the roll, pitch and yaw angles of the points of the trajectory to be followed.

Initially the controller gains were all set to 1.0, but this caused the robot to fail in precisely following the given trajectories, as the low gains allowed for the appearance of a constant error in following the positions,

while the low gains on the orientation angles caused the robot a failure in following the orientation of the trajectory. That is why the gains were all increased, and after testing and making use of a MSE-based analysis tool in the task space to evaluate the goodness of the controller gains, the following set of parameters was achieved:

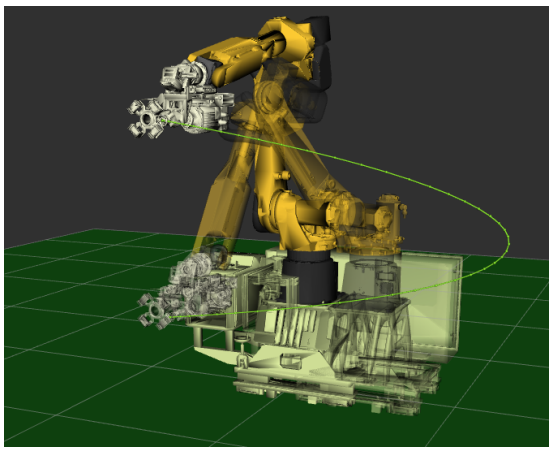
- for the XYZ position coordinates, [200.0 200.0 200.0]
- for the RPY orientation angles, [40.0 40.0 40.0]

This set of gains was obtained by gradually increasing the values on the controller gains (mostly the position ones) until the MSE of the test trajectories was very low. Also, no higher values were considered for the gains to avoid oscillations during the trajectory execution.

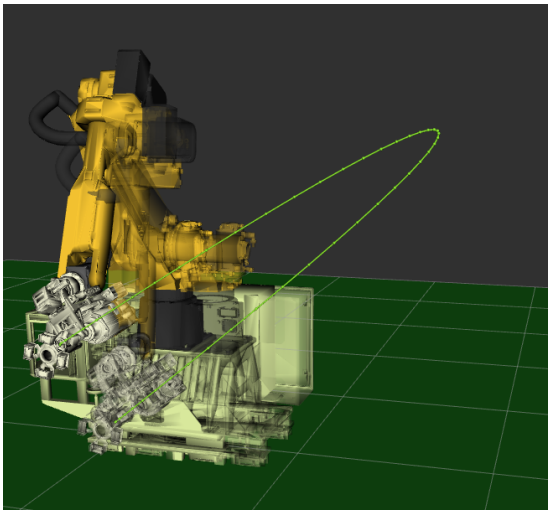
5.3 Data acquisition: design of MATLAB interface

5.3.1 Our case study

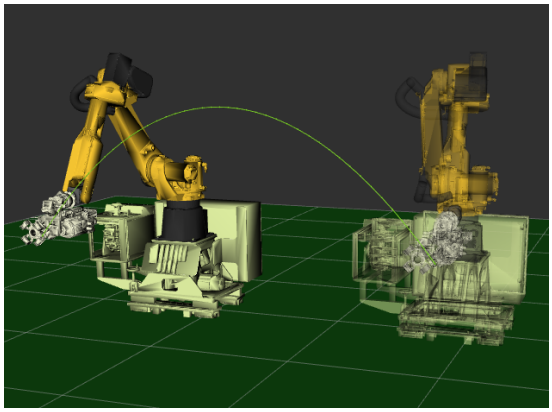
While describing how the gains for the task space velocity controller, ”test trajectories” were cited. To be more specific, these trajectories are the same 4 parabolas that were used during the tuning for the PID gains, as the control task is the same but the robot should be able to accomplish it with any of the 2 available controllers.



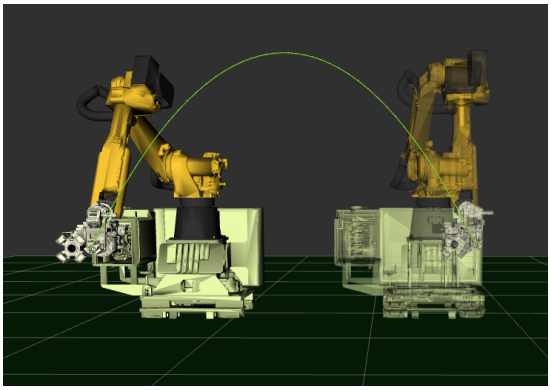
(a) *basic.db3*



(b) *yrotatedplane.db3*



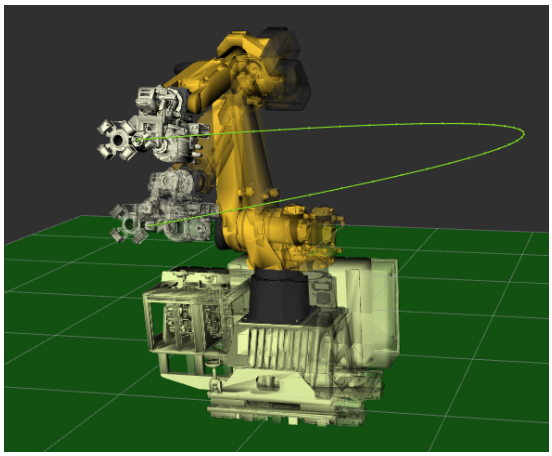
(c) *horizontal_low.db3*



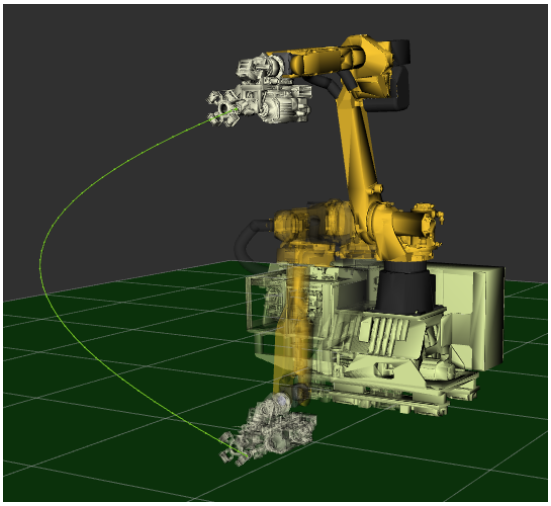
(d) *zrotatedplane.db3*

Figure 5.1: Four parabolas considered during task space tuning phase in RViz

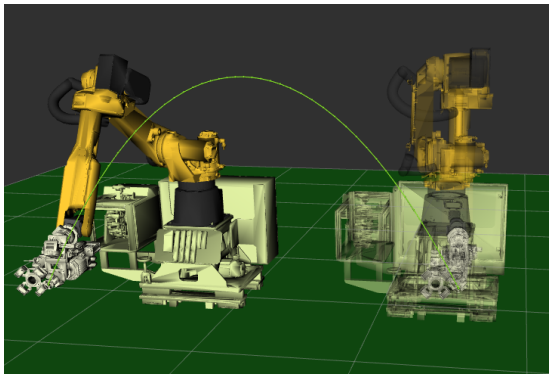
While the 4 trajectories displayed in 5.3.1, the same 4 parabolas used for validation during the PID gain tuning were used to test the capabilities of the robot under the usage of the task space controller.



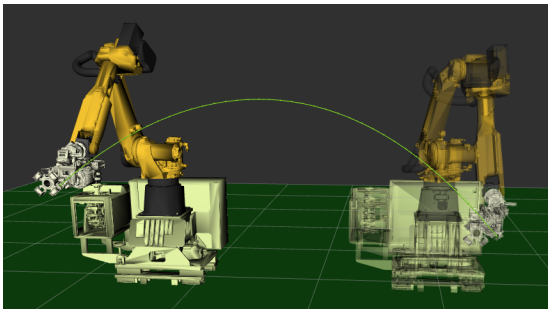
(a) *tight_left.db3*



(b) *low_ground.db3*



(c) *low2high.db3*



(d) *basicplane_lowoblique.db3*

Figure 5.2: Four parabolas considered after task space tuning phase in RViz

5.3.2 Metrics used: Mean Square Error (MSE)

While describing the tuning of the gains for the task space controller, a "MSE-based analysis module" was also cited and said to be used to evaluate the goodness of the gains. Generally speaking, such a tool can be used to quantify the goodness of a controller, but since in this case a task space controller was developed, it would be useful to check whether the controller follows a task space-defined trajectory, rather than checking the joint errors, which means that it needs to differ from the analysis module described in 4.2.

Once more the analysis module is developed as a MATLAB script which includes an interface that allows the user to load the bag files and to extract the XYZ coordinates and the RPY angles. The module is defined in the *MSEoperativo.m* file and, upon execution through the terminal, it will ask the user to give the name of the trajectory to analyze. Upon receiving the name, the script opens two bagfiles: one containing the desired sequence of poses (described through positions and orientations) as planned, and one containing the actual sequence of poses obtained through the execution of the trajectory on the dynamic simulation in Ignition Gazebo. After retrieving the values for each point (in the case of the RPY angles it makes use of a function from the MATLAB Aerospace Toolbox to turn quaternions into angles), the module computes the

Mean Square Error for each pose of the trajectory in terms of XYZ positions and RPY angles. After that, for each position and angle variable, two plots are made: the first one is the plot of the actual value of the position and the desired one, and the second one is the plot of the MSE for each point of the trajectory.

