UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Corso di Laurea Magistrale in Ingegneria Informatica

Robotics Project Work Report

Group 4

Authors:

Antonio LANGELLA
Mat. 0622702011
a.langella31@studenti.unisa.it

Salvatore PAOLINO
Mat. 0622702016
s.paolino6@studenti.unisa.it

Michele MARSICO
Mat. 0622702012
m.marsico10@studenti.unisa.it

Adamo ZITO
Mat. 0622702026
a.zito32@studenti.unisa.it

ANNO ACCADEMICO 2023/2024

CONTENTS

INTRODUCTION

This report is the result of the final project of the course of Robotics. The objectives of the project are:

- design a ROS2-based planning system that allows planning for sequences of points in the task space;
- execute trajectories on a simulated robot by using independent joint control;
- assess the performance of the controller in terms of joint space tracking;
- execute trajectories on a simulated robot by using task space control;
- assess the performance of the controller in terms of task tracking.

For the project, the robot used is the Fanuc M20iA/35M, that is not natively supported by ROS2 [1], therefore the project work aims at creating the necessary ROS2 support, to allow for kinematic calculations, planning of trajectories and simulation in Ignition Gazebo. The project is divided into four work packages (WPs):

- WP1: create a URDF model of the Fanuc robot that can be visualized in RViz and integrate the kinematic description of the robot with semantic description;
- WP2: implement the planning strategy to generate a task space trajectory and turn it into a joint space trajectory, with the necessary components to generate references at the controller frequency;
- WP3: extend the system in terms of robot description and configuration to accommodate dynamic simulations, implement the controller, the collection of measurements coming from the plant and store them for performance analysis;
- WP4: extend the planning system to be able to generate references in the task space, develop a new controller to perform task space control and extend the trajectory analysis system to analyze the task space error.

CHAPTER 1

WP1 - SYSTEM SETUP FOR PLANNING WITH MOVEIT!

This chapter is dedicated to the development of the description package `acg_resources_fanuc_m20ia_35m_description` as well as the moveit configuration package `acg_resources_fanuc_m20ia_35m_moveit_config`. Section 1.1 serves as an introduction to the robotic manipulator we are working with. Section 1.2 describes the development of the robot description package and more specifically the design choices and the conventions used to produce the URDF file. Section 1.3 describes some problems we encountered with the provided meshes and how we fixed them. Section 1.4 describes the choices we made when generating the MoveIt! configuration package as well as the manual changes we made to the generated files.

1.1 Robot overview

The robot being considered is a FANUC M-20iA/35M, an anthropomorphic robot manipulator with an open kinematic chain and 6 degrees of freedom given by 6 rotational joints. The robot is shown in Fig.1.1. Given the number of degrees of freedom, the robot is capable of performing tasks in which both position and orientation of the end effector are completely specified. However, for such tasks the robot is not redundant.

The manipulator is mounted on a mechanical slide base (Fig. 1.2), which allows the robot to move along a linear axis. From a kinematic point of view, the slide can be seen as a prismatic joint. This means that by including the slide, the robot has 7 degrees of freedom, and it becomes redundant. Adding the slide to the kinematic chain of the robot has several advantages:

- The robot can reach a larger workspace, which is useful when performing tasks that require the robot to move along a linear axis. The slide can also mounted on a wall, both horizontally and vertically, and this allows the robot to reach more points in the workspace;
- The redundancy can be exploited to optimize some metrics when planning the robot's

Figure 1.1: FANUC M-20iA/35M

Figure 1.2: Mechanical slide base

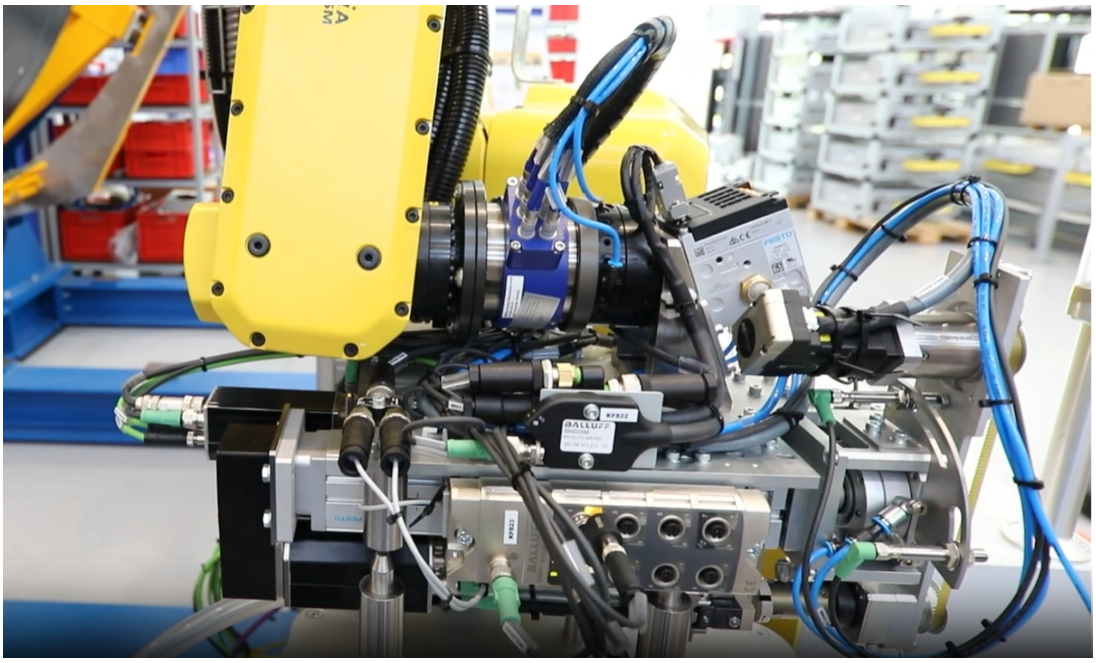


Figure 1.3: End effector

trajectory, for example by minimizing the energy consumption or the time required to perform a task;

- The redundancy is also useful when controlling the robot in task space. In this context redundancy implies that the null space of the Jacobian is not empty, and this allows joint movements that do not affect the end effector pose. This can also be exploited to optimize some metrics, for example by avoiding obstacles or singularities. The section 4.2.4 shows how redundancy was exploited both to keep the joints away from their limits and to maximize the extent of manipulability of the robot;

The considered manipulator also mounts a multi-tool end effector (Fig. 1.3). The end effector is equipped both with a drilling tool and a visual inspection tool. The end effector was built for the assembly process of fuselage parts of an airplane, but this is not relevant for the scope of this work project. Adding the end effector to the kinematic chain of the robot greatly increases the complexity of the trajectory planning problem described in Chapter 2. This problem arises from the fact that the end effector is very bulky, and it limits joints 5 and 6 in their movements. Joint 6 is particularly limited in its movements and this means that a degree of freedom is almost lost. This can lead to undesired artifacts in planned trajectories. The presence of the end effector also leads to decreased performances of the task space controller as described in Chapter 4. This is due to the fact that the end effector can easily collide with link 5 of the arm, and this leads to unexpected behaviors when controlling the robot with the chosen control technique, as this algorithm is supposed to be used in a collision-free environment. To overcome this problem, an attempt was made to exploit the redundancy of the robot as described in the section 4.2.4.

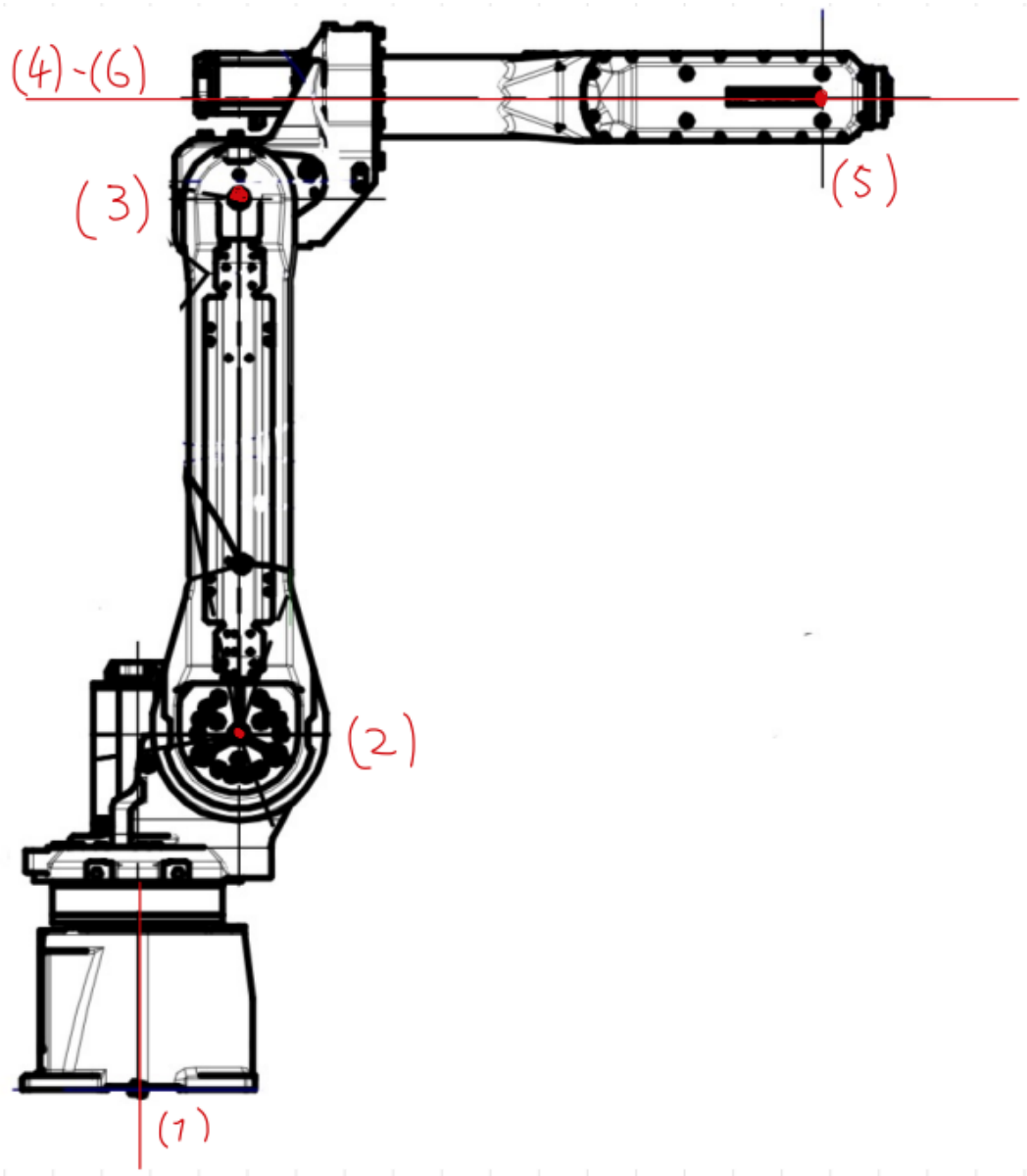


Figure 1.4: Robot rotation axis

1.2 Robot description package

1.2.1 Kinematic description of the robotic manipulator

For the kinematic description of the robot, the robotic manipulator was first considered without the slide and end effector, and then added later. For this reason, in the following we will focus on the kinematic chain of the robotic manipulator.

Joint axes First, we identified the axes of rotation of the 6 revolute joints of the robot, as shown in Fig.1.4. Notice how in the shown configuration, axes 4 and 6 coincident.

Denavit-Hartenberg frames In order to describe the kinematics of the robot, we used the Denavit-Hartenberg (DH) convention [2]. According to these conventions, the following rules were applied:

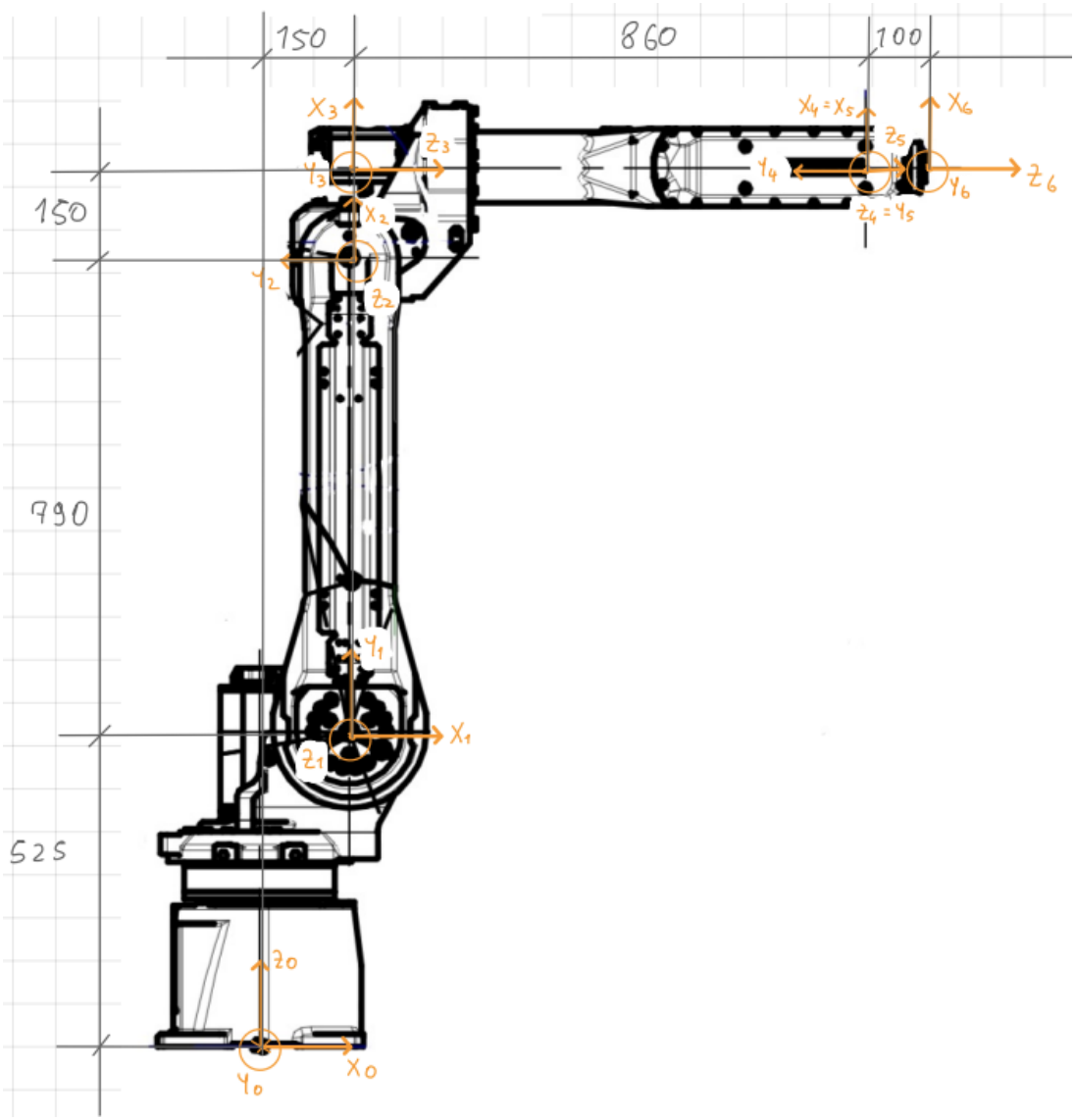


Figure 1.5: Denavit-Hartenberg frames for the robotic manipulator

- The z_i axis is placed along the axis of motion of the $i+1$-th joint;
- The origin of the frame i is placed at the intersection of the common normal between the z_{i-1} and z_i axes with the z_i axis;
- Axis x_i is placed along the normal common to the axes z_{i-1} and z_i directed from joint i to joint $i+1$;
- The y_i axis is chosen to complete a right-handed orthonormal reference frame.

By applying these rules recursively, we assigned a reference frame to each joint of the robot. The resulting frames are shown in Fig. 1.5. Notice that sometimes these rules leave some freedom in the choice of the frame, so in the following we will describe the cases where we have freedom to choose the frame:

Frame 0

- For the z_0 axis we chose the upward direction;
- Since it is the first joint, there is no previous axis and no common normal, so we have freedom in the choice of both the origin O_0 and the x_0 axis, as long as x_0 is normal to z_0. We chose to place the origin at the lowest part of the robot base and the x_0 axis toward the right.

Frame 1

- For the z_1 axis we chose the outward direction (with respect to the drawing plane);

Frame 2

- For the z_2 axis we chose the outward direction (with respect to the drawing plane);
- Since joint axes 2 and 3 are parallel, we have freedom in the choice of the common normal. In order to simplify the calculations, we chose the common normal to be along the arm, so that the origin O_2 is at the same depth as the origin O_1. The extension of the common normal is the x_2 axis.

Frame 3

- We directed the z_3 axis towards the right;

Frame 4

- We directed the z_4 axis towards the right;
- As joint axes 4 and 5 are orthogonal and incident, there is only one common normal but the direction of the x_4 axis can be either upward or downward. We chose the upward direction.

Frame 5

- We directed the z_5 axis towards the right;
- As in the previous case, joint axes 5 and 5 are orthogonal and incident, so we have freedom in the choice of the direction of the x_5 axis (upward or downward). For simplicity, we chose the same direction as the x_4 axis, which is upward.

Frame 6

- According to the first Denavit-Hartenberg rule the axis z_6 must be positioned along the joint axis $i+1 = 7$, but this axis does not exist, so we could position it arbitrarily;
- According to the third rule axis x_6 must be prolongation of the common normal, so in this case the only constraint is that it should be positioned orthogonally with respect to the axis z_6;
- We chose the "joint axes 7" to be coincident with the joint axes 4 and 6, so that the axis z_6 is concordant with the previous frame;
- We directed the z_6 axis towards the right;

	a_i	d_i	α_i	θ_i
1	0.150	0.525	$\frac{\pi}{2}$	q_1
2	0.790	0	0	q_2
3	0.150	0	$\frac{\pi}{2}$	q_3
4	0	0.860	$-\frac{\pi}{2}$	q_4
5	0	0	$\frac{\pi}{2}$	q_5
6	0	0.100	0	q_6

Table 1.1: Denavit-Hartenberg parameters

– Since there are infinite common normals between the axes 5 and 6, we have freedom in the choice of the origin O_6 and the direction of the x_6 axis, as long as it is normal to z_6. For simplicity, we chose to place the origin at the end of the flange and the x_6 axis upward (to be consistent with the previous frame).

Denavit-Hartenberg parameters Once the frame are defined, the position and orientation of the frame i with respect to the frame $i-1$ result completely specified by the following parameters:

- a_i: distance between z_i and z_{i-1} axes;
- d_i: coordinate on z_{i-1} of $O_{i'}$ (the projection of O_i on the z_{i-1} axis);
- α_i: angle about the x_i axis between the z_{i-1} axis and the z_i axis evaluated positive counterclockwise;
- θ_i: angle around the z_{i-1} axis between the x_{i-1} axis and the x_i axis evaluated positive counterclockwise.

Parameters a_i and α_i are constant and depend on the geometry of the robot. For revolute joints, the parameters θ_i are equal to the joint variables q_i, while for prismatic joints the parameters d_i are equal to the joint variables q_i. In our case all the joints are revolute. The parameters we identified according to the rules are specified in Tab. 1.1.

Manufacturer's conventions According to Denavit-Hartenberg conventions, a positive rotation q_i is a counterclockwise rotation about the z_{i-1} axis. However, manifacturers not always use this convention, so in order to conform with the robot's specifications, where necessary we need to invert the sign of the joint variable. In Fig. 1.6 we show the manifacturer's conventions for the joint variables. These conventions were extracted from the extended datasheet for the Fanuc M20-iA series [3]. As shown in the figure, Joints 1, 3 and 5 have the same convention as the DH convention, while Joints 2, 4 and 6 have the opposite convention and therefore we need to invert the sign of the associated joint variables.

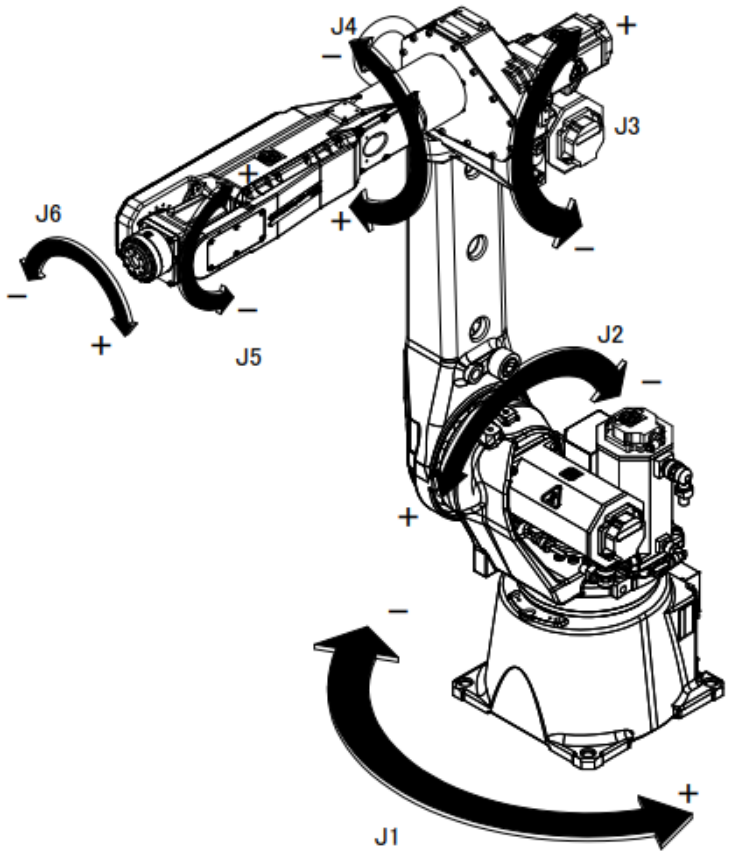


Figure 1.6: Manufacturer's conventions for the joint variables

```
<link name="fanuc_m20ia_35m_link_0">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://acg_resources_fanuc_m20ia_35m_description/meshes/visual/link_0.dae" />
        </geometry>
    </visual>
    <inertial>
        <inertia ixx="0.81917" ixy="0.11076" ixz="-0.15612" iyy="1.46533" iyz="0.02762" izz="1.37957"/>
        <origin rpy="0.00000 0.00000 0.00000" xyz="-0.01251 0.00295 0.13601"/>
        <mass value="55.00000"/>
    </inertial>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://acg_resources_fanuc_m20ia_35m_description/meshes/collision/link_0.stl" />
        </geometry>
    </collision>
</link>
```

Figure 1.7: Example of a link tag in the URDF file

```
<joint name="fanuc_m20ia_35m_joint_2" type="revolute">
    <origin xyz="0.150 0 0.525" rpy="1.57079632679 0 0" />
    <parent link="fanuc_m20ia_35m_link_1" />
    <child link="fanuc_m20ia_35m_link_2" />
    <axis xyz="0 0 -1" />
    <dynamics damping="7.11" friction="1.2"/>
    <limit effort="2500" velocity="3.14" lower="-1.745329" upper="2.792527" />
</joint>
```

Figure 1.8: Example of a joint tag in the URDF file

1.2.2 Kinematic description in the URDF

After obtaining the kinematic description of the manipulator in terms of the DH parameters, we translated it into the URDF format.

Manipulator Links In order to represent the base frame, we defined an empty link named `world` that acts as a global reference frame. Then we defined a `link` element for each of the 7 links of the robot. Each link has both a `visual` and a `collision` mesh defined with the appropriate tags. In order to correctly place the meshes, we used the open source software `Blender` [4] to identify the position and orientation of the mesh frames with respect to the link frame. Then we expressed this displacement in terms of translation and Roll-Pitch-Yaw angles in the `origin` tag. An example of the `link` tag is shown in Fig. 1.7.

Manipulator Joints We defined a `joint` element for each of the 6 joints of the robot. Each joint has a `parent` and a `child` link, and a `origin` tag that specifies the position and orientation of the joint frame with respect to the parent link frame. Notice how the i-th frame we defined in the previous section becomes the frame of the $i+1$-th joint in the URDF file (i.e. the frame 0 is relative to the joint 1 in the URDF file). It is also worth pointing out that the `origin` tag of the $i+1$-th joint in the URDF file contains the DH parameters a_i, d_i and α_i that we defined in the previous section (eventually expressed in terms of RPY angles).
The `joint` tag also specifies the type of joint and the joint axis. In our case the type of all

the manipulator joints is `revolute`. Because we used DH convention, the rotation axis will always z. According to the used convention, positive rotations are always counterclockwise. However, manufacturers not always use this convention, so in order to conform with the robot's specifications, where necessary we need to invert the sign of the joint variable by specifying "`0 0 -1`" instead of "`0 0 1`" in the `axis` tag. An example of the `joint` tag is shown in Fig. 1.8.

Slide and end effector Finally, we added the slide and the end effector to the kinematic chain of the robot in the URDF file. The slide is modeled as a prismatic `joint` between the base frame `world` and the slide `link` element. The latter also contains a `visual` and a `collision` mesh. The junction between the slide and the robot base is modeled as a fixed `joint` between the slide frame and the frame 0 of the robotic manipulator, defined with the appropriate transformation in the `origin` tag.
The end effector is actually implemented as two separate `link` elements: one for the drilling tool and one for the visual inspection tool. Because both of them are in the same mesh, only the drilling tool `link` has a `visual` and a `collision` mesh. Both of them are connected to the flange of the robot with a fixed `joint`, defined with the appropriate transformation in the `origin` tag.

1.2.3 Motors identification

In order to obtain a good estimate of the inertial parameters of the robot as well as the damping and friction parameters, we need to identify the motors used in the robot. All the datasheets of the robot that can be openly accessed online do not contain the model of the motors used in the robot. For this reason, we first looked for the datasheets of the motors produced by Fanuc, and then we tried to match the motors shown in pictures and datasheets of the manipulator with the motors produced by Fanuc. The reference datasheet for the motors produced by Fanuc is the one in [5]. This matching operation was done by comparing the dimensions and the torques of the motors. The result of this operation is that we were able to identify motors that are as similar as possible to those used in the robot and to obtain the datasheets of the said motors. This allowed us to obtain the important parameters such as the mass of the motor, the torque constant, the voltage constant and the armature resistance, that are necessary to estimate the inertial parameters as well as the joint effort limits and the damping and friction parameters of the robot.

1.2.4 Joint limits specification

Next we added the joint limits to the URDF file. The joint limits are specified in the `limit` tag of each `joint` element.

Joint position limits In order to get the joint position limits for the manipulator we referred to the datasheet in [6]. For the slide limits we referred to the paper in [7] whose experimental setup is based on a real-case scenario using the Fanuc M20-iA/35M together with the same slide and end effector we are using.

Joint velocity limits The velocity limits for the manipulator joints easily obtained from the datasheet in [8]. The velocity limit for the slide were obtained from the paper in [7], as for the joint limit.

Joint effort limits The effort limits for the joints 4, 5 and 6 were obtained from the datasheet in [8]. The effort limit for the joints 1, 2 and 3 were chosen to be higher because according to the datasheet they are only limited in velocity. However, we decided to choose the most reasonable values we could produce with the available data, so we assumed a matching between the motors shown in pictures and datasheets of the manuipulator with the motors produced by Fanuc (as described in section 1.2.3) and we estimated the effort limits starting from the maximum torque of the corresponding motors and considering the presence of a gear ratio between the motors and the joints. The effort limit for the slide was considered to be the highest one.

1.2.5 Inertial parameters

Link mass estimates Next, we moved to the inertial parameters of the robot. The first step was to estimate the mass of each link. From the datasheet in [8] we only have the total mass of the robot, that is not comprehensive of the slide and the end effector. In order to estimate the mass of each link, we used the following approach:

1. We estimated the mass of the motors by assuming a matching between the motors shown in pictures and datasheets of the manipulator with the motors produced by Fanuc (as described in section 1.2.3);

2. We estimated the relative mass of each link without the motors by comparing their dimensions. We assumed that all the links have the same density. The result was a list of links from the lightest to the heaviest;

3. We estimated the mass of each link without the motors by considering the total mass of the robot and the mass of each motor. The result was a mass for each link of the manipulator;

4. We estimated the mass of the slide and the end effector independently by considering the maximum payload of the Fanuc M20-iA/35M as reported in the datasheet in [8].

As far as the method we used to estimate the masses of the links was designed to provide the most reasonable estimates we could produce with the available data, it is worth pointing out that in a real life scenario it would be desirable that the datasheet of the robot provides the mass of each link as well as the model of each motor. Alternatively, having access to the real robot, we could use system identification techniques to estimate the mass of each link.

Identification of inertia matrices As the mass of each link was obtained, the next step was to identify the corresponding inertia matrices. In order to do this, we thought of two different approaches:

- Assume that the links are uniformly dense cylinders and use the formulas for the inertia of a cylinder to calculate the inertia matrices;

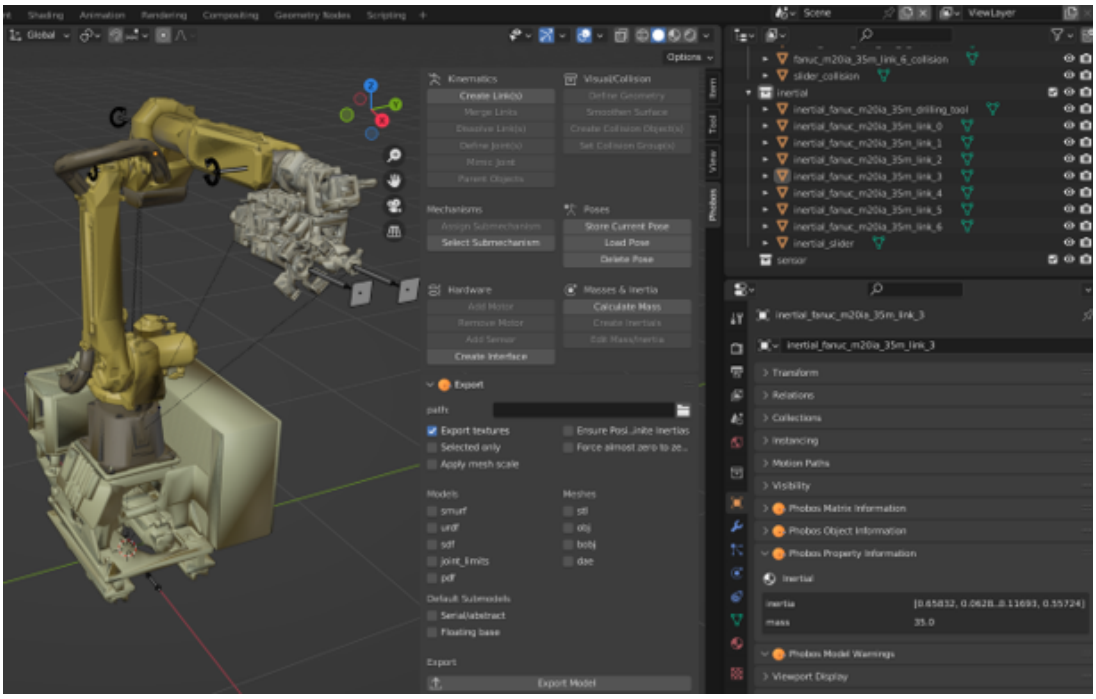


Figure 1.9: Phobos plugin GUI

- Use the information contained in the URDF and the estimated mass of each link to estimate the inertia matrices by means of appropriate piece of software.

We choose the second approach as it was both the easiest and the most accurate of the two. Various tools were explored to obtain the most accurate results.
First we tried **SolidWorks** [9], but due to its inability to convert complex meshes into solid bodies (on which the inertia calculation is based), we had to abandon this tool. We also assumed that the error was due to non-closed meshes, so we tried to close them with **MeshInspector** [10], but this didn't solve the problem.
Second attempt was to try **MeshLab** [11] [12] directly, but the lack of control over the reference frames with respect to which the inertia matrices were calculated made it inconvenient to use this tool, so we abandoned it.
Finally, we tried **Phobos** [13][14], a Blender plugin that allows to load URDF files and calculate the inertia matrices. This tool was the most convenient to use, as it automatically calculates the inertia matrices according to the correct frame, and it is also capable of automatically handling non-closed meshes. The matrices are estimated from the robot meshes that are specified in the URDF file. The tool needs the mass of each link to be specified, so we used the estimated mass of each link to calculate the inertia matrices. The Phobos GUI is shown in Fig. 1.9. Thanks to this tool, that proved to be both the easiest and most powerful of the three, we were able to obtain the inertia matrices for each link of kinematic chain and export them directly in the URDF file.
However, it is worth pointing out that the inertia matrices obtained from Phobos are only estimates based on the estimated masses of the links and the hypothesis that each link is uniformly dense. To obtain more accurate results in a real scenario, it would be better to start from real measurements to identify inertia matrices.

