POLITECNICO DI BARI

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING

MASTER'S DEGREE IN AUTOMATION ENGINEERING

MASTER THESIS IN

Dynamical Systems Theory

POSE ESTIMATION AND CONSTRAINED CONTROL OF A ROBOTIC MANIPULATOR IN OPERATIONAL SPACE

Supervisor:
Prof. Engr. Mariagrazia Dotoli

Candidate:
Tommaso Savino

Co-Supervisors:
Prof. Engr. Raffaele Carli
Engr. Fabio Mastromarino

ACADEMIC YEAR 2024-2025

Politecnico di Bari

LIBERATORIA ALLA CONSULTAZIONE DELLA TESI DI LAUREA DI CUI ALL'ART.4 DEL REGOLAMENTO DI ATENEO PER LA CONSULTAZIONE DELLE TESI DI LAUREA (D.R. n. 479 del 14/11/2016).

Il sottoscritto **Tommaso Savino** matricola **589467**
Corso di Laurea: **Ingegneria dell'Automazione**
Autore della presente tesi di Laurea dal titolo: **POSE ESTIMATION AND CONSTRAINED CONTROL OF A ROBOTIC MANIPULATOR IN OPERATIONAL SPACE**
Parole chiave: **Pose Estimation, Task Space Control, Manipulator, SLAM, ROS2, MoveIt2**
Abstract: **This work lies within the field of industrial robotics, specifically focusing on motion planning and control of articulated manipulators using open-source tools based on ROS2. The project concentrates on operational space control of a fixed-base manipulator by integrating a trajectory planning tool for trajectory planning with a visual localization system based on SLAM to estimate the end-effector pose. The goal of this thesis is to develop an operational space controller that uses the error between the desired end-effector pose and the one estimated through SLAM to generate torque commands for trajectory tracking. This work includes the integration of an RGB-D camera mounted on the robot, the Niryo Ned2, the implementation and configuration of a SLAM system for pose estimation, trajectory planning with the MoveIt2 tool, the design of a custom controller as a ROS2 plugin, and the execution of simulations to evaluate the trajectory tracking performance. The results confirm the feasibility of the proposed approach and highlight both its limitations and its potential for future developments in real-time visual control scenarios.**

☒ Autorizzo
☐ Non autorizzo

la consultazione della presente tesi, fatto divieto a chiunque di riprodurre in tutto o in parte quanto in essa contenuto.

Bari, data 12/07/2025
Firma

Al termine di questo percorso, desidero esprimere la mia più sincera gratitudine a tutte le persone che hanno reso possibile la realizzazione di questa tesi e che mi hanno accompagnato durante gli anni universitari.

In primo luogo, ringrazio la mia relatrice, Prof.ssa Mariagrazia Dotoli, il mio correlatore, Prof. Raffaele Carli, e l'Ing. Fabio Mastromarino. La loro guida, disponibilità e i preziosi suggerimenti hanno orientato e arricchito costantemente il mio lavoro.

Un ringraziamento speciale va alla mia famiglia, per il supporto incrollabile, la fiducia e l'incoraggiamento che non mi hanno mai fatto mancare. In particolare, un pensiero va a mio fratello gemello Francesco, compagno di studi e di vita, con cui ho condiviso fatiche, successi e paure.

Ai miei genitori, spero di rendervi sempre orgogliosi di me, come oggi.

Ai miei nonni,

a chi c'è sempre stato,

a chi se n'è andato.

Ringrazio i compagni di corso con cui ho condiviso questo viaggio con passione e spirito di gruppo. Infine, un pensiero affettuoso va ai miei amici di sempre, che hanno saputo farmi sorridere e ricaricare le energie nei momenti più intensi.

A tutti voi, grazie di cuore.

Abstract

This work lies within the field of industrial robotics, specifically focusing on motion planning and control of articulated manipulators using open-source tools based on ROS2. The project concentrates on operational space control of a fixed-base manipulator by integrating a trajectory planning tool for trajectory planning with a visual localization system based on SLAM to estimate the end-effector pose. The goal of this thesis is to develop an operational space controller that uses the error between the desired end-effector pose and the one estimated through SLAM to generate torque commands for trajectory tracking. This work includes the integration of an RGB-D camera mounted on the robot, the Niryo Ned2, the implementation and configuration of a SLAM system for pose estimation, trajectory planning with the MoveIt2 tool, the design of a custom controller as a ROS2 plugin, and the execution of simulations to evaluate the trajectory tracking performance. The results confirm the feasibility of the proposed approach and highlight both its limitations and its potential for future developments in real-time visual control scenarios.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent decades, robotics has assumed an increasingly central role in manufacturing, research, and education. This expansion is largely driven by steady improvements in computational hardware, sensor technologies, and the emergence of open-source software ecosystems that support flexible and scalable robot development. One of the most prominent frameworks is the Robot Operating System 2 (ROS 2) [1], which provides an extensible and modular infrastructure for designing, simulating, and deploying robotic systems across both academic and industrial environments. Within this technological context, serial manipulators are instrumental in applications requiring precise spatial interaction, such as pick-and-place operations, inspection tasks, accurate trajectory tracking, and human-robot collaboration. Their multiple degrees of freedom, inherent configurability, and compatibility with standard middleware like ROS 2 establish them as valuable platforms not only for established industrial automation but also for advanced research involving prototyping and algorithmic validation.

This project is situated within this dynamic field, specifically focusing on the workspace control of articulated manipulators. These robotic systems, designed to execute precise end-effector movements within a three-dimensional environment, are fundamental to a multitude of applications. The present study centers on the Niryo Ned2 robotic manipulator [2], a fixed-base articulated system chosen for its relevance to research and educational applications. A key challenge in the workspace control of such manipulators, particularly in partially known or dynamic contexts, lies in the accurate and robust real-time estimation of the end-effector's pose.

Concurrently, it is important to recognize that many established motion planning approaches, including those commonly integrated into widely used frameworks like MoveIt2 [3] (which often employ planners such as OMPL or CHOMP), operate primarily in the joint space. While effective for numerous scenarios, this characteristic can result in less direct or reactive control for tasks demanding precise and intrinsically adaptive end-effector interaction with the workspace. The conventional reliance on kinematic models and internal joint encoders for pose estimation can introduce further limitations due to model inaccuracies or not calibrated parameters, thereby necessitating the use of external sensor modalities for richer environmental perception and more accurate state awareness. The CHOMP algorithm (Covariant Hamiltonian Optimization for Motion Planning) formulates motion planning as a gradient-based trajectory optimization problem, enabling smooth and collision-free paths for high-DOF manipulators [4]. OMPL (Open Motion Planning Library) provides a collection of sampling-based planners (e.g., PRM, RRT, KPIECE) and benchmarks for motion planning, integrated within ROS frameworks such as MoveIt [5].

This thesis work naturally extends the activities undertaken during a foundational curricular intern-

ship [6]. That prior work successfully addressed the integration of the Niryo Ned2 manipulator into the ROS 2 environment and its configuration with the MoveIt2 framework. A complete system for trajectory planning was developed, culminating in the implementation of a simulated pick-and-place operation. This implementation utilized the CHOMP planner, which, due to its joint-space operational nature, required the application of Inverse Kinematics (IK) to translate workspace objectives into joint-space commands. The experience gained, and particularly the observed limitations concerning joint-space planning and the less direct management of dynamic tasks within the workspace, laid the groundwork and provided the primary motivation for the research and developments pursued in this thesis. This aligns with the perspective, highlighted within the internship report, of developing a custom controller to achieve more refined and adaptive robot control, thereby overcoming the limitations associated with standard controllers typically provided with MoveIt2.

Experimental activities are centered on the validation of this integrated approach through comprehensive simulations conducted within the ROS 2/Gazebo environment. The efficacy of the developed system is evaluated by analyzing its accuracy in tracking various predefined end-effector trajectories. This involves quantifying the error between the desired path and the one actually executed by the manipulator, based on the pose information provided by the SLAM subsystem. The results obtained confirm the technical soundness of the approach: the controller demonstrates the ability to follow complex 3D trajectories with generally low pose tracking errors, despite the absence of realistic feedforward velocity and acceleration terms. Notably, discrepancies emerge in the final segments of the path due to accumulated drift in SLAM-based pose estimation, as well as dynamic limitations of the robot model. Nonetheless, the utilization of SLAM for closing the control loop represents a meaningful advancement toward more intelligent control strategies, enabling robotic systems to operate effectively even in partially unknown or changing environments.

Building upon the current work, several avenues for future investigation are envisaged. One key direction involves enhancing the controller's robustness by incorporating realistic feedforward terms for velocity and acceleration, which are currently set to zero. Furthermore, strategies for compensating visual drift, such as error correction via loop closure or integration with inertial data, may improve long-term pose accuracy. Another promising enhancement concerns the interpolation of reference trajectories, e.g., using LERP, SLERP, or spline-based methods—to generate smoother motion commands and reduce the discontinuities associated with waypoint-based control. Finally, transitioning from simulation to real-world experiments on the physical Niryo Ned2 platform will be essential to assess the system's practical viability and performance under real sensor noise, actuation delays, and mechanical constraints.

1.1 Joint space and operational space control

Controlling a robotic manipulator to perform tasks effectively and accurately is a cornerstone of robotics research and application. The methodologies for achieving this control can be broadly categorized based on the domain in which the control objectives and actions are defined: joint space or operational space (also known as task space or Cartesian space). Understanding the distinctions between these two approaches is crucial for designing appropriate control strategies tailored to specific tasks and robotic systems [7].

1.1.1 Joint space control

Joint space control refers to techniques where the control law is formulated based on the manipulator's joint variables, typically the angles for revolute joints or displacements for prismatic joints. The desired

motion is specified as a trajectory of joint positions, velocities, and accelerations. The primary objective is to drive the manipulator's joints to follow these desired joint-space trajectories. The joint space control problem is actually articulated in two sub-problems. First, manipulator inverse kinematics is solved to transform the motion requirements x_d from the operational space into the corresponding motion q_d in the joint space. Then, a joint space control scheme is designed that allows the actual motion q to track the reference inputs

In the joint space control paradigm, control laws are formulated to act directly upon the manipulator's articulated variables, such as the angular positions of its joints. Classic controllers, including Proportional-Integral-Derivative (PID) regulators, or more advanced model-based schemes, directly command the actuators of each joint. Any task initially defined in terms of the end-effector's desired pose in Cartesian space must first be translated into a corresponding joint space trajectory by solving the inverse kinematics problem. This conversion, which is a critical step, is typically performed offline or at a higher planning level before the real-time control loop begins [8, p. 304].

This approach offers several significant advantages, primarily stemming from its direct correspondence with the robot's physical configuration. Control signals are directly related to the state of the joints, which simplifies the command of individual actuators and allows for the straightforward incorporation and monitoring of physical joint position, velocity, and torque limits. In certain scenarios, particularly if a decentralized control strategy is adopted or if the full robot dynamics are explicitly modeled in joint coordinates, the controller design itself can be more direct. Furthermore, if the task is defined entirely within joint space or if kinematic singularities are handled at the trajectory planning stage, the controller can operate without directly encountering these problematic configurations.

However, these benefits are counterbalanced by notable drawbacks, especially in applications requiring precise interaction with the external environment. Defining and visualizing tasks such as following a straight line on a surface or inserting a component is less intuitive, as it requires a computational or mental mapping to specific joint configurations. The accuracy of the end-effector's pose is critically dependent on the precision of the robot's kinematic model used for these transformations; any error in the model parameters, such as link lengths or joint offsets, translates directly into a positioning error in the workspace. This paradigm also makes it difficult to directly specify or control the forces exerted by the end-effector or to implement complex behaviors like impedance control without intricate transformations of these requirements back into the joint space. Finally, trajectory planning can present its own complexities: planners like OMPL or CHOMP, often used within frameworks like MoveIt2, generate joint-space trajectories that may not always result in optimal or intuitive Cartesian paths, a challenge highlighted by the need for explicit IK during the preceding internship work [6].

1.1.2 Operational space control

In contrast, operational space control (OSC) (or task space control) formulates the control problem directly in terms of the end-effector's behaviour in its Cartesian workspace. The desired motion is specified as a trajectory of the end-effector's position and orientation (pose), and their time derivatives (velocity, acceleration) in a chosen task frame. The operational space control problem follows a global approach that requires a greater algorithmic complexity; notice that inverse kinematics is now embedded into the feedback control loop. Its conceptual advantage regards the possibility of acting directly on operational space variables; this is somewhat only a potential advantage, since measurement of operational space variables is often performed not directly, but through the evaluation of direct kinematics functions starting from measured joint space variables [8, p. 305].

This approach relies on the manipulator's Jacobian matrix, $J(q)$, which provides the differential relationship between joint velocities $(\dot{q})$ and end-effector velocities $(\dot{x})$. Control commands, initially computed in the operational space (e.g., desired forces or accelerations for the end-effector), are then mapped back to the joint space, typically as joint torques (τ)). This mapping utilizes the Jacobian, often its transpose $J(q)^\top$ or a pseudo-inverse $J(q)^+$, and for high-performance applications, may also incorporate the robot's dynamic model. The pioneering work by Takegaki and Arimoto [9] first introduced such a feedback method for dynamic control directly in the task space, highlighting its applicability even in the presence of singularities or redundancy.

The primary advantage of this paradigm is its intuitive nature for tasks that are inherently defined in Cartesian space, such as welding, drawing, or assembly, making their planning and programming more straightforward. This approach allows for the direct specification and control of end-effector dynamics, including forces, torques, and impedance, which is crucial for tasks involving physical interaction or contact with uncertain environments. Consequently, by focusing the control effort on the task variables, it becomes possible to achieve higher precision in the end-effector's behaviour. This potential for greater accuracy is particularly pronounced when feedback is derived directly from the operational space, for instance through vision or force sensors, which is a central motivation for this thesis's objective to use VSLAM for operational space feedback. Furthermore, it is often possible to design controllers where the dynamic behaviour of the end-effector in different Cartesian directions is decoupled, which can simplify the controller tuning process.

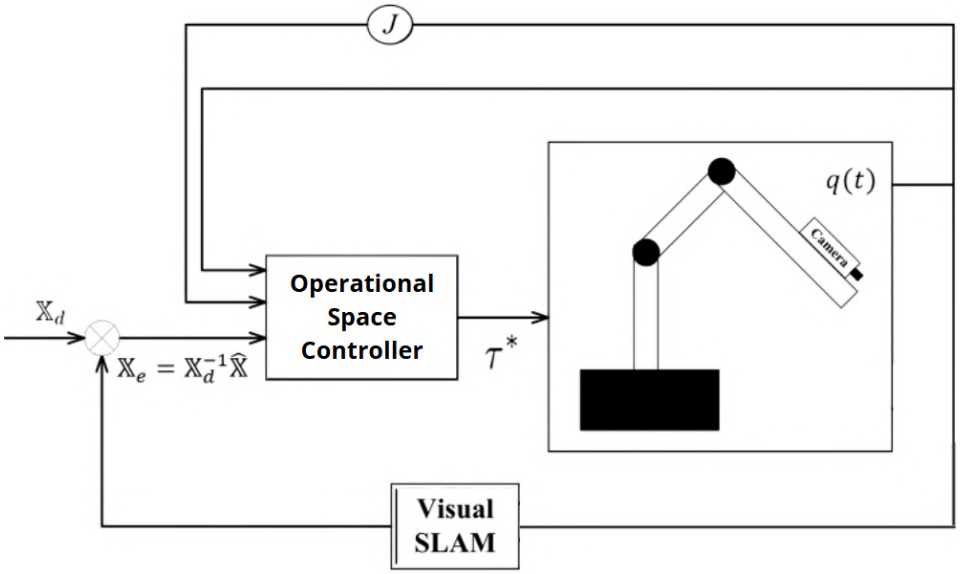


Figure 1.1: Block diagram of a SLAM feedback operational space control.

However, the benefits of operational space control are accompanied by several challenges. The control laws can be computationally more intensive due to the real-time requirement of calculating the Jacobian, and potentially its derivatives or inverse, along with the robot's operational space inertia matrix. Moreover, the manipulator is susceptible to kinematic singularities, which are configurations where the Jacobian loses rank. At or near these points, certain end-effector motions may become impossible or require excessively high joint velocities, leading to control instability. For high-performance control, especially methods involving acceleration or precise force regulation (such as computed torque control), a significant dependence on an accurate dynamic model of the manipulator arises. The transformation from desired

operational space forces to the necessary joint torques is non-trivial and requires careful consideration of the robot's full dynamic properties.

1.2 Trajectory tracking

A fundamental objective in robot control is to command the manipulator to move in a desired manner. This objective can be broadly divided into two categories: point-to-point control (or regulation) and trajectory tracking. While regulation focuses on moving the end-effector from an initial pose to a static final pose, trajectory tracking is concerned with a more dynamic task: ensuring that the end-effector follows a predefined, time-varying path in its workspace as closely as possible. This desired path, which specifies the end-effector's pose at every instant in time, is known as the trajectory [10].

The ability to accurately track trajectories is essential for a vast range of robotic applications, including automated welding, painting, inspection along a surface, gluing, or any task that requires the robot to execute a precise motion path rather than simply reaching a destination. The control problem in trajectory tracking is to design a control law, typically for the joint torques (τ)), that minimizes the tracking error, the difference between the desired end-effector pose $X_d(t)$ and the actual pose $X(t)$, at all times. This is generally more challenging than simple regulation because the controller must continuously compensate for both dynamic effects and path deviations in real time.

High-performance tracking controllers are often structured with two primary components: a feed-forward term and a feedback term. The feed-forward component utilizes the robot's dynamic model and the desired trajectory information (e.g., desired velocities, $\dot{X}_d$, and accelerations, $\ddot{X}_d$) to proactively calculate and apply the nominal torques required to follow the path. In an ideal scenario with a perfect model and no disturbances, this component alone could achieve perfect tracking. The feedback component, on the other hand, is responsible for correction; it measures the tracking error and applies corrective actions to drive it towards zero, ensuring stability and robustness.

The effectiveness of this feedback action is critically dependent on the quality and accuracy of the actual state measurement, $X(t)$. However, obtaining a perfect state measurement is a significant challenge. State estimation can rely on proprioceptive sensors, such as joint encoders, where the end-effector pose is calculated via the robot's forward kinematics. Any inaccuracies in the kinematic model limit this approach; errors in parameters, such as link lengths, directly lead to errors in the estimated end-effector pose. Alternatively, state estimation can use exteroceptive sensors, like cameras or laser trackers, which observe the end-effector directly in its workspace. While potentially more accurate, these perception-based systems introduce their own challenges, including sensor noise, processing delays, and estimation drift over time. These inherent difficulties in achieving perfect tracking, arising from model uncertainties, external disturbances, and state estimation imperfections, motivate the ongoing development of advanced control strategies that aim to enhance robustness and precision.

1.3 Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is a fundamental computational problem in robotics and autonomous systems. It addresses the challenge of a robot navigating an unknown environment while concurrently building a map of that environment and tracking its own position within it. This capability is essential for enabling autonomy in a wide range of applications, from self-driving cars to domestic service robots, particularly in scenarios where external positioning systems like GPS are unavailable or unreliable, such as indoor environments.

The SLAM problem is often described as a "chicken-and-egg" dilemma: accurate localization requires a pre-existing map, while accurate mapping demands precise localization. This circular dependency is further complicated by the presence of sensor noise and drift, errors in pose estimation that accumulate over time. To address this, most SLAM systems incorporate Loop Closure Detection (LCD) mechanisms, which identify when the robot revisits a previously mapped location. Upon successful loop closure, global optimization can be applied to adjust the entire trajectory and map, thereby correcting accumulated drift and ensuring long-term consistency [11].

Over the years, two main classes of approaches have emerged to solve the SLAM problem: filter-based and optimization-based methods. Filter-based approaches, such as those using the Extended Kalman Filter (EKF) or Particle Filters (as in FastSLAM), treat SLAM as an online state estimation problem, recursively updating the probability distribution of the current robot pose and map as new measurements become available. Optimization-based approaches, also known as smoothing or graph-based SLAM, have become more dominant in modern systems. These methods formulate the problem as a graph where nodes represent robot poses and environmental landmarks, and edges represent spatial constraints derived from sensor measurements. The goal is then to find the configuration of all poses and landmarks that best explains all the measurements, typically by minimizing a global error function. Many modern SLAM solutions adopt a common framework that separates the process into a front-end and a back-end. The front-end is sensor-dependent and responsible for processing raw sensor data to estimate odometry and detect potential loop closures, while the back-end is sensor-agnostic and performs the graph optimization to ensure global consistency.

SLAM systems are also often categorized by their primary sensor modality. LiDAR SLAM uses laser range finders to build precise 2D or 3D point cloud maps of the environment. Visual SLAM (VSLAM), on the other hand, utilizes one or more cameras as its main sensor. VSLAM algorithms process image data to extract features or track pixel intensities to estimate the robot's motion and reconstruct the scene's structure. Depending on the camera setup, VSLAM can be further categorized as monocular (single camera), stereo (two cameras), or RGB-D (using a camera that provides both color and depth information). Numerous implementations of these SLAM algorithms are available within the ROS 2 ecosystem, facilitating their integration into complex robotic applications.

1.3.1 SLAM for fixed base manipulators

While SLAM is traditionally associated with mobile robots, recent research has demonstrated its applicability in scenarios involving fixed-base robotic manipulators, especially those operating in unstructured or partially known environments. In these contexts, SLAM is not employed for global navigation, but rather to enhance the robot's environmental awareness, enabling capabilities such as obstacle avoidance, object pose estimation, and adaptive manipulation in dynamic settings [12].

A particularly relevant use case arises when a depth or RGB-D camera is rigidly mounted on the end-effector of the manipulator. In this setup, the camera's motion through the workspace, induced by the manipulator's kinematics, generates a sequence of sensor observations that can be processed by a SLAM system. This allows the robot to incrementally build a local 3D map of its surroundings, which can be used for high-level planning, semantic scene understanding, or feedback control.

In this thesis, a SLAM-based visual perception pipeline is implemented using an RGB-D camera mounted on the wrist-link of a Niryo Ned2 manipulator (due to the robot's design, positioning the camera on the wrist-link rather than the end-effector makes no difference.). The SLAM system processes the camera stream to construct a dense point cloud map and estimate the motion of the camera over time.

Although the base of the manipulator is fixed, the articulated motion of the arm effectively provides the parallax and viewpoint changes required by SLAM algorithms to reconstruct the environment. This mapping process complements the robot's kinematic model by offering a redundant and sensor-driven estimate of the end-effector pose, useful for tasks such as visual servoing or workspace exploration in partially known scenes.

1.4 3D Mapping with RGB-D sensors

Accurate 3D mapping is a critical component for enabling autonomous interaction in robotic systems. This section introduces and explains the key technologies used in this thesis for volumetric map construction, including the OctoMap framework and RGB-D sensors. The integration of these tools enables both real-time perception and reliable motion planning in dynamic environments.

1.4.1 OctoMap: Probabilistic 3D mapping framework

OctoMap [13] is a widely adopted framework for probabilistic 3D mapping in robotics. It provides an efficient and flexible representation of volumetric environments using an octree-based data structure. Unlike traditional 2D occupancy grids, OctoMap enables the representation of free, occupied, and unknown volumes in three-dimensional space, which is essential for applications such as autonomous navigation, object manipulation, and obstacle avoidance in complex and unstructured environments.

The core idea of OctoMap is to model the environment as a hierarchical tree of cubic volumes, called voxels, each representing a fixed-size region of space. Each voxel stores a probabilistic estimate of occupancy, which is updated incrementally using Bayesian filtering. This probabilistic formulation allows the map to handle noisy sensor inputs, such as those from stereo cameras or depth sensors, in a robust and consistent manner.

OctoMap's hierarchical octree structure offers significant advantages in terms of memory efficiency and scalability. It allows the map resolution to adapt dynamically: fine details are stored only where needed, while large free or empty regions are coarsely represented. This makes OctoMap suitable for large-scale mapping tasks, especially in environments where the distribution of relevant geometry is sparse.

In robotic systems, OctoMap plays a central role in integrating data from depth sensors, particularly RGB-D cameras, into a coherent 3D model of the robot's surroundings. When used in conjunction with SLAM algorithms, such as RTAB-Map, the octomap is built incrementally as the robot explores the environment, providing a continuously updated representation that supports localization, planning, and interaction tasks.

1.4.2 RGB-D sensors in robotic systems

RGB-D sensors, such as the Intel RealSense or Microsoft Azure Kinect, provide synchronized color (RGB) and depth (D) images by projecting structured light or using time-of-flight technology. These sensors have become increasingly popular in robotics due to their ability to deliver dense and real-time 3D information at relatively low cost.

RGB-D cameras differ fundamentally from standard monocular sensors by capturing both photometric and geometric information in a single frame. The difference is in the content of the image data. Each pixel is stored as a 32-bit float or sometimes as a 16-bit unsigned integer. If it's a float, the values represent metres, and if it's an int then they are millimeters [Figure 1.2].

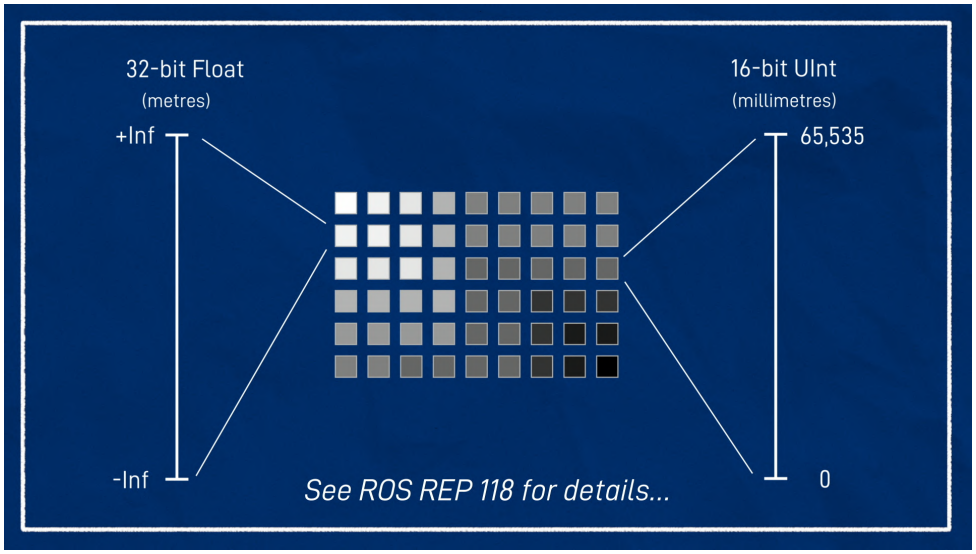


Figure 1.2: Depth image pixel explanation

This dual sensing capability enables robots to perceive not only the visual appearance of the environment but also its spatial structure. In robotics, this advantage is crucial for tasks requiring 3D awareness, such as object detection, mapping, scene understanding, and obstacle avoidance [14].

In ROS 2, RGB-D sensors are typically interfaced through standardized topic structures. Depth images are often published on topics such as `/camera/depth/image_raw`, RGB images on `/camera/color/image_raw`, and 3D point clouds on `/camera/depth/points`, using message types like `sensor_msgs/Image` and `sensor_msgs/PointCloud2`. These data streams can be consumed by perception and mapping frameworks such as RTAB-Map, MoveIt 2, and OctoMap [15].

In mobile and manipulation scenarios, RGB-D sensors are widely used for obstacle detection, object recognition, 3D reconstruction, and SLAM. The depth images are typically converted into point clouds, which are then fused into 3D maps using probabilistic occupancy representations like OctoMap, or directly integrated by graph-based SLAM. The combination of visual (RGB) and spatial (depth) data allows for richer perception compared to monocular or stereo vision alone.

In this thesis, an RGB-D camera is mounted on the end-effector of the Niryo Ned2 manipulator within the Gazebo simulation environment, rather than using a physical sensor. This setup enables real-time perception of the robot's surroundings from a task-relevant viewpoint. The sensor is configured using the `gazebo_ros_depth_camera` plugin, which emulates depth sensing by publishing RGB, depth, and point cloud data over the standard ROS 2 interfaces. The resulting point clouds are used to incrementally construct a 3D map via OctoMap, which is then shared with the MoveIt 2 planning framework for collision checking and motion planning.

1.5 Pose estimation

Accurate pose estimation is essential in robotic manipulation, enabling precise interaction with objects and the surrounding environment. In this thesis, the end-effector's pose is estimated using a hybrid approach that combines kinematic data from the Niryo Ned2 robot with visual feedback from an RGB-D camera mounted on its tool link. While forward kinematics provides reliable pose estimates in static or well-calibrated settings, visual sensing enhances performance in dynamic or partially unknown environments,

particularly when occlusions or modeling errors occur.

In this research, visual pose estimation through SLAM has been considered as a complementary mechanism to the robot's kinematic pose. However, experimental findings revealed that SLAM-based estimates exhibit drift and less precision compared to those derived from the robot's joints and encoders. Therefore, the control architecture mainly relies on TF-based kinematic transformations for real-time end-effector pose. Nonetheless, visual pose estimation serves valuable roles in mapping validation, environment interpretation, and performance benchmarking. RGB-D cameras support both visual odometry (VO) and SLAM techniques by leveraging combined color and depth data. Research shows that RGB-D odometry methods outperform monocular or stereo-only approaches due to their access to depth-derived scale information and robustness to varying environments [16] [17].

Additionally, RGB-D–based visual pose estimation supports advanced techniques such as visual servoing, where the planned trajectory is dynamically adjusted based on visual feedback, and object-relative positioning, increasing task adaptability in cluttered or changing workspaces [18]. Although primarily demonstrated here in simulation, this approach provides a foundation for future extensions to real-time visual loop closure and sensor-aided control on physical manipulators.

1.6 Tools

To develop the robotic system described in this thesis, some of the most widely used software tools in open-source robotics were adopted. This section provides an overview of the main technologies employed, highlighting their general purpose within a typical robotic architecture. Specifically, we introduce ROS 2 as a communication framework between modules, MoveIt for motion planning, RViz for real-time visualization, and Gazebo for physical simulation of the robot in virtual environments.

1.6.1 ROS 2

The development of complex robotic applications is greatly facilitated by the use of dedicated frameworks. The Robot Operating System (ROS) is a flexible and widely adopted open-source middleware framework that provides libraries, tools, and conventions to simplify the creation of robotic systems. It is based on a distributed computing model where software processes, called nodes, communicate with each other by exchanging messages. Messages, which are typed data structures, are routed via named buses called topics, using a publish/subscribe communication semantic. This architecture promotes a high degree of modularity and software component reusability [1].

ROS 2, the next generation of this framework, was redesigned from the ground up to meet the challenges set forth by modern robotic systems in new and exploratory domains. While the original ROS proved invaluable for research, it was not designed with many necessary production-grade features such as security and reliability in non-ideal network environments. ROS 2 was therefore conceived to address these limitations while building upon the community-driven success of its predecessor.

One of the most significant architectural changes is the abandonment of the centralized Master node. In ROS 1, the master was responsible for node registration and lookup, representing a potential single point of failure. In contrast, ROS 2 adopts a fully distributed peer-to-peer discovery mechanism, which improves the system's overall robustness and scalability [19]

From a communication standpoint, ROS 2 is based on the industrial standard Data Distribution Service (DDS), an open standard used in critical infrastructure such as military, aerospace, and financial systems. This choice enables ROS 2 to provide best-in-class security, support for embedded and real-time

systems, and robust multi-robot communication even over unreliable networks. The DDS middleware also introduces configurable Quality of Service (QoS) policies, which allow developers to fine-tune how data flows through the system, for instance by choosing between reliable delivery or best-effort performance for sensor data where newer messages quickly make older ones obsolete.

