Constraint-based Specification, Planning and Control for Mobile Manipulators in Dynamic Environments

Matthias Stüben

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Universität
Augsburg
University

Fakultät für Angewandte Informatik

18. März 2024

Constraint-based Specification, Planning and Control for Mobile Manipulators in Dynamic Environments

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Tag der mündlichen Prüfung:	27. Mai 2024

Abstract

Assistive robots have the potential to improve human lives significantly in many different forms. The possible applications are widespread, ranging from workplace assistance to healthcare, therapy, and household applications. Mobile manipulators, combining mobility and manipulation abilities, are an especially promising type of robot for these kinds of applications. Robots operating in unstructured, human-centered environments, as well as interacting with humans, have to be able to safely and robustly deal with a dynamically changing environment and sometimes unexpected motions by humans. Creating robot motions that respect and appropriately react to any possible change in the environment is a challenging research question. It begins with the specification of the requirements for robot motions for the respective task. Both planning and reactive control are required for robust robot motions. These two approaches are not always simple to combine: following a plan often conflicts with reacting to changes and a method of reconciling them is required.

This dissertation presents a method of specifying and generating robot motions for mobile manipulators based on geometric constraints. Constraints express whether the current robot position satisfies its requirements, and if not, how far off it is. Constraint rules are introduced, and are used to adapt the parameters of the constraints to changes in the environment. Constraint controllers are applied to generate reactive robot motions based on the current value of the constraints. Different types of constraints allow for different types of reactions. The concept of an action brings together multiple constraint specifications, in which priorities, tolerances, and weights are used to express the relative importance of the requirements. Besides reactive control, a method to create and safely execute motion plans for action specifications is presented. Furthermore, a method for automatically detecting when planning is required is described.

In order to demonstrate the capabilities of the presented approach, it is evaluated on two different case studies. The first consists of an object handover between robot and human, the second considers a robot shining a flashlight to provide light to a human. Both case studies are evaluated in a successful case and different failure scenarios to assess the robustness of the approach.

Kurzfassung

Assistenzroboter bieten das Potenzial, das menschliche Leben auf vielfältige Art und Weise zu verbessern. Die möglichen Anwendungen sind dabei vielfältig. Unterstützung am Arbeitsplatz, Pflege und Therapie sowie Unterstützung im Haushalt stellen nur ein Teil der Möglichkeit dar. Mobile Manipulatoren, die Mobilität mit der Fähigkeit zur Manipulation vereinen, sind für viele derartige Anwendungen der vielversprechendste Robotertyp.

Solche Anwendungen erfordern meist, dass die Roboter in unstrukturierten und Mensch-zentrierten Umgebungen eingesetzt werden und dabei auch direkt mit Menschen interagieren. Daher müssen sie in der Lage sein, mit unvorhergesehenen Änderungen in der Umgebung sowie verschiedensten Bewegungen des Menschen auf sichere und robuste Art umzugehen. Es ist eine herausfordernde Forschungsfrage, wie Roboterbewegungen, die auf alle möglichen Umgebungsänderungen angemessen und sicher reagieren, generiert werden können. Die Fragestellungen beginnen bereits bei der Spezifikation der Anforderungen an die Bewegung. Des Weiteren müssen sowohl Planung als auch Reaktion berücksichtigt werden, wobei diese Ansätze sich oftmals widersprechen. Daher muss eine Methode gefunden werden, wie sie kombiniert werden können.

In dieser Dissertation wird eine Methode vorgestellt, mittels derer Roboterbewegungen für mobile Manipulatoren spezifiziert und generiert werden können. Sie basiert auf geometrischen Constraints, die ausdrücken, ob die aktuelle Roboterposition den Anforderungen entspricht, und falls nicht, den Grad der Abweichung angeben. Sogenannte *Constraint Rules* werden benutzt, um die Parameter der Constraints an Änderungen in der Umgebung anzupassen. *Constraint Controller* setzen die aktuellen Werte der Constraints in reaktiven Bewegungen um. Verschiedene Arten von Controllern können für verschiedene Reaktionen eingesetzt werden. Im Konzept der *Action* werden mehrere Constraint-Spezifikationen zusammengeführt, wobei diese mit Prioritäten, Gewichten und Toleranzwerten parametriert werden, um die relative Wichtigkeit des Constraints zu erfassen. Die Spezifikation einer Action kann sowohl für reaktive als auch geplante Bewegungen benutzt werden. Eine Methode, um geplante Bewegungen auch in dynamischer Umgebung sicher auszuführen wird vorgestellt. Auch ein Verfahren, um automatisch zu erkennen, wann Planung nötig ist, wird präsentiert.

Um die Leistungsfähigkeit des vorgestellten Ansatzes zu demonstrieren, wird er anhand zweier Fallstudien evaluiert. Die erste Fallstudie besteht aus einer Objektübergabe zwischen Roboter und Mensch. Die Zweite betrachtet einen Roboter, der mittels einer Taschenlampe den Arbeitsbereich eines Menschen beleuchtet. Beide Fallstudien werden sowohl im Erfolgsfall als auch in verschiedenen Störfällen evaluiert, um die Robustheit des Ansatzes bewerten zu können.