Data sheet

Parameter	Symbol	Value			Unit
Stall Torque (*)	Ts	40 408			Nm kgfcm
Stall Current (*)	Is	36.2			A (rms)
Rated Output (*)	Pr	5.5 7.3			kW HP
Rating Speed	Nr	3000			min^{-1}
Maximum Speed	Nmax	4000			min^{-1}
Maximum Torque (*)	Tmax	115 1170			Nm kgfcm
Rotor Inertia	Jm	0.0099 0.101			kgm^2 $kgfcms^2$
Rotor Inertia (with Brake)	Jm	0.0105 0.107			kgm^2 $kgfcms^2$
Torque constant (*)	Kt	1.10 11.3			Nm/A (rms) kgfcm/A (rms)
Back EMF constant (1phase) (*)	Ke Kv	39 0.37			V (rms)/1000 min^{-1} V (rms)sec/rad
Armature Resistance (1 phase) (*)	Ra	0.058			Ω
Mechanical time constant	tm	0.001			s
Thermal time constant	tt	40			min
Static friction	Tf	1.2 12			Nm kgfcm
Weight	w	28			kg
Weight (with Brake)	w	34			kg
Maximum Current of Servo Amp.	Imax	160			A (peak)

Figure 1.10: Example of a motor datasheet from Fanuc [5]

1.2.6 Estimation of damping and friction parameters

In order to estimate the damping and friction parameters for the robot joints we decided to follow a pragmatic approach because of the notorious difficulty in accurately calculating these parameters, even when the real hardware is available. We decided to leverage the motor data sheets to derive plausible values for friction and damping.

First, we started from the motor identification we described in section 1.2.3. As a result of this procedure, we have reasonable parameters for each of the motors. An example of the parameters we obtained from the motor datasheets is shown in Fig. 1.10.

Notice that while the friction parameter is directly available from the motor datasheet (as "Static friction"), the damping parameter is not. In order to calculate the damping parameter, we used the formula

$$F_d = \frac{k_v k_t}{R_a} \tag{1.1}$$

where k_v is the voltage constant ("Back EMF constant" in the datasheet), k_t the torque constant and R_a the armature resistance. All of these parameters can be found on the datasheet of each motor.

Note that this estimation approach provides a reasonable starting point for configuring the damping and friction parameters, but it is critical to recognize its inherent approximation. Values obtained from motor data sheets may not perfectly match the robot's dynamics,

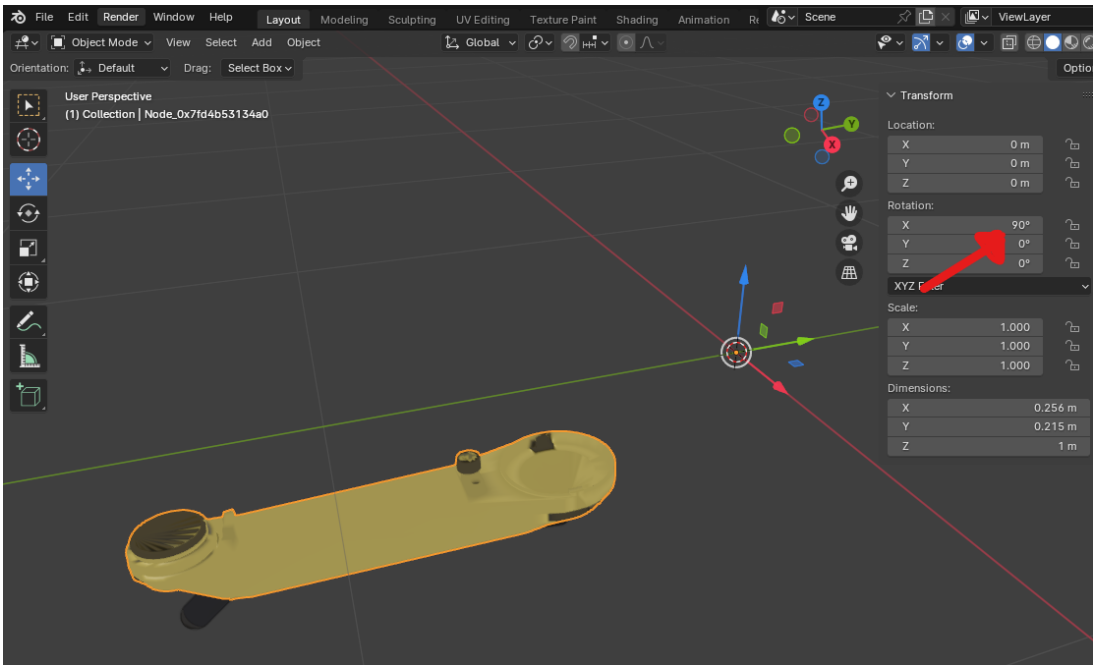


Figure 1.11: Visual meshes as they appear in Blender

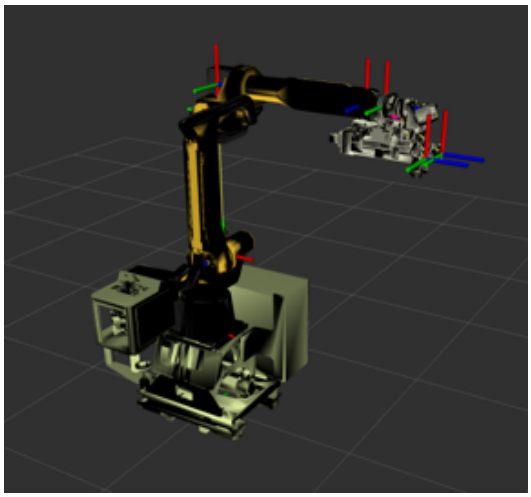
necessitating future refinements based on empirical testing and further optimization.

1.3 Fixing the meshes

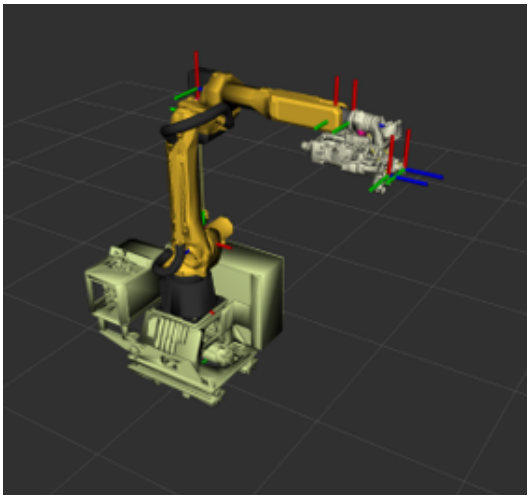
1.3.1 Resolution of mesh rotation issues and shadows in RViz2

To correctly specify the origin of the meshes in the URDF file, we decided to use Blender to precisely determine the position and orientation of each mesh base frame with respect to the corresponding link frame and populate the URDF file accordingly. In doing so, we noticed a discrepancy in orientation between the meshes displayed in Blender and the same meshes displayed in RViz. Fig. 1.11 shows the visual meshes as they appear in Blender. As highlighted by the red arrow, the mesh has a rotation of 90° on the z-axis. However, when visualized in RViz, the mesh doesn't appear to have any rotation as shown in Fig. 1.12a.

We found out that the Collada format in which the visual meshes were provided (`.dae` extension) supports local transforms for the mesh, which are not correctly applied by RViz. In order to further investigate this issue, we decided to manually remove these local transforms from the Collada files and re-export them with Blender. After that we loaded the new meshes in RViz and as we suspected the meshes still had the same orientation as before. Solving this issue indirectly solved another issue related to the visualization of shadows in RViz. As shown in Fig. 1.12, after removing the rotations from the meshes, the shadows are now correctly visualized in RViz. Our hypothesis is that RViz is somehow considering the local transforms of the meshes when calculating the shadows but not when placing the meshes in the scene.



(a) Provided meshes



(b) Fixed meshes

Figure 1.12: Meshes in RViz before and after removing the rotations

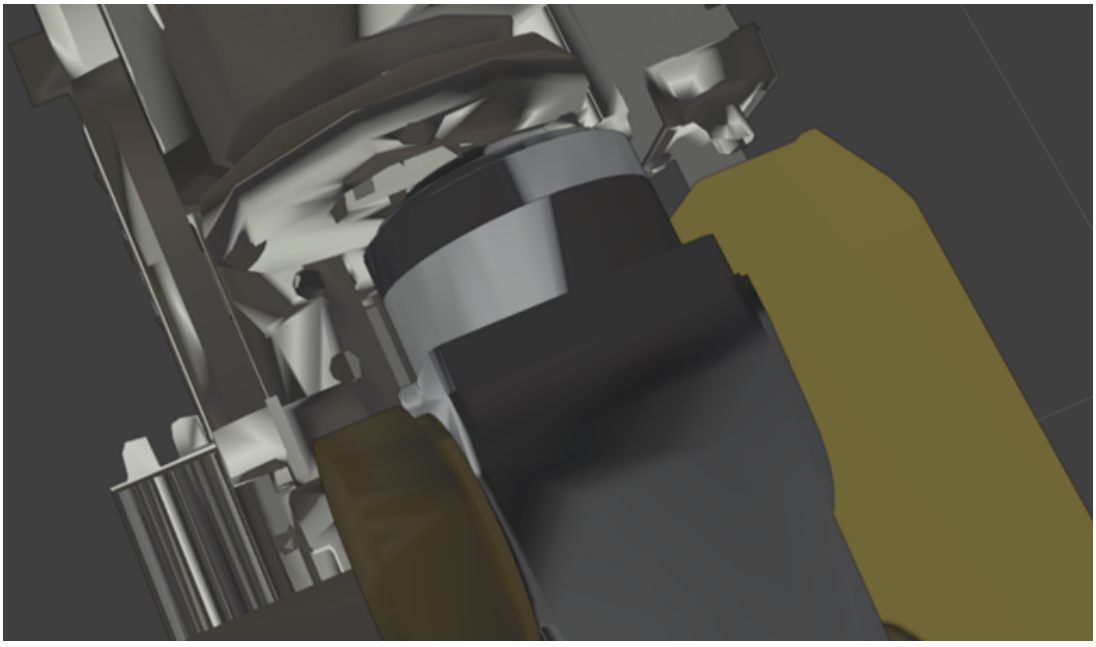


Figure 1.13: Zoom on the end effector after a 90° rotation along the z axis

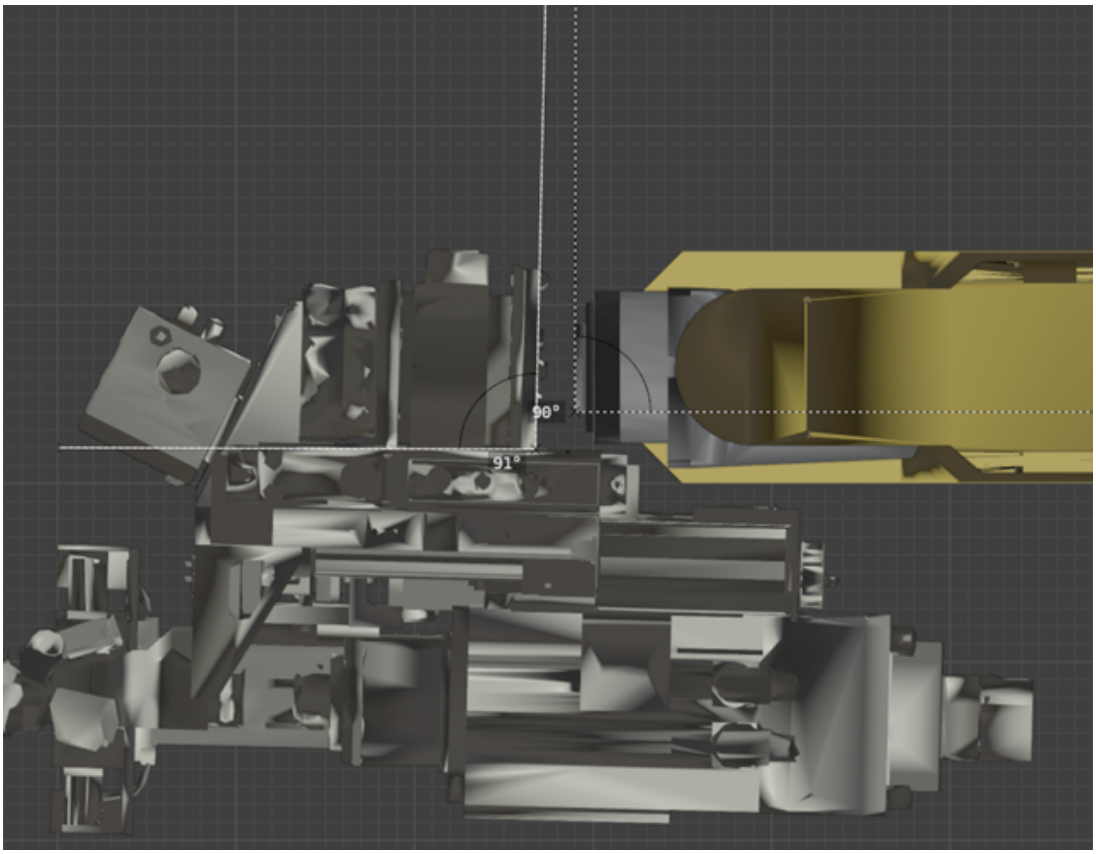
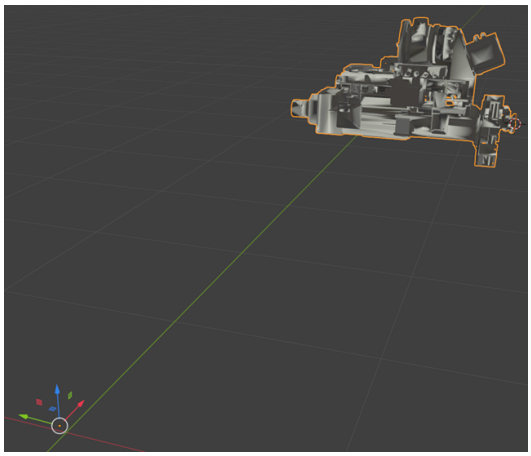


Figure 1.14: End effector misalignment

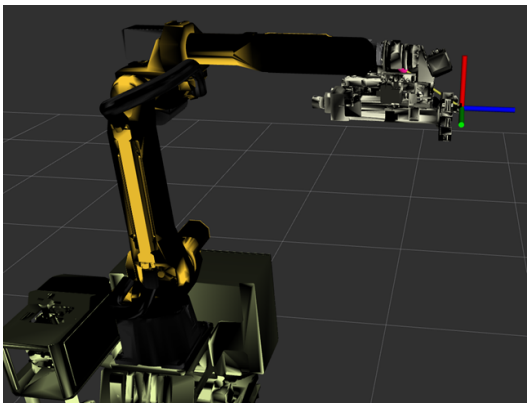
1.3.2 End effector anomalous rotation issue

As said before we used Blender to calculate the transform between the link frame and the mesh reference frame. For most of the meshes, the translation part of this transform could be calculated by using the measurements in the datasheet in [8] while the rotational part of this transform is usually a composition of 90° rotations along the main axes. However, we found out that the end effector is an exception to the last statement: by applying a +90° rotation on the z axis, we noticed that the end effector was misaligned with respect to the robot flange as shown in Fig. 1.13. Initially we thought that the misalignment was due to a translational issue, but after inspecting the mesh more closely, we found out that the flange is not parallel to the contact surface of the end effector, as seen in Fig. 1.14.
At this point we aligned the two surfaces by hand in Blender, determining an error of 1.5° on the x axis and -1.6° on the y axis (fixed XYZ frame rotations on the mesh frame). It is worth pointing out that although the angles are apparently small, because the origin of the mesh is far away, they result in a significant positional error.
In order to complete the URDF, we needed to calculate the RPY rotation to express the mesh frame m seen in Fig. 1.15a, with respect to the end effector link frame f, seen in Fig.1.15b. In order to do this, we exploited the fact that the Blender origin coincides with the RViz origin (frame 0). By using the composition of current frame rotations, the rotation matrix that we need can be expressed as

$$R_m^f = R_0^f R_m^0$$



(a) Mesh frame m



(b) End effector frame f

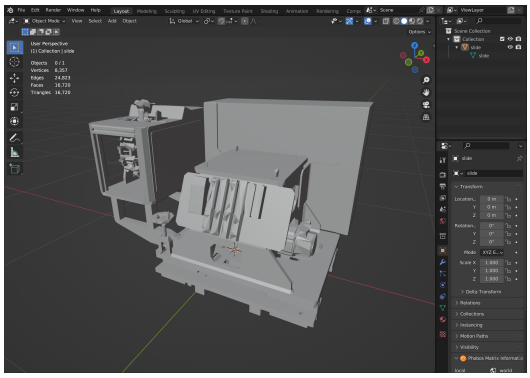
Figure 1.15: Mesh frame and end effector frame

The matrix R_m^0 was calculated from the Euler angles determined above (1.5° on the x axis and -1.6° on the y axis) while the matrix R_0^f was determined by expressing the versors of the frame 0 with respect to those of the frame f. Finally, we had to convert the rotational matrix in RPY angles. For more details, we leave the Matlab script that performs these calculations in Listing 1.1.

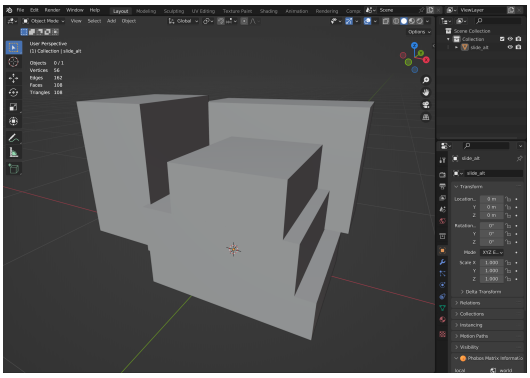
Listing 1.1: Matlab code to calculate the required rotation of the end effector

```
R_0_f=[0 0 1; 0 -1 0; 1 0 0];
gamma = deg2rad(1.6);
beta = deg2rad(1.5);
alpha = deg2rad(90);
Rx=[1 0 0; 0 cos(gamma) -sin(gamma); 0 sin(gamma) cos(gamma)];
Ry=[cos(beta) 0 sin(beta); 0 1 0; -sin(beta) 0 cos(beta)];
Rz=[cos(alpha) -sin(alpha) 0; sin(alpha) cos(alpha) 0; 0 0 1];
R_0_m= Rx*Ry*Rz;
R_f_0 = R_0_f.';
R_f_m = R_f_0*R_0_m;
phi = atan2(R_f_m(2,1),R_f_m(1,1))
theta = atan2(-R_f_m(3,1),sqrt(R_f_m(3,2)^2+R_f_m(3,3)^2))
psi = atan2(R_f_m(3,2),R_f_m(3,3))
```

1.3.3 Collision meshes simplification

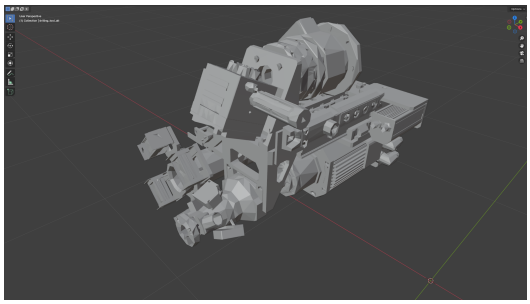


(a) Before

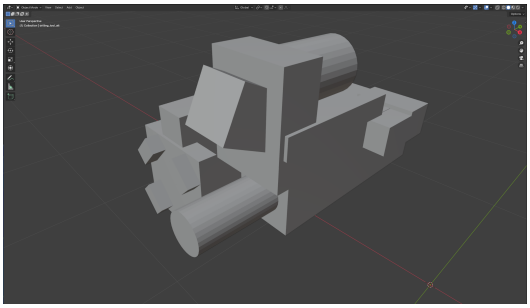


(b) After

Figure 1.16: Slide collision mesh before and after simplification



(a) Before



(b) After

Figure 1.17: End effector collision mesh before and after simplification

While simulating the robot in Ignition Gazebo we found out that the simulation performances were heavily dropping when adding the slide into the simulation. After some investigation we found out that when temporarily disabling collisions performances improved significantly. By checking the collision meshes, we found out that the slide and the end effector had unnecessarily complex collision meshes, at least for our purposes. In order to improve the simulation performances, we decided to simplify these two collision meshes, as seen in Figs. 1.16 and 1.17. Notice how this operation was meticulously perform to guarantee that the new collision meshes are "less permissive" than the previous ones, in the sense that the latter contain the former. Table 1.2 shows the difference in number of edges and faces before and after. We provided the new meshes in the `acg_resources_fanuc_m20ia_35m_description` package. The `resources` folder also contains the blender projects used to create the new collision meshes.

	Vertices	Edges	Faces
Before	8.357	24.823	16.720
After	56	162	108

(a) Slide

	Vertices	Edges	Faces
Before	15.396	47.486	31.818
After	614	11.862	1.249

(b) End effector

Table 1.2: Vertices and edges counts before and after collision mesh simplification

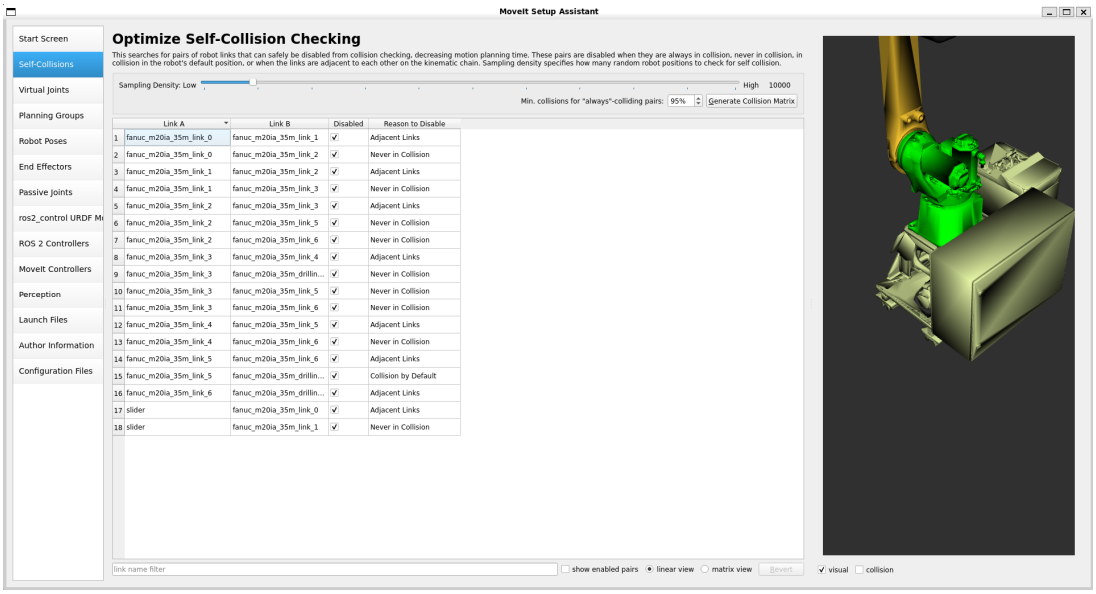


Figure 1.18: MoveIt Setup Assistant

1.4 MoveIt! configuration package

In this section we describe the decision we made for configuring the MoveIt2 motion planning framework for our robot. The `acg_resources_fanuc_m20ia_35m_moveit_config` package was originally generated through "MoveIt Setup Assistant" (Fig. 1.18), but several modifications were manually made after the generation of the package according to our needs.

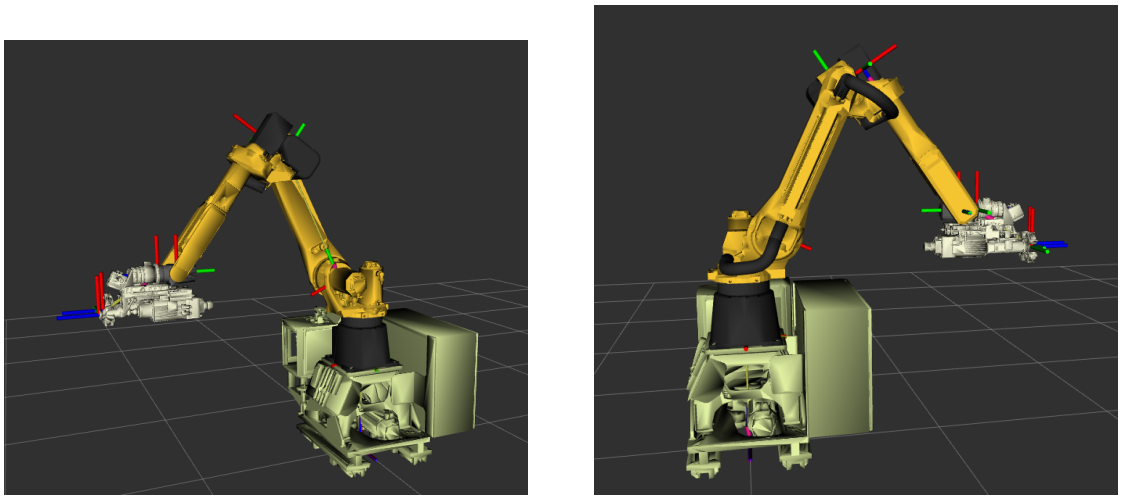
1.4.1 Allowed collisions matrix

First we set up the allowed collisions through the setup assistant. It is important to disable collision checking between adjacent links, otherwise those would be constantly considered in collision. Besides that, the setup assistant has an automatic procedure to identify links that never collide, so these collisions were also disabled. Finally, collision between link 5 and the drilling tool were disabled as they collide by default.

1.4.2 Planning groups

In order to embrace all the use cases that we could think of, we defined four planning groups:

- `fanuc_m20ia_35m_slide_to_drilling_tool`: this can be used to plan movements for the drilling tool, by also using the degree of freedom given by the slide;
- `fanuc_m20ia_35m_slide_to_inspection_tool`: this can be used to plan movements for the inspection tool, by also using the degree of freedom given by the slide;
- `fanuc_m20ia_35m_base_to_drilling_tool`: this can be used to plan movements for the drilling tool, without moving the slide;
- `fanuc_m20ia_35m_base_to_inspection_tool`: this can be used to plan movements for the inspection tool, without moving the slide;



(a) Home (b) Home inverse

Figure 1.19: Named targets we defined for the planning groups

For each of the planning groups we set the `KDL` kinematic solver with a kinematic resolution of 0.005. For further information on the kinematic solvers see section 1.4.6. The chosen planner for each of the planning groups is `OMPL` with the `BiTRRT` algorithm. This decision was made after testing the performances of the available planners and finding out that this combination of solver and planner was the most reliable and fast for our robot. For further information on the planning pipelines see section 1.4.7.

1.4.3 Named poses

For each of the planning groups we defined the correct link as the end effector, and we defined two named poses for all of them: `home`, which is shown in Fig. 1.19a and `home_inverse`, which is shown in Fig. 1.19b. We chose those poses as starting from these it is easy to reach the entirety of the robot workspace. Furthermore, we choose poses with a (relatively) high manipulability, which can be seen as a measure of distance from the singularities calculated as

$$w(\mathbf{q}) = \sqrt{\det(\mathbf{J}(\mathbf{q})\mathbf{J}^T(\mathbf{q}))}$$

where $\mathbf{J}(\mathbf{q})$ is the geometric Jacobian matrix of the robot at configuration $\mathbf{q}$. The `home` configuration has a calculated manipulability of 1.0854 and for the `home_inverse` configuration the manipulability is equal to 1.0848.

1.4.4 ROS2 controllers and MoveIt interface

Through the setup assistant we also configured the `ros2_control` controllers needed to execute the trajectories planned by MoveIt on a mock hardware. We chose to control the robot with a single `JointTrajectoryController` that can be used to control all the robot joints at once. The controller is configured to send `position` commands to the hardware as the mock hardware is not simulated and it doesn't support `velocity` or `effort` commands. Because we defined planning groups that do not contain every joint of the robot, we had to manually modify the `ros2_controllers.yaml` file to enable the `allow_partial_joints_goal` parameter for the

`JointTrajectoryController`. This is needed to allow MoveIt to execute trajectories for the planning groups that do not contain all the robot joints with the same controller.

We also defined a `joint_state_broadcaster` controller that is used to publish the joint states of the robot to the `/joint_states` topic. This is needed to calculate the direct kinematics of the robot (which is managed by `robot_state_publisher`) and visualize the robot in RViz as well as to plan trajectories with MoveIt.

Finally, in order to interface MoveIt with the said controller, we had to include a `MoveItSimpleControllerManager` in the package configuration. This is an interface configured to send action goals to the `JointTrajectoryController` we configured before and to receive feedback from it. This is needed to execute the trajectories planned by MoveIt on the mock hardware.

1.4.5 Editing the URDF xacro file

After generating the MoveIt configuration package with the setup assistant, we manually edited the `.urdf.xacro` file of the package in order to make it possible to specify the initial state of the robot in the configuration files `intial_positions.yaml`, `intial_velocities.yaml` and `intial_efforts.yaml`. We also added some arguments to choose which hardware and which command interface to use for the robot joints. This was needed to allow for dynamic simulations of our robot in Ignition Gazebo. For more information about the modifications we made to the URDF xacro file, please refer to section 3.3.

1.4.6 Kinematic solver configuration

After generating the MoveIt configuration package with the setup assistant, we manually edited the `kinematics.yaml` of the package in order to set up the kinematic solver configuration.

Kinematic solvers in MoveIt2 The most important purpose of the kinematic solver is to solve the position inverse kinematics problem (i.e. finding a joint configuration that corresponds to a given end effector pose). In MoveIt2, a kinematic solver is used to solve the inverse kinematics problem for each of the planning groups. The kinematic solvers for each planning group are configured in `kinematics.yaml`.
Several approaches can be used to solve the inverse kinematics problem. Figure 1.20 provides an overview of the most common approaches. These approaches can be divided in analytical and numerical approaches. Among the numerical approaches, the two most common approaches are gradient-based and Jacobian-based algorithms.

- Analytical algorithms are based on the analytical solution of the inverse kinematics problem. These algorithms are usually very fast, but they are not always available for all robots and they are not always reliable. In principle, they should only work for non-redundant robots, but in practice some techniques were developed to extend them to redundant robots. An advantage of analytical algorithms is that they can enumerate all the solutions to the inverse kinematics problem. The problem is that one should be able to choose the best solution among the ones found;

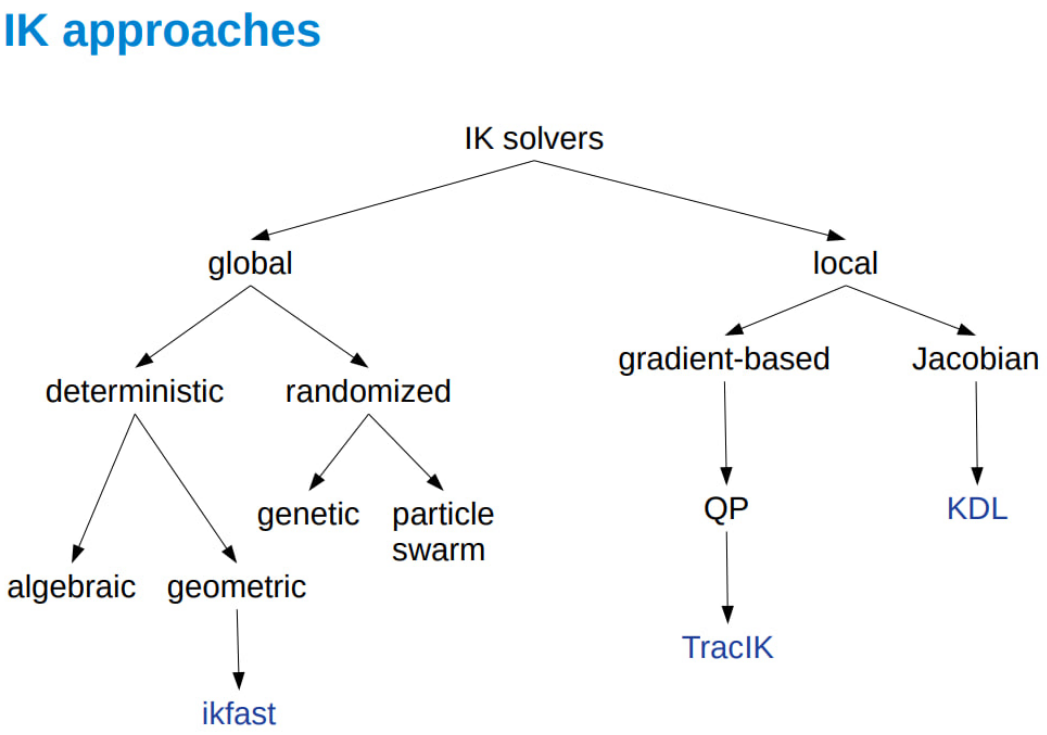


Figure 1.20: Inverse Kinematics approaches

- Gradient based algorithms are based on the minimization of a cost function that measures the distance between the current end effector pose and the desired end effector pose (positional inverse kinematics). These algorithms have the advantage of being able to handle redundant robots and they allow to specify additional constraints on the joint configuration. However, being a numerical method, they are usually slower than analytical algorithms;
- Jacobian-based algorithms are based on first-order inverse kinematics. These algorithms are usually faster than gradient-based algorithms. They are also able to handle redundant robots. In theory, one could exploit null space movements to bias the solution towards a specific joint configuration.

