Insights from Developing a Robotic Bimanual Manipulation System

Atharva Anil Pradhan

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2025

Committee:

Siddhartha Srinivasa

Santosh Devasia

Ashis Banerjee

Per Reinhall

Program Authorized to Offer Degree:
Mechanical Engineering

University of Washington

Abstract

Insights from Developing a
Robotic Bimanual Manipulation System

Atharva Anil Pradhan

Chair of the Supervisory Committee:
Siddhartha Srinivasa
Paul G. Allen School of Computer Science & Engineering

This thesis presents the development of a bi-manual robotic manipulation system utilizing open-source robotics frameworks, including *ros2_control* and *MoveIt2*. In the realm of control, we address key challenges such as coordinating multiple robots, independently managing the hardware life-cycle via a state machine, and operating under limited communication bandwidth. For motion planning, we enhance sampling-based planners by incorporating goal set planning by leveraging the Task Space Regions (TSR) framework, enabling intuitive constraint representation. Additionally, we utilize *IKFast*, an analytical inverse kinematics solver, to compute inverse kinematics solutions in as little as one microsecond. Finally, we explore both synchronous and asynchronous execution strategies for trajectory execution in a dual-arm setup and discuss the scope for future work.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

ROS2: Robot Operating System (version 2.0)

IK: Inverse Kinematics

PID: Proportional-Integral-Derivative

TSR: Task Space Regions

URDF: Unified Robot Description Format

DOF: Degrees of Freedom

WAM: Whole Arm Manipulator robot arm (colloquially referred to as the left/right *WAM* throughout this document.)

YAML: YAML Ain't Markup Language

CANBUS: Controller Area Network bus

PCIE: Peripheral Component Interconnect Express (bus standard meant to attach hardware devices in a computer)

API: Application Programming Interface

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor and Master's thesis committee chair, Professor Siddhartha Srinivasa, for his invaluable guidance and unwavering support throughout this project. I am also sincerely thankful to my thesis committee members, Professor Santosh Devasia, Professor Per Reinhall, and Professor Ashis Banerjee, for their encouragement and insights. Additionally, I am profoundly grateful to the members of the Personal Robotics Lab at the University of Washington, especially Quanquan Peng, Ethan Gordon, Helen Wang, Amal Nanavati, and Alex Lin, for their invaluable assistance and support during my time in the lab.

DEDICATION

Dedicated to my parents and friends for their constant support and encouragement

Chapter 1
INTRODUCTION

1.1 Motivation

Bi-manual robotic manipulation is essential for performing complex tasks that require coordinated motion between two arms, such as tearing a piece of paper, opening a bottle by twisting its cap, or folding clothes. However, enabling such capabilities in robotic systems presents several challenges, including system complexity, dynamic task execution, and motion planning in cluttered environments. This work is motivated by the need to develop a bi-manual robotic system that can efficiently handle these challenges while leveraging open-source frameworks for scalability and ease of development. Hence, the motivation for this work is threefold:

- Managing System Complexity in Bi-Manual Manipulation: The integration of multiple high-degree-of-freedom components i.e., two 7-DOF *WAM* arms, two *Barrett Hand*s, and two Vention linear actuators introduces significant system complexity. Coordinating these components requires a structured approach to hardware control and motion planning. To address this, this work leverages open-source robotics frameworks such as *ros2_control* and *MoveIt2* [1], which provide standardized tools for hardware integration, real-time control, and motion planning.

- Handling Dynamic Task Constraints: Certain tasks, such as replacing a tire or juggling with robotic arms, impose constraints on velocity and synchronization. These tasks require precise coordination, where one arm may need to move at a different speed than the other or follow an asynchronous motion pattern. Developing a control architecture that supports both synchronous and asynchronous motion execution is essential for

expanding the capabilities of bi-manual robotic systems in real-world applications.

- Motion Planning in Cluttered Environments: Motion planning in constrained environments presents challenges related to grasp feasibility. Traditional single-goal planning approaches often fail due to pose being outside of robot's joint limits, clutter, pose uncertainty. To improve robustness, this work employs a multi-goal planning approach using Task Space Regions (TSRs) [2] as a constraint representation. By defining a set of feasible goal poses instead of a single goal, TSR-informed planning increases the likelihood of success while accounting for uncertainties in grasping.

By addressing these three aspects, this work aims to advance the development of adaptable and efficient bi-manual robotic systems.

1.2 Literature Review

Bimanual or dual-arm robotic manipulation often requires coordinated control architectures to synchronize arms. Early work in [3] introduced a behavior-based control system for assistive bimanual tasks, structuring dual-arm actions through a state-based behavior hierarchy. Later, researchers explored hierarchical or finite state machines (FSMs) for coordinating two arms. For example, [4] implemented a central state machine to orchestrate two anthropomorphic arms in a coordinated task, drawing inspiration from human bimanual strategies. To improve flexibility and reusability, [5] proposed a modular FSM-based architecture for dual-arm programming, enabling synchronous/asynchronous arm behaviors through well-defined state transitions. More recently, [6] introduced a multi-arm control framework that synthesizes multiple interacting FSMs with priority schemes, allowing coordinated manipulation in dynamic settings. These works span foundational approaches and modern advances, highlighting the enduring role of structured state-based control in multi-arm robot coordination.

Manipulation tasks often involve pose constraints, such as keeping a grasp within a certain alignment or moving an object to a defined workspace. Traditional motion planning to a single goal pose may be overly restrictive or infeasible when multiple valid goal poses exist.

The Task Space Regions [2] (TSR) framework provides an elegant solution for encoding such constraints. TSRs have also been used in the past to address pose uncertainty [7]. In addition, the TSR framework has also been used in complex robotic systems such as whole-body planning for the HRP3 humanoid robot [8].

In dual-arm robotics, execution can be synchronous or asynchronous [9]. Synchronous execution tightly coordinates both arms, ensuring step-by-step motion synchronization for tasks like lifting an object or performing a precise handover. While it simplifies collision avoidance, it can be inefficient, as one arm may remain idle while waiting for the other. Additionally, synchronizing all joints constrains both robots to move at the same velocity, limiting the quality of generated plans.

Conversely, asynchronous execution allows each arm to move independently, improving efficiency for tasks that don't require direct coordination, such as simultaneous pick-and-place operations. However, it introduces the risk of collisions. To address this, trajectory reservation techniques dynamically allocate workspace to prevent inter-arm conflicts [10]. In this approach, as one arm moves, it marks occupied volumes in space-time, which are then treated as temporary collision objects by the other arm's planner. This allows for safe concurrent motion, maximizing efficiency without compromising collision safety.

Chapter 2

ROBOT CONTROL

2.1 Description of Hardware Components

As shown in Figure 2.1, the Geodude hardware setup consists of two cable driven *Barrett WAM* robot (7-DOF) arms. These robot arms are lightweight, back-drivable and emit minimal operational noise.

Both robot arms are controlled from an external control computer over the CAN bus, which guarantees real-time control for a control loop running typically at 500 Hz. The CAN bus data cable plugs into a PCIe card mounted on the motherboard of the control computer, which is also the starting point for sending control signals in the form of CAN frames.

Figure 2.1: The Geodude hardware setup consisting of two 7-DOF *Barrett WAM* arms, *Barrett Hands*, stereo cameras and linear actuators

On the other end, the CAN bus data cable plugs into the base of the *WAM* arms from where the robot's internal CAN bus cable begins and runs all the way to the seventh joint of the robot or end of the kinematic chain. On the CAN bus branching out from the seventh joint, a three-fingered robotic (*Barrett Hand BH8-282* in Figure 2.2) is mounted as end-effector on both robot arms.

