UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Robotics - Project

Nome	Matricola	Email
Beniamino Squitieri	062272021	b.squitieri@studenti.unisa.it

L.M. INGEGNERIA INFORMATICA - A.A. 2023/2024

Contents

1 Introduction and objective

The project focuses on implementing task space impedance control, a crucial technique for enabling safe and effective interactions between robots and their environments, which can be static, dynamic, or include humans. Impedance control is widely used in robotic systems as it allows the robot to handle contact forces by simulating a virtual mass-spring-damper(not in the case of our controller) system. This approach ensures smooth and compliant behavior when interacting with external environments, enhancing both safety and functionality.

In this project, we aim to implement task space impedance control for the Panda robot using a ROS2/Ignition software stack. The impedance control will be configured through the robot's effort interface, allowing the robot to interact dynamically with the environment. The control system will utilize a mass-spring model, with each direction in the task space independently controlled.
The objectives of the project include:

1. Implementing the impedance control algorithm using the ROS2 control framework.
2. Designing a simulation scenario to demonstrate the controller's effectiveness in both free-space and obstacle-rich environments.
3. Planning an interaction trajectory using MoveIt!2, focusing on a straight-line path within the manipulator's workspace.
4. Validating the performance of the impedance controller through simulated testing.
5. Assessing position tracking during the interaction using Python plots.

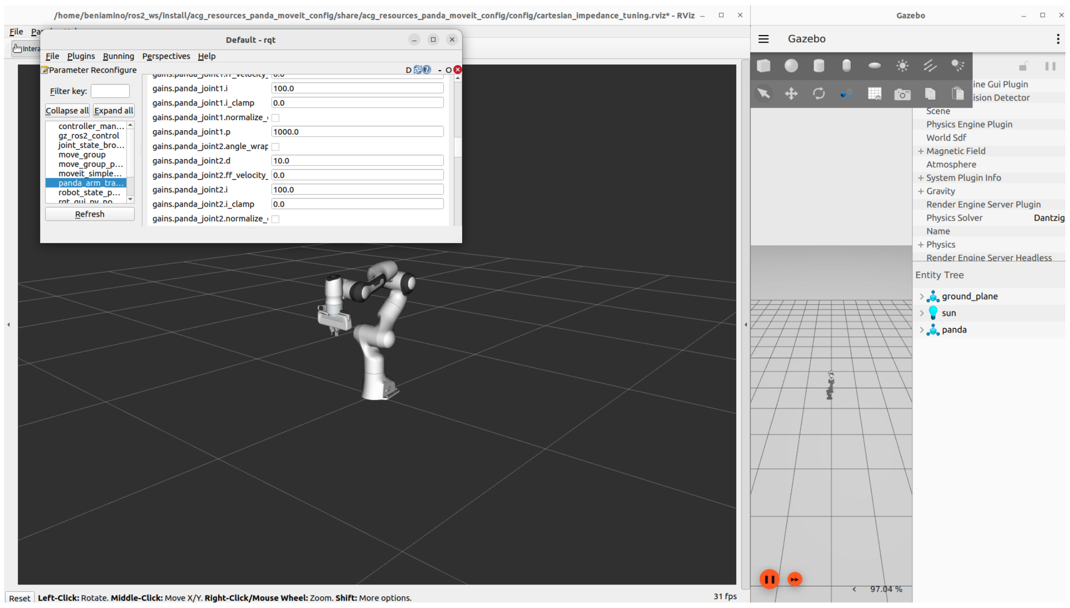
Given the complexity of this task, various constraints must be taken into account, including the need for robot-agnostic solutions, dynamic compensation, and seamless integration with existing frameworks. In this contest the following project aims to deliver a robust and flexible control solution for dynamic robotic environments.

2 Preliminary phase

- **PID Controller**: A first preprocessing phase was to test the "effort" interfaces on the available Panda robot. For this reason, after modifying the "pandaṙos2_control.xacro" file by adding the corresponding "effort" interfaces, a quick PID controller was implemented to verify that the robot could execute trajectories with this interface active. To test this case, once the PID "ros2_trajectory_controller" was calibrated, the same demo seen in class with the non-PID version of the "ros2_trajectory_controller" was re-executed. The results obtained were the same as those seen in the class simulation, thus concluding this preprocessing phase before implementing the actual controller.
- **Calculate pose utility**: Since the robot controller takes in input a desired position(position in x,y,z and orientation) firstly it has been implemented an utility which based on the joint position (written in the file) it calculates the desired position in the task space , which has been given as input to the controller.

3 Impedance Controller

The impedance controller for the Panda robot is implemented using the ROS2 control framework. The primary objective of this controller is to regulate the interaction forces between the robot and its environment by controlling the joint efforts based on a task space impedance control strategy. The following provides a detailed breakdown of the controller's key components and functionality.



gains.panda_joint1.i	100.0
gains.panda_joint1.i_clamp	0.0
gains.panda_joint1.normalize_	☐
gains.panda_joint1.p	1000.0
gains.panda_joint2.angle_wrap	☐
gains.panda_joint2.d	10.0
gains.panda_joint2.ff_velocity_	0.0
gains.panda_joint2.i	100.0
gains.panda_joint2.i_clamp	0.0
gains.panda_joint2.normalize_	☐

3.1 Features

3.1.1 Effort-Based Control:

The controller is designed to compute and apply torques (efforts) to the robot's joints, which are influenced by both the robot's dynamic model and the external forces applied during contact with the environment. To achieve this:

- The robot's Coriolis forces, and gravity compensation are integrated into the control loop to ensure that the dynamics of the robot are fully accounted for.
- The controller uses effort commands to directly control each joint of the robot.

3.1.2 Impedance Control Structure

The task space impedance control is based on a virtual mass-spring system, where each Cartesian direction (X, Y, Z) is governed by:

- Spring stiffness (K), which defines the desired stiffness in the task space direction.
- Damping coefficient (D), which modulates the damping force in response to velocity changes.
- Desired trajectory or position reference in task space, where the goal trajectory is followed while maintaining compliance with external forces.

3.1.3 Trajectory Handling

The controller operates in two primary states: *static impedance control* and *trajectory following*. When executing a trajectory:

- The controller listens for a Cartesian trajectory action goal, provided via ROS2's action server mechanism.
- Once a trajectory is received, the internal goal trajectory is updated, and the controller begins to compute the appropriate joint torques to track this trajectory.
- For each point in the trajectory, the controller computes the error between the desired position/orientation and the actual state of the robot and applies corrective forces using the impedance control law.

3.1.4 Dynamic Model Compensation

To ensure accurate force application, the controller accesses the Panda robot's full dynamic model, which includes:

- Coriolis and centrifugal effects: accounted for in real-time to manage complex dynamic behaviors during high-speed or interaction-heavy tasks.
- Gravity compensation: ensuring that the robot's weight is neutralized to focus purely on external forces.