In the case of the test and validation trajectories, upon using the previously described MSE-based analysis module for task space, a critical analysis of the controller with a numeric base can be done.

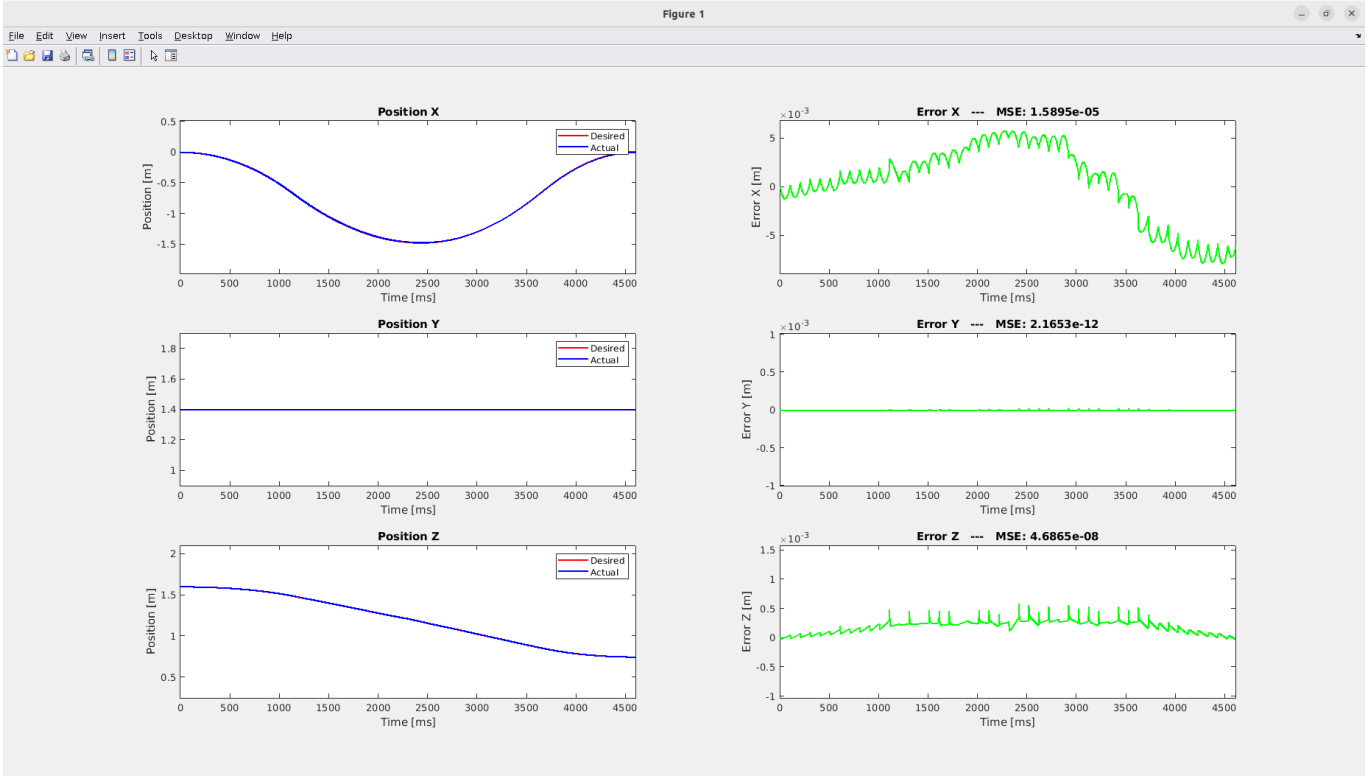


Figure 5.3: *basic.db3* XYZ positions

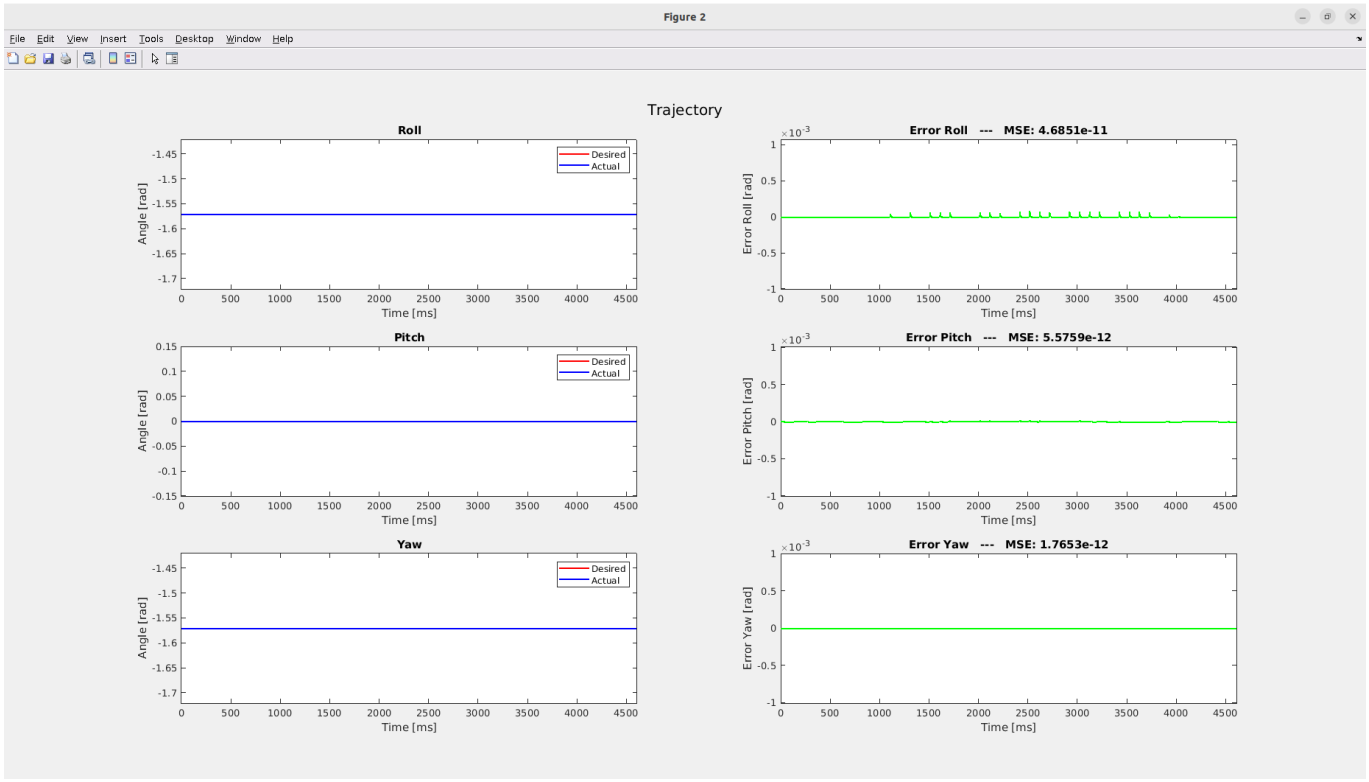