Finally, ROS 2 has expanded its cross-platform compatibility, offering native support not only for Linux but also for Windows, macOS, and Real-Time Operating Systems (RTOS). Thanks to these foundational improvements, which also include better support for multi-robot systems and small embedded devices, ROS 2 is establishing itself as the reference standard for the development of next-generation robotic applications, from academic research to industrial products.

ROS 2 is developed through a release cycle with named distributions, each with its own support schedule. Among these, the Long-Term Support (LTS) releases, such as Humble Hawksbill, are particularly significant for the development community. An LTS version is frequently chosen for long-term research and development projects due to its guaranteed stability, an extended support window (typically five years), and a mature and well-maintained ecosystem of software packages, thereby ensuring a reliable foundation for the integration of complex systems.

1.6.2 MoveIt

In the development of robotic manipulation applications, the ability to generate complex, safe, and collision-free movements is a fundamental necessity. *MoveIt* is a powerful and widely used open-source software framework within the ROS ecosystem, designed specifically to address the challenges of motion planning and manipulation. It is conceived as an integrated platform that provides a cohesive set of tools for kinematics, planning, control, 3D perception, and collision checking, significantly reducing the complexity for developers [3].

Originally developed for ROS 1 and later evolved into *MoveIt2* to leverage the architecture and features of ROS 2, the framework maintains its core principles. The primary objective of MoveIt is to provide a robot-agnostic platform, meaning it is capable of being configured for a wide variety of robotic manipulators.

The main capabilities of MoveIt include:

- **Motion Planning**: The core of the framework is its ability to generate trajectories for a robotic arm from a starting configuration to a goal configuration, avoiding collisions with itself (self-collisions) and with objects in the environment. To do this, it utilizes a plugin-based architecture that allows for the integration and use of various planning algorithms. The default implementation uses the OMPL, a vast collection of sampling-based planning algorithms, but can be extended to include other planners such as *CHOMP* or *SBPL*.

- **Kinematics Management**: MoveIt provides a unified interface for kinematic solvers. It handles both forward kinematics (calculating the end-effector pose from joint angles) and IK (calculating the necessary joint angles to reach a desired end-effector pose), the latter being essential for any planning task defined in Cartesian space.

- **Planning Scene**: A central concept in MoveIt is the *Planning Scene*, which is an internal, dynamically updated representation of the robot and the surrounding world, including obstacles detected by sensors. This scene is used for collision checking during the planning process.

- **Collision Checking**: MoveIt integrates high-performance libraries (such as the *Fast Collision Library*, FCL) to efficiently perform collision checks between the robot's links and between the robot and objects in the Planning Scene.

A key aspect that has contributed to the success and widespread adoption of MoveIt is the focus on lowering the barrier to entry for new users. This has been achieved primarily through the *MoveIt Setup Assistant*, a graphical user interface (GUI) that guides the user step-by-step through the process of configuring a new robot. Starting from a standard robot model (in URDF format), the Setup Assistant automatically generates all the necessary configuration files. Among these, the most important is the *Semantic Robot Description Format* (SRDF), a file that adds semantic information to the robot's physical model. It defines, for example, which groups of joints constitute an "arm" or an "end-effector," or which pairs of links can be ignored during self-collision checks to optimize performance.

In summary, MoveIt acts as a bridge between high-level commands (e.g., "Move the end-effector to this position") and the low-level control of the robot, providing a complete pipeline that greatly simplifies the development of complex, safe, and reliable manipulation applications.

1.6.3 RViz: 3D visualization tool for ROS

RViz (ROS Visualization) is an essential graphical tool in the ROS ecosystem, designed to visualize the state and perception of a robot in a 3D environment. It plays a crucial role in development, debugging, and demonstration by providing real-time feedback on a wide variety of sensor data, robot states, and planning outputs [20].

Originally developed for ROS 1 and later adapted for ROS 2, RViz allows users to visualize robot models described in URDF, display sensor data such as point clouds, laser scans, and camera images, and monitor coordinate frames through the TF transform system. Its modular architecture supports a wide variety of display types, each customizable through a graphical user interface.

RViz is especially valuable in applications involving motion planning and perception, as it allows the developer to inspect planned trajectories, visualize collisions, and understand how the robot perceives its environment. For example, in motion planning frameworks such as MoveIt, RViz is tightly integrated to show the planning scene, visualize inverse kinematics results, and animate planned trajectories before execution.

In the context of this work, RViz serves as a critical tool for verifying the correct configuration of the robot model, evaluating SLAM outputs (such as the generated 3D map and estimated trajectory), and monitoring the execution of trajectories during operational space control. Its real-time, visual feedback accelerates development and helps ensure system correctness throughout the robotic application pipeline.

1.6.4 Gazebo simulation environment

In contemporary robotics research and development, the capability to test, validate, and iterate algorithms in a safe, repeatable, and cost-effective environment is of paramount importance. *Gazebo* is an open-source 3D robotics simulator specifically designed to fulfill this need. It enables accurate simulation of entire robotic systems, including sensors, actuators, and their interactions with complex environments, making it an indispensable tool for prototyping, testing, and system validation [21].

Gazebo offers a high-fidelity physics-based simulation environment by supporting multiple physics engines, such as ODE, Bullet, DART, and Simbody, which model realistic phenomena including gravity, friction, contacts, and collisions. A significant strength of the simulator is its extensive support for virtual

sensors, such as RGB-D cameras, LiDARs, IMUs, GPS, and contact sensors. These sensors generate realistic data streams that can be directly consumed by perception pipelines, enabling tasks like SLAM, obstacle detection, and sensor fusion to be validated entirely in simulation before deployment to physical hardware.

Gazebo's flexibility and extensibility are key enablers in advanced robotic applications. For example, NASA developed specialized plugins within Gazebo to replicate extraterrestrial terrain and lighting conditions for simulating the VIPER rover's lunar mission scenarios [22]. Such adaptability makes Gazebo suitable not only for ground robots, but also for aerial and underwater platforms, extending its utility across domains.

In the development lifecycle, Gazebo facilitates full-stack integration testing by simulating both low-level hardware interfaces and high-level autonomy algorithms. Moreover, Gazebo is tightly integrated with the ROS 2 middleware through the `gazebo_ros2` and `gazebo_ros2_control` plugins. These plugins provide bridges that expose simulated sensors and actuators as standard ROS 2 topics, services, and actions. Developers can write, test, and debug ROS 2 nodes in simulation using the exact same interfaces as would be used on real hardware. This continuity allows seamless transition from simulation to deployment, significantly shortening development cycles and improving system reliability.

1.7 Niryo Ned2

The robotic platform used in this thesis is the *Niryo Ned2* (Figure 1.3), a 6-DOF collaborative robotic manipulator designed for educational and research applications. The Niryo Ned2 offers a compact and modular architecture based on stepper motors, position encoders, and interchangeable end-effectors, making it well-suited for experimentation in tabletop manipulation tasks such as object grasping, sorting, and pick-and-place.

Figure 1.3: Niryo Ned2 robotic arm.

Originally developed with a software stack based on ROS 1, the Ned2 does not offer native support for ROS 2, which is increasingly adopted in modern robotic systems due to its improved performance, modularity, and support for real-time applications. As part of the preliminary work for this thesis, the

ROS 1 packages provided by Niryo were ported to ROS 2 (Humble distribution), including the robot description, control interfaces, and MoveIt 2 configuration. This porting enabled full integration of the Niryo Ned2 within the ROS 2 ecosystem, allowing for advanced features such as motion planning, trajectory execution, and real-time perception in simulation.

Despite its flexibility, the Niryo Ned2 has inherent limitations that must be considered in advanced control tasks. These include the use of stepper motors without torque sensing, relatively low joint actuation bandwidth, and the lack of native integration with ROS 2 and physics simulators like Gazebo. Consequently, the system was adapted and extended to support torque-mode control in simulation, a key requirement for the implementation of the proposed operational space controller.

1.8 Thesis objectives

The primary goal of this thesis is to develop and validate an advanced operational space control system for the Niryo Ned2 manipulator. This system is conceptualized to be inherently more dynamic and adaptive by leveraging real-time end-effector pose estimation derived from Visual Simultaneous Localization and Mapping (VSLAM) techniques, thereby addressing and aiming to overcome the identified limitations of traditional joint-space planning and control paradigms.

To achieve this goal, the core methodology of this thesis involves the synergistic integration of several key technologies. The proposed system architecture incorporates an RGB-D camera for rich sensory input from the simulated environment (Gazebo), a VSLAM module (specifically, RTAB-Map [23]) for real-time estimation of the end-effector's spatial pose, and a custom-designed operational space controller. This controller, implemented as a ROS 2 plugin, processes the desired trajectory and the SLAM-derived pose feedback to generate appropriate torque commands τ for the manipulator's joints, effectively closing the control loop through visual perception. The fundamental principles behind each of these components, the operating space control, the VSLAM with RGB-D cameras and their integration into the ROS 2 and Gazebo ecosystem, have been explored in depth in the previous sections of this chapter to provide the necessary basic understanding.

1.9 Thesis outline

This thesis is organized into the following chapters, in order to present the context, development, and research results in a structured manner:

- **Chapter 1 – Introduction:** Introduces the research context and motivation behind combining perception, mapping, and controlling in robotic manipulation. The chapter outlines the objectives of the thesis and provides a high-level overview of the methodology.
- **Chapter 2 – State of the art:** Reviews relevant literature and existing approaches in the fields of SLAM, RGB-D perception, pose estimation and and various approaches to operational space control. Particular attention is given to the positioning of the proposed method with respect to existing pose estimation and control strategies. It includes a comparative analysis of SLAM algorithms available in ROS 2, highlighting the reasons for selecting RTAB-Map for this work.
- **Chapter 3 – System architecture:** Defines the specific research challenges addressed in this thesis and presents a high-level architectural view of the proposed system. It details the structure of the

software modules, the communication interfaces between ROS 2 nodes, and the overall data flow between the sensing, mapping, planning, and control components. This chapter formalizes the problem of enabling a fixed-base robotic manipulator to sense and interact with its surroundings and perform trajectory tracking.

- **Chapter 4 – System implementation:** Provides a detailed account of the system implementation. It describes the simulation environment in Gazebo, the configuration of the RGB-D sensor mounted on the wrist-link, and the integration of the RTAB-Map SLAM pipeline with OctoMap generation and MoveIt 2. Special attention is given to the design and development of the operational space controller, implemented as a custom ROS 2 plugin. The choice of using C++ over Python is motivated by the need for real-time performance, low-latency communication, and seamless integration with the ROS 2 control and plugin interfaces.
- **Chapter 5 – Simulations:** Reports the experimental results obtained in a simulated Gazebo world evaluating the quality of the control and therefore of the pursuit of the pre-established trajectory.
- **Chapter 6 – Conclusions:** Summarizes the contributions of the thesis, reflects on the limitations of the proposed system, and discusses future improvements.

Chapter 2

State of the art

This chapter presents a comprehensive overview of the current state of research and development in the fields of SLAM, RGB-D sensing for robotic perception, pose estimation techniques, and operational space control strategies. The objective is to situate the methodology adopted in this thesis within the broader scientific landscape and justify the technical decisions made throughout the system design.

2.1 Simultaneous Localization and Mapping

SLAM is a broad field, and its various implementations are often categorized based on the primary sensor modality employed. The choice of sensor fundamentally shapes the algorithm's characteristics, defining its strengths, weaknesses, and ideal applications. The main categories in the literature include LiDAR-based SLAM and various forms of vision-based SLAM (VSLAM).

Lidar-based SLAM This category of SLAM utilizes laser scanners (LiDAR) to obtain direct and precise distance measurements of the surrounding environment. By emitting laser beams and measuring the time of flight for their reflections, these systems generate accurate 2D or 3D point cloud maps. The primary advantage of LiDAR SLAM is its high accuracy and robustness to varying lighting conditions, making it a popular choice for autonomous navigation, especially in large-scale outdoor environments. However, LiDAR-based systems can be more computationally expensive than their visual counterparts and may struggle in geometrically feature-poor environments, such as long, uniform corridors. A relevant example is **Hector SLAM** [24], which uses an approach based on the transformation estimation between successive 2D Lidar scans, exploiting the Gauss-Newton algorithm to minimize the error between local and global maps. This method is effective even without odometry, and is successfully used in indoor environments for UAVs and mobile robots. **LOAM** (Lidar Odometry and Mapping) and its extension **LIO-SAM** [25] integrate Lidar-based odometry with inertial information from an IMU. LIO-SAM, in particular, builds an incremental 3D map and continuously optimizes the trajectory using factors from Lidar, IMU and loop closure. It is one of the most accurate methods for large-scale environments, but it is mainly designed for mobile robots and ground vehicles.

However, these methods require expensive sensors and do not provide semantic information about the scene, limiting their use in indoor robotic scenarios where a fixed manipulator is present.

Monocular Visual SLAM: Monocular SLAM systems use a single RGB camera to extract visual features from the environment and estimate the map and motion using Structure-from-Motion (SfM)

techniques. The main advantage is hardware simplicity, but ambiguity in scale is an intrinsic limitation. **ORB-SLAM2** [26] and **ORB-SLAM3** [27] are among the most advanced monocular systems. ORB-SLAM2 uses ORB keypoints to perform tracking, local mapping and pose-graph optimization. ORB-SLAM3 extends the approach to multi-map scenarios and supports stereo and RGB-D configurations. Both methods are able to provide sparse maps, but suffer from *scale drift* in the absence of metric information.

These algorithms are widely used in mobile robotics, but are not ideal for manipulating robots where scale estimation is critical for precise operations.

Stereo and RGB-D SLAM: Stereo SLAM systems resolve scale ambiguity by using two calibrated cameras to estimate depth. Alternatively, **RGB-D SLAM** systems rely on depth cameras (e.g. Kinect, RealSense) that directly provide dense depth maps, enabling accurate reconstruction of the environment. **ElasticFusion** [28] is a real-time SLAM system that uses *surfels* (surface elements) representation to dynamically update the map as the sensor moves. It uses a pose-graphless approach, but continuously optimizes the global map with topological deformation techniques. **RTAB-Map** (Real-Time Appearance-Based Mapping) [23] is an RGB-D SLAM framework that uses a graph structure to handle incremental map growth. It integrates visual loop closure methods with BRIEF/BRIEF-GRID or SURF descriptors, allows dense or sparse mapping, and is compatible with OctoMap for generating 3D voxel maps. It also supports stereo and Lidar input, but stands out for its multi-threaded optimization and ROS 2 compatibility.

This type of SLAM is particularly suitable for indoor robot manipulators equipped with RGB-D cameras mounted on the end-effector, since it provides metrically consistent estimates and useful maps for planning.

Multi-sensor and hybrid approaches: The latest solutions combine multiple sensors (e.g. camera, IMU, Lidar) to increase robustness and reduce drift. **VINS-Fusion** [29] integrates monocular vision and inertial data to estimate trajectory in dynamic environments. The system leverages IMU pre-integration, pose-graph, and bundle adjustment optimization to provide consistent estimates even in the presence of sparse texture. **Cartographer** by Google is another SLAM solution that combines Lidar and IMU to build 2D and 3D maps. Its modular architecture supports local maps, global optimization, and loop closure, but its integration in ROS 2 is still partial compared to ROS 1.

These approaches are particularly effective in dynamic scenarios or on mobile platforms, but are less practical to integrate in fixed manipulators due to hardware and computational complexity.

2.1.1 SLAM technique used in this work

In the context of this thesis, the Niryo Ned2 robot is a fixed-base manipulator equipped with an RGB-D camera mounted on the end-effector. In such scenarios, vision-based SLAM (RGB-D SLAM) methods represent the most suitable choice since they allow reconstructing three-dimensional environments starting from visual information only without the need for LIDAR or mobile-based odometry. Among the algorithms available in ROS 2, **RTAB-Map (Real-Time Appearance-Based Mapping)** has been selected for several key reasons:

- Is one of the few visual SLAM algorithms fully compatible with ROS 2 and consistently maintained;
- Natively supports RGB-D cameras and integrates a visual odometry, loop closure and map optimization pipeline;
- allows the generation of 3D maps via OctoMap or voxel grids, essential for arm motion planning;

- Is particularly suitable for manipulating robots, as it allows to provide an external pose estimate (e.g. from TF) when available, and builds the map by observing the environment from the perspective of the robotic arm.

These features make RTAB-Map an effective and efficient solution for the goal of enabling environment perception and end-effector pose estimation in partially known contexts, such as the one treated in this thesis.

2.2 Operational Space Control

Operational Space Control refers to a class of control strategies where the robot manipulator is controlled directly in the task or operational space, typically defined by the pose or position of the end-effector, rather than in joint space. This approach allows for intuitive specification of tasks and facilitates the design of controllers that consider the dynamics and kinematics of the manipulator within the space where the task is executed.

2.2.1 Classical formulation (Khatib-style control)

The foundational work by Khatib [30] formalized the control of robot manipulators in operational space, highlighting the importance of the manipulator Jacobian and dynamics. The Jacobian matrix $J(q)$ maps joint velocities $\dot{q} \in \mathbb{R}^n$ to end-effector velocities $v \in \mathbb{R}^m$:

$$v = J(q)\dot{q} \tag{2.1}$$

where q represents the vector of joint positions.

The dynamic model of an n-degree-of-freedom manipulator can be written as:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau + \tau_{ext} \tag{2.2}$$

where $M(q)$ is the inertia matrix, $C(q,\dot{q})$ represents Coriolis and centrifugal effects, $g(q)$ the gravity vector, τ the joint control torques, and τ_{ext} external torques (e.g., from interaction forces).

Khatib showed that by using the dynamically consistent generalized inverse of the Jacobian J^*, the dynamics can be projected into operational space as:

$$\Lambda(x)\ddot{x} + \mu(x,\dot{x}) + p(x) = F + F_{ext} \tag{2.3}$$

where $\Lambda(x)$ is the operational space inertia matrix, $\mu(x,\dot{x})$ the centrifugal and Coriolis terms in operational space, $p(x)$ the gravity vector, F the control wrench applied at the end-effector, and F_{ext} external forces.

This formulation enables the design of control laws directly in operational space, for example by defining a desired end-effector trajectory $x_d(t)$ and computing F to track it.

2.2.2 Interaction control: impedance and admittance

In tasks involving physical interaction with the environment, it is crucial to regulate the dynamic relationship between the end-effector motion and the interaction forces. Two widely used frameworks are impedance and admittance control:

- **Impedance control** [31] aims to regulate the dynamic relationship from end-effector position to applied force, effectively making the robot behave like a mass-spring-damper system. The control law can be expressed as:

$$F = M_d(\ddot{x}_d - \ddot{x}) + D_d(\dot{x}_d - \dot{x}) + K_d(x_d - x) \tag{2.4}$$

  where M_d, D_d, and K_d are the desired inertia, damping, and stiffness matrices, respectively.

- **Admittance control** [32] instead regulates the motion in response to external forces, effectively inverting the impedance relationship. It is often used when the robot is position-controlled but must respond compliantly to external forces.

These interaction controllers extend OSC by incorporating environment compliance, enabling safe and robust manipulation in unstructured or partially known environments.

2.2.3 The controller used in this work: the role of Inverse Dynamics

Model-based operational space controllers leverage the robot's dynamic model to achieve improved tracking performance and system linearization. One prominent example is the Computed Torque Control (CTC), which cancels nonlinear dynamics by computing feedforward torques based on inverse dynamics:

$$\tau = M(q)v + C(q,\dot{q})\dot{q} + g(q) \tag{2.5}$$

where v is a new control input, often designed as a PD controller in joint or operational space.

Within OSC, inverse dynamics allows disassociating the control problem along each degree of freedom, effectively linearizing the system and simplifying the design of stable and performant controllers.

In summary, these foundational methodologies provide the theoretical basis for controlling robot manipulators in operational space, offering both trajectory tracking and interaction capabilities. The choice of a particular approach depends on task requirements, system dynamics, and the available sensory feedback.

2.3 Pose estimation and visual feedback

For any operational-space controller, the availability of an accurate and timely estimate of the end-effector pose $\mathbf{X}(t) \in \mathrm{SE}(3)$ is crucial. In real-world scenarios, this information is not always directly accessible, especially when one seeks to avoid relying solely on ideal kinematic models. In this section, we compare the major methodologies found in the literature for pose estimation, highlighting their strengths, limitations, and applicability in robotic manipulation tasks.

Motion Capture Systems: accuracy at a cost. Motion capture (MoCap) systems, such as the widely used *Vicon*, are considered the gold standard for pose estimation accuracy. These systems employ infrared cameras and reflective markers to reconstruct the position and orientation of rigid bodies in 3D space. Their high precision makes them ideal as a ground truth provider in controlled environments, as seen in [33]. However, their use comes at a high financial cost, and they require structured environments, careful calibration, and a fixed infrastructure. This makes them impractical for mobile or unstructured settings.

Visual markers: low-cost alternative. A more accessible alternative is the use of *visual markers*, such as AR or QR codes. In this case, the camera pose (and hence the end-effector pose if the camera is mounted on it) is estimated by detecting and localizing these markers in the image. These approaches, as described by Marchand in [34], offer robustness under good visual conditions and are sufficiently accurate for many robotic applications. Nonetheless, they require that either the environment or the manipulated objects be modified to include artificial markers, reducing system generality and potentially interfering with object appearance or functionality.

Learning-based methods: end-to-end pose from images. Recently, *deep learning* methods have been employed for direct pose estimation from RGB or RGB-D images. Architectures such as PoseCNN [35] and DeepIM [36] learn a mapping from visual input to 6D object pose without requiring explicit geometric models. While these methods are flexible and marker-free, they demand large annotated datasets and suffer from generalization issues. Their performance can degrade under occlusion, poor lighting, or when faced with previously unseen objects. Additionally, real-time guarantees are often hard to meet, making them less suitable for closed-loop control.

Markerless VSLAM: *VSLAM* approaches, especially *markerless visual SLAM*, provide a general and autonomous solution to the pose estimation problem. Systems like ORB-SLAM3 and RTAB-Map estimate the full camera trajectory using visual information (and optionally depth or IMU), without requiring artificial markers or external systems. There is an extensive literature full of various methods for solving the VSLAM problem. Consequently, existing VSLAM algorithms fall into three categories:

- Optimization-based methods,
- Geometric type techniques,
- Kalman-type algorithms.

All these techniques have their advantages and disadvantages. For example, geometric-type algorithms can only guarantee almost global stability due to the existence of sets with Lebesgue measure zero in SO(3). Likewise, Kalman-type methods and optimization-based strategies suffer from performance dependency on the initialization and also cannot ensure stability. In Hashemi and Mattila's work [7], a monocular SLAM system is integrated into an operational-space controller, replacing feedback from forward kinematics with visual-based estimation. Similarly, this thesis adopts RTAB-Map in an RGB-D configuration, leveraging its robust 3D mapping and mitigating scale drift.

However, SLAM methods come with significant challenges: they are sensitive to environmental changes, accumulate drift over time, require substantial computation, and are complex to integrate into real-time control loops. Nevertheless, they are ideal for unstructured and dynamic environments where other solutions fail.

In conclusion, each pose estimation method involves a trade-off among accuracy, robustness, infrastructure requirements, and flexibility. The most suitable choice depends on the robotic platform, the operating environment, and the intended control architecture. In this work, a markerless SLAM approach using an RGB-D sensor is adopted as a practical compromise between accuracy and flexibility, enabling vision-based feedback control without the need for markers or external infrastructure.

2.4 Pose estimation and constrained control

Having established VSLAM as a promising alternative for pose estimation in robotic control, we now present a detailed analysis of one of the most recent and representative works integrating VSLAM with operational space control: the approach proposed by Hashemi and Mattila [7], which introduces a globally stable hybrid controller that uses a vision-based SLAM observer for task-space control of a 6-DOF robotic manipulator.

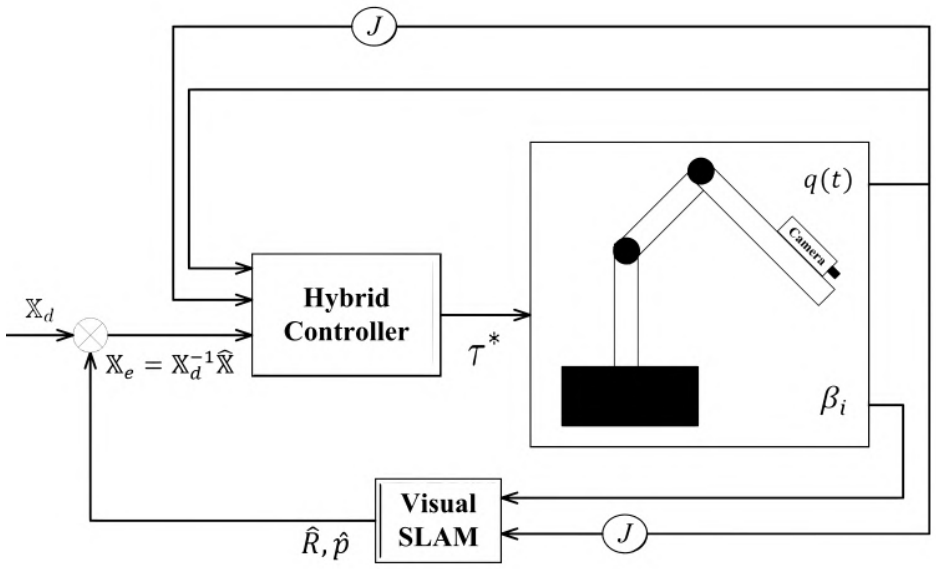


Figure 2.1: Block diagram of the proposed control.

This study is notable for demonstrating how vision-based SLAM, specifically a monocular ORB-SLAM system, can be used to estimate the pose of a robot end-effector in real time, replacing feedback from forward kinematics or external sensors. The estimated pose is then used directly in a closed-loop control law to track a desired trajectory in task space. It is noteworthy because it addresses the fundamental challenge of designing a globally stable vision-based controller, a significant step beyond the local stability guarantees often found in the visual services literature.

The proposed architecture replaces the traditional forward kinematics pipeline with a pose observer based on monocular vision. A geometric SLAM observer is derived on the Lie group SLAMn(3), and its output is a pose estimate $\hat{\Psi}(\hat{R}, \hat{p}) \in \mathrm{SE}(3)$ of the end-effector. The SLAM observer receives as input, [Figure 2.1]:

- The camera observations β_i, containing range and bearing to landmarks,
- And the Jacobian $J(\mathbf{q})$ of the manipulator, linking joint states to end-effector velocities.

In our work, this SLAM observer is implemented via the `rtabmap_ros` package, which processes RGB-D data from a camera rigidly attached to the end-effector.

The proposed **hybrid controller** is designed directly in the Lie group $\mathrm{SE}(3)$ by constructing a positive-valued continuously differentiable potential function $U(X_e, h)$, and taking its gradient. The error in pose is defined as:

$$X_e(t) = X_d^{-1}(t) \cdot \hat{X}(t) \tag{2.6}$$

where $X_d(t)$ is the desired pose, and $\hat{X}(t)$ is the SLAM-estimated pose of the end-effector.

The hybrid control law is then defined as:

$$\tau^* = N(\mathbf{q}, \dot{\mathbf{q}}) - \mathbf{M}(\mathbf{q}) J^{-1} \left(\dot{J}\dot{\mathbf{q}} + \phi \left(X_e^{-1} \Delta_{X_e} U(X_e, h) + G_d Y \right) \right) \tag{2.7}$$

with:

- $N(\mathbf{q}, \dot{\mathbf{q}}) = C(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + G(\mathbf{q}) + F(\dot{\mathbf{q}})$: the nonlinear dynamic model including Coriolis, gravity, and friction terms;
- $\mathbf{M}(\mathbf{q})$: the inertia matrix;
- $J(\mathbf{q})$: the manipulator Jacobian;
- ϕ, G_d: positive control gains;
- Y: an auxiliary variable in the controller design;
- $\Delta_{X_e} U$: the gradient of the potential function on $\mathrm{SE}(3)$.

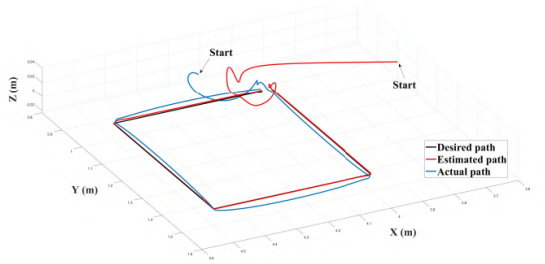
The use of the SLAM-estimated pose $\hat{X}(t)$ in the control loop does not compromise stability, since the observer is proven to converge globally, i.e., $\lim_{t\to\infty} \hat{X}(t) = X(t)$, allowing the replacement of the true pose with the estimated pose in the controller.

The proposed architecture (Figure 2.1) integrates the VSLAM observer and the task-space controller. The inputs to the controller are:

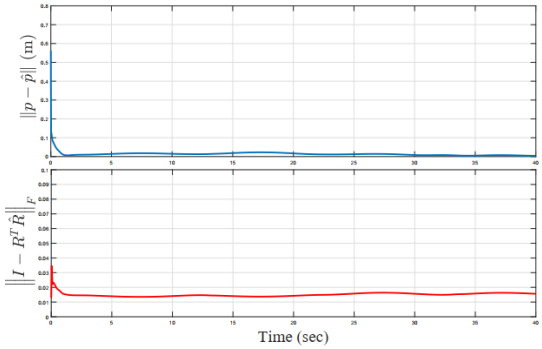
- The estimated pose $\hat{X}(t)$ from VSLAM,
- The desired trajectory $X_d(t)$,
- And the robot model outputs $J(\mathbf{q})$, $\mathbf{q}(t)$.

Experimental results and discussion: To validate their control framework, the authors simulate a 6-DOF robotic manipulator tasked with executing a square trajectory of 50 cm per side. Notably, the initial estimate of the end-effector pose, derived from the SLAM-based visual observer, is deliberately offset from the true state. Despite this, the control scheme effectively compensates for the initial misalignment, demonstrating robustness to pose estimation errors and the ability to converge to the desired trajectory.

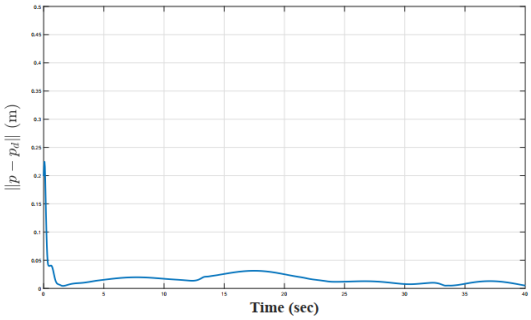
The results demonstrate the controller's capability to track the desired end-effector trajectory even in the presence of significant initial uncertainty, as evidenced by the visualization of the executed, desired, and estimated paths in 3D space (see Figure 2.2a). The estimation errors in both position and orientation evolve over time and are progressively reduced by the observer, as quantified using the Euclidean and Frobenius norms, respectively (Figure 2.2b). Despite an initial pose estimation error of up to 1%, the tracking error in task space remains consistently below 2% throughout the trajectory (Figure 2.2c), confirming the effectiveness of the proposed inverse dynamics control law in rejecting disturbances and achieving accurate motion tracking. Additionally, the joint torque profiles are smooth and well-behaved, remaining within realistic bounds (Figure 2.2d), which further supports the feasibility of this approach from a physical actuation standpoint.



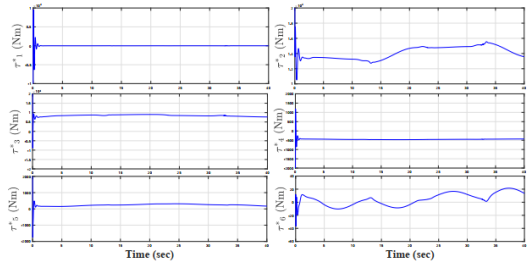
(a) End-effector trajectory, desired trajectory, and estimated trajectory in 3D space.