MoveIt2 provides the following kinematic solvers by default:

- `KDL`: this is a solver provided by the Orocos KDL library. It is based on a first order inverse kinematic algorithm that uses the geometric jacobian pseudo-inverse [15];
- `IKFast`: this is a solver that uses a precomputed analytical solution to inverse kinematic [16]. Because the analytical solution is robot-specific, this solver is not provided by default and it must be generated for each robot. This solver can only be usedd for robots with 7 or less degrees of freedom. In order to solve the problem when the robot is redundant, `IKFast` uses one of the joints as a free parameter and it returns a solution parametrized by the value of the free parameter. This means that the user must be able to choose the best solution among the ones found;

- `cached_ik`: this is not a real solver but it can be used to cache the solutions to the inverse kinematics problem. This can be useful to speed up the planning process when the same inverse kinematics problem is solved multiple times;
- `srv`: this is not a solver, but it can be used to call an external service to solve the inverse kinematics problem.

Another solver that is not provided by default but that can be easily configured for MoveIt2 is the `TracIK` solver [17] [18]. This solver runs a pseudoinverse Jacobian IK method (jacobian-based) and a nonlinear optimization IK method concurrently (gradient-based) [15]. This solver is based on a gradient-based algorithm and it is able to handle redundant robots. `TracIK` allows to specify an optimization objective among:

- `Speed`: the solver returns the first solution found by either the pseudoinverse Jacobian IK method or the nonlinear optimization IK method;
- `Distance`: the solver runs for the full timeout and returns the solution that minimizes the distance between the current joint configuration and the desired joint configuration;
- `Manupolation1`: the solver runs for the full timeout and returns the solution that maximizes the manipulability of the robot;
- `Manupolation2`: the solver runs for the full timeout and returns the solution that minimizes the ratio of min to max singular values of the Jacobian.

This solver is able to handle joint constraints better than `KDL`, expecially on robots with a low number of DOFs [15]. This solver is not provided by default and it must be generated for each robot.

Where is the kinematic solver used The kinematic solver can influence both the `plan_service` and the `cartesian_path` capabilities of `move_group` node.
The `plan_service` capability is used to plan a trajectory from a start state to a goal state (point to point movement) and it internally uses one of the loaded planning pipelines to solve the problem. More precisely, the planning problem can be solved both in task and joint spaces: only when it is solved in task space the kinematic solver is used. `MoveGroupInterface`'s `plan` API internally uses the `plan_service` capability.
The `cartesian_path` capability is used to plan a cartesian movement through a sequence of waypoints. `move_group` internally uses a class named `CartesianInterpolator` to interpolate the waypoints and then it uses the provided kinematic solver to invert each of the interpolated waypoints and find the joint configurations that correspond to them. `MoveGroupInterface`'s `computeCartesianPath` API internally uses the `cartesian_path` capability.

Provided kinematic solver The planning module described in Chapter 2 uses `MoveGroupInterface`'s `computeCartesianPath` API to plan a cartesian movement through a sequence of waypoints. Using an analytical solver, one should choose which of the returned solutions is the closest one to the current configuration of the robot. For this reason, we chose to use a numerical kinematic solver as this allows to start the search from the current joint

configuration to minimize the joint space distance between the current joint configuration and the found solution.
We also tried to use the `TracIK` solver, but we found out that even when using the `Speed` optimization objective, the solver was slower than the `KDL` solver. This was not noticeable when planning a trajectory but it was when running the task space controller presented in chapter 4. For this reason, we ultimately chose to use the `KDL` solver with a kinematic resolution of 0.005. This resolution was chosen as it is the default resolution used by the `KDL` solver and it is the resolution that provides the best trade-off between speed and accuracy for our robot.

1.4.7 Planning pipelines configuration

After generating the MoveIt configuration package with the setup assistant, we manually edited the configuration files of the package in order to set up the planning pipelines configuration.

Planning pipelines, planning plugins and planner IDs Even if it is not documented online, we found out that the default MoveIt launch files look for all the configuration files in the `config` folder that have the pattern `*_planning.yaml`. Then it creates a planning pipeline for each of the configuration files found. A planning pipeline is a composition of a planning plugin (which is a `pluginlib` plugin that provides planning capabilities), the configuration for the said planning plugin and several request and response adapters which can be seen as functions that are called on the planning request and on the planning result respectively. The most important parameter in each of the configuration files is the `planning_plugin` parameter, which defines the planning plugin that will be used by that pipeline.
Some planning plugins provide multiple planners. For example, Pilz provides the `PTP`, `CIRC` and `LIN` planners. These planners are identified by a unique ID, which is used to select the planner to be used in the planning request. The ID of the planner to be used in the planning request can be set in the planning request that is sent to `move_group`. The `MoveGroupInterface` API also provides a method to set the planner ID to be used in the planning request (as well as one for setting the planning group to be used in the planning request).

Provided planning pipelines We provided the following planning pipelines:

- `ompl`: this pipeline uses the `ompl_interface/OMPLPlanner` planning plugin. This plugin provides a wrapper around the `ompl` library, which is a popular motion planning library. The `ompl` library can be configured to use different optimization algorithms. We chose to use the `BTRRT` algorithm, which is a bidirectional tree-based planner. We found out that this planner is the most reliable and fast for our robot. In the provided package, the `ompl_planning.yaml` file the configuration we chose for `BTRRT`. For more information about the choice of the `BTRRT` algorithm, please refer to section 2.2;
- `chomp`: this pipeline uses the `chomp_interface/CHOMPPlanner` planning plugin;
- `pliz_industrial_motion_planner`: this pipeline uses the `pilz_industrial_motion_planner/CommandPlanner` planning plugin. The default

planner ID we set for this pipeline is `PTP`. This pipeline is provided as, differently from most of OMPL planners, it is a deterministic planner. This means that it is possible to plan a trajectory and then execute it multiple times and always get the same result.

For all of the planning pipelines we provided the same request and response adapters that can be found in the correspective configuration files. By default, the planning module we developed uses the `ompl` pipeline. For more details about this choice, please refer to section 2.2.

1.4.8 Results

The outcome of this work package is a ROS2 package that contains the MoveIt configuration for the Fanuc M20iA/35M robot and allows to plan and execute trajectories for the robot through RViz's motion planning pluigin. For the usage instructions, please read the README of said package.

CHAPTER 2

WP2 - PLANNING SYSTEM

This chapter is dedicated to the development of the task space path generation module (Section 2.1) as well as the development of the planning module (Section 2.2).

2.1 Task space path generation module

2.1.1 Module Overview

The task space path generator module we developed contains `ouroboros`, a robot-agnostic tool for generating and visualizing task space paths and exporting them to a bag file. The module is composed of three ROS2 nodes:

- `ouroboros`: it is the main node of the module, responsible for the generation and visualization of task space paths in RViz;
- `ouroboros_gui`: it is the graphical user interface (GUI) node, which allows users to interact with the `ouroboros` node. Spawning this node will also spawn a managed instance of `ouroboros` node;
- `path_viewer`, a simple node used to visualize a path already generated by `ouroboros` in RViz.

The package also contains three launch files:

- `path_generator_rviz`: allows to generate a task space trajectory for a robot and visualize it together with the robot model in RViz. The node `joint_state_publisher` is also spawned, so the user can control the robot joints through the GUI while visualizing the generated path.
- `path_generator_moveit`: allows to generate a task space trajectory for a robot and visualize it together with the robot model in RViz. The MoveIt2 planning system is also spawned together with the MoveIt RViz plugin, so the user can plan ad execute trajectories with MoveIt while visualizing the generated path.

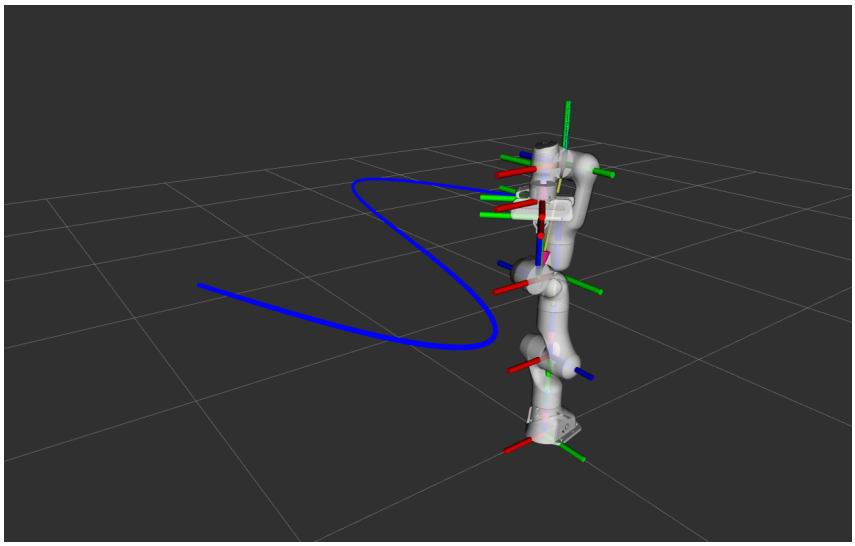


Figure 2.1: Path visualization in RViz

- `rviz`: launches RViz with the correct configuration to visualize the generated path as well as the robot model.

For more information about how to use these launch files, please refer to the README file of the package.

2.1.2 Usage

When launching one of the provided launch file, RViz will open showing both the robot model and the generated path as shown in Fig. 2.1. At the same time, the Ouroboros GUI shown in Fig. 2.2 will open alongside RViz. The user can interact with the GUI to change the path parameters and visualize the changes in real time. It allows to dynamically select the type of path to be generated and to change the parameters of the selected path. The supported paths so far are:

- `Sine Wave`: a sine wave path;
- `Sine Wave (one period)`: a sine wave path with only one period;
- Lines: as a degenerate case of the sine wave, it can generate a straight line.

Each path has dynamic parameters that can be changed at runtime through the GUI.

The GUI also allows to change the base frame used for drawing the path in RViz2. This is internally named `draw frame` and is different from the base frame used for exporting the path to a bag file, which is internally named `export frame`. Usually the export frame will be the robot base frame, while the draw frame will be the end effector frame. However, the draw frame can be changed at runtime according to the user needs. The user can control the robot joints by means of external methods, and the generated path will move as the chosen draw frame moves.

The planar paths are generated in the XY plane of a frame called `cartesian_path_frame`. The orientation of the exported path is always perpendicular to this plane. The `cartesian_path_frame` frame is published on the `/tf` topic and can be visualized in RViz2 in

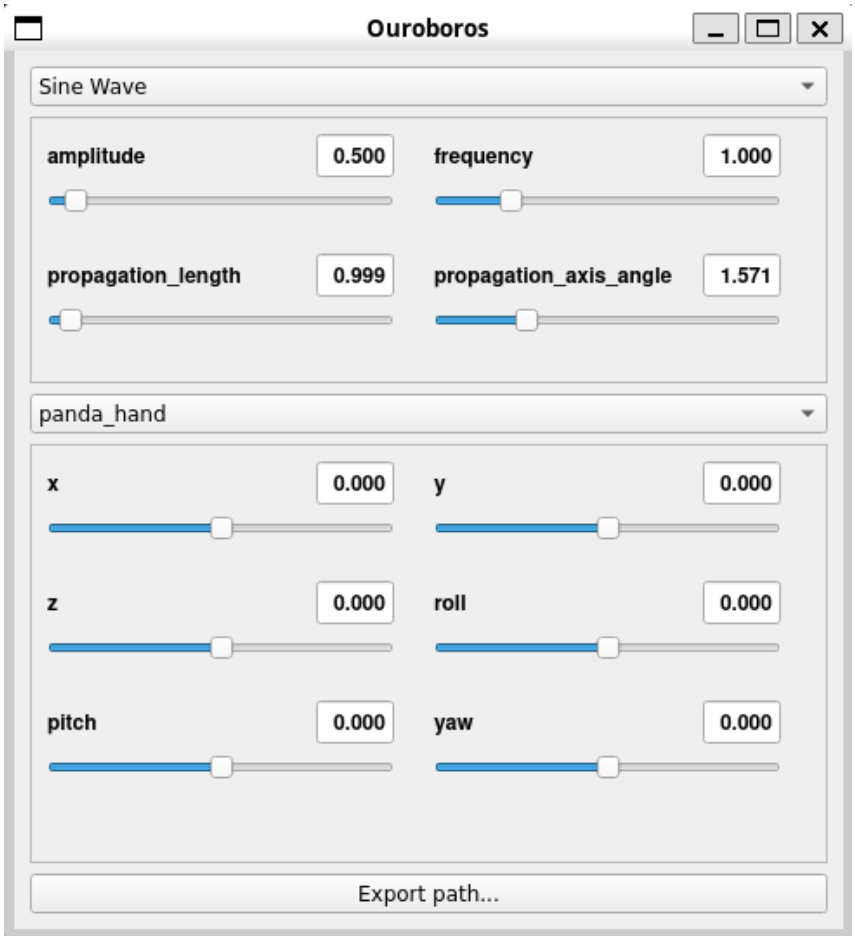


Figure 2.2: Ouroboros GUI

real time. The transform from the `draw frame` to the `cartesian_path_frame` can be changed at runtime through the GUI, which allows to specify both the position and the orientation of the `cartesian_path_frame` w.r.t the `draw frame`. However, it is not recommended to change these parameters too much because it could lead to unfeasible paths. Notice that the path will always be exported w.r.t the `export_base_frame` parameter of the node.

When the user exports the path to a bag file, the path is saved in the `/cartesian_path` topic and the path is saved as a `geometry_msgs/msg/PoseArray` message. The base frame of the path is the `export_base_frame` parameter of the node. When exporting the path, `ouroboros` dynamically calculates the transform between `draw_base_frame` and `export_base_frame` to get the correct coordinates for the path. The number of samples of the path is determined by the `n_samples_export` parameter of the node. The exported path can be used for planning a trajectory with our planning module (Section 2.2).
The recommended way of generating a path is the following:

1. launch either the `path_generator_rviz` or the `path_generator_moveit` launch file as described in the README file of the package;
2. select the type of path to be generated from the GUI;
3. change the draw frame as the end effector frame of the robot (or any other frame you want to use);
4. move the robot to a feasible configuration which will be the starting point of the path (either with `joint_state_publisher` or with MoveIt2);
5. change the path parameters through the GUI until you are satisfied with the generated path;
6. export the path to a bag file by pressing the `Export path` button.

The user is encouraged to use MoveIt2 to plan and execute a trajectory towards the desiderd starting point of the path and only then choose the appropriate parameters and export it. This will ensure that the path is feasible and that the robot will not collide with the environment while moving to the starting point. However, a collision-free path is not guaranteed by the path generation algorithm. Furthermore, it is not guaranteed that all the points of the path are within the robot workspace. For these reasons, after exporting the path, the user should check whether the path is feasible by planning a trajectory with our planning module (Section 2.2).

2.1.3 Configuring a new robot

The launchfile are also designed to be easily extensible to support a new robot. In order to configure a new robot, the user has to add a set of configuration files to the `config` folder of the package. The needed files are:

- `<robot_name>.yaml`: contains the ROS2 parameters for the robot model `<robot_name>` as well as the needed parameters for the launch files;
- `<robot_name>.rviz`: contains the RViz configuration for the robot model `<robot_name>` without the MoveIt2 plugin;

- `<robot_name>_moveit.rviz`: contains the RViz configuration for the robot model `<robot_name>` with the MoveIt2 plugin.

2.1.4 Design choices

The key aspects that were considered while developing this module are robot-agnosticism, modularity, extensibility and ease of use for the user. Some design choices made during the development of the module are:

- The nodes were implemented in Python, leveraging key ROS2 libraries such as `rclpy`, `rosbag2_py`, and `tf2_ros`. The Python language was chosen to simplify the development of the GUI;
- The GUI was implemented with PyQT, as in the case of the `joint_state_publisher_gui` node;
- Path visualization was implemented on top of RViz, which is a widely used tool in the ROS2 community. This choice was made to ensure compatibility with a broad range of robotic applications;
- The design of `ouroboros` is modular, with a focus on extensibility. This makes it simple to integrate of additional path generation algorithms by implementing the abstract class `PathGenerator`.
- `ouroboros` was designed to be independent from its GUI, however at this stage the recommended way to use it is through the GUI. This design choice was inspired by the architecture of ROS2's `joint_state_publisher` node.

2.1.5 Comunication with other nodes

The `ouroboros` node communicates with other nodes through the following topics:

- `/tf` (publisher): the `cartesian_path_frame` frame is published on this topic. This is the frame in which the planar paths are generated. The frame publishing rate is specified by the `tf_publish_rate` parameter of the node;
- `/tf` (subscriber): the `/tf` topic is read to discover all the available frames. The GUI allows the user to change the draw frame between any of these frames, except for the `cartesian_path_frame` frame. This topic is also used to calculate the transform between the `draw_base_frame` and the `export_base_frame` frames when exporting the path to a bag file;
- `/visualization_marker`: the generated path is published on this topic as a `visualization_msgs/msg/Marker` message. This allows to visualize the path in RViz trough the `Marker` plugin. The name of the topic on which the path is published is specified by the `marker_topic` parameter of the node, while the publising rate is specified by the `path_publish_rate` parameter of the node. The number of published samples of the path is determined by the `n_samples_draw` parameter of the node.

The `ouroboros` node also communicates indirectly with our planning module by exporting the generated path to a bag file. The path is saved in the `/cartesian_path` topic and the path is saved as a `geometry_msgs/msg/PoseArray` message. The base frame of the path is the `export_base_frame` parameter of the node. The number of samples of the path is determined by the `n_samples_export` parameter of the node.
The `path_viewer` node publishes the path to be visualized on the `/visualization_marker` topic. The name of the topic on which the path is published is specified by the `marker_topic` parameter of the node. The publishing rate of the path is specified by the `rate` parameter of the node.

2.1.6 Limitations and future improvements

As we already said at the moment Ouroboros does not guarantee that the generated path is feasible. The path could be unfeasible because it leads to self-collisions or because it goes outside the robot workspace. One way we tried to overcome these issues is to integrate Ouroboros with the MoveIt planning demo. Using the provided launch file, one could ensure that the first point of the trajectory is reachable without collisions. However, it does not guarantee that the other points of the path are feasible.
To improve the situation, a relatively easy solution colud be to add a configuration parameter to specify the robot workspace and to check whether the generated path is inside this workspace before exporting it. Notice that while it is easy to implement this check for our robot, it is more difficult to implement it while keeping the module robot-agnostic. The resolution of the self-collision problem is more difficult and it is left as future work.

2.2 Planning module

2.2.1 Module overview

The module is a robot-agnostic planning system which implements the planning strategy to turn a task space path, defined by the user, into a joint space or task space trajectory, obeying the robot's position, velocity and possibly acceleration limits, together with the necessary components to generate references at the controller frequency. It is possible to give as input any type of trajectory regardless of how it is generated. The only constraint is that it must be saved in a bag file in the form of a `geometry_msgs/msg/PoseArray`. Multiple path executions are possible without recompiling the code. Furthermore, the module is able to record the executed trajectories and store them in a bag file whose name is chosen by the user. This allows for subsequent analyses to be performed. Obviously, regarding trajectories' execution, the module is provided with an interface toward **JointTrajectoryController** as well as **TaskSpaceTrajectoryController** which are the supported controllers' types.

The planner is provided in the package `fanuc_m20ia_35m_planning_demo` which contains two source files:

- **fanuc_m20ia_35m_planning_demo**: it contains the business logic of the planner, i.e. the execution flow of the key steps associated with it. It also manages the visualization,

in RViz, of both the end-effector's planned trajectory and planned joint space trajectory as a sequence of robot configurations, together with the original desired path;

- **planning_node**: it is the main node of the module whose implements some planning utility methods, manages the interface toward the controllers and also the trajectories' recording and storage.

The package also contains a launch file:

- **plan_and_execute**: manages the configuration and launch of the necessary ROS2 node to execute path planning demo. The launch file accepts arguments specifying the name of the configuration file and the path to the bag file containing the path to be planned and executed. Utilizing these arguments, the launch file generates the required MoveIt!2 configuration for the specified robot and starts the corresponding ROS2 node for the path planning demo. The launch file also handles the initialization of the ROS2 parameters required for the planning node to work properly.

For more information about how to use this launch file, please refer to the README file of the package. In this section, we focus on joint space planning, as planning and generating references for the task space controller will be adequately detailed and justified in Chapter 4.

2.2.2 Features and Key Steps

The planning module has been meticulously designed with a primary objective of optimizing its versatility and impartiality towards specific robotic systems, thereby enhancing its capacity for extensibility, adaptability, and recurrent utilization.

Node Parameters The adaptability mentioned above is realized through the strategic definition of a set of customizable parameters whose values have to thoughtfully provided by the user.

Below, a concise delineation of the node parameters is presented, deemed functional for comprehending the ensuing functionalities:

- `input bag file topic`: the name of the topic under which the desired trajectory is saved in the bag file;
- `planning group`: the name of the planning group used for the planning process;
- `recorded trajectory home pose topic`: the name of the topic under which the trajectory executed to reach the home pose is recorded in a bag file (it is noted that the home pose is dynamically chosen based on the path's location in the space);
- `recorded trajectory point2point topic`: the name of the topic under which the trajectory executed to reach the first point of the desired path is recorded in a bag file;
- `recorded trajectory cartesian topic`: the name of the topic under which the planned trajectory corresponding to the user-provided Cartesian path is recorded in a bag file;

- `feedback sampling period`: allows adjustment of the period with which feedback messages sent from the controller's action server to the planning node's action client are recorded (e.g., setting it to 10 saves a message every 10);
- `max velocity scaling factor`: sets a scaling factor for optionally reducing the maximum joint velocity. Allowed values are in (0,1]. The maximum joint velocity specified in the robot model is multiplied by the factor. If outside the valid range (which includes being set to 0.0), the factor is defaulted to 1.0 (i.e., maximum joint velocity);
- `recorded trajectory file name`: the name of the bag file in which the executed trajectories are saved;
- `goal position tolerance`: sets the position tolerance used for reaching the goal when moving to a pose;
- `goal orientation tolerance`: sets the orientation tolerance used for reaching the goal when moving to a pose;
- `goal joint tolerance`: sets the position and orientation tolerances used for reaching the goal when moving to a pose;
- `controller name`: the name of the controller;
- `controller type`: the type of controller, which can be either TaskSpaceTrajectoryController or JointTrajectoryController;
- `base frame`: the name of the base frame;
- `named_targets`: It is a structured parameter that contains within it:
  - `home_pose`;
  - `home_pose_inverse`.

  which are the names of any home poses defined in the SRDF that respectively represent the starting pose of the robot and its "inverse," that is, a mirrored pose compared to the previous one, which can prove useful in case the trajectory to follow is positioned behind the robot.

The parameters of the node are fundamental to ensure that the planning module is general and reusable, thus making its operation independent of the robot and adaptable to the user's needs. However, it is important to note that the last parameter may not be entirely robot agnostic in the sense that while the user can define the poses mentioned in the SRDF and configure the parameter accordingly, great care must be taken regarding the check performed on the trajectory's position relative to the base frame: if the user's base frame is not faithful to that defined in the WP1 under consideration, in order to avoid the robot making a wrong choice regarding the home pose, it would be advisable not to configure this parameter. Further details and improvements are provided in the section 2.2.8.

Planning Processes Let's explore the main functionalities of the planning module, providing an overview of the different planning processes carried out and demonstrating their usefulness in optimizing the robot's performance in various operational scenarios.

Initially, the desired path stored in the bag file, whose path is provided by the user from the command line, is read. If the file is located within the module's workspace, the absolute path of the file is automatically reconstructed. After retrieving the path and verifying that it contains at least one point, automatic planning towards a "home" pose is performed. This pose is dynamically defined based on the path's position relative to the base frame (WP1), aiming to facilitate subsequent planning and encourage the robot to approach the target path location as closely as possible in space. However, this step of planning towards the home pose is strictly dependent on the definition of the `named_targets` parameter: if this is not defined by the user, this step is skipped, and planning proceeds directly towards the first point of the trajectory, which will be addressed shortly. It is noteworthy that the status of all planning processes is displayed both through the Command Line Interface (CLI) and RViz.

If a planning process is successful, the planned trajectory of the end-effector and the joint trajectory, presented as a set of robot configurations, are displayed. This occurs not only for the initial planning but for all subsequent planning processes as well. At this stage, the user is required to authorize the execution of the trajectory, and similar to the planning phase, the progress status is visible through both CLI and RViz. This occurs not only for the initial execution but for all subsequent execution as well. During execution, the trajectory is recorded in a bag file: the name of the file and the topic on which it is saved are chosen by the user through the respective parameters `recorded_trajectory_file_name` and `recorded_trajectory_home_pose_topic`.

Only after the successful completion of this operation does the module proceed to the second planning phase to position the robot at the first point of the path. Before the planning is executed, the starting point (current pose of the end-effector indicated by the red sphere) and the target point (highlighted by the green sphere) can be observed in RViz. The user can then authorize this step, which involves planning and displaying the result as explained earlier. Additionally, a validation step is included, as MoveIt! sometimes returns solutions labeled as `APPROXIMATED` by the OMPL planner, even though they are considered correct. This is a known issue fixed in the OMPL 1.6.0 release in January 2023 but may not yet be integrated into MoveIt! as we encountered related issues with planned trajectories. The following GitHub issues were referred to regarding the issue just described: [19] and [20]. To address this, a sanity check is performed to ensure that the distance between the final point of the planned trajectory and the target point is below a certain threshold defined by the `goal_position_tolerance` parameter. This is a crucial step, as inaccuracies in the outcome of this planning phase could significantly impact the subsequent one, which concerns the actual path desired by the user. Also, this execution must be authorized and is going to be recorded, but this time the topic on which it is saved depends on `recorded trajectory point2point topic` parameter.

The final planning involves the Cartesian path, which initially populates the waypoints with poses read from the input bag file and potentially sets orientation constraints according to specifications. Subsequently, planning is executed. In order to allow for analysis by the human operator, the planned end-effector's trajectory together with the desired path are displayed

in RViz. Upon verifying the successful outcome of this step, the trajectory can finally be executed, recorded on the topic defined by `recorded trajectory cartesian topic` parameter and concluded.

It is important to note that all planning processes are performed using the planning group defined by the user in the node parameters. For further details on the defined planning groups, see Chapter 1. Additionally, once the trajectory is planned, the module is capable of generating references for controllers in both configuration space and task space. However, detailed explanations for the latter are provided in Chapter 4. This choice is made by setting the parameters related to the controller type and name (`controller_type`, `controller_name`).

2.2.3 Usage

This package serves as a tool for executing planning operations. The functionality of the node is adjustable through parameters, providing flexibility to cater to various planning needs. While the package comes with pre-configured settings, it also offers the possibility for users to create their own configurations. This can be achieved by generating files that follow the structure of those located in the config directory.

The configurations provided within this package facilitate planning operations through several methods. These supports the utilization of joint trajectory controller and task space controller. Additionally, users have the option to create new configurations by following the examples of the proposed ones. The launch file in question includes a `config:=` argument, which allows for switching between different configurations. Another argument, `bagfile_path:=`, is used to specify the path to the bag file containing the trajectory to be planned.

In general, the launch file of the `plan_and_execute` package launches the planning node with the provided configuration. However, it is important to note that before this step, for the correct operation of the node, the following nodes must have already been spawned:

- `MoveGroup`;
- `ControllerManager`;
- The chosen `controller(s)`;
- `robot_state_publisher`;
- `joint_state_broadcaster`.

RViz must be launched for visualization, and to ensure the proper functioning of the planning module, the following plugins must be loaded:

- `RVizVisualToolsGUI` to step through the demo;
- `MarkerArray` whose topic must be `/rviz_visual_tools` for visualization of trajectories and the status of planning and execution operations.

When the planning node is launched, after checking that everything is correctly configured, the parameter values are first displayed in the CLI so that the user can verify that they correspond to those set by them. The outcome of reading the desired path from the bag file is also logged.

As anticipated in the previous paragraph, planning to the dynamic home pose is executed if the corresponding parameter is defined, and this is the only planning that is automatically executed without user authorization. If the user finds this planning satisfactory, they can give consent for execution via `RVizVisualToolsGUI` by pressing `Next`: until this button is pressed, the node waits and does not proceed with the execution flow. If this execution is authorized, the robot assumes the new pose, and the trajectory is recorded. The end of execution is notified by the CLI, and when this happens, the node waits again for user authorization, but this time to perform planning. To plan and then eventually execute and record subsequent trajectories, the user only needs to use `RVizVisualToolsGUI` by pressing the `Next` button, as already explained, and the operation is similar.

In order to successfully launch the demonstrations provided with this package, the reader is encouraged to refer to the README of the package.

2.2.4 Design choices

The key aspects that were considered while developing this module are robot-agnosticism, modularity, extensibility and ease of use for the user. Some design choices made during the development of the module are:

- Planning operations was implemented by exploiting `move_group_interface` on top of `MoveGroup`, for the following reasons:

  1. **Ease of use**: `move_group` is a library designed to simplify the robot motion planning process. It provides an intuitive interface that allows developers to specify movement goals clearly and simply.
  2. **Integration with MoveIt**: `move_group` is part of the MoveIt framework, which is widely used and well-supported in the robot developer community. This ensures that the planning module is compatible with other MoveIt components and can benefit from its features and future enhancements.
  3. **Abstraction from inverse kinematics**: `move_group` internally handles the inverse kinematics calculations required to translate user-specified movement goals into robot joint commands. This simplifies the development process and allows developers to focus on robot control logic rather than the intricacies of inverse kinematics.
  4. **Flexibility and adaptability**: `move_group` offers a range of configurable options and parameters that allow developers to customize planning behavior according to the specific needs of the robot and application. This flexibility makes the planning module adaptable to a wide range of robotic scenarios and configurations.

- Trajectory executions are not realized through `move_group_interface` but by implementing an action server for communicating directly with the controllers, therefore, not using the `moveit_simple_controller_manager`. This choice was made to record the executed trajectories: initially, the execute method of `move_group_interface` was used, and a `lifecycle_node` was implemented to act as a recorder, intercepting feedback messages published on the corresponding hidden topic. However, it was realized that

this solution required synchronization primitives as we found out that the code contained a race condition: because of this issue, sometimes messages from previous executions were recorded. Synchronization would have made the code heavier, representing a more cumbersome and complex approach, so an alternative solution was chosen, which proved to be more natural: the executed trajectory is recorded in the `feedback_callback`, where a feedback message is practically constructed and then serialized and saved in the bag file on the topic whose name is settable by parameters. Since two types of controllers are supported, the type of the action client and the type of the associated actions are deduced from the parameters `controller_name` and `controller_type`; furthermore the function `execute_trajectory` is defined as a template function, where `TrajectoryType` is a generic data type that can be specified at the time of using the function. This allows the function to be used with different types of trajectories, making it more flexible and reusable. Of course, based on the type of action client, the respective callbacks have also been implemented to send references to the controllers in both joint space and operational space. Once again, for further details on generating references in operational space and thus extending the planning module, the reader is encouraged to read Chapter 4.