Figure 5.4: *basic.db3* RPY angles

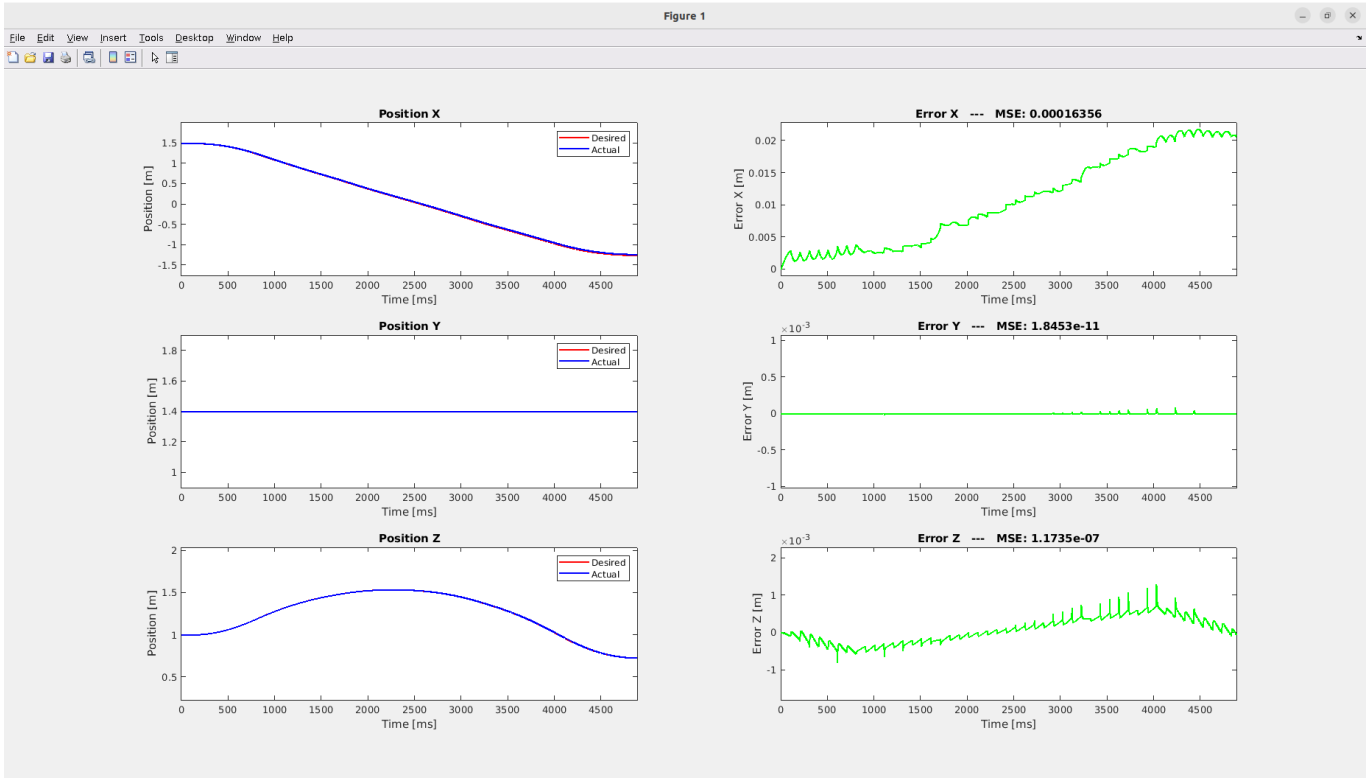


Figure 5.5: *basicplane_lowoblique.db3* XYZ positions

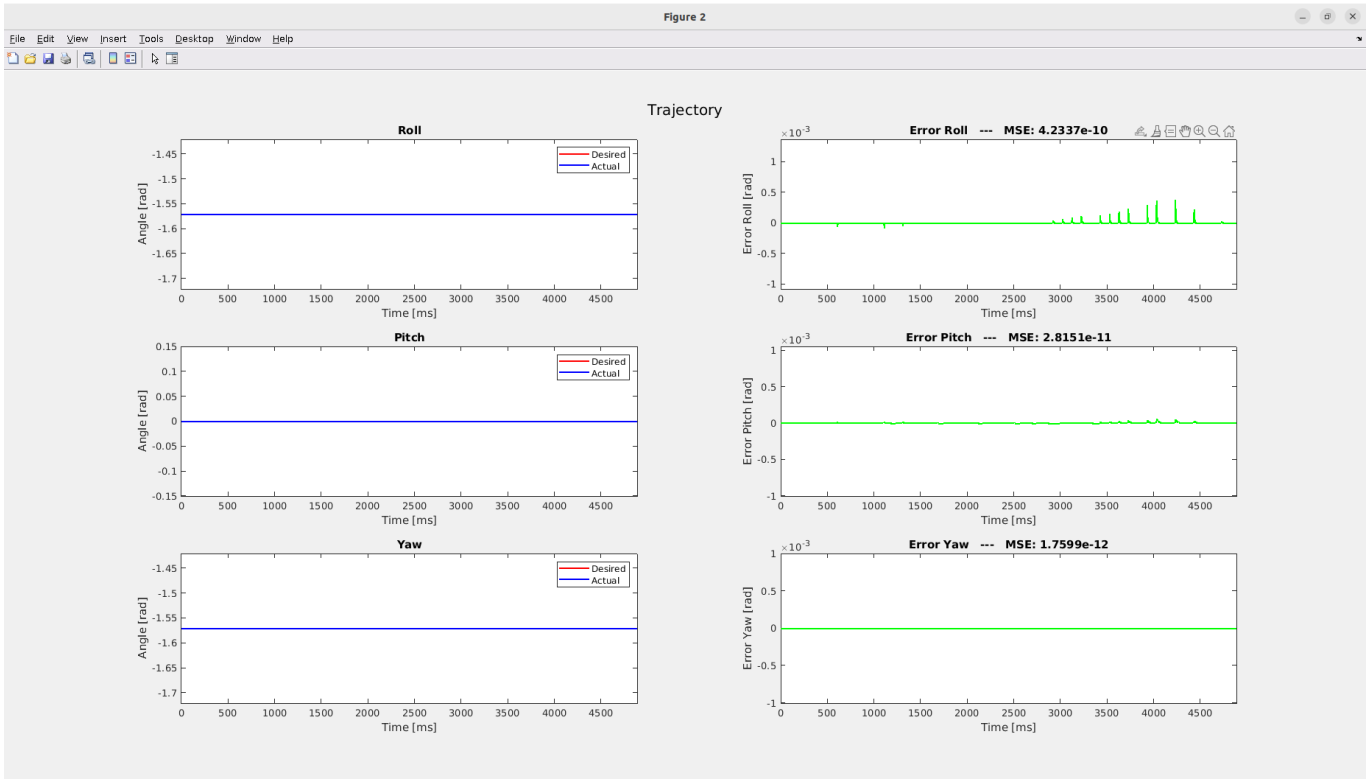


Figure 5.6: *basicplane_lowoblique.db3* RPY angles

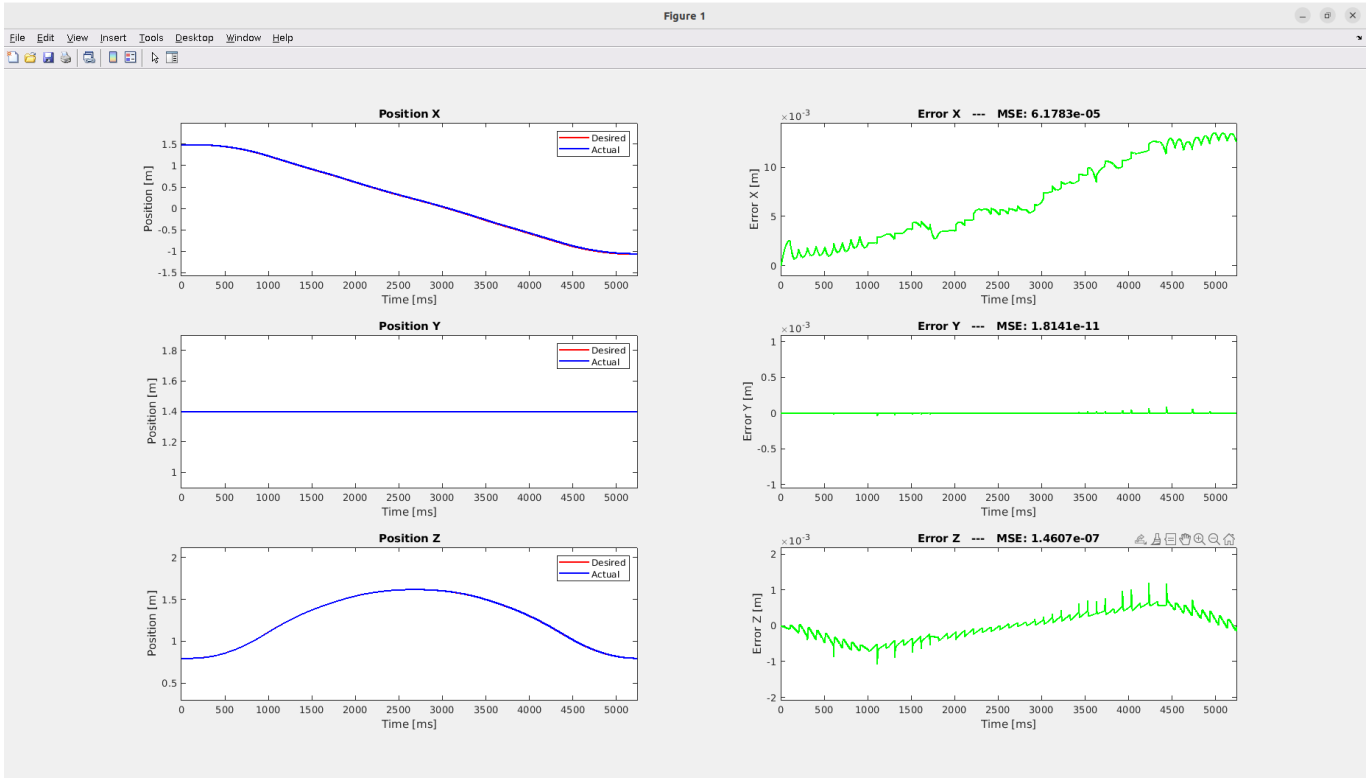


Figure 5.7: *horizontal_low.db3* XYZ positions