3.1.5 Action Server Integration

The impedance controller uses a ROS2 action server to handle Cartesian trajectory following. This action server allows:

- Reception of goal trajectories in the form of Cartesian paths that the robot's end-effector should follow.
- Continuous feedback of the robot's state to ensure the trajectory is being followed accurately.

3.1.6 Saturation and Safety Mechanisms

To prevent excessive forces or unsafe interactions, the controller includes saturation mechanisms that limit the torques applied to the joints:

- Joint effort saturation: ensures that the torques applied do not exceed safe operational limits.
- Nullspace optimization: allows for redundancy resolution when multiple solutions exist for a given task, ensuring that internal joint motions remain stable and safe.

3.1.7 Parameter Tuning via YAML

The impedance controller's key parameters, such as stiffness, damping, desired_position, desired_orientation ecc... are configured via a YAML file '(impedance_controller_parameters.yaml)'. This provides flexibility in tuning the control behavior without modifying the core controller code.

3.2 Stiffness and damping tuning parameters

Let's expand further on some of the critical parameters and their impact on the controller's behavior, focusing specifically on stiffness tuning and other key elements that play a significant role in task space impedance control.

3.2.1 Stiffness Tuning

In impedance control, stiffness is a vital parameter as it determines how rigidly or compliantly the robot will follow the desired position in the task space. Stiffness is modeled as a spring that resists displacement from a reference position.

3.2.2 Stiffness in Cartesian Space

- **stiffness.x, stiffness.y, stiffness.z:** These values represent the stiffness of the robot along the X, Y, and Z axes of the task space. A higher stiffness value means that the robot will strongly resist deviations from the desired position, making the system more rigid and responsive to external disturbances.

  Low stiffness: The robot will allow more compliance, meaning it will be easier to push or interact with, and this is typically useful when interacting with humans or soft environments.

  High stiffness: The robot will resist external forces more strongly, useful in tasks that require precise positioning or when handling rigid objects.

3.2.3 Stiffness in Orientation

- **stiffness.orientation_x, stiffness.orientation_y, stiffness.orientation_z:** These values control the stiffness for orientation about the robot's task space axes (roll, pitch, yaw). Similar to positional stiffness, higher orientation stiffness means that the robot will maintain its desired orientation more rigidly.
- Low orientation stiffness: The robot will allow its orientation to be slightly altered by external forces.
- High orientation stiffness: Useful when strict control over the tool's or end-effector's orientation is required, such as during precision assembly tasks.

3.2.4 Damping (D)

In addition to stiffness (K), an other important components in the impedance control model is damping The damping term resists the velocity of the robot's movements. Tuning damping is crucial for ensuring stability and smooth motion:

- **Low damping**: Results in faster, more dynamic movements, but can cause oscillations or instability if the robot overshoots the desired position.
- **High damping**: Slows down the system, resulting in smoother, more controlled movements. This is useful when precision and smooth interaction are required, such as during assembly tasks or when interacting with delicate objects.

The damping rule used is the same of the cartesian_impedance_example_controller used by Franka.

3.3 Gravity Compensation

The "gravity_compensation" flag is used to determine whether the control law should account for gravity compensation, as some robots, like the Panda, are already gravity-compensated, while others may not be.

3.4 Solver

The controller relies on the KDL Inverse Dynamics solver from the acg_unisa_ros2 repository. This solver provides the dynamic matrices of the robot, such as the inertia matrix, the Coriolis forces matrix, and the gravity vector. These dynamic properties are used during the robot's dynamic compensation process.

The solver retrieves these dynamic matrices by leveraging the KDL parser, which reads the robot's URDF (Unified Robot Description Format) and converts it into a KDL representation. The KDL chain is then used by the inverse dynamics solver to compute the robot's dynamic parameters. These parameters are essential for understanding how the robot moves and how external forces, like gravity, impact its movement. This approach allows for accurate dynamic compensation, ensuring that the robot can perform precise movements while taking its physical properties and external forces into account.

3.5 Let's breakdown the code

The design of the impedance controller was developed in strict adherence to the official guidelines outlined in the ROS2 Control documentation, specifically following the instructions provided in the guide for writing new controllers, which can be found at the following link: ROS2 Controllers Documentation. These guidelines ensure that the controller is implemented in a manner that is fully compliant with the ROS2 Control framework, promoting modularity, reusability, and seamless integration within the broader ROS2 ecosystem.

In the following section, we will provide a detailed explanation of the functionality of each method implemented in the impedance controller.

- **on_init()**

- **command_interface_configuration()**
- **state_interface_configuration()**
- **on_configure()**
- **on_activate()**
- **on_deactivate()**
- **update()**
- **on_error()**
- **on_shutdown()**

These functions are essential components of any controller developed within the ROS2 Control framework. They enable the lifecycle management of a controller and ensure that it integrates seamlessly with the broader ROS2 environment, handling state transitions, configurations, and updates.

3.5.1 on_init() – Initializes the controller

This function is responsible for the initial setup of the controller. It ensures that the necessary parameters are loaded, resources are allocated, and the system is ready for configuration. The initialization step is crucial because it lays the groundwork for the controller's subsequent operations. It prepares the controller to be configured and activated by establishing the basic conditions needed to proceed.

3.5.2 command_interface_configuration() – Provides the configuration for the command interfaces used by the controller

This function defines which command interfaces (e.g., joint efforts, velocities, or positions) are required by the controller to send control commands to the robot. Command interfaces are essential because they allow the controller to actuate the robot by specifying the physical inputs, such as torques or forces, necessary to achieve the desired behavior.

3.5.3 state_interface_configuration() – Provides the configuration for the state interfaces used by the controller

This function defines the state interfaces (e.g., joint positions, velocities, or external forces) that the controller will read from the robot. These state interfaces provide feedback from the robot, which is necessary for the controller to make informed decisions about how to adjust its commands. Without accurate state feedback, the controller would not be able to perform effective closed-loop control.

3.5.4 on_configure() – Configures the controller based on the provided settings

This function is called to configure the controller when it transitions from an unconfigured state to an inactive state. In this step, the controller loads additional configuration details and prepares the necessary resources for operation (e.g., setting up action servers or reading from configuration files). The configuration stage is critical because it tailors the controller's behavior to the specific hardware and task at hand.