(b) Position and attitude estimation errors.



(c) Position tracking error of the end effector.



(d) Control input torque.

Figure 2.2: Simulation results from the VSLAM-based control approach.

Among the key advantages of this approach, the authors emphasize its global asymptotic stability, rigorously proven through Lyapunov analysis. The framework relies exclusively on visual feedback, completely omitting joint encoder data for task-space estimation. Additionally, the proposed control law is computationally efficient, as it avoids costly numerical optimization or matrix inversion, relying instead on gradient-based updates.

Nonetheless, several limitations are also acknowledged. First, the system requires a careful offline initialization of the visual observer, including scale alignment and gain tuning. Second, although the observer converges to the true state, transient errors due to SLAM drift can affect short-term tracking performance. Finally, it is important to note that the entire validation is conducted in simulation; real-world experiments remain an open direction for future work.

2.4.1 Novelty of this work

The control architecture presented by Hashemi and Mattila [7] provides conceptual inspiration for this thesis, particularly in the way it integrates pose estimation via Visual SLAM into a task-space control loop. While their work proposes a novel hybrid feedback law derived from the gradient of a potential function defined on $\mathrm{SE}(3)$, such a geometric formulation is not directly implemented in this thesis.

Instead, we adopt the broader idea of using a VSLAM-based observer as a source of feedback for task-space control. Specifically, we replace the theoretical geometric VSLAM observer algorithm from [7] with a practical implementation based on RTAB-Map, a graph-based SLAM system compatible with ROS2 and capable of estimating the end-effector pose from RGB-D data in real time. This markerless

solution is well-suited for deployment in real-world, unstructured environments.

Furthermore, like [7], the control law is evaluated in simulation, this thesis presents a custom torque-level controller based on the inverse dynamics of the Niryo Ned2 robot. The controller is developed as a plugin for the `ros2_control` framework and utilizes the pose estimate $\hat{X}(t)$ provided by RTAB-Map as task-space feedback.

Importantly, the control law used in this work does not follow the exact mathematical formulation of the gradient-based controller in [7]. Instead, it is independently designed and tailored to the specific hardware and software constraints of the Niryo platform and the ROS2/MoveIt2 ecosystem. The details of this control law, including its mathematical structure and implementation,will be presented and discussed thoroughly in Chapter 4.

2.4.2 Research gap and thesis contribution

The preceding sections have provided a comprehensive overview of the state of the art in several key areas relevant to this thesis. It is clear that methodologies for *OSC* are well-established, offering a rich theoretical foundation for controlling manipulators directly in their task space. Similarly, a wide array of methods exists for pose estimation, ranging from high-precision external systems like Vicon to flexible, infrastructure-free techniques like Visual SLAM. Recent and compelling research, such as the work by Hashemi and Mattila, has demonstrated the theoretical viability of integrating VSLAM and OSC to create globally stable, vision-based control systems.

However, a significant gap often exists between the theoretical formulation of such advanced algorithms and their practical, robust implementation within a modern, standardized robotics framework. This thesis aims to bridge this gap by addressing several aspects of engineering, integration, and application that are often not the primary focus of theoretical works.

While many academic papers present SLAM as a mathematical algorithm, this work focuses on its practical implementation by leveraging and integrating a well-supported, off-the-shelf system, RTAB-Map, within the ROS 2 ecosystem. This shifts the challenge from algorithmic design to system integration, addressing the real-world engineering problems of making such systems work reliably. A core contribution of this thesis is the implementation of a modular software architecture. Rather than a monolithic script, the system is composed of decoupled ROS 2 nodes and components that communicate through standard interfaces. This engineering effort, often abstracted away in theoretical literature, is crucial for creating a system that is reusable, extensible, and easy to debug.

The vast majority of SLAM research focuses on mobile robots for navigation purposes. This thesis explores a different application: repurposing VSLAM as an exteroceptive pose sensor for a fixed-base manipulator. In this context, SLAM is not used for localizing the robot's base but for creating a high-frequency, external feedback loop for the end-effector, presenting a unique set of challenges and opportunities for control. The developed operational space controller is not implemented as a simple ROS 2 node, but as a formal ros2_control plugin. This represents a deeper and more robust level of integration into the ROS 2 framework, adhering to modern standards for robot control. This allows for seamless interaction with simulation interfaces like Gazebo and standard controller management tools, a significant step beyond a proof-of-concept implementation.

Finally, the entire project was developed using a modern but complex toolchain involving the Windows Subsystem for Linux (WSL). Navigating the specific challenges and potential issues associated with this

environment has added a layer of practical engineering problem-solving to the project, reflecting the realities of contemporary robotics software development.

In summary, while building upon the strong theoretical foundations of OSC and VSLAM-based control, this thesis distinguishes itself by focusing on the practical engineering and integration challenges required to create a functional, modular, and modern robotic system. It aims to translate advanced theoretical concepts into a robust, working implementation on an accessible research platform, providing valuable insights for future real-world applications.

Chapter 3

System architecture

The goal of this thesis is to design and evaluate a robotic system capable of performing operational space trajectory tracking using pose estimates obtained from visual SLAM. This objective requires a system architecture that is both modular and robust, supporting seamless integration of heterogeneous components within the ROS 2 ecosystem.

From an architectural standpoint, the proposed system must ensure tight coordination between multiple functionalities: sensor acquisition, environment mapping and localization, motion planning, and control. Each of these stages introduces specific design challenges, especially when aiming for a deployable and generalizable pipeline that goes beyond purely theoretical formulations.

In particular, the following research challenges are addressed:

- ***Pose estimation via Visual SLAM:*** Unlike classical industrial manipulators that rely on forward kinematics or external motion capture systems, this work explores the use of an onboard RGB-D camera to estimate the pose of the end-effector using visual SLAM techniques. The integration of this estimate into the control loop introduces uncertainty and delay, requiring careful consideration.
- ***Operational space control:*** Designing a controller that operates in the task space (end-effector pose), rather than joint space, necessitates access to accurate and real-time feedback of the end-effector pose in the world frame. The controller must also account for the system dynamics to generate joint torques accordingly.
- ***Integration of heterogeneous modules:*** The architecture must integrate various open-source components, such as RTAB-Map for SLAM, MoveIt 2 for motion planning, and a custom `ros2_control` plugin for control, in a synchronized and modular way. Interoperability across these components is non-trivial due to differences in data formats, timing, and system assumptions. Additionally, the use of the Pinocchio library [37] for computing the robot's full rigid-body dynamics is critical to enable inverse dynamics control within the custom controller.

These challenges frame the architectural decisions made in this work and motivate the system design presented in the next sections.

3.1 High-level system overview

This section provides a high-level overview of the proposed system architecture, designed to enable a fixed-base robotic manipulator to perceive its environment, perform localization and 3D mapping, plan motion trajectories, and execute them through torque-level control. The flow of information among the system's core modules, along with the associated data dependencies and interface connections, is schematically summarized in the system diagram (Figure 3.1).

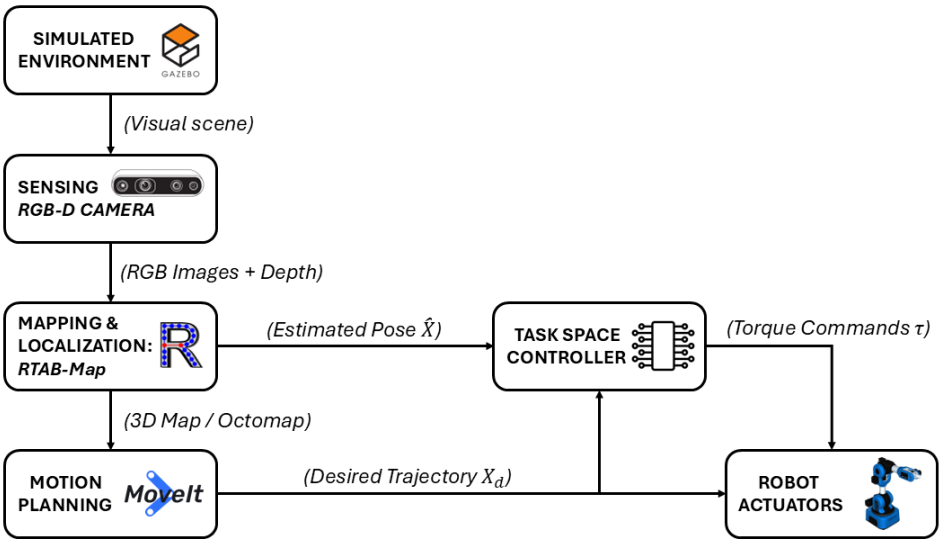


Figure 3.1: High-level architecture of the proposed system, showing the modules and the data flow.

The system is composed of four main functional components, each implemented as independent ROS 2 nodes communicating through topics, TF frames, and services.

1. **Sensing:** An RGB-D camera is rigidly mounted on the `wrist` of the Niryo Ned2 manipulator. This configuration allows the robot to dynamically perceive the workspace from task-relevant perspectives. The camera provides synchronized colour and depth images, which serve as input for the SLAM and mapping pipeline. A custom Gazebo world is used to simulate a realistic workspace containing tables and various obstacles, allowing the mapping pipeline to perceive meaningful 3D structure and populate the OctoMap accordingly.

2. **Mapping & Localization:** Visual odometry and loop closure detection are handled by the RTAB-Map SLAM algorithm, which estimates the pose of the camera in the `world` frame. A volumetric representation of the environment is built incrementally using OctoMap, allowing for real-time obstacle awareness and collision checking.

3. **Motion Planning:** The MoveIt 2 framework is used to generate a collision-free trajectory in joint space using the CHOMP planner. To enable operational-space control, a dedicated ROS 2 node (which will be explained in the next chapter) samples the resulting joint trajectory and extracts the corresponding sequence of desired end-effector poses by computing the forward kinematics at each sampled timestep. The desired trajectory is expressed in the `world` → `tool_link` transform.

4. **Control:** A custom ROS 2 controller plugin implements an operational space controller that receives the desired end-effector poses and computes joint torques in real-time. The current pose of the end-effector is estimated directly using the camera pose provided by SLAM, combined with the known transform between the `wrist` and `tool_link`. The inverse dynamics computations are performed using the Pinocchio library, allowing accurate and smooth torque control based on the computed error between desired and estimated poses.

The architecture follows a modular design pattern that separates sensing, localization, planning, and control into loosely coupled components, promoting clarity, reusability, and extensibility.

3.2 Data flow and communication interfaces

A fundamental aspect of the proposed system architecture is the definition of well-structured data flows and communication interfaces that connect its modular components. Leveraging the ROS 2 middleware, the system relies on a combination of topics for asynchronous data streaming and the TF2 transform tree for managing dynamic spatial relationships. This ensures consistent, real-time interaction among the sensing, mapping, planning, and control modules. The primary information exchange occurs over ROS 2 topics, where each module operates as an independent node that publishes or subscribes to specific data streams. The main data flows include various elements.

The exchange of data between modules follows a well-defined structure. At the sensing level, the RGB-D camera continuously publishes raw image data and intrinsic calibration parameters on topics such as `/gazebo_depth_camera/image_raw` and `/gazebo_depth_camera/camera_info`. These image streams serve as input to the SLAM pipeline, while the associated 3D perception is captured in the form of point cloud data, published as `sensor_msgs::msg::PointCloud2` messages. This 3D data plays a dual role: it is processed by RTAB-Map for real-time visual mapping and simultaneously integrated into the OctoMap framework of MoveIt 2 to enable collision-aware trajectory planning.

Moving up the stack, both the desired motion and the estimated position of the robot's end-effector are handled through pose messages. The estimated pose, critical for feedback control, is typically represented using `geometry_msgs::msg::PoseStamped` messages and can originate either from the SLAM system or from forward kinematics. On the other hand, the planner (e.g., CHOMP) generates a desired trajectory as a time-parameterized sequence of similar pose messages, which define the reference path to be tracked.

At the control level, the robot's internal state, including joint positions and velocities, is streamed via the standard `/joint_states` topic. The custom controller subscribes to this information, computes the necessary torques based on the tracking error in operational space, and applies them through effort interfaces exposed by the `ros2_control` framework. These joint torques represent the final control action applied to the robot, closing the loop between sensing, planning, and actuation. While topics manage data exchange, temporal and spatial consistency are maintained through two additional ROS 2 mechanisms. Temporal alignment is ensured by the timestamps embedded in each message header, which allow the system to correlate data from different sources accurately; for instance, aligning a pose estimate with the corresponding point along the desired trajectory.

Spatial synchronization is managed by the TF2 transform tree, which provides real-time transformations between all relevant coordinate frames. In this system, the `world` frame serves as the fixed inertial

reference. The pose of the end-effector, identified by the `tool_link` frame, is computed at runtime by querying the transformation from `world` to `tool_link`. This mechanism is highly flexible, as it enables pose estimation to be derived seamlessly from either the kinematic model or the visual SLAM pipeline, depending on the chosen source of the transformation. Within this framework, the control module receives both the desired trajectory (as a feedforward reference) and the estimated pose (as a feedback measurement). It computes the tracking error in operational space and calculates the corresponding joint torques. To this end, the controller leverages the `Pinocchio` library to maintain a real-time model of the robot's dynamics, enabling efficient and accurate inverse dynamics computations. The resulting torque commands are applied through ROS 2's effort-based control interfaces.

Finally, latency remains a critical consideration in this loop architecture. Delays introduced by image acquisition, SLAM processing, and data transmission can negatively impact the stability and responsiveness of the control loop. These latencies are mitigated through careful system configuration, including buffer tuning and optimization of data pipelines.

The data flow architecture described above is designed to be modular and adaptable to various robotic platforms. In this work, the Niryo Ned2 robotic manipulator has been selected as the experimental platform for implementing and testing the proposed control strategy.

3.3 Problem statement

The control problem addressed in this work consists in accurately tracking a desired end-effector trajectory defined in the operational space, using as feedback a pose estimate obtained through a visual SLAM-based localization system. Unlike classical approaches that rely solely on forward kinematics for feedback, the system proposed here integrates visual perception in the loop, introducing both opportunities and challenges in terms of uncertainty, latency, and robustness.

Let us denote by $\mathbf{X}_d(t) \in SE(3)$ the desired trajectory of the end-effector in the operational space at time t, representing a rigid body transformation composed of a position $\mathbf{p}_d(t) \in \mathbb{R}^3$ and an orientation $\mathbf{R}_d(t) \in SO(3)$. Although orientation can be equivalently represented by a unit quaternion $\mathbf{q}_d(t) \in \mathbb{H}$, in this work we use the homogeneous transformation matrix notation consistent with the underlying Lie group structure of $SE(3)$.

In practice, this trajectory is generated offline using a sampling-based planner (CHOMP) within MoveIt 2 and is represented as a discrete-time sequence of N time-stamped waypoints $\{\mathbf{x}_d(t_k)\}_{k=0}^{N}$, sampled at fixed intervals (100 ms). This sampled representation is suitable for downstream processing in the control loop; its detailed generation is presented in Chapter 4.

The current pose of the end-effector is obtained via perception, specifically by querying the TF2 transform tree. It is represented in the same operational space $\hat{\mathbf{X}}(t) \in SE(3)$, where $\hat{\mathbf{X}}(t)$ denotes the estimated rigid transformation of the end-effector frame with respect to the base frame. This estimate may be derived either from the robot's forward kinematics or from the SLAM-based visual localization system, depending on the configuration.

The pose tracking error is defined as the relative transformation between the desired and estimated pose:

$$\mathbf{X}_e(t) = \mathbf{X}_d(t)\hat{\mathbf{X}}(t)^{-1} \in SE(3) \tag{3.1}$$

For control design, this error in $SE(3)$ (3.1) is mapped to its corresponding vector representation in the Lie algebra $\mathfrak{se}(3)$ using the logarithmic map. This yields a minimal 6D vector representation, often

called a *twist*, that combines translational and rotational errors:

$$\boldsymbol{\xi}(t) = \log\big(\mathbf{X}_e(t)\big)^{\vee} \in \mathbb{R}^6 \tag{3.2}$$

This error vector $\boldsymbol{\xi}(t)$ serves as the primary input to the feedback controller. The control objective is to design a control law that drives this error vector asymptotically to zero, i.e., $\lim_{t\to\infty} \boldsymbol{\xi}(t) = \mathbf{0}$ thereby ensuring the manipulator's end-effector pose converges to the desired trajectory, even in the presence of estimation inaccuracies and model uncertainties.

To achieve this goal, the controller computes the required joint torques $\boldsymbol{\tau} \in \mathbb{R}^n$ (with n being the number of actuated joints) that produce the desired end-effector motion in the workspace. This is done by leveraging a dynamic model of the robot, which is maintained and updated in real-time using the `Pinocchio` library. The robot's state, composed of joint positions $\mathbf{q}$ and joint velocities $\dot{\mathbf{q}}$, is obtained from the `state_interface_configuration` of ROS 2 control and used to evaluate the kinematic and dynamic quantities required by the inverse dynamics law.

The overall control architecture thus consists of:

- A **feedforward** reference trajectory $\mathbf{X}_d(t)$;
- A **feedback** measurement $\hat{\mathbf{X}}(t)$ from SLAM;
- A **tracking error** $\boldsymbol{\xi}(t)$ (3.2) computed in the Lie algebra of $SE(3)$;
- An **inverse dynamics controller** that outputs joint torques $\boldsymbol{\tau}$, applied through the ROS 2 effort-based control interface.

This formulation enables the robot to execute complex, perception-aware motions in partially unknown environments. By formalizing the control task as an explicit operational space tracking problem with perception-based feedback, the architecture provides a principled way to close the loop between sensing, planning, and actuation.

3.4 Modular software architecture: nodes and packages

The proposed system is founded on a modular software architecture built upon ROS 2 Humble, following the core design principle of functional separation. Each logical component of the robotic pipeline is encapsulated within its own ROS 2 package and node, enabling independent development, testing, debugging, and future reuse. This separation of concerns improves the maintainability and scalability of the system, allowing individual components to be modified or replaced with minimal impact on the rest of the architecture.

3.4.1 Overview of key packages

The software is primarily organized into three custom ROS 2 packages:

1. `niryo_moveit2_config` is the central configuration and launch package. It contains the robot description files in URDF and XACRO formats, the Semantic Robot Description Format (SRDF) for motion planning, and various configuration files for joint limits, kinematics solvers, controller integration, and 3D perception. The package also includes a `launch/` directory with Python launch files to start the full simulation environment, including the SLAM node, MoveIt 2, Gazebo, and custom utilities for trajectory generation.

2. `worlds_niryo` is a utility package containing the Gazebo world definition and 3D models required to simulate the Niryo Ned2 manipulator in different test environments.

3. `op_space_controller` is the custom controller package developed in this thesis. It implements an operational space controller as a plugin for the `ros2_control` framework. The package includes C++ source and header files, a plugin description file, a parameter configuration file, and a dedicated URDF used by the Pinocchio dynamics library for inverse dynamics computations.

3.4.2 ROS 2 nodes and their roles

At runtime, these packages give rise to a set of interconnected nodes that form the robotic execution pipeline. Each node is responsible for a specific subsystem and communicates with the others via standard ROS 2 interfaces (topics, services, and actions), ensuring minimal coupling and high modularity. Below is a list of the main nodes initialized via launch files and used in the context of this thesis project.

The **Visualization Node** (RViz) provides an interactive 3D interface for monitoring the robot's state, planned trajectories, SLAM-generated map, and sensor streams. This component plays a critical role during development and testing by enabling real-time feedback and debugging.

The **MoveIt 2 Node** (`/move_group`) handles high-level motion planning. It consumes the robot description and semantic information to plan collision-free trajectories in operational space. These trajectories are sampled and stored via a custom utility node included in the configuration package, which outputs a sequence of desired end-effector poses over time.

The **Gazebo Node** runs the physics-based simulation of the Niryo Ned2 robot and its environment, interfacing with the control framework through the ROS 2 middleware. Sensor data, such as RGB and depth images, is published by Gazebo to replicate a realistic perception pipeline.

The **SLAM Node**, based on the RTAB-Map algorithm, processes RGB-D data from the simulated depth camera to estimate the end-effector's pose relative to the environment. It publishes this information as part of the TF2 tree and on dedicated ROS 2 topics, making it available to other components such as the planner and controller.

The **TF Node** and the **Robot State Publisher Node** are responsible for maintaining and broadcasting the transform tree of the robot. While TF handles the dynamic transformation lookup, the state publisher continuously publishes the current joint states and associated transformations based on the robot description and controller outputs.

The control logic is implemented as a custom plugin for the `ros2_control` framework, representing a core contribution of this thesis. This operational space controller, written in C++, is dynamically loaded by the `ros2_control_node`, which acts as the central control manager. The controller consumes the desired end-effector trajectory and the pose feedback from the SLAM node to compute joint torque commands using inverse dynamics based on the Pinocchio model. Although the controller is not a standalone ROS 2 node, its integration as a plugin allows seamless interaction with the robot's hardware interface (simulated via Gazebo) and ensures compatibility with the ROS 2 control lifecycle. The controller's parameters and configuration are provided via YAML files loaded at runtime, and the plugin is activated through the standard `spawner` utility provided by the `controller_manager` package.

As described in the high-level architecture (Figure 3.1), the system is organized into functional modules connected through standardized ROS 2 communication interfaces. While that figure provides a conceptual overview, this section has detailed the concrete ROS 2 nodes and packages that implement each module in practice. This modular implementation ensures that each component, such as the SLAM backend, the motion planner, or the operational space controller, can be independently replaced, tuned, or extended. Thanks to the decoupled design enabled by ROS 2, the framework remains highly adaptable and scalable, allowing for integration with different sensors, planning strategies, or robotic platforms across varied experimental setups.

3.5 Design considerations

The system architecture presented in this chapter was guided by a set of key design considerations aimed at ensuring performance, modularity, and relevance to modern robotics development practices. These requirements influenced the choice of software frameworks, implementation languages, and the overall structural design.

Compatibility with ROS 2 Humble: A primary requirement was to build the system upon a stable and modern robotics framework. ROS 2 Humble Hawksbill was selected as the foundation, being a Long-Term Support (LTS) release. This choice guarantees a stable API, ongoing security updates, and a mature ecosystem of packages (e.g., MoveIt 2, RTAB-Map, ros2_control). Using standard ROS 2 interfaces ensures loose coupling between components and supports long-term maintainability and extensibility of the software stack.

WSL 2 environment: The project was developed entirely within the WSL 2, reflecting a contemporary cross-platform workflow common in academic and industry contexts. However, this introduced specific technical constraints, such as reduced support for graphical rendering and limitations in hardware access (e.g., USB pass-through, real-time scheduling). These challenges reinforced the need for a decoupled and network-transparent architecture, allowing computationally intensive or graphically demanding nodes, such as Gazebo or RViz, to communicate seamlessly with the core logic, regardless of the host operating system. To address these limitations, the system avoids shared memory transports and adopts strategies compatible with user-space simulation in WSL. Moreover, all nodes communicate through standard ROS 2 middleware, allowing the system to remain portable to native Linux deployments or future hardware setups with minimal changes.

Real-time performance and controller design: A critical requirement for the control architecture was ensuring a high-frequency, low-latency feedback loop, especially for motion tracking based on vision. This motivated the implementation of the task-space controller in C++, which offers superior runtime performance compared to Python-based approaches. In addition, the controller was developed as a plugin within the `ros2_control` framework rather than as an independent ROS node. This architectural choice minimizes latency by enabling direct access to joint state interfaces and command handles, while benefiting from the scheduling capabilities of the ROS 2 controller manager. The result is a tightly integrated and performant control loop, better suited to real-time task-space control than a standard ROS processing pipeline.

Modularity and scalability: A core design goal was to achieve modularity across the perception, planning, and control layers, allowing each component to be replaced or upgraded independently. The

system is structured around loosely coupled nodes that communicate via standard ROS 2 messages and services. Crucially, the task-space controller is designed to be robot-agnostic: it loads the robot description at runtime from a URDF/Xacro file, enabling deployment on other fixed-base manipulators with minimal modifications. Similarly, the perception pipeline (e.g., RTAB-Map) is not hardcoded to specific camera drivers or frame semantics, making it easy to substitute sensors or mapping algorithms. This level of abstraction ensures the scalability and reusability of the framework across diverse experimental setups.

In the next chapter, each functional block of the architecture will be discussed in detail, including configuration files, key parameters, and the custom code developed for the system's implementation.

Chapter 4

System implementation

This chapter presents the implementation of the system architecture introduced in Chapter 3. Each functional module, simulation, perception, planning, and control, is described in detail, with a focus on the configuration of ROS 2 packages, custom C++ nodes, and the integration of third-party libraries such as MoveIt 2, RTAB-Map, and Pinocchio.

The chapter begins with the setup of the Gazebo simulation environment, including the robot model modifications and camera integration. It then covers the configuration of the perception pipeline using RTAB-Map, followed by the generation of a reference trajectory for the end-effector. The final and most critical section details the implementation of the operational-space controller as a `ros2_control` plugin, including the control law and the inverse dynamics computations. The chapter concludes with the launch and execution of the complete system.

4.1 Setting up the simulation environment in Gazebo

The practical implementation and validation of the proposed control architecture were conducted within a high-fidelity simulation environment. This approach allows for safe, repeatable, and cost-effective testing of the entire perception and control pipeline. This section details the setup of the virtual world in Gazebo and the necessary modifications made to the robot's URDF model.

4.1.1 Creating the virtual world

A custom simulation world was designed in Gazebo to provide a controlled and repeatable environment for testing the perception and control system. The environment (Figure 4.1) is configured to resemble a simplified industrial or laboratory workspace and includes a variety of standard models to create a scene with sufficient visual and geometric complexity.

The world is built upon the standard `ground_plane` and contains several objects, such as a table, a `euro_pallet`, a conveyor, and a cylindrical support structure for the robot. A distinctly colored `red_cube` is also included as a potential target for manipulation or object recognition tasks. These objects were specifically selected to provide the VSLAM system with a rich set of visual cues, including edges, corners, and textured surfaces, which are essential for reliable real-time localization and for constructing a consistent 3D map.

The complete Gazebo world is defined in a `.world` file, available in the public code repository associated with this project for reproducibility purposes. A representative code snippet illustrating the

insertion of the `euro_pallet` model is included in Appendix 1.

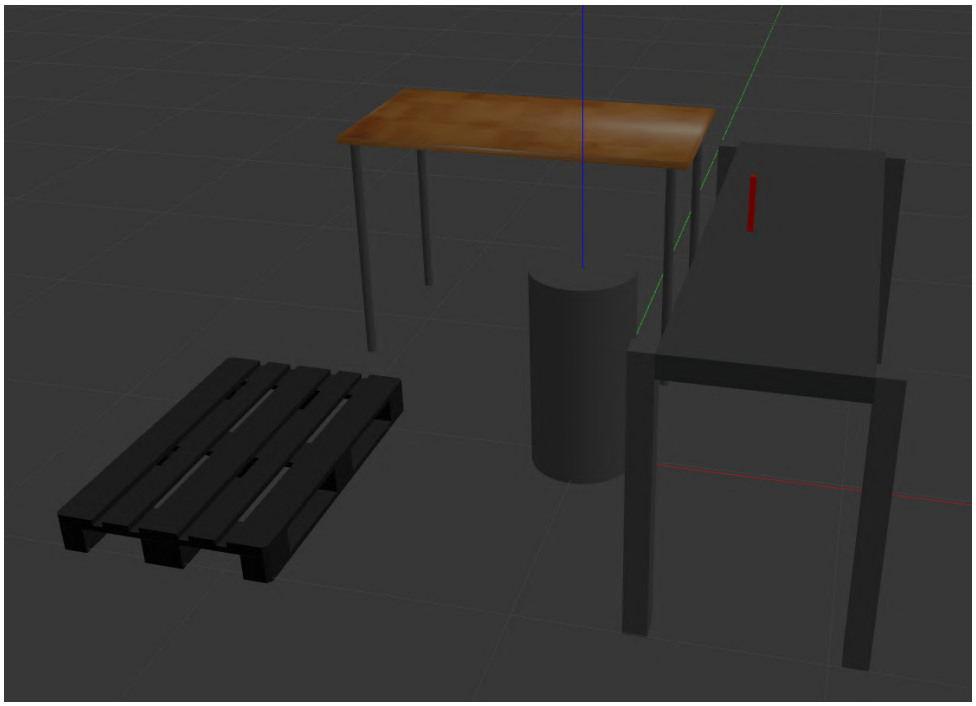


Figure 4.1: Overview of the Gazebo simulation world.

4.1.2 Robot model integration and camera mounting

The standard URDF model for the Niryo Ned2, provided within the `niryo_robot_description` package, was extended to enable high-fidelity simulation in Gazebo and to support visual perception tasks. These extensions were implemented using XACRO macros for modularity and maintainability, and focused on two primary aspects: integrating the `ros2_control` interface for actuation in Gazebo and mounting an RGB-D camera on the robot's wrist.

To **enable the custom controller** to actuate the simulated robot, the URDF was integrated with the `ros2_control` framework. This was achieved by adding the `gazebo_ros2_control` plugin, as shown in Appendix 2. This plugin acts as the simulated hardware interface, loading the controller configurations from the specified `.yaml` file, which defines the names and types of the available controllers for the controller manager, and exposing the robot's joint states (position, velocity) and command interfaces (e.g., for effort/torque) to the ROS 2 ecosystem. The standard `<transmission>` tags, included via a separate XACRO file, were also necessary to mechanically link the abstract actuators defined in `ros2_control` to the corresponding physical joints in the simulation.

For visual perception, an **RGB-D camera** was physically and kinematically integrated into the robot model through modifications to the URDF/XACRO description (Figure 4.2). As detailed in Appendix 3, a new link named `depth_camera_link` was defined to represent the physical body of the camera. This link is rigidly attached to the robot's `wrist_link` via a fixed joint, whose transformation defines the extrinsic calibration, i.e., the static pose of the camera with respect to the wrist.

In line with ROS coordinate frame conventions (REP 103 [38]), an additional link named `depth_camera_optical_link` was introduced. This link is rotated relative to the physical camera frame so that it conforms to the standard optical coordinate system, where the Z-axis points forward through the

lens, the X-axis to the right, and the Y-axis downward. This alignment is essential for compatibility with perception algorithms and visualization tools such as RViz and RTAB-Map.