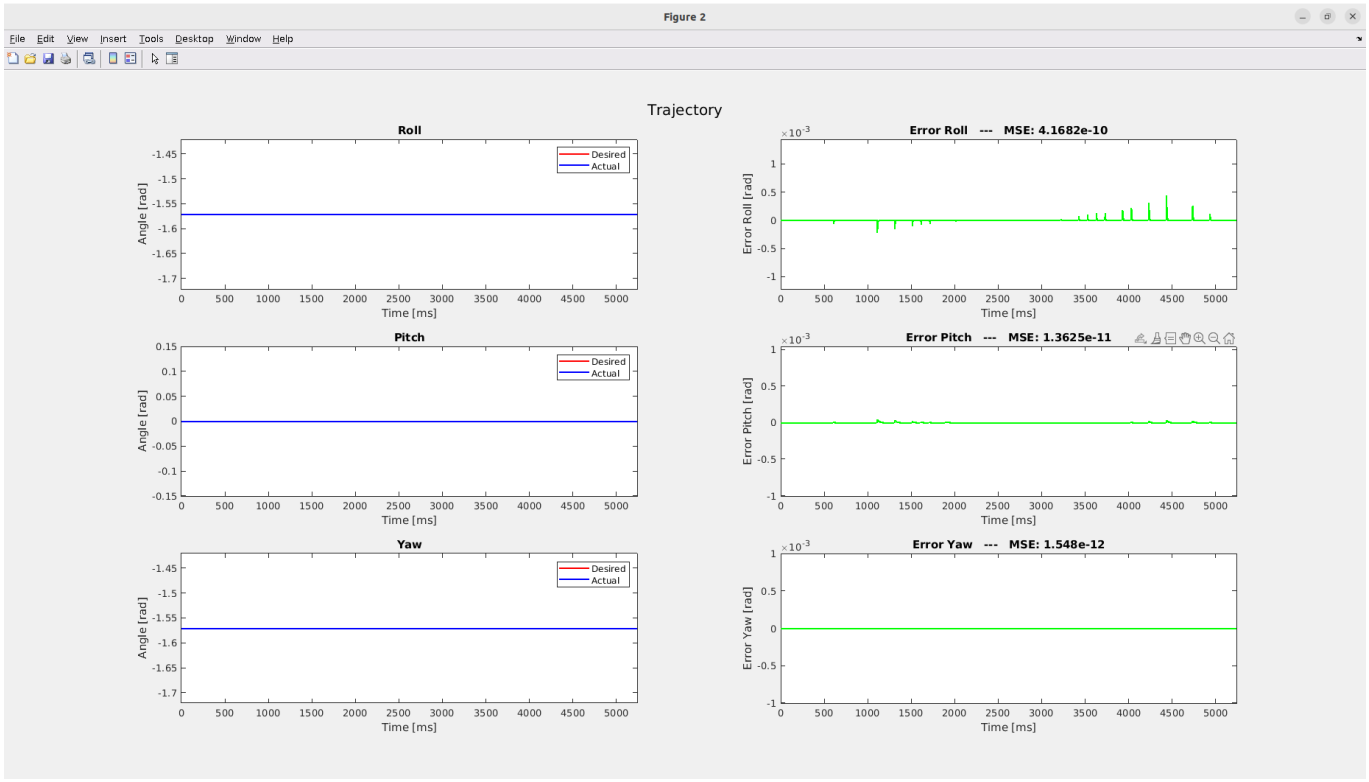


Figure 5.8: *horizontal_low.db3* RPY angles

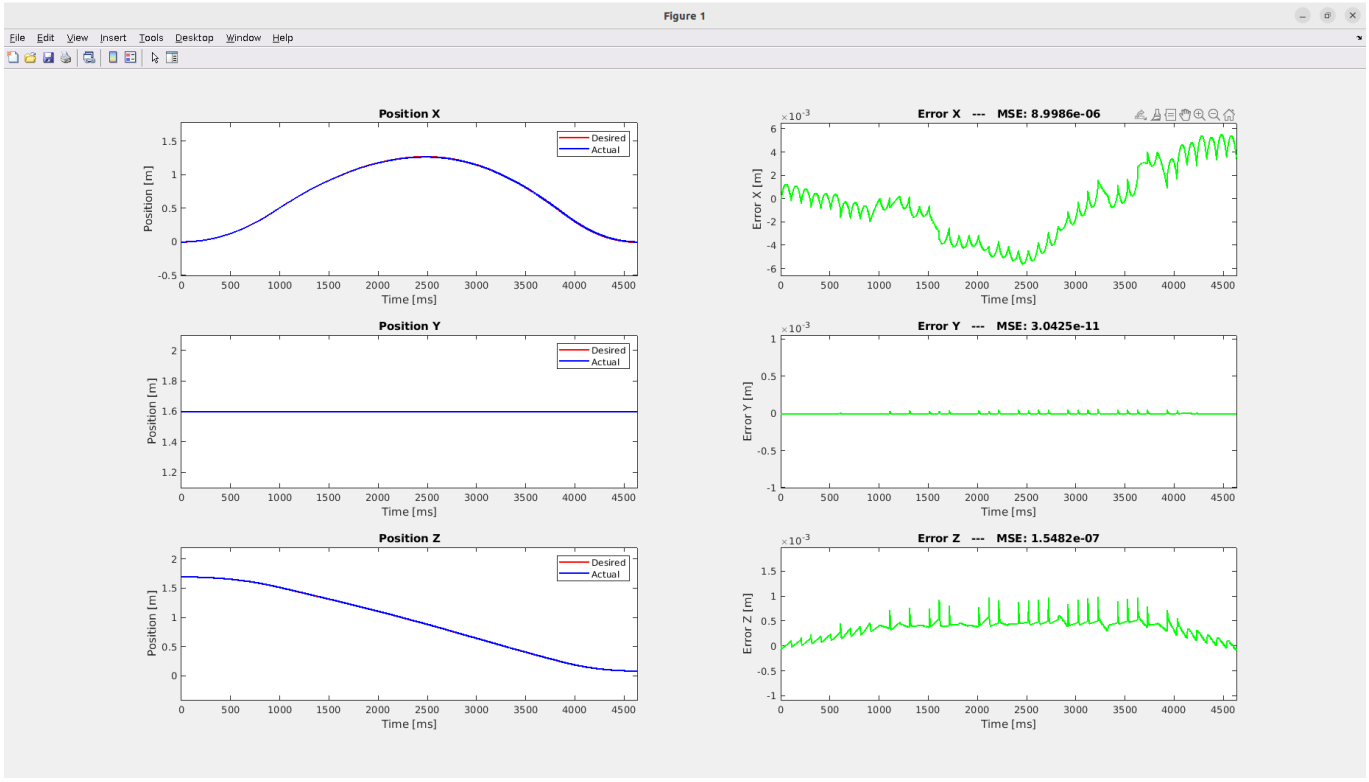


Figure 5.9: *low_ground.db3* XYZ positions

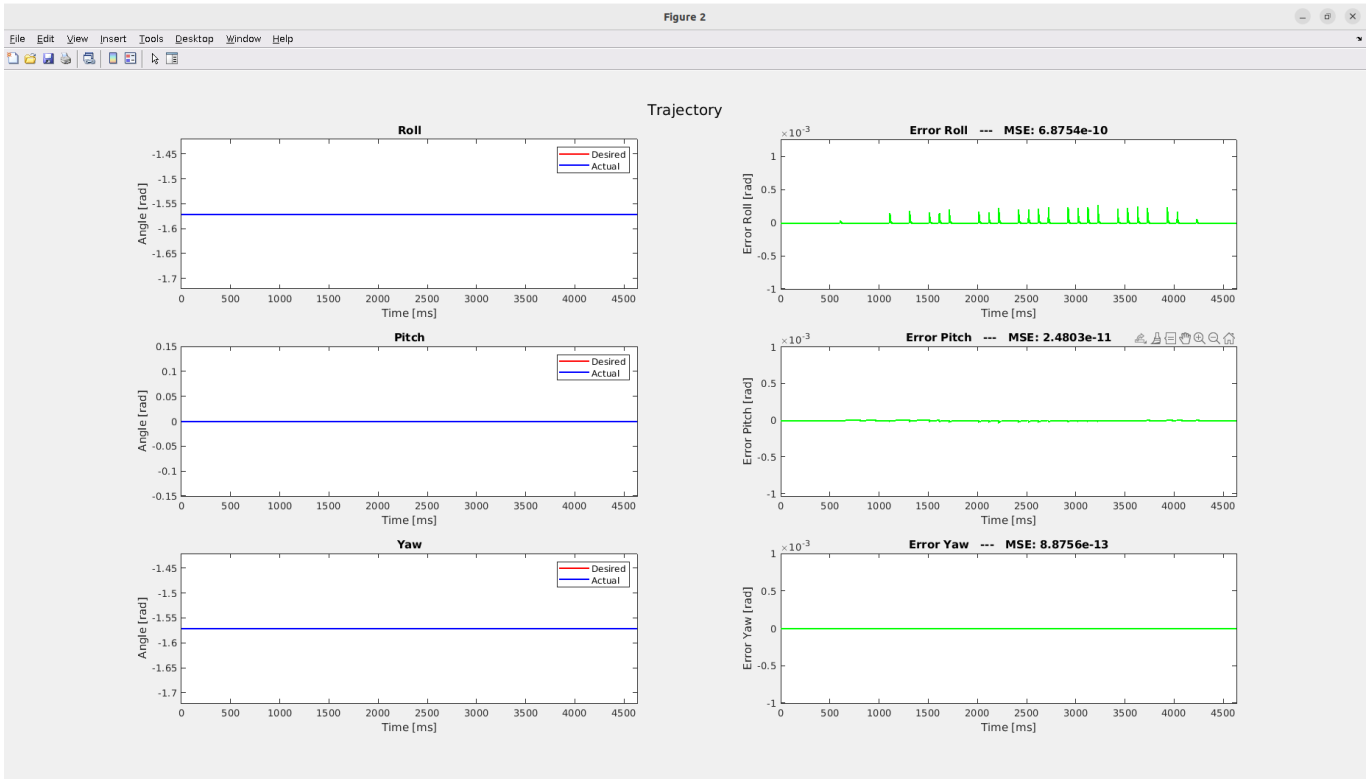


Figure 5.10: *low_ground.db3* RPY angles

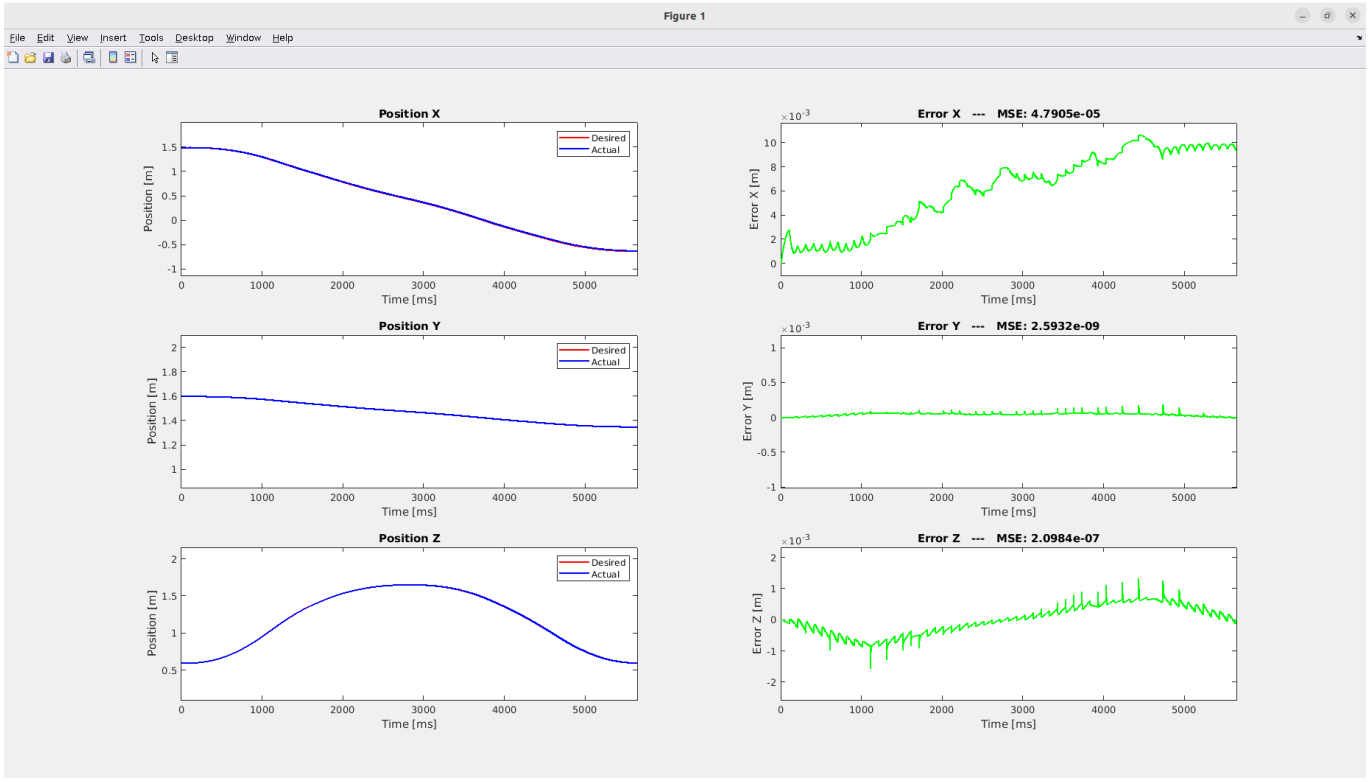