Acknowledgments

This thesis would not have been possible without the support I received from different sides. First, I want to give special thanks to my advisor, Professor Dr. Wolfgang Reif. He provided me with the resources and guidance that I needed for this work, while also giving me enough freedom to work on the topics of my choice and develop my own ideas.

I would like to thank all the researchers and students that I worked with during the many years of my employment at this chair. Especially the members of the robotics group, namely Dr. Alexander Poeppel, Dr. Julian Hanke, Dr. Christian Eymüller, Dr. Constantin Wanninger, Martin Schörner, Daniel Bermuth and Michael Filipenko deserve thanks for the many discussions and the hands-on support they provided for the various hardware issues of the robot. Thanks go to my former colleague Dr. Alwin Hoffmann for introducing me to the field of robotics as a student assistant and later on guiding my initial path of the work on this thesis. I am grateful to our Technician Stefan Wolff, for the tremendous practical support in the realization of the robot system.

I would like to thank my family for their unwavering support and understanding throughout my studies, during easier and difficult times. Lastly, I want to thank my partner Anna for her love and encouragement during the often challenging times of writing this dissertation.

Matthias Stüben

Contents

1 Overview and Motivation **1**
1.1 Motivation and Goals . . . 1
1.2 Main Contributions and Thesis Outline . . . 3

2 Preliminaries **5**
2.1 Basic Concepts of Robotics . . . 5
2.1.1 Representations in Cartesian Space . . . 5
2.1.2 Kinematic Modeling of Robots . . . 6
2.1.3 Mobile and Redundant Robots . . . 7
2.1.4 Redundancy . . . 8
2.2 Safety Aspects in Human-Robot Interaction . . . 9
2.3 Behavior Modeling . . . 10
2.3.1 Behavior Trees . . . 10
2.3.2 State Machines . . . 11
2.3.3 Terminology Overview . . . 11

3 Description of the Case Studies and the Robot Platform **13**
3.1 The Case Studies . . . 13
3.1.1 Case Study A: Object Handover . . . 14
3.1.2 Case Study B: Following and Lighting . . . 16
3.2 Description of the Robot Platform . . . 17
3.2.1 Actuators . . . 17
3.2.2 Sensors . . . 19
3.2.3 Computing Hardware . . . 20
3.3 Simulation . . . 20

4 Perception and World Modeling **23**
4.1 Image Processing . . . 24
4.1.1 3D Obstacle Map . . . 25
4.1.2 Monitoring of the Surroundings . . . 27
4.1.3 People Detection and Pose Estimation . . . 27
4.2 Environment Model and Robot Model . . . 28
4.3 External Torque Estimation from Motor Torques . . . 30
4.3.1 Related Work . . . 30
4.3.2 Model-based methods . . . 31
4.3.3 Learning-based methods . . . 31
4.3.4 Model-based Torque Estimation . . . 32
4.3.5 External Torque Estimation with LSTM Networks . . . 37
4.3.6 Validation . . . 39
4.3.7 Evaluation . . . 40

5 Constraint-based Robot Control **47**
5.1 Related Work . . . 48
5.2 Definition of Constraints . . . 50
5.3 Handling Dependencies on the Environment: Constraint Rules and Tasks . . . 54
5.3.1 Inputs to Constraint Rules . . . 55
5.3.2 Types of Constraint Rules . . . 57
5.4 From Constraints to Velocity Bounds: Constraint Controllers . . . 67
5.4.1 The `Follow`-Controller . . . 68
5.4.2 The `Limit`-Controller . . . 69
5.4.3 The `Stopping`-Controller . . . 71
5.4.4 The `Hybrid`-Controller . . . 72
5.5 Tasks and Actions . . . 73
5.5.1 Specification of Tasks . . . 73
5.5.2 Definition of Actions . . . 75
5.6 Finding an Optimal Control Signal . . . 80
5.6.1 Formulation of a Quadratic Optimization Problem . . . 80
5.7 Evaluation . . . 85
5.7.1 Linear Motions to Cartesian Targets . . . 85
5.7.2 Null-space Motion . . . 86
5.7.3 Reaction to Obstacles . . . 88
5.7.4 Real-time Requirements . . . 94

6 Robotic Path Planning with Constraints **97**
6.1 Related Work . . . 98
6.1.1 Basics of Path Planning . . . 98
6.1.2 Multi-level Planning . . . 101
6.1.3 Planning with Constraints . . . 103
6.1.4 Planning in Dynamic Environments . . . 109
6.2 Planning Pipeline . . . 110
6.3 Handling Path Constraints . . . 111
6.3.1 Path constraints in multi-level planning . . . 112
6.4 Finding Goals from Constraints . . . 113
6.5 Initial configuration . . . 114
6.6 Planning with Soft Constraints . . . 116
6.7 Evaluation and Results . . . 117
6.7.1 Cartesian Goal with Obstacle . . . 119
6.7.2 Narrow Passage . . . 122
6.7.3 Making room . . . 126
6.7.4 Conclusions . . . 128