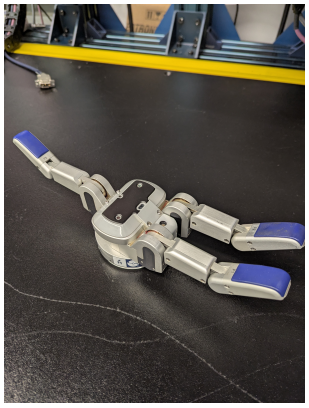

Figure 2.2: *Barrett Hand*

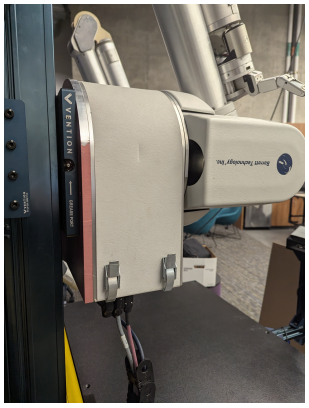

Figure 2.3: Linear actuator (in blue)

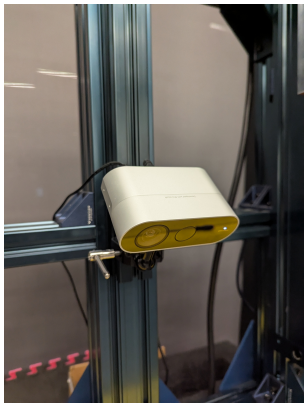

Figure 2.4: Orbbec depth camera

The *Barrett Hand* is also cable driven and has finger-tip torque sensors, tactile sensors for feedback. The bases of both arms are mounted on linear actuators (shown in figure 2.3) giving the system an additional degree of freedom. The linear actuators themselves are part of a custom designed Vention frame and are controlled from a Vention *MachineMotion* control box that can be accessed via Ethernet. There are two Orbbec Femto Bolt depth cameras (shown in figure 2.4) mounted on the Vention frame for collision avoidance and object detection.

The entire system is controlled using two computers. One of them, referred to as the control computer, runs a linux low-latency kernel offering certain soft real-time guarantees on running processes. This computer is responsible for performing all control calculations, hosting the hardware interface, and sending control signals to the robot arm, end-effector, and the linear actuator alike. The other computer, referred to as the perception computer, hosts applications that require 3D rendering and/or significant computing resources. For instance, running a motion planning framework, photo-realistic simulation application, or deploying an object detection model would all be carried out on the perception computer.

2.2 System Architecture

2.2.1 Component wise hardware interfaces

For each hardware component shown in Figures 2.1 - 2.3, there exists a *ros2_control* hardware interface for sending control signals by going through a series of states in a predictable manner at run-time. As shown in Figure 2.5, the system has three hardware interfaces:

1. **Barrett Hardware**: To control the Barrett robots i.e. the Barrett *WAM* (robot arm) and *Barrett Hand* (end-effector). This is further instantiated as **Barrett Hardware Left** and/or **Barrett Hardware Right** depending on those in use.

2. **Vention Hardware**: To control the Vention linear actuator.

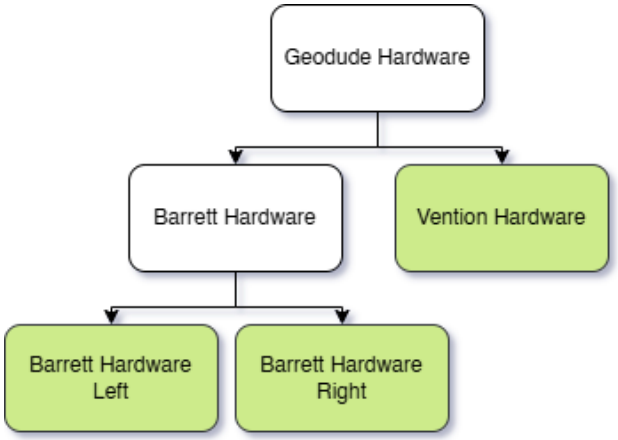

Figure 2.5: Component wise hardware interfaces

Every hardware interface is a collection of *C++* abstractions that give us access to a state machine that is essentially hollow. We then fill this state machine with our own implmentation of hardware life-cycle. In other words, we decide what should happen when the hardware starts, how it should run, how it should stop and what happens in case of an exception. All of this happens in a series of steps that are part of the *ros2_control* state machine.

2.2.2 Need for a state machine

The manufacturer (Barrett technology LLC) provides a real-time controls API, i.e. *libbarrett*, for controlling the Barrett *WAM*s (Whole Arm Manipulator - Figure 2.1) and *Barrett Hand*s (Figure 2.2). Similarly, Vention provides a web interface for controlling linear actuators

(Figure 2.3). However, controlling the two *WAM* arms, *Barrett Hand*s and the Vention linear actuators as a homogeneous system, and even managing control modes for each hardware component in a centralized manner necessitates the need for a state machine that can handle all communications in real-time so that each hardware component is equally prioritized and the system as a whole does not fail if and when a single hardware component fails. Hence, *ros2_control* is an ideal candidate because it provides a state machine that can handle these communications in a real-time manner, especially when it comes to interfacing multiple hardware components and running them simultaneously or asynchronously.

2.2.3 Why ros2_control?

The *ros2_control* is a control library for (real-time) robot control that comprises a set of packages for controlling robot hardware through low-level hardware interfaces and high-level control algorithms. The hardware interface serves as a standardized interface for reading data from sensors and writing to actuators in closed-loop or open-loop control. The control algorithms are made available off-the-shelf as controller plugins and can be invoked by simply adding the desired controller plugin (along with the corresponding joints on the robot that the controller is supposed to actuate). Moreover, this gives non-expert users the ability to get started with a fairly complicated system (as such) without reinventing the wheel, especially when using something as common as a way-point tracking/joint trajectory controller.

2.2.4 State Machine life-cycle

Non-real time states: The state machine takes in as inputs, the robot URDF which specifies each joint on the robot that can be actuated as well as the configuration file of the controllers specifying which joints on the robot does a given controller have access to. When the *ros2_control* state machine begins its life-cycle, it goes through a series of steps through the functions: *on_init(), export_interfaces(), on_configure(), on_activate()*. The work-flow of these functions is demonstrated in Figure 2.6. Note how each step is responsible for performing some kind of check on either the robot URDF or the hardware to ensure. Hence,