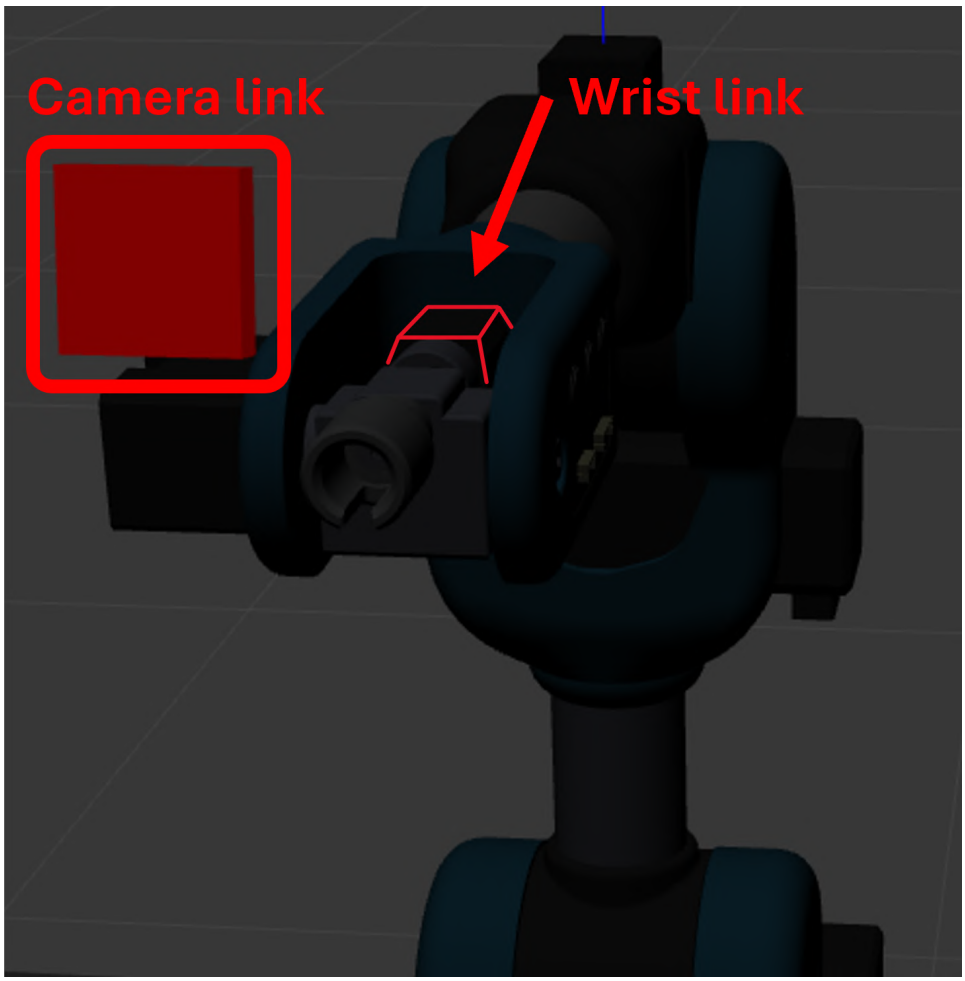


Figure 4.2: Visualization of the camera attached to the robot's wrist_link.

The simulated camera behavior in Gazebo is specified using a `<gazebo>` tag associated with the `depth_camera_link`. This block instantiates the `libgazebo_ros_camera.so` plugin, which emulates an RGB-D sensor with configurable physical and publishing parameters. The main parameters used in the simulation are as follows:

- **Update rate:** `10.0 Hz`, specifying how often new images and depth data are published.
- **Resolution:** `640×480`, defining the size of each image frame.
- **Field of View (FOV):** `1.0 rad`, defining the horizontal angular span of the sensor.
- **Clip range:** `near = 0.05 m`, `far = 8.0 m`, setting the minimum and maximum sensing distance for valid depth data.
- **Depth range:** `min_depth = 0.1 m`, `max_depth = 100.0 m`, constraining the generation of valid depth points.

The plugin publishes the sensor output using the standard ROS 2 message types `sensor_msgs/Image`, `sensor_msgs/CameraInfo`, and `sensor_msgs/PointCloud2`, and sets the frame ID of all messages to `depth_camera_optical_link`. The following topics are made available:

- `/gazebo_depth_camera/image_raw`
- `/gazebo_depth_camera/camera_info`
- `/gazebo_depth_camera/depth/image_raw`
- `/gazebo_depth_camera/depth/camera_info`
- `/gazebo_depth_camera/points`

These topics serve as the primary input for SLAM modules, scene understanding, and the visual feedback control loop described in subsequent sections.

4.2 SLAM integration

In order to enable autonomous perception and localization within a previously unknown environment, this work integrates the *RTAB-Map* framework into the simulated control architecture. RTAB-Map is a graph-based SLAM approach capable of real-time 3D mapping and localization using RGB-D, stereo, or lidar data. It is particularly well-suited for scenarios involving visual SLAM due to its modularity and support for depth cameras, making it an appropriate choice for the requirements of the proposed control system.

This section presents the essential configuration choices that have been adopted to ensure robust localization in simulation. For reproducibility and clarity, the complete launch-time configuration of both nodes is provided in Appendix 4, which will be revisited in section "4.5 Starting and Running the Complete System" to explain the entire project workflow.

4.2.1 Configuring RTAB-Map nodes

The integration of RTAB-Map is realized through two primary ROS 2 nodes: `rtabmap`, which executes the core SLAM process, and `rtabmap_viz`, which provides real-time visualization of the mapping and localization pipeline. The `rtabmap` node is responsible for several key functionalities, including frame-to-frame visual odometry, loop closure detection, global graph optimization, and 3D occupancy map construction. It subscribes to image, depth, and point cloud data and publishes estimated camera poses and transforms within the TF tree.

The `rtabmap_viz` node provides a dedicated graphical interface for real-time monitoring and debugging of the SLAM process. Unlike the general-purpose RViz, this tool is specifically tailored to visualize the internal data structures of the RTAB-Map algorithm. It subscribes to the SLAM outputs and displays the evolving 3D map, the estimated trajectory of the camera, and various diagnostic metrics useful for system tuning. The interface includes multiple specialized panels, such as a real-time 3D map view rendering the point cloud and trajectory, as well as loop closure detection overlays that highlight visual matches with past keyframes. In the current simulation setup, only the 3D map and depth image views are activated, allowing focused observation of the environment reconstruction and pose estimation processes (Figure 4.3).

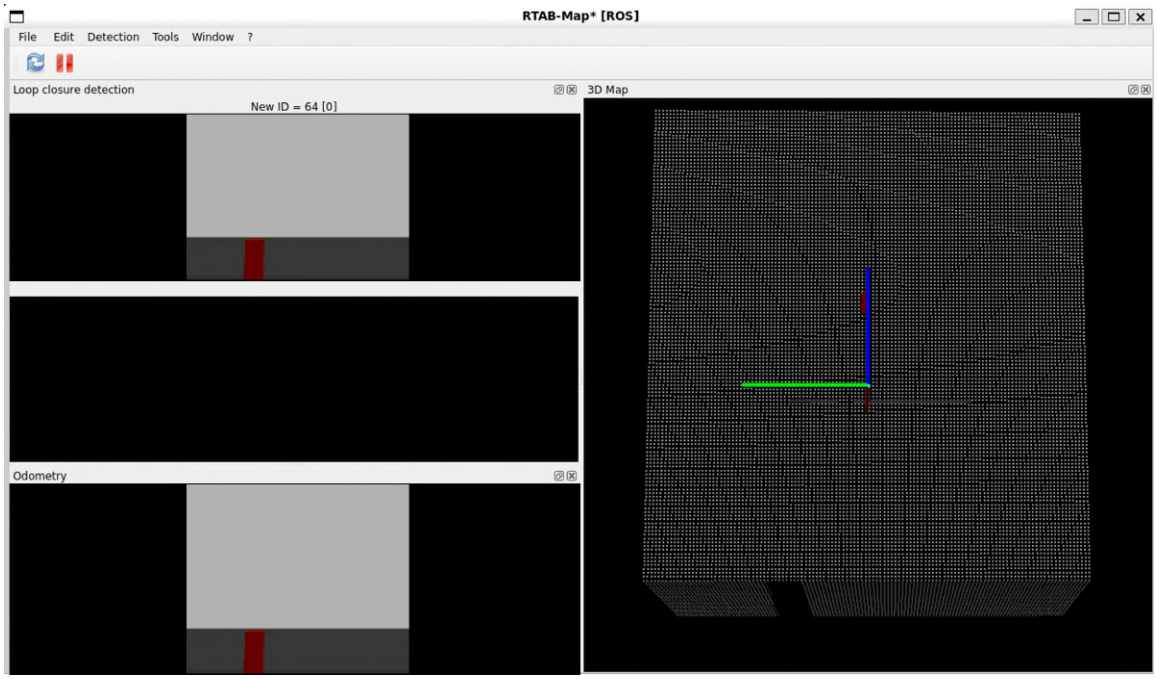


Figure 4.3: The `rtabmap_viz` interface, showing the 3D map view, depth image and odometry displays.

Both nodes are launched using a dedicated ROS 2 launch file, `niryo_slam.launch.py`, which specifies all required parameters and topic remappings. The RGB-D camera publishes its outputs on topics such as `/depth_camera/color/image_raw` and `/depth_camera/depth/image_raw`, which must be remapped to the topic names expected by RTAB-Map, namely `/rgb/image` and `/depth/image`, respectively. This remapping is explicitly handled in the launch file to ensure compatibility between the simulated camera and RTAB-Map's internal subscription interfaces.

4.2.2 Tuning key parameters

To ensure correct and robust behavior within the Gazebo simulation, several of RTAB-Map's ROS 2 parameters were tuned from their default values. This process was crucial for adapting the general-purpose SLAM algorithm to the project's specific constraints, namely, a fixed-base manipulator that relies exclusively on visual depth sensing.

Temporal, frame, and synchronization configuration: First, both the `rtabmap` and `rtabmap_viz` nodes are configured to use simulation time by enabling the `use_sim_time` flag. This ensures temporal consistency between the simulated camera data and the SLAM algorithm's internal time stamps, which is crucial for correct data association.

Since no external odometry source (e.g., wheel encoders or IMU) is used, the robot's base frame `base_link` is specified as both the odometry and sensor reference frame. This configuration effectively makes RTAB-Map responsible for generating the full transformation chain `map` → `base_link`, with no intermediate odometry frame. The estimated global pose of the robot is therefore published directly as a transform between `map` and `base_link`, which simplifies the TF tree (Figure 4.4) and removes dependencies on additional localization sources.

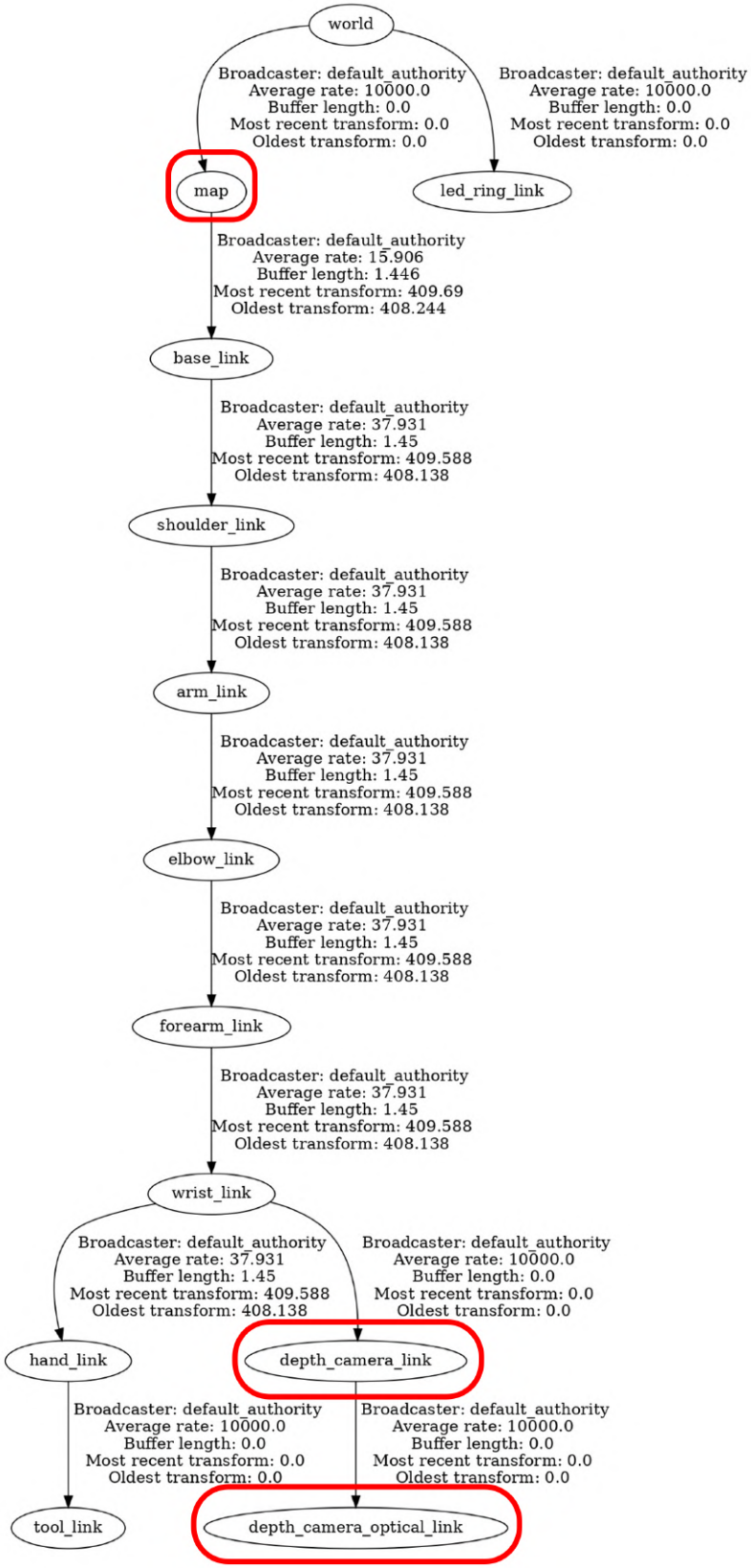


Figure 4.4: The updated TF tree with the addition of the frame `map` between the `world` and `base_link` frames, and the two new camera frames attached to the `wrist_link`

Sensor input configuration is equally critical. In this setup, the RGB and depth streams are used directly from the simulated camera, while RGB-D and laser scan messages are explicitly disabled. This choice reflects the structure of the simulated camera plugin, which provides raw `/image_raw` and `/depth/image_raw` topics, rather than pre-synchronized `rgbd` messages. Due to potential timing mismatches between image and depth streams in simulation, approximate time synchronization is enabled by setting the `approx_sync` parameter to true. A sufficiently large synchronization queue is also configured to improve robustness against minor delays. This ensures that corresponding RGB and depth frames can still be paired and processed correctly, even if their timestamps are not perfectly aligned.

Finally, *QoS* policies are adjusted to ensure compatibility with the transient local publishers used by the Gazebo camera plugin. In particular, the `qos_image` and `qos_camera_info` parameters are set to guarantee reliable subscription and message reception, even in presence of startup delays or timing inconsistencies.

Core SLAM and mapping strategy: Beyond these initial settings, further parameter tuning is necessary to tailor RTAB-Map's behavior to a visual-only SLAM pipeline in a static, structured environment. The visual registration strategy is explicitly set to feature-based depth matching by selecting `Reg/Strategy = 0`, and full six degrees of freedom are retained by disabling `Reg/Force3DoF`. Moreover, to allow processing of every frame without filtering based on displacement or rotation, the thresholds `RGBD/LinearUpdate` and `RGBD/AngularUpdate` are set to zero. This results in a denser map and more frequent pose updates, which are especially valuable when operating with a low-speed manipulator arm. The rate at which these new keyframes are processed for loop closure detection is then managed by `Rtabmap/DetectionRate`, which was set to 1 Hz to balance consistency with performance.

Visual processing and loop closure: The maximum valid depth used for feature extraction is capped at 3.5 meters via the `RGBD/MaxDepth` parameter, reducing the influence of noisy far-field measurements. The incremental memory mode (`Mem/IncrementalMemory = true`) is activated to continuously grow the SLAM graph, and the input images are downsampled using a decimation factor of four (`Mem/ImagePreDecimation = 4`) to reduce computational load during real-time operation. Additionally, depth images are compressed using the `.png` format, which is compatible with RTAB-Map's internal processing and avoids decompression warnings. To enhance robustness during registration, the minimum number of inliers required to validate a transform is set to 15 using `Vis/MinInliers`, while the optimization process is configured to discard constraints with an error above 3.0 meters by setting `RGBD/OptimizeMaxError`. Proximity-based loop closure detection is disabled (`RGBD/ProximityBySpace = false`) to prevent false positives in constrained environments.

Loop closures are handled within the same visual SLAM framework without external lidar data, and the SLAM graph is explicitly configured to operate in 3D (`Optimizer/Slam2D = false`). The system also accepts all extracted visual features without applying a maximum cap (`Kp/MaxFeatures = -1`), allowing full use of available visual information.

3D occupancy map generation: Regarding map generation, the occupancy grid is derived directly from depth data (`Grid/FromDepth = true`) and structured in three dimensions by enabling `Grid/3D`. The voxel size is fixed at 0.1 meters (`Grid/CellSize`), providing a balance between resolution and computational efficiency. Ray tracing is disabled (`Grid/RayTracing = false`) to simplify the occupancy update logic.

This comprehensive parameterization enables RTAB-Map to operate robustly in simulation, relying solely on the camera mounted on the robot's wrist and maintaining consistent spatial estimates over time through a carefully controlled TF tree. It forms the foundation of the perception subsystem described in this work.

4.2.3 Visual inspection of the RTAB-Map database

In order to verify the effectiveness and behavior of the SLAM process, the database file generated by RTAB-Map during simulation (`rtabmap.db`) was analyzed using the official RTAB-Map database viewer. This tool provides insight into the structure of the constructed graph, the depth images acquired, and the resulting occupancy grid map.

One of the captured depth images reveals a noticeable geometric distortion: the intersection between the floor and the wall appears skewed rather than orthogonal, deviating from the expected 90-degree angle (Figure 4.5). This visual artifact is indicative of an imperfect depth calibration, likely stemming from the limitations of the simulated RGB-D sensor provided by Gazebo. Such inconsistencies, although relatively common in virtual environments, may reduce the accuracy of spatial representations, especially when planar surfaces are essential for localization and mapping tasks.

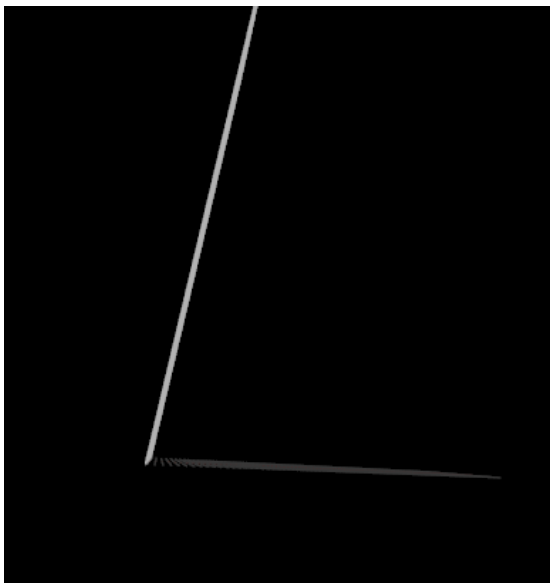


Figure 4.5: Example of a depth image showing distortion due to imperfect sensor calibration in simulation.

The SLAM graph contains a large number of captured nodes (Figure 4.6), confirming that the system correctly processed visual information over time. However, many keyframes appear to have been generated in rapid succession while the robot was stationary. This behavior may stem from slight variations in the camera input due to noise or simulation lag, or from the fact that parameters such as `RGBD/LinearUpdate` and `RGBD/AngularUpdate` were explicitly set to zero. While this configuration enables dense mapping, it may also introduce redundant keyframes that do not contribute additional spatial information.

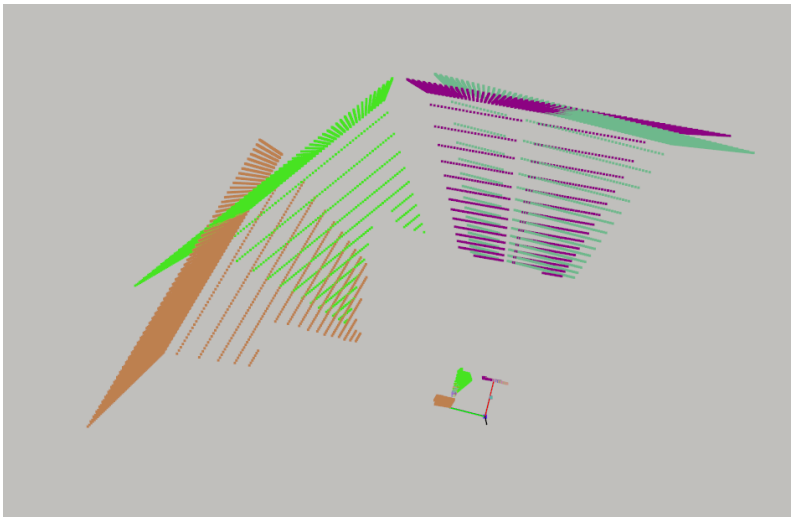


Figure 4.6: High density of keyframes in the RTAB-Map graph, some of which appear to have been generated while the robot was not moving.

The occupancy map constructed by RTAB-Map includes not only the immediate surroundings of the robot but also more distant elements of the environment, such as walls and background structures (Figure 4.7). This extended perception is made possible by increasing the `Grid/RangeMax` parameter to 10 meters, enabling the depth sensor to capture and integrate surfaces at greater distances. The resulting 3D map provides a richer spatial context, which can enhance path planning and obstacle avoidance strategies in complex scenarios.

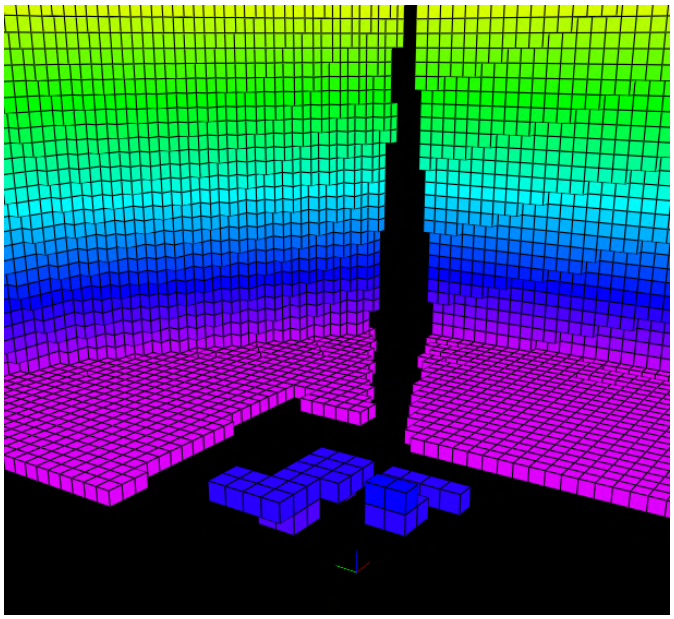


Figure 4.7: 3D occupancy grid generated from depth data, including distant structures captured by extending the sensor's maximum range.

4.2.4 MoveIt 2 integration for OctoMap

To enable collision-aware motion planning in the dynamically reconstructed environment, the external 3D occupancy map produced by RTAB-Map must be made available to the MoveIt 2 planning framework. This integration is achieved by configuring MoveIt's `occupancy_map_monitor` to subscribe to a live point cloud topic published by the simulated RGB-D camera, and to use it to incrementally update an internal OctoMap representation.

MoveIt 2 supports this mechanism through the use of a dedicated YAML configuration file, `sensors_3d.yaml`, which defines one or more 3D perception sources. In this project, a single sensor, referred to as `niryo_depth_camera`, is configured as shown in Appendix 5. The sensor is associated with the plugin `occupancy_map_monitor/PointCloudOctomapUpdater`, which subscribes to the point cloud topic generated by the Gazebo depth camera.

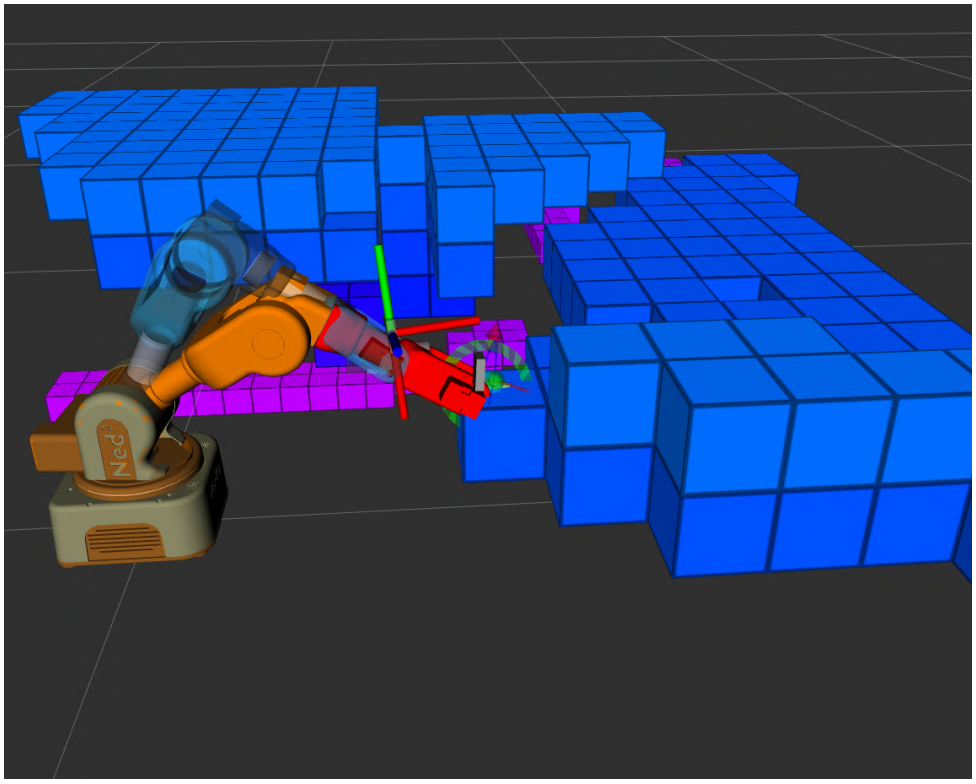


Figure 4.8: Collision detection based on the OctoMap generated from the RTAB-Map point cloud.

The parameter `point_cloud_topic` is set to `/gazebo_depth_camera/points`, ensuring that the 3D data generated by the simulated sensor is correctly consumed by the planner. While two general strategies exist for this integration, subscribing to the filtered map cloud published by the SLAM node (e.g., `/rtabmap/cloud_map`) or subscribing directly to the raw point cloud from the sensor, the latter approach was chosen for this implementation. This choice ensures that the planning scene is updated with the most immediate, lowest-latency sensor information available, making the system highly responsive. The maximum range of the depth sensor is limited to 5 meters using the `max_range` parameter, both to reduce noise and to limit the size of the OctoMap to the workspace relevant for manipulation. Additional parameters, such as `point_subsample`, `padding_offset`, and `padding_scale`, are tuned to balance performance with accuracy, and to account for minor sensor and model imperfections during collision checking.

The field `max_update_rate` is set to 1 Hz, which regulates the frequency at which the OctoMap is updated. This value provides a compromise between responsiveness and computational load in simulation. Finally, a filtered point cloud can optionally be published on the topic `/filtered_cloud` for visualization or debugging purposes.

Once correctly configured, this integration enables MoveIt 2 to plan trajectories that account for the current 3D representation of the environment, including objects reconstructed in real time by the SLAM pipeline. A representative example of this interaction occurs when one of the robot's links enters an occupied region of the OctoMap (Figure 4.8): the system detects the collision and highlights the affected area in red within the RViz interface.

4.3 Reference trajectory generation

The generation of reference trajectories is a critical step in the proposed control pipeline, as it provides the desired end-effector poses in $SE(3)$ that will be tracked by the operational space controller described in the following section.

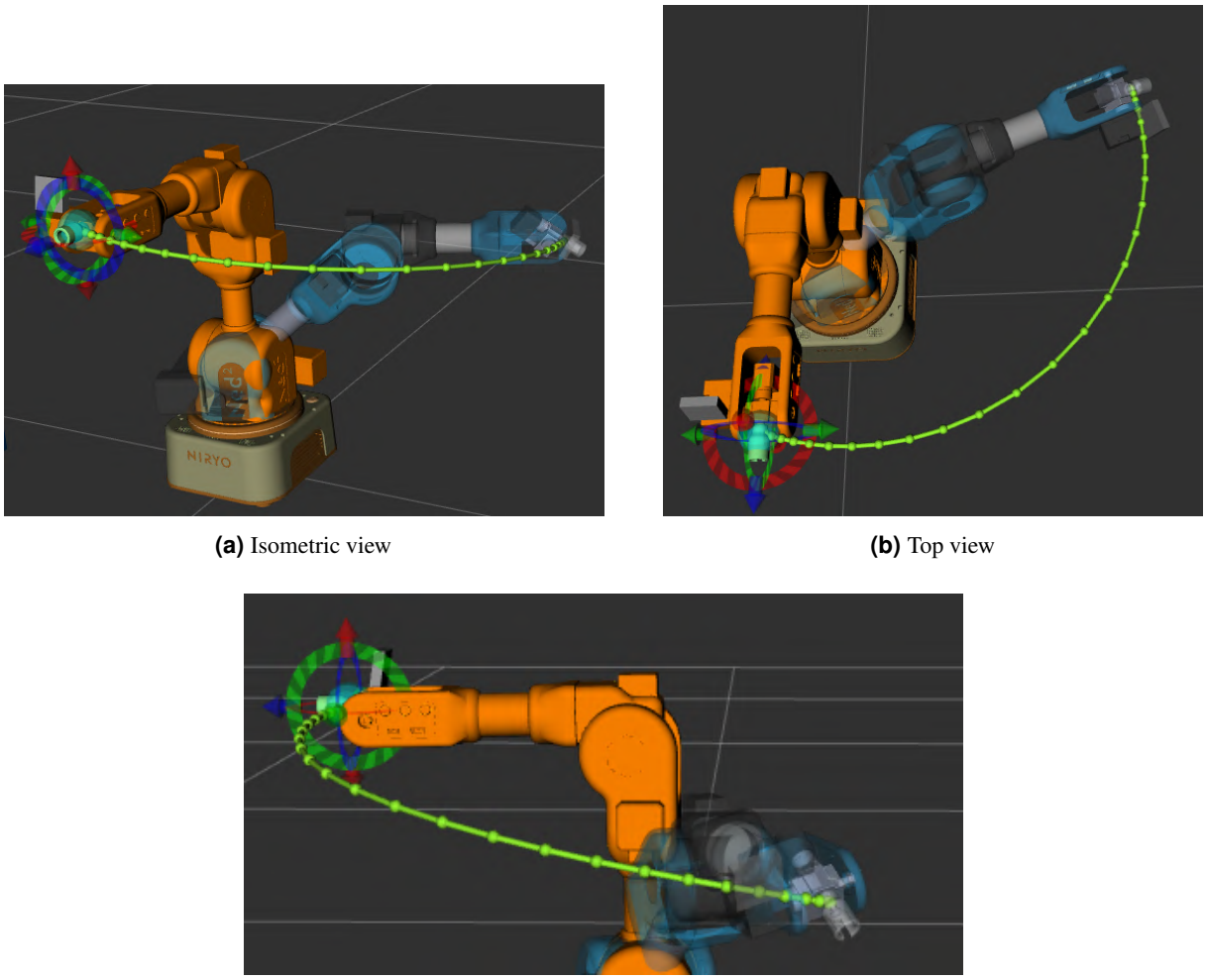
To this end, a dedicated C++ node named `desired_trajectory_sampling` was developed, Appendix 6. This node is based on MoveIt 2's `MoveGroupInterface` and leverages the CHOMP planning pipeline to compute a Cartesian trajectory from the current robot configuration to a predefined target pose. The trajectory planner and visualization components were originally implemented during the previous internship project, and have been extended in this work to enable real-time sampling and storage of the resulting trajectory for closed-loop control purposes. A detailed explanation of the planning pipeline implemented in this C++ file is available in [6].