7 Connecting Planning and Reactive Control **131**
7.1 Execution of Plans in Dynamic Environments . . . 131
7.1.1 State of the Art . . . 132
7.1.2 Execution of plans in reactive control . . . 133

7.2 Execution Modes . . . 134
7.2.1 Reactive Action Execution . . . 134
7.2.2 Planned Action Execution . . . 135
7.2.3 Autoplanning Action Execution . . . 137
7.2.4 Evaluation . . . 138
7.3 Combining Actions to Behaviors . . . 138

8 Implementation and Software Architecture 139
8.1 Integration into the ROS2 environment . . . 139
8.1.1 Defining actions . . . 141
8.1.2 Defining Behaviors . . . 144
8.2 Extending the Framework . . . 145
8.2.1 Adding new Types of Rules, Controllers, Inputs, Solvers . . . 145
8.2.2 Integration of a new robot . . . 145

9 Realization and Evaluation of the Case Studies 147
9.1 Realization . . . 147
9.1.1 Description of the graphical notation of FlexBE . . . 147
9.1.2 Case Study A: Object Handover . . . 148
9.1.3 Case Study B: Following and Lighting . . . 149
9.2 Evaluation . . . 151
9.2.1 Case Study A: Object Handover . . . 151
9.2.2 Case Study B: Following and Lighting . . . 161

10 Conclusion and Outlook 167
10.1 Summary of Research Contributions and Evaluation Results . . . 167
10.2 Open Research Challenges and Future Directions . . . 168

Bibliography 171

Own Publications 185

List of Figures 185

List of Tables 189

Listings 191

List of Algorithms 191

1 Overview and Motivation

This chapter provides the motivation and scope of this thesis. The research area and its context, its current limitations and goals are briefly presented. Furthermore, the structure of this thesis and the main scientific contributions are introduced.

1.1 Motivation and Goals

Ever since the idea of robots has been created, humans have dreamed of living naturally alongside robots in their daily lives and in their homes and workplaces. Whether it is in their early conceptions in the works of Karel Čapek and Isaac Asimov, or in series popular to this day such as George Lucas' Star Wars, robots are most often envisioned as a technology that integrates seamlessly and effortlessly into the living environment of humans.

The reality of robot use in modern practice is often very different, with robots mostly being used as industrial robots strictly separated from humans. Some statistics are given below to give an overview of the current state of robotics. They are taken from the Artificial Intelligence Index Report 2023 created by Stanford University [93]. In industrial applications, only 2.8% of newly installed robots were intended for collaboration with humans in 2017. By the year 2021, this number has increased to 7.5%, which shows a strongly increasing interest in collaborative robots, but also that they are still vastly outnumbered by robots without the ability for collaboration.

In the area of service robotics, the data similarly shows that their use is sharply increasing, although overall still low. Between 2020 and 2021, the number of professional service robots installed in the world in the application area of hospitality almost doubled from 11,000 to 20,000. In the same time, the number of installed service robots for transportation and logistics has increased by about 150% from 34,000 to 50,000. While this signifies a strong trend, the absolute numbers are small compared to the 517,000 industrial robots that are reported to have been installed in 2021. These numbers show that robots working together and interacting with humans are not just science fiction, but are increasingly becoming a reality for practical applications. However, their adoption is still lagging far behind industrial robots, and many open research questions hinder their use. One of the many problems faced is the safety and robustness of the robots in unpredictable and dynamic environments. This is especially true for mobile robots, which have no fixed environment.

While industrial robots are relatively easy to program in their static environments with clearly defined processes, this is much more difficult for assistive robots in environments shared with humans. Environmental changes such as moved furniture, pets walking around or other obstacles have to be accounted for. In the interaction with a person, the robot motion can not be fully planned in advance, but the robot has to react and adapt to the behavior of the human. Whether the human behaves as predicted or not, the robot has to prevent causing damage to its environment and itself. Furthermore, the robot should be efficient and pleasant for the humans in its environment. A robot that is safe, but not able to fulfill its tasks, or is more bothersome than useful to its human users, has no practical value. Therefore, the requirements for robot behaviors in interaction with humans are complex and not easy to realize.

In the industrial context, robot motions are usually programmed by explicitly specifying the motions that the robot has to perform. The motions are specified as joint positions or as poses in Cartesian space. This type of programming is clearly not sufficient for interactive robots in dynamic environments. Instead of simply performing a series of static motions, the tasks in human-robot interaction are more abstract. An object handover between a person and a robot, for example, can not be specified as a sequence of static motions. The target positions depend on the environment, such as the pose of the human engaged in an interaction, and can not be known in advance. The robot has to continuously adjust to the environment, in what is called *reactive control*. At the same time, many different requirements on the motion have to be respected: The robot should move its gripper towards the object it is handed, but also make sure not to collide with any obstacles or harm the human. Optional requirements, that make the process more pleasant for the human, should also be respected, as long as they do not disturb the other satisfaction of the more important requirements.