In the on_configure function, several critical parameters are initialized to ensure the controller is properly configured and ready to execute its tasks. The process begins by retrieving the ROS parameters, such as joint model group names and stiffness/damping gains, from the parameter server. This allows the controller to adapt dynamically to different robot configurations. The robot's kinematic model is then loaded, enabling precise control over the robot's joints and end-effector. Joint limits, positions, and velocities are initialized, providing necessary boundary conditions for safe motion execution. The inverse dynamics solver is also set up to calculate the torques required to follow the desired trajectories, considering the robot's dynamic properties like gravity matrix(if enabled) and Coriolis effects. Stiffness and damping gains are configured to control the robot's compliance during interactions, and nullspace gains are set to optimize joint movements. Additionally, the action server is created to handle trajectory following, allowing the controller to execute paths in real-time.

3.5.5 on_activate() – Activates the controller, making it ready to begin operation

In the on_activate function, the controller transitions from an inactive state to an active state, which means it becomes fully operational and ready to control the robot. The function begins by logging the activation process, followed by checking whether the controller is already active, in which case it simply returns success without reactivating. Next, it verifies that all necessary hardware interfaces (such as joint position, velocity, and effort interfaces) are available and valid, ensuring the controller has access to the required feedback and control signals from the robot's hardware. If any of the hardware interfaces are missing or invalid, activation fails.

Once the interfaces are confirmed, the controller reads the current state of the robot from the hardware, which includes joint positions and velocities.The controller also updates any dynamic parameters that might have been modified since the last configuration, ensuring the latest control gains or limits are applied.

3.5.6 on_deactivate() – Deactivates the controller, stopping its operation

This function transitions the controller from an active state to an inactive state. During deactivation, the controller stops sending commands and ceases to control the robot, ensuring that the system is safely halted. This is important to allow the system to stop in a controlled manner when the controller is no longer needed or when switching between different controllers.

3.5.7 on_error() – Handles any errors encountered by the controller

This function is invoked when an error occurs in the controller. It allows the system to handle faults gracefully, often by stopping the controller or resetting certain aspects. Error handling is important to ensure system safety and robustness, particularly in real-world scenarios where unexpected issues may arise during operation.

3.5.8 on_shutdown() – Shuts down the controller, performing any necessary cleanup

This function handles the transition to the final state, where the controller is shut down and cleaned up. It releases resources, stops any remaining actions, and prepares the system for safe termination. The shutdown process is vital for ensuring that no residual operations or resources are left in an inconsistent state, which could affect future operations.

3.5.9 update() – Performs an update cycle of the controller, computing new commands

The update() function is the core of the control loop. It is called periodically to compute new control commands based on the current state of the robot and the control objectives (e.g., impedance control). This function handles the real-time decision-making and command issuance. It is critical because it determines how the robot behaves in response to changes in its environment and how it follows the desired trajectories.

- **Cycle Time Measurement** : The function starts by logging the current time to track the duration of each control cycle. This is useful for performance monitoring and ensuring that the control loop meets real-time requirements.

- **Check Active State**:Before proceeding with any control logic, the function checks whether the controller is active. If it is not active, the function returns without performing any operations. This ensures that the controller only operates when it is in the correct state.

- **Update Dynamic Parameters**:Any dynamic parameters, such as stiffness or damping gains, that may have changed since the last update are refreshed. This step ensures that the controller is always using the latest parameters during execution. Useful during tuning phaase.

- **Read Current Robot State**:The current joint positions and velocities are read from the robot's hardware interfaces and stored in Eigen vectors. This state feedback is crucial for the controller to compute the necessary joint torques and effort for proper movement.

- **Compute Dynamic Parameters**:The function computes important dynamic properties such as the gravity vector, and Coriolis forces using an inverse dynamics solver.

- **Trajectory Tracking**:

  Trajectory Reception:When a new trajectory is received, it is stored in goal_trajectory_, and the controller checks whether the trajectory consists of a single point (static impedance control) or multiple points (trajectory execution mode). This decision determines whether the controller will simply hold a position or follow a dynamic path.

  Trajectory Initialization:The internal variables internal_time_ (tracking time) and number_reached_trajectory_points_ (keeps track of the last reached trajectory point) are initialized. This setup allows the controller to monitor how much of the trajectory has been executed and how much remains. the bigger timestamp(next waypoint so) but timestamp smaller than the controller one

  Time-Based Interpolation:During each control cycle (inside the update function), the controller increments internal_time_ by the duration of the cycle. This internal time keeps track of how far along the trajectory the robot should be at any given moment.The controller then compares internal_time_ to the timestamps of the trajectory points (stored in goal_trajectory_.points). It checks which trajectory point corresponds hasto the current time. This is done by iterating through the trajectory and comparing the current time with the time_from_start value for each point. If the current time matches a trajectory point, the controller updates the desired end-effector position and orientation to the values at that point.

  Handling Trajectory Points:As the controller executes the trajectory, it keeps track of the number of points reached using number_reached_trajectory_points_. Once a trajectory point is reached, the controller moves to the next point. When the robot reaches the final trajectory point, the controller logs that the trajectory execution is complete and transitions back to static impedance control, where the robot will maintain the last position and orientation.

- **Compute Joint-Space Effort**:During both static_impedance_controller and trajectory execution phases , the controller applies an impedance control law to compute the joint torques that are required to follow the interpolated trajectory while maintaining compliance with external forces.

- **Command the Robot**:The computed joint torques are applied to the robot's joints through the hardware interfaces, allowing the robot to move according to the computed control inputs.

- **Completion of Trajectory Execution:** : Once the controller reaches the last point of the trajectory, it logs that the execution is complete, and signals the action server that the trajectory has been completed. The controller then returns to a static control mode, holding the final position.

3.5.10 Control law

The function compute_effort is the core part of the control loop, responsible for calculating the joint torques required to drive the robot's end-effector to the desired position and orientation in task space, while also optimizing the robot's posture through nullspace control. This is the start of the torque calculation process.

Breakdown of Key Steps in compute_effort:

1. **Position and Orientation Error Calculation:**

   The position error is calculated as the difference between the desired position (desired_ee_position_) and the current end-effector position (actual_ee_position_) .

   The orientation error is computed using quaternions to avoid discontinuities. The current and desired orientations are represented as quaternions, and their difference is calculated by quaternion multiplication. $\mathbf{e_q} = q_{\text{actual}} \cdot q_{\text{desired}}^{-1}$

2. **Impedance Control Law**:

   The position and orientation errors are used to calculate task-space forces according to the impedance control law: $F = K \cdot (x_{\text{desired}} - x_{\text{current}}) + D \cdot (v_{\text{desired}} - v_{\text{current}})$. K is the stiffness gain, D represent the damping one, while x_desired and v_desired represents the desired position and velocity(zero in our case) and x_current and v_current represents the actual position and velocity. The result is the task-space force F_task which specifies how much force should be applied in each direction to reduce the position and orientation errors.