- The planning for the `home` pose and the first point of the desired path was carried out to separate the planning of the actual desired path from the movements required for the robot to reach the first point of the trajectory. Therefore, this choice was primarily made to achieve such separation, towards a more modular and clean analysis of the trajectories.
- Reducing the use of closures to prioritize the definition of utility functions and methods for most operations promotes code cohesion and modularity, facilitating maintenance and comprehension, making the code more generic and flexible for potentially extending its functionality in the future.

2.2.5 Planner's choice

The choice of the planner falls under the category of design choices. However, it was deemed appropriate to dedicate a section specifically to this aspect in order to adequately justify the motivations behind the proposed solution with a sufficient level of detail.

OMPL (Open Motion Planning Library) [21] is an open-source motion planning library that primarily implements randomized motion planners. MoveIt integrates directly with OMPL and uses the motion planners from that library as its primary/default set of planners. The planners in OMPL are abstract; i.e. OMPL has no concept of a robot.

The planner that the OMPL library defaults to is **RRTConnect**, so the initial, simplest, and most natural choice was to use the default configuration. The performance achieved was acceptable; however, some "unnatural" planning of certain trajectories was observed. For instance, a point-to-point planning towards a point very close to the initial pose of the robot resulted in a complex trajectory. This prompted an investigation into other planners provided by OMPL. To avoid proceeding blindly without any basis, reference was made to the papers in [22] and [23].

From which, in general, it emerged that roadmap-based planners found better solutions regarding the path specification, that is, path length and path time. On the other hand,

computational cost of Tree-based planners are much lower than roadmap-based planners. Among Tree-based planners, Bi-directional Transition-based RRT (**BiTRRT**), **RRTConnect** and **RRTstar** have superior performances in comparison with other RRT methods. Having already tried RRTConnect, the other two were tested, and it was concluded that sometimes RRTstar had better performance; however, being asymptotically optimal, it used up all the provided planning time: in some specific cases, this was very useful, but in the majority of cases BiTRRT returned trajectories almost identical in a very short time. The latter was preferred in accordance with the intention of identifying a trade-off between trajectory quality and execution time. Although not completely eliminating the problem initially encountered with RRTConnect, BiTRRT was preferred because, through a large number of experiments, it tended to return trajectory artifacts less frequently than RRTstar. This algorithm tries to add new states to the tree in order to find short, low-cost paths by performing transition tests. With the intention of further improving the situation, other planners such as **RRT**, **TRRT**, **PRM**, and **PRMstar** were considered. However, for the first three, the results were inferior to both BiTRRT and RRTConnect, while for the last one, similar considerations to RRTstar can be made. All the mentioned planners were tested both with the default configuration and by changing parameters such as the optimization objective, and also by introducing a termination condition.

Not satisfied with the results, efforts were made to further enhance the planning, and thus we considered using two other planners:

- **CHOMP** [24]
- **Pilz Industrial Motion Planner**

The motivation that led us to test CHOMP is that, first of all, while most high-dimensional motion planners separate trajectory generation into distinct planning and optimization stages, CHOMP capitalizes on covariant gradient and functional gradient approaches to the optimization stage to design a motion planning algorithm based entirely on trajectory optimization. Secondly, there was considerable curiosity about the possibility of using OMPL and then employing CHOMP as post-processors by using it as an OptimizerAdapter. However, the enthusiasm was dampened as simulations using CHOMP as planners yielded discouraging results, with planning failures in many cases. When used as post-processors, no significant improvements were observed, which is why this option was ultimately discarded.

The reason for investigating Pilz, on the other hand, was more subtle and reasoned. After initial attempts, we delved into the details of Cartesian path planning and found that the chosen planner has no influence on this step of the planning process, indeed, for historical reasons, the MoveIt MoveGroup interface exposes a `computeCartesianPath` API that uses the default Cartesian Interpolator functionality in MoveIt. This Cartesian planner is greedy and easily gets stuck in local minima as it does not provide any functionality for avoiding joint limits or restarting. In practice, the use of this Cartesian Planner is not recommended. In light of the aforementioned considerations, and considering that Pilz allows concatenation of multiple trajectories and planning the trajectory all at once (through the sequence functionality), reducing the computational load of planning and enabling following a predefined path without stopping at intermediate points, it seemed more than reasonable to test this option. Another motivation for using Pilz lies in its determinism. The initial idea was to use this planner

for all planning tasks; however, due to time constraints, we were unable to implement `moveit_msgs::msg::MotionSequenceRequest`.

Nonetheless, to ensure that the configuration of the new planner was not merely an exercise, we attempted to use Pilz in the point-to-point planing. This is due to the fact that from our experiments it emerged that the cartesian planning often failed or was suboptimal when the immediately preceding point-to-point planning put the robot in configurations that are close to the joint limits. However, this approach also presented an issue: configuring Pilz requires defining the joint acceleration limits, and when these were added, the Cartesian Interpolator of MoveIt stopped interpolating, resulting in non-smooth curved trajectories. The reason why this happened, however, remained an open issue due to time constraints, and for details on future improvements and solutions, please refer to the section 2.2.8.

Therefore, at the end, the choice of planner has fallen on BiTRRT for the point-to-point plannings and MoveIt's Cartesian Interpolator for the cartesian path provided as input by the user.

2.2.6 Kinematic redundancy management

In the context of the project developed for the Fanuc M20iA/35M robot, a prismatic slide was introduced, thus bringing the robot to have seven degrees of freedom. The effective management of this redundancy is closely linked to the kinematic solver employed, which in this case is KDL (Kinematics and Dynamics Library), the default solver for MoveIt!2. This solver is based on a numerical approach, which certainly has the advantage of converging very quickly. The KDL plugin wraps around the numerical Jacobian-based inverse kinematics solver provided by the Orocos KDL package and works fine with robots having a number of DOF greater than 6. Being a Jacobian-based algorithm, this solver exploits redundancy by minimizing the squared norm of velocities in the first order inverse kinematic problem (see section **??**). As a result the inverted pose should be as close as possible to the current pose, joint space-wise.

It was also tried to change the default solver of MoveIt!2 to test the performances with a different optimization goal. Indeed, an attempt was made to maximize manipulability using TRAC-IK, which runs a pseudoinverse Jacobian IK method and a nonlinear optimization IK method concurrently. Although an improvement in performance was expected, tests actually revealed the opposite, so KDL was chosen.

The management of redundancy is also strongly influenced by the planner used, which in this case, as explained is BiTRRT. We configured the planning algorithm to minimize the path length.

It was deemed necessary to notify the reader that, in addition to the use of the slide, the end-effector has also been integrated. This step is significant because due to the considerable size of the end-effector, it easily self-collides with the robot arm. So the degree of freedom gained with the addition of the prismatic slide is "almost lost" by the reduced movements imposed by the size of the end effector.

2.2.7 Communication with other nodes

The `planning_node` communicates with other nodes to fulfill the functionalities presented in section 2.2.2 as follows:

- **move_group** through `move_group_interfaces`, a simple and intuitive interface that, by allowing to exploit the functionalities of the aforementioned node, has played a role simplifying various operations, particularly the performed planning;
- **RViz** thanks to the topic `/rviz_visual_tools`. However, this is an indirect communication realized through APIs:
  - **MoveItVisualTools** used to display in RViz the planned trajectories, the user's desired path, the starting and ending points of point-to-point planning, as well as the status of planning and execution operations. This is made easier by the presence of helper functions that publish markers precisely on the mentioned topic and has proved to be very useful not only for visualization but also for debugging, as it made it possible to visually appreciate the quality, or lack thereof, of the trajectories.
  - **RVizVisualToolsGUI** whose main role is to wait for the user to authorize the planning and execution of trajectories.
- **Controllers** through the respective action clients: in this case, it is not appropriate to delve into detail, but it was chosen to remain generic as such communication can dynamically change depending on the certified controllers and with it the names of the actions and the type of instantiated action client. Reference is made to actions because for execution the `move_group` interface was not used, but direct communication with the controllers was implemented; for further information on this implementation choice, the reader is invited to refer to the section 2.2.4 where the motivations for such choice are provided.

The planning node operates generically with any path saved in a bag file in the form of a `geometry_msgs/msg/PoseArray`. Therefore, it is independent of the tool used to generate the trajectory. However, in the present case, it is utilized by reading trajectories generated with the Task Space Path Generation Module (2.1). Specifically, indirect communication is established with the over mentioned module because the path is read by the `/cartesian_path` topic and interpreted as a `geometry_msgs/msg/PoseArray` message.

2.2.8 Possible improvements

The goal of developing a general, robot-agnostic module was kept in mind during each phase of the design, however, some improvements can certainly be made. First, an attempt could be made to implement the action clients of the `planning_node` class as generic types, so that there is only one attribute that accepts different types of action clients. A similar refactoring could be done for the related callbacks, also implementing them as template methods.

As mentioned in the section 2.2.5, Pilz could be used for point-to-point planning at the home pose, where this was defined, and try to use the LIN motion command of the same

planner for planning at the first point of the desired path, in case this failed re-execute this planning as a point-to-point as well. As for the Cartesian path, a `MotionSequenceRequest` should be implemented that contains a list of successive goals and a `blend_radius` parameter to be configured appropriately. Not surprisingly, if the given `blend_radius` is greater than zero, the corresponding trajectory is smoother.

Another idea would be to test the performance of `ST-RRT*`: a bidirectional, time-optimal planner for planning in space-time that was included in the latest version of OMPL 1.6.0 but has not yet been integrated into MoveIt!2. This planner outperforms `RRT-Connect` and `RRT*` on both initial solution time and final cost obtained, as seen in [25].

CHAPTER 3

WP3 - SIMULATION, EXECUTION AND ANALYSIS

This chapter is dedicated to the description of the design choices we made for the independent joint controllers and the trajectory analysis tool we developed.

In order to design the controllers we first estimated the dynamical model of the robot. The estimation method that has been used is described in section 3.1. Section 3.2 describes the design choices we made for the controllers. After designing the controllers we performed several simulations to test the performance of the system. Section 3.3 describes the configuration of the simulation environment Ignition Gazebo.

The trajectory analysis tool we developed is used to analyze the performance of the controllers in terms of trajectory tracking error. Section 3.4 describes the trajectory analysis tool we developed, and section 3.5 contains a critical analysis of the results we obtained.

3.1 Dynamical System Estimation

Independent joint control is the simplest control strategy for a robot manipulator, as it aims to control each joint independently of the others. The dynamical model of the robot is

$$\mathbf{B}(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{F}_v\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}) = \tau \tag{3.1}$$

Note that there are unit gear ratios for all joints in the URDF file of the robot, so the model doesn't need to take them into account.
It can be set $\mathbf{B}(\mathbf{q}) = \overline{\mathbf{B}} + \Delta\mathbf{B}(\mathbf{q})$ where $\overline{\mathbf{B}}$ is a diagonal matrix in which the mean value of the inertia of each joint is present and $\Delta\mathbf{B}(\mathbf{q})$ is a matrix that contains all the interaction terms of the inertia matrix between the various terms. Thus, the equation 3.1 can be rewritten as

$$\overline{\mathbf{B}}\ddot{\mathbf{q}} + \mathbf{F}_v\dot{\mathbf{q}} + \mathbf{d} = \tau \tag{3.2}$$

where $\mathbf{d} = \Delta\mathbf{B}(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q})$ contains all the non-linear terms of the model that depend on the configuration of the robot.

Decentralized control techniques such as Independent Joint Control are based on the dynamical model in equation 3.2. The idea is to design a controller for each joint that is able to compensate for the non-linear terms of the model d that are treated as disturbances. This control strategy works well if two conditions are met:

- The reference trajectory is slow enough: the module of the disturbance d grows with the velocity and the acceleration of the joints, so if the reference trajectory is slow enough the disturbance will be small;
- The gear ratio of the joints is high, because it has the effect of reducing the disturbances. This may be a problem because transmission ratios were not taken into account in the robot description file.

In order to design the controllers $\overline{\mathbf{B}}(\mathbf{q})$ and $\mathbf{F}_v$ need to be obtained. The latter is the vector of viscous friction coefficients of the joints. They can be easily obtained by using the `friction` parameter of the `joint` tag in the URDF file. Deriving $\overline{\mathbf{B}}(\mathbf{q})$ is more challenging. Three different approaches have been thought of:

- Use first principles (Lagrange's formulation or Newton-Euler's formulation) to derive the matrix $\mathbf{B}(\mathbf{q})$ and then derive $\overline{\mathbf{B}}(\mathbf{q})$ from it. This approach requires some effort, since there are only the inertia matrices of the individual links in the URDF file and there is a need to derive the inertia matrix of the whole robot. Moreover, it would be a waste of time, since there is no need for the entire matrix $mathbfB(\mathbf{q})$, but only its diagonal;
- Use a system identification approach to identify the model from observed data taken from the Ignition Gazebo simulator. This approach is very interesting, but it requires a lot of time and effort to be implemented, and it is not the focus of this project;
- Use the `massMatrix` function of MATLAB's Robotics System Toolbox to directly obtain the matrix $\mathbf{B}(\mathbf{q})$ from the URDF and then derive $\overline{\mathbf{B}}(\mathbf{q})$ from it. The problem with this approach is that the `massMatrix` function doesn't work with symbolic variables, so it is not possible to get the dynamic model in symbolic form;

It was decided to use the third approach because it is the simplest one. To circumnavigate the problem of not being able to obtain the dynamical model in symbolic form it was decided to use the `massMatrix` function to obtain observations of $\mathbf{B}(\hat{\mathbf{q}})$ at feasible random configuration of the robot (within joint limits), and then approximate the mean value of the $\mathbf{B}$ matrix by means of Monte Carlo method (that is, by taking the mean value of the observations). In this way the matrix $\overline{\mathbf{B}}'(\mathbf{q})$ was obtained. By eliminating the terms that are not on the main diagonal, we obtain the $\overline{\mathbf{B}}(\mathbf{q})$ that can be used to design controllers.

The transfer function of the i-th joint is $G = \frac{1}{(bs^2+fs)}$ where b is the i-th element of $\overline{B}$ and f is the i-th element of the vector of frictions (friction) both corresponding to the i-th joint.

3.2 Independent Joint Controller Design

In order to distribute the controllers we developed, the package `acg_resources_fanuc_m20ia_35m_controllers` was created. This package contains the

configuration files for the provided controllers and the launch files for the simulation in Ignition Gazebo with the said controllers.

3.2.1 Joint space controllers

As for the joint space controllers, we used the independent joint control with both position and velocity feedback loops. A controller was designed for each joint. As this controller can be reduced to a PID controller, we decided to use `ros2_control`'s `JointTrajectoryController` that, according to the documentation, implements a PID law when activating the `effort` control interfaces. In this way, we only needed to provide the PID parameters for each joint and put them in the `ros2_controllers.yaml` file inside the `config` folder of the `acg_resources_fanuc_m20ia_35m_controllers` package.

3.2.2 Joint trajectory controller configuration

The designed independent joint controller is called `fanuc_m20ia_35m_PID_controller`. It is a `JointTrajectoryController` and as such it it configured to receive the trajectory from the action server `/fanuc_m20ia_35m_arm_joint_trajectory_controller/follow_joint_trajectory`.
Furthermore, the reference trajectory must be of of type `trajectory_msgs/JointTrajectory`. The controller handles the control of all the joints simultaneously. In order to allow executions for all of the planning groups that were defined in section 1.4, the parameter `allow_partial_joints_goal` is set to `true`.

3.2.3 PID controllers design

The PID controllers were designed for each joint through Matlab's `rltool` with PID Tuning. To do this, it was decided to impose some design requirements on ourselves: a settling time of around $1s$ and a maximum overshoot of 10%. Bringing these controllers to the simulated robot, it was realized that the performance were not as good as expected, due to excessive oscillations from using independent joint control.
Therefore, we decided on to calibrate the robot directly in simulation, by changing the parameters of the PIDs at runtime via `rqt dynamic reconfigure`. For PID calibration, the empirical procedure [26] was then considered, and the following steps were applied:

- **Set all parameters to zero**: Start by setting all the PID parameters (K_p, K_i, K_d) to zero. In this way, the controller will have no effect on the controlled system.
- **Increase Proportional (K_p)**: Gradually increase the proportional parameter (K_p) until the system starts to respond in an oscillatory manner. This means that the control signal is oscillating above and below the desired value. Find the point where these oscillations are just barely present.
- **Adjust the Integral Action (K_i)**: At this point, gradually introduce the integral action (K_i). This will help to reduce the steady-state error, i.e., the discrepancy between the desired value and the one actually achieved. Start with a small value for K_i and increase it until the steady-state error decreases without introducing significant new oscillations.

- **Add Derivative (K_d)**: Finally, you can add the derivative action (K_d) to reduce overshoots and speed up the system response. Again, start with a small value for K_d and increase it gradually until you get a stable and fast response.
- **Refine and Optimize**: Once you have introduced all three components of the PID, continue experimenting with the parameter values to optimize the system's performance. This might require some practice, and you might need to find a compromise between response speed and system stability.
- **Verify and Evaluate**: Finally, check the system's response to setpoint changes or disturbances and make any necessary corrections to the PID parameters if necessary. Continue to monitor and evaluate the system's performance over time.

At the end of this process, the parameters obtained are summarized in table 3.1.

	K_p	K_i	K_d
Slide	60600	3000	12000
Joint 1	8250	5500	1000
Joint 2	2000000	70000	8000
Joint 3	500000	1000	500
Joint 4	8000	300	10
Joint 5	200000	5000	100
Joint 6	8000	100	30

Table 3.1: PID parameters

3.3 Configuring Ignition Gazebo

In order to verify the performance of the controller that was designed, the **Ignition Gazebo** simulator was set up to run a dynamic simulation of the robot in closed loop with the controller.

First, it was necessary to manually edit the URDF file of the robot to make it compatible with the Gazebo simulator. This means that:

- Each `<link>` element must have an `<inertia>` element specifying the inertial properties of the link;
- A `<link>` named "world" must be the first link of the robot description;
- The correct command and state interfaces must be specified for each joint. The chosen command interfaces will have an impact on the kind of simulation we are going to perform with the simulator;
- The hardware `ign_ros2_control/IgnitionSystem` must be specified in the robot description file;
- The `IgnitionROS2ControlPlugin` must be loaded in the `<gazebo>` element for the `<robot>` element and the path to the `ros2_controllers.yaml` file must be provided to the plugin.

As for the inertial elements and the connection with the "world," they were already added in the basic URDF of the robot that was produced in WP1, that is, the one contained in the robot description package. The other modifications needed to the robot description file were dynamically added to the URDF by means of `xacro` in the MoveIt! configuration package. This choice was made to avoid having multiple URDF files for the same robot for each kind of dynamical simulation (or simple visualization) we want to perform. For this purpose the following xacro arguments were added to the `.urdf.xacro` file:

- `use_mock_hardware`: if true, the `ign_ros2_control/IgnitionSystem` is added to the robot description file, and the `IgnitionROS2ControlPlugin` is loaded in the `<gazebo>` element, otherwise the fake hardware `mock_components/GenericSystem` is used;
- `ros2_controllers_path`: the path to the `ros2_controllers.yaml` file. This is useful to choose which controllers to load in the simulation.
- `command_interface`: the command interface to be loaded for each joint. This is useful to choose which kind of simulation to perform. The available options are `position`, `velocity` and `effort`.
- `initial_positions_file`: the path to the file containing the initial positions of the joints. This is useful to set the initial state of the robot in the simulation.
- `initial_velocities_file`: the path to the file containing the initial velocities of the joints. This is useful to set the initial state of the robot in the simulation.
- `initial_efforts_file`: the path to the file containing the initial efforts of the joints. This is useful to set the initial state of the robot in the simulation.

Secondly, it was necessary to write some launch files to spawn the robot in Gazebo and to load the controllers that were designed into the `controller_manager` node of the robot. The following launch files were written:

- `gazebo_ros2_control_demo`: this launch file spawns the robot in Gazebo with the command interfaces specified in the `command_interface` argument and loads the controllers specified in the `controller_list` argument. It also loads the `joint_state_broadcaster` controller and the `robot_state_publisher` to publish the state of the joints and of the robot in the `/joint_states` and `/robot_description` topics respectively. RViz is launched to visualize the state of the robot in real time. This launch file can be used to perform a dynamic simulation of the robot with the specified controllers;
- `gazebo_moveit_demo`: this launch file is similar to the previous one, except it also launches the `move_group` node and the `moveit_simple_controller_manager` nodes to allow for trajectory planning and execution with MoveIt!. RViz is loaded with the MoveIt! planning plugin to easily plan and execute through the GUI. This can only be used with `JointTrajectoryController` controllers because `moveit_simple_controller_manager` only supports this kind of controllers.

Some problems encountered while working with Ignition Gazebo:

- Normally Gazebo should prevent the robot from falling under the effect of gravity when the simulation starts, until a controller is spawned. However, this is not the case when:
  - `effort` command interfaces are used. In this case, the robot falls under the effect of gravity and stops falling when the motion controller is loaded (this is a safety feature of `JointTrajectoryController` controllers). Differently from old versions of Gazebo, the controllers are only loaded if the simulator is not paused; this prevents loading the controllers before the action of gravity. To mitigate this problem, the PID controller spawner node is run before the simulation starts. By default, the spawner waits until the controller manager is instantiated, then loads the controller. The maximum waiting time of the spawner has been increased to 60 seconds to avoid problems;
  - Joint effort limits are too low in the URDF of the robot. This is the case of the panda robot (!). In this case even using `position` or `velocity` command interfaces, Gazebo is not able to prevent the robot from falling under the effect of gravity. By raising the maximum joint efforts on the URDF of the robot the issue is solved;
- The `create` command in `ros_gz_sim` package that was used to spawn the robot in Gazebo doesn't allow to set the initial joint positions of the robot. For further info see this github issue;
- The provided collision mesh for the slide was penetrating the floor and this cause a generic ODE error in Gazebo. After several days of debugging, the cause of this problem was discovered, and it was by spawning the robot slightly above the floor, passing the topic `-z 0.0001` to the `create` command in `ros_gz_sim` package;

3.4 Trajectory Analysis tool

The Python package `acg_resources_tools` was created to contain the trajectory analysis tool `trajectory_analyzer`. This tool takes from the command line the name of the file bag containing all the trajectories made by the robot, and it plots the planned and executed trajectories from that bag file, and calculates the Mean Squared Error (MSE) between them. Through the `topic` parameter, it is possible to specify the topic contained in the bag file that contains the trajectory you want to plot, for example, the sine wave trajectory or the trajectory that takes the robot from the home position to the initial point of the sine wave. In addition, through the optional parameter `task-space`, it is possible to enable the analysis of trajectories in the task space, as discussed in more detail in the chapter 4.

The messages contained in the bag file, whatever the trajectory type as long as it is in the joint space, must be: `control_msgs/action/FollowJointTrajectory_FeedbackMessage`. This type of message contains all the information we need: actual position, planned position, actual velocity, planned velocity and also the error between the two, for each joint. The tool reads the bag file, extracts the messages, and then for each joint is plotted:

- Executed vs planned trajectory;

- Actual speed vs planned speed;
- MSE between executed and planned trajectory.

The tool is robot agnostic, it can be used for any robot that uses the `FollowJointTrajectory` action, and it is also trajectory agnostic, it can be used for any trajectory that is in the joint space, as long as the bag file contains the correct messages. This can be seen in the Figs. 3.1 and 3.2, where the tool was used to plot the trajectory, and relatives velocities, of the Panda robot, which is a different robot from the one used in the project. The Matplotlib graphics library was used to make these plots. There are two windows, one for positions and one for velocities, and in each of these the corresponding plots for each joint are arranged on two columns. It should be noted that being robot agnostic the number of rows changes according to the number of joints in the robot. Also, being a ROS package, the tool does logging on the ROS network.

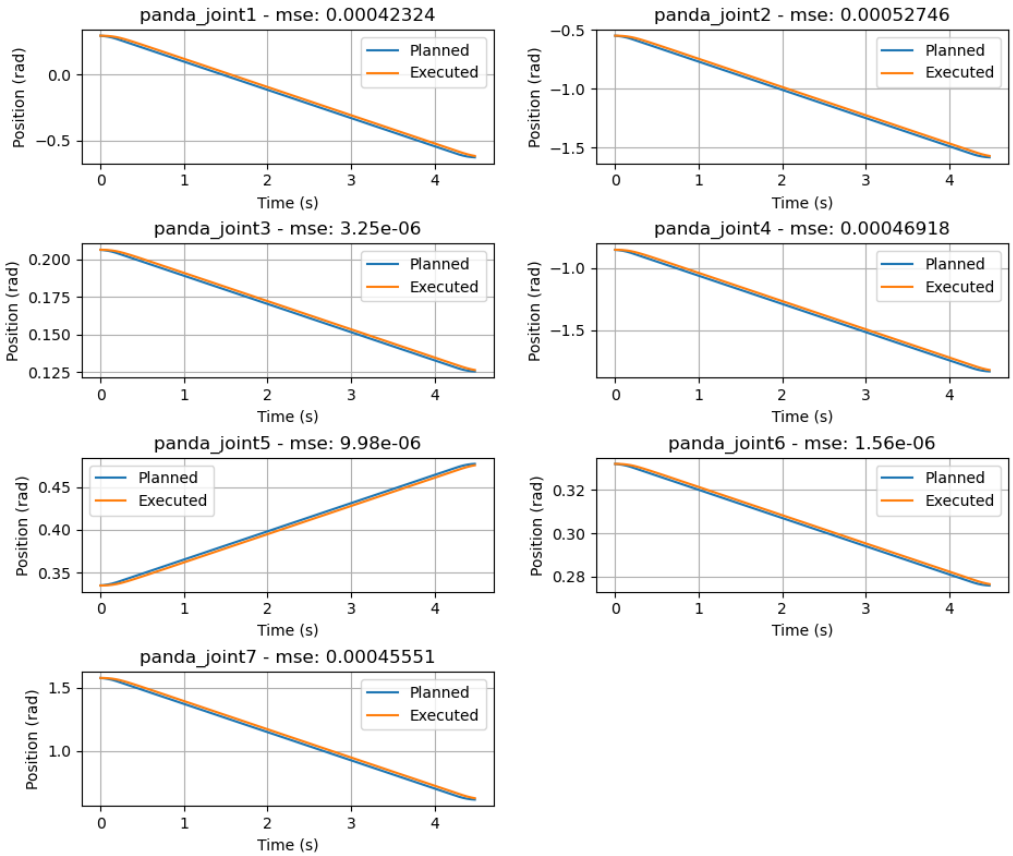


Figure 3.1: Panda position trajectory

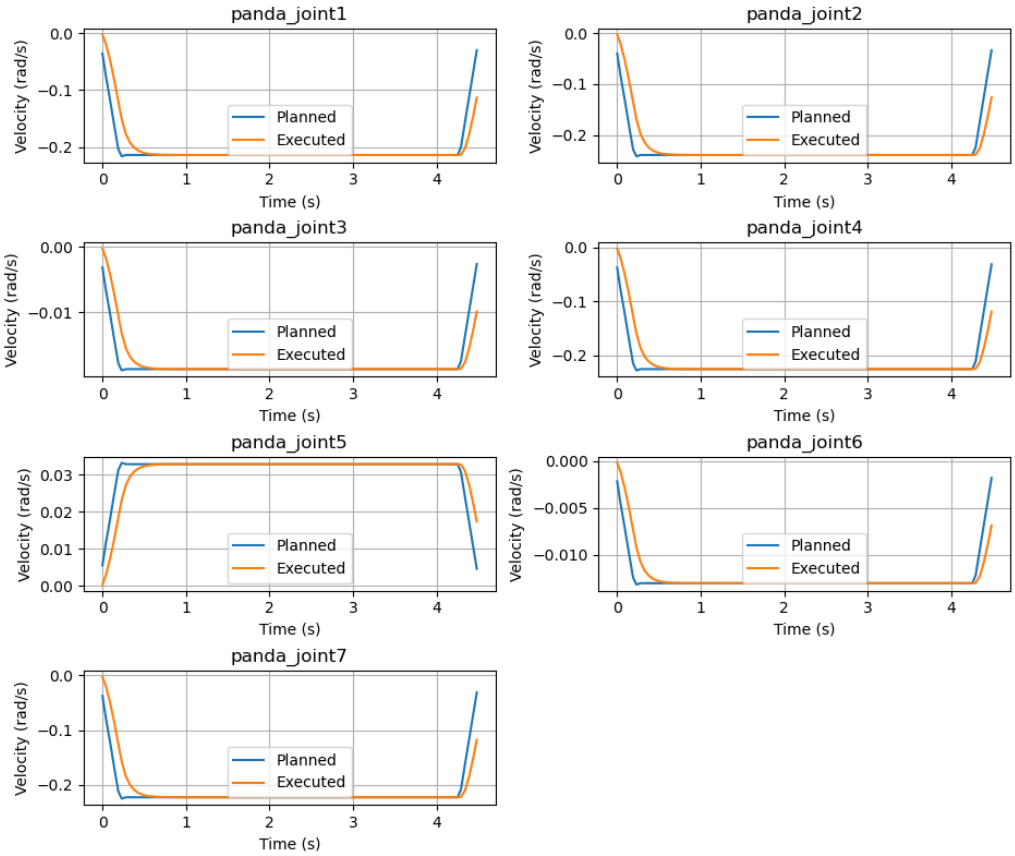


Figure 3.2: Panda velocity trajectory

3.5 Tracking performance analysis

Downstream of WP3 is intended to perform an analysis of a set of trajectories chosen to show how the robot behaves. Specifically, the path required is a sine period with amplitude A $\geq$ 0.5 m. For this, 7 sinusoidal paths with one period all with different characteristics were chosen, and are presented below. These paths were provided in the `task_space_path_generator` package in the `resource\Example_trajectories` folder, with the `resource\Example_trajectories\Screen` folder that has previews to visualize what the trajectory is. The analyzed trajectory files, on the other hand, are located in the folder `resource\executed_joint_space_trajectories` of the package `acg_resource_tools`.

3.5.1 Trajectory 1

The first trajectory analyzed is a sine with an amplitude of 0.5 m and path length of 1 m (Fig. 3.3). The plane on which the sine is drawn is perpendicular to the ground and parallel to the axis along which the slide moves. This trajectory is executed perfectly by the robot, as can be seen in Fig. 3.4, and the velocities references are also tracked perfectly (Fig. 3.5). The tracking error is minimal and the robot is able to follow the trajectory with precision.

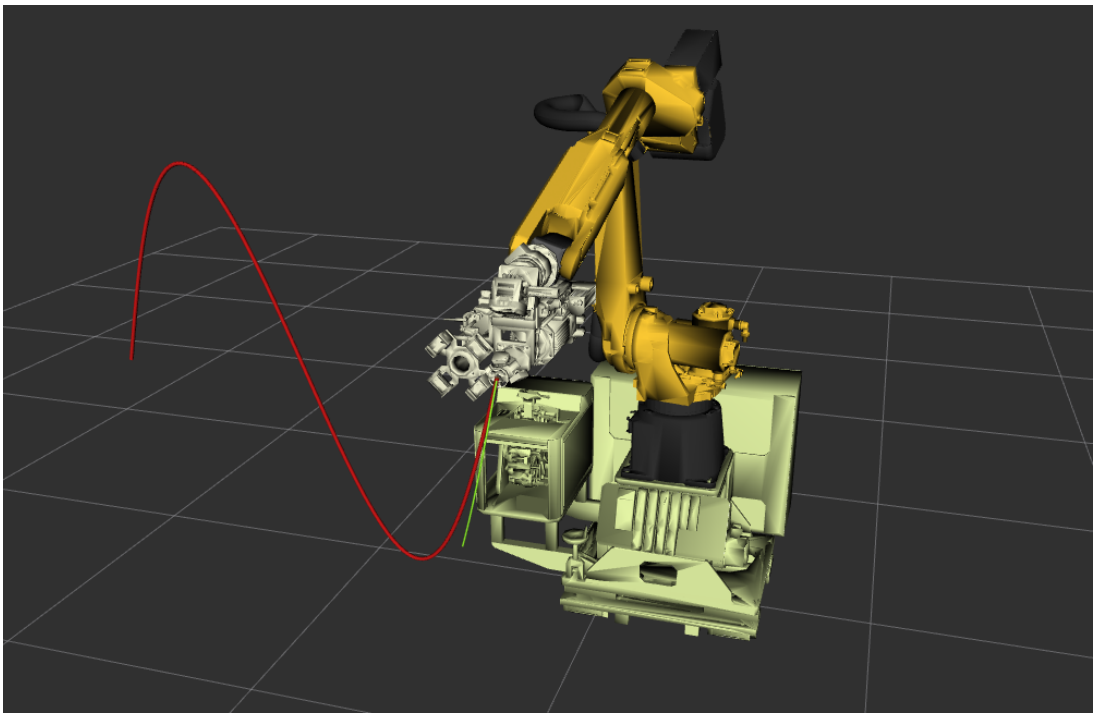


Figure 3.3: Trajectory 1

3.5.2 Trajectory 2

The second trajectory analyzed is a sine with an amplitude of 0.5 m and path length of 2 m (Fig. 3.6). The plane on which the sine is drawn is perpendicular to the ground and inclined to the axis along which the slide moves. With this trajectory things start to get complicated, in fact if this inclination was too high toward the robot, then this would get out of the robot's workspace and could neither be planned nor executed since the robot would go into self-collision. The robot perfectly tracks position and velocities references, as can be seen in Figs. 3.7 and 3.8. The tracking error is practically 0 for each joint.