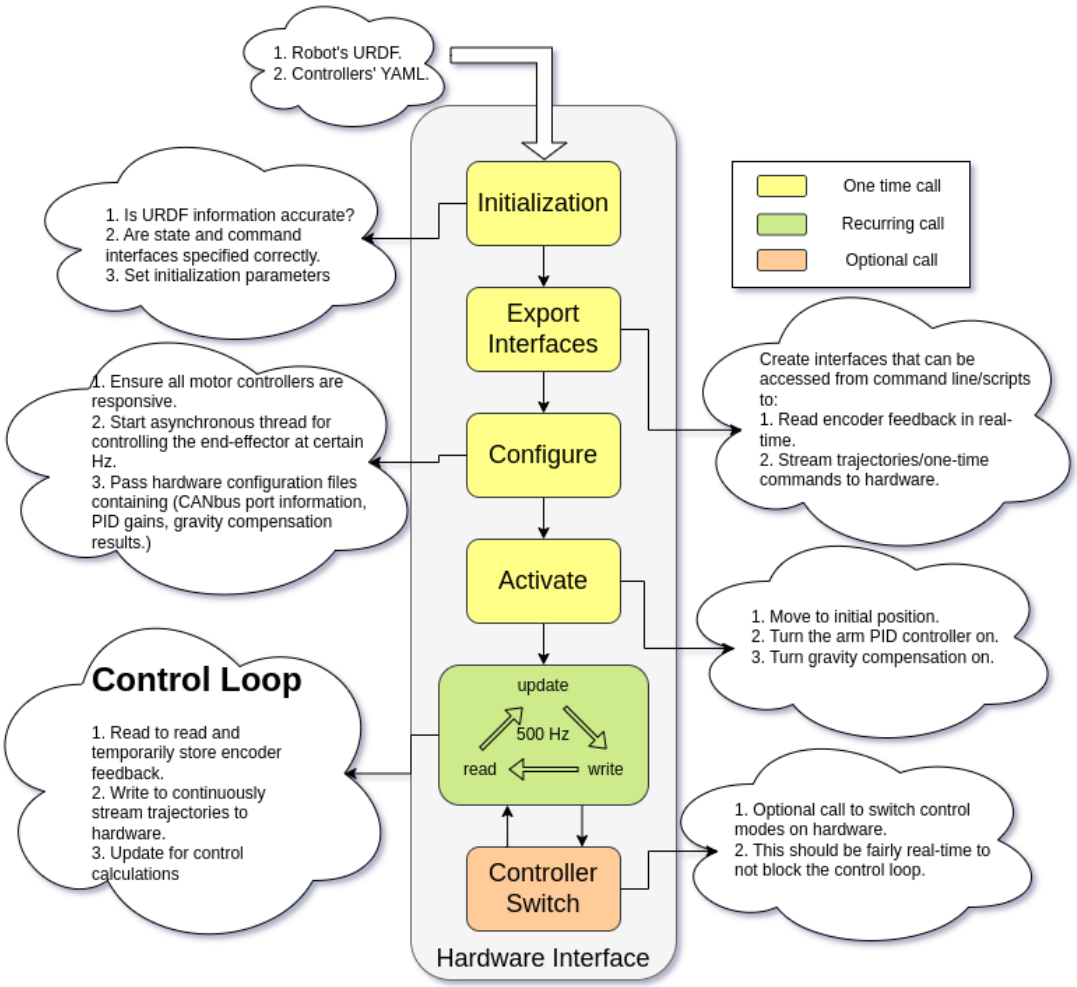

Figure 2.6: Flow diagram of the *ros2_control* state machine

it is critical that each of these checks passes, or else connection to the hardware is not initiated. If an error occurs after these checks pass and while the control loop is running, the user can exit the hardware interface using control+c key press and the robot is configured to go back to home position.

Real-time states: As shown in Figure 2.6, the control loop is where the real-time computation and transmission of control commands as well as the receiving of joint encoder

data takes place. A typical control loop comprises read, update, and write methods, all responsible for a real-time operation. For the *Barrett WAM*, the control loop executes at 500 Hz and for the *Barrett Hand*, the control loop executes at 200 Hz, whereas for the linear actuator, the control loop executes at 10 Hz. The different control loop frequencies reflect the physical constraints and operational needs of each hardware component. The *WAM* requires a high frequency (500 Hz) due to its torque-based control, whereas the Vention actuator operates at 10 Hz due to its slower mechanical response. In case of *ros2_control*, the control loop is partly handled by the resource manager and partly by the controller manager. The read and write methods, handled by the resource manager, are responsible for requesting and temporarily storing joint encoder data from the robot through *libbarrett*. The joint encoder data are then used as a reference by the controller manager's update method to compute the necessary magnitude of control input to minimize the position/velocity/effort error (in case of feedback controllers). This computed control input is then sent to the ros2 controllers and eventually the hardware (through *libbarrett*) by the resource manager's write method.

The way that this single hardware interface instance comprising the non-real-time and real-time states fits into the broader context of the system is that three such hardware interfaces are instantiated at run-time; One for the left *WAM* and *Barrett Hand*, another one for the right *WAM* and *Barrett Hand* and another one for the Vention actuators as shown in Figure 2.5. In this way, each hardware component goes through this hardware life-cycle, making the hardware behavior legible and predictable.

2.2.5 Bridge between ros2_control *and* libbarrett

The *WAM*s are fundamentally designed for high-frequency torque control (500 Hz). This means that any kind of input to the *WAM*s will be mapped to joint torques using a PID loop. Hence, if one wanted to use a JointTrajectory controller (available in *ros2_controllers*) with velocity as the command interface, there would be two PID control loops involved here. The first PID control loop would run on the *ros2_control* layer for converting positions from

the robot trajectory to velocities and then the second PID control loop, running in *libbarrett*, would convert these velocities that are passed through the hardware interface to torques. This is shown in Figure 2.7.

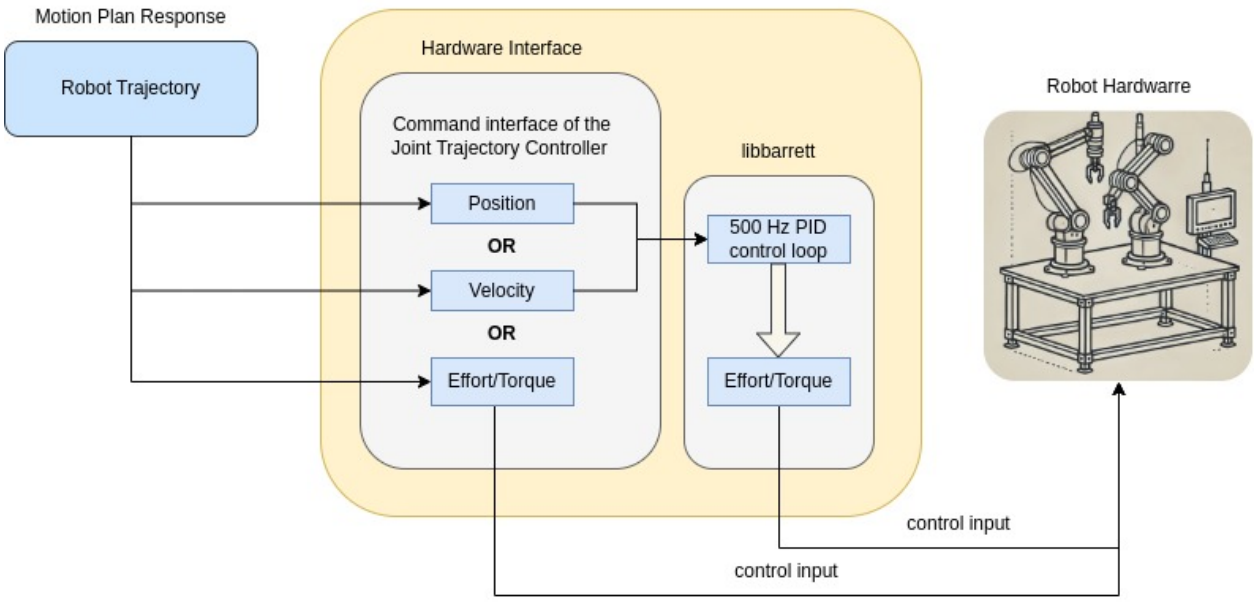

Figure 2.7: Bridge between *ros2_control* and *libbarrett*

2.3 Communication Bandwidth Issue

The robot arm and the end effector are independent robots that share a communication link i.e. CANbus (Controller Area Network bus data cable) to receive control signals from the control computer, as shown by yellow box in the Figure 2.11. This makes it challenging for the two to be controlled simultaneously as the number of control signals that can be sent in a given time frame are limited by the CANbus baud rate (1 Mbaud CANbus in this case). If we try to control the *Barrett Hand* at the same time as the *WAM* arm, the required bandwidth exceeds the available bandwidth. To handle this exception, the hardware API i.e. *libbarrett* synchronizes the two high-frequency control loops and runs a single low-frequency control loop that delivers control commands to the motor controllers of both