3. **Transforming Task-Space Forces into Joint-Space Torques:**

   To control the robot's joints, the task-space forces F_task must be converted into joint torques using the Jacobian transpose: $\boldsymbol{\tau}_{\text{task}} = \mathbf{J}^T \cdot \boldsymbol{F}_{\text{task}}$ where: $\boldsymbol{\tau}_{\text{task}}$ is the joint torques derived from the task-space forces. J^T is the traspose Jacobian matrix, representi joint movements.

4. **Nullspace Control for Redundancy Resolution**: Nullspace control is used when the robot has redundant degrees of freedom (DOF), meaning more joints than necessary to achieve a task. Nullspace control allows the robot to optimize its configuration while performing the primary task.

   Steps in Nullspace Control:

   - Nullspace Torques:

     The nullspace torque is computed to maintain the robot's posture and avoid joint limits while it tracks the desired task-space motion:

     $$\boldsymbol{\tau}_{\text{nullspace}} = \left(\mathbf{I} - \mathbf{J}^T \mathbf{J}^{T^\dagger}\right) \cdot (K_{\text{null}} \cdot (\boldsymbol{q}_{\text{desired}} - \boldsymbol{q}_{\text{current}}) + D_{\text{null}} \cdot (-\dot{\boldsymbol{q}_{\text{current}}}))$$

     I: Identity matrix (same size as the number of joints).

     $\mathbf{J}^{T^\dagger}$:Pseudo-inverse of the Jacobian transpose, ensuring the system can handle redundancy.

     K_{null}:Stiffness in the nullspace (from nullspace_stiffness_)

     D_{null}:Stiffness in the nullspace (from nullspace_damping_)

     $q_{\text{desired}}, q_{\text{current}}$:Desired and current joint positions.

   - **Total Joint-Space Torque:** The total joint torque combines the task-space torque and the nullspace torque: $\boldsymbol{\tau}_{\text{total}} = \boldsymbol{\tau}_{\text{task}} + \boldsymbol{\tau}_{\text{nullspace}}$

5. **Gravity and Coriolis Compensation** Once the joint torques have been calculated based on the desired trajectory, the gravity vector and Coriolis forces are added to ensure the robot can compensate for its own weight and other dynamic effects.

   Gravity Compensation:The controller adds gravity compensation to the computed torques if enabled, making sure the robot moves as expected without being affected by its own weight.

   Coriolis Compensation:Coriolis forces, which arise from the movement of the robot's joints, are also added to the final torque to make the motion smoother and more predictable.

6. **Saturation of Torque Changes (saturateTorqueRate)** After the total joint torques have been computed, the saturateTorqueRate function is used to limit how quickly the torques can change. This is critical to ensure safe and smooth movements.

   Torque Rate Limitation: The function compares the desired torque $\tau_{\text{d\_calculated}}$ with the current torque $\tau_{\text{J\_d}}$, and limits the difference based on $\Delta\tau_{\text{max}}$. This ensures that torque changes are not too abrupt, protecting both the robot and the environment.

3.5.11 Action Server Implementation in the Impedance Controller

Action server implementation comes from the following ros2 guide: Writing an action server and client (C++).

(a) **Receiving a Goal (received_goal_trajectory_callback):**
When the action client sends a new trajectory goal to the impedance controller, the received_goal_trajectory_callback function is triggered. This function checks if the controller is ready to accept the goal and stores the received trajectory for execution.
The main steps are:

Controller State Check: The callback first checks whether the controller is currently active. If the controller is not in an active state, the goal is rejected with a log message. This ensures that goals are only accepted when the controller is ready to execute them.

Log and Store the Goal: If the controller is active, it logs the receipt of the trajectory and stores the trajectory in an internal variable (goal_trajectory_). Optionally, the callback can log details about the trajectory points for debugging purposes.

Trajectory Initialization: The function marks that a new trajectory has been received (trajectory_received_ = true) and resets the internal executio time (internal_time_ = rclcpp::Duration(0, 0)), allowing the controller to start from the eginning of the trajectory.

Return ACCEPT_AND_EXECUTE: If the goal is accepted, the action server returns ACCEPT_AND_EXECUTE, indicating that the trajectory will be executed.

(b) **Canceling a Goal (cancelled_goal_trajectory_callback):**
The cancelled_goal_trajectory_callback is invoked when the action client requests the cancellation of a current goal trajectory. This function handles the cancellation request and prepares the controller to stop executing the current trajectory.
The steps are:

Log Cancellation Request: The callback logs that a cancellation request has been received, including the goal's UUID for tracking and debugging purposes.

Accept Cancellation: The callback returns ACCEPT to confirm that the cancellation request has been accepted. This tells the action client that the server will stop executing the trajectory.

(c) Accepting a Goal (accepted_goal_trajectory_callback):
When the impedance controller accepts a new trajectory goal, the accepted_goal_trajectory_callback function is triggered. This callback initializes the goal for execution and prepares the controller to follow the trajectory. The key steps are:

Log Acceptance: The callback logs the acceptance of the goal along with its UUID for debugging purposes. This ensures there is a clear record of which trajectory is being executed.

Goal Handle Initialization:The function initializes a goal handle (goal_handle_) for the accepted goal. This handle is important as it allows the server to track and manage the goal's execution state.

Prepare for Execution:The function marks the controller as ready to execute the trajectory by setting trajectory_received_ = true. If the controller is in the static_impedance_control state, it prepares to transition to trajectory execution.

4 Simulation scenario

panda_moveit_cartesian_demo_nobag.cpp

The simulation scenario for the Panda robot is designed to test and demonstrate task-space impedance control with Cartesian trajectories. Using MoveIt! and ROS 2, the robot is set up to plan and execute complex trajectories, allowing interaction with both free space and obstacles. The main components of the simulation scenario include trajectory generation, visualization in RViz, and execution through an action client connected to a server.

- **Trajectory Generation and Planning**

  The simulation begins by generating a Cartesian trajectory using the *generate_oblique_trajectory* function. This function creates a sequence of waypoints in 3D space that the Panda robot will follow. The initial position of the end-effector is obtained from the robot's current state, and the trajectory length is set to 60 cm. The trajectory moves along both the x- and y-axes, with a ratio of 1:3, meaning the robot will move more along the y-axis than along the x-axis. The function computes the required displacements and populates a list of waypoints that represent the trajectory.

- **RViz Visualization**

  Once the waypoints are generated, they are visualized in RViz using MoveItVisualTools. The visualize_waypoints function publishes both start and end waypoints as a green sphere in RViz, allowing the user to see the planned trajectory. This is helpful for verifying the trajectory before execution. The simulation prompts the user through RViz to execute the planned Cartesian trajectory by pressing 'Next' in the RViz interface. This is done via the execute_trajectory function, which triggers the execution process.