The reference pose used to initiate trajectory generation is manually specified within the script and expressed in the `world` frame, which serves as the global reference for both the robot and its environment. This design choice ensures that the resulting planned motion is defined consistently with respect to the static scene and the external mapping frame (e.g., `map`), thereby simplifying both the visualization and subsequent control phases. The pose itself is hard-coded as a `geometry_msgs::msg::PoseStamped` message, initialized with position and quaternion orientation components. The use of `world` as the `frame_id` aligns with the standard frame hierarchy adopted in the rest of the simulation and MoveIt 2 configuration. The node samples the planned trajectory at a fixed temporal resolution of 100 ms (10 Hz), computing the end-effector pose at each sampled instant using the robot's forward kinematics. Each sampled pose is then recorded in a plain-text file called `desired_trajectory.txt`. Each line in the `desired_trajectory.txt` file represents a sampled waypoint of the planned end-effector trajectory. These waypoints are expressed as seven real values: the first three columns denote the position in Cartesian space (x, y, z), while the remaining four specify the orientation as a unit quaternion (qx, qy, qz, qw). This combination of position and orientation defines a rigid-body transformation in three-dimensional space. Consequently, the entire reference trajectory can be understood as a discretized path in $\mathrm{SE}(3)$. The resulting file contains 28 discrete waypoints. An extract is provided in Appendix 7 to illustrate its structure.

To integrate this node within the simulation framework, a dedicated launch file was created. This file includes the execution of the `desired_trajectory_sampling` node, which is delayed by 20 seconds using a `TimerAction`. This delay ensures that all essential components, such as MoveIt 2 and the planning scene, are fully initialized, preventing errors in kinematics or collision checking due to incomplete system startup. The python code part that implements the node startup is in Appendix 8

Notably, `use_sim_time` is enabled to synchronize the planner with the simulated clock.

The planned trajectory and its corresponding sampled waypoints are visualized in RViz through the use of `RViz Visual Tools`. The green line represents the continuous end-effector path computed by the CHOMP planner, while green spheres denote the discrete sampled poses that are stored in the `desired_trajectory.txt` file. In total, 28 waypoints are generated, each marking a reference pose to be tracked during execution (Figure 4.9).



(a) Isometric view

(b) Top view

(c) Lateral view

Figure 4.9: Visualization of the sampled trajectory in RViz.

4.4 Operational space controller implementation

The final and most technically sophisticated component of the proposed control architecture is the implementation of a custom *OSC*. This module is responsible for ensuring that the robot's end-effector accurately tracks a desired trajectory in $\mathrm{SE}(3)$, computed a priori through motion planning and sampled at discrete time intervals.

Unlike traditional position or velocity controllers operating in joint space, this controller operates directly in the task space, formulating the control law based on the error between the current and desired pose of the end-effector, while accounting for the full nonlinear dynamics of the manipulator. The

implementation is based on the principles of *inverse dynamics control*, also known as *computed torque control*, and makes extensive use of the Pinocchio library for efficient model-based computation of kinematics and dynamics.

To clearly present both the theoretical foundations and the practical implementation of the proposed control strategy, this section is organized into three main parts: a rigorous mathematical derivation of the control law, starting from the general equations of motion and leading to its final expression formulated in the space of rigid-body transformations; a structured overview of the developed ROS 2 plugin, illustrating how the controller is integrated into the `ros2_control` framework; and finally, a detailed analysis of the core C++ implementation. This modular organization is intended to enhance clarity and traceability, while also ensuring transparency and reproducibility for future developments and research extensions.

4.4.1 Mathematical formulation of the control law

The operational space controller implemented in this work is based on an inverse dynamics approach, commonly referred to as *CTC*. This strategy explicitly compensates for the nonlinear dynamics of the manipulator while enforcing a desired behavior of the end-effector in task space ($\mathrm{SE}(3)$).

The final control input, in the form of joint torques $\boldsymbol{\tau} \in \mathbb{R}^n$, is computed as:

$$\boldsymbol{\tau}^* = \mathbf{M}(\mathbf{q})\, \ddot{\mathbf{q}}_{\text{des}} + \mathbf{N}(\mathbf{q}, \dot{\mathbf{q}}) \tag{4.1}$$

where:

- $\mathbf{q} \in \mathbb{R}^n$ is the vector of joint positions,
- $\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}_{\text{des}}$ are the joint velocities and desired joint accelerations, respectively,
- $\mathbf{M}(\mathbf{q})$ is the joint-space inertia matrix,
- $\mathbf{N}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\, \dot{\mathbf{q}} + \mathbf{g}(\mathbf{q})$ is the vector of nonlinear forces, which includes Coriolis and centrifugal effects via the matrix $\mathbf{C}$, and gravity effects via $\mathbf{g}$.

The desired joint acceleration $\ddot{\mathbf{q}}_{\text{des}}$, in (4.1), is not provided directly, but is computed by inverting the manipulator's differential kinematics. The relationship between joint accelerations and end-effector Cartesian accelerations is given by:

$$\ddot{\mathbf{X}} = \mathbf{J}(\mathbf{q})\, \ddot{\mathbf{q}} + \dot{\mathbf{J}}(\mathbf{q}, \dot{\mathbf{q}})\, \dot{\mathbf{q}} \tag{4.2}$$

where:

- $\ddot{\mathbf{X}} \in \mathbb{R}^6$ is the spatial acceleration of the end-effector, expressed in mixed Cartesian coordinates (linear and angular),
- $\mathbf{J}(\mathbf{q}) \in \mathbb{R}^{6\times n}$ is the geometric Jacobian matrix,
- $\dot{\mathbf{J}}(\mathbf{q}, \dot{\mathbf{q}})$ is the time derivative of the Jacobian.

Solving the (4.2) for the joint accelerations yields:

$$\ddot{\mathbf{q}}_{\text{des}} = \mathbf{J}^{\dagger}(\mathbf{q})\, \left(\ddot{\mathbf{X}}_{\text{cmd}} - \dot{\mathbf{J}}(\mathbf{q}, \dot{\mathbf{q}})\, \dot{\mathbf{q}}\right) \tag{4.3}$$

where $\mathbf{J}^{\dagger}(\mathbf{q})$ denotes the Moore–Penrose pseudo-inverse of the Jacobian. In practice, the pseudo-inverse is computed using a damped least-squares approach to improve numerical stability near singularities. Specifically, the pseudo-inverse $\mathbf{J}^{\dagger} \in \mathbb{R}^{n\times 6}$ is computed as:

$$\mathbf{J}^{\dagger} = \mathbf{J}^{\top}\left(\mathbf{J}\,\mathbf{J}^{\top} + \lambda^2\,\mathbf{I}_6\right)^{-1} \tag{4.4}$$

where $\lambda \in \mathbb{R}$ is a small damping factor (e.g., $\lambda = 0.01$), and $\mathbf{I}_6$ is the 6×6 identity matrix. This formulation ensures robustness in the inversion process by penalizing large joint velocities in ill-conditioned configurations.

This formulation (4.4) allows the control law to enforce a desired Cartesian acceleration $\ddot{\mathbf{X}}_{\text{cmd}}$ at the end-effector level.

This commanded acceleration is computed using a PD+ control strategy:

$$\ddot{\mathbf{X}}_{\text{cmd}} = \ddot{\mathbf{X}}_{d} + \mathbf{K}_p\,\boldsymbol{\xi} + \mathbf{K}_d\,\dot{\boldsymbol{\xi}} \tag{4.5}$$

where:

- $\ddot{\mathbf{X}}_{\text{d}}$ is the desired feedforward acceleration,
- $\boldsymbol{\xi} \in \mathbb{R}^6$ is the pose error in $\mathrm{SE}(3)$ between the current and desired end-effector poses, represented in the Lie algebra $\mathfrak{se}(3)$ via a minimal parametrization;
- $\dot{\boldsymbol{\xi}} \in \mathbb{R}^6$ is the spatial velocity error, i.e., the difference between desired and actual end-effector velocities.
- $\mathbf{K}_p$, $\mathbf{K}_d$ are positive definite gain matrices for proportional and derivative action, respectively.

For clarity, the controller gains are block-diagonal matrices in $\mathbb{R}^{6\times 6}$:

$$K_p = \begin{bmatrix} k_{p,\text{lin}} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{p,\text{lin}} & 0 & 0 & 0 & 0 \\ 0 & 0 & k_{p,\text{lin}} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{p,\text{ang}} & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{p,\text{ang}} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{p,\text{ang}} \end{bmatrix}$$

$$K_d = \begin{bmatrix} k_{d,\text{lin}} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{d,\text{lin}} & 0 & 0 & 0 & 0 \\ 0 & 0 & k_{d,\text{lin}} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{d,\text{ang}} & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{d,\text{ang}} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{d,\text{ang}} \end{bmatrix} \tag{4.6}$$

where $k_{p,\text{lin}}$ and $k_{d,\text{lin}}$ denote the proportional and derivative gains for the translational motion components, and $k_{p,\text{ang}}$, $k_{d,\text{ang}}$ are the corresponding gains for the rotational degrees of freedom.

A common rule of thumb, valid for systems with double-integrator dynamics, suggests choosing the derivative gain as: $K_d = 2\sqrt{K_p}$. This relationship ensures a critically damped behavior of the closed-loop system, avoiding overshoots while achieving fast convergence. Although the full dynamics of a robotic manipulator do not exactly match a double integrator, this heuristic provides a useful starting point for

empirical gain tuning. In practice, increasing K_p helps reduce steady-state tracking error, while increasing K_d mitigates oscillations caused by aggressive corrections. The final gain values used in this work were selected through extensive simulation trials, balancing responsiveness and stability, see Chapter 6.

The error $\boldsymbol{\xi}$ is computed by projecting the relative rigid transformation between the current and desired end-effector poses into the Lie algebra $\mathfrak{se}(3)$ using the matrix logarithm, which ensures a consistent treatment of translational and rotational discrepancies. Formally:

$$\boldsymbol{\xi} = \log\left(\mathbf{X}_d^{-1}\,\hat{\mathbf{X}}\right) \tag{4.7}$$

where $\mathbf{X}_d, \hat{\mathbf{X}} \in \mathrm{SE}(3)$ are the desired and estimated (current) homogeneous transformation matrices of the end-effector, respectively.

This control formulation allows the system to produce torque commands that generate a smooth, dynamically consistent motion in operational space, while accounting for the nonlinearities of the manipulator model. The use of model-based dynamics through *Pinocchio* ensures efficient and accurate computation of all required quantities, including $\mathbf{M}(\mathbf{q})$, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$, $\mathbf{g}(\mathbf{q})$, $\mathbf{J}(\mathbf{q})$ and $\dot{\mathbf{J}}(\mathbf{q}, \dot{\mathbf{q}})$.

4.4.2 Plugin structure and file

The implementation of the operational space controller as a real-time torque-based plugin for `ros2_control` requires a well-defined package structure and adherence to the plugin interface conventions established in the ROS 2 control framework.

The overall architecture of the controller package `op_space_controller` has already been introduced in Chapter 3, in the context of the broader system architecture. In this section, we now provide a detailed analysis of each file involved in the definition, configuration, and integration of the controller as a reusable ROS 2 component. The design follows the official guidelines presented in the ROS 2 controllers documentation [39]. Each component is described in detail below, except for the `op_space_controller.cpp` implementation file, which will be examined comprehensively in the next subsection.

The **`CMakeLists.txt`** file defines the build and installation logic for the `op_space_controller` package. It is responsible for compiling the plugin as a shared library, linking it against the necessary ROS 2 and third-party dependencies, and exporting it as a reusable component in the ROS 2 control framework:

- The file starts by **setting the minimum CMake** version and project name, and it conditionally enables additional compiler warnings to encourage code quality and portability. All required ROS 2 dependencies are explicitly declared using `find_package`, including core packages such as `rclcpp`, `controller_interface`, and `hardware_interface`, as well as external libraries like `Pinochio`, `Eigen3`, and `tf2_eigen`.
- **The controller source file** (`op_space_controller.cpp`) is compiled as a shared library, and appropriate include directories are set to ensure both development and installation compatibility. The linking step uses `ament_target_dependencies` for ROS-related packages and `target_link_libraries` for system libraries.

- Special attention is paid to **symbol visibility** by defining the macro `OP_SPACE_CONTROLLER_BUILDING_DLL`, which interacts with the `visibility_control.h` header to ensure proper symbol exportation across platforms.
- To **expose the controller** as a ROS 2 plugin, the file `op_space_controller_plugin.xml` is registered using `pluginlib_export_plugin_description_file`. Furthermore, installation directives are provided for the compiled library, header files, and configuration files, allowing the controller to be reused and distributed as a standard ROS 2 component.

A complete version of the `CMakeLists.txt` file is provided in Appendix 9.

The **`package.xml`** file serves as the manifest of the `op_space_controller` package, providing essential metadata such as its name, version, description, license, and maintainer information. It is structured according to the ROS 2 package format (version 3), ensuring compatibility with the `ament` build system.

In the `<license>` tag, the controller package declares the use of the *Apache License 2.0*, a widely adopted permissive open-source license. This choice ensures that the developed software can be freely used, modified, and redistributed by others, including for commercial purposes, while also providing legal protections to the original author. In contrast to more restrictive copyleft licenses (e.g., GPL), the Apache 2.0 license does not require derivative works to adopt the same license, offering greater flexibility for integration into diverse robotic systems and software stacks. Explicitly declaring a license, rather than omitting it, is a best practice in open-source software development, as it removes legal ambiguity and encourages code reuse by making the terms of use clear to potential users and contributors. In the context of this project, the Apache License 2.0 facilitates future research, experimentation, and industrial adoption of the operational space controller developed during this work.

All runtime and build-time dependencies are explicitly declared. These include core ROS 2 packages such as `rclcpp`, `controller_interface`, and `hardware_interface`, as well as third-party libraries like `pinocchio`, `tf2_ros`, and `eigen3_cmake_module`. This declaration allows ROS tools such as `rosdep` and `colcon` to resolve and install dependencies automatically.
In the `<export>` section, the controller plugin is registered with the ROS 2 plugin infrastructure. Specifically, the line `<controller_interface plugin="${prefix}/op_space_controller_plugin.xml"/>` ensures that the plugin loader can discover and load the controller during runtime. This file also includes testing dependencies, such as `ament_lint_auto` and `ament_lint_common`, which support automatic code quality checks. These dependencies are particularly useful during development but do not affect runtime behavior.
The full `package.xml` manifest is reported in Appendix 10.

The file **`op_space_controller_plugin.xml`**, located in the root of the package, is essential for registering the controller as a plugin within the `ros2_control` framework. It defines the necessary metadata for the *pluginlib* system to correctly instantiate the controller at runtime.

The XML structure, Appendix 11, contains a single `<library>` element, whose `path` attribute refers to the compiled shared library (`op_space_controller.so`). Inside this element, a `<class>` tag specifies three key attributes:

- `name`: a unique identifier used in controller configuration files (`ros2_controllers.yaml`);
- `type`: the fully qualified C++ class name that implements the plugin;

- `base_class_type`: the ROS 2 interface that the plugin derives from, in this case, `controller_interface::ControllerInterface`.

The embedded `<description>` tag provides a short documentation string, which may be used by tools such as `pluginlib_tools` to display plugin information. This plugin declaration is automatically exported during the build process via the call to `pluginlib_export_plugin_description_file()` in the `CMakeLists.txt` file. It also appears in the `<export>` section of the `package.xml`, ensuring that the controller can be discovered and dynamically loaded at runtime by ROS 2 control components. This structure adheres to the conventions established in the ROS 2 controller plugin interface and ensures seamless integration of the custom torque-based controller with the broader ROS 2 ecosystem.

The **`visibility_control.h`** header file is a standard component in ROS 2 C++ packages that is used to manage symbol visibility when compiling shared libraries. Its purpose is to ensure cross-platform compatibility, particularly with Windows, where class methods and functions must be explicitly marked for export from a dynamic-link library (DLL) to be accessible by other applications. The file uses preprocessor directives to define a set of macros, such as `OP_SPACE_CONTROLLER_PUBLIC` and `OP_SPACE_CONTROLLER_LOCAL`. The `OP_SPACE_CONTROLLER_PUBLIC` macro is then used in the controller's header file (`op_space_controller.hpp`) to decorate the class declaration. This ensures that the controller's class is correctly exported from the shared library, making it discoverable and loadable by the ROS 2 plugin system at runtime. As this file contains standard boilerplate code, its full content is omitted for brevity.

The **`op_space_controller.hpp`** header file defines the interface and internal structure of the custom operational space controller. This component inherits from the standard `controller_interface::ControllerInterface` class provided by the ROS 2 control framework and is declared within the dedicated namespace `op_space_controller`. It contains the class declaration, member variables, ROS 2 interface method signatures, and the supporting structures needed to implement a fully functional torque-based controller plugin within the `ros2_control` framework and *Pinocchio* library. The class is annotated with visibility macros such as `OP_SPACE_CONTROLLER_PUBLIC` to ensure proper symbol export when building shared libraries, especially on Windows systems.

This header plays a crucial role in defining the controller's lifecycle and integration with the real-time ROS 2 infrastructure. It declares all mandatory lifecycle methods including:

- `on_init()`, used to allocate basic structures;
- `on_configure()`, for reading and parsing user-defined ROS parameters;
- `on_activate()` and `on_deactivate()`, to enable and disable the control loop;
- `on_cleanup()` and `on_error()`, for graceful teardown and recovery;
- `update()`, the real-time method where the control law is evaluated and joint torques are computed and dispatched.

The detailed explanation of these methods, which were implemented in `op_space_controller.cpp`, is given in Section 4.4.3.

Furthermore, the file specifies the command and state interface configurations, indicating which joint interfaces are required to operate the controller correctly (namely, joint positions, velocities, and efforts).

A key feature of this controller is its integration with the Pinocchio library. The file includes the necessary headers from the `Pinocchio` library to allow for symbolic modeling of the robot's dynamics. The header declares a `pinocchio::Model` object to represent the robot's kinematic and dynamic structure, which is loaded from a URDF file specified via a ROS parameter, `urdf_file_path_`. A corresponding `pinocchio::Data` structure is instantiated to hold intermediate computation results, allocated dynamically via `std::shared_ptr`. The controller also identifies the end-effector frame within the Pinocchio model using a `pinocchio::FrameIndex`, enabling precise computation of task-space quantities such as pose, velocity, and Jacobians.

To obtain the current end-effector pose from the SLAM-based localization system, the controller relies on the TF2 framework. The header includes two shared pointers, `tf_buffer_` and `tf_listener_`, used to query real-time transformations between frames such as `map` and `tool_link`. This allows the controller to use a vision-based estimate of the robot's pose as feedback, rather than relying exclusively on forward kinematics.

The header also defines a custom `struct EEWaypoint`, which encapsulates the *desired pose*, *velocity*, and *acceleration* of the end-effector at each time step of the trajectory. The trajectory itself is stored as a vector of these waypoints, loaded from a file and indexed during runtime to provide time-synchronized motion references. This vector of waypoints is stored in the protected member `desired_trajectory_`, while the index of the currently active waypoint is tracked via `current_trajectory_index_`.

Internally, the controller maintains the robot's joint state and control output through three `Eigen::VectorXd` vectors: `q_`, `dq_`, and `tau_`, which respectively represent the current joint positions, joint velocities, and the torque commands computed at each control step. These variables act as the interface between the low-level measurements provided by the hardware abstraction layer and the high-level dynamic computations performed using the *Pinocchio* library. The controller's behavior is further governed by a set of runtime-configurable parameters, which are loaded from the ROS parameter server during the `on_configure()` phase of the lifecycle. These include:

- The names of the joints involved in the control loop (`joint_names_`),
- The identifiers of the base link (`base_link_name_`),
- The identifiers of the end-effector links (`ee_link_name_`),
- The name of the world reference frame used for localization (`map_frame_name_`).

Additional parameters specify the path to the URDF file used to build the internal dynamic model with Pinocchio (`urdf_file_path_`) and the absolute path to the text file containing the sampled desired trajectory (`trajectory_file_path_`). Finally, four scalar values define the feedback gains used in the operational space control law:

- Two for the translational motion (`kp_linear_` and `kd_linear_`),
- Two for the rotational part (`kp_angular_` and `kd_angular_`).

To access the hardware-level information and issue torque commands in real time, the controller also declares specific interface handles, namely `state_if_position_handles_` and `state_if_velocity_handles_` for reading joint states, and `cmd_if_effort_handles_` for writing control

efforts to the actuators. These are implemented using the `LoanedStateInterface` and `LoanedCommandInterface` abstractions, which represent non-owning, real-time safe handles to hardware resources managed by the `ros2_control` framework. By using these interfaces, the controller can safely access and modify joint state and command data during the real-time execution loop without incurring memory allocation or ownership issues.

In addition to these core attributes, the header file defines two helper methods: `read_trajectory_from_file()`, responsible for parsing the desired end-effector trajectory from disk, and `wait_for_transform()`, which ensures that the required transformation between coordinate frames is available before the controller proceeds with execution.

Taken together, these declarations encapsulate the internal state, configuration parameters, hardware interfaces, and supporting utilities necessary to implement the proposed operational space controller. The concrete logic that makes use of these declarations is implemented in the corresponding source file, which will be discussed in detail in Section 4.4.3. For the complete source code of this header file, refer to Appendix 12.

The **`op_space_controller_parameters.yaml`** YAML file was initially created to isolate the runtime parameters of the `op_space_controller` into a dedicated configuration resource, in line with ROS 2 best practices for modular and reusable component design. The original intention was to include this file from within the main controller configuration file `ros2_controllers.yaml`, using the `from_file` directive provided by the `ros2_control` framework.

However, this mechanism failed to operate correctly at launch time, likely due to path resolution issues or limitations in the underlying ROS 2 parameter parsing logic. As a result, the full contents of `op_space_controller_parameters.yaml` were manually copied into the `ros2_controllers.yaml` file, which resides in the `config` directory of the `niryo_moveit2_config` package. Despite this workaround, maintaining a standalone YAML file is still beneficial for documentation clarity and potential reuse in future deployments or benchmarking scenarios.

This configuration defines all the key runtime parameters needed to initialize and operate the controller: the list of controlled joints, the command interface type (`"effort"`), the file paths to the URDF model and desired trajectory, the names of relevant reference frames (`base_link`, `tool_link`, and `world`), as well as the scalar gains used by the control law for both translational and rotational feedback. Although the file is not loaded directly at runtime, its structure faithfully reflects the expected configuration syntax of the `ros2_control` controller manager. For completeness, its full content is reported in Appendix 13.

The **`niryo_ned2_pinoc.urdf`** file contains a complete and flattened URDF model of the Niryo Ned2 robot, in which all XACRO macros and parameter expressions (such as `$(arg ...)` and `$(eval ...)`) have been expanded into static, concrete values. The resulting description is a standalone XML file that can be directly parsed by the `Pinocchio` library without requiring any pre-processing. The motivation for generating this static URDF stems from the fact that, while tools like MoveIt and Gazebo are capable of parsing XACRO or parameterized robot descriptions at runtime (typically via launch files that resolve substitutions and arguments), `Pinocchio` expects a fully expanded URDF file as input. As such, this model is generated offline using the `xacro` command-line tool and saved as a self-contained file. The file includes the complete kinematic structure of the robot, including links, joints, joint limits, inertial properties, and transmission definitions, as required by `ros2_control`. However, unlike the robot description loaded in simulation or visualization contexts, this file is used exclusively within the custom controller implementation to instantiate the `pinocchio::Model` object and enable the computation of

forward/inverse dynamics, Jacobians, and spatial transformations. Notably, this URDF is not referenced by any MoveIt or Gazebo launch files, nor is it published to the ROS parameter server. Instead, it is directly read at runtime by the controller, based on the `urdf_file_path` parameter defined in the configuration YAML. This approach ensures full determinism and avoids any dependency on launch-time substitution mechanisms.

In summary, this package provides all the infrastructure required to define, build, and register a custom torque controller in ROS 2. The next subsection will provide a deep dive into the implementation of the main controller logic within the `op_space_controller.cpp` file.

4.4.3 Detailed explanation of the op_space_controller_cpp file

This section presents a detailed walkthrough of the `op_space_controller.cpp` file, which contains the full implementation of the operational space controller class declared in the corresponding header. The file is responsible for interfacing with the ROS 2 control infrastructure, initializing and configuring all internal data structures, parsing robot and trajectory data, and executing the control loop in real time.

The file begins by including the main header `op_space_controller.hpp`, along with standard C++ and ROS 2-specific headers for logging, hardware abstraction, and plugin registration. It also imports a set of modules from the `Pinocchio` library, which enable symbolic computation of kinematic and dynamic quantities such as Jacobians, inertia matrices, and joint torques. Finally, the class is registered as a plugin using the `PLUGINLIB_EXPORT_CLASS` macro, allowing it to be discovered at runtime by the ROS 2 controller manager. The overall class is encapsulated in the `op_space_controller` namespace, and its methods are implemented in a modular and lifecycle-aware fashion, consistent with the ROS 2 controller interface specifications. This ensures proper integration into a real-time control loop and compatibility with the `ros2_control` plugin architecture.

In the remainder of this section, we provide an in-depth explanation of each method in the class, including their purpose, the internal variables affected, and how they contribute to the execution of the controller. To improve readability, the description is organized into logical groups: the constructor and internal state initialization, lifecycle methods, helper functions, and finally the `update()` method, where the control law is computed and applied. Each method is accompanied by an inline Doxygen-style docstring, presented immediately after its explanation for clarity and ease of reference.

Constructor and Initial state: The class `OpSpaceController` begins with a default constructor, whose primary role is to perform minimal initialization of the controller object and ensure that key internal flags are set to a known and consistent state. The constructor explicitly inherits from the base class `controller_interface::ControllerInterface`, in accordance with the `ros2_control` architecture for real-time controllers.

Internally, three boolean and numerical flags are initialized:

- `pinocchio_model_loaded_` is set to `false` to indicate that the Pinocchio model has not yet been built from the URDF file. This flag is later updated in the `on_configure()` method once the model is successfully parsed.
- `trajectory_active_` is set to `false`, denoting that no desired trajectory has yet been loaded or activated. This flag is used during the control loop to determine whether a reference motion is currently being tracked.

- `current_trajectory_index_` is initialized to zero and is used to keep track of the active waypoint within the loaded trajectory vector. This index is incremented at each control cycle once the trajectory is active.

At this stage, no memory allocation or heavy computation takes place; the constructor simply ensures that the controller object is brought to a consistent baseline state. The core logic of loading parameters, allocating data structures, and establishing hardware interfaces is deferred to the standard ROS 2 lifecycle methods such as `on_init()` and `on_configure()`, which will be discussed in the subsequent sections.

```
/**
 * @brief Default constructor for the OpSpaceController class.
 * Initializes the OpSpaceController object, setting the initial state of ...
    internal flags
 * such as pinocchio_model_loaded_, trajectory_active_, and ...
    current_trajectory_index_.
 * Inherits from controller_interface::ControllerInterface.
 */
```

The **`on_init()`** method is the first lifecycle callback executed when the controller is instantiated. Its primary purpose is to declare all required ROS 2 parameters and immediately retrieve those that are essential for configuring the controller's interfaces. Specifically, the method begins by declaring two critical parameters: `joints`, which lists the names of the actuated joints, and `command_interface_type`, which specifies the control interface type (set by default to `"effort"`).

These parameters are then retrieved via the controller's internal node interface. If either is missing or incorrectly typed, the method logs a detailed error and returns `CallbackReturn::ERROR`, preventing the controller from progressing in its lifecycle. This early validation step ensures that mis-configurations in the YAML file are caught as soon as possible. If both parameters are correctly declared and retrieved, the controller proceeds to declare additional parameters with default values. These include the file path to the URDF model, the names of the base and end-effector links, the reference frame for SLAM-based localization (`map_frame_name`), and trajectory-related parameters such as the file path and sampling time step. The method also sets default values for the linear and angular proportional and derivative gains.

A confirmation log message is printed upon successful declaration of all parameters. This approach ensures that the controller is properly configured before activation, and complies with the lifecycle design principles of the `controller_interface` API.

```
/**
 * @brief Returns the state interface configuration for the OpSpaceController.
 * This method constructs and returns a ...
    controller_interface::InterfaceConfiguration
 * object specifying the required state interfaces for each joint managed by ...
    the controller.
 * For every joint name in joint_names_, it adds both the "/position" and ...
    "/velocity" interfaces to the configuration.
 * @return controller_interface::InterfaceConfiguration
 * The configuration specifying individual position and velocity state ...
    interfaces for all controlled joints.
 */
```

Command and State Interface configuration: The `OpSpaceController` must explicitly declare which joint interfaces it intends to use for both commanding the robot and reading its state. This is achieved by overriding the virtual methods `command_interface_configuration()` and `state_interface_configuration()`, which are invoked by the `ros2_control` infrastructure during the controller lifecycle. The `command_interface_configuration()` method returns a list of joint interfaces required to send control commands. In this case, it specifies one command interface per joint, dynamically constructed using the joint name and the type specified in the YAML file (typically set to `"effort"`). Conversely, the `state_interface_configuration()` method declares the interfaces from which the controller expects to receive feedback about the robot's state. For each joint, both `position` and `velocity` interfaces are added, allowing the controller to access the full joint state required for dynamics computation and feedback control.

These methods return instances of the `InterfaceConfiguration` structure with the type set to `INDIVIDUAL`, meaning that the controller is explicitly listing each interface it requires. This fine-grained declaration is necessary for precise allocation of hardware resources by the controller manager.

```
/**
 * @brief Returns the configuration for the command interfaces required by ...
    the controller.
 *
 * This method constructs and returns a ...
    controller_interface::InterfaceConfiguration object
 * specifying the individual command interfaces needed for each joint ...
    controlled by the
 * OpSpaceController. For each joint name in the joint_names_ vector, it ...
    appends the
 * command_interface_type_ (e.g., "position", "velocity") to form the full ...
    interface name.
 *
 * @return controller_interface::InterfaceConfiguration
 * The configuration object containing the type and names of required command ...
    interfaces.
 */
```

```
/**
 * @brief Returns the state interface configuration for the OpSpaceController.
 * This method constructs and returns a ...
    controller_interface::InterfaceConfiguration
 * object specifying the required state interfaces for each joint managed by ...
    the controller.
 * For every joint name in joint_names_, it adds both the "/position" and ...
    "/velocity" interfaces to the configuration.
 * @return controller_interface::InterfaceConfiguration
 * The configuration specifying individual position and velocity state ...
    interfaces for all controlled joints.
 */
```

The **`on_configure()`** method is invoked during the controller's lifecycle transition into the `CONFIGURING` state. Its purpose is to initialize all the resources and internal data needed before real-time execution begins. This includes reading ROS 2 parameters, loading the robot model, preparing

transform tools, and validating data consistency.

The method first retrieves parameters from the ROS 2 parameter server, including:

- The absolute path to the URDF file used by Pinocchio (`urdf_file_path`);
- The names of the base link, end-effector link, and SLAM reference frame;
- The control gains for both translational and rotational motion;
- The path to a text file containing the sampled trajectory (`trajectory_file_path`) and its sampling interval.

Subsequently, the robot model is parsed and loaded into a `Pinocchio::Model`. The gravity vector is explicitly set to align with ROS conventions (Z axis pointing upward), and the `Pinocchio::Data` structure is initialized. The method verifies that the specified end-effector frame is present in the model and stores its frame ID for use during control computations.

A `tf2::Buffer` and `tf2::TransformListener` are then instantiated to enable transform lookups between the world frame and the robot base. These are necessary for computing the desired end-effector pose in the SLAM frame at runtime.

If a trajectory file is specified, it is parsed using the internal `read_trajectory_from_file()` method, explained in the next section, and the data is stored in memory. The controller will later use this data during the `update()` phase. If no file is provided or the file is invalid, the controller logs a warning and disables trajectory tracking. To finalize the configuration, the controller allocates and zero-initializes the vectors `q_`, `dq_`, and `tau_` used for joint positions, velocities, and control torques, respectively. These are sized based on the Pinocchio model's degrees of freedom.