At the same time, pure reactive control is at risk of creating robot behaviors that are too passive and can easily get stuck. The robot should not only react to its environment, but also actively seek to fulfill its task. Therefore, *motion planning*, which allows the robot to plan ahead of the current state, is also required. As *following a plan* and *reacting to the environment* are often contradictory, some method of reconciling these two approaches is required.

To summarize, the generation of robot motions in dynamic environments for the purpose of human-robot-interaction requires the use of reactive control reacting to its environment, motion planning to enable proactive behaviors, and a way to connect the two approaches. As the requirements on the robot motions in the various unpredictable situations that can occur are complex in themselves, a suitable method of specifying them is required as well.

1.2 Main Contributions and Thesis Outline

This work presents a method of specifying complex robot behaviors for mobile manipulators in dynamic environments. The intended use cases are in the field of human-robot interaction. Methods of using the same specifications for reactive control as well as motion planning are presented. The paths generated by the planner are executed safely by monitoring safety-critical requirements during the executions. Furthermore, an approach for automatically detecting *when* planning is required is presented. The specification is using modular building blocks, which can be recombined in various ways to flexibly form a wide range of robot behaviors. If the present building blocks do not suffice, the framework is also easy to extend through the use of a plugin mechanism. The specified robot motions have a simple interface, making it easy to combine them to complex robot behaviors, for example by using state machines.

The method is evaluated in practical experiments on a mobile manipulator. Two case studies are considered. The first is an object handover between a person and the robot. The second case study presents an application in which the robot is continuously following a person and providing light using a flashlight. While the robot is following the person, it has to ensure that it is not getting stuck on the obstacles and not getting in the way of the human, while still being close enough to provide light.

This thesis begins with an overview of the basic concepts of robotics and related technologies that are required for the understanding of this work. Afterward, the case studies considered in this work and the robot platform used in their realization are described in chapter 3. The robot's sensors and the processing of their data to a coherent world model are presented in chapter 4. The following chapter 5 introduces a method of specifying and generating reactively controlled robot motions based on a constraint specification. The same method of specification is then used in chapter 6 to formulate motion planning problems. Different solvers are evaluated in their performance for typical problems. Chapter 7 connects these two approaches, and presents different execution modes for given specifications which combine planning and reactive control in different ways. The implementation and practical realization of the case studies are presented in chapter 8, before the evaluation results are presented in chapter 9. Chapter 10 concludes the thesis with a summary and presents possible future directions of research.

2
Preliminaries

In this chapter, the most important theoretical foundations of robotics are briefly explained and their challenges regarding the work of this thesis are outlined. The relevant terminology and notations are introduced. For further information, the reader is referred to the Springer Handbook of Robotics [132].

2.1 Basic Concepts of Robotics

Most robots are systems of rigid bodies which are connected by joints. The individual rigid bodies are called *links*. The links have a *position* and *orientation* in Cartesian space. Together, they are called the *pose* of the link. The joints connecting the links can be either actively controlled joints or passive joints. Passive joints do not occur in our case studies and are not considered further in this work. Throughout this work, we refer to the number of controlled joints as n. In general, n is a positive natural number, $n \in \mathbb{N} \setminus \{0\}$. Different topologies in which the links are connected are possible. The most common is the serial kinematic chain, where each link except the first and last are connected to exactly two other links. The first and last are connected to exactly one other link. The robot used in the case studies is modeled as a kinematic chain. The approaches presented in this work are also applicable to the more generic *kinematic tree*, in which the structure of links and joints describes a tree. Other topologies, such as parallel mechanisms, are possible but not considered in this work.

2.1.1 Representations in Cartesian Space

Representations of poses of various bodies in space are a fundamental part of robotics. Poses can be described by various representations, with different advantages. Only a brief overview of the approaches used in this work is given here. The representation of a pose requires at least six parameters. Many representations however use more parameters, which are not fully independent.

A basic concept in the description of poses is that of *coordinate reference frames* or just *frames* for short. A frame consists of an origin and three orthogonal basis vectors, typically called (x, y, z). The basis vectors are also called the *axis vectors*. Poses are always expressed relative to a frame. The pose of a link can also be expressed as a frame relative to another frame.

The position of a frame relative to another can thus be denoted by a three-element vector. The components of this vector correspond to the displacement along the direction of the basis vectors of the reference coordinate system:

$$p = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

For the representation of rotations, various different conventions are in use. Only the two most important representations for this work are quickly described here: *rotation matrices* and *rotation vectors*.