3.5.3 Trajectory 3

The third trajectory analyzed is a sine with an amplitude of 0.5 m and path length of 2 m (Fig. 3.9). The plane on which the sine is drawn is perpendicular to the ground and parallel to the axis along which the slide moves. It was decided to present this trajectory to highlight the working area of the robot, and how high it can go by performing a full trajectory. The position references are followed perfectly (Fig. 3.10), while for the velocity references it is observed that in some cases there are small spikes, especially of joint 4 (Fig. 3.11). This is due to the fact that the robot is at the limit of its workspace and the controller has to make an effort to follow the trajectory. The tracking error is minimal and the robot is able to follow the trajectory with precision.

3.5.4 Trajectory 4

The fourth trajectory analyzed is a sine with an amplitude of 0.65 m and path length of 2.5 m (Fig. 3.12). The plane on which the sine is drawn is perpendicular to the ground and parallel

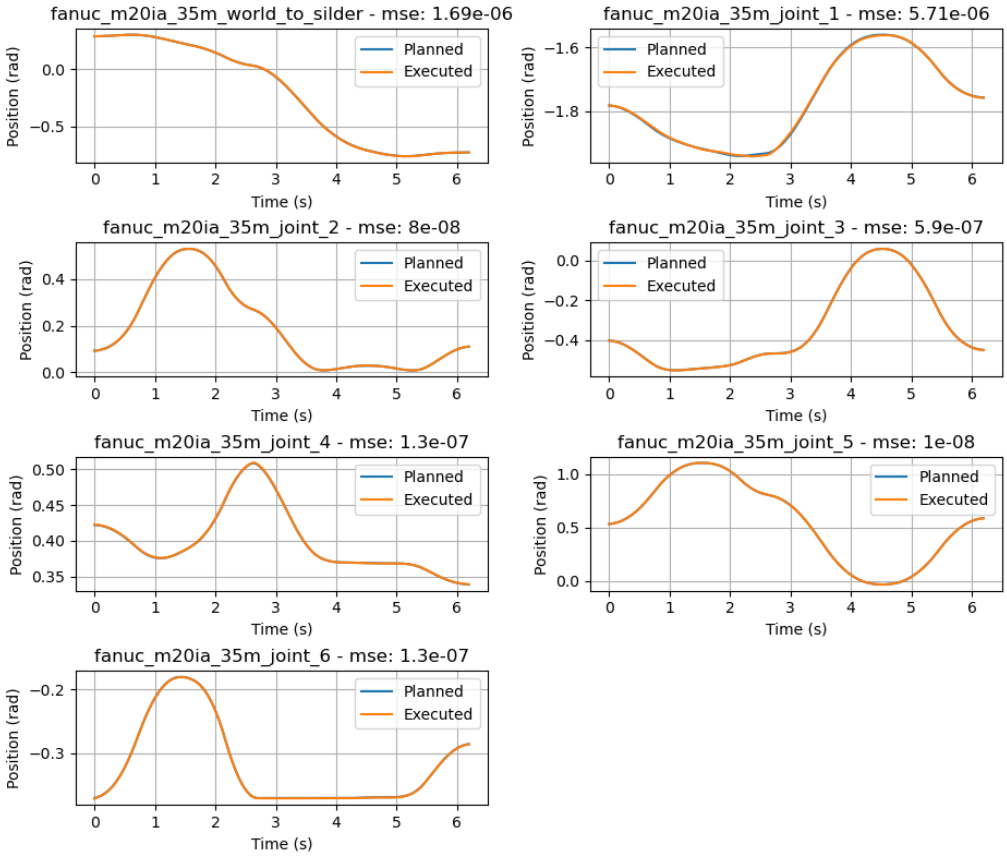


Figure 3.4: Trajectory 1 - Joint positions

to the axis along which the slide moves. This trajectory highlights how due to the presence of the slide, the robot can make very long and wide trajectories. This trajectory is executed perfectly by the robot, as can be seen in Fig. 3.13, and the velocities references are also tracked perfectly (Fig. 3.14). The tracking error is minimal and the robot is able to follow the trajectory with precision.

3.5.5 Trajectory 5

The fifth trajectory analyzed is a sine with an amplitude of 0.55 m and path length of 1.4 m (Fig. 3.15). The plane on which the sine is drawn is perpendicular to the ground and parallel to the axis along which the slide moves. This sine unlike trajectory 1 propagates upward, so it serves to show how the robot again manages to follow the trajectory perfectly despite the limitations that may be given by the drilling tool (Fig. 3.16). Also, the velocities are perfectly tracked, except for joint 1, as can be seen in Fig. 3.17.

3.5.6 Trajectory 6

The sixth trajectory analyzed is a sine with an amplitude of 0.55 m and path length of 1.4 m (Fig. 3.18). The plane on which the sine is drawn is parallel and is exactly above the robot to show how it can execute even such a trajectory. In case this was placed higher, the robot would be with its arm fully extended and would go into singularity, so the planning and execution

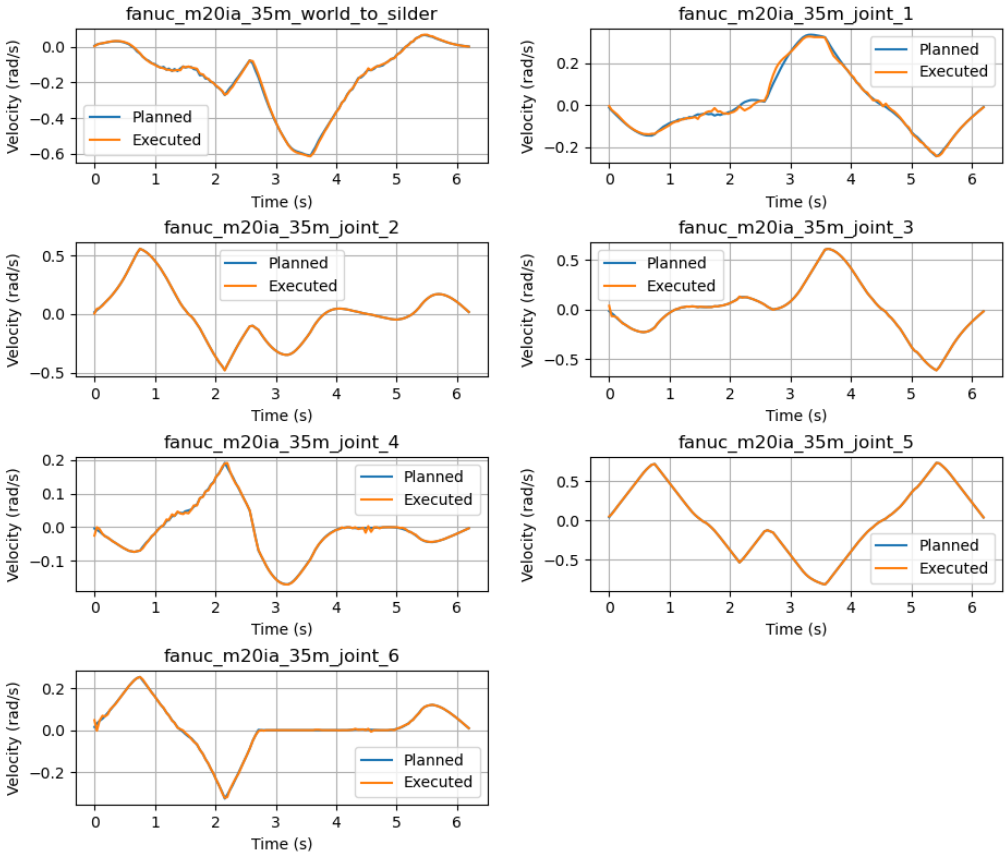


Figure 3.5: Trajectory 1 - Joint velocities

would fail. This trajectory is executed perfectly by the robot, as can be seen in Fig. 3.19, and the velocities references are also tracked perfectly (Fig. 3.20), with a tracking error equals to zero.

3.5.7 Trajectory 7

The seventh trajectory analyzed is a sine with an amplitude of 0.55 m and path length of 1.25 m (Fig. 3.21). This trajectory is the most complex that the robot is able to execute, as it is inclined with respect to both the ground and the axis along which it sleds, and then the sine propagates diagonally. Despite the complexity of this trajectory, even if the planning does not always manage to be perfect, the robot manages to execute it completely as can be seen in Figs. 3.22 and 3.23, with a tracking error equals to zero.