- **Trajectory Execution**

  The execution of the planned trajectory is managed by the TaskSpaceClient node. This node communicates with an action server using ROS 2 actions, specifically the FollowCartesianTrajectory action. The execute_trajectory function waits for the user input in RViz and then triggers the trajectory execution using the action client. The action client sends the trajectory to the server, which controls the robot's movement.The Panda robot follows the Cartesian path, which consists of several waypoints in space.

- **Recording Trajectories with ROS Bag**

  To record the planned and executed trajectories for further analysis, the simulation uses ROS 2 bags. The BagFileHandler is responsible for writing the trajectory data to bag files. This is crucial for validating the controller's performance, as it allows for replaying the trajectories and assessing position tracking performance.During the execution, the simulation records feedback on the trajectory via topics such as panda_feed and stores this data in a .bag file. These recorded trajectories can be reviewed later to evaluate how closely the robot followed the planned path.

4.1 TaskSpaceClient

As for the action server also the action client has been configured with the following guide: Writing an action server and client (C++) .
The TaskSpaceClient node is designed to handle the execution of Cartesian trajectories. It acts as an interface between the robot's impedance controller (through an action server) and the motion planning components. This node also handles trajectory planning in task space, monitors feedback from the action server, and saves trajectory data into ROS 2 .bag files for analysis or replay.

Communication with the Action Server:
The TaskSpaceClient class creates an action client for the FollowCartesianTrajectory action, allowing it to send trajectory goals to the server and monitor their execution.

- **Sending Trajectories**:
  The function *task_space_goal_forwarding* sends a pre-planned Cartesian trajectory to the impedance controller. This trajectory defines the path the robot should follow in its task space.

    - The trajectory is sent as a goal to the action server. If the server is not available, the node logs an error and throws an exception. Otherwise, the trajectory is processed asynchronously.
    - The goal is accompanied by several callback functions:
        * **Goal Response Callback**: Confirms if the trajectory goal has been accepted or rejected by the server.
        * **Feedback Callback**: Receives continuous feedback from the server during execution, tracking the robot's progress.
        * **Result Callback**: Informs the user whether the goal was completed successfully or if it failed.

Trajectory Planning and Generation
The node includes functionality for generating Cartesian trajectories from a series of waypoints and converting joint-space trajectories into task-space trajectories.

- **Cartesian Path Planning**: The function cartesian_path_generation computes a Cartesian path based on waypoints. It uses the MoveIt! planning interface to interpolate between the waypoints, calculating the trajectory the robot should follow in task space.
    - The planning process includes parameters like the end-effector step size (eef_step) and the jump threshold for smoother motion planning.
    - The generated Cartesian path is then converted into a trajectory that can be sent to the action server for execution.

Converting Joint-Space Trajectories to Cartesian Trajectories
The node can convert joint-space trajectories into Cartesian trajectories using the robot's kinematic model

- **Joint to Cartesian Conversion**: The get_task_space_trajectory function takes a joint-space trajectory and calculates the corresponding Cartesian trajectory. It does this by computing the end-effector pose and velocity for each joint configuration using direct and first-order kinematics.
- **Direct Kinematics**: This involves calculating the end-effector's pose in Cartesian space given the joint positions, which is crucial for ensuring the robot reaches the desired task-space position.
- **Velocity Computation**: The first-order kinematic computation ensures that the robot's velocity in task space is derived from the joint velocities, allowing for smooth motion transitions during trajectory execution.

Updates

- **Result Monitoring**: The task_space_result_callback checks the result of the trajectory execution. If the action succeeds, the node logs this information. If the goal is aborted or canceled, it logs an error and informs the user of the failure.

4.2 Bag file utility

The *BagFileHandler* class was implemented as a separate utility to provide flexibility in saving both the planned trajectory and the executed trajectory during robot operations. This separation allows for an efficient logging process where different stages of the robot's movement, from planning to execution, are captured and stored in distinct .bag files.

- **Initialization and Bag File Creation**
  When an instance of BagFileHandler is created, it opens a new .bag file for writing. This file is named dynamically using a base filename provided during initialization, appended with a timestamp to ensure uniqueness. The timestamp ensures that every execution or recording session gets saved in a separate file. The BagFileHandler uses the rosbag2_cpp::Writer API to handle the bag file creation. By calling the *set_filename_with_timestamp method*, the class dynamically generates the file name, and the writer opens the file with appropriate storage options.

- **Topic Creation**
  The *create_bag_topics* method defines the topics within the bag file where trajectory data and feedback messages will be recorded. Each topic is associated with metadata, such as its name, message type, and serialization format. The method prints information about each created topic, helping with debugging and ensuring proper topic registration.
- **Writing Feedback to Bag Files**
  The *write_feedback* function saves feedback data received during the robot's trajectory execution. It serializes the feedback messages and writes them to the designated topic in the .bag file, along with a timestamp.
- **Writing Trajectories to Bag Files**
  The *write_trajectory* method records serialized trajectory data into the .bag file under specific topics.
- **Reading Data from Bag Files**
  The *bag_file_path* method allows the utility to retrieve a sequence of poses from a previously saved .bag file. It reads messages from the bag file, deserializes them, and extracts the relevant path or trajectory data.

5 Trajectory analyzer

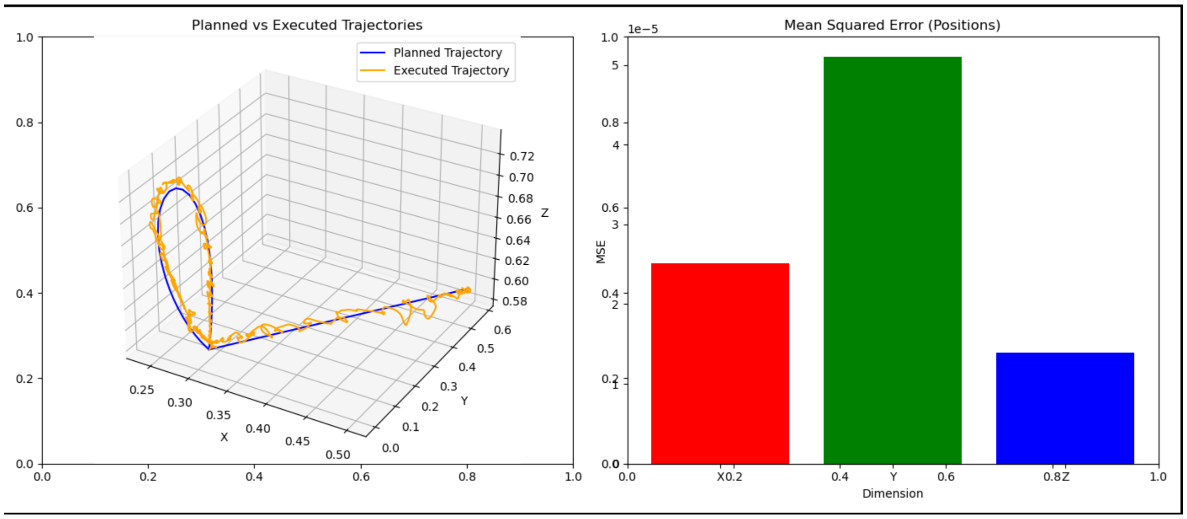
The trajectory comparison tool, implemented in Python, is designed to analyze the differences between a planned trajectory and an executed trajectory in a ROS2 environment. This tool leverages data from ROS bag files, where both trajectories are stored, and provides a comprehensive evaluation of positional and orientational accuracy by comparing the two paths.