A rotation matrix is a matrix of size 3×3. Conventionally, rotation matrices are called R. The components of R are the dot products of the basis vectors of reference and target frame. While a representation of a rotation only requires three independent parameters, a rotation matrix contains nine. Rotation matrices are therefore not a minimal representation.

The second type of representation used in this work is the rotation vector. Having only three parameters, this representation is minimal. It is based on the observation that an orientation can be described as a single angle θ in combination with a unit vector w, where w describes the axis of rotations. This representation is called the *axis-angle* representation. To go from this representation to the minimal rotation vector, the angle θ is expressed as the length of the vector w. In this way, the rotation can be expressed as a single three-element vector, called the rotation vector.

In this work, poses are assumed to be relative to a global, fixed origin frame, unless noted otherwise.

2.1.2 Kinematic Modeling of Robots

The robot considered in this work can be described as a kinematic chain with nine degrees of freedom. Six degrees of freedom correspond to the joints of the arm, three correspond to the position and orientation of the base on the horizontal plane. The modeling of the robot is described in more detail in the following chapter, where the robot system is presented in detail.

A vector containing the position of each joint of a robot is called a robot *configuration*, and commonly called $\mathbf{q} \in \mathbb{R}^n$. In the case of the mobile manipulator used in the experiments, q is thus a nine-dimensional vector. The components of the vector are indexed as follows: $\mathbf{q} = (\mathbf{q}_x, \mathbf{q}_y, \mathbf{q}_\theta, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{q}_4, \mathbf{q}_5, \mathbf{q}_6)$. $\mathbf{q}_x$ and $\mathbf{q}_y$ refer to the position of the base in the plane, $\mathbf{q}_\theta$ is the orientation (yaw) of the base. The remaining components of $\mathbf{q}$ refer to the position of the six arm joints. The vectors $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ stand for the velocities and accelerations of the joints. Their components are used with indices analogously to $\mathbf{q}$.

Finding the pose of robot frames, given a configuration, is the problem of *forward* or *direct* kinematics. For kinematic chains, this can be solved by calculating the transformation between the links of the robot and concatenating them. *Inverse* kinematics is the problem of determining configurations that achieve a target pose of a link. In contrast to *forward* kinematics, this problem does not have a unique solution. For robots with more than six joints, the number of solutions is generally infinite, and some method of choosing an optimal solution is required.

The velocity of a robot link in Cartesian space is found by what is called *instantaneous* or *velocity* kinematics. The required inputs are the current configuration $\mathbf{q}$ and the joint velocities $\dot{\mathbf{q}}$. The velocity v of the referenced link can then be found from the following equation:

$$v = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}$$

The matrix $\mathbf{J}(\mathbf{q}) \in \mathbb{R}^{6\times n}$ depends on the current configuration $\mathbf{q}$ and is called the *Jacobian matrix*, or just *Jacobian* for short, of the robot link. The calculation of $\mathbf{J}$ is generally easy given the kinematic structure of the robot [132], and can be assumed to be known for any robot link.

The problem of *inverse instantaneous* or *inverse velocity* kinematics is described by the following equation:

$$\dot{\mathbf{q}} = \mathbf{J}^{-1}(\mathbf{q})v$$

It determines the required joint velocities $\dot{\mathbf{q}}$ to achieve a given link velocity v in the current configuration $\mathbf{q}$. However, the above equation is only valid for six-joint robots, as $\mathbf{J}$ is otherwise not a square matrix and cannot be inverted. For robots with more joints, the pseudo-inverse of the Jacobian can be used instead. It can be computed by the following equation:

$$\mathbf{J}^{\dagger} = \mathbf{J}^{T}(\mathbf{J}\mathbf{J}^{T})^{-1}$$

Using the pseudoinverse of the Jacobian for inverse velocity kinematics will calculate the minimum-norm joint velocities that achieve the target link velocity v.

2.1.3 Mobile and Redundant Robots

The robot used in the experiments is a mobile manipulator, consisting of a manipulator arm mounted on a mobile base. The fact that the manipulator arm is extended by the mobile base makes the robot both mobile and redundant. This section introduces the most important foundations for these types of robots.

Holonomic and omnidirectional mobile robots

An important property in the control of wheeled mobile robots is whether the robots are *holonomic* [132]. In practical terms, a mobile robot is holonomic if it can move and rotate in any direction on the horizontal plane from any configuration. Robots with a two-wheel differential drive or a car-like Ackerman steering geometry are thus not holonomic, because they can not move sideways directly, without changing their configuration first.

Mathematically, a holonomic robot has only kinematic constraints that can be expressed as explicit functions of position variables. In contrast, non-holonomic robots are subject to kinematic constraints that depend on the derivatives of the position variables. This definition is equivalent to the following: a robot is holonomic if and only if the amount of *degrees of freedom* of the workspace is equal to the amount of *differential degrees of freedom* of the robot [134]. The number of *differential degrees of freedom* is equal to the number of independently achievable velocities.