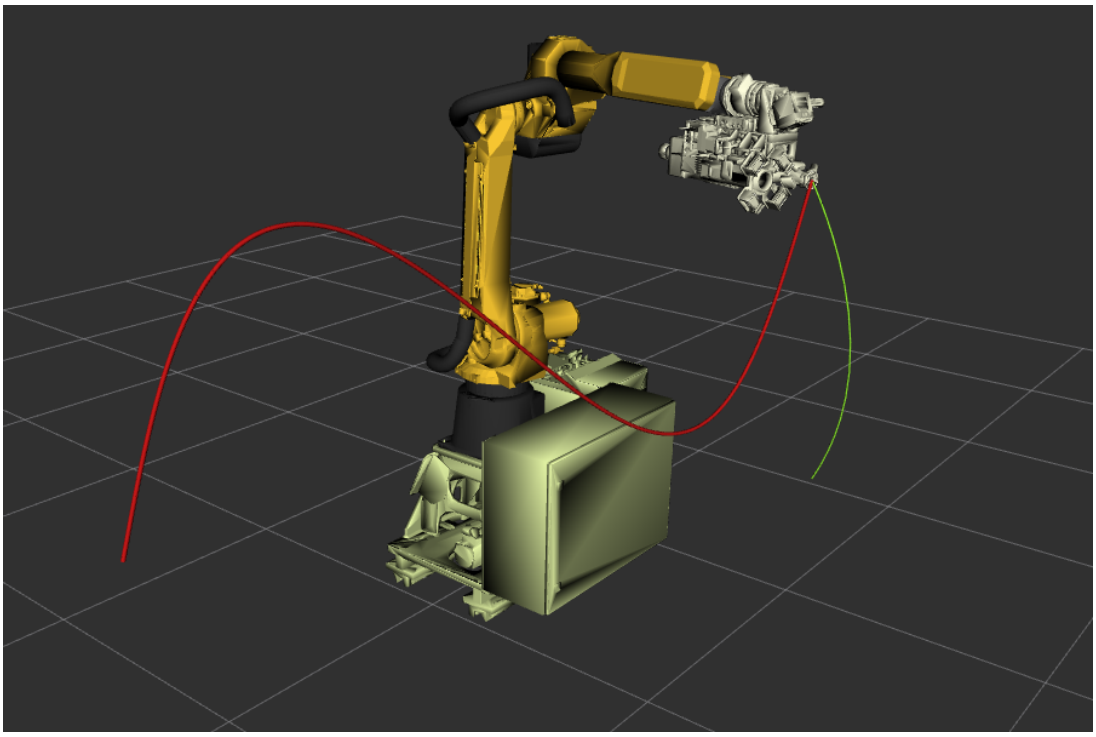


Figure 3.6: Trajectory 2

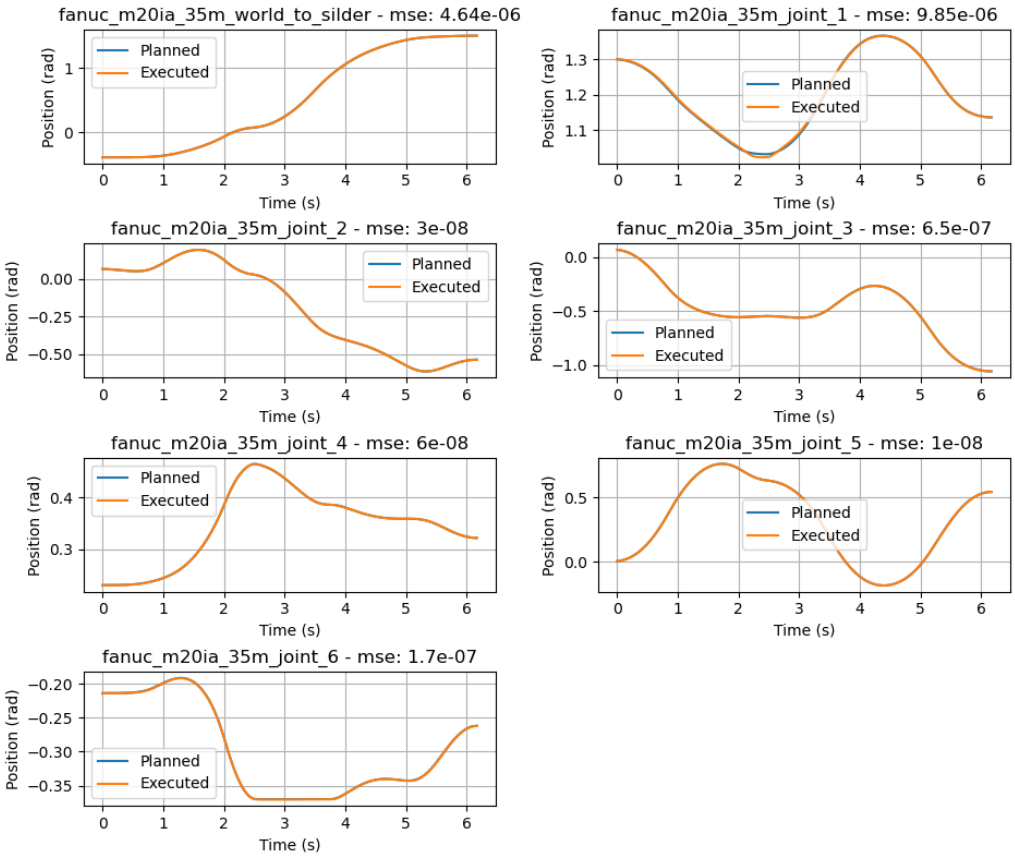


Figure 3.7: Trajectory 2 - Joint positions

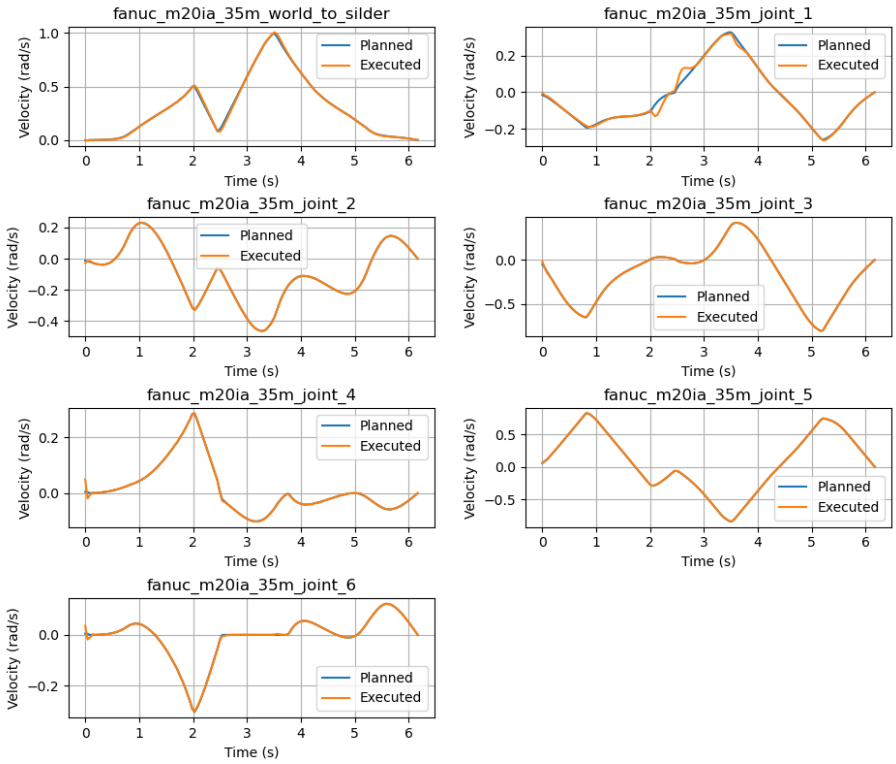


Figure 3.8: Trajectory 2 - Joint velocities

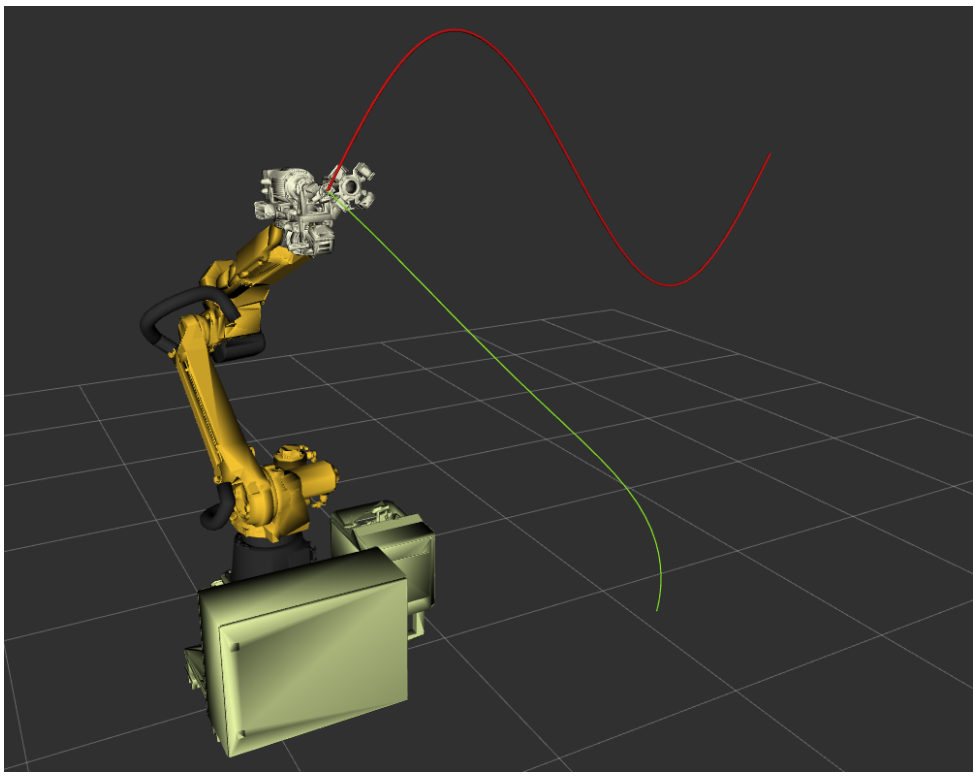


Figure 3.9: Trajectory 3

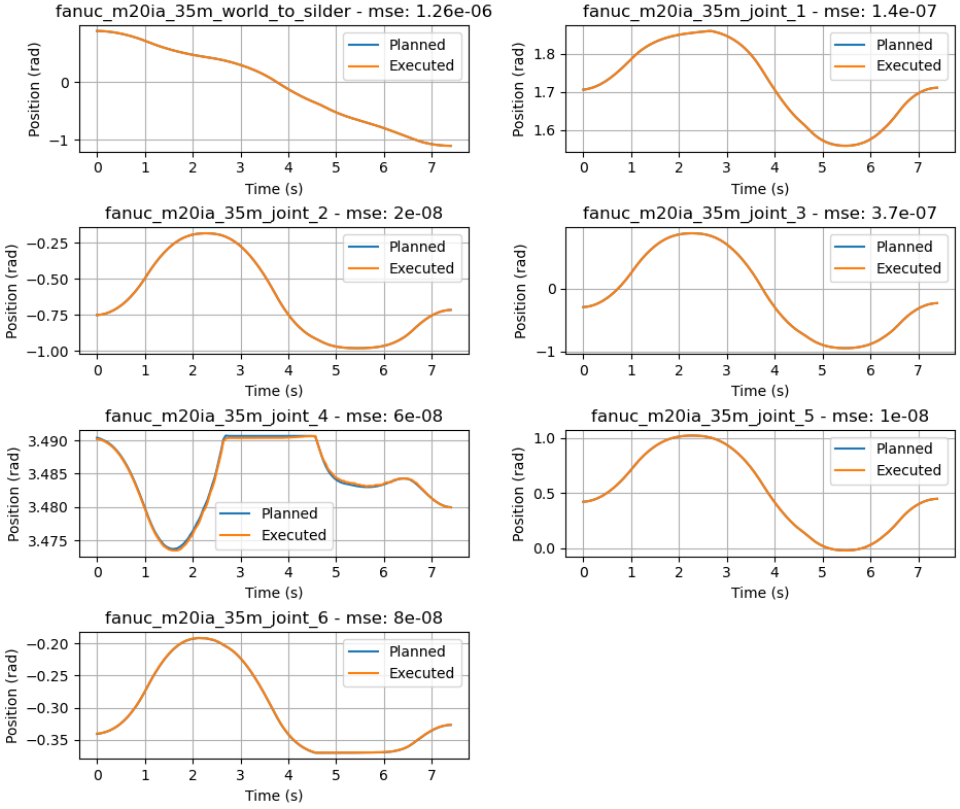


Figure 3.10: Trajectory 3 - Joint positions

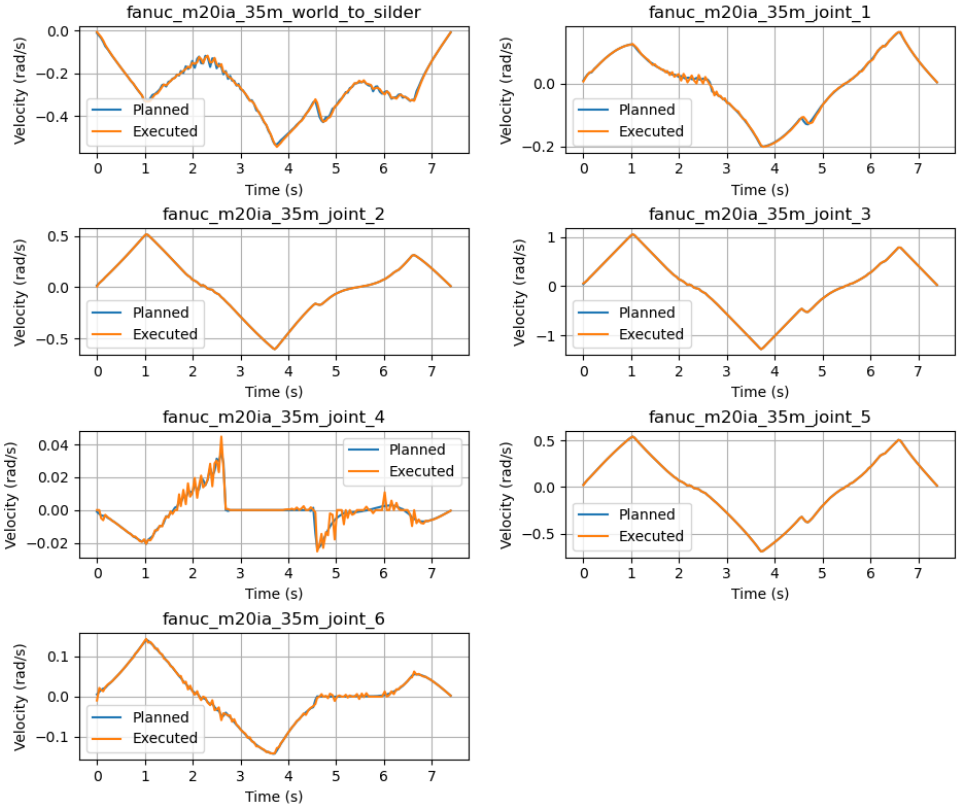


Figure 3.11: Trajectory 3 - Joint velocities

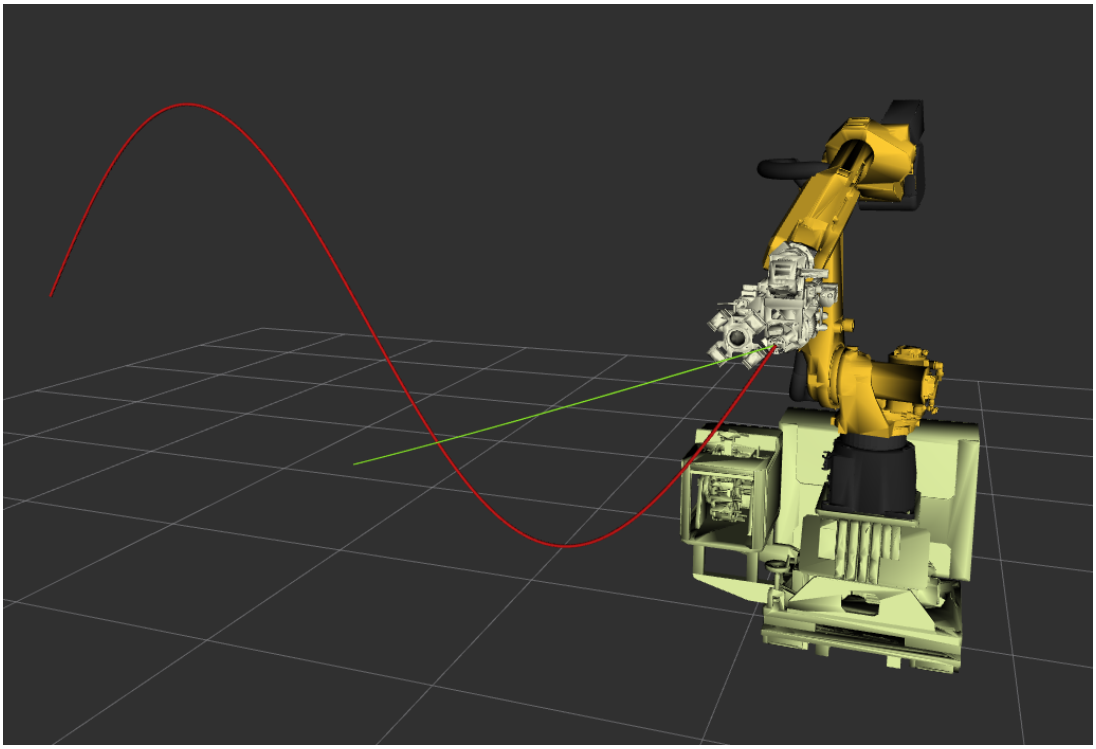


Figure 3.12: Trajectory 4

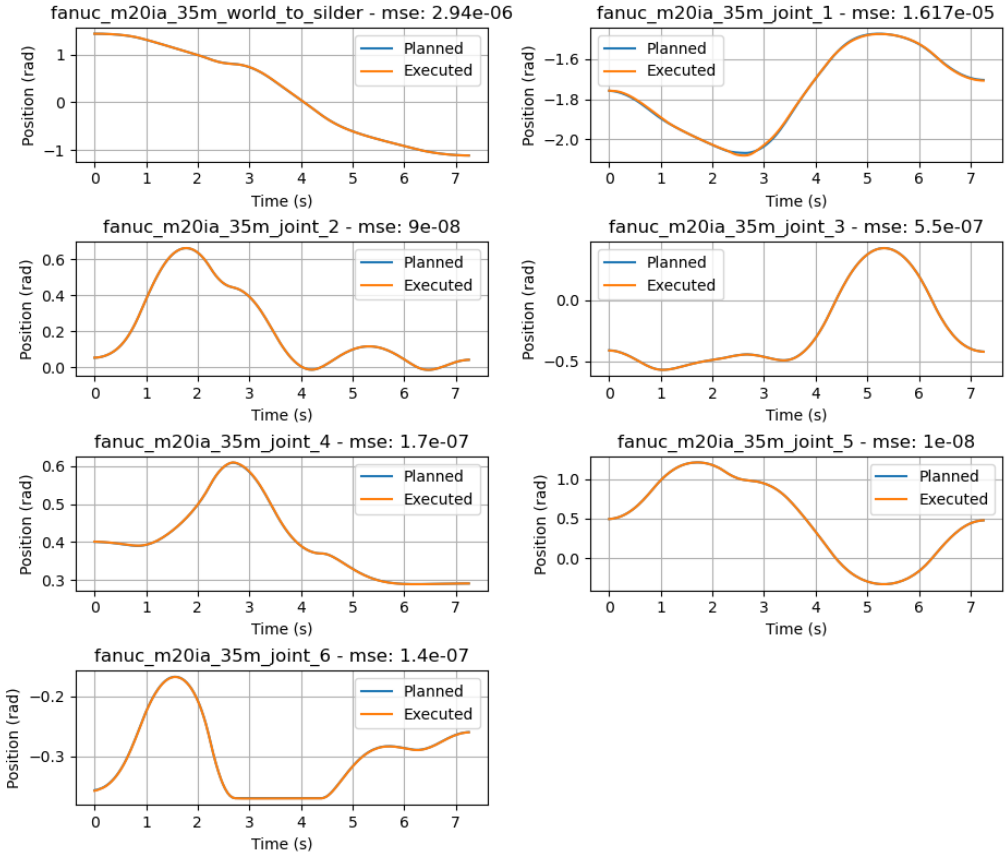


Figure 3.13: Trajectory 4 - Joint positions

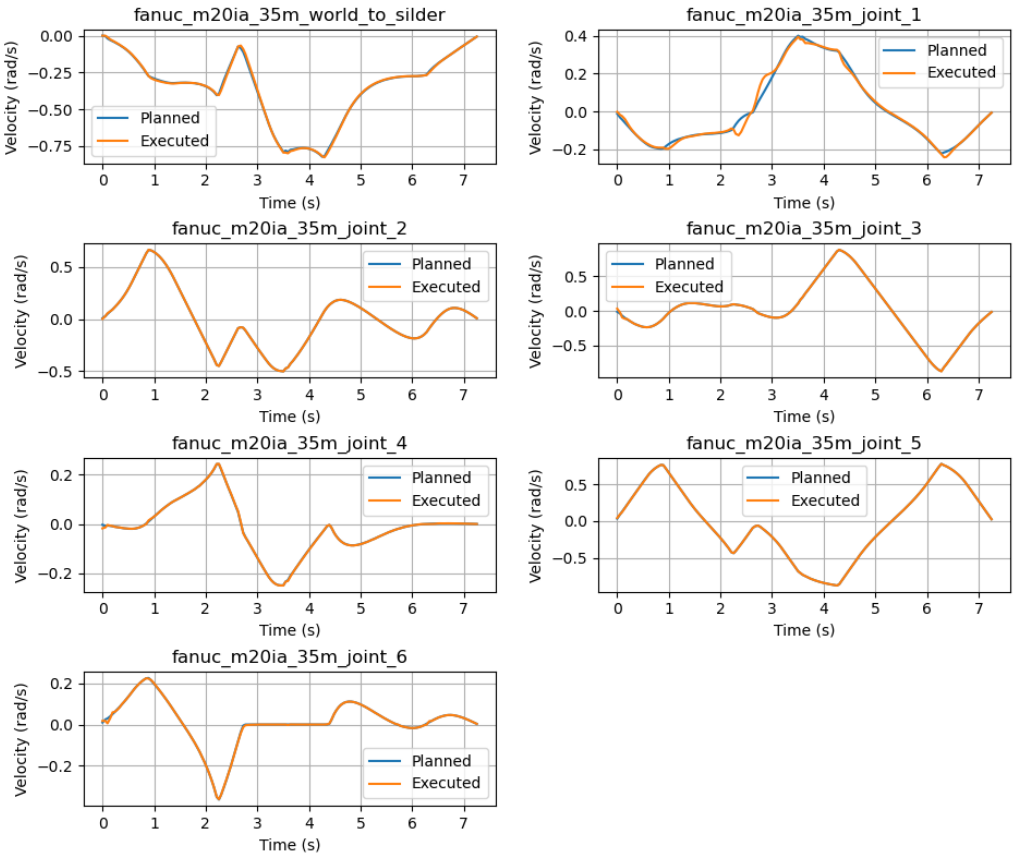


Figure 3.14: Trajectory 4 - Joint velocities

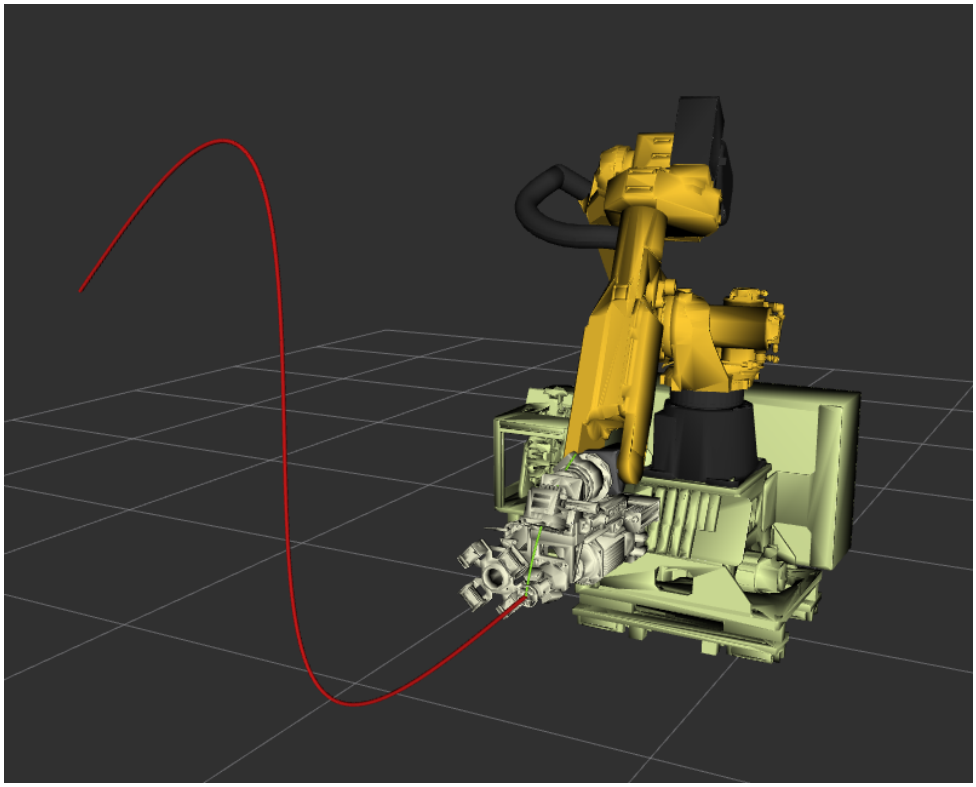


Figure 3.15: Trajectory 5

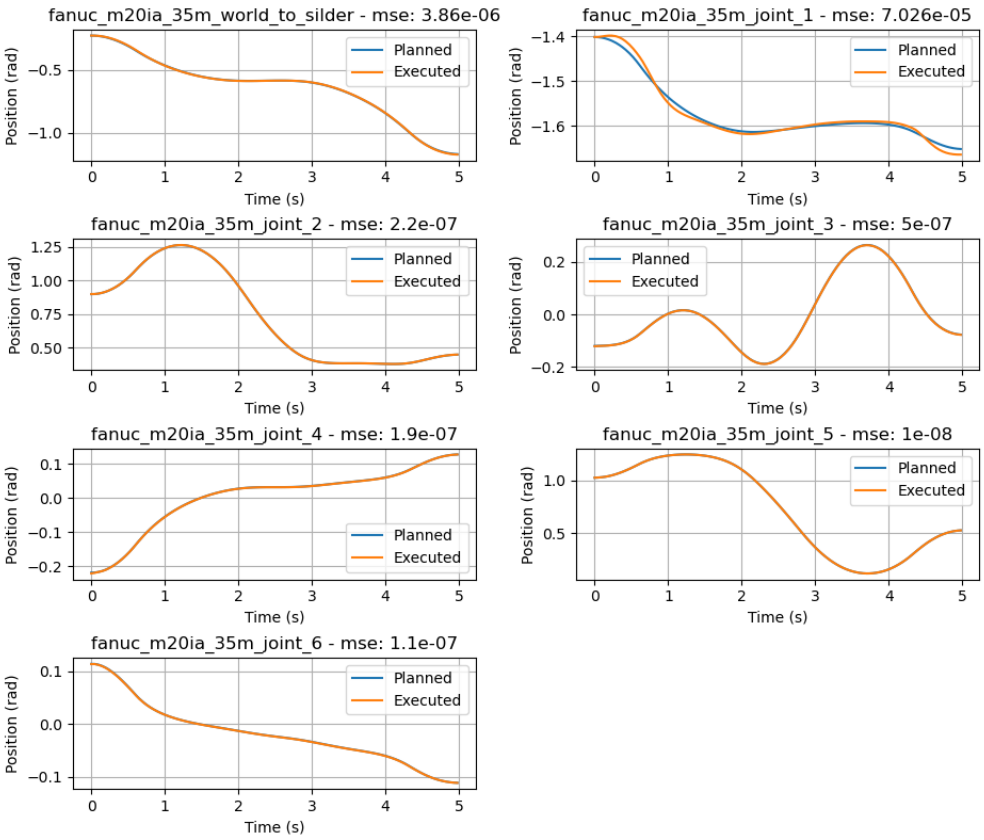


Figure 3.16: Trajectory 5 - Joint positions

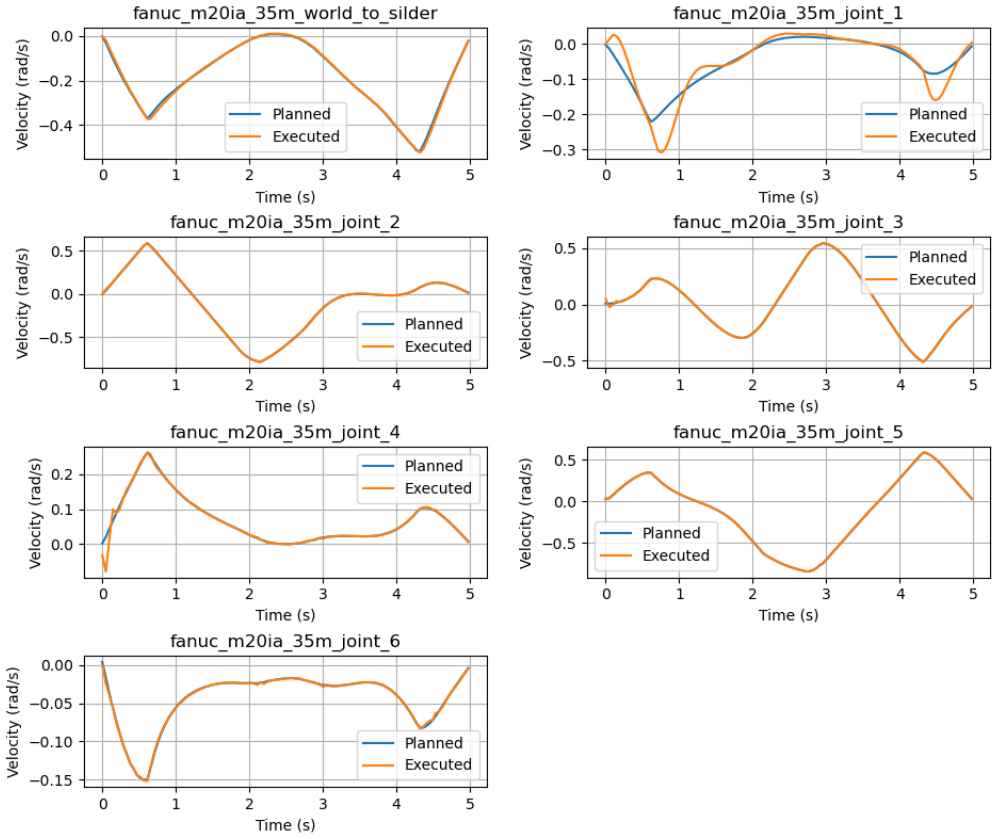


Figure 3.17: Trajectory 5 - Joint velocities

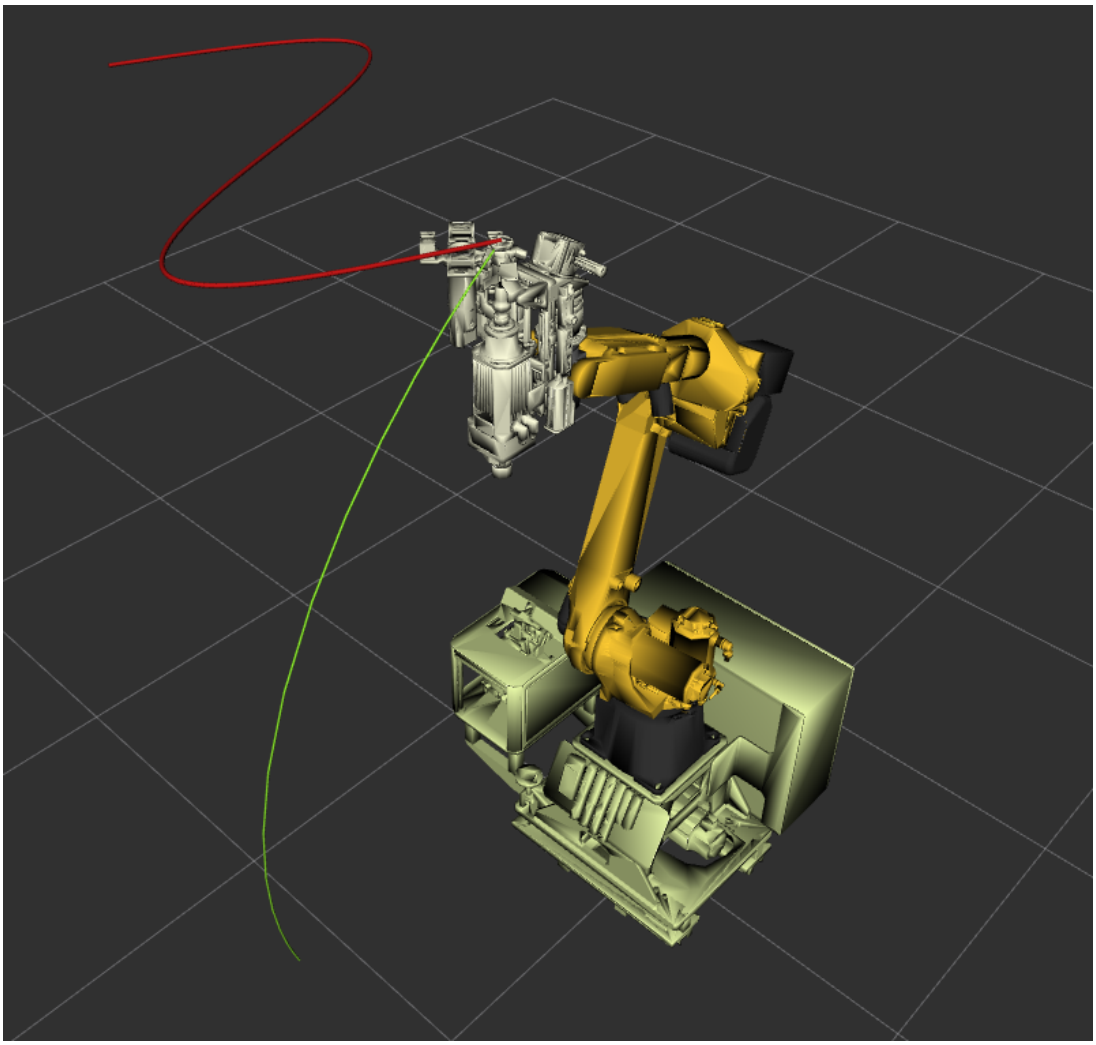


Figure 3.18: Trajectory 6

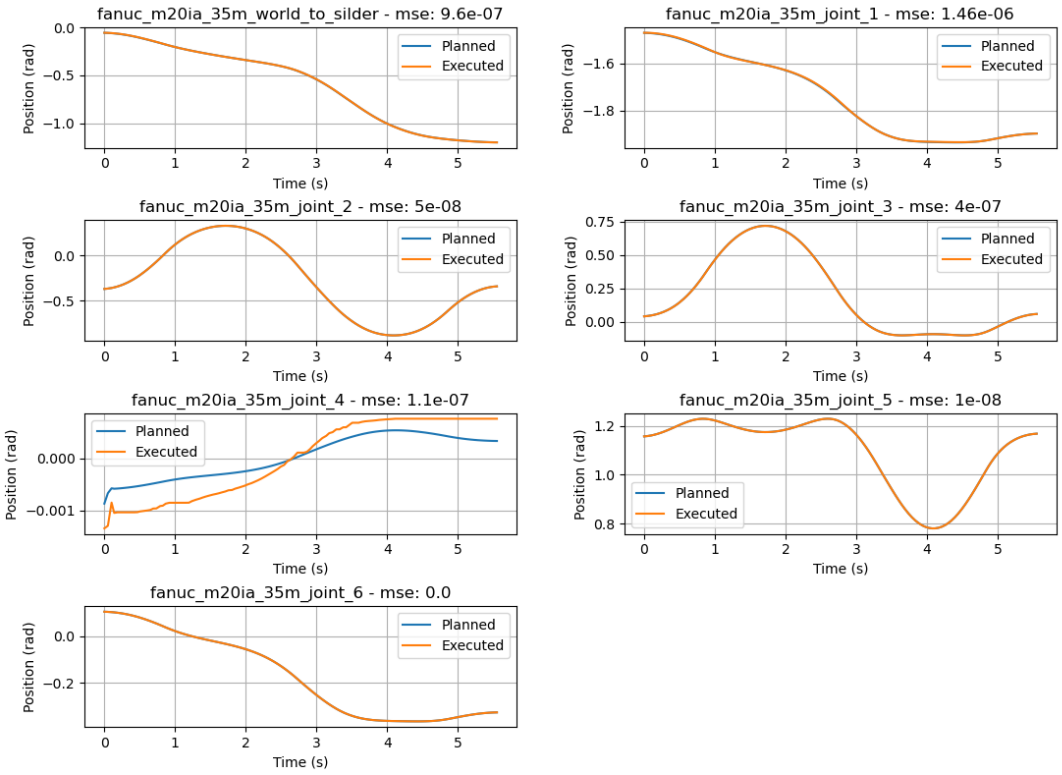


Figure 3.19: Trajectory 6 - Joint positions

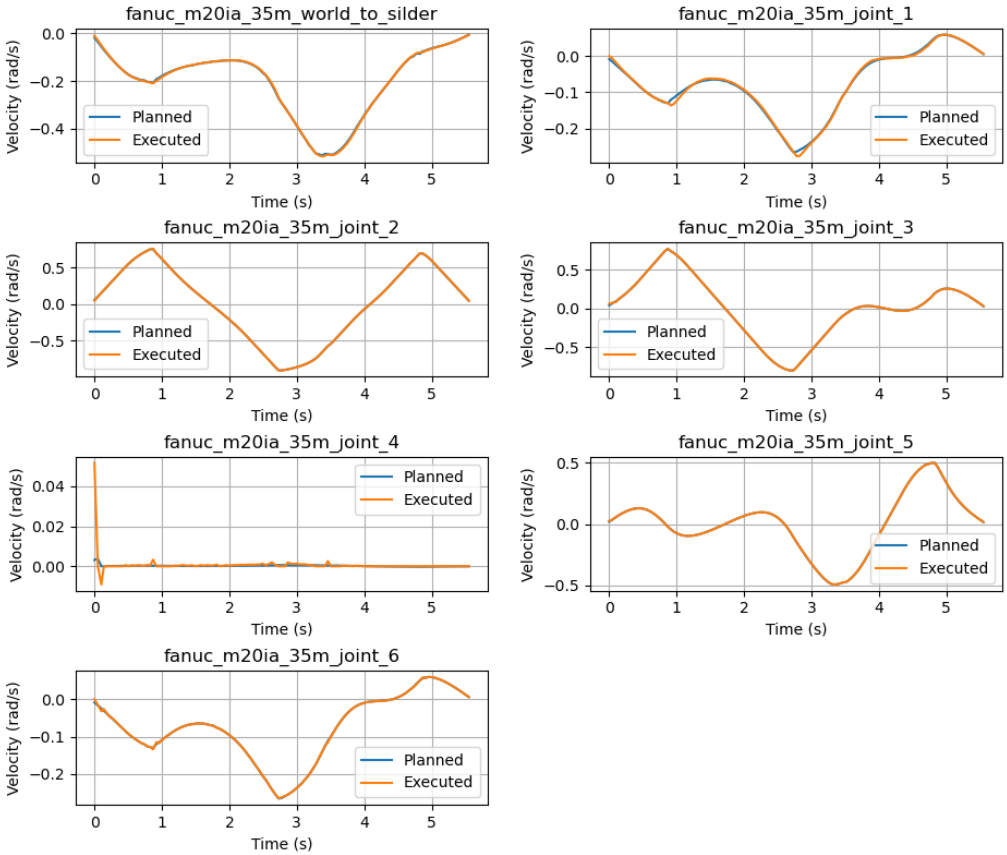


Figure 3.20: Trajectory 6 - Joint velocities

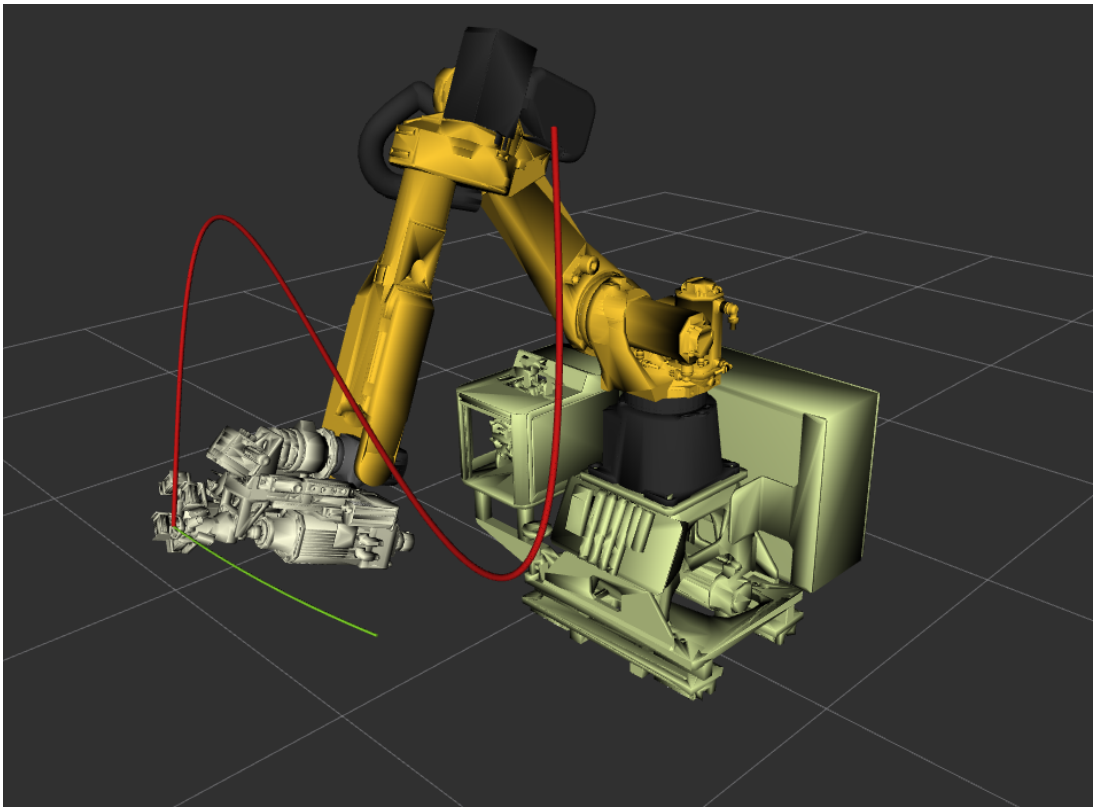


Figure 3.21: Trajectory 7

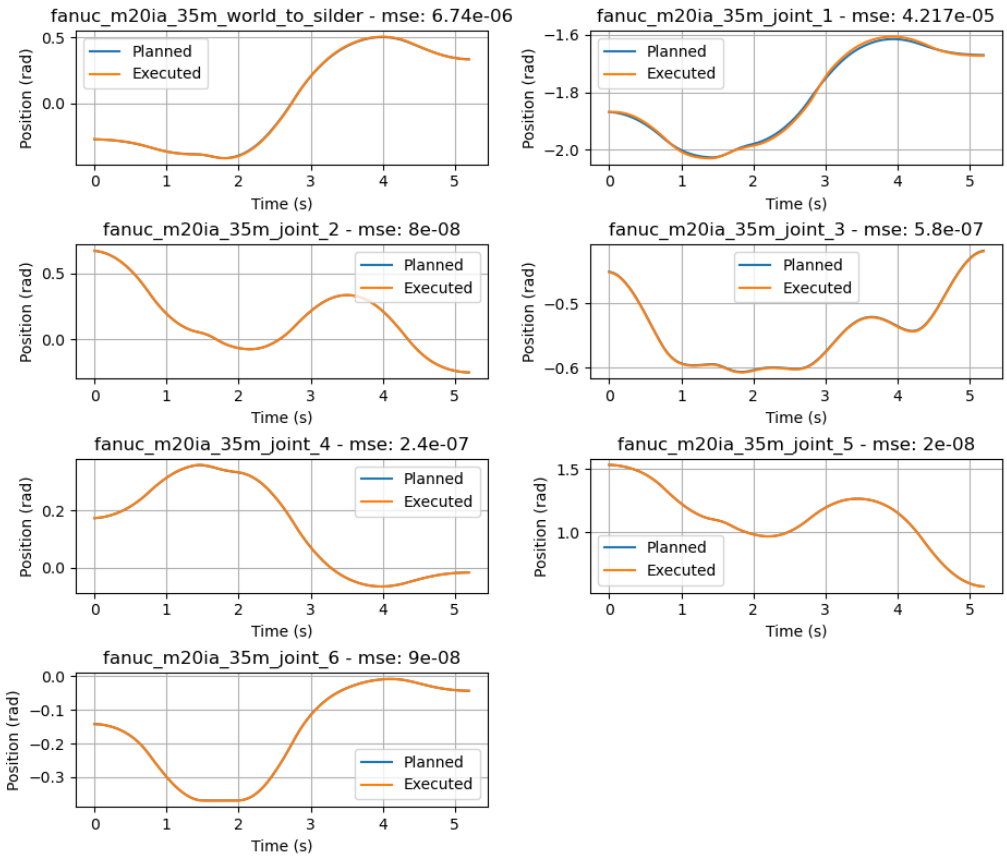


Figure 3.22: Trajectory 7 - Joint positions

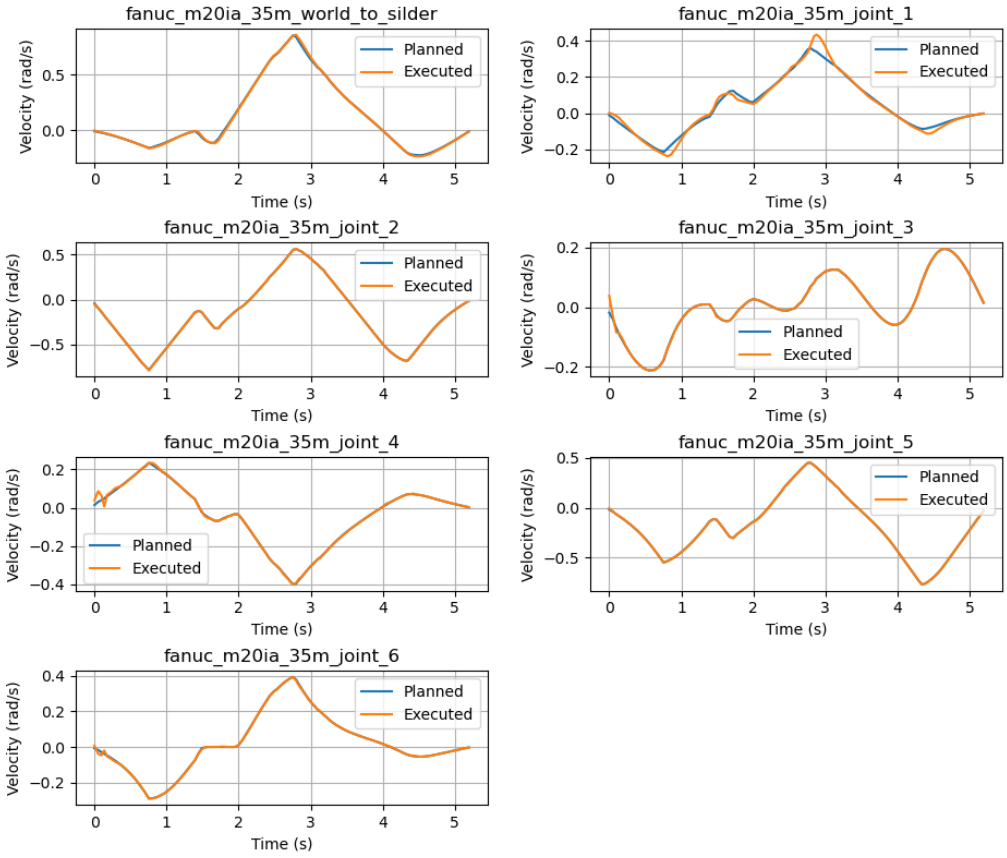


Figure 3.23: Trajectory 7 - Joint velocities

CHAPTER 4

WP4 - CONTROL AND ANALYSIS IN THE TASK SPACE

This chapter is dedicated to the development of the task space controller and all the related extensions to the other modules we developed.

In order to control the robot in task space, the planning module was extended to be able to generate references in the task space. Section 4.1 describes the extension of the planning module. The task space controller was developed to perform task space control. Section 4.2 describes the design choices we made for the controller as well as its development process. After designing the controller, we performed several simulations to test the performance of the system. The trajectory analysis tool we developed is used to analyze the performance of the controllers in terms of trajectory tracking error. Section 4.3 describes how we extended the trajectory analysis tool to be able to analyze the performance of the task space controller. Finally, section 4.4 contains a critical analysis of the results we obtained with the controller.

4.1 Generation of references in the task space

Within this paragraph, the main focus will be on extending the planning module to incorporate the generation of references in the task space.

During this phase, a detailed description of the adopted methodology, design choices, and configuration settings made to implement this functionality will be provided.

To fulfill the function of generating references in the task space, the planning module presented in 2.2.1 has been extended. In particular, as already mentioned, a particularly important role is played by the `controller_type` and `controller_name` parameters as they determine the following operations:

- setting the type of action client that communicates with the certified controllers;
- setting the action sent to the controller's action server;

- the `is_task_space_controller` flag variable in `fanuc_m20ia_35m_planning_demo`, which determines whether the trajectory passed to the `execute_trajectory` utility function is a joint space trajectory or in the task space. As anticipated, this function has been implemented as a template function to ensure flexibility so that it can accept trajectories of different types as input.

Obviously, the type of instantiated action client also affects the callbacks related to sending the action to the action server. If certified controllers are configured to accept joint space references (as opposed to operational space) but, due to incorrect parameter configuration, operational space commands (as opposed to joint space) are sent to the controllers, errors are reported to the user in the CLI, where, furthermore, the status and outcome of the goals sent to the action server are shown.

It is worth dwelling on how references in the task space are generated in order to understand the implementation choices underlying them.

In particular, the generation of references in the task space is based on three key methods of the `planning_node` class, one public, `get_task_space_trajectory`, and two private, `direct_kinematic` and `first_order_kinematic`:

1. The `get_task_space_trajectory` method takes as input a trajectory in joint space (`trajectory`), the robot state (`robot_state`), a logger for log messages (`logger`), the target frame (`target_frame`), the frame ID (`frame_id`), and the planning group (`planning_group`). It uses forward kinematics to compute the end-effector pose for each point in the joint trajectory and first-order kinematics to calculate velocities in the task space. This information is then used to create a `CartesianTrajectory` message representing the trajectory in the task space which will be sent to controller's action server.

2. The `direct_kinematic` function calculates the position and orientation of the end-effector using forward kinematics, given a configuration of joint positions (`positions`) and the target frame (`target_frame`). The forward kinematics is computed due to `RobotState` API.

3. The `first_order_kinematic` function computes linear and angular velocities in the task space using first-order kinematics, given joint velocities (`velocities`) and the planning group (`planning_group`).

In summary, when `is_task_space_controller` is true, there are executed the following steps:

- the plan is computed due to plan method of `move_group_interface`;
- from this plan the planned trajectory is extracted;
- the extracted trajectory is passed as input to the `get_task_space_trajectory` method which returns a `CartesianTrajectory`;
- the `CartesianTrajectory` become the input of `execute_trajectory` function which, in turn, send the task space goal to the proper controller.

4.2 Task Space Controller

In this section we describe the design and development of a `ros2_control` controller for controlling the end-effector pose of a robot in the task space. The designed controller allows either to regulate the pose of the end-effector on a setpoint or to track an entire trajectory specified in the task space.

4.2.1 Controller Overview

The high-level behavior of the `TaskSpaceTrajectoryController` can be described as a Finite State Machine with two states: `IDLE` and `TRACKING_TRAJECTORY`. In the `IDLE` state, the controller maintains the current pose of the end-effector. In the `TRACKING_TRAJECTORY` state, the controller tracks an entire trajectory in the task space. When the controller is activated, it starts from the `IDLE` state and maintains the current pose of the end-effector. When a trajectory is sent to the controller, it switches to the `TRACKING_TRAJECTORY` state and starts tracking the trajectory. When the trajectory execution is completed, the controller returns to the `IDLE` state and maintains the current pose of the end-effector.

While the controller is tracking a trajectory, it is not possible to send a new trajectory to the controller, but it is possible to cancel the current trajectory through the action handler (see 4.2.2). If the target is canceled, the controller returns to the `IDLE` state and maintains the current pose of the end-effector. However, if a new trajectory is sent to the controller while it is tracking a trajectory, the new trajectory will be rejected.

While the controller is in the `IDLE` state, it is possible to send a pose setpoint to the controller through the dedicated topic (see 4.2.2). When the controller recieves the setpoint, it instantly changes the internal setpoint to the new one and moves the robot accordingly. Notice that this is discouraged as it may lead to sudden movements of the robot, and it should be used only for debugging purposes. While the controller is in the `TRACKING_TRAJECTORY` state, the setpoints sent in this way are ignored by the controller.

4.2.2 Usage

Track a trajectory In order to track a trajectory in the task space, the user can send a goal to the action server `~/FollowCartesianTrajectory`. The message type is `cartesian_control_msgs/action/FollowCartesianTrajectory`. The trajectory is validated before the controller accepts the goal. The trajectory is accepted only if:

- the trajectory is not empty;
- when the fist point of the trajectory has a timestamp equal to zero, the euclidean distance between the current pose of the end-effector and the first point of the trajectory is less than the `first_point_tolerance` parameter of the controller;
- the timestamp of each point of the trajectory is greater than the timestamp of the previous point;
- the header frame of the trajectory is equal to the `robot_base_link` parameter of the controller;

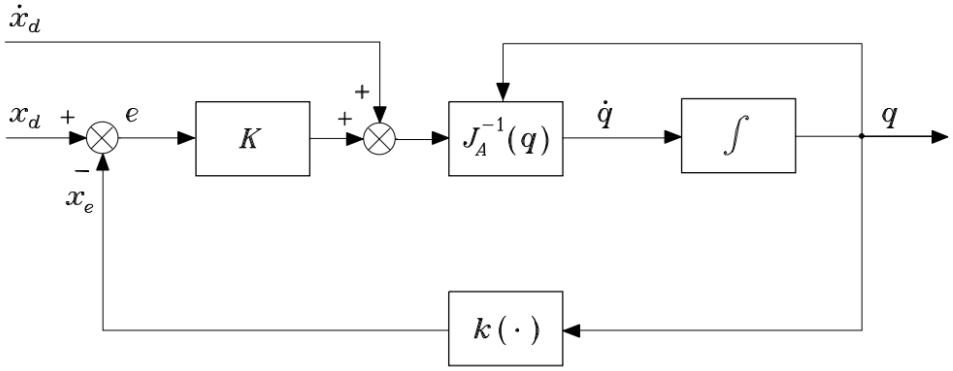


Figure 4.1: Closed loop kinematic inversion algorithm

- the controlled frame of the trajectory is equal to the `end_effector_link` parameter of the controller.