Figure 5.11: *low2high.db3* XYZ positions

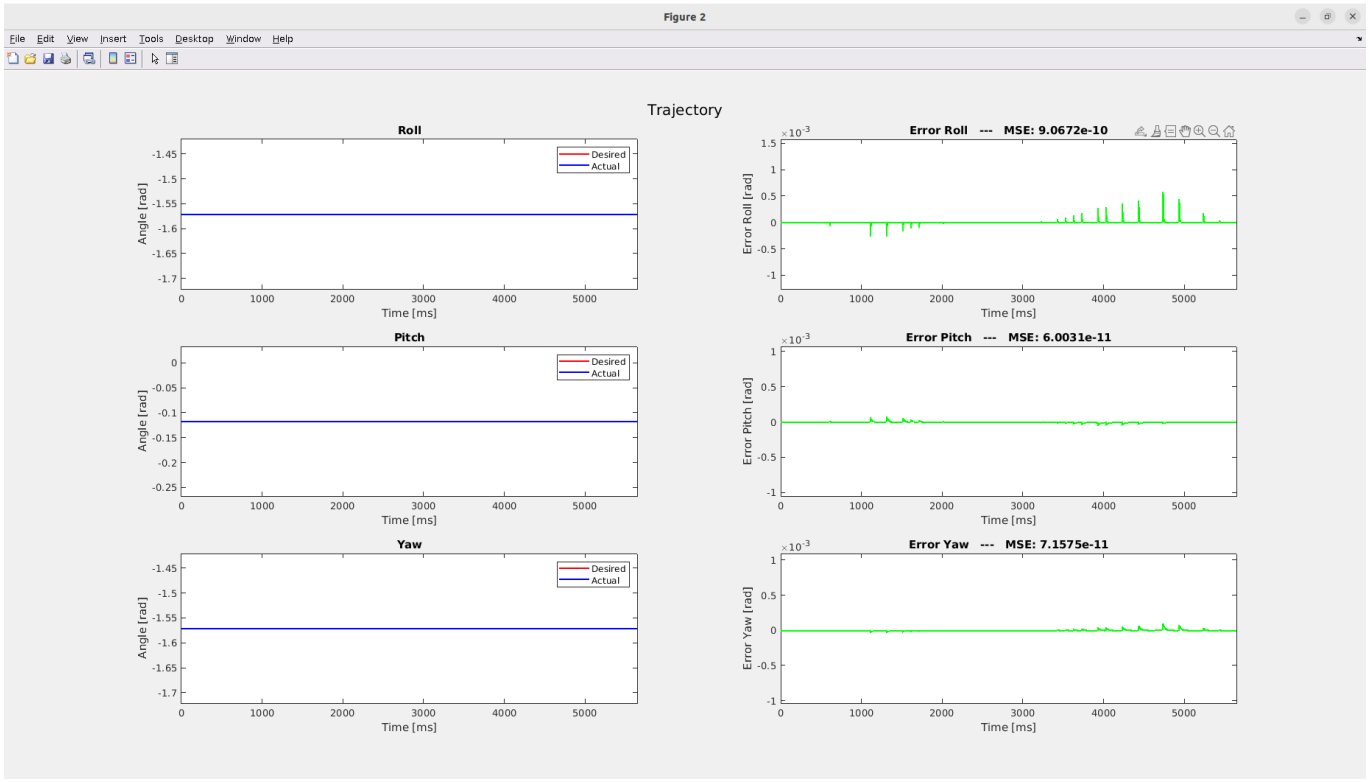


Figure 5.12: *low2high.db3* RPY angles

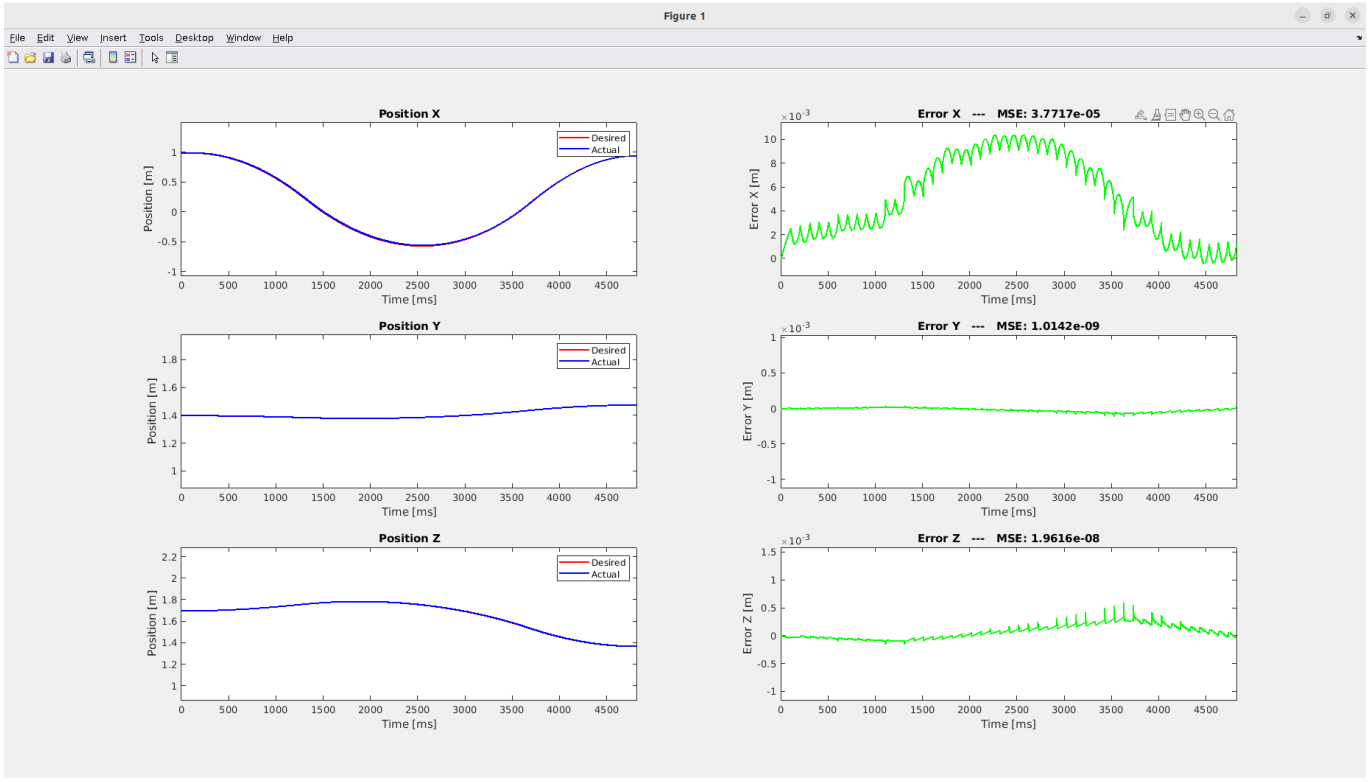


Figure 5.13: *tight_left.db3* XYZ positions

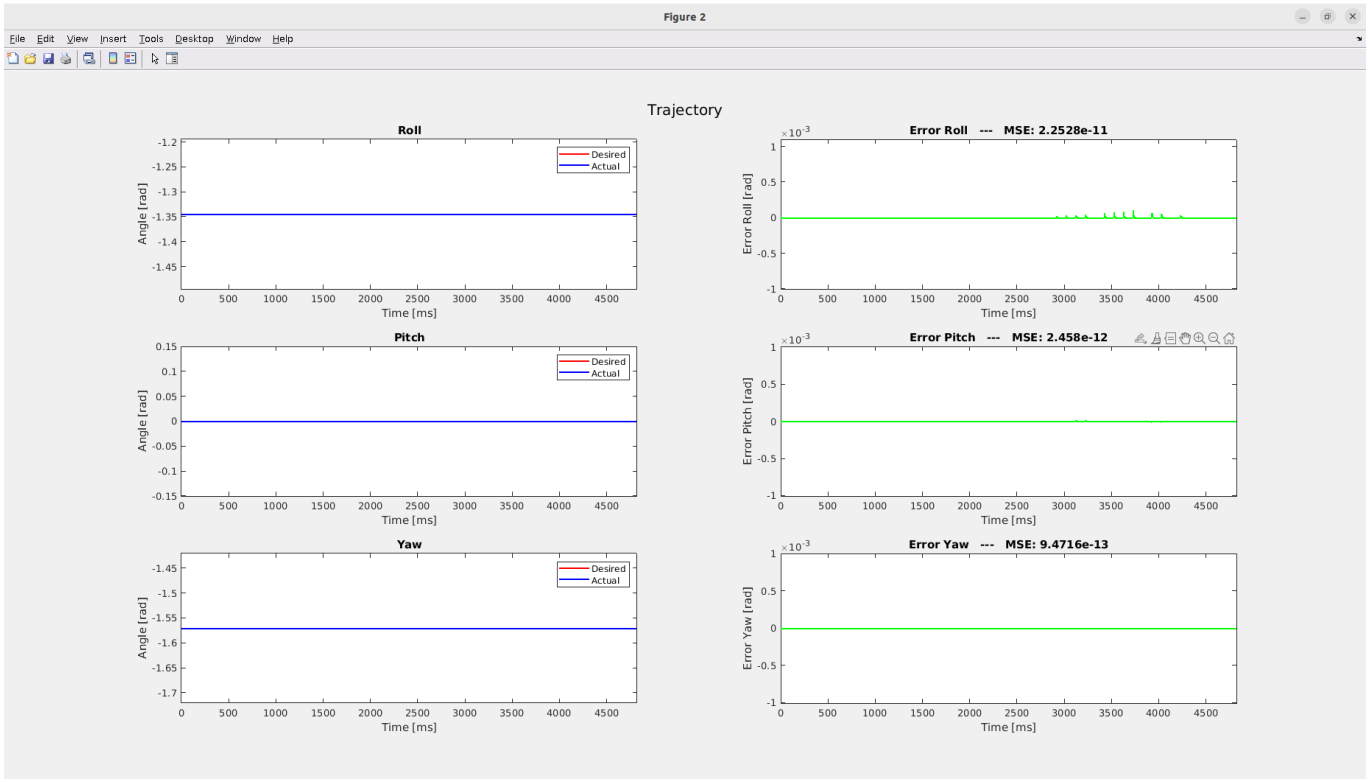


Figure 5.14: *tight_left.db3* RPY angles