For a mobile robot moving on a horizontal plane, the workspace has three degrees of freedom: (x, y, θ). A car-like robot has only two *differential degrees of freedom*, corresponding to the linear velocity and the steering angle. Therefore, this type of robot is not holonomic. While such a robot can achieve each pose (x, y, θ) (possibly with complex maneuvering), it can not follow each path. This has to considered in path planning and motion generation for these types of robots and limits their ability to move freely.

A mobile robot on the horizontal plane thus needs three differential degrees of freedom to be holonomic. This specific type of robot is called *omnidirectional robot*. The ability to react to unforeseen scenarios is highly important for interactive scenarios. Therefore, we use an omnidirectional mobile robot as the base of the controlled system in this work.

2.1.4 Redundancy

Kinematically redundant robots have more joints than are strictly required for their task. In a strict sense, it can not be said that a robot is inherently redundant, but only redundant with respect to a given task. Typically, the task of a robot is assumed to be the motion of the end-effector in space, which requires six degrees of freedom. In practice, robots with more than six joints are therefore usually called redundant.

The remaining degrees of freedom can be exploited for various purposes, such as avoiding singularities, joint limits, or obstacles. Redundant robots are able to execute motions that do not disturb the main task, i. e. typically the pose of the end-effector. These motions are called *self-motions*, *internal motions*, or *nullspace motions*. The presence of redundancy means that additional freedom exists in the solution of inverse kinematics and inverse velocity kinematics. Usually, some method of optimization of given metrics is used to find optimal ways to use this

freedom. In general, redundancy resolution thus requires a formulation of different tasks with different importance. A similar approach is used in the concepts described in this thesis. However, we mostly do not use the term *redundancy*, as it inherently assumes a hierarchy of a *main task* and *secondary* tasks. Instead, a more general formulation of task importance is used, that generalizes these concepts.

2.2 Safety Aspects in Human-Robot Interaction

The topic of safety is most important in human-robot interaction. While the topic of safety is very broad, the most relevant consideration in robotics is the collision or unwanted force exertion between robot and human [132].

The physical safety of the human interaction partner is the most important aspect, but the safety of the robot and any handled objects are also considered. Since human-robot interaction by definition considers applications where robots and humans interact, safety is much more difficult to guarantee compared to industrial applications where robots are completely separated from any human, at least during their operation. The methods of achieving safety in human-robot interaction can be either hardware design, software design, or a combination of both [24]. An example of safety achieved through hardware design are robots that are inherently soft and light, and thus unable to cause serious harm to a person. These types of robots are however also limited in their ability to manipulate their environment, which naturally requires the ability to produce larger forces. This type of intrinsic safety is thus feasible for all robots, and usually a trade-off between safety and performance has to be made [50].

However, even if the notion of safety is reduced to the aspect of collision between human and robot, safety depends on many different factors. They range from software dependability and mechanical failures to human errors [2]. To fully ensure a safe system, a thorough hazard analysis has to be performed that takes all these factors into account. The ISO standard **ISO/TS 15066:2016** [27] has been developed to give specifications for the safety requirements for collaborative industrial robot systems, and proposes measures to reduce the risk of human injury. Different collaboration modes and their requirements are defined.

Several times in this thesis, we refer to *safe* robot motions. In these cases, we do not refer to safety in the sense of creating a standard-conforming, verified safe system. The required aspects lie outside the scope of this work. Safety is used only in the sense as it applies to motion planning and control. We thus refer to *safe* motions and commands if, given the currently available information, the robot does not actively cause a collision.

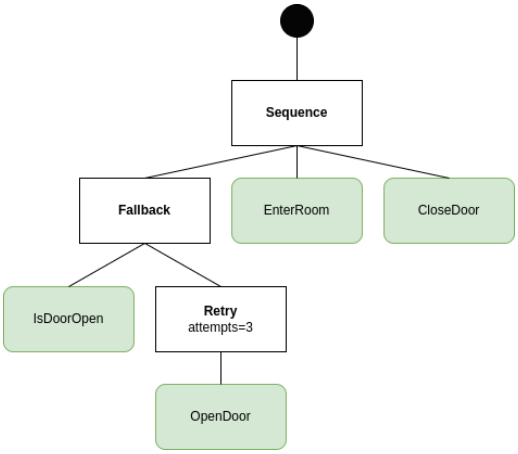


Figure 2.1. Example of a behavior tree.

2.3 Behavior Modeling

Even after individual robot motions have been created, a specification of when which motion is executed is required. Various methods of building behaviors from a finite set of building blocks have been developed, from the field of robotics as well as other fields. This section briefly describes the two most popular methods: behavior trees and state machines.

2.3.1 Behavior Trees

Behavior trees have originally been developed for video games, but have since been extensively studied and used for robotics as well [21]. Their main advantage is claimed as handling complexity better than traditional state machines. Formally, they can be shown to generalize state machines.