Send an instantaneous setpoint In order to regulate the end-effector pose on a given point, the user can send a reference `moveit_msgs/msg/CartesianTrajectoryPoint` on the `~/command` topic of the controller. Doing this is only recommended for testing purposes as the reference of the controller is instantly changed and this can lead to undesired behaviours when the reference is too far from the current pose of the end-effector. When the controller is in `TRACKING_TRAJECTORY` state, the references sent on the `~/command` topic are ignored.

4.2.3 Trajectory Interpolation

The controller interpolates the given trajectory in the task space. The interpolation is performed dynamically, i.e. the controller computes the trajectory in the task space at each control cycle. By default, a linear interpolation is used. For the orientations, SLERP interpolation in the quaternion space is used.
A better interpolation method would be the quintic spline interpolation that allows to satisfy both the position and velocity constraints at the trajectory waypoints. This is a future improvement that we plan to implement.

4.2.4 Control Law

Formulation To perform control in the task space we implemented the so-called "kinematic control" algorithm. This control law is based on the Closed-Loop Inverse Kinematic algorithm (CLIK), that is a first-order kinematic-inversion algorithm (Fig. 4.1). In other words, CLIK allows to invert a task space trajectory (specified both in terms of pose and velocity) into a joint space trajectory (specified both in terms of joint positions and velocities). However, it can also be used to control a robot in real time as in this case. The control law (generalized to the case of a redundant robot) is given by

$$\dot{\mathbf{q}} = \mathbf{J}^{\dagger}(q)(\mathbf{K}_v\dot{\mathbf{x}}_d + \mathbf{K}\mathbf{e}) + (\mathbf{I}_n - \mathbf{J}^{\dagger}(q)\mathbf{J}(q))\dot{\mathbf{q}}_0 \tag{4.1}$$

where

- $\mathbf{J}$ is the geometric Jacobian matrix of the robot;
- $\mathbf{J}^{\dagger}$ is the Moore-Penrose pseudoinverse of the Jacobian matrix;
- $\dot{\mathbf{x}}_d$ is the desired velocity of the end-effector in the task space. This is a feedforward term that can be used to track a trajectory in the task space;
- $\mathbf{e}$ is the error vector in the task space. This is a feedback term used both to regulate the end-effector pose on a setpoint and to track a trajectory in the task space. The error vector is defined below;
- $\mathbf{K}_v$ is a diagonal matrix of proportional gains for the velocity error;
- $\mathbf{K}$ is a diagonal matrix of proportional gains for the position error;
- $(\mathbf{I}_n - \mathbf{J}^{\dagger}(q)\mathbf{J}(q))$ is the nullspace projector;
- $\dot{\mathbf{q}}_0$ is the nullspace component of the control law.

The error vector $\mathbf{e}$ is defined as a concatenation of the position error $\mathbf{e}_P$ and the orientation error $\mathbf{e}_O$

$$e = \begin{bmatrix} \mathbf{e}_P \\ \mathbf{e}_O \end{bmatrix} \tag{4.2}$$

where the meaning of the position error is clear while the definition of the orientation error depends on the Jacobian matrix used (geometric or analytical). In the following we will discuss why we chose to use the geometric Jacobian matrix, how we calculate the error vector and how we used the nullspace component.

Geometric and Analytical Jacobians The control law in Eq.4.1 is usually formulated in terms of the analytical Jacobian matrix. The only difference between the two Jacobian matrix is in how the end-effector angular velocity is expressed: the analytical Jacobian uses the derivative of a triplet of Euler angles with respect to time $\dot{\phi}_e$, which has no physical meaning, while the geometric Jacobian uses the angular velocity of the end-effector w.r.t. the base frame $\boldsymbol{\omega}_e$, which is a physical quantity. The controller uses the `moveit_core` library to perform the direct and inverse kinematics. The Jacobian matrix calculated by this library is the geometric one. This is not documented and we deduced this by analysing the source code [27]. The orientation component of the pose references that are sent to the controller is expressed in terms of quaternions. This means that, in order to implement the control law, some conversion is needed in order to calculate the orientation error $\mathbf{e}_O$, both if we decide to use the geometric and the analytical Jacobian.
All things considered, in order to implement the control law we had to choose between two main options:

1. Calculate the analytical Jacobian matrix from the geometric Jacobian matrix and express $\mathbf{e}_O$ in terms of Euler angles. This involves choosing a specific representation of the orientation of the end-effector (in terms of a sequence of Euler angles) ϕ_e and then to find the relationship between $\dot{\phi}_e$ and the end-effector angular velocity $\boldsymbol{\omega}_e$. This relationship is

expressed in terms of a rotation matrix $\mathbf{T}(\phi_e)$ which depends on the particular orientation of the end-effector. By inverting this matrix, one can calculate the analytical Jacobian matrix from the geometric Jacobian matrix. This approach is not always possible, as the relationship between $\dot{\phi}_e$ and $\boldsymbol{\omega}_e$ is not always invertible and these situations should be properly handled;

2. Use the geometric Jacobian matrix directly in the control law. This requires either to express the orientation error $\mathbf{e}_O$ in terms of angular velocities or to use the vetor components of the error quaternion $\Delta\mathcal{Q}$ as described below.

We chose the second approach because it is easier to implement and has the advantage of not having to deal with the singularity of the relationship between $\dot{\phi}_e$ and $\boldsymbol{\omega}_e$. This approach has also a clear geometrical interpretation, whereas dealing with Euler angles is confusing and error-prone.

Orientation error As mentioned above, after choosing which Jacobian matrix to use, one should calculate the orientation error $\mathbf{e}_O$ accordingly. We defined the orientation in terms of angular velocity w.r.t. the base frame. In order to calculate it, we first calculate the error quaternion $\Delta\mathcal{Q}$ as

$$\Delta\mathcal{Q} = \mathcal{Q}_d * \mathcal{Q}_e^{-1} \tag{4.3}$$

This quaternion is associated with the rotation matrix $\overline{\mathbf{R}}_d^e = \mathbf{R}_d^0(\mathbf{R}_e^0)^T$ that is the rotation w.r.t. the base frame 0 that brings the end-effector frame from the current orientation to the desired orientation (fixed frame rotation). This rotation is then expressed as a rotation θ around the axis $\hat{\mathbf{r}}$ (the axis-angle representation of the rotation matrix $\overline{\mathbf{R}}_d^e$). Finally, the angular velocity is calculated as a vector proportional to the axis $\hat{\mathbf{r}}$ and to the angle θ as

$$\boldsymbol{\omega}_{error} = k_\theta\, \theta\, \hat{\mathbf{r}} \tag{4.4}$$

where k_θ is a controller gain that can be used to scale the orientation error.
As a side note, we also tried a version of the algorithm where the orientation error is expressed in terms of the vector components of the error quaternion $\Delta\mathcal{Q} = \{\Delta\eta, \Delta\boldsymbol{\epsilon}\}$, i.e. $\mathbf{e}_O = \boldsymbol{\epsilon}$, but we found that this approach is not as stable as the one described above.

Redundancy handling In order to handle the redundancy of the robot, a joint velocity $\dot{\mathbf{q}}_0$ is projected into the null space of the Jacobian matrix. This joint velocity can be used to perform additional tasks, that can be specified as a maximization of a scalar function of the joint positions $w(\mathbf{q})$ by choosing

$$\dot{\mathbf{q}}_0 = k_0 \left(\frac{\delta w(\mathbf{q})}{\delta \mathbf{q}}\right)^T \tag{4.5}$$

The `TaskSpaceTrajectoryController` we implementedd can be configured (via ROS parameters) to use this null space component to perform two different tasks:

1. Maximization of the distance from the joint limits. In this case, the function $w(\mathbf{q})$ is defined as

$$w(\mathbf{q}) = -\frac{1}{2n}\sum_{i=1}^{n}\left(\frac{q_i - \overline{q}_i}{q_{iM} - q_{im}}\right)^2 \tag{4.6}$$

where q_{iM} and q_{im} are the maximum and minimum joint limits and $\overline{q}_i$ is the average of the joint limits;

2. Maximization of the manipulability measure of the robot. In this case, the function $w(\mathbf{q})$ is defined as

$$w(\mathbf{q}) = \sqrt{\det(\mathbf{J}(\mathbf{q})\mathbf{J}^T(\mathbf{q}))} \tag{4.7}$$

In the case of the first task, the gradient of the function $w(\mathbf{q})$ is calculated analytically, while for the second task the gradient is calculated numerically as described in [28]. For more information on how to configure the controller to use the null space component, please refer to the README file of the package `task_space_trajectory_controller`.

4.2.5 Implementation of the control loop

The main steps of the control loop are described below:

1. The joint positions are read from the state interfaces of the robot;
2. The forward kinematics is used to calculate the current position and orientation of the end-effector, as well as the geometric Jacobian matrix in the current configuration;
3. The position error is calculated by subtracting the current position of the end-effector from the desired position;
4. The orientation error quaternion is calculated as in Eq.4.3;
5. The error quaternion is converted to axis-angle representation and the orientation error is calculated;
6. The error vector is composed by the concatenation of position and orientation errors;
7. Control law described in Eq.4.1 is used to calculate the control inputs $\dot{q}$;
8. If the null space control is enabled, the null space component is calculated as specified in section 4.2.4 and added to the control inputs;
9. The control inputs $\dot{q}$ are written to the command interfaces of the robot.

4.2.6 Limitations and future improvements

The main limitiation of the chosen control law is that it is not collision-aware. This means that the controller expects the robot to move in free space and wihout self-collisions. This is a common limitation of kinematic control algorithms. The main implication of this is that the controller could not manage to follow a trajectory when the end effector is in collision with the manipulator.
Even if the most natural way to overcome it is to chose a different control law, we tried to exploit the null space component to avoid self-collisions. In order to do this we implemented a version of the control law that maximizes a function $w(\mathbf{q})$ which is the minimum distance between two colliding bodies. This metric was evaluated with a collision checking algorithm provided by the

`moveit_core` package. The problem of this approach is that, as we are not able to calculate the gradient of the function $w(\mathbf{q})$ analytically, we have to calculate it numerically. This is a very expensive operation as it requires to evaluate the function $w(\mathbf{q})$ at least $2n$ times, where n is the number of joints of the robot. This led to a very slow simulation which would make the controller unusable in a real-time scenario, so we decided to abandon this approach. As a side note, we tried both `FCL` and `Bullet` collision detectors obtaining the same performances.
Another problem that is common to all kinematic control algorithms is that the controller could impose high joint velocities to the robot when the robot is close to a singularity. We tried to overcome this problem by using the null space to maximize the manipulability measure of the robot, however in order to keep the controller stable we had to use a very low gain for the null space component. This leds to a very slow convergence of the nullspace component (but it does not affect the tracking of the trajectory in the task space). As a consequence, when using the task space controller **one should wait for the robot to stand still** before sending a new trajectory to the controller. Doing so minimizes the risk of the robot to be close to a singularity and allows for better tracking performances.
Finally, at the moment the controller is only capable of tracking which are expressed in the robot's base frame. However, this limitation would be easy to overcome by calculating the Jacobian matrix in a different frame. This is a future improvement that we plan to implement.

4.3 Extension of the tracking analysis tool

The trajectory analysis tool implemented in Chapter 3 has been extended to allow the analysis of trajectories in the task space. In particular, with the optional parameter `task-space` it is possible to enable the analysis of trajectories in the task space. The messages contained in the bag file, whatever the trajectory type as long as it is in the task space, must be: `cartesian_control_msgs/action/FollowCartesianTrajectory_FeedbackMessage`. This type of message contains all the information we need: X, Y and Z for the position, and the quaternion for the orientation, the linear and angular velocity, all of them both for the executed and planned trajectory. The tool reads the bag file, extracts the messages, and then for each joint is plotted:

- Executed vs planned trajectory with orientation;
- Executed vs planned velocities;
- MSE between executed and planned trajectory;
- 3D plot of the executed and planned trajectory.

In Fig. 4.2 an example of the 3D plot of the executed and planned trajectory is shown. The tool is still robot agnostic, it can be used for any robot that uses the `FollowCartesianTrajectory` action, and it is also trajectory agnostic, it can be used for any trajectory that is in the task space, as long as the bag file contains the correct messages.

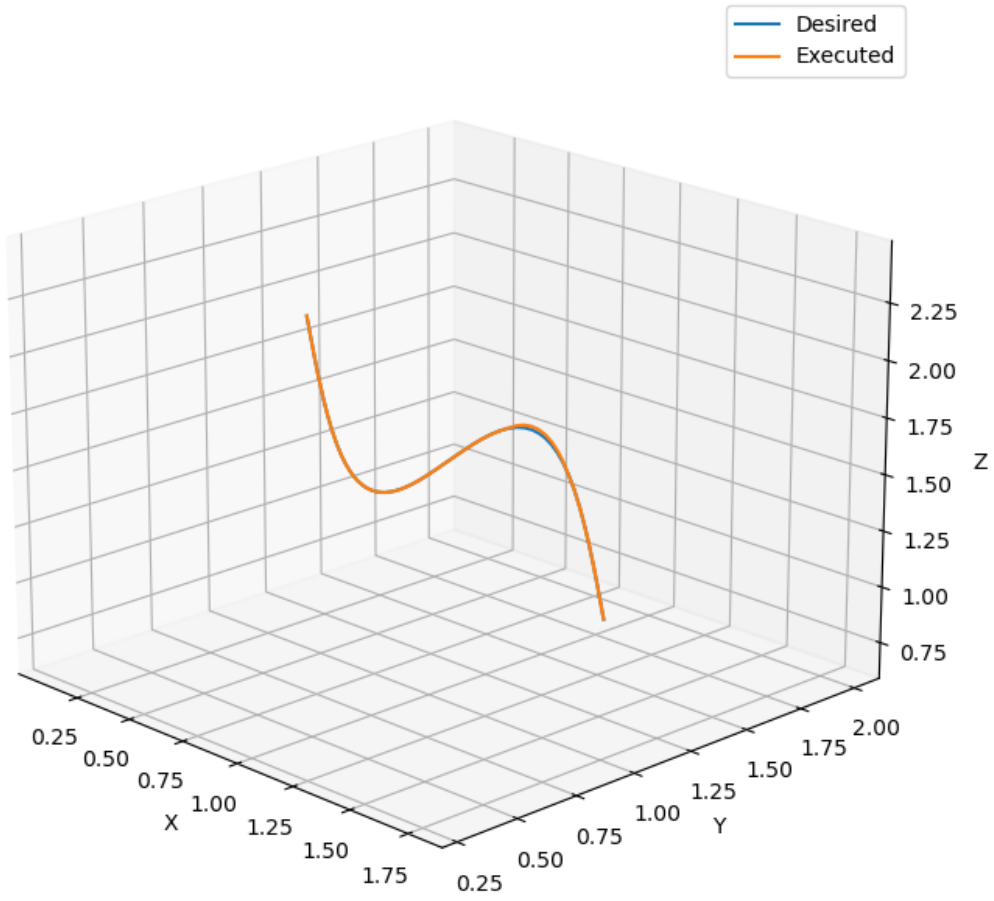


Figure 4.2: 3D plot of the executed and planned trajectory

4.4 Tracking performance analysis

Just as in WP3 also downstream of this chapter, after implementing planning and control in the task space, the analysis of the same pathways seen in the section 3.5 is carried out. The analyzed trajectory files, are located in the folder `resource\executed_task_space_trajectories` of the package `acg_resource_tools`.

4.4.1 Trajectory 1

Just as in joint space, in task space this trajectory is executed almost perfectly, both in position and orientation with an MSE equals to zero, as can be seen in Fig. 4.3. The velocity references (Fig. 4.4) are also tracked very well, and note that the actual linear velocities are stepped because of the linear interpolation done in the controller, this is explained in detail in the section 4.2.3

4.4.2 Trajectory 2

Analyzing the execution of trajectory 2 (Fig. 4.5) it can be seen that the position references are tracked perfectly both in position and orientation, despite the fact that the trajectory is

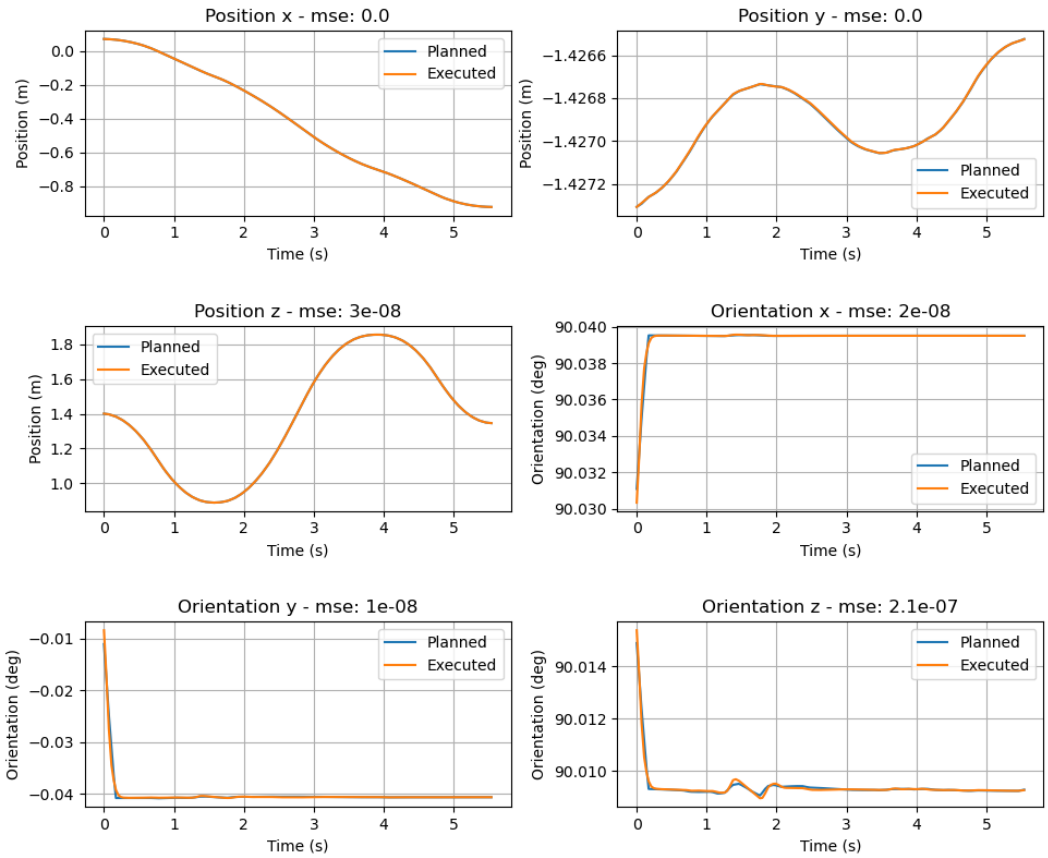


Figure 4.3: Task space trajectory 1 positions analysis

inclined with respect to the axis along which the slide is moving. The velocity references are also tracked well (Fig. 4.6).

4.4.3 Trajectory 3

Analyzing trajectory 3 (Fig. 4.7) shows that the robot manages to follow both position and orientation references perfectly, with virtually no error, with near-perfect tracking of speeds as well (Fig. 4.8).

4.4.4 Trajectory 4

As for trajectory 4, again the tracking of position and orientation is perfect (Fig. 4.9), while it can be seen that the velocities are also with perfect tracking with the usual problem of interpolation mentioned earlier (Fig. 4.10).

4.4.5 Trajectory 5

For trajectory 5 (Fig. 4.11) it is observed that the position references are tracked quite well, although with some inaccuracies, while for the orientation references they are not tracked at all well, with considerable error, again due to the control law and self-collisions, in the range 2-4 seconds, where we observe the effect of this self-collision and how it is also reflected in the

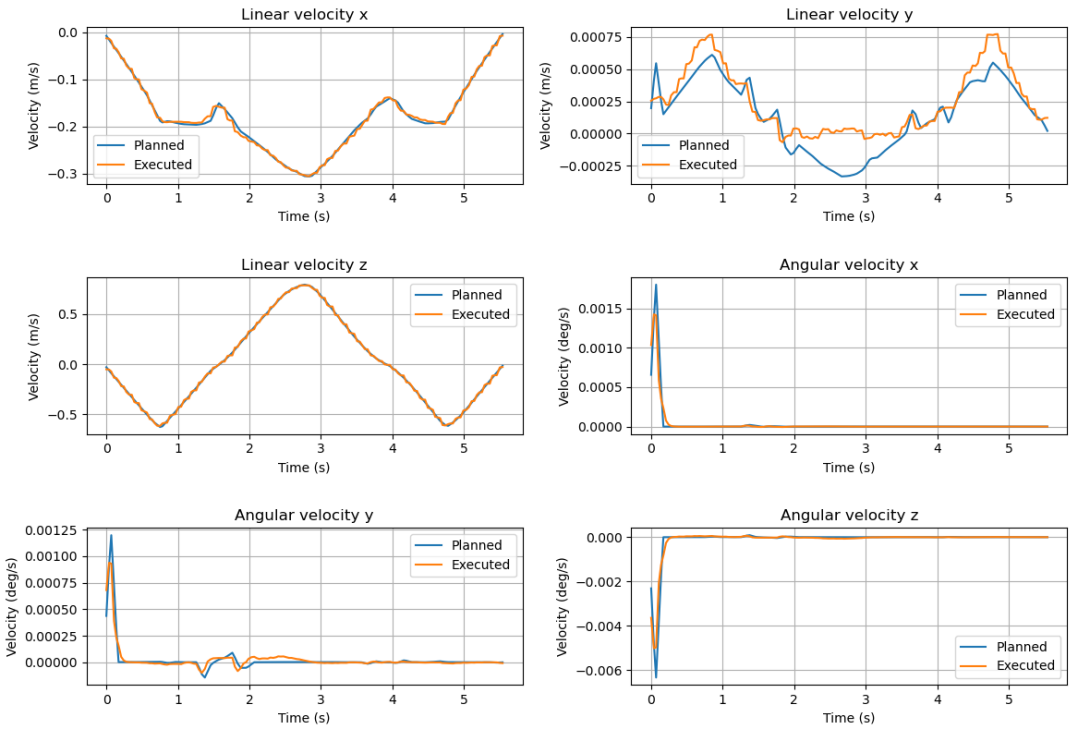


Figure 4.4: Task space trajectory 1 velocities analysis

velocities, where we observe a sudden change in value in the same time range (Fig. 4.12). This problem is explained in detail in the section 4.2.6.

4.4.6 Trajectory 6

Trajectory 6 is executed perfectly by the robot in both positions and orientations with virtually no MSE. This is shown in Fig. 4.13. And the velocities are also tracked perfectly (Fig. 4.14).

4.4.7 Trajectory 7

Analyzing trajectory 7 (Fig. 4.15), there is no error on the positions, but significant error on the orientations. In this case the errors emerge not only because of a self-collision, but also because the desired trajectory, being inclined with respect to the horizontal plane, has a particular complexity of tracking. This is reflected in the velocities where a rapid change in value is observed when the robot goes into self-collision. This is shown in Fig. 4.16.