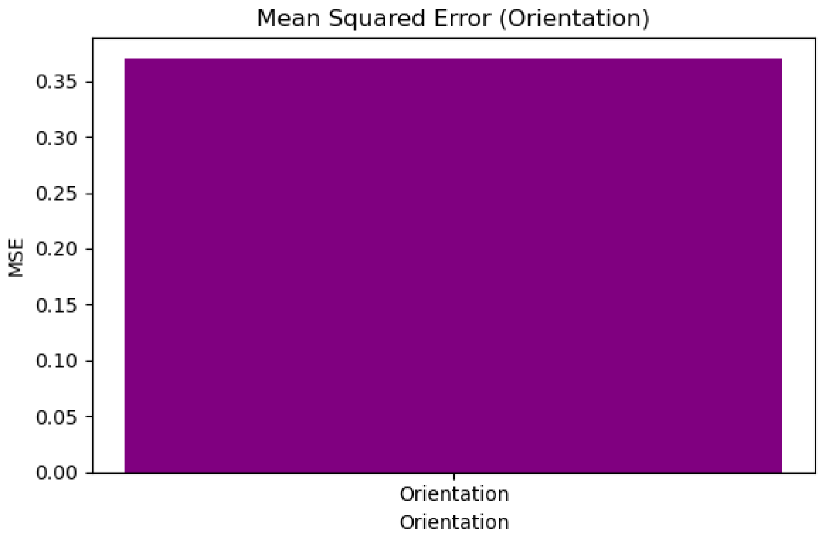
5.1 Functionalities

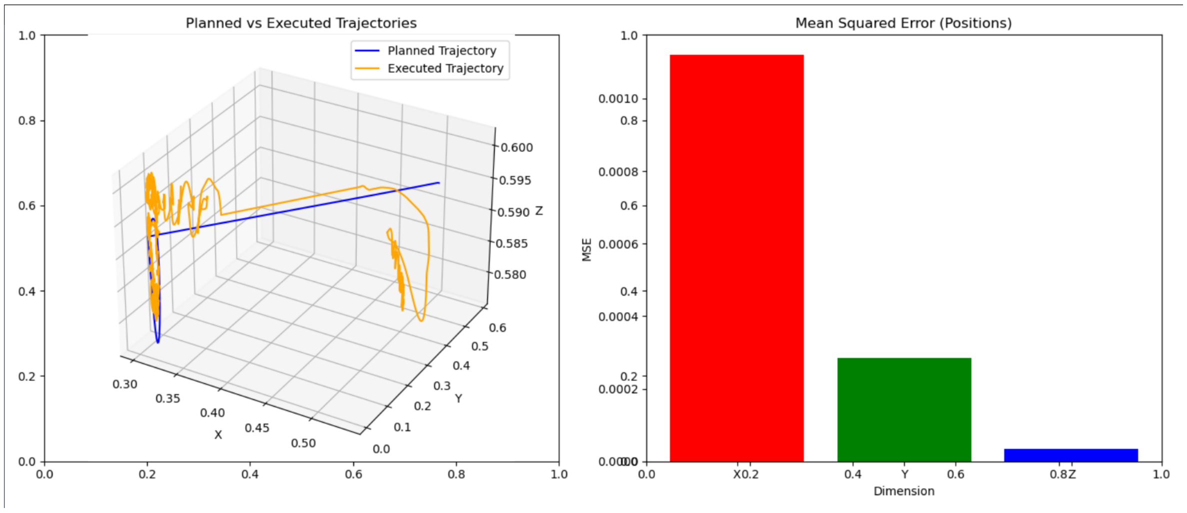
- **Message Reading** :
  - The tool reads messages from the provided ROS2 bag files and filters them by specific topics that contain the planned or executed trajectories.
- **Position and Orientation Extraction:**
  - From each message, the tool extracts the 3D position (X, Y, Z) and orientation (quaternions) data. These are stored in NumPy arrays to facilitate further mathematical operations and error calculations.
- **Error Calculation:**
  - The MSE calculation compares the planned and executed trajectories. A spatial KD-Tree is used to match the closest points between the two trajectories for a precise comparison.
  - The positional errors are calculated for each axis (X, Y, Z), while the orientation error is derived using quaternion-based rotation differences.
- **Visualization:**
  - The tool plots both the planned and executed trajectories in a 3D space, allowing the user to visually assess the differences between the two.
  - It also provides separate MSE bar charts for positional errors (X, Y, Z) and orientation errors. This enables a clear and concise representation of where deviations occur and how significant they are.