A behavior tree's leaf nodes represent individual operations. Internal nodes represent different compositions of the underlying operations. In the execution, an activation signal called a *tick* is sent to its children with configured frequency. Whenever a node receives a tick, it is executed and returns to its parent a status which is one of *Success, Running, or Failure*. The leaves are usually *Action* nodes, which perform their corresponding action as long as they receive ticks regularly. The semantics of the execution of a behavior tree are defined according to the types of composition nodes that are used. Standard composition nodes such as *Sequence*, *Condition*, *Parallel*, and *Fallback* have different methods of sending ticks to their children, and thus different execution schemes can be realized. Other types of composition nodes can also be defined to extend the capabilities.

In figure 2.1 a simple example behavior tree is shown. Green nodes show individual actions, while white nodes are composition nodes. The modeled behavior lets the agent open a door, if it is not already open, then enter the room and close the door afterward. It will attempt to open the door at most three times.

Despite their advantages, behavior trees have also been criticized in comparison to state machines. One main criticism is their lack of readability. While the possible execution paths of state machines are fairly easy to understand from a human perspective, by following possible paths through its states, the condition-based syntax of behavior trees can be harder to read. The fact that *ticks* start from the root in every cycle means that paths through the tree have to be repeated often to understand the resulting behaviors. Another problem of behavior trees is that of concurrent execution and handling of orthogonal physical subsystems that are ubiquitous in robotics [20].

2.3.2 State Machines

State machines are the established and traditional model of behavior modeling in robotics, and a plethora of other applications. Different variants exist, including hierarchical state machines. The most common variant might be the state machine diagram of the UML standard [122]. Their simplicity and easy-to-understand control mechanism make them an attractive tool. As they grow in size, they can however be hard to maintain. Given their extremely widespread use and the wealth of existing literature, we will not give a further introduction to state machines here.

Recent research on the use of state machines and behavior trees in open-source robotics projects has found that state machines are far more popular at the moment. [46]. Behavior trees are a powerful control architecture that generalizes multiple other approaches, including state machines. However, their advantages mostly come into play when the complexity of state machines becomes too high to handle. On the other hand, state machines benefit from their simplicity and ease of use, as well as their extensive support in theory and practical tooling. Since the high-level control in our use cases is relatively simple, we decided to model them as state machines. In particular, we chose to use the implementation FlexBE [124] for its visualization capabilities, ROS2 integration, and easy extensibility.

2.3.3 Terminology Overview

Organizational concepts for the specification of robot motions are notoriously hard to name precisely and concisely. In existing literature, terms are often used with vastly different meanings by different authors and in different contexts. For example, the term *task* is routinely used to refer to either a discrete high-level action, or to distinguish control in Cartesian space from joint-level control. This continues in composite terms, so that for example *task planning* [42] and *task-constrained planning* [135], both commonly used terms in the field of robotics, might seem closely related at first glance. However, they use entirely different interpretations of the word *task* and there is very little overlap in these two research areas. Many other terms, including *behavior*, *action*, *capability*, *skill*, and *activity*, are similarly overloaded.

The present work includes concepts on many different levels of abstraction in robot control. We try to keep our terminology consistent with existing literature and implementations as far as possible. Since the existing literature is in itself not consistent, some deviation can however not be avoided.

In general, we use the following terms with the meanings given below:

- **Action**: A specification of a robot motion that can not be subdivided into discrete states. Given a *goal*, actions execute the corresponding robot motion until they either *succeed*, *fail*, or get *canceled*. During the execution, they provide feedback about the current state to the caller. Planned Actions are an exception, as they contain discrete state changes as a planning phase and a plan execution phase. It is however not part of the specification of a planned action to specify a sequence of discrete states. We use this term to be consistent with the conventions of the Robot Operating System [89]. Examples of actions are moving the platform to a target position, following a person, or moving the gripper to a target pose.
- **Task**: Elements of the definition of an action that define requirements on the robot's motion. Examples include the target end-effector pose, joint limits, or avoidance of dynamic obstacles. Internally, tasks correspond to constraints along with a definition of how the constraint is controlled, its dependency on the environment, priorities, and other required information. The name is chosen as it generalizes tasks as they are used, for example in redundancy resolution, where the task of the robot usually refers to its end-effector pose.
- **Behavior**: A specification that defines the sequence in which individual actions are performed, depending on the results of the actions and outside information. Common tools to define behaviors are behavior trees or state machines. The term is used in this way to be consistent with software tools for *behavior modeling* and related publications, e.g. [124]. An example to would be specified as a behavior is a complete object handover, that consists of multiple individual actions with discrete changes between them, such as navigating to the person, moving the gripper to the handover pose, grasping the object, et cetera.

Detailed descriptions of these concepts follow in the remainder of this work. Here, the descriptions are given to clarify the intuitive meanings of the terminology for the reader in the context of this work.

Summary. The ability of the presented approach to create robust and reactive motions in dynamic environments is evaluated in two case studies. The first is an object handover between robot and person. Both unexpected motions by the person and obstacles in the environment have to be considered. The second case study consists of the robot shining a flashlight to provide light to a human walking around a factory setting, while avoiding obstacles and staying out of the way of the person. Both case studies are realized using a mobile manipulator with a six-joint arm and a holonomic base.