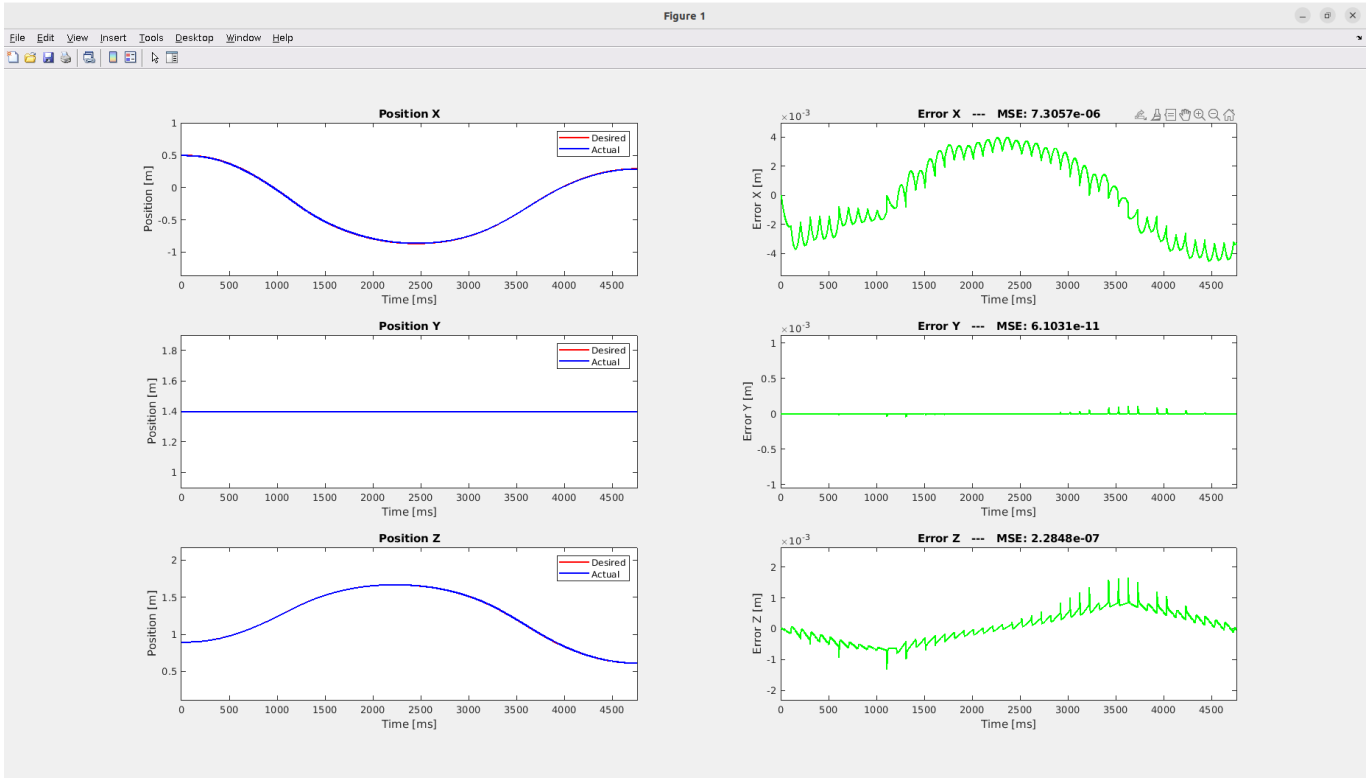


Figure 5.15: *yrotatedplane.db3* XYZ positions

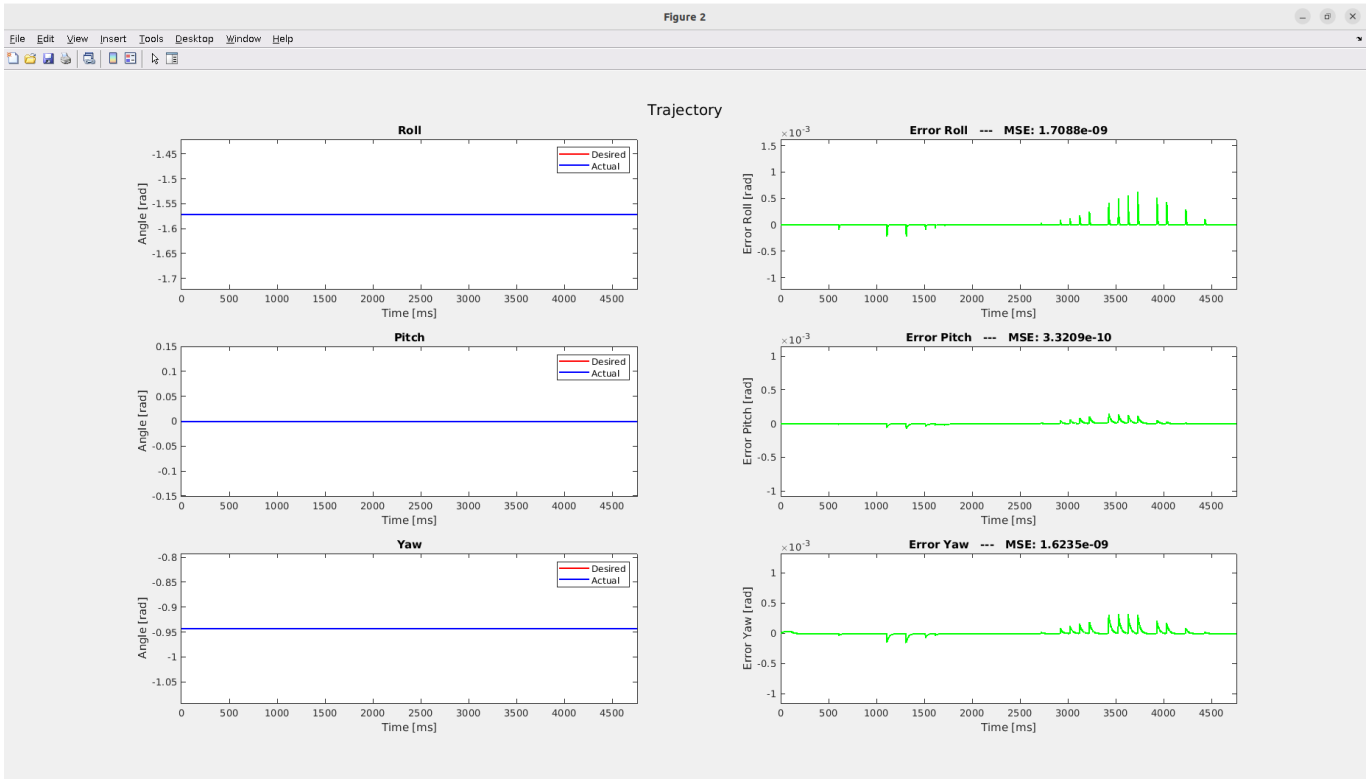


Figure 5.16: *yrotatedplane.db3* RPY angles

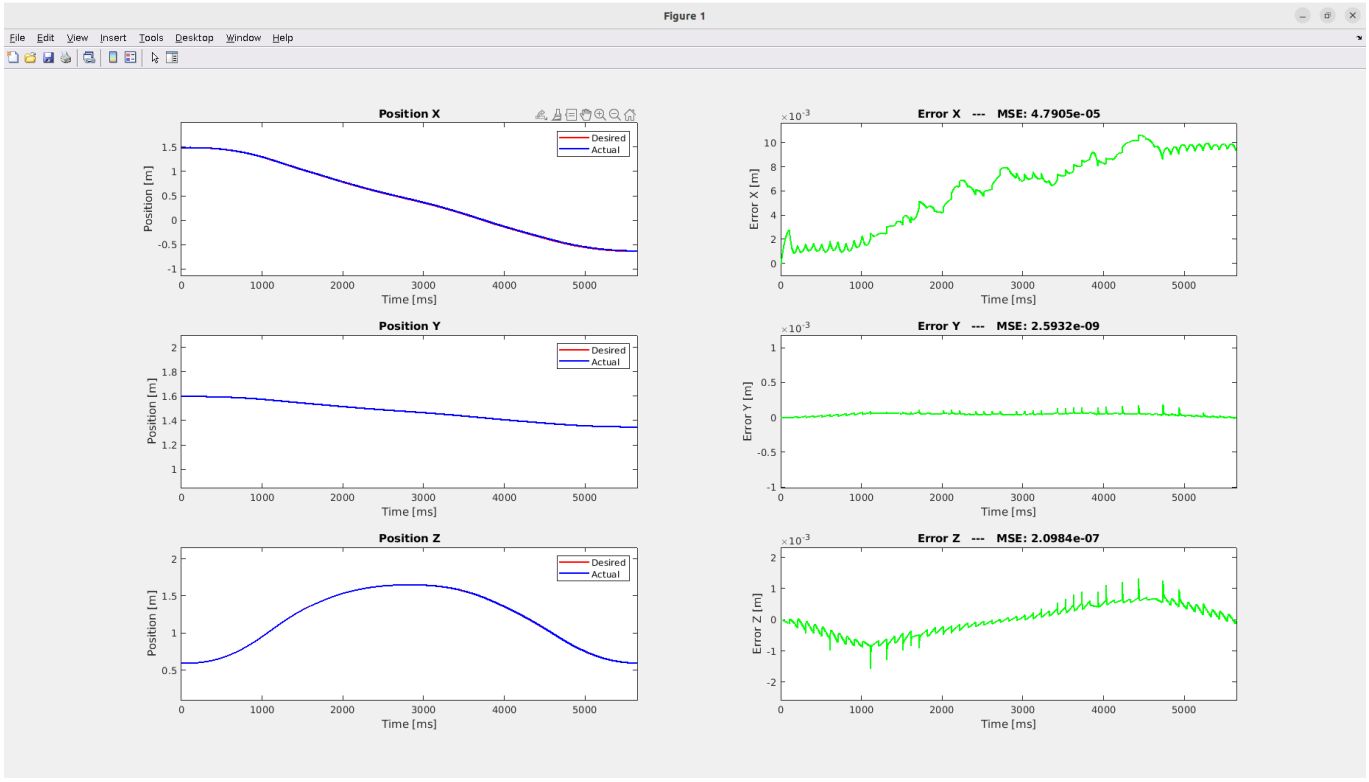


Figure 5.17: *zrotatedplane.db3* XYZ positions

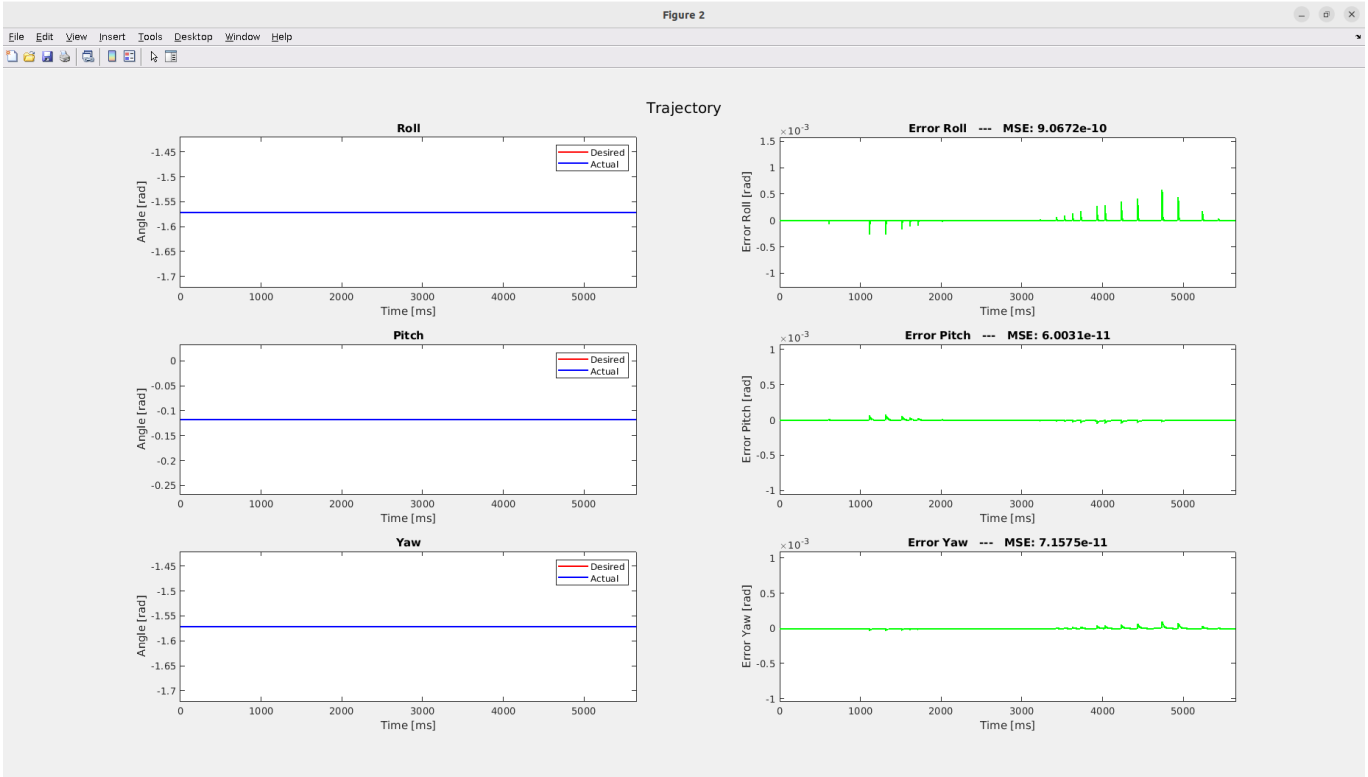


Figure 5.18: *zrotatedplane.db3* RPY angles

Thanks to these plots it is possible to appreciate how in most cases the error never exceeds very low orders of values, as:

- for orientation values, the MSE ranges from the 10^{-7} order to the 10^{-12} one;
- for position values, the MSE ranges from the 10^{-4} order to the 10^{-11}.