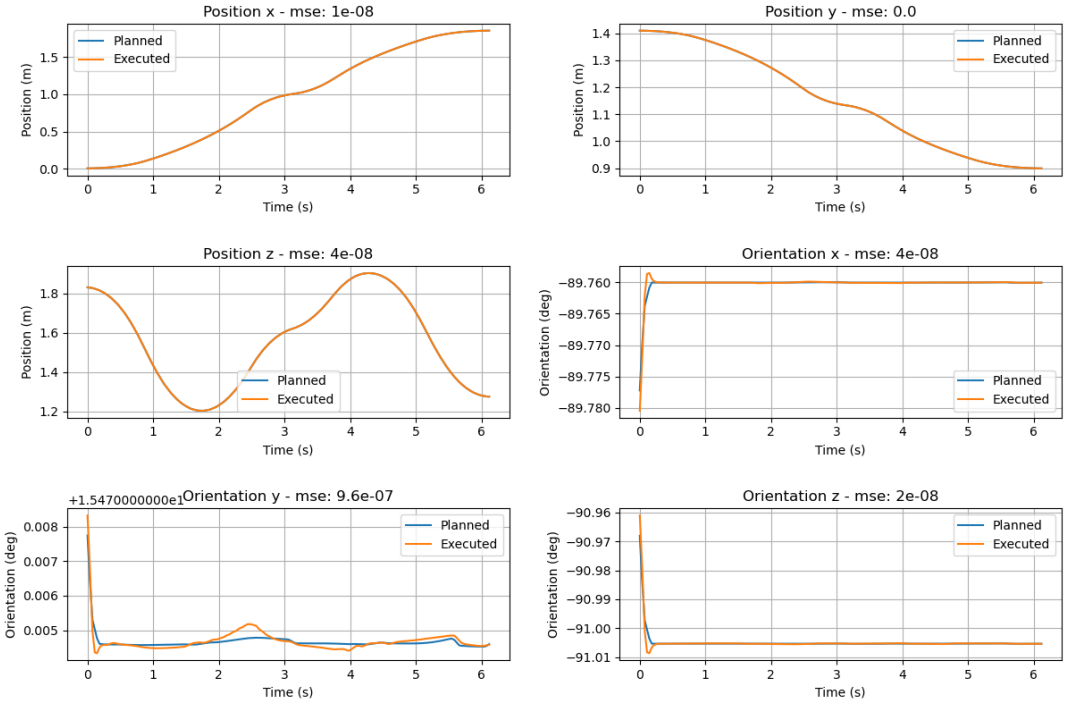


Figure 4.5: Task space trajectory 2 positions analysis

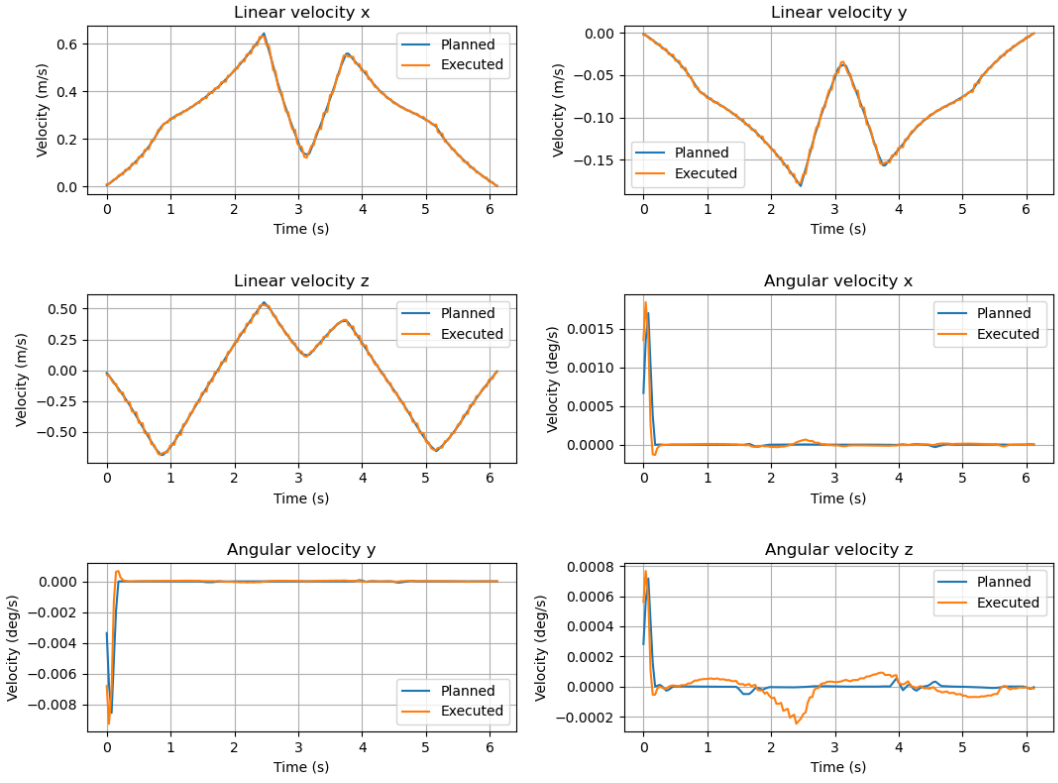


Figure 4.6: Task space trajectory 2 velocities analysis

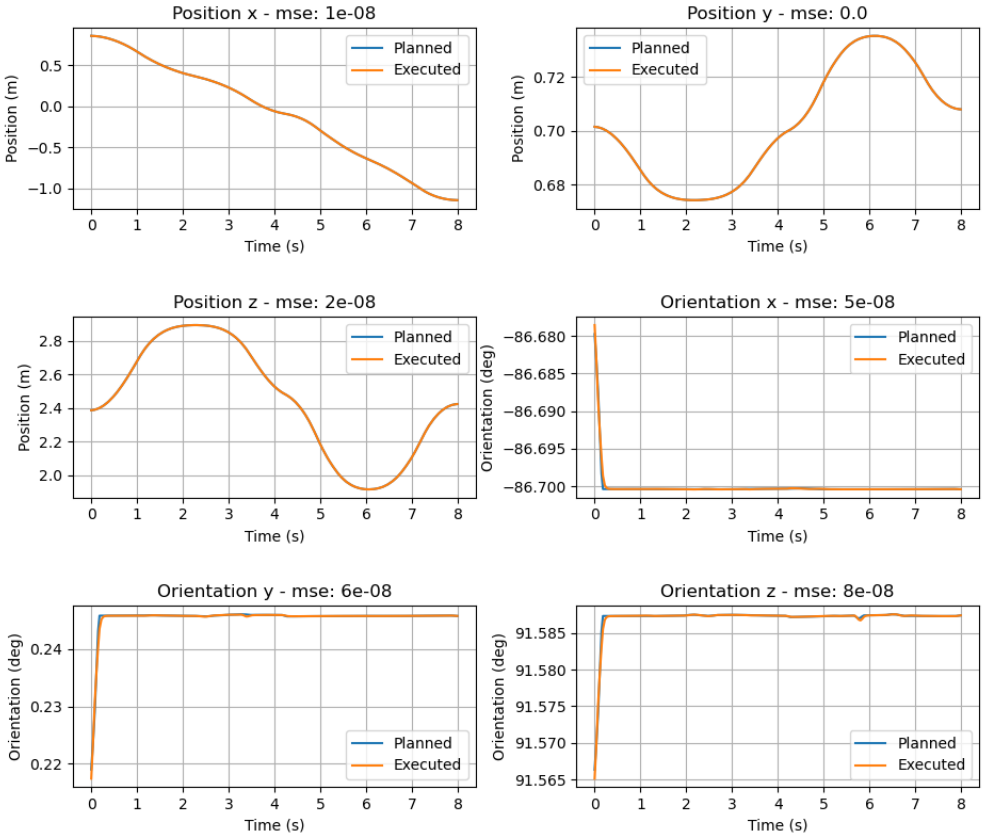


Figure 4.7: Task space trajectory 3 positions analysis

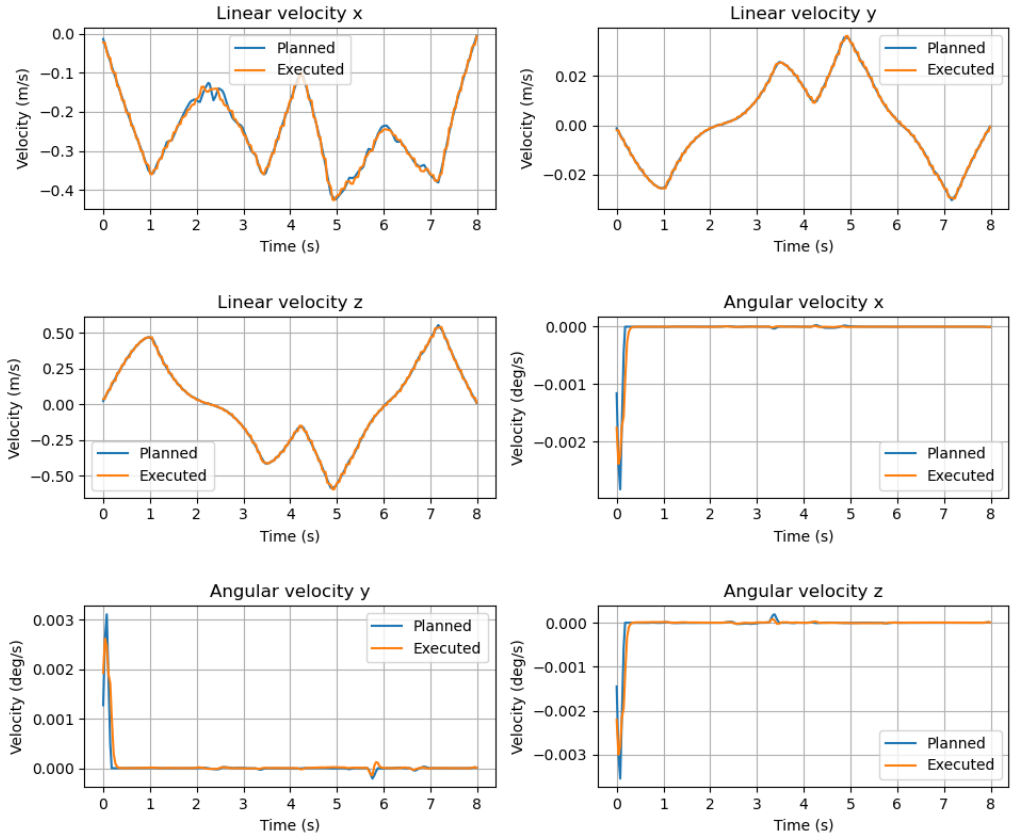


Figure 4.8: Task space trajectory 3 velocities analysis

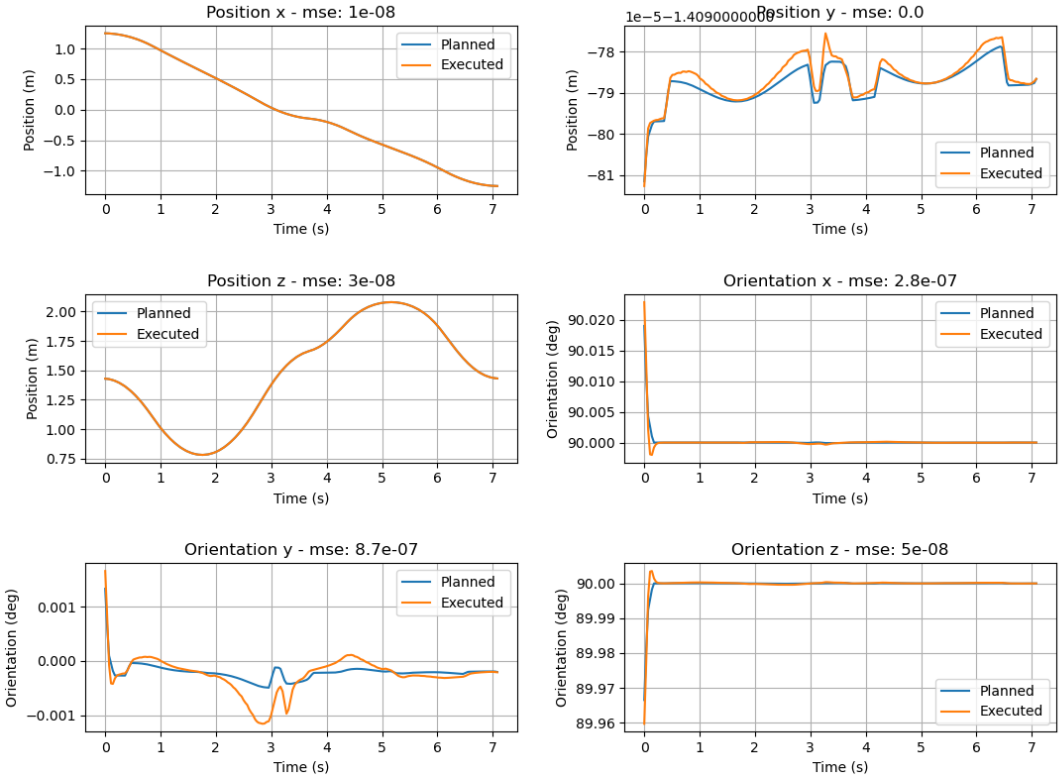


Figure 4.9: Task space trajectory 4 positions analysis

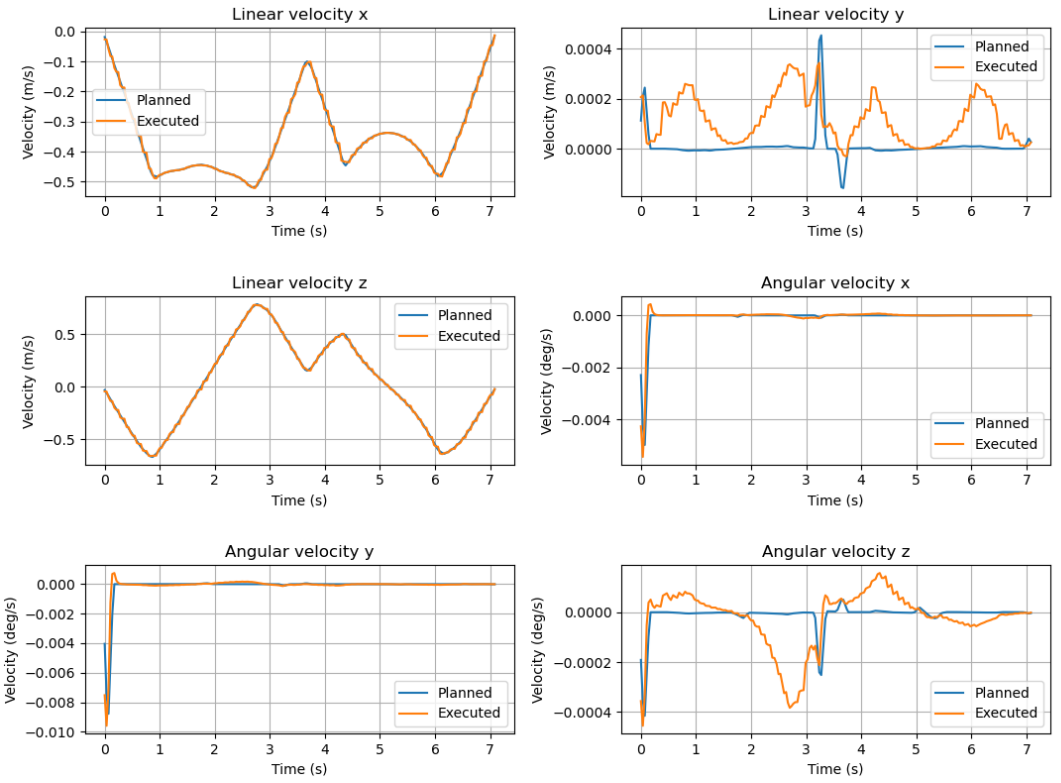


Figure 4.10: Task space trajectory 4 velocities analysis

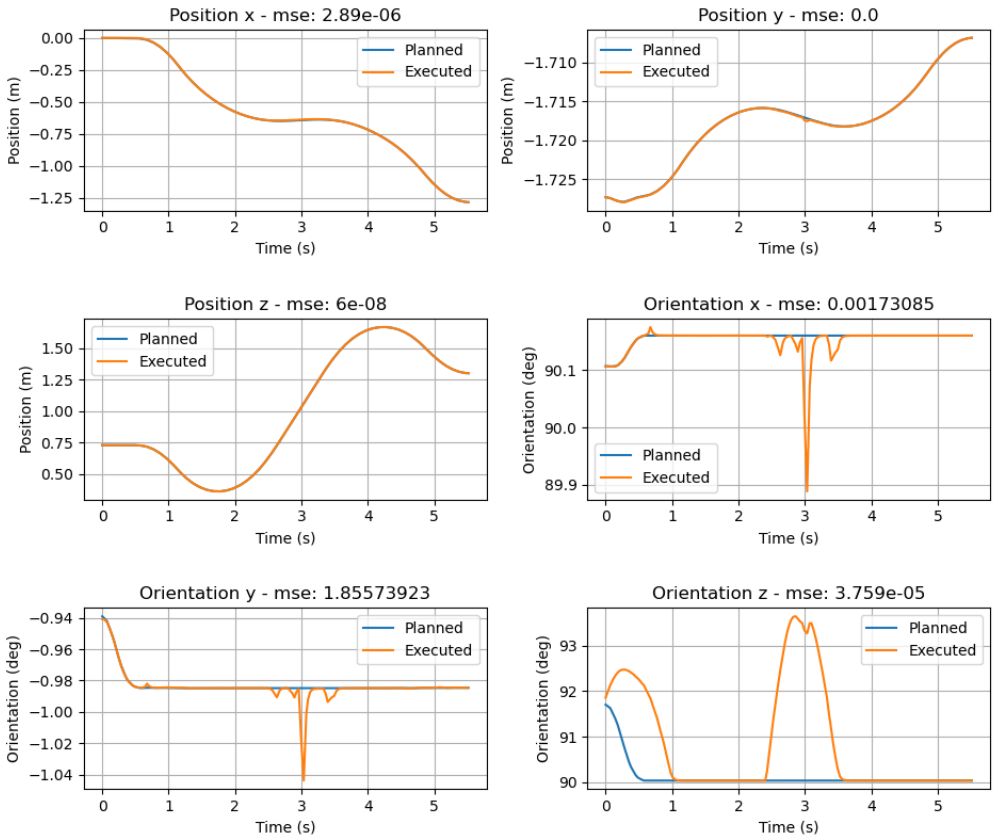


Figure 4.11: Task space trajectory 5 positions analysis

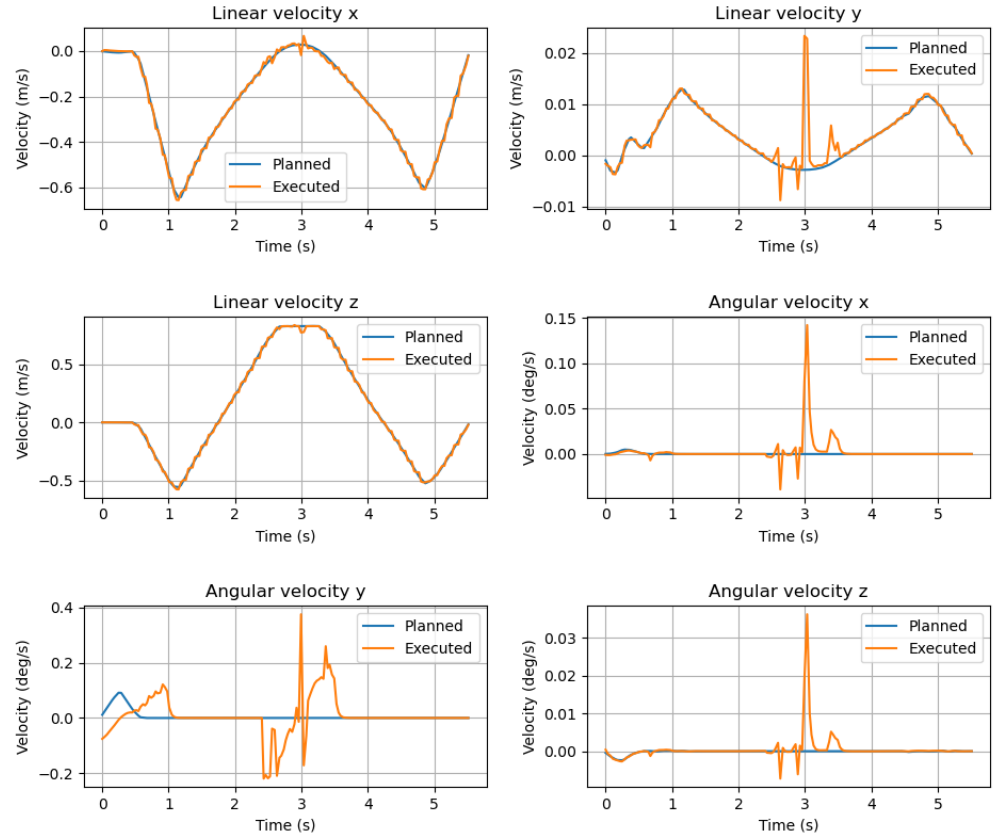


Figure 4.12: Task space trajectory 5 velocities analysis

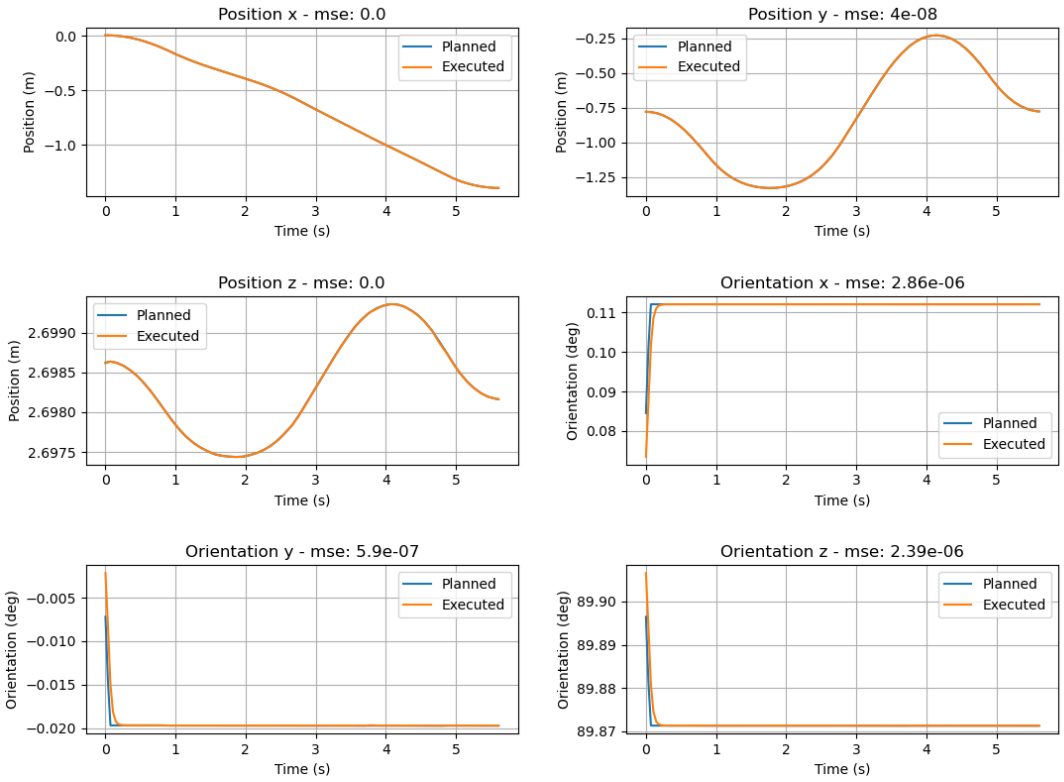


Figure 4.13: Task space trajectory 6 positions analysis

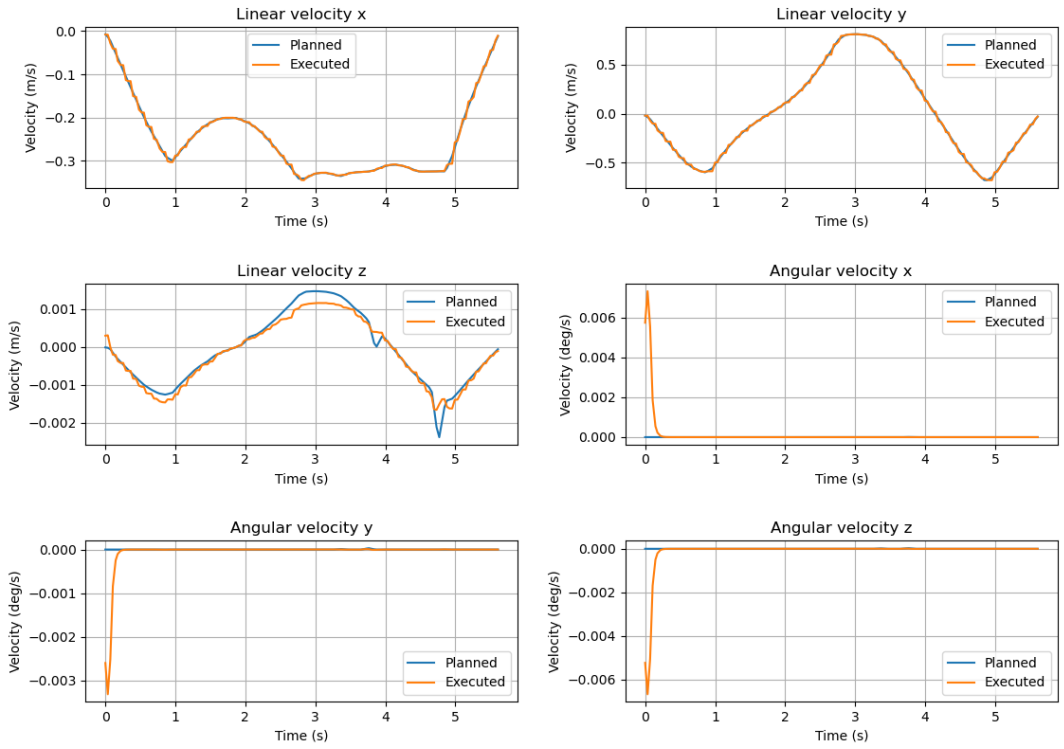


Figure 4.14: Task space trajectory 6 velocities analysis

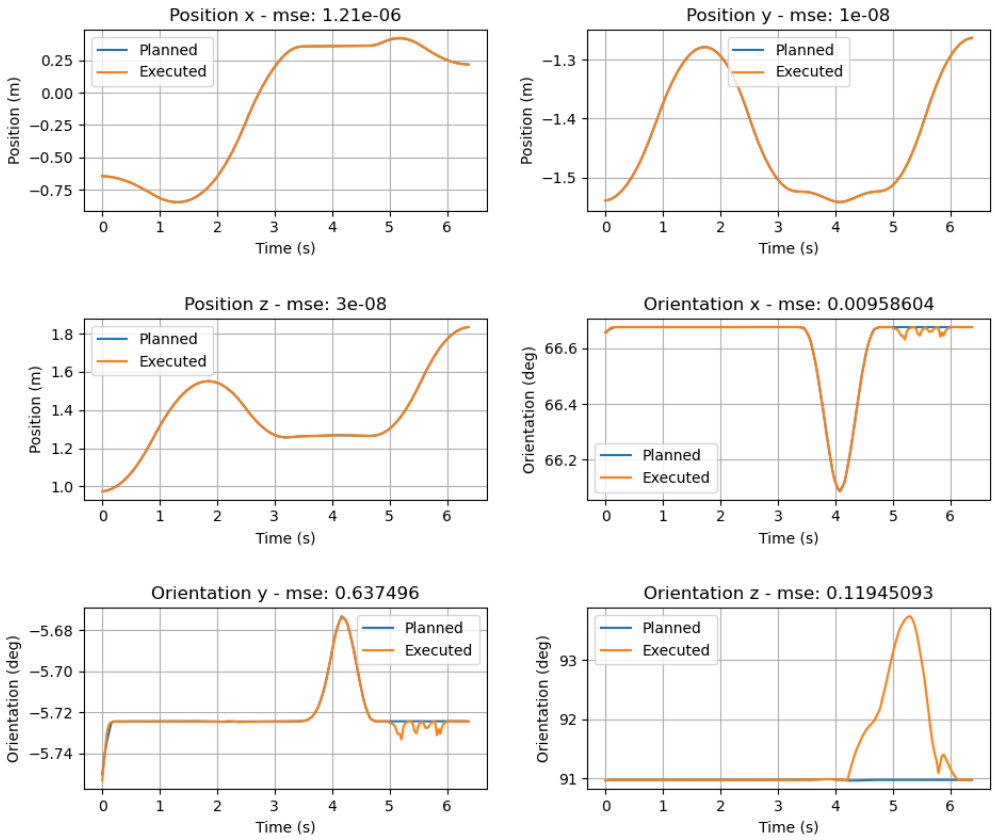


Figure 4.15: Task space trajectory 7 positions analysis

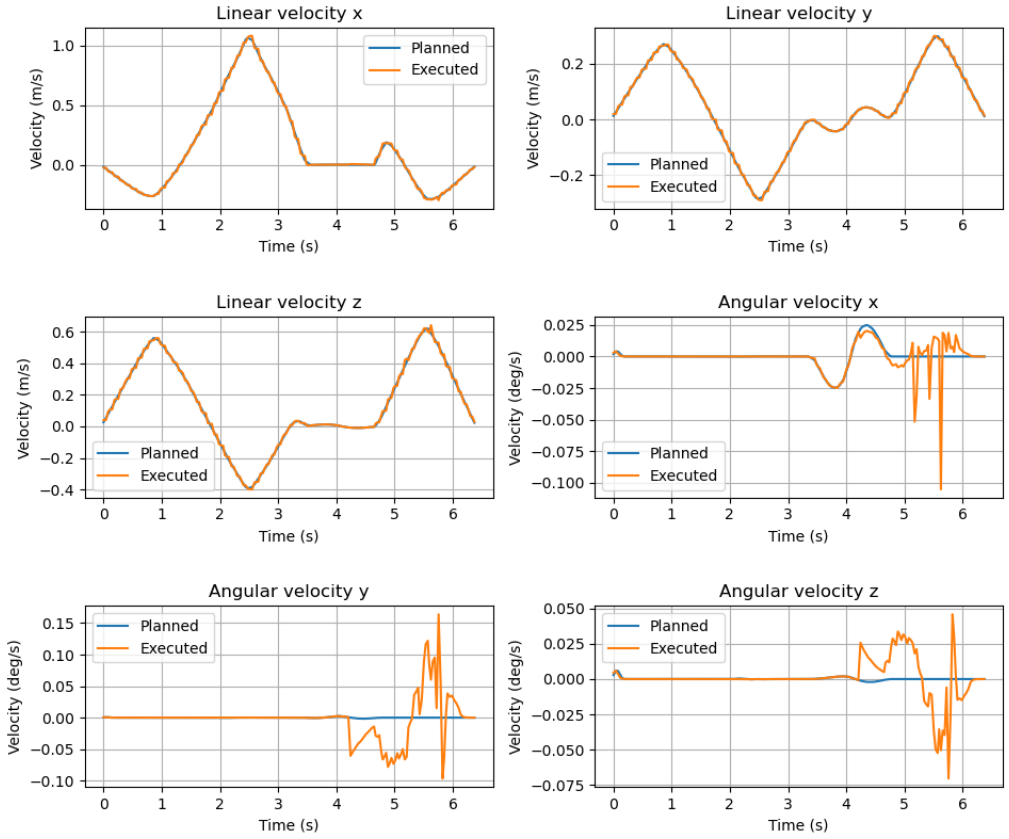


Figure 4.16: Task space trajectory 7 velocities analysis

FINAL CONSIDERATIONS

In this project, we explored the capabilities and challenges associated with the implementation and programming of the Fanuc M-20iA/35M robot. Through a combination of theoretical research, simulations, and practical experimentation, we were able to gain a deeper understanding of this robot's capabilities.
During the initial phase of the project, we thoroughly analyzed the technical specifications of the manipulator, identifying its key features including load capacity, speed, joint masses, etc. This understanding provided us with a solid foundation to plan and develop our programming and control activities. Subsequently, we explored various planners and solvers to adapt trajectory planning to the specific needs of our case study.
During the phases of practical experimentation, we encountered several challenges related to configuring and optimizing the robot's trajectories, aiming to strike a balance between correctness, trajectory smoothness, and performances. We also examined motion control strategies with the goal of achieving a sufficient level of operational efficiency for the robot.
The results obtained during this project were deemed satisfactory for most general cases. It is advisable to position the desired path in the task space in such a way that the robot can utilize the degree of freedom provided by the slide while being in the most "comfortable" position as this significantly facilitates the proper functioning of the planning module. Such a trajectory is shown in Fig. 4.17. However, we have also identified some areas where the robot could be further improved. For instance, we encountered limitations in handling more unusual situations. For example

- Trajectories parallel to the surface on which the slide lies (above or below the robot);
- Trajectories for which it is not possible to take advantage of the degree of freedom provided by the slide;
- Trajectories mirrored with respect to those shown in the figure. In these cases, the planning module sometimes plans a joint space trajectory in which the elbow of the manipulator is pointing downwards and this may cause issues as the robot collides with the slide.

Most of the issues we encountered both in planning and control are ultimately caused by the presence of the end-effector: the end effector greatly reduces the dexterity of the robot, and

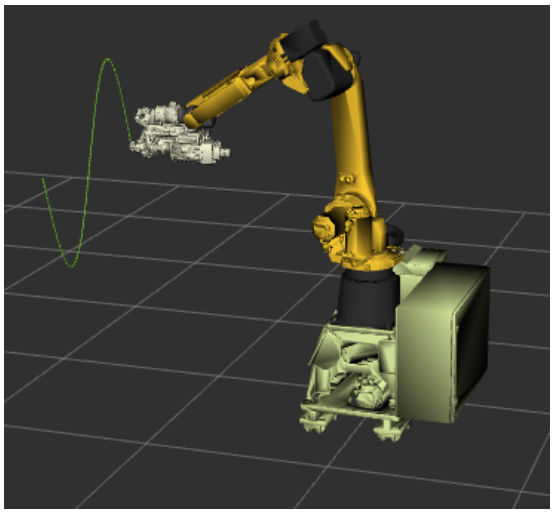


Figure 4.17: Sample trajectory

the presence of the slide, while providing an additional degree of freedom, is not enough to compensate for this loss of dexterity.
For the Cartesian path planning, the `computeCartesianPath` API we employed forced us to constaint all of the 3 degrees of freedom for the end-effector orientation. However, the task specified could be accomplished by constraining the orientation along only two axes and leaving the third axis free. In such a scenario, we could have benefited from an additional degree of freedom (rotations along the end-effector's $z - axis$). This would have lead to an increase of the dimension of the null space which in turn would have simplified the self-collision avoidance for the chosen planner. For this reason, one of the most important improvements that could be made to the proposed system is the usage of a modern planner for the cartesian path. According to PickNik Robotics, the default Cartesian planner in MoveIt is obsolete and should be avoided [29] as it is greedy and easily gets stuck in local minima. Moreover, it does not provide any functionality for avoiding joint limits or restarting. A planner that could handle such a scenario is `OMPL Constrained Planning` [30] but it would require custom interpolation of the cartesian path in task space.
Another possible development is to consider the presence of obstacles in the workspace.
In conclusion, this project has provided us with a comprehensive overview of the capabilities and challenges associated with the field of robotics. We are confident that the knowledge and skills acquired during this experience will be valuable in tackling future challenges in the field of robotics and industrial automation.

Ouroboros logo ~~designed~~ stolen by Group 4

LIST OF FIGURES

BIBLIOGRAPHY

[1] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[2] J. Denavit and R. S. Hartenberg. "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices". In: *Journal of Applied Mechanics* 22.2 (June 2021), pp. 215–221. ISSN: 0021-8936. DOI: 10.1115/1.4011045. eprint: https://asmedigitalcollection.asme.org/appliedmechanics/article-abstract/22/2/215/1110292/A-Kinematic-Notation-for-Lower-Pair-Mechanisms. URL: https://doi.org/10.1115/1.4011045.

[3] Migatronic. *Fanuc M-20iA Operator's Manual*. 2013. URL: https://www.migatronic.com/media/1384/manual_am-120ic_operator_manual_b-82874en_07.pdf.

[4] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: http://www.blender.org.

[5] Fanuc. *Fanuc AC Servo Motor Alpha-is/Alpha-i Descriptions*. URL: https://www.nexinstrument.com/assets/images/pdf/A06B-0272-B605%20(1).pdf.

[6] *Fanuc M-20iA Datasheet 2*. URL: https://www.fanucamerica.com/cmsmedia/datasheets/M-20iA_35M%20Product%20Information_299.pdf.

[7] Federica Storiale, Enrico Ferrentino, and Pasquale Chiacchio. "Planning of efficient trajectories in robotized assembly of aerostructures exploiting kinematic redundancy". In: *Manufacturing Review* 8 (2021), p. 8. ISSN: 2265-4224. DOI: 10.1051/mfreview/2021007. URL: http://dx.doi.org/10.1051/mfreview/2021007.

[8] *Fanuc M-20iA Datasheet 1*. URL: https://www.flexibleassembly.com/core/media/media.nl?id=641295&c=337772&h=kwPV47SLAnLms8Xe_rrYHL3Z2zAp04Hlnjh9sIPQAXYunCsm.

[9] Dassault Systèmes. *SolidWorks*. Version SolidWorks 2023 SP5. Sept. 29, 2022. URL: https://www.solidworks.com/.

[10] *MeshInspector*. 2024. URL: https://meshinspector.com/.

[11] Paolo Cignoni, Alessandro Muntoni, Guido Ranzuglia, Marco Callieri. *MeshLab*. DOI: 10.5281/zenodo.5114037.

[12] Paolo Cignoni et al. "MeshLab: an Open-Source Mesh Processing Tool". In: *Eurographics Italian Chapter Conference*. Ed. by Vittorio Scarano, Rosario De Chiara, and Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[13] *Phobos GitHub Repository*. https://github.com/dfki-ric/phobos. Version 2.0.0. Sept. 27, 2023.

[14] Kai von Szadkowski and Simon Reichel. "Phobos: A tool for creating complex robot models". In: *Journal of Open Source Software* 5.45 (2020), p. 1326.

[15] Patrick Beeson and Barrett Ames. "TRAC-IK: An open-source library for improved solving of generic inverse kinematics". In: *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*. 2015, pp. 928–935. DOI: 10.1109/HUMANOIDS.2015.7363472.

[16] Shuxin Xie et al. "A speedup method for solving the inverse kinematics problem of robotic manipulators". In: *International Journal of Advanced Robotic Systems* 19.3 (2022), p. 17298806221104602. DOI: 10.1177/17298806221104602. eprint: https://doi.org/10.1177/17298806221104602. URL: https://doi.org/10.1177/17298806221104602.

[17] *TracIK*. URL: https://bitbucket.org/traclabs/trac_ik/src/master/.

[18] *TracIK ROS2 porting*. URL: https://github.com/aprotyas/trac_ik/tree/ros2/trac_ik_kinematics_plugin.

[19] *approximate solutions from OMPL should be handled as failure*. https://github.com/ros-planning/moveit/issues/1920 [Accessed: 12/01/2024]. 2020.

[20] *OMPL Issue*. https://github.com/ompl/ompl/issues/868 [Accessed: 12/01/2024]. 2020.

[21] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. "The Open Motion Planning Library". In: *IEEE Robotics & Automation Magazine* 19.4 (2012). https://ompl.kavrakilab.org, pp. 72–82. DOI: 10.1109/MRA.2012.2205651.

[22] Jonathan Meijer, Qujiang Lei, and Martijn Wisse. "An empirical study of single-query motion planning for grasp execution". In: July 2017, pp. 1234–1241. DOI: 10.1109/AIM.2017.8014187.

[23] Mehdi Shahabi, Hashem Ghariblu, and Manuel Beschi. "Comparison of different sample-based motion planning methods in redundant robotic manipulators". In: *Robotica* 40.9 (2022), 3104–3119. DOI: 10.1017/S026357472200008X.

[24] Nathan Ratliff et al. "CHOMP: Gradient optimization techniques for efficient motion planning". In: *2009 IEEE International Conference on Robotics and Automation*. 2009, pp. 489–494. DOI: 10.1109/ROBOT.2009.5152817.

[25] Francesco Grothe et al. *ST-RRT*: Asymptotically-Optimal Bidirectional Motion Planning through Space-Time*. 2022. arXiv: 2203.02176 [cs.RO].

[26] Kiam Heong Ang, G. Chong, and Yun Li. "PID control system analysis, design, and technology". In: *IEEE Transactions on Control Systems Technology* 13.4 (2005), pp. 559–576. DOI: 10.1109/TCST.2005.847331.

[27] PickNik Inc. *moveit_core getJacobian implementation*. Version 2.5.5. 2024. URL: https://github.com/ros-planning/moveit2/blob/main/moveit_core/robot_state/src/robot_state.cpp#L1476 (visited on 10/02/2024).

[28] Kévin Dufour and Wael Suleiman. "On Maximizing Manipulability Index while Solving a Kinematics Task". In: *Journal of Intelligent & Robotic Systems* 100 (Oct. 2020). DOI: 10.1007/s10846-020-01171-7.

[29] Dr. Mark Moll Dr. Dave Coleman. *Guide to cartesian planners in moveit*. 2021. URL: https://picknik.ai/cartesian%20planners/moveit/motion%20planning/2021/01/07/guide-to-cartesian-planners-in-moveit.html.

[30] Mark Moll Ioan A. Şucan. *OMPL Constrained Planning*. URL: https://ompl.kavrakilab.org/constrainedPlanning.html.