3

Description of the Case Studies and the Robot Platform

3.1 The Case Studies **13**
3.1.1 Case Study A: Object Handover 14
3.1.2 Case Study B: Following and Lighting 16
3.2 Description of the Robot Platform **17**
3.2.1 Actuators 17
3.2.2 Sensors 19
3.2.3 Computing Hardware 20
3.3 Simulation **20**

The approaches described in this work are evaluated on two types of case studies. These two case studies as well as the mobile manipulator and related equipment used in them are presented in this chapter.

3.1 The Case Studies

Two types of case studies are considered in this work. They are intended to illustrate the applicability of the approaches to realistic scenarios. The first case study consists of an object handover between a person and the mobile manipulator. This task occurs in many different applications of assistance robotics. Examples include mobile robots supporting shop floor workers by bringing and returning tools as they are needed. Another example lies in healthcare and household applications, where robots can prove useful by delivering various objects, such as medicine and water or by helping people with limited mobility to gain access to hard-to-reach objects, such as objects that fell to the floor.

The second case study serves as an example of a long-running robot behavior that is continuously adapting to the state of the environment. The chosen case study consists of the mobile manipulator shining a light from a flashlight to provide light

for a person, for example, a person working in a mechanical workshop where the worker needs to move around to different parts of the room. Different application examples that are very similar from a robot control perspective also exist. By exchanging the flashlight with a camera, and the lighting application with various monitoring and inspection tasks, the same control definition can be used for entirely different applications.

3.1.1 Case Study A: Object Handover

The object handover is one of the most essential capabilities in service robotics. It is a complex skill, consisting of multiple phases and highly dependent on factors such as the type of object, expectations of the interaction partner, and the available sensors.

In the strictest sense, it can be said that the (physical) handover only begins once both giver and receiver are in contact with the object, and ends as soon as only the receiver is holding the object [104]. During this phase, no large robot motions occur typically. Instead, the main topics of interest during this phase are the small-scale reactions to the interactions partners hand, using techniques such as force feedback. Perhaps the most important decisions during this phase are determining when it is safe to release the object, when it is safe to close the grasp without hurting the interaction partner, and when an object is safely held. Tactile and force sensing are core techniques in this field. Much of the research focus lies on accurate sensing of interaction forces and the handled objects, as well as designing controllers based in this information [18, 74, 29, 23].

However, other phases of a handover process are just as important, as both pre-handover and post-handover phases are critical to the success of the process. One important aspect of human-robot object handovers is *communication*: to achieve coordinated and successful object handovers, the robot needs to be aware of the human's intention. In human-to-human handovers, indicators such as speech, gaze, and body movements are used to coordinate the handovers. Researchers are trying to understand human-to-human handovers in order to apply the results to robots [97, 136].

Other research questions related to the field include grasp planning, which is concerned with questions such as how an object being received from a person should be grasped, or determining the optimal way to grasp an object in order to then hand it over to a person. The related works take into account how a person can and wants to grasp an object considering the physical limitations of human hands, as well as what the person might want to use the object for [17, 73, 4].

The sub-field that is most relevant for the present work is motion planning and control for robot handovers. This field deals with the questions of how the robot should move before, during, and after a handover. Requirements for robot motions for object handovers can be categorized into two categories [104]:

1. **Legibility and Predictability**: These are two related, but separate characteristics that describe how easy to understand and conforming to expectations the robot's motions are from a psychological standpoint. Specifically, *legibility* enables a person to infer the goal of an action by observing the motion. On the other hand, *predictability* means that the robot's motions conform to the expectations of a person who knows the goal of the action.
2. **Robustness, Reactivity, and Context Awareness**: These characteristics describe the robot's ability to accommodate changes in the environment, and to accommodate the behaviors of different persons. The realization of the characteristics require both planning and the consideration of feedback about the current state of the environment.

In the present work, the focus lies on the latter category. Motion planning and reactive control are used in conjunction with constraint specifications to achieve optimal and safe robot motions in dynamic environments. The former category is considered to a lesser degree. While the described approach enables the implementation of these characteristics through different constraints, costs and other specification methods, the question of what constitutes legible and predictable motions can only be answered through psychological research and studies, which lies outside the scope of this work.

Object handovers in both directions, from robot to person and from person to robot, are considered. The focus of this work lies on the motions leading up to the point where both participants are in contact with the object. Therefore, the differences in the handover directions are small, and they are considered as a single case study. For the sake of simplicity, the same flashlight that is also used for case study B is used as the object that is handed over.

At the beginning of the scenario, it is assumed that the robot is situated at a waiting position, not in the immediate surroundings of the person. Therefore, the robot has to navigate to the person in the first step. Successful execution of the object handover from a mobile manipulator consists of the following steps:

1. Navigate to the proximity of the person.