This means that the controller is able to follow the desired trajectory with a very high precision, espicially in the case of the orientation, where the error is almost negligible.

5.4 Final simulations

Finally, given the extension of the planning module and the creation of the new task space velocity controller, a new way of running the entire system is required. In order to accomplish this the *demo.launch.py* launch file inside the `fanuc_m20ia_35m_planning_demo` package has been updated, and the requirement for arguments while running the launch file has been added. To be more specific, a command in the form of

```
ros2 launch fanuc_m20ia_35m_planning_demo demo.launch.py joint_model_ group_name:=<joint_model_group_name>
base_link:=<base_link> end_effector_ link:=<end_effector_link> [continous_planning:=<continous_planning>]
controller_action:=<controller_action> work_space:=<work_space> [trajectory_name:=<trajectory_name>]
```

should be run, with the arguments containing the following possible values:

- `joint_model_group_name` can be any one of the 2 defined planning groups

- `base_link` can be any one of the links of the robot (the really used value is the `world` link)
- `end_effector_link` can be any one of the 2 possible end effector links
- `continous_planning`, which is an optional argument, can be 1 or 0 according to whether the user wants the planning to keep on until forcing the closure of the node (1) or for it to be a one shot planning (0)
- `controller_action` can be one of the 2 possible names for the action server
  - `follow_joint_trajectory` for joint space control
  - `follow_cartesian_path` for task space control
- `trajectory_name` is an optional argument that can be used to specify the name of the trajectory to be executed
- `work_space` can be 1 or 0 according to whether the user wants to use task space control (1) or joint space control (0)

The last argument is the one that allows the launch file to order the architecture to work according to the desired configuration, as the value of the `work_space` argument is used to set the value of flag `use_effort_interface` which is then passed as an argument to the command that runs the *moveit_gazebo_ros2_control_demo.launch.py* launch file in the `acg_resources_fanuc_m20ia_35m_moveit_config` package. The *moveit_gazebo_ros2_control_demo.launch.py* launch file has also been updated to check any input arguments, and it is through this mechanism that the launch file loads the *fanuc_m20ia_35m.urdf.xacro*, which in turn accesses the *fanuc_m20ia_35m.ros2_control.xacro* file and sets the command interfaces of all the joints to `"velocity"` if the `use_effort_interface` flag is set to 0, otherwise they are set to `"effort"`. Furthermore, it is in the *fanuc_m20ia_35m.ros2_control.xacro* file that the correct controller configuration is set:

- if the `use_effort_interface` flag is set to 1, the *ros2_controllers_effort.yaml* controller configuration file is loaded
- if the `use_effort_interface` flag is set to 0, the *ros2_controllers.yaml* or the *ros2_controllers_fastening.yaml* controller configuration file , depending on the joint model group specified, is loaded

In order to finally run the simulations and make the robot execute the trajectories using any of the 2 required types of control, the user must first get into the folder containing all the packages, then run

```
colcon build
```

and then

```
source install/setup.bash
```

After this, only the launch command must be run to boot up the system. Here are 2 examples of how to boot up the planning and simulation system using the extended version (be careful to run the command from the `unisa_acg_ros2_group2` folder):

- `ros2 launch fanuc_m20ia_35m_planning_demo demo.launch.py joint_model_group_name:=fanuc_m20ia_35m_slide_to_drilling base_link:=world end_effector_link:=drilling_tool controller_action:=fanuc_m20ia_35m_slide_to_end_effector_controller/follow_joint_trajectory work_space:=0`

  this way a continous planning with joint space control of the drilling tool is used

- `ros2 launch fanuc_m20ia_35m_planning_demo demo.launch.py joint_model_group_name:=fanuc_m20ia_35m_slide_to_drilling base_link:=world end_effector_link:=drilling_tool controller_action:=fanuc_m20ia_35m_slide_to_end_effector_controller/follow_cartesian_trajectory work_space:=1`

  this way a continous planning with task space control of the drilling tool is used

Of course this is valid also for the fastening tool, as if the user wants to control the robot by taking into account the fastening tool instead of the drilling one they have to type `fanuc_m20ia_35m_slide_to_fastening` as value for the `joint_model_group_name` argument and `fastening_tool` as value for the `end_effector_link` argument.

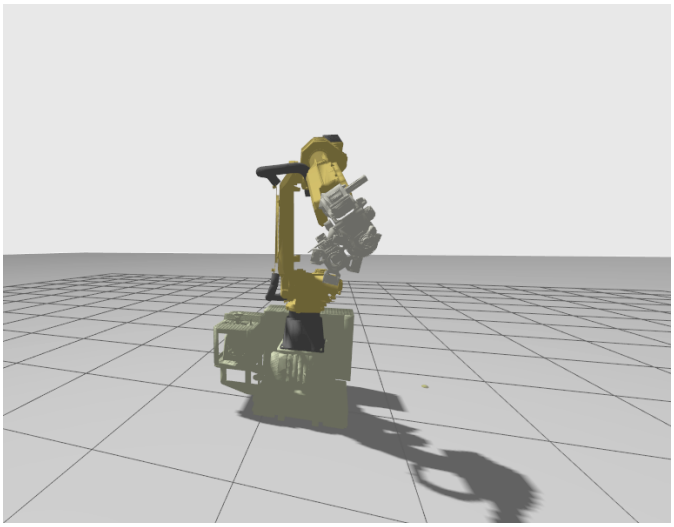


Figure 5.19: Fanuc robot inside the Ignition Gazebo environment, following the *yrotatedplane* trajectory with the task space controller.

In order to use one of the trajectory analysis modules, the user must simply run one of the 2 analysis modules between *`MSEgiunti.m`* (for joint space control) and *`MSEoperativo.m`* (for task space control) in the *matlab_scripts* folder, as if they do not run them in that folder they will not work as they search for the trajectory bagfiles through relative paths.

5.5 Final considerations

By the end of this chapter and of the entire project work a control system based on both joint space and task space control for the Fanuc robot with analysis modules is finally available for the ROS environment, allowing any other type of project that requires the usage of the Fanuc robot to have a digital twin for the robot in ROS2 to start working with. For the specific scope of the project work, all the requirements have been met and all the design choices regarding the most critical aspects have been described. A possible upgrade on the final system, other than the other possible choices regarding the planning pipeline, includes real-time computing to increase the robot's reliability and reduce most elements of non-determinism in the system.

REFERENCES

[1] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2010. ISBN: 1849966346.

[2] Fanuc Corporation. *Fanuc Robot M-20iA*. 2007. URL: https://www.fanuc.com/fvl/vn/product/catalog/RM-10iA(E)-07.pdf.

[3] Fanuc Corporation. *Fanuc Robot M-20iA/35M*. URL: https://www.fanucamerica.com/cmsmedia/datasheets/M-20iA_35M%20Product%20Information_299.pdf.

[4] Antonio Annunziata. "Parametric identification in configurable workcells". PhD thesis. University of Salerno, 2023.

[5] Open Source Robotics Foundation. "Inertial parameters of triangle meshes". In: *Gazebo: An Open-Source Multi-Robot Simulator* (2014). URL: https://classic.gazebosim.org/tutorials?tut=inertia&cat=build_robot.

[6] Konstantinos Karapas. *Disable approximate solutions*. Tech. rep. 2023. URL: https://github.com/ros-planning/moveit_task_constructor/issues/443.

[7] Ros Documentation. *KDL Documentation*. URL: https://docs.ros.org/en/indigo/api/orocos_kdl/html/namespaceKDL.html.

[8] Lars Henning Kayser. *Optimizing MoveIt*. Tech. rep. MoveIt Chief Architect, 2023. URL: https://roscon.ros.org/2023/talks/Optimizing_MoveIt_-_Costs_Constraints_and_Betterments.pdf.

[9] Sebastian Jahr. "Parallel Planning with MoveIt 2". In: (2023). URL: https://moveit.ros.org/moveit%202/parallel%20planning/motion%20planning/2023/02/15/parallel-planning-with-MoveIt-2.html.

[10] Davide Faconti. *Plot Juggler*. URL: https://github.com/facontidavide/PlotJuggler.

[11] Ros Documentation. *Writing a new controller*. URL: https://control.ros.org/humble/doc/ros2_controllers/doc/writing_new_controller.html.

LIST OF FIGURES