A critical final step is the *verification of joint name ordering* between the ROS 2 parameter list (`joint_names_`) and the internal order used by Pinocchio. If the two orders differ, dynamic computations (e.g., Jacobians and torques) would be applied incorrectly. The controller detects this mismatch and aborts the configuration process with a fatal log if needed.

```
/**
 * @brief Configures the OpSpaceController by reading parameters, loading the ...
    robot model, and initializing resources.
 *
 * This method is called during the lifecycle transition to the "configuring" ...
    state. It performs the following steps:
 * 1. Reads required parameters from the ROS2 parameter server, including ...
    file paths, link names, and control gains.
 * 2. Loads the robot's URDF model into a Pinocchio model and sets up the ...
    gravity vector.
 * 3. Initializes the Pinocchio data structure and checks for the existence ...
    of the end-effector frame.
 * 4. Sets up the TF2 buffer and listener for frame transformations.
 * 5. Loads a desired trajectory from file if specified, and stores it for ...
    later use.
 * 6. Initializes Eigen vectors for joint positions, velocities, and torques ...
    based on the loaded model.
 * 7. Verifies that the order of joint names from ROS parameters matches the ...
    internal Pinocchio model order.
 *
```

```
 * If any critical step fails (e.g., missing parameters, URDF/model loading ...
     errors, or joint order mismatch),
 * the method logs an error and returns CallbackReturn::ERROR to prevent the ...
     controller from activating.
 * If all steps are successful, it returns CallbackReturn::SUCCESS, ...
     indicating that the controller is ready for activation.
 * @return controller_interface::CallbackReturn SUCCESS if configuration is ...
     successful, ERROR otherwise.
 */
```

The method **`read_trajectory_from_file()`** is invoked during the `on_configure()` phase to load a discretized Cartesian trajectory from a user-provided text file. This trajectory will later be tracked in real time by the operational space controller. The expected format of the input file consists of a series of lines, each containing seven floating-point values: `x y z qx qy qz qw`. These represent the desired end-effector position and orientation (as a quaternion) for each waypoint. Lines starting with the `#` symbol are treated as comments and ignored. The first encountered comment is assumed to be a header and skipped accordingly. Each parsed waypoint is stored in an internal `EEWaypoint` structure, which includes:

- The desired pose (as a `pinocchio::SE3` transformation),
- The desired velocity and acceleration (initialized to zero),
- And the time-from-start, which is computed incrementally using the member variable `trajectory_sample_dt_`.

The function checks for file validity and handles common error cases such as:

- File not found or unreadable;
- Malformed lines with incorrect number of values;
- Empty files or files without any valid data after the header.

In all such cases, appropriate error logs are generated and `false` is returned. If the file is valid and at least one waypoint is successfully parsed, the internal buffer `desired_trajectory_` is populated and the function returns `true`. This design ensures robustness to user input and a clear separation between file parsing and trajectory tracking logic.

```
/**
 * @brief Reads a trajectory from a file and populates the ...
     desired_trajectory_ member.
 *
 * The file is expected to contain waypoints, each specified on a separate ...
     line with 7 values:
 * x y z qx qy qz qw
 * representing the position (x, y, z) and orientation as a quaternion (qx, ...
     qy, qz, qw).
 * Lines starting with '#' are treated as comments and skipped. The first ...
     comment line is considered a header.
 * The time_from_start for each waypoint is reconstructed using the class ...
     member trajectory_sample_dt_.
```

```
 * Velocity and acceleration are set to zero for each waypoint.
 *
 * @param file_path The path to the trajectory file to read.
 * @return true if at least one valid waypoint is loaded or if no file is ...
     specified; false otherwise.
 *
 * @note
 * - If the file cannot be opened, is empty, or contains only invalid lines, ...
     an error is logged and false is returned.
 * - If no file is specified (empty file_path), no waypoints are loaded and ...
     true is returned.
 * - Logs the number of loaded waypoints and the estimated trajectory duration.
 */
```

The **`on_activate()`** method is invoked during the controller's transition to the `ACTIVE` state in the ROS 2 lifecycle. Its primary goal is to ensure that all required hardware interfaces are successfully retrieved and bound, that internal state vectors are correctly initialized, and that trajectory execution is properly reset or initiated.

The procedure follows a well-defined sequence:

1. **Cleanup of stale handles.** All previously bound hardware interface handles (`state_if_position_handles_`, `state_if_velocity_handles_`, `cmd_if_effort_handles_`) are cleared to avoid conflicts during reactivation. This step is especially important when switching from another controller, such as the Joint Trajectory Controller (JTC), to the OSC, as lingering handles could lead to undefined or conflicting behavior. A dedicated section later in this chapter explains the procedure for deactivating the JTC and safely transitioning to OSC.

2. **Validation of configuration state.** The method checks that the list of joint names has been successfully loaded and that the Pinocchio model was correctly initialized in the `on_configure()` step. If either is missing or invalid, activation is aborted.

3. **Acquisition of state interfaces.** The method retrieves position and velocity state interfaces for each joint using ROS 2 helper functions. These are stored in ordered vectors to match the joint sequence declared in the configuration.

4. **Acquisition of command interfaces.** The controller then attempts to bind to the command interfaces of the specified type (e.g., `effort`), ensuring that the number and order match expectations.

5. **Initialization of dynamic state.** Internal vectors such as `tau_` (joint torque command) are zeroed out. The initial joint positions `q_` and velocities `dq_` are also read from the hardware to set a consistent starting point for control.

6. **Trajectory setup.** If a trajectory was loaded during configuration, the trajectory index is reset and the execution state is marked as active. Unlike traditional time-parameterized controllers, this implementation does not rely on time stamps to advance through the trajectory. Instead, the transition to the next waypoint occurs when the norm of the current Cartesian error falls below a predefined threshold (i.e., 1 mm). This ensures more robust waypoint tracking based on actual convergence rather than elapsed time.

This method serves as the critical bridge between controller setup and execution. Its correct operation ensures that the controller can seamlessly enter the real-time loop with valid and synchronized state.

```
/**
 * @brief Activates the OpSpaceController.
 *
 * This method is called when the controller transitions to the "active" ...
    state in the ROS 2 lifecycle.
 * It performs the following steps:
 *   1. Clears any previously stored interface handles to ensure a clean ...
    activation.
 *   2. Checks that required parameters (such as joint_names_) and the ...
    Pinocchio model have been loaded.
 *   3. Retrieves and orders state interfaces for joint positions and ...
    velocities, verifying their count.
 *   4. Retrieves and orders command interfaces of the specified type (e.g., ...
    "effort"), verifying their count.
 *   5. Initializes or resets Eigen vectors for joint state and command ...
    variables.
 *   6. Reads the initial joint state to populate internal state vectors.
 *   7. Initializes or resets trajectory tracking state, starting trajectory ...
    execution if a trajectory is loaded.
 *
 * If any step fails, an error is logged and the activation is aborted.
 *
 * @param previous_state The previous lifecycle state before activation.
 * @return controller_interface::CallbackReturn::SUCCESS if activation succeeds,
 *         controller_interface::CallbackReturn::ERROR otherwise.
 */
```

The **`on_deactivate()`** method is invoked during the transition of the controller to the `INACTIVE` state, in accordance with the ROS 2 lifecycle paradigm. Its primary objective is to halt all ongoing control activity in a safe and deterministic manner, while ensuring that hardware resources are left in a consistent state.

The function executes the following key operations:

- **Trajectory deactivation.** The controller disables trajectory tracking by setting the internal flag `trajectory_active_` to `false`. This halts the progression through the desired trajectory and prevents further control commands based on outdated motion plans.
- **Safe shutdown of joint commands.** If the command interface handles (`cmd_if_effort_handles_`) are valid and their size matches the number of controlled joints, the method issues zero-effort commands to all joints. This is crucial to prevent unintended motion or residual torque application once the controller is deactivated. Exception handling ensures that any invalid or already-released handles do not result in runtime crashes.
- **Clearing of hardware interface references.** All internal references to state and command interfaces are cleared, including those for joint positions, velocities, and efforts. It is important to note that this step does not release the interfaces at the controller manager level, but merely cleans up the controller's internal references, as required by the `ControllerInterface` contract.

- **Diagnostic logging.** Informative messages are published to the ROS 2 logging system at each stage of the deactivation process, providing insight into the controller's internal state and facilitating debugging or system introspection.

This method is essential for maintaining the integrity of the control pipeline, particularly in scenarios where multiple controllers are being switched at runtime. By ensuring that all joint commands are neutralized and internal resources are cleared, the controller is left in a well-defined state, ready for potential reconfiguration or reactivation.

```
/**
 * @brief Callback invoked when the controller is deactivated.
 *
 * This method is called during the transition to the "inactive" state of the ...
    controller's lifecycle.
 * It performs the following actions:
 * - Logs the deactivation event.
 * - Stops any active trajectory tracking by setting `trajectory_active_` to ...
    false.
 * - Sends safe commands (zero effort) to all controlled joints, if the ...
    command handles are valid and match the expected number of joints.
 *   - Logs errors or warnings if the handles are invalid or mismatched.
 * - Clears all internal references to hardware interface handles (position, ...
    velocity, effort).
 *   - Note: This does not release the interfaces from the controller ...
    manager, but only cleans up internal references.
 * - Logs the completion of the cleanup and deactivation process.
 * @return controller_interface::CallbackReturn Returns SUCCESS if ...
    deactivation completes without critical errors.
 */
```

The **`on_cleanup()`** method is executed when the controller transitions to the `CLEANING UP` state as part of the ROS 2 managed lifecycle. Its purpose is to release internal resources, reset controller state, and prepare the controller for potential re-initialization or permanent shutdown.

The method performs the following steps:

- **Trajectory reset.** The vector storing the desired trajectory (`desired_trajectory_`) is cleared, and the internal flag `trajectory_active_` is explicitly set to `false`, ensuring that any residual motion plan is discarded.

- **Pinocchio resource release.** If the Pinocchio model has been successfully loaded (as indicated by the `pinocchio_model_loaded_` flag), the method explicitly releases associated resources. In particular, the `data_` object, which depends on the loaded model, is reset using the `std::shared_ptr::reset()` operation. This is essential to avoid memory leaks and maintain a clean separation between lifecycle states.

- **Clearing interface handles.** All stored references to hardware interface handles are cleared. These include state interfaces for joint position and velocity, as well as command interfaces (e.g., effort). As in the `on_deactivate()` method, this step cleans up only the internal references within the controller and does not release the interfaces from the controller manager's resource allocator.

- **Logging.** Informational messages are published to the ROS 2 logging system to document the completion of the cleanup procedure.

```
/**
 * @brief Cleanup routine for the OpSpaceController.
 *
 * This method is called during the cleanup phase of the controller's lifecycle.
 * It performs the following actions:
 * - Logs the start of the cleanup process.
 * - Clears the desired trajectory and deactivates the trajectory flag.
 * - Releases resources associated with the Pinocchio model, if loaded.
 * - Clears all interface handles (position, velocity, and effort command ...
    handles).
 * - Logs the completion of the cleanup process.
 *
 * @param previous_state The previous lifecycle state before cleanup.
 * @return controller_interface::CallbackReturn Returns SUCCESS upon ...
    successful cleanup.
 */
```

The **`on_error()`** method defines the controller's behavior during transitions into the `ERROR` state, which may occur due to failed configuration, activation, or runtime faults. Its implementation ensures safety and system integrity by:

- Logging the transition into the error state.
- Deactivating any active trajectory tracking by setting the `trajectory_active_` flag to `false`.
- Attempting to send zero-effort commands to all joints, provided that the command interface handles are available and match the expected number of joints.
- Logging warnings or errors in case of handle mismatches or exceptions, without raising further faults to avoid cascading failures.

This function is designed to be minimal and fault-tolerant. No assumptions are made about the availability of other controller resources, and no operations are performed that could exacerbate the system's unstable state. The `on_error()` method prioritizes stopping the robot safely, making it a critical component in the controller's lifecycle.

```
/**
 * @brief Handles error state transitions for the OpSpaceController.
 *
 * This method is called when the controller enters an error state.
 * It logs the error, stops any active trajectory tracking, and attempts to send
 * safe stop commands (e.g., zero effort) to all joints to ensure the robot ...
    is safely halted.
 * If command handles are not ready or mismatched, a warning is logged instead.
 *
```

```
    * @param previous_state The previous lifecycle state before the error occurred.
    * @return controller_interface::CallbackReturn Returns SUCCESS after ...
        attempting safe stop procedures.
    */
```

The **`update()`** method represents the central execution loop of the OSC, invoked periodically by the `controller_manager` to compute joint effort commands based on the robot's current state and the desired trajectory in operational space. Upon invocation, the method begins with a series of safety and consistency checks. First, it verifies that the *Pinocchio model* has been successfully loaded; if not, an error is logged and a vector of zero effort commands is sent to all actuators, ensuring safe robot behavior. The same safety fallback is triggered if no trajectory is active or the trajectory vector is empty, although in this case the method may still return a success code to avoid unnecessary controller termination.

Next, the method retrieves the current joint positions and velocities from the respective state interface handles. The controller assumes that the order of the joints matches that of the Pinocchio model. If the number of available state handles does not match the expected number of joints, an error is logged and the method exits. Otherwise, joint states are copied into the internal `q_` and `dq_` vectors.

The current pose of the end-effector is then obtained via a TF lookup from the frame specified by the parameter `ee_link_name_` (typically `tool_link`) to the global frame (e.g., `map`). The retrieved transform is converted into an `Eigen::Isometry3d` and logged for monitoring purposes. If the TF lookup fails, the controller sends zero torques to all joints and exits the update cycle.

Once the robot state is available, the method checks if a desired trajectory is present and selects the current target waypoint by indexing into the `desired_trajectory_` vector using `current_trajectory_index_`. If the trajectory has been completed or the index exceeds the trajectory length, the controller enters a damping mode aimed at stabilizing the robot in its final configuration. In this case, no operational space control is applied. Instead, the method resorts to computing the feedforward joint torques required for gravity compensation by invoking the *Recursive Newton-Euler Algorithm* (RNEA) provided by the Pinocchio library. The RNEA computes the joint torques necessary to achieve a desired joint acceleration, given the current configuration and velocity of the robot. Specifically, it computes:

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\,\ddot{\mathbf{q}} + \mathbf{N}(\mathbf{q}, \dot{\mathbf{q}}) \tag{4.8}$$

where $\ddot{q} = 0$ is imposed to obtain pure gravity compensation.

Subsequently, a simple joint-space damping term proportional to the joint velocities is applied to achieve asymptotic stabilization of the manipulator. The resulting torque command is given by:

$$\boldsymbol{\tau}^{*} = \boldsymbol{\tau} - \mathbf{K}_{d}^{\text{joint}}\,\dot{\mathbf{q}} \tag{4.9}$$

where K_d^{joint} is a diagonal matrix representing joint damping gains. In the implementation, this is realized by subtracting a scaled version of the current joint velocities from the gravity compensation torques returned by RNEA. The torques are then sent directly to the actuators via the command effort interfaces. This fallback behavior ensures passive safety and prevents uncontrolled joint motion when no trajectory is active.

To prepare for the subsequent control computation, the method invokes the function `pinocchio::computeAllTerms()`, which serves as a unified call to compute all essential dynamic quantities required for inverse dynamics. Specifically, it updates the joint-space mass matrix $M(q)$, the nonlinear effects vector $n(q, \dot{q})$, which includes both Coriolis and gravitational contributions, and intermediate

quantities such as forward kinematics and frame placements. These are stored within the internal `data_` structure of Pinocchio, and will later be accessed to evaluate Jacobians, time derivatives of Jacobians, and the dynamic model required for computed torque control. Importantly, the symmetry of the mass matrix is explicitly enforced by copying the upper triangular part onto the lower one, which is a common step to ensure numerical stability in subsequent matrix operations. The end-effector pose computed by Pinocchio is also retrieved and used as the reference frame for all subsequent kinematic computations. The method then computes the spatial Jacobian and its time derivative in the `WORLD` frame using Pinocchio's `getFrameJacobian()` and `getFrameJacobianTimeVariation()` APIs.

Following the computation of geometric and dynamic terms, the controller computes the pose error in operational space. The error is first expressed as a *6D twist* using the logarithmic map of the rigid body transformation between the current and desired end-effector poses. This local error is then projected into the global frame for consistency with the desired motion quantities. Similarly, the end-effector velocity is obtained via Pinocchio and compared against the desired velocity to compute a velocity error in operational space.

The desired Cartesian acceleration is then computed using a *PD+ feedforward* control law, whose mathematical formulation has been extensively explained in Section 4.4.1. The proportional and derivative gains are represented as diagonal matrices, whose values are tuned via ROS 2 parameters. The Cartesian acceleration command is mapped to joint space via the *damped pseudo-inverse* of the Jacobian. This yields the desired joint accelerations, which are then used in the inverse dynamics computation to obtain the commanded joint torques, combining the mass matrix and nonlinear effects already computed.

The method then evaluates whether the translational component of the pose error is below a predefined threshold (1 mm). If so, the waypoint is considered reached and the `current_trajectory_index_` is incremented. To quantify the deviation between the desired and actual end-effector positions, the controller computes the Euclidean norm of the position error vector, defined as:

$$e_p = \|\mathbf{p}_{\text{des}} - \mathbf{p}_{\text{act}}\|_2 \tag{4.10}$$

where $\mathbf{p}_{\text{des}} \in \mathbb{R}^3$ and $\mathbf{p}_{\text{act}} \in \mathbb{R}^3$ denote the desired and actual end-effector pose expressed in the same coordinate frame. This scalar value e_p provides an intuitive metric for evaluating the positional tracking accuracy at each control cycle. Upon reaching the final waypoint, the `trajectory_active_` flag is deactivated to signal trajectory completion. The controller will then default to damping behavior in subsequent cycles.

Finally, the computed torques are sent to the hardware effort command interfaces. Before doing so, the method verifies that both the torque vector and the interface vector have the expected length; otherwise, it logs an error and sends zero effort commands for safety. The method then returns a success status to complete the cycle.

```
/**
 * @brief Main update loop for the Operational Space Controller.
 *
 * This method is called periodically by the controller manager to compute ...
    and send joint effort commands
 * based on the current robot state and the desired end-effector trajectory. ...
    It implements a computed torque
 * control law in operational space using the Pinocchio library for ...
    kinematics and dynamics.
```

```
 *
 * The update performs the following steps:
 *  - Checks if the Pinocchio model is loaded and if a trajectory is active.
 *  - Reads the current joint positions and velocities.
 *  - Obtains the current end-effector pose from the TF tree estimated by SLAM.
 *  - Selects the current target waypoint from the desired trajectory.
 *  - Computes all necessary kinematic and dynamic terms (forward kinematics, ...
    Jacobian, mass matrix, etc.).
 *  - Calculates pose and velocity errors in operational space.
 *  - Computes the desired Cartesian acceleration using a PD+ feedforward law.
 *  - Maps the desired Cartesian acceleration to joint accelerations via the ...
    Jacobian pseudoinverse.
 *  - Computes the required joint torques using the computed torque method.
 *  - Advances the trajectory waypoint if the goal is reached.
 *  - Sends the computed torques to the robot's actuators.
 *
 * Safety checks are performed throughout to ensure valid state and command ...
    dimensions.
 * Extensive logging is provided for debugging and monitoring.
 *
 * @param time   The current time (ROS 2 time).
 * @param period The duration since the last update call.
 * @return controller_interface::return_type
 *         - OK if the update was successful and commands were sent.
 *         - ERROR if a critical error occurred (e.g., model not loaded, TF ...
    lookup failed).
 */
```

4.5 System launch and execution

The final section of this chapter illustrates how the full robotic system, including perception, mapping, and operational space control, is launched and executed as a cohesive pipeline. This process is coordinated through the main launch file named `slam_rtab.launch.py`, which orchestrates the activation of all required nodes and configuration files within the ROS 2 ecosystem.

The primary entry point for launching the system is the `slam_rtab.launch.py` file, located in the `niryo_moveit2_config` package. The launch command is:

```
ros2 launch niryo_moveit2_config slam_rtab.launch.py
```

The launch file performs two key actions:

- It includes and executes the internal launch file `niryo_slam.launch.py`, which initializes the robot model, MoveIt 2, Gazebo, RViz, and the controller infrastructure.
- It delays the startup of the SLAM subsystem (RTAB-Map) using a `TimerAction`, ensuring that all robot components are fully initialized before the perception pipeline is activated.

The nested launch file `niryo_slam.launch.py` is responsible for initializing the following core components:

- **URDF-based robot description**: The full robot model, including all links and joints as well as the wrist-mounted depth camera, is loaded and published.
- **Gazebo simulation**: The robot is spawned in a Gazebo world environment, with the `gazebo_ros2_control` plugin activated for torque-level control.
- **Robot state publisher and TF static transform**: These nodes publish the robot's kinematic tree and static transforms between frames such as `world`, `map`, and `base_link`.
- **MoveIt 2 planning**: The `move_group` node is launched with CHOMP-based planning enabled, allowing generation of Cartesian motion trajectories.
- **RViz2**: A preconfigured RViz visualization is launched to display the robot model, TF frames, RTAB-Map point cloud, and planned trajectories.
- **Controllers**:
  - The `joint_state_broadcaster` is launched to continuously publish joint states.
  - The standard `niryo_arm_controller` (a JointTrajectoryController) is activated by default to maintain an idle pose.
  - The custom `niryo_op_space_controller` is also loaded, but remains initially `INACTIVE`.

After launching the full simulation and perception stack, the `niryo_op_space_controller` is loaded but remains in an `inactive` state. By default, the system uses the standard JTC `niryo_arm_controller` to maintain compatibility with MoveIt 2 and standard control interfaces during startup.

To verify the current state of all loaded controllers, the following command can be executed in a new terminal:

```
ros2 control list_controllers
```

This returns the status of all configured controllers. Initially, the output is:

Controller name	Controllor type	State
niryo_arm_controller	joint_trajectory_controller/JointTrajectoryController	active
joint_state_broadcaster	joint_state_broadcaster/JointStateBroadcaster	active
niryo_op_space_controller	op_space_controller/OpSpaceController	inactive

Table 4.1: Controller states at system startup.

To switch from the default controller to the custom operational space controller, it is must explicitly invoke the controller switch command provided by `ros2_control`. This is achieved by deactivating the current controller and activating the OSC in a strict, atomic operation:

```
ros2 control switch_controllers \
  --deactivate niryo_arm_controller \
  --activate niryo_op_space_controller \
  --strict
```

Once the switch is completed, the `niryo_op_space_controller` becomes active and starts computing joint-level torque commands for operational space tracking, as described in Section 4.4.1. The new controller states will appear as shown in the Table 4.2.

Controller name	Controllor type	State
niryo_arm_controller	joint_trajectory_controller/JointTrajectoryController	inactive
joint_state_broadcaster	joint_state_broadcaster/JointStateBroadcaster	active
niryo_op_space_controller	op_space_controller/OpSpaceController	active

Table 4.2: List of configured controllers after OSC activation.

This switching mechanism provides modularity and flexibility for experimentation: it allows the system to start in a stable and safe configuration using standard controllers and then transition to the custom control strategy in a controlled and deterministic way. This approach is particularly valuable for testing advanced control algorithms in simulation before deploying them to real hardware.

This section concludes the implementation chapter by demonstrating how the high-level architecture described in Chapter 3 is fully realized in practice through the ROS 2 launch system. The next chapter presents a series of experimental simulations designed to validate the controller's performance, the trajectory tracking accuracy, and the robustness of the system. Moreover, the next chapter includes a set of performance plots, including end-effector tracking errors in both position and orientation, structured similarly to those presented in the referenced paper [7]. This allows for a qualitative assessment of the controller's behavior and facilitates a preliminary comparison against state-of-the-art results.

Chapter 5

Simulations

This chapter presents the simulation-based evaluation of the operational space control strategy implemented in Chapter 4. The primary objective is to assess the performance of the custom controller in a fully integrated ROS 2 environment, with a focus on dynamic accuracy and system responsiveness.

The simulations are designed to achieve the following goals:

- Validate the behavior of the operational space controller developed as a ROS 2 plugin.
- Analyze the accuracy of Cartesian trajectory tracking with respect to a reference trajectory sampled from a CHOMP-based motion plan.
- Highlight current limitations and issues encountered during the simulation phase.

The experimental setup used for these simulations corresponds to the architecture and components described in Chapter 4, including the Gazebo physics simulator, MoveIt 2 motion planning, and RTAB-Map-based perception. The controller operates in torque mode, using inverse dynamics computations to generate joint commands in real time.

Extensive simulation trials were conducted to tune the controller gains and assess their impact on system performance. Initially, small values for the proportional and derivative gains resulted in sluggish behavior, with the end-effector taking a considerable amount of time to reach the desired waypoints. Through iterative testing and trial-and-error adjustment, the following gain configuration was identified as a satisfactory trade-off between responsiveness and stability:

$$\begin{cases} K_p^{\text{linear}} = 40.0, & K_d^{\text{linear}} = 12.0 \\ K_p^{\text{angular}} = 20.0, & K_d^{\text{angular}} = 9.0 \end{cases} \tag{5.1}$$

These values correspond to the diagonal terms of the proportional and derivative gain matrices used in the operational space control law described in Section 4.4.1.

For clarity, the controller gains are block-diagonal matrices in $\mathbb{R}^{6\times 6}$:

$$K_p = \begin{bmatrix} 40 & 0 & 0 & 0 & 0 & 0 \\ 0 & 40 & 0 & 0 & 0 & 0 \\ 0 & 0 & 40 & 0 & 0 & 0 \\ 0 & 0 & 0 & 20 & 0 & 0 \\ 0 & 0 & 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 & 0 & 20 \end{bmatrix} \qquad K_d = \begin{bmatrix} 12 & 0 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 \end{bmatrix} \tag{5.2}$$

With this configuration, the controller achieves smoother convergence to the target poses, while avoiding overshooting and maintaining acceptable damping behavior.

To improve the responsiveness of the controller, we increased the proportional gains (K_p) in order to enhance the system's ability to promptly reduce position errors. Consequently, the derivative gains (K_d) were also raised to counteract the more aggressive corrective actions and to prevent potential oscillations or overshoots.

In fact, it was also observed that excessively increasing the gain values beyond this configuration leads to undesirable oscillations, particularly in rotational motion. Therefore, the selected gains represent a balanced solution for accurate Cartesian trajectory tracking in the simulated environment, within the constraints imposed by the torque-level control architecture.

5.1 Execution and results of trajectory tracking

This section presents the experimental results obtained by executing a predefined Cartesian trajectory using the custom operational space controller. The main goal of the experiment was to evaluate the ability of the controller to track a series of desired end-effector poses in a simulated environment with high precision and stability.

Despite successfully tracking the geometric path of the desired trajectory, several practical limitations and issues were observed during simulation:

- **Control frequency tuning:** In order to achieve a responsive and stable behavior, it was necessary to increase the update frequency of the `ros2_control` manager to 1000 Hz, which is an order of magnitude higher than the default 100 Hz used for standard controllers such as the JointTrajectoryController. This adjustment was essential to reduce latency and maintain consistent real-time behavior for torque-level control.

- **Tuning of controller gains:** The selection of proportional and derivative gains (K_p and K_d) had a significant impact on performance. Extensive manual tuning was required to balance reactivity, noise sensitivity, and stability. High gains led to overshooting and oscillations, while low gains caused sluggish motion and large tracking errors.

- **Temporal mismatch:** Although the controller follows the spatial trajectory correctly, it fails to reproduce the desired timing profile. In particular, the planned trajectory is designed to complete in approximately 3 s, while the actual execution takes significantly longer (over 60 s). This discrepancy is due to the fact that the feedforward terms $\dot{\mathbf{X}}_{\text{des}}$ and $\ddot{\mathbf{X}}_{\text{des}}$ were set to zero in the controller implementation. Imposing zero desired velocities effectively instructs the robot to decelerate and hold its position, without exploiting the natural dynamics that would otherwise facilitate a smoother progression toward the next target. Moreover, torque-level control does not directly impose velocity constraints, further contributing to this limitation.

- **Absence of time synchronization:** The controller currently advances through the desired waypoints based on a spatial proximity criterion, without enforcing a time-synchronized trajectory playback. This design choice simplifies implementation but limits the controller's ability to accurately reproduce time-dependent motion profiles.

These issues highlight both the capabilities and the current limitations of the implemented controller. The following subsections present a quantitative analysis of the system performance, supported by a

series of plots showing the actual and desired trajectories, tracking errors, and joint torque profiles. These plots also facilitate a qualitative comparison with the results presented in the reference paper discussed in Section 2.4.

All plots and quantitative analyses presented in this section are generated using `MATLAB`, providing high-quality visualizations and enabling comparison with reference literature.

The results are organized into three main categories, each discussed in the subsequent subsections:

- **Comparison in 3D space:** A visual comparison between the desired Cartesian trajectory and the actual end-effector trajectory, shown in a three-dimensional space, to assess the overall tracking behavior.
- **Position tracking error of the end-effector:** A time-series plot of the norm of the translational tracking error, highlighting deviations from the reference path.
- **Control input torque:** The joint-level torque commands computed by the controller during execution, which reflect the dynamic effort required to achieve the desired Cartesian motion.

Each of the following subsections presents and discusses these aspects in detail.

5.1.1 Comparison in 3D space

In this subsection, we present a visual comparison between the desired trajectory and the actual trajectory followed by the end-effector during the simulation. The trajectories are plotted in a three-dimensional workspace using two complementary viewpoints to highlight spatial alignment and deviations. These plots were generated using a `MATLAB` code based on logged positional data and interpolated using cubic splines for clarity.

In all figures, the **desired trajectory** is represented by a **black line**, while the **trajectory followed by the robot** is shown as a **blue line**.

In an initial scenario where the controller parameters were not properly tuned, overly high proportional and derivative gains caused the system to respond too aggressively to pose errors. As a result, the robot exhibits unstable behavior with large oscillations and significant deviations from the desired path (Figure 5.1). In fact, the robot attempts to reduce the tracking error with respect to the current reference waypoint; however, due to excessively high control gains, it exhibits oscillatory behavior around the target position. Since the waypoint advancement logic relies solely on the norm of the pose error, two problematic scenarios may occur. In some cases, the robot remains stuck at the same waypoint for an extended period, unable to reduce the error below the specified threshold. In other cases, the error norm temporarily falls below the threshold, prompting the controller to advance to the next waypoint even though the robot has not yet stabilized around the previous one. This results in the propagation of oscillations along the trajectory, as the robot continues to track a moving target while still dynamically unstable. Notably, the final point traced by the end-effector lies well beyond the admissible error threshold, with a deviation of nearly 3 cm from the desired pose.

This distorted trajectory highlights the crucial role of gain tuning in ensuring accurate and stable execution of Cartesian trajectories.