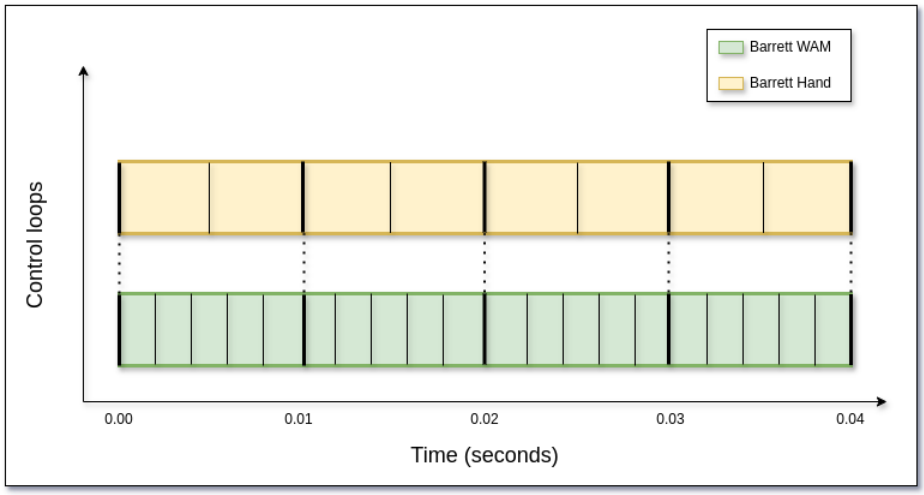

Figure 2.8: Control loop synchronization

the hand and the arm. Figure 2.8 shows the darker lines, which indicate the control signals that are delivered as a result of loop synchronization. However, the lighter lines indicate the signals that are dropped. In case of the *WAM* arm, this means that out of every 10 control commands, each sent 2 milliseconds apart, only the 5th and the 10th singals are sent to the motor controllers. This has a blocking effect on the *WAM*'s control loop as evident by the jumps in the recorded velocity of the robot's elbow joint as a function of time shown in Figure 2.9. Figure 2.10 indicates the same joint velocity graph that the arm would follow if the *Barrett Hand* were not controlled simultaneously. Ideally, we want Figure 2.9 to look like Figure 2.10 instead. To achieve the same, we implement a relay mechanism in the Controller Switch step (as shown in Figure 2.6) of the Hardware Interface. This mechanism allows us to limit the control signals on the communication link by only sending control signals to the *Barrett Hand* when we intend to grasp an object with the robot. This can be better understood with the help of Figure 2.11 which shows the communication link to the Hand Controller being highlighted when the robot is grasping an object with the *Barrett Hand*. At this point, loop synchronization still comes into effect but since the arm is not moving while grasping, we don't experience the control jitter. Similarly, when the communication link is disabled, we can move the arm without causing any control jitter because the re-

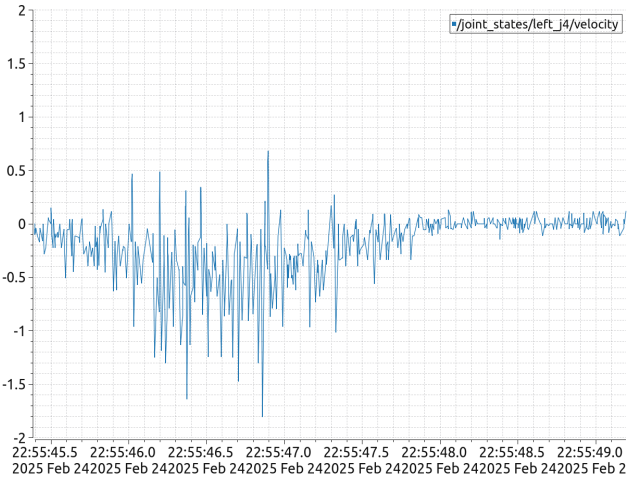

Figure 2.9: Elbow joint velocity against time when hand and arm are controlled simultaneously.

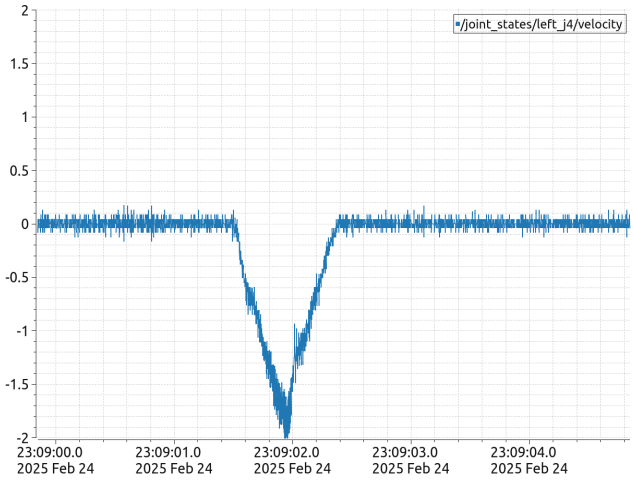

Figure 2.10: Elbow joint velocity against time when hand and arm are controlled asynchronously.

quired communication link bandwidth is still less than the available bandwidth. In this way, we are able to run two high-frequency control loops using the same communication link.

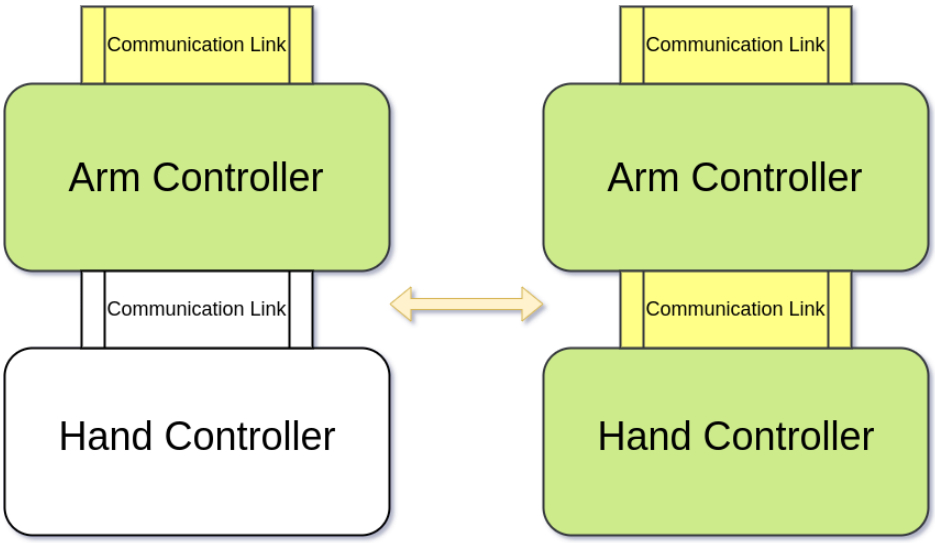

Figure 2.11: Controller switching mechanism

Chapter 3
MOTION PLANNING

We use the *MoveIt2* [1] motion planning framework as the base of our motion planning stack. By doing so, we are able to leverage a wide array of sampling-based planners from the Open Motion Planning Library without necessarily reinventing the wheel to implement planning algorithms. Secondly, this enables us to improve the throughput of these planners by optimizing how they conventionally work by leveraging the Task Space Regions (TSR framework) as shown in Section. Using *MoveIt2* also allows us to use various Inverse Kinematics (IK) solvers in a plugin architecture for IK computation.