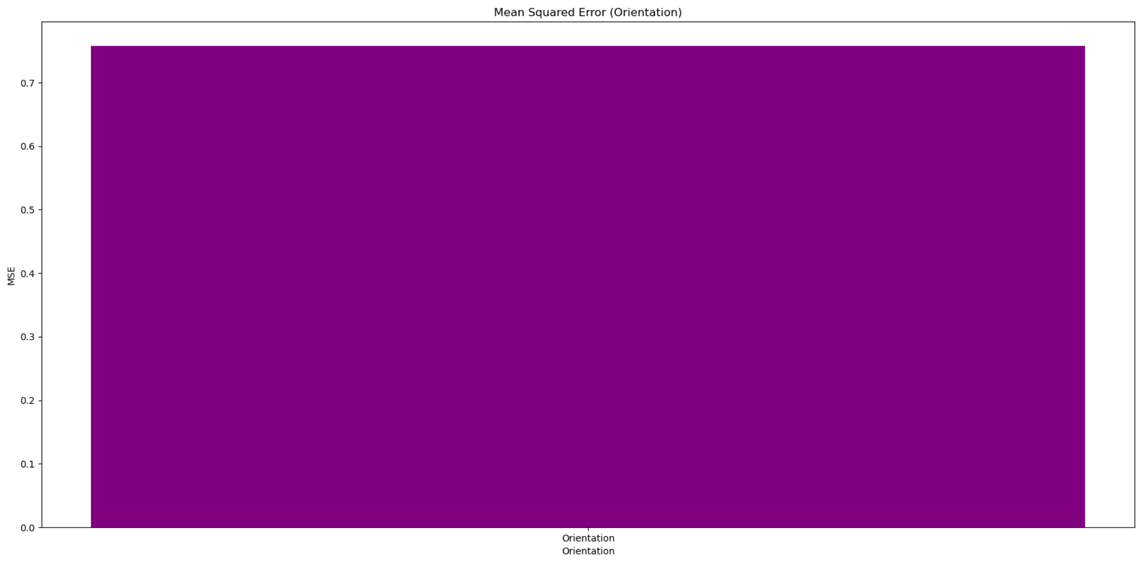
6 Results

In this section, we focus on the comparison between the planned and executed trajectories, highlighting any deviations and quantifying the accuracy of the execution. By analyzing the generated plots, we gain insight into how well the robot adhered to the planned path and how environmental factors, such as obstacles, influenced the execution. We utilize Mean Squared Error (MSE) metrics to assess the differences in position and orientation between the two trajectories.









As we can observe from the plots, as long as the robot does not encounter an obstacle, the performance in both simulations is similar, as the robot is able to follow the planned trajectory quite well. However, when the robot encounters an obstacle (a wall in the case of the simulation), it can no longer proceed along the axis where the wall is located, thereby accumulating error on both the X and Y axes. Finally, regarding orientation, upon impact with the wall, the error also increases.

6.1 Issues and limitations

The following code faces several limitations:

1. **Trajectory starting from the current robot pose**: The executed trajectory starts from the robot's current pose when the planning node is launched. Given that the trajectory length is set to 60 cm, there is a risk of the robot easily exceeding its workspace limits, potentially causing errors or inefficiencies in the movement.

2. **Inflexibility of trajectory execution**: The trajectory is defined directly within the simulation file, making it difficult to execute different trajectories simply by rerunning the simulation file from the terminal. To address this, a bag file was created to allow user-defined trajectories that

can be input into the planning program. This approach enables more flexibility in executing different trajectories without the constraint of starting from the robot's current pose. (This last constraint could have been also removed by modifying the function that defines the trajectory to avoid setting the initial point to the robot's current pose).

3. **Impedance controller performance variability**: The Impedance controller is subject to issues related to the performance of the computer running the simulation. It was observed that, when running the code on two different machines with the same code, the robot started from different positions, indicating performance-related discrepancies.

Specifica	Machine 1	Machine 2
Processor Model	intel core i7 8th gen	Apple M2 con CPU 8-core, GPU 8-core(virtual machine)
RAM Model and Capacity	DDR4(Samsung)(16 GB)	Apple GPU(8 GB)

4. **Tuning parameters issue**: Another problem encountered was with parameter tuning. It was observed that parameters found using the "dynamic reconfigure" tool in RQT, which successfully stabilized the robot in a given position during dynamic configuration, did not always stabilize the robot correctly when saved in the ros2_controller.yaml file and used at launch.

7 Improvements

To overcome the limitations outlined in the first two points, a new package called "acg_resources_panda_trajectory_generation" was created thanks to which the trajectory is no longer hardcoded within the simulation file. Instead, the user can define the trajectory in advance and input it into the simulation, allowing for more diverse and complex movements. This package handles the generation of user-defined trajectories, providing more flexibility and control over the robot's movements.

Additionally, an executable file named "panda_moveit_cartesian_demo.cpp" . This file is responsible for executing the user-defined trajectory and integrating it into the robot's planning and execution pipeline. This further simplifies the process of executing varied trajectories, as the user can simply feed different bag files into the system, making it much more versatile than the previous hardcoded approach.

7.1 Trajecory generation

The file trajectory_node.py is a ROS 2 node designed for generating and writing user-defined straight line trajectories into a ROS bag file. This node allows users to specify the length, offset, and rotation (in radians) of the trajectory, and saves the trajectory data in a bag file for future use in simulation or robot execution.

- **Parameter Setup**: The node accepts user inputs for trajectory length, offset, and rotation (gamma) to define the path.
- **Rotation Matrix Calculation**: The node computes the rotation matrix based on user-defined angles to adjust the trajectory orientation.
- **ROS Bag Writing**: The generated trajectory (start and end points) is serialized and written into a ROS bag file, which can be used as input for trajectory execution.
- **User Interaction**: Prompts the user to input the trajectory's length, offset, and orientation directly through the terminal.

7.2 Simulation scene with bag file trajectory as input

panda_moveit_cartesian_demo.cpp
This node is an extended version of the earlier panda_moveit_cartesian_demo.cpp, which adds advanced features, such as ROS bag interaction. It demonstrates more robust planning and execution capabilities

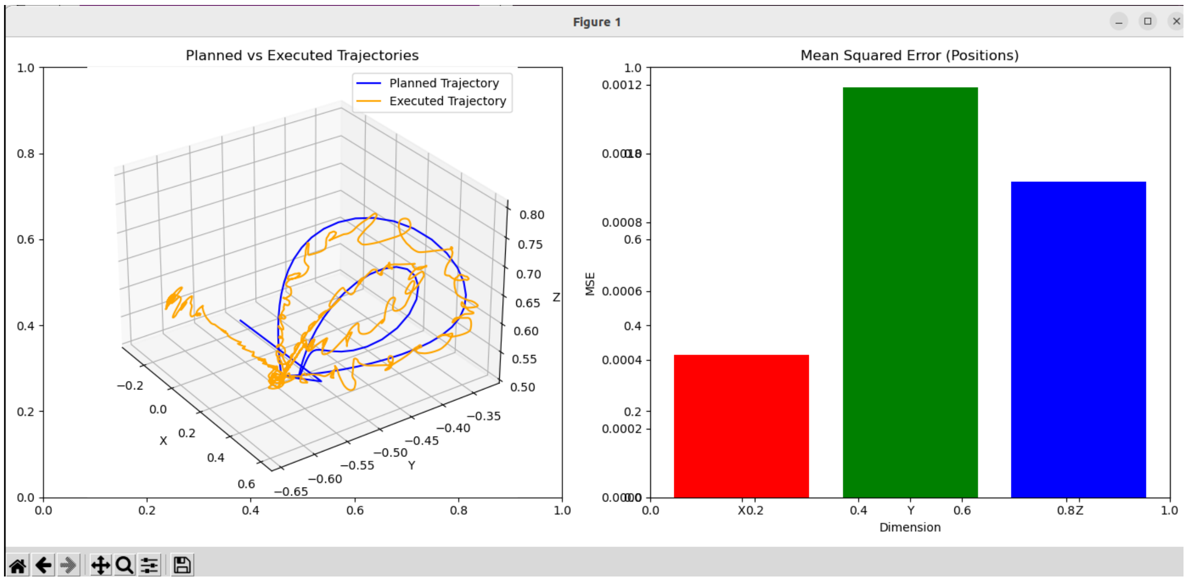
for robot trajectories, particularly integrating user-defined trajectories from a ROS bag.
The following sequence is performed:

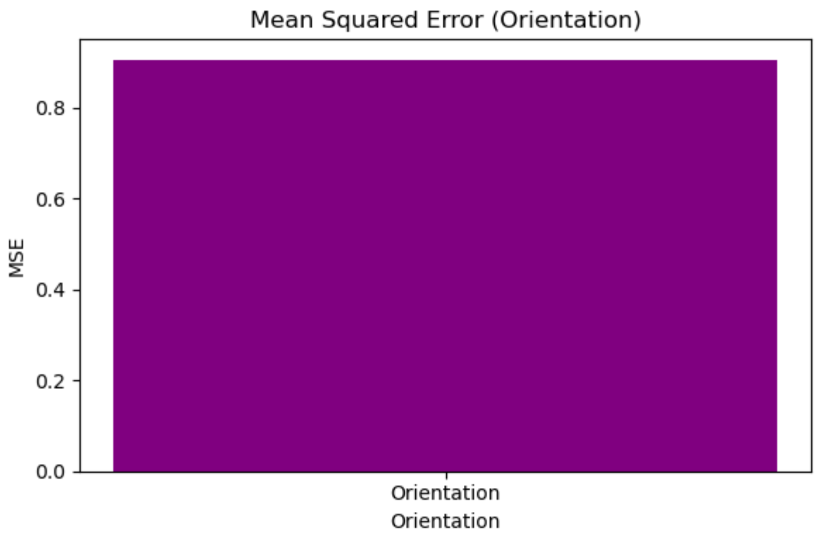
1. **Point-to-Point Simulation**: Before executing the main Cartesian trajectory, the file performs a point-to-point simulation. This step ensures that the robot plans its motion towards specific poses or waypoints before executing the full trajectory. The point-to-point planning is visualized and confirmed using RViz, providing a more controlled and incremental approach.

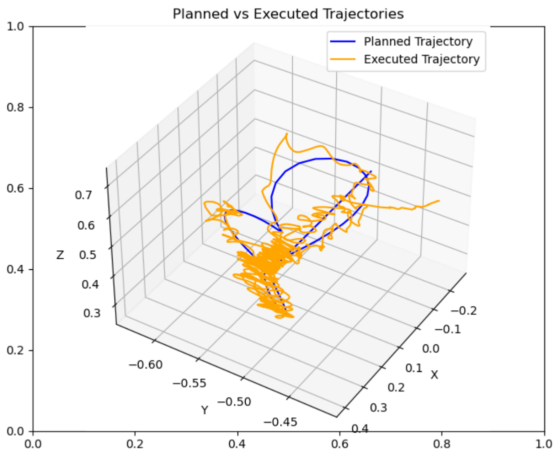
2. **Execution of the Trajectory**: After the point-to-point simulation is successfully planned, the trajectory is then executed.

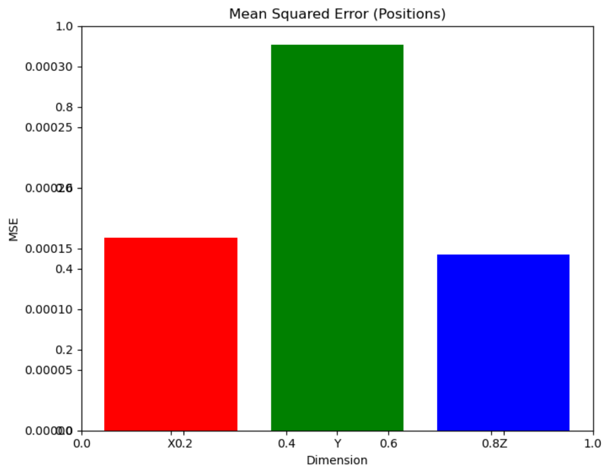
3. **Return to Ready Position**: Once the trajectory is executed, the robot is commanded to return to its ”ready” position. This is an important step that brings the robot back to a safe or neutral pose, ensuring that it is ready for subsequent operations or simulations.

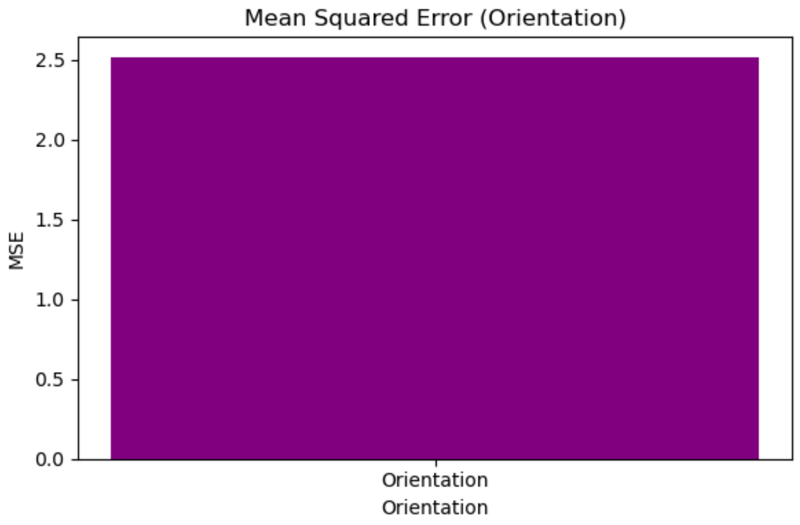
8 Results pt_2











As we can observe from the images, the same observations made in the previous case also hold true for the trajectory executed from the bagfile. Specifically, when the robot encounters an obstacle, its position deviates, following the surface of the wall (the obstacle being considered) in parallel. It is important to note that the plots shown above can also be generated for the point-to-point trajectory (the one that moves from the robot's current pose to the first of the two waypoints defining the straight-line trajectory) and for the trajectory leading to the ready position (defined in the "panda.srdf" file).

9 Conclusions

In conclusion we can say that the proposed controller meets the requirements of the task. Thanks to the corresponding .yaml file, the controller can be configured for any joint group and any controller, as no assumptions were made regarding the robot itself. A similar reasoning applies to the files that simulate interaction scenarios with obstacles. Additionally, as expected with impedance control, when the robot comes into contact with an obstacle, it changes its direction and follows the trajectory by moving parallel to the obstacle (in this case, the wall in the simulation scenario).

However, regarding performance, although the errors are not significant—especially in terms of position (less so for orientation)—the MSE (Mean Squared Error) could be reduced with better parameter tuning. Unfortunately, this is complicated by the issues previously discussed in Section 6.1.