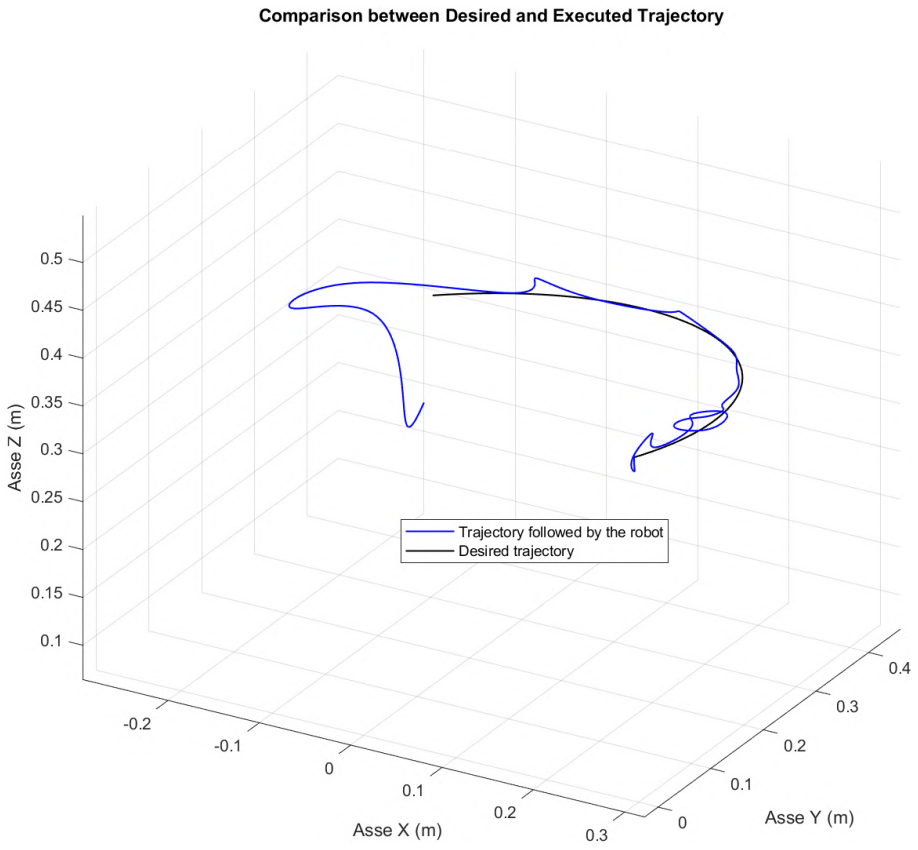


Figure 5.1: Executed trajectory with excessive control gains. Instability causes large oscillations and deviations from the desired path.

In contrast to the previous case, a more refined tuning process yields improved trajectory tracking performance.

An isometric viewpoint (Figure 5.2)(with an elevation of 30°) enables a comprehensive visualization of the trajectory across all three spatial axes (x, y, z). From this perspective, the initial portion of the executed trajectory closely follows the desired path, suggesting that the controller is able to accurately track the first few waypoints. As the trajectory evolves, however, a growing discrepancy becomes apparent between the planned and actual motion, particularly in the spatial alignment and orientation of the end-effector poses. This drift becomes increasingly noticeable toward the final segment of the trajectory.

A complementary view oriented along the lateral direction (Figure 5.3) emphasizes deviations in the yz-plane. In this configuration, small oscillations or overshoots can be observed, especially near the final portion of the path. Notably, the end-effector does not reach the final desired waypoint: the motion concludes prematurely, and the final position remains slightly above and short of the intended target.

This behavior confirms the observations discussed in Section 5.1 regarding the lack of time synchronization and absence of velocity feedforward in the control input. Despite the absence of catastrophic instability, these results highlight the limits of the current implementation in precisely following time-parameterized spatial trajectories.

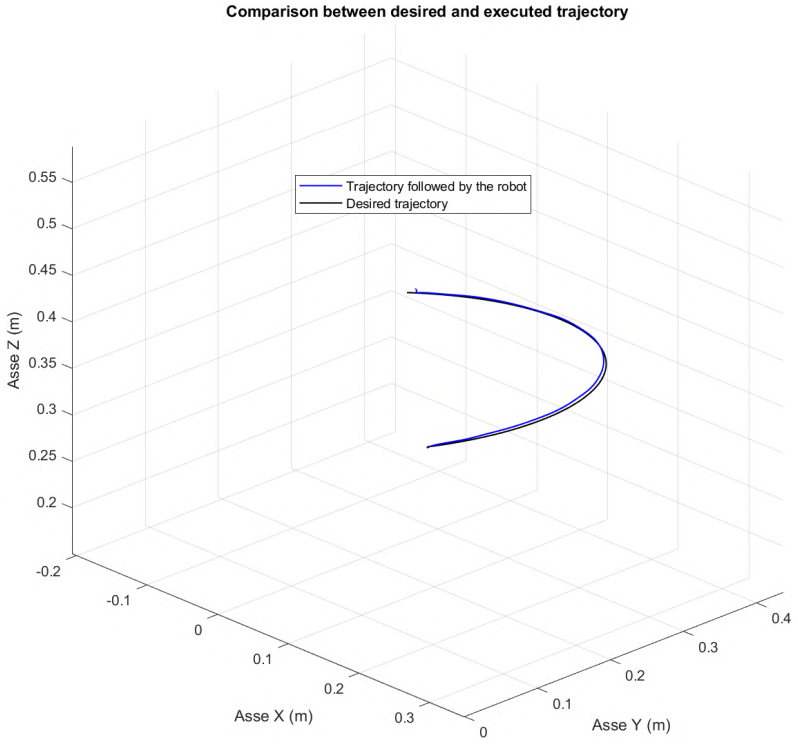


Figure 5.2: (a) 3D comparison between desired and executed trajectories of the end-effector. Isometric view (x-y-z).

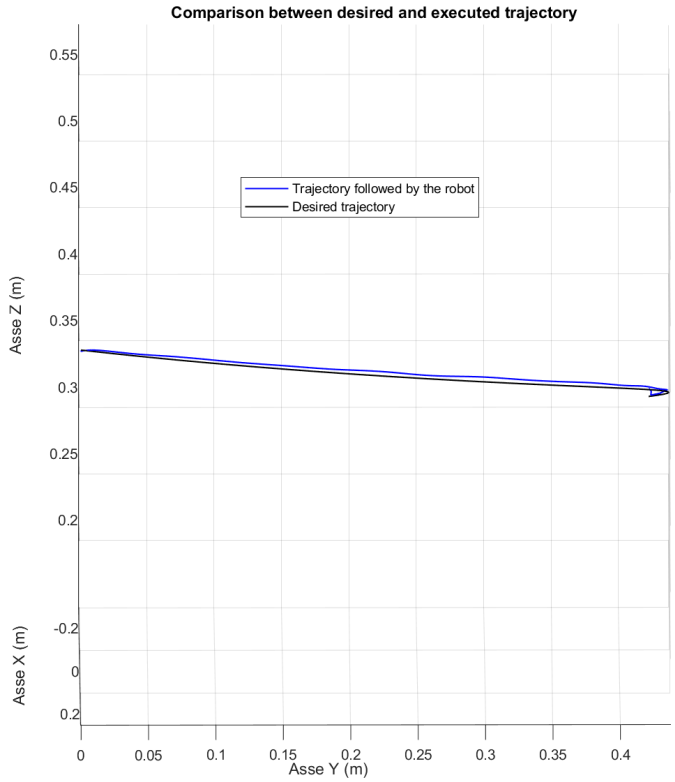


Figure 5.3: (b) 3D comparison between desired and executed trajectories of the end-effector. Lateral view (y-z plane).

5.1.2 Position tracking error of the end-effector

To quantitatively assess the controller's tracking performance, the positional error norm described in Section 4.4.1 was evaluated at each desired waypoint. Specifically, the Euclidean distance between the desired and actual position of the end-effector was computed when the robot reached the vicinity of each sampled pose.

The temporal evolution of the positional tracking error throughout the trajectory execution (Figure 5.4) reveals how the controller performs over time. Each data point represents the error measured at a specific waypoint and is associated with the corresponding simulation timestamp. The overall profile provides insight into the stability and accuracy of the controller as it progresses along the reference path.

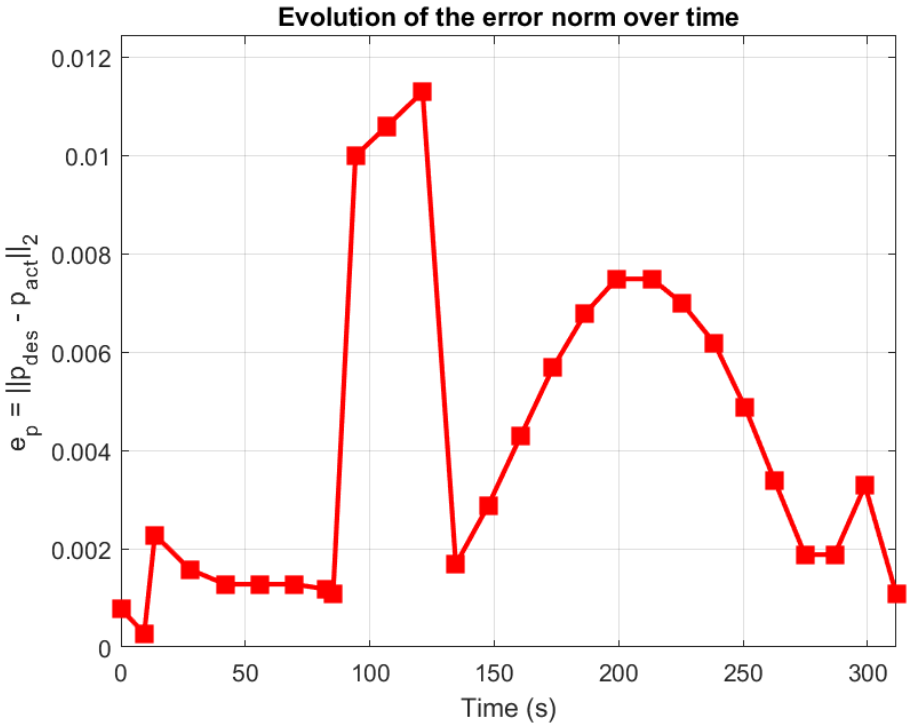


Figure 5.4: Norm of the position tracking error of the end-effector evaluated at each waypoint during execution.

The results indicate that the tracking error remains generally low across most of the trajectory, especially during the initial segments, where the error is consistently below 2 mm. In the middle portion of the trajectory, a moderate increase in the error norm is observed, with peak values slightly above 10 mm. This degradation in accuracy may be attributed to the accumulation of dynamic discrepancies, the lack of feedforward velocity terms, and the progressive drift from the reference path due to underactuated timing.

Moreover, the use of a visual SLAM-based pose estimation system introduces an additional source of uncertainty. Although RTAB-Map provides a reliable estimation of the end-effector pose in the global `map` frame, it is still subject to drift and latency, which can impact the accuracy of the measured tracking error. These effects become more evident in the central and final portions of the trajectory, where the robot deviates more substantially from the planned path.

Despite these deviations, the controller is able to maintain the tracking error within acceptable bounds for the majority of the motion. The trajectory ends with a gradual reduction in the error, suggesting a partial re-alignment with the desired path during the final waypoints.

These results confirm the capability of the implemented control scheme to follow Cartesian paths with reasonable positional precision, while also highlighting the need for further refinement in the temporal coordination and velocity tracking strategies.

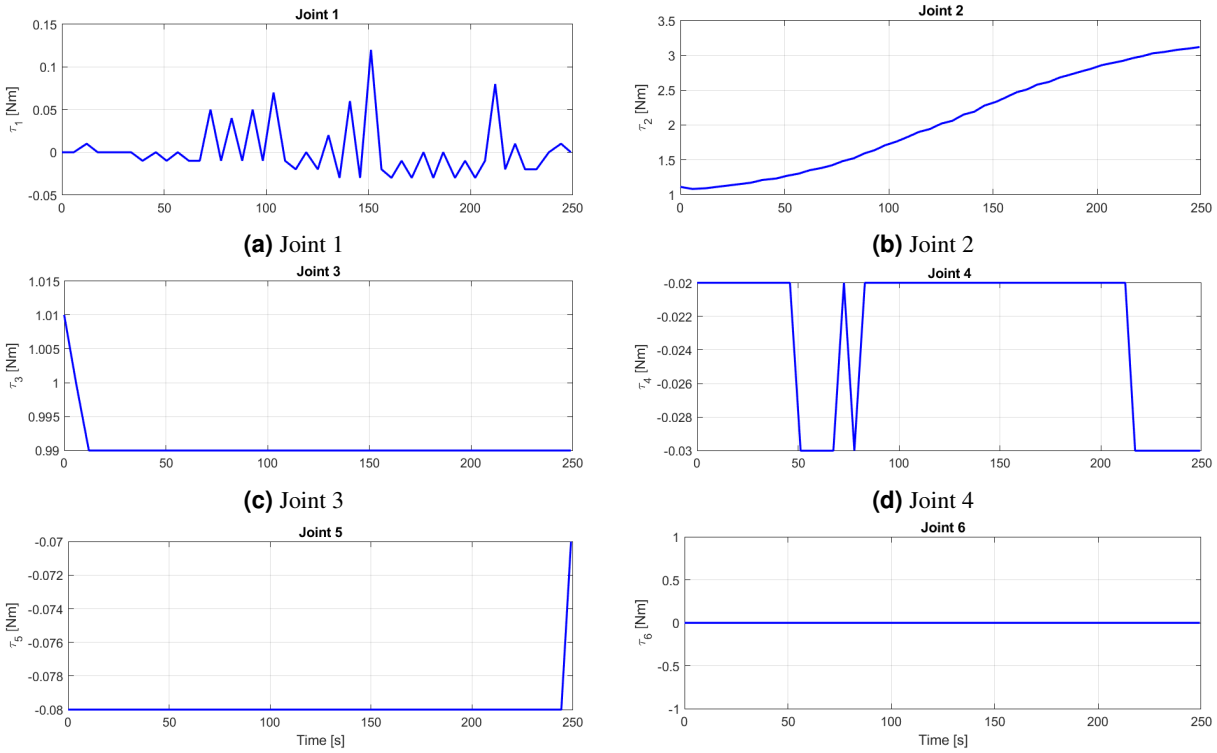
5.1.3 Control input torque

This subsection presents the evolution of the control input torques generated by the operational space controller (Equation (4.8)) during the trajectory tracking experiment. These torques represent the output of the inverse dynamics computation used to regulate the motion of each joint in accordance with the tracking objectives.

The profiles associated with each of the six joints exhibit distinct behaviors (Figure 5.5), so, several trends can be observed from the plot:

- **Joint 1** (Figure 5.5a) exhibits a moderately dynamic behavior, with torque values oscillating between approximately $-0.05\,\mathrm{N\,m}$ and $1.15\,\mathrm{N\,m}$. This indicates that the joint is actively compensating for base-level orientation adjustments throughout the trajectory.
- **Joint 2** (Figure 5.5b) displays a smooth increase in torque following a profile similar to an arctangent function, suggesting a continuous and progressive effort to reposition the arm along a curved segment of the path.
- **Joint 3** (Figure 5.5c) starts with a torque of about $1.01\,\mathrm{N\,m}$ and gradually stabilizes to around $0.99\,\mathrm{N\,m}$. This quasi-constant value suggests that this joint contributes a relatively steady load to maintain the elevation of the manipulator.
- **Joint 4** (Figure 5.5d) demonstrates low-frequency oscillations between $-0.03\,\mathrm{N\,m}$ and $-0.02\,\mathrm{N\,m}$, indicating fine-grained torque modulation, likely related to minor attitude corrections of the end-effector.
- **Joint 5** (Figure 5.5e) maintains a constant torque around $0.8\,\mathrm{N\,m}$ for most of the trajectory, with a slight decrease toward $0.7\,\mathrm{N\,m}$ in the final phase. This behavior suggests a stabilizing contribution to wrist articulation.
- **Joint 6** (Figure 5.5f) consistently applies zero torque throughout the entire execution. This may be attributed to the fact that the sixth joint (typically the wrist roll) is not significantly engaged by the planned trajectory or is constrained by task geometry.

Overall, the observed torque profiles confirm that the controller generates smooth and physically plausible control efforts. While none of the joints appear to reach saturation, the range of actuation, especially in joints 1 and 2, highlights the nontrivial dynamics involved in achieving Cartesian tracking via torque-level control.



(a) Joint 1 **(b)** Joint 2

(c) Joint 3 **(d)** Joint 4

(e) Joint 5 **(f)** Joint 6

Figure 5.5: Joints torque profiles during trajectory tracking.

It is worth noting that some of the joint torque profiles (Figure 5.5) exhibit behaviors that may resemble saturation—namely, prolonged periods where the torque value remains almost constant or plateaus. However, a closer examination reveals that none of the torque values approach the physical effort limits defined in the robot's URDF file. For instance, Joint 1 reaches a maximum torque of approximately 1.15 N m, which is significantly below its defined effort limit of 10.0 N m. So, no joint reaches its defined limit during the simulation.

The apparent saturation-like behavior is a byproduct of the control strategy and trajectory design. In the current implementation, the desired velocity $\dot{\mathbf{x}}_{\text{des}}$ and acceleration $\ddot{\mathbf{x}}_{\text{des}}$ are both manually set to zero, instead of being computed from the reference trajectory. As a consequence, the control law operates purely in feedback mode, compensating for gravitational and positional errors without predictive action.

This results in relatively constant torque outputs, particularly in joints that primarily counterbalance the weight of the manipulator. While this does not compromise safety or violate physical constraints, it may limit the expressiveness and time fidelity of the controller.

5.2 Comparison with reference literature results

This section provides a qualitative comparison between the results obtained through the simulation campaign and those reported in the reference paper discussed in Section 2.4. While the general control strategy and architectural structure are inspired by the same operational space control paradigm, several key differences emerge in terms of performance, implementation choices, and environmental constraints.

The reference paper evaluates the controller performance using a simple square trajectory executed in free space. In contrast, this work adopts a trajectory generated by the CHOMP planner, characterized

by continuous curvature and more complex motion in a simulated environment. Despite the increased difficulty, the robot is able to follow the desired trajectory with satisfactory accuracy: the end-effector position error remains below 1 mm for most of the execution. The resulting behavior in terms of joint effort, smoothness, and dynamic feasibility is coherent with the expectations of a torque-level inverse dynamics controller.

A notable difference with respect to the reference implementation lies in the treatment of feedforward terms. In the paper, desired end-effector velocities and accelerations are computed and used to improve dynamic tracking. In this work, these terms were explicitly set to zero, resulting in a controller that requires the robot to stop upon reaching each waypoint. This simplification affects both the transient response and the timing of the motion. While the reference trajectory was designed to complete in approximately 3 s, the execution in this work spans over 60 s, highlighting the controller's limited responsiveness without motion feedforward.

A direct visual comparison between the curved trajectory adopted in this work and the idealized square trajectory used in the reference literature highlights significant differences in execution accuracy. The trajectories generated via CHOMP are represented by the blue (executed) and black (desired) curves, while the square path from the reference study is depicted through the red (desired) and green (executed) segments, which exhibit a reported tracking error of approximately 2% (Figure 5.6).

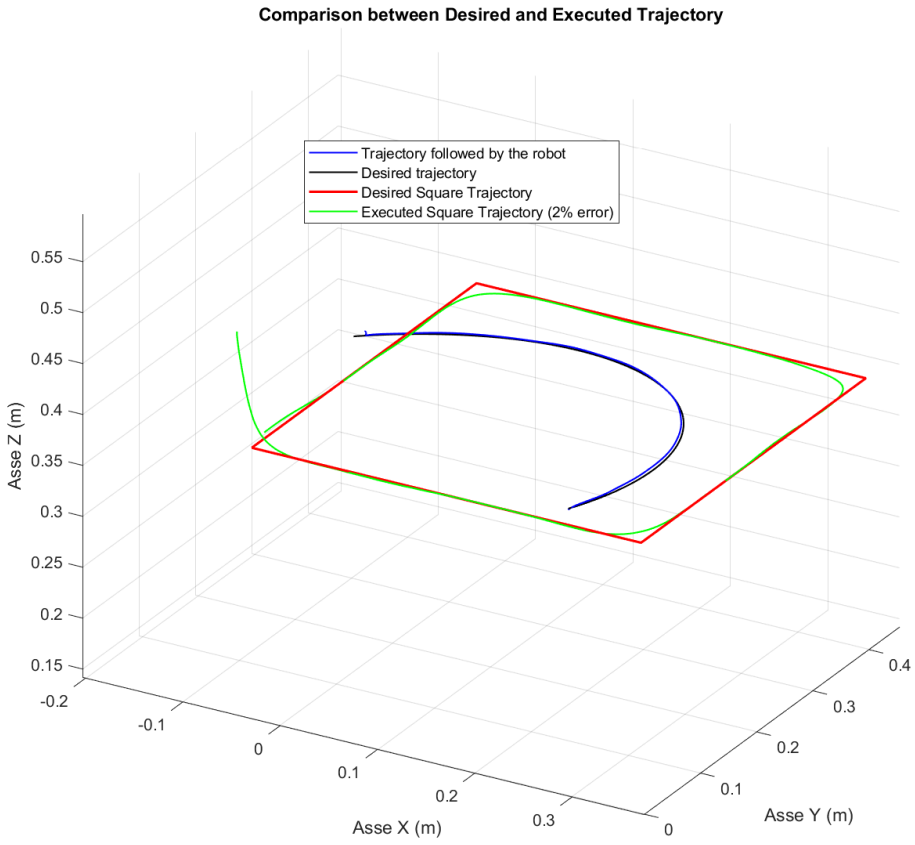


Figure 5.6: Comparison between the trajectory tracking in this work (blue–black) and the square reference trajectory used in the literature (red–green)

This comparison clearly highlights a key advantage of the proposed implementation. While the reference work reports trajectory deviations of approximately 2–3 cm, the system developed in this thesis

maintains sub-millimeter accuracy for the majority of the execution. Despite the increased geometric complexity of the CHOMP-generated path, the torque-based controller demonstrates a superior capability to track the desired motion, both in terms of precision and stability. This underscores the robustness of the control design and its effective integration within the ROS 2-based architecture.

To further explore the limitations of the implemented controller, an additional simulation was conducted using a reference trajectory whose initial waypoint does not match the robot's actual initial pose. This test case was explored in the reference paper. The result showed that the robot attempts to reach the initial waypoint but is unable to reduce the tracking error, which remains constant at approximately 5.6 cm. Consequently, the controller remains stuck at the first waypoint, as the internal logic for advancing along the trajectory is based on error convergence. This behavior reveals a lack of robustness when handling significant initial deviations, especially in the absence of a transition strategy or feedforward velocity terms to guide convergence. While the reference implementation assumes perfect alignment between the robot and the trajectory at initialization, this work deliberately explores more challenging and realistic conditions.

In summary, while this work adopts a more complex and realistic trajectory and includes experiments outside the idealized assumptions of the reference paper, the controller demonstrates a comparable ability to track Cartesian motions with torque-based commands. Nevertheless, the absence of motion feedforward and an initialization handling strategy leads to significant performance gaps in terms of speed and robustness. These insights define a concrete basis for future controller improvements, which will be addressed in Chapter 6.

Chapter 6

Conclusions

This final chapter summarizes the outcomes of the research work and outlines possible directions for future development. The first section presents the concluding remarks, emphasizing the main contributions of the thesis and their relevance within the broader context of operational space control in robotics. The second section discusses open challenges and improvements that could enhance the robustness, generality, and performance of the proposed control architecture.

6.1 Contributions

This thesis explored the problem of operational space control for a 6-DoF manipulator, specifically, the Niryo Ned2, within a ROS 2-based simulation environment. The objective was to design, implement, and evaluate a custom torque-level controller capable of accurately tracking end-effector trajectories generated through sampling from a motion planner (CHOMP), while incorporating localization feedback derived from a visual SLAM system.

The developed controller was implemented as a modular ROS 2 plugin, fully integrated into the `ros2_control` framework. It leverages inverse dynamics computations based on the Pinocchio library to calculate joint-level torque commands from desired accelerations in the Cartesian space. The control strategy was formulated within the mathematical framework of rigid-body transformations ($\mathrm{SE}(3)$), enabling unified treatment of both translational and rotational errors in the operational space.

Comprehensive simulation experiments were conducted to evaluate the effectiveness of the proposed solution. The system was tested on smooth, dynamically feasible trajectories generated in a realistic environment, and tracking performance was quantified in terms of Cartesian pose error, torque profiles, and stability. The results demonstrate that the controller can achieve sub-millimeter tracking accuracy, with torque values that remain well within the actuator limits defined in the robot model. These findings validate the correctness and effectiveness of the inverse dynamics-based control strategy for precision motion execution.

Overall, the work contributes a reusable and extensible control module, validated in simulation, and provides a solid basis for future research into more robust and adaptive torque-level controllers for real-world robotic applications. The path forward involves improving controller generality, integrating sensor fusion, and extending the framework to enable compliant and contact-rich tasks. Nevertheless, certain limitations were observed. In particular, the lack of non-zero feedforward velocity and acceleration

terms reduced the responsiveness of the controller, especially during trajectory transitions. Additionally, the assumption of initial pose alignment with the first trajectory waypoint was shown to affect robustness: when this condition is not met, the system may exhibit oscillatory or stalled behavior due to the error-based waypoint switching logic.

Development challenges: The development of this project was also shaped by several practical constraints that influenced the design decisions and testing process. In particular, the use of WSL2 introduced significant challenges in running complex simulation environments such as Gazebo in conjunction with ROS 2. Despite its convenience, WSL2 presented limitations in terms of graphical support, networking behavior, and system-level hardware access, which occasionally hindered the integration and debugging of key components.

Moreover, the computational cost of running real-time physics simulation, motion planning, visual SLAM, and torque-level control concurrently imposed a heavy load on the available hardware resources. The experiments were conducted on a consumer-grade laptop, which constrained the achievable simulation rate and occasionally led to frame drops or synchronization issues between nodes. These limitations required careful tuning of system parameters, such as controller update rates, SLAM settings, and trajectory sampling resolution, to maintain acceptable performance without overloading the system.

While these constraints did not prevent the successful completion of the work, they highlight the importance of adequate computational resources and robust system design in real-time robotic control applications. Future developments on more powerful hardware or directly on the physical robot are expected to alleviate many of these limitations and further validate the proposed approach under more realistic conditions.

6.2 Future developments

The work presented in this thesis represents a foundational step towards developing torque-based control architectures for manipulators operating in unstructured environments, leveraging full-body inverse dynamics and vision-based localization. However, several opportunities exist to further extend and improve the system in both theoretical and implementation aspects.

- **Incorporation of feedforward motion terms.** One of the most impactful directions is the integration of non-zero desired end-effector velocities and accelerations ($\dot{\mathbf{X}}_{\text{des}}, \ddot{\mathbf{X}}_{\text{des}}$) into the control law. This would enable improved tracking accuracy and responsiveness, particularly in dynamic motions and during transients between waypoints.

- **Trajectory reinitialization and adaptive synchronization.** The current controller assumes perfect alignment with the reference trajectory at the start of the execution. Future work could investigate strategies for automatic alignment with the trajectory through a pre-motion phase, as well as methods to adaptively synchronize with the desired trajectory over time using dynamic time warping or error-based progression. In fact, imposing a desired trajectory whose first waypoint differs from the robot's initial pose effectively instructs the system to reach that point without specifying how to do so. This lack of initial guidance may result in erratic or unstable motion, especially in torque-level control, where the path taken to reach the target is not explicitly regulated.

- **Trajectory interpolation between waypoints.** Future implementations could integrate interpolation strategies between discrete waypoints before sending them to the controller. By employing algorithms such as linear interpolation (LERP), spherical interpolation (SLERP), or spline-based

smoothing, it is possible to generate a continuous and time-parametrized reference trajectory. This would improve the smoothness of the commanded motion, allow consistent time-based sampling, and pave the way for realistic feedforward term computation.

- **Robustness to pose estimation drift.** Since the end-effector pose is obtained from SLAM-based estimation in the `map` frame, future work could explore the fusion of visual odometry with proprioceptive sensing (e.g., joint encoders) using Extended Kalman Filters or factor graph optimization to mitigate the effect of drift in long-duration tasks.
- **Extension to interaction tasks.** The current implementation focuses on free-space trajectory tracking. Further research could extend the controller to compliant tasks, including physical human-robot interaction or contact-rich manipulation, by integrating force/torque feedback and admittance control strategies.

References

[1] Steven Macenski et al. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (May 2022). ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074. URL: http://dx.doi.org/10.1126/scirobotics.abm6074.

[2] Niryo. *Niryo Ned2 - 6-Axis Educational Robot Arm*. 2024. URL: https://niryo.com/product/niryo-ned2-educational-robot/.

[3] Sachin Chitta, Ioan Sucan, and Steve Cousins. “Moveit!” In: *IEEE Robotics & Automation Magazine*. Vol. 19. 1. IEEE, 2012, pp. 1–2.

[4] Nathan Ratliff et al. “CHOMP: Covariant Hamiltonian optimization for motion planning”. In: *ICRA 2009 — IEEE International Conference on Robotics and Automation*. IEEE. 2009, pp. 489–494.

[5] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. “The Open Motion Planning Library”. In: *IEEE Robotics & Automation Magazine* 19.4 (2012), pp. 72–82.

[6] Tommaso Savino. *ROS2 and MoveIt2 Integration and Motion Planning for the Niryo Ned2 with Pick-and-Place Implementation*. https://github.com/ItsTomSav/Tirocinio_lab_ROS2. Curricular internship report, Politecnico di Bari. 2025.

[7] Seyed Hamed Hashemi and Jouni Mattila. *Task Space Control of Robot Manipulators based on Visual SLAM*. 2023. arXiv: 2302.04163 [eess.SY]. URL: https://arxiv.org/abs/2302.04163.

[8] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. 1st ed. London: Springer, 2009. ISBN: 978-1846286414.

[9] MORIKAZU TAKEGAKI and SUGURU ARIMOTO and. “An adaptive trajectory control of manipulators”. In: *International Journal of Control* 34.2 (1981), pp. 219–230. DOI: 10.1080/00207178108922527. eprint: https://doi.org/10.1080/00207178108922527. URL: https://doi.org/10.1080/00207178108922527.

[10] Yeping Wang and Michael Gleicher. *Anytime Planning for End-Effector Trajectory Tracking*. 2025. arXiv: 2502.03676 [cs.RO]. URL: https://arxiv.org/abs/2502.03676.

[11] Matteo Scucchia. “Simultaneous Localization and Mapping for Robots: An Experimental Analysis Using ROS”. Supervisor: Prof. Davide Maltoni, Session IIII, Academic Year 2020–2021. Master’s Thesis. Bologna, Italy: Alma Mater Studiorum – University of Bologna, School of Science, 2021.

[12] Petri Mäkinen et al. “Application of Simultaneous Localization and Mapping for Large-scale Manipulators in Unknown Environments”. In: *2019 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. 2019, pp. 559–564. DOI: 10.1109/CIS-RAM47153.2019.9095770.

[13] Armin Hornung et al. “OctoMap: An efficient probabilistic 3D mapping framework based on octrees”. In: *Autonomous Robots* 34.3 (2013), pp. 189–206.

[14] Articulated Robotics. *What are depth cameras?* Accessed: 2025-06-04. 2023. URL: https://articulatedrobotics.xyz/tutorials/mobile-robot/hardware/donotad%20d10-depth-camera/#what-are-depth-cameras.

[15] Articulated Robotics. *How are depth cameras used in ROS?* Accessed: 2025-06-04. 2023. URL: https://articulatedrobotics.xyz/tutorials/mobile-robot/hardware/donotadd10-depth-camera/#how-are-they-used-in-ros.

[16] Javier Civera and Seong Hun Lee. “RGB-D Odometry and SLAM”. In: *arXiv preprint* (2020). URL: https://arxiv.org/abs/2001.06875.

[17] Zheng Fang and Yu Zhang. “Experimental Evaluation of RGB-D Visual Odometry Methods”. In: *IntechOpen* (2015). URL: https://doi.org/10.5772/59991.

[18] Jared Herron et al. “RGB-D Robotic Pose Estimation For a Servicing Robotic Arm”. In: *arXiv preprint* (2022). URL: https://arxiv.org/abs/2207.11537.

[19] Joseph Stewart and Matt Church. “2024”. MA thesis. 10, 2019.