3.1 Planning with multiple goals

3.1.1 Challenges in Motion Planning

Traditionally, for reaching to grasp an object, any motion planning algorithm would take as input a single Cartesian 6-DOF pose and generate a trajectory to make the end-effector reach the specified pose, as demonstrated by Figure 3.1. However, if that pose was infeasible either due to a joint limit violation of one or more of the robot joints, planning would fail, and the user would have to re-plan. Secondly, even if we assume that the end-effector pose was feasible and the robot could successfully generate a motion plan to reach that pose, grasping action could fail if the pose of the object predicted by the camera

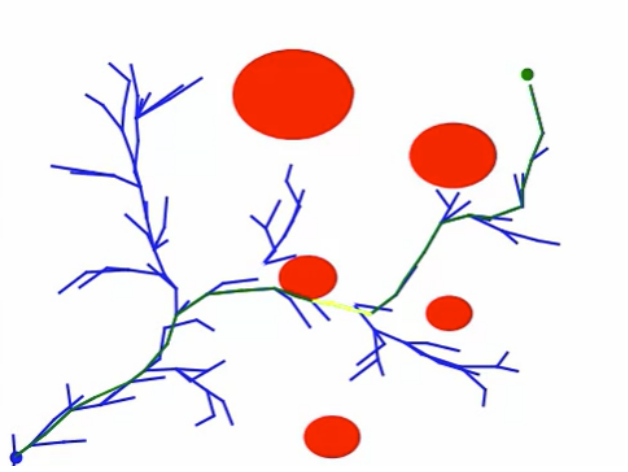

Figure 3.1: Single goal planning with RRT*

is inaccurate. Lastly, if the object we want to grasp is surrounded by clutter, planning would fail again because in this case, the robot is required to either assume a grasp pose such that the grasp is not obstructed or first pick the objects that are in front of the object that we want to eventually grasp. Furthermore, depending on how complex the planning scene is, re-planning could take up a significant amount of time, delaying the whole planning process and increasing planning time. The source code for generating Figure 3.1 was adapted from this article.

3.1.2 More than conventional motion planning

To tackle the above three challenges to motion planning, we incorporate an aspect of redundancy in our planning so that we have multiple back-ups in case planning fails due to one or more of the above reasons. Specifically, we incorporate redundancy by constructing a list of grasp poses and passing the same as input to the planner. Doing so lets the motion planner decide the most feasible grasp pose without letting the plan fail, thereby saving the time required for re-planning. The advantage of such an approach is demonstrated by Figure 3.2 where the green dots represent goals and the big red circles represent obstacles. Notice how the length of the tree is smaller compared to the tree shown in Figure 3.1. The length of the tree has a direct correlation with collision checking, such that the more edges/branches there are, the greater the time spent in evaluating those edges i.e. collision checking. Similarly, more time spent on collision checking implies more overall time spent in planning. This goes to show that single goal planning (shown in Figure 3.1) may/may not always produce the best results in terms of planning time. The source code for generating Figure 3.2 was adapted from this article.

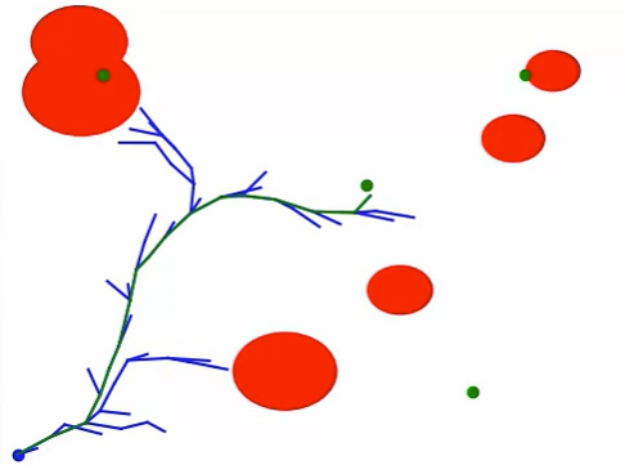

Figure 3.2: Multi goal planning with RRT*

3.2 Task Space Regions

To construct the list of grasp poses that are input to the planner, we use a sampling-based approach based on the Task Space Regions (TSR.) [2]. A TSR is a simple description of robot manipulation task that is useful for motion planning. The building block of a TSR is given by equation (3.1)

$$\mathbf{T}_b^a = \begin{bmatrix} \mathbf{R}_b^a & \mathbf{t}_b^a \\ 0 & 1 \end{bmatrix} \tag{3.1}$$

where $\mathbf{T}_b^a$ is a homogeneous 4 x 4 matrix that defines the 6-DOF pose of b w.r.t. a. It consists of a 3 x 3 rotation vector $\mathbf{R}_b^a$ and a 3 x 1 translation vector $\mathbf{t}_b^a$. Based on this, a TSR comprises three components defined below. Figure 3.3 visualizes the same.

- $\mathbf{T}_w^0$: Transformation matrix from origin/fixed frame 0 to TSR frame w.
- $\mathbf{T}_e^w$: Transformation matrix from TSR frame w to end-effector frame e.
- $\mathbf{B}_w$: The matrix of bounds (in Euler angle conventions) for sampling a TSR w.r.t. TSR frame w shown by equation (3.2).

$$\mathbf{B}_w = \begin{bmatrix} x_{\min} & x_{\max} \\ y_{\min} & y_{\max} \\ z_{\min} & z_{\max} \\ \psi_{\min} & \psi_{\max} \\ \theta_{\min} & \theta_{\max} \\ \phi_{\min} & \phi_{\max} \end{bmatrix} \tag{3.2}$$

where x, y, z are the translation components and ψ, θ and ϕ are the rotation components.

3.2.1 How to specify multiple goals using TSRs

A TSR comprises three key transformations:

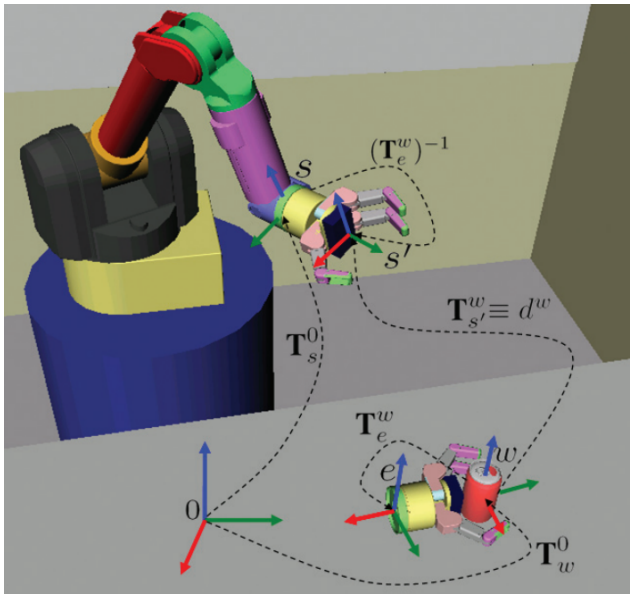

Figure 3.3: Transforms involved in defining a TSR

- $\mathbf{T}_w^0$: Transformation from the world frame to the TSR frame w.
- $\mathbf{T}_{\text{sample}}^w$: A sampled pose in the TSR frame.
- $\mathbf{T}_e^w$: The transformation from the TSR frame to the end-effector frame, which constrain the end-effector.

3.2.2 Constraining the End-Effector with $\mathbf{T}_e^w$

The transformation $\mathbf{T}_e^w$ defines how the end-effector is positioned relative to the TSR frame. This ensures that the grasp pose adheres to task constraints. It is given by:

$$\mathbf{T}_e^w = \begin{bmatrix} 0 & 0 & 1 & -\text{offset} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \text{height} \times 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

This matrix ensures:

- End-effector is aligned along the z-axis of the TSR frame.
- Gripper is positioned at the center height of the object (height $\times$ 0.5).
- Offset along the x-axis ensures the gripper maintains an appropriate grasp distance.

3.2.3 Sampling in the TSR Frame

Typically, a pose is sampled within the matrix of bounds $\mathbf{B}_w$ defined in equation (3.4):

$$\mathbf{B}_w = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -0.02 & 0.02 \\ 0 & 0 \\ 0 & 0 \\ -\pi & \pi \end{bmatrix} \tag{3.4}$$

This matrix allows:

- No variation in the x and y directions.
- Small vertical variation (± 2 cm) in z, allowing minor height adjustments.
- Rotation about the z-axis ($\pm\pi$) to allow different wrist orientations.