[20] Hyeong Ryeol Kam et al. “RViz: A toolkit for real domain data visualization”. In: *Telecommunication Systems* 60.2 (2015), pp. 337–345. ISSN: 1572-9451. DOI: 10.1007/s11235-015-0034-5. URL: https://doi.org/10.1007/s11235-015-0034-5.

[21] Nathan Koenig and Andrew Howard. “Design and use paradigms for Gazebo, an open-source multi-robot simulator”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2004, pp. 2149–2154.

[22] Jacob Michael Perron. “NASA’s VIPER Simulation: From Moon Terrain to ROS 2 Integration”. In: *ROSCon 2021*. Presentation available via NASA Technical Reports Server. New Orleans, LA, USA, 2021. URL: https://ntrs.nasa.gov/citations/20210018027.

[23] Mathieu Labbé and François Michaud. “RTAB-Map as an open-source Lidar and visual SLAM library for large-scale and long-term online operation”. In: *Journal of Field Robotics* 36.2 (2019), pp. 416–446. DOI: 10.1002/rob.21831.

[24] Stefan Kohlbrecher et al. “A flexible and scalable SLAM system with full 3D motion estimation”. In: *2011 IEEE international symposium on safety, security, and rescue robotics*. IEEE. 2011, pp. 155–160.

[25] Tixiao Shan and Brendan Englot. “LIO-SAM: Tightly-coupled lidar inertial odometry via smoothing and mapping”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020).

[26] Raul Mur-Artal and Juan D. Tardos. “ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras”. In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262.

[27] Carlos Campos et al. “ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial and Multi-Map SLAM”. In: *IEEE Transactions on Robotics* 37.6 (2021), pp. 1874–1890.

[28] Thomas Whelan et al. “ElasticFusion: Real-time dense SLAM and light source estimation”. In: *The International Journal of Robotics Research*. Vol. 35. 14. 2015, pp. 1697–1716.

[29] Tong Qin et al. “A general optimization-based framework for local odometry estimation with multiple sensors”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2019, pp. 5835–5841.

[30] Oussama Khatib. “A unified approach for motion and force control of robot manipulators: The operational space formulation”. In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53.

[31] Neville Hogan. “Impedance control: An approach to manipulation: Part I—Theory”. In: *Journal of dynamic systems, measurement, and control* 107.1 (1985), pp. 1–7.

[32] Nannan Du et al. “Simulation Analysis of Discrete Admittance Control of Manipulator”. In: *2022 IEEE 17th Conference on Industrial Electronics and Applications (ICIEA)*. 2022, pp. 926–930. DOI: 10.1109/ICIEA54703.2022.10006131.

[33] Jeffrey Delmerico and Davide Scaramuzza. “A Dataset for Benchmarking Visual-Inertial Odometry”. In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 1988–1995.

[34] Eric Marchand, Hideaki Uchiyama, and Florian Spindler. “Markerless tracking for augmented reality: A learning-based approach”. In: *Computer Graphics and Applications* 25.6 (2005), pp. 18–26.

[35] Yu Xiang et al. “PoseCNN: A convolutional neural network for 6D object pose estimation in cluttered scenes”. In: *Robotics: Science and Systems (RSS)*. 2018.

[36] Yisheng Li et al. “DeepIM: Deep Iterative Matching for 6D Pose Estimation”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.

[37] Justin Carpentier et al. “The Pinocchio C++ library – A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives”. In: *SII 2019 - International Symposium on System Integrations*. Paris, France, Jan. 2019. URL: https://hal.laas.fr/hal-01866228.

[38] Tully Foote and Mike Purvis. *REP 103: Standard Units of Measure and Coordinate Conventions*. https://www.ros.org/reps/rep-0103.html. Accessed: 2025-06-18. 2010.

[39] ROS 2 Control Maintainers. *Writing a New Controller*. ROS 2 Control Project. 2023. URL: https://control.ros.org/humble/doc/ros2_controllers/doc/writing_new_controller.html.

Project codes

Listing 1: Example of a wooden pallet model present in the Gazebo world, `world_file_pallet.world`.

```
<model name='euro_pallet'>
  <link name='link'>
    <pose>0 0 0 0 -0 0</pose>
    <inertial>
      <mass>20</mass>
      <inertia>
        <ixx>2.43</ixx>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyy>1.1</iyy>
        <iyz>0</iyz>
        <izz>3.47</izz>
      </inertia>
      <pose>0 0 0 0 -0 0</pose>
    </inertial>
    <collision name='collision'>
      <pose>0 0 0.05 0 -0 0</pose>
      <geometry>
        <mesh>
          <uri>model://euro_pallet/meshes/pallet.dae</uri>
          <scale>0.1 0.1 0.1</scale>
        </mesh>
      </geometry>
      <max_contacts>10</max_contacts>
    </collision>
    <visual name='visual'>
      <pose>0 0 0.05 0 -0 0</pose>
      <geometry>
        <mesh>
          <uri>model://euro_pallet/meshes/pallet.dae</uri>
          <scale>0.1 0.1 0.1</scale>
        </mesh>
      </geometry>
    </visual>
    <self_collide>0</self_collide>
    <enable_wind>0</enable_wind>
    <kinematic>0</kinematic>
  </link>
  <pose>-1.81925 -0.555672 0 0 -0 0</pose>
</model>
```

Listing 2: Activating Gazebo control in the robot URDF File

```
<gazebo>
    <plugin name="gazebo_ros2_control" filename="libgazebo_ros2_control.so">
        <robot_param>robot_description</robot_param>
        <parameters>$(find ...
            niryo_moveit2_config)/config/ros2_controllers.yaml</parameters>
    </plugin>
</gazebo>

<!-- Import transmission definitions -->
<xacro:include filename="$(find ...
    niryo_robot_description)/urdf/ned2/niryo_ned2.transmission.xacro"/>
```

Listing 3: Integrating the camera into the Robot's URDF file

```
<joint name="depth_camera_joint" type="fixed">
    <axis xyz="0 0 1"/>
    <origin xyz="0.0385 0 0.05" rpy="0 0 ${-PI/2+deg_to_rad*10}"/>
    <parent link="wrist_link"/>
    <child link="depth_camera_link"/>
</joint>

<link name="depth_camera_link">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.01 0.037 0.037"/>
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.01 0.037 0.037"/>
        </geometry>
    </collision>
    <inertial>
        <mass value="0.05"/>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <inertia ixx="1e-4" ixy="0.0" ixz="0.0" iyy="1e-4" iyz="0.0" ...
            izz="1e-4"/>
    </inertial>
</link>

<joint name="depth_camera_optical_joint" type="fixed">
    <parent link="depth_camera_link"/>
    <child link="depth_camera_optical_link"/>
    <origin xyz="0 0 0" rpy="${-PI/2} 0 ${-PI/2}"/>
</joint>

<link name="depth_camera_optical_link"/>

<gazebo reference="depth_camera_link">
    <sensor type="depth" name="gazebo_depth_camera">
        <visualize>true</visualize>
```

```
            <update_rate>10.0</update_rate>
            <camera>
                <horizontal_fov>1.0</horizontal_fov>
                <image>
                    <width>640</width>
                    <height>480</height>
                    <format>R8G8B8</format>
                </image>
                <clip>
                    <near>0.05</near>
                    <far>8.0</far>
                </clip>
            </camera>
            <plugin name="depth_camera_controller" ...
                filename="libgazebo_ros_camera.so">
                <frame_name>depth_camera_optical_link</frame_name>
                <min_depth>0.1</min_depth>
                <max_depth>100.0</max_depth>
            </plugin>
        </sensor>
    </gazebo>
```

Listing 4: Launch file for all nodes (Gazebo, RViz, Moveit) and RTAB-Map configuration and launch

```
import os

from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, TimerAction
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():

    niryo_moveit_config_pkg = get_package_share_directory('niryo_moveit2_config')

    niryo_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(niryo_moveit_config_pkg, 'launch', ...
                'niryo_slam.launch.py')
        )
    )

    # Static transform world - map
    static_world_map = Node(
        package='tf2_ros',
        executable='static_transform_publisher',
        name='static_world_map',
        output="log",
        arguments=['0', '0', '0', '0', '0', '0', 'world', 'map'],
        parameters=[
            {'use_sim_time': True}
        ],
    )
```

```
30
31      # RTAB-Map node
32      rtabmap_node = Node(
33          package='rtabmap_slam',
34          executable='rtabmap',
35          name='rtabmap',
36          output='screen',
37          parameters=[{
38              'use_sim_time': True,
39              'frame_id': 'base_link',
40              'odom_frame_id': 'base_link',
41              'map_frame_id': 'map',
42              'subscribe_depth': True,
43              'subscribe_rgbd': False,
44              'subscribe_scan': False,
45              'approx_sync': True,
46              'sync_queue_size': 30,
47              'RGBD/NeighborLinkRefining': 'false',
48              'Reg/Force3DoF': 'false',
49              'RGBD/LinearUpdate': '0.0',
50              'RGBD/AngularUpdate': '0.0',
51              'RGBD/MaxDepth': '3.5',
52              'Mem/IncrementalMemory': 'true',
53              'Mem/DepthCompressionFormat': '.png',
54              'Vis/MinInliers': '15',
55
56              # --- Mapping/Memory Parameters ---
57              'RGBD/OptimizeMaxError': '3.0',
58              'RGBD/ProximityBySpace': 'false',
59              'Mem/ImagePreDecimation': '4',
60              'Rtabmap/DetectionRate': '1',
61              'Grid/FromDepth': 'true',
62              'Grid/RayTracing': 'false',
63              'Grid/3D': 'true',
64              'Grid/CellSize': '0.1',
65
66              # --- Recording/Loop Closure Parameters ---
67              'Reg/Strategy': '0',
68              'Optimizer/Slam2D': 'false',
69
70              # --- TF Parameters ---
71              'wait_for_transform': 0.3,
72              'tf_delay': 0.05,
73              'tf_tolerance': 0.1,
74
75              # --- Keypoint Parameters ---
76              'Kp/MaxFeatures': '-1',
77
78              # --- Quality Of Service - QoS ---
79              'qos_image': 2,
80              'qos_camera_info': 2,
81          }],
82          arguments=['-d'],
83          remappings=[
84              ('rgb/image', '/gazebo_depth_camera/image_raw'),
```

```
            ('depth/image', '/gazebo_depth_camera/depth/image_raw'),
            ('rgb/camera_info', '/gazebo_depth_camera/camera_info'),
        ]
    )

    # RTAB-Map Viz Node
    rtabmap_viz_node = Node(
        package='rtabmap_viz',
        executable='rtabmap_viz',
        name='rtabmap_viz',
        output='screen',
        parameters=[{
            'use_sim_time': True,
            'frame_id': 'base_link',
            'odom_frame_id': 'base_link',
            'subscribe_rgbd': False,
            'subscribe_scan': False,

            'approx_sync': True,
            'sync_queue_size': 30,

            #Quality Of Service QoS
            'qos_image': 2,
            'qos_camera_info': 2,
        }],
        remappings=[
            ('rgb/image', '/gazebo_depth_camera/image_raw'),
            ('depth/image', '/gazebo_depth_camera/depth/image_raw'),
            ('rgb/camera_info', '/gazebo_depth_camera/camera_info'),
        ]
    )

    # Delay rtabmap_node
    delayed_rtabmap = TimerAction(
        period=20.0,
        actions=[rtabmap_node, rtabmap_viz_node]
    )

    return LaunchDescription([
        niryo_launch,
        static_world_map,
        delayed_rtabmap,
    ])
```

Listing 5: YAML configuration file used by MoveIt 2 to integrate a point cloud source into the planning scene.

```
sensors:
  - niryo_depth_camera

niryo_depth_camera:
  sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
  point_cloud_topic: /gazebo_depth_camera/points
  max_range: 5.0
  point_subsample: 1
```

```
    padding_offset: 0.03
    padding_scale: 1.0
    max_update_rate: 1.0
    filtered_cloud_topic: /filtered_cloud
```

Listing 6: C++ file implementing a ROS2 node for motion planning and sampling of the desired trajectory.

```
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit_visual_tools/moveit_visual_tools.h>
#include <geometry_msgs/msg/pose_stamped.hpp>
#include <memory>
#include <rclcpp/rclcpp.hpp>
#include <thread>
#include <Eigen/Geometry>
#include <tf2_eigen/tf2_eigen.hpp>
#include <moveit/planning_scene_monitor/planning_scene_monitor.h>
#include <moveit/robot_trajectory/robot_trajectory.h>
#include <trajectory_msgs/msg/joint_trajectory_point.hpp>
#include <fstream>

int main(int argc, char * argv[])
{
// Start up ROS 2
rclcpp::init(argc, argv);

auto const node = std::make_shared<rclcpp::Node>(
    "desired_trajectory_sampling", ...
        rclcpp::NodeOptions().automatically_declare_parameters_from_
    overrides(true)
);

// Creates a "logger" for print out information or error messages
auto const logger = rclcpp::get_logger("desired_trajectory_sampling");

rclcpp::executors::SingleThreadedExecutor executor;
executor.add_node(node);
auto spinner = std::thread([&executor]() { executor.spin(); });

// Create the MoveIt MoveGroup Interfaces
using moveit::planning_interface::MoveGroupInterface;
auto arm_group_interface = MoveGroupInterface(node, "niryo_arm");

// Get the current pose of the end-effector before planning
auto current_pose = arm_group_interface.getCurrentPose();

// Specify a planning pipeline to be used for further planning
arm_group_interface.setPlanningPipelineId("chomp");

// Specify the maximum amount of time in seconds to use when planning
arm_group_interface.setPlanningTime(5.0);

// Set a scaling factor for optionally reducing the maximum joint velocity. ...
    Allowed values are in (0,1].
```

```
46 arm_group_interface.setMaxVelocityScalingFactor(1.0);
47
48 //  Set a scaling factor for optionally reducing the maximum joint ...
       acceleration. Allowed values are in (0,1].
49 arm_group_interface.setMaxAccelerationScalingFactor(1.0);
50
51 // Set a target pose for the end effector of the arm
52 auto const arm_target_pose1 = [&node]{
53     geometry_msgs::msg::PoseStamped msg;
54     msg.header.frame_id = "world";
55     msg.header.stamp = node->now();
56     msg.pose.position.x = -0.178;
57     msg.pose.position.y = -0.202;
58     msg.pose.position.z = 0.363;
59     msg.pose.orientation.x = 0.250;
60     msg.pose.orientation.y = 0.808;
61     msg.pose.orientation.z = 0.377;
62     msg.pose.orientation.w = -0.378;
63     return msg;
64 }();
65
66 std::ofstream ...
       ee_file("/home/tommaso/lab_ROS2/ws_moveit2/src/niryo_moveit2_config/cpp_scri
67 pt/desired_trajectory.txt");
68
69 //  Planning function
70 auto plan_and_execute = [&](const geometry_msgs::msg::PoseStamped& ...
       arm_target_pose) -> bool {
71     // Find the kinematic model of the robot
72     const moveit::core::JointModelGroup* joint_model_group =
73     arm_group_interface.getCurrentState()->getJointModelGroup("niryo_arm");
74     // Create a robot state to calculate IK
75     moveit::core::RobotState ...
           target_state(*arm_group_interface.getCurrentState());
76     // Calculate inverse kinematics to find joint values
77     bool found_ik = target_state.setFromIK(joint_model_group, ...
           arm_target_pose.pose, true);
78
79     if (found_ik) {
80     std::vector<double> joint_target;
81     target_state.copyJointGroupPositions(joint_model_group, joint_target);
82     RCLCPP_INFO(node->get_logger(), "IK found! Joint target:");
83     for (size_t i = 0; i < joint_target.size(); ++i) {
84         RCLCPP_INFO(node->get_logger(), "  Joint %zu: %f", i, joint_target[i]);
85     }
86     // Set the joint target obtained from IK
87     arm_group_interface.setJointValueTarget(joint_target);
88     } else {
89     RCLCPP_ERROR(node->get_logger(), "IK not found for desired pose!");
90     return false;
91     }
92
93     // Create a plan to that target pose
94     auto const [success, plan] = [&arm_group_interface] {
95     moveit::planning_interface::MoveGroupInterface::Plan msg;
```

```
96      auto const ok = static_cast<bool>(arm_group_interface.plan(msg));
97      return std::make_pair(ok, msg);
98      }();
99
100     // Execute the plan
101     if (success)
102     {
103
104     robot_trajectory::RobotTrajectory robot_traj(
105         arm_group_interface.getRobotModel(),
106         "niryo_arm"
107     );
108     robot_traj.setRobotTrajectoryMsg(
109         *arm_group_interface.getCurrentState(),
110         plan.trajectory_
111     );
112
113     std::vector<std::vector<double>> sampled_joint_trajectory;
114     std::vector<std::vector<double>> sampled_ee_trajectory;
115
116     double dt_sampling = 0.1;  // 100 ms = 10Hz
117     double traj_duration = robot_traj.getDuration();
118
119     for (double t = 0.0; t ≤ traj_duration; t += dt_sampling) {
120         // Create a shared_ptr at RobotState
121         moveit::core::RobotStatePtr state = ...
                std::make_shared<moveit::core::RobotState>(robot_traj.getFirstWay
122         Point());
123         robot_traj.getStateAtDurationFromStart(t, state);
124
125         // Extract joint positions
126         std::vector<double> joint_positions;
127         state->copyJointGroupPositions("niryo_arm", joint_positions);
128         sampled_joint_trajectory.push_back(joint_positions);
129
130         // Extract end-effector poses
131         const moveit::core::LinkModel* ee_link = ...
                state->getLinkModel("tool_link");
132         const Eigen::Isometry3d& ee_pose = ...
                state->getGlobalLinkTransform(ee_link);
133         const Eigen::Vector3d& position = ee_pose.translation();
134         const Eigen::Quaterniond orientation(ee_pose.rotation());
135
136         std::vector<double> ee_pose_vec = {
137             position.x(), position.y(), position.z(),
138             orientation.x(), orientation.y(), orientation.z(), orientation.w()
139         };
140         sampled_ee_trajectory.push_back(ee_pose_vec);
141     }
142
143     // Log joints position
144     for (size_t i = 0; i < sampled_joint_trajectory.size(); ++i) {
145         std::ostringstream joint_line;
146         joint_line << "Joint[" << i << "]: ";
147         for (double val : sampled_joint_trajectory[i])
```

```
148             joint_line << val << " ";
149         RCLCPP_INFO(logger, "%s", joint_line.str().c_str());
150     }
151
152     // Write to file and log pose EE
153     for (size_t i = 0; i < sampled_ee_trajectory.size(); ++i) {
154         std::ostringstream ee_line;
155         for (double val : sampled_ee_trajectory[i]) {
156             ee_line << val << " ";
157         }
158         ee_file << ee_line.str() << "\n";
159     }
160     ee_file.close();
161     } else {
162     RCLCPP_ERROR(logger, "Planning failed!");
163     }
164     // Exit
165     return true;
166 };
167
168 // Perform the first target
169 if (!plan_and_execute(arm_target_pose1)) {
170     RCLCPP_ERROR(logger, "Error reaching first pose, abort the program.");
171     return 1;
172 }
173
174 spinner.join();
175 return 0;
176 }
```

Listing 7: Excerpt from the file `desired_trajectory.txt`, showing the SE(3) waypoints sampled from the planned trajectory.

```
0.296064 0.0014779 0.425173 0.707807 0.00150363 0.706401 -0.00203438
0.297528 0.00592898 0.424887 0.709553 0.00599073 0.70458 -0.00813547
0.299814 0.0134475 0.424389 0.712389 0.0135349 0.701417 -0.0182507
0.302683 0.0241746 0.423644 0.716201 0.0242326 0.696725 -0.0322955
0.305782 0.0382895 0.422605 0.720824 0.0382144 0.690244 -0.0501429
```

Listing 8: Part of the launch file dedicated to starting the desired_trajectory_sampling node which performs trajectory sampling.

```
    # THE PART FOR THE DESIRED TRAJECTORY
    desired_traj_node = Node(
    package="niryo_moveit2_config",
    executable="desired_trajectory_sampling",
    output="screen",
    parameters=[
        moveit_config.robot_description,
        moveit_config.robot_description_semantic,
        moveit_config.robot_description_kinematics,
        moveit_config.planning_pipelines,
        {"use_sim_time": True},
```

```
12      ],
13      )
14
15      delayed_desired_traj_node = TimerAction(
16          period=20.0,                              # wait 20 seconds
17          actions=[desired_traj_node]
18      )
```

Listing 9: `CMakeLists.txt` file of the controller package.

```
1  cmake_minimum_required(VERSION 3.8)
2  project(op_space_controller)
3
4  if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
5    add_compile_options(-Wall -Wextra -Wpedantic)
6  endif()
7
8  # --- 1. FIND DEPENDENCIES ---
9  find_package(ament_cmake REQUIRED)
10 find_package(rclcpp REQUIRED)
11 find_package(rclcpp_lifecycle REQUIRED)
12 find_package(controller_interface REQUIRED)
13 find_package(hardware_interface REQUIRED)
14 find_package(pluginlib REQUIRED)
15 find_package(pinocchio REQUIRED)
16 find_package(Eigen3 REQUIRED CONFIG)
17 find_package(geometry_msgs REQUIRED)
18 find_package(sensor_msgs REQUIRED)
19 find_package(std_msgs REQUIRED)
20 find_package(tf2 REQUIRED)
21 find_package(tf2_ros REQUIRED)
22 find_package(tf2_eigen REQUIRED)
23 find_package(nav_msgs REQUIRED)
24 find_package(octomap_msgs REQUIRED)
25 find_package(trajectory_msgs REQUIRED)
26
27 add_definitions(-DROS_BUILD_SHARED_LIBS=1)
28
29 # --- 2. BUILD PLUGIN LIBRARY ---
30 add_library(op_space_controller SHARED
31   src/op_space_controller.cpp
32 )
33 # Set the C++ standard for the library
34 target_compile_features(op_space_controller PUBLIC cxx_std_17)
35
36 # Specify include directories for the library
37 target_include_directories(op_space_controller PUBLIC
38   $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
39   $<INSTALL_INTERFACE:include>
40 )
41
42 # --- 3. LINKING OF DEPENDENCIES ---
43 ament_target_dependencies(op_space_controller PUBLIC
44   rclcpp
```

```
45    rclcpp_lifecycle
46    controller_interface
47    hardware_interface
48    pluginlib
49    pinocchio
50    geometry_msgs
51    sensor_msgs
52    tf2
53    tf2_ros
54    nav_msgs
55    octomap_msgs
56    trajectory_msgs
57    tf2_eigen
58  )
59  target_link_libraries(op_space_controller PUBLIC
60    Eigen3::Eigen
61    tf2_ros::tf2_ros
62  )
63
64  # For visibility_control.h
65  target_compile_definitions(op_space_controller PRIVATE ...
        "OP_SPACE_CONTROLLER_BUILDING_DLL")
66
67  # Export as a plugin
68  pluginlib_export_plugin_description_file(controller_interface ...
        op_space_controller_plugin.xml)
69
70  # --- 4. INSTALLATION ---
71  # Installa la libreria del controller compilata
72  install(TARGETS
73      op_space_controller
74    EXPORT export_${PROJECT_NAME}
75    ARCHIVE DESTINATION lib
76    LIBRARY DESTINATION lib
77    RUNTIME DESTINATION bin
78  )
79
80  # Install library headers
81  install(
82    DIRECTORY include/op_space_controller/
83    DESTINATION include/op_space_controller
84  )
85  install(
86    DIRECTORY config
87    DESTINATION share/${PROJECT_NAME}
88  )
89
90  # --- 4. EXPORT DEPENDENCIES ---
91  ament_export_dependencies(
92    rclcpp
93    rclcpp_lifecycle
94    controller_interface
95    hardware_interface
96    pluginlib
97    pinocchio
```

```
  geometry_msgs
  sensor_msgs
  tf2_ros
  tf2_eigen
  nav_msgs
  octomap_msgs
  trajectory_msgs
)
# Export library target
ament_export_targets(export_${PROJECT_NAME} HAS_LIBRARY_TARGET)
ament_package()
```

Listing 10: `package.xml` file of the controller package.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" ...
    schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>op_space_controller</name>
  <version>0.1.0</version>
  <description>Operational space controller for the Niryo Ned2 manipulator, ...
      using Pinocchio for dynamics and SLAM-based pose estimation feedback ...
      for trajectory tracking.</description>
  <maintainer email="savino.tommaso10@gmail.com">Tommaso Savino</maintainer>
  <license>Apache License 2.0</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>rclcpp_lifecycle</depend>
  <depend>controller_interface</depend>
  <depend>hardware_interface</depend>
  <depend>pluginlib</depend>
  <depend>pinocchio</depend>
  <depend>geometry_msgs</depend>
  <depend>sensor_msgs</depend>
  <depend>trajectory_msgs</depend>
  <depend>nav_msgs</depend>
  <depend>octomap_msgs</depend>
  <depend>tf2</depend>
  <depend>tf2_ros</depend>
  <depend>tf2_eigen</depend>
  <depend>eigen3_cmake_module</depend>
  <depend>std_msgs</depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
    <controller_interface plugin="${prefix}/op_space_controller_plugin.xml"/>
  </export>
</package>
```

Listing 11: `op_space_controller_plugin.xml` file for registering the controller as a ros2_control plugin.

```
<library path="op_space_controller">
    <class name="op_space_controller/OpSpaceController" ...
        type="op_space_controller::OpSpaceController" ...
        base_class_type="controller_interface::ControllerInterface">
    <description>
      Operational Space Controller for the Niryo Ned2 manipulator, using
      Pinocchio for dynamics and SLAM-based pose estimation feedback for
      trajectory tracking.
    </description>
  </class>
</library>
```

Listing 12: Header file `op_space_controller.hpp` defining the ROS 2 operational space controller interface.

```
#ifndef OP_SPACE_CONTROLLER__OP_SPACE_CONTROLLER_HPP_
#define OP_SPACE_CONTROLLER__OP_SPACE_CONTROLLER_HPP_
#include <memory>
#include <string>
#include <vector>

#include "controller_interface/controller_interface.hpp"

#include "rclcpp_lifecycle/node_interfaces/lifecycle_node_interface.hpp"
#include "rclcpp/rclcpp.hpp"

#include <tf2_eigen/tf2_eigen.hpp>
#include <tf2_ros/buffer.h>
#include <tf2_ros/transform_listener.h>

#include "pinocchio/fwd.hpp"
#include "pinocchio/multibody/model.hpp"
#include "pinocchio/multibody/data.hpp"
#include "pinocchio/spatial/se3.hpp"
#include "pinocchio/spatial/motion.hpp"

#include "Eigen/Dense"

// Visibility macros
#include "op_space_controller/visibility_control.h"

namespace op_space_controller
{

struct EEWaypoint
{
  pinocchio::SE3 pose_desired;
  pinocchio::Motion velocity_desired;
  pinocchio::Motion acceleration_desired;
  rclcpp::Duration time_from_start;

  EEWaypoint():
```

```
38      pose_desired(pinocchio::SE3::Identity()),
39      velocity_desired(pinocchio::Motion::Zero()),
40      acceleration_desired(pinocchio::Motion::Zero()),
41      time_from_start(0, 0)
42      rclcpp::Duration
43    {}
44  };
45
46  class OP_SPACE_CONTROLLER_PUBLIC OpSpaceController : public ...
        controller_interface::ControllerInterface
47  {
48  public:
49    OpSpaceController();
50    ¬OpSpaceController() override = default;
51
52    controller_interface::CallbackReturn on_init() override;
53    controller_interface::InterfaceConfiguration ...
          command_interface_configuration() const override;
54    controller_interface::InterfaceConfiguration ...
          state_interface_configuration() const override;
55    controller_interface::CallbackReturn on_configure(const ...
          rclcpp_lifecycle::State & previous_state) override;
56    controller_interface::CallbackReturn on_activate(const ...
          rclcpp_lifecycle::State & previous_state) override;
57    controller_interface::CallbackReturn on_deactivate(const ...
          rclcpp_lifecycle::State & previous_state) override;
58    controller_interface::CallbackReturn on_cleanup(const ...
          rclcpp_lifecycle::State & previous_state) override;
59    controller_interface::CallbackReturn on_error(const rclcpp_lifecycle::State ...
          & previous_state) override;
60    controller_interface::return_type update(const rclcpp::Time & time, const ...
          rclcpp::Duration & period) override;
61
62  protected:
63    std::vector<std::reference_wrapper<hardware_interface::LoanedStateInterface>>
64      state_if_position_handles_;
65    std::vector<std::reference_wrapper<hardware_interface::LoanedStateInterface>>
66      state_if_velocity_handles_;
67    std::vector<std::reference_wrapper<hardware_interface::LoanedCommandInterface>>
68      cmd_if_effort_handles_;
69
70    // Parameters
71    std::vector<std::string> joint_names_;
72    std::string command_interface_type_;
73    std::string urdf_file_path_;
74    std::string base_link_name_;
75    std::string ee_link_name_;
76    std::string map_frame_name_;
77    std::string trajectory_file_path_;
78
79    // Controller gain
80    double trajectory_sample_dt_;
81    double kp_linear_, kd_linear_, kp_angular_, kd_angular_;
82
83    size_t num_joints_;
```

```
84
85    // Pinocchio model and data
86    pinocchio::Model model_;
87    std::shared_ptr<pinocchio::Data> data_;
88    pinocchio::FrameIndex ee_frame_id_pin_;
89    bool pinocchio_model_loaded_ = false;
90
91    // TF2 for SLAM pose
92    std::shared_ptr<tf2_ros::Buffer> tf_buffer_;
93    std::shared_ptr<tf2_ros::TransformListener> tf_listener_;
94
95    // Trajectory waypoints
96    std::vector<EEWaypoint> desired_trajectory_;
97    bool trajectory_active_ = false;
98    size_t current_trajectory_index_ = 0;
99
100   // Robot internal state
101   Eigen::VectorXd q_;
102   Eigen::VectorXd dq_;
103   Eigen::VectorXd tau_;
104
105   // Helpers
106   bool read_trajectory_from_file(const std::string & file_path);
107   bool wait_for_transform(const std::string & target_frame, const std::string ...
          & source_frame, const rclcpp::Duration & timeout);
108 };
109 }  // namespace op_space_controller
110
111 #endif  // OP_SPACE_CONTROLLER__OP_SPACE_CONTROLLER_HPP_
```

Listing 13: YAML file `op_space_controller_parameters.yaml` defining the ROS parameters.

```
1  niryo_op_space_controller:
2    ros__parameters:
3      # List of joint names this controller will handle
4      joints: [
5        "joint_1",
6        "joint_2",
7        "joint_3",
8        "joint_4",
9        "joint_5",
10       "joint_6"
11     ]
12
13     # Command interface type effort
14     command_interface_type: "effort"
15
16     # ----- Parameters for Pinocchio and TF -----
17     # Path to the robot URDF file
18     urdf_file_path: ...
           "/home/tommaso/lab_ROS2/ws_moveit2/src/op_space_controller/config/niryo_
19           ned2_pinoc.urdf"
20
21     # Names of reference frames used by the controller
```

```
22      base_link_name: "base_link"
23      ee_link_name: "tool_link"
24      map_frame_name: "map"
25
26      # ----- Parameters for the desired trajectory -----
27      # Path to the .txt file containing the trajectory waypoints
28      trajectory_file_path: ...
            "/home/tommaso/lab_ROS2/ws_moveit2/src/niryo_moveit2_config/cpp_script/
29           desired_trajectory.txt"
30
31      # ----- Controller gains -----
32      kp_linear: 40.0
33      kd_linear: 12.0
34      kp_angular: 20.0
35      kd_angular: 9.0
```