The sampled transform in the TSR frame is:

$$\mathbf{T}^{w}_{\text{sample}} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & z_{\text{sample}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

where z_{sample} is sampled from $\mathbf{B}_w$.

3.2.4 Transforming to the Robot's Base Frame

Using the TSR constraints, the final sampled pose in the robot's base frame is computed as:

$$\mathbf{T}^0_{\text{sample}} = \mathbf{T}^0_w \mathbf{T}^w_{\text{sample}} \mathbf{T}^w_e \tag{3.6}$$

Expanding:

$$\mathbf{T}^0_{\text{sample}} = \begin{bmatrix} \mathbf{R}^0_w & \mathbf{t}^0_w \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}^{\text{sample}}_w & \mathbf{t}^{\text{sample}}_w \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}^e_w & \mathbf{t}^e_w \\ 0 & 1 \end{bmatrix} \tag{3.7}$$

This transformation ensures that the sampled grasp pose adheres to the TSR constraints, aligning the end-effector appropriately.

3.2.5 Generating Multiple Grasp Poses

By repeating this process, multiple grasp poses are obtained by varying:

- The vertical height z_{sample} within the small allowable range.
- The wrist rotation ψ, allowing different grasp orientations.

3.2.6 Summary

1. Define the TSR for the cylindrical object.
2. Use $\mathbf{T}^w_e$ to enforce the gripper's grasp alignment.
3. Sample a pose $\mathbf{T}^w_{\text{sample}}$ within the TSR bounds $\mathbf{B}^w$.
4. Convert to the robot's base frame using equation (3.6).
5. Repeat to generate multiple grasp poses as shown by red arrows in Figure 3.4.

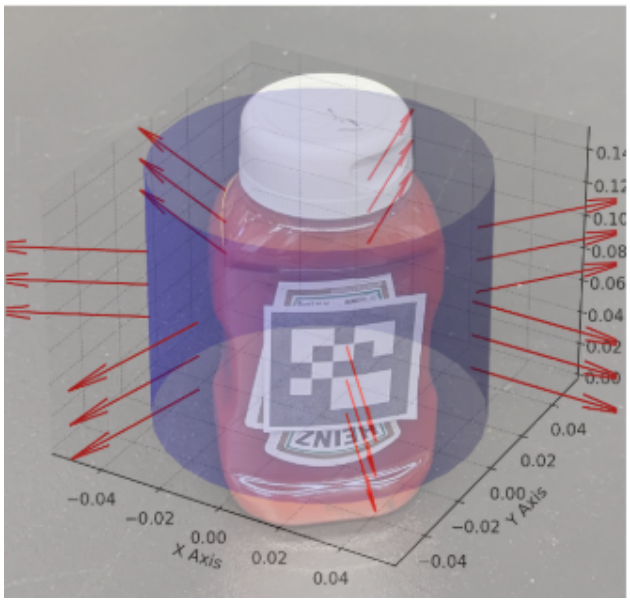

Figure 3.4: Visualization of the poses sampled from a TSR chain

3.3 Inverse Kinematics

3.3.1 Introduction to IK

Given a robot's joint values, the position and orientation of the end-effector with respect to a fixed link (usually the base link of the robot) can be calculated directly using homogenous transforms, a process known as forward kinematics [11]. Inverse kinematics makes use of these kinematics equations to determine the joint values that provide a desired configuration (position and orientation) for each of the robot's end-effectors.

Numerical vs Analytical IK: Broadly, there are two approaches to solving IK:

Analytical IK: This approach uses mathematical equations derived from the kinematics of the robot. These equations directly map the desired end-effector pose to joint angles. Once derived, the solution is essentially a look-up table search that is executed quickly in *O(1)* time. Because no iterative optimization is used, there is no possibility of getting stuck in local minima. Hence, the computation time is constant and extremely fast.

Numerical IK: This approach uses iterative methods (e.g., Newton-Raphson, gradient descent) to converge to a solution based on a convergence criterion. In this case, each iteration requires evaluating the forward kinematics and the Jacobian matrix, and accordingly adjusting the joint configuration step-by-step. The number of iterations depends on complexity of the problem and the convergence criteria, making it slower and less predictable in computation time [12].

3.3.2 IKFast

IKFast [13] is an analytical IK solver developed by Rosen Diankov et al. for the OpenRAVE motion planning framework. One of the benefits of using *MoveIt2* is being able to leverage off the shelf IK solvers such as *IKFast*. Given the context of analytical and numerical approaches, *IKFast* is able to compute IK solutions as fast as 1 microsecond, as shown in Figure 3.5. *IKFast* can be easily set up as a *MoveIt2* plugin following the instructions on the *Moveit2* documentation page. The source code for generating these benchmarks can be found at *ik_benchmarking*.

3.3.3 How IKFast handles redundancy

The end-effector pose in Cartesian space is a 6-DOF pose (i.e., three DOF for position xyz and three DOF for orientation rpy). To achieve a fully specified end-effector pose, a manipulator needs at least 6 DOF. However, since the *WAM* is a 7-DOF robot, it has one redundant DOF, making the IK problem under-constrained. In such cases, multiple joint configurations can satisfy the same end-effector pose. *IKFast* leverages this redundancy by iteratively discretizing one joint (for the *WAM*, this is typically Joint 3) and then solving IK for the remaining six joints. This approach ensures a solution while accounting for the robot's redundancy.

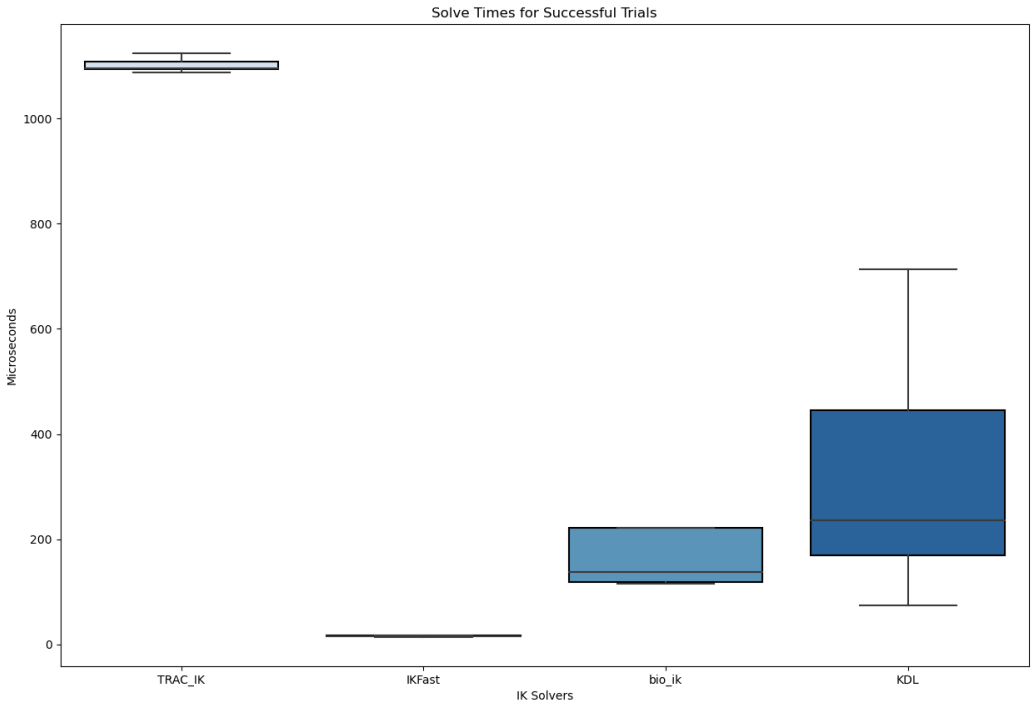

Figure 3.5: *IKFast* solve times comparison

3.3.4 IKFast Benchmarking

Figure 3.5 compares *IKFast* with 3 other numerical IK solvers that are available off the shelf as *MoveIt2* plugins. On the Y-axis, it indicates the time taken in microseconds to obtain an IK solution given a random valid 6-DOF pose in cartesian space. On the X-axis, is the name of the IK Solver. For a reasonable estimate of the solver runtime, a sample size of 10000 random, valid cartesian poses was considered. The bar plot indicates the minimum to maximum solve times required to evaluate different samples with the average solve time shown by a horizontal line through the bar. As demonstrated by the plot, *IKFast* solver takes a single microsecond to compute an IK solution. For a fair comparison, the same input parameters to the IK solver were being used.

The parameters used were:

1. *kinematics_solver_search_resolution*: Specified in radians, this is the resolution used by the IK solver to search over the redundant space for inverse kinematics, e.g. Iterative joint value discretization of a 7-DOF arm with the specified resolution.

2. *kinematics_solver_timeout*: Specified (in seconds) for each iteration that the solver runs. If this timeout is larger than the planning time, it might often lead to planning failures. Upon timeout, the IK solver will restart the search from a random seed state.

3. *kinematic_solver_attempts*: The number of times that the solver is allowed to restart after not being able to find an IK solution within kinematics_solver_timeout seconds.

3.4 Collision Avoidance

3.4.1 Background in Occupancy Grid Mapping

Occupancy Grid Mapping [14] is an approach that is typically used by mobile robots for 2D scene representation and collision avoidance. It represents the scene as occupancy grid cells that can be free or occupied depending on the occupancy probability. The occupancy probability of each cell is independent of each other, which may not always be the case, but greatly simplifies the calculations. In addition, this approach also assumes that the world is static.

3.4.2 OctoMap - A 3D adapatation of Occupancy Grid Mapping

In our case, we use the OctoMap [15] framework (available off-the-shelf with *MoveIt2*) which implements a 3D Occupancy Grid Mapping approach using the Octree data structure to represent the scene using voxels. It uses a static-state binary Bayes filter to iteratively model the occupancy probability required to represent the state of each voxel. It takes in as input, either a Depth Image or a Point Cloud and outputs voxels that envelope objects

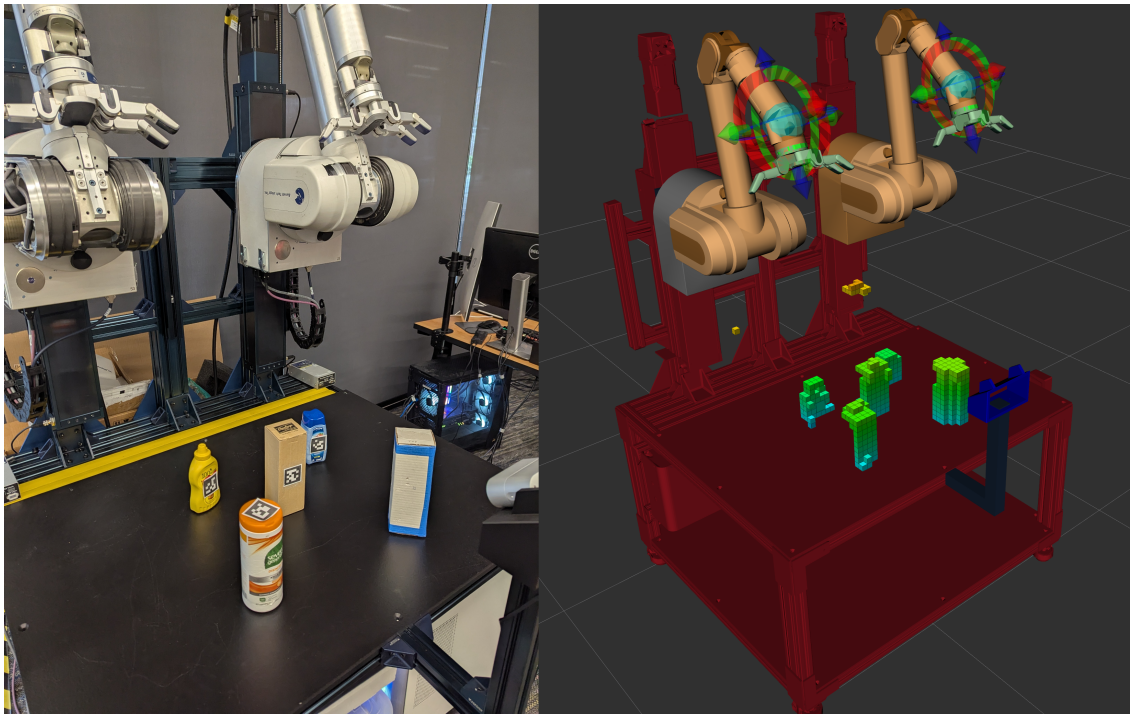

Figure 3.6: Scene representation using OctoMap

detected in the planning scene. These voxels are then treated as collision objects to inform collision checking. Figure 3.6 shows a scene with real-world collision objects on the left and the corresponding OctoMap representation on the right.

3.4.3 How it works

The input to the OctoMap plugin subscribes to the depth image of the camera to receive periodic updates about state of the scene to update the state of obstacles and free space. Hence, if there is an object in the planning scene that the robot needs to grasp, we need to get rid of the voxels surrounding that object in order to allow collision between the robot and the object for the grasp to succeed. However, if OctoMap is running, planning to the grasp pose would always fail because the specified pose/poses will always cause collisions between the object and the robot. For this reason, *MoveIt2* filters out collision objects from

the OctoMap such that there are no voxels surrounding objects whose identity is published as Collision Objects in the planning scene.

3.4.4 Insight from using OctoMap

It is also important for the OctoMap to not spawn stray voxels (Voxels near the Vention drag chains as shown in Figure 3.6) on any part of the robot detected by the camera as that will cause self-collisions with the robot, resulting in planning failure. For this reason, it is essential for the robot description (URDF) to accurately represent the collision geometry of the robot such that the collision geometry overlaps with and overshadows the Point Cloud to prevent the undesirable stray voxels on the robot. Secondly, stray voxels can also be a result of poor camera calibration, hindering the overlap of robot model and point cloud.

3.5 Simultaneous Trajectory Execution for two arms

3.5.1 Synchronous/14-DOF Trajectory Execution

This involves setting up all 7 joints of the left arm + all 7 joints from the right arm as a single move group. Hence, in the motion planning request message, we append 4 constraints, namely left position and left orientation specifying the constraint on left arm and right position, right orientation specifying the constraint on right arm. The resultant motion plan contains a single 14-DOF robot trajectory which is sent to the respective left and right arm controllers by the Trajectory Execution Manager. The two arms are guaranteed to not collide during execution because the planner considers the two kinematic chains as a single entity. Further, the planner also ensures that this single entity does not collide with itself, i.e., the left arm does not collide with the right arm. Hence, considering the two move groups as a single entity conveniently takes care of collision between the two robot arms. However, because the resultant motion plan contains a single trajectory for both arms, the entire trajectory is time stamped. The controller is also bound to respect these timestamps or else there comes the risk of the trajectory not being collision free. This implies that the collision

free attribute comes with the trade-off that the two arms must start and end executing their trajectories at the same time. This constraint of the two arms moving synchronously severely limits the number and quality of motion plans.

3.5.2 7-DOF simultaneous planning and execution with wall & without collision checking

This option enables the two arms to move simultaneously in an asynchronous manner, implying that the number of motion plans will be much more than the synchronous execution. The setup for this option is to directly publish the robot trajectory to the controller by bypassing *MoveIt2*'s trajectory execution manager. However, because we bypass Trajectory Execution Manager, the entity that prevents multiple trajectories from being executed simultaneously, we also lose collision checking. Hence, we introduce a separator/wall in our simulation and give it the properties of a collision object. Hence, all generated plans are guaranteed to be collision-free because the wall acts as a collision body. However, with this approach, we severely limit the valid search space of both robot arms.

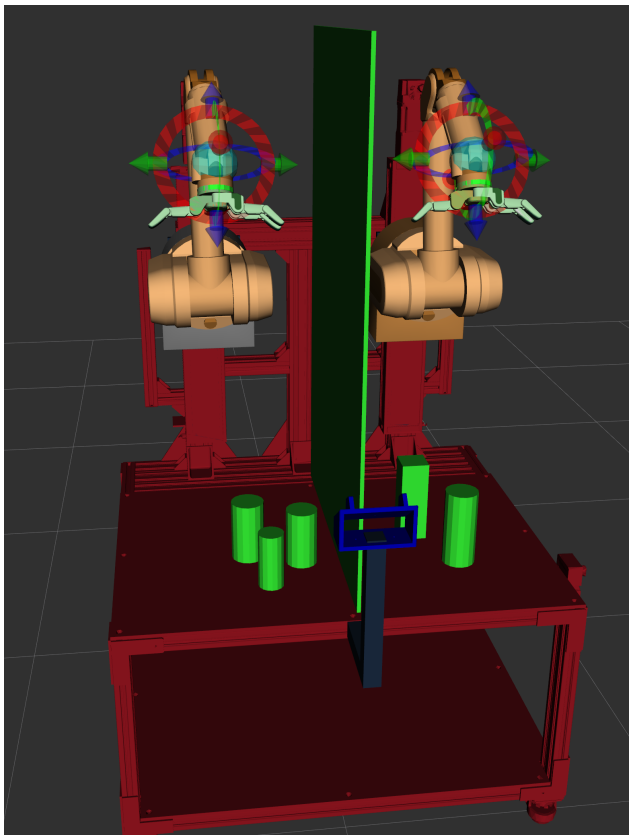

Figure 3.7: Separator acting as a collision wall between two arms.

3.5.3 7-DOF simultaneous planning and execution with collision checking

This approach enables asynchronous execution of 7-DOF arm trajectories while ensuring continuous collision-free operation. Unlike synchronous execution, it allows independent arm

motion by introducing a central execution scheduler that manages multiple trajectories concurrently. When a new trajectory is added, it undergoes real-time collision checking against all currently executing trajectories; if a collision is detected, it is queued until safe execution is possible. To optimize efficiency, collision checking is performed in a time-discretized manner, evaluating interactions at incremental time steps rather than reserving entire trajectory volumes. This extends *MoveIt2*'s capabilities by integrating online collision detection, where the combined state of all active trajectories is assessed dynamically. The MoveIt Flexible Collision Library (FCL) is leveraged for discrete checks, with future improvements aimed at incorporating continuous collision detection to prevent undetected collisions between time steps. This implementation is adapted from [16] with ongoing efforts focused on refining trajectory scheduling, optimizing planning scene updates, and evaluating performance in a simulated dual-arm pick-and-place scenario.

Chapter 4

CONCLUSION

This thesis details the development of a hardware interface for a bi-manual robotic system using the *ros2_control* framework, integrating motion planning capabilities with *MoveIt2* to tackle challenges related to system complexity, dynamic task execution, and grasping in cluttered environments. The approach builds on existing methods such as Task Space Regions and *IKFast* to refine conventional motion planning techniques while utilizing structured state machine execution to address control challenges, enhancing the system's scalability and adaptability for future advancements.

4.1 Key Contributions

- Developed a structured hardware interface for a bi-manual system consisting of two 7-DOF *Barrett WAM* arms, two *Barrett Hand*s, and Vention linear actuators.

- Demonstrated how state machines enable predictable and modular control, improving system reliability.

- Identified control jitter issues caused by CAN bus bandwidth limitations when controlling both the *WAM* arms and *Barrett Hand*s simultaneously.

- Developed a relay mechanism for controller switching, ensuring smooth motion execution while maintaining high-frequency control loops.

- Integrated multi-goal motion planning using Task Space Regions (TSRs) to improve grasp feasibility and robustness in constrained environments.

- Implemented three strategies for simultaneous trajectory execution of two robot arms, analyzing the trade-offs between the same.

4.2 Scope for future work

- Asynchronous Trajectory Execution with online collision detection has been partially integrated and needs a lot of careful examination and testing.

- Switch to nvblox: Replace OctoMap with nvbox to reduce planning latency and improve planning scene environment update rate.

- Decouple *IKFast* from *MoveIt2* – Make *IKFast* a standalone solver for direct, fast and deterministic inverse kinematics computations.

- Integrate Torque and Compliance Control – Implement computed torque control for the *WAM* arms either through *libbarrett* or *Mujoco* to enable dynamic and force-sensitive manipulation.

BIBLIOGRAPHY

[1] David T. Coleman, Ioan A. Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *JOURNAL OF SOFTWARE ENGINEERING IN ROBOTICS*, 2014.

[2] Dmitry Berenson, Siddhartha Srinivasa, and James Kuffner. Task space regions: A framework for pose-constrained manipulation planning. *The International Journal of Robotics Research*, 30(12):1435–1460, March 2011.

[3] Aaron Edsinger and Charles C. Kemp. Two arms are better than one: A behavior based control system for assistive bimanual manipulation. In *Recent Progress in Robotics: Viable Robotic Service to Human*, pages 345–355. Springer, 2007.

[4] Jan Steffen, Christof Elbrechter, Robert Haschke, and Helge Ritter. Bio-inspired motion strategies for a bimanual manipulation task. In *2010 IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 625–630, 2010.

[5] Héctor Herrero, José L. Outón, Unai Esnaola, Didier Sallé, and Koldo L. López de Ipiña. State machine based architecture to increase flexibility of dual-arm robot programming. In *Bioinspired Computation in Artificial Systems (LNCS vol. 9108)*, pages 98–106. Springer, 2015.

[6] Yuki Onishi and Mitsuji Sampei. Priority-based state machine synthesis that relaxes behavior design of multi-arm manipulators in dynamic environments. *Advanced Robotics*, 37(5):395–405, 2023.

[7] Dmitry Berenson, Siddhartha S. Srinivasa, and James J. Kuffner. Addressing pose uncertainty in manipulation planning using task space regions. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1419–1425, 2009.

[8] Dmitry Berenson, Joel Chestnutt, Siddhartha S. Srinivasa, James J. Kuffner, and Satoshi Kagami. Pose-constrained whole-body planning using task space region chains. In *2009 9th IEEE-RAS International Conference on Humanoid Robots*, pages 181–187, 2009.

[9] Yukiya Nakanishi, Masaaki Fukuoka, Shunichi Kasahara, and Maki Sugimoto. Synchronous and asynchronous manipulation switching of multiple robotic embodiment

using emg and eye gaze. In *Augmented Humans 2022*, AHs 2022, page 94–103. ACM, March 2022.

[10] Charles A. Meehan, Mark Roberts, and Laura M. Hiatt. Asynchronous motion planning and execution for a dual-arm robot. In *Proceedings of the ICAPS 2022 Workshop on Planning and Robotics (PlanRob)*, Singapore, 2022.

[11] Duke University Motion Lab. Robotic systems: Inverse kinematics, 2023. Accessed: March 13, 2025.

[12] Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

[13] Rosen Diankov and James J. Kuffner. Openrave: A planning architecture for autonomous robotics. 2008.

[14] Thomas Collins, J.J. Collins, and Donor Ryan. Occupancy grid mapping: An empirical evaluation. In *2007 Mediterranean Conference on Control amp; Automation*, page 1–6. IEEE, June 2007.

[15] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: an efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, February 2013.

[16] Pascal Stoop, Tharaka Ratnayake, and Giovanni Toffetti. A method for multi-robot asynchronous trajectory execution in moveit2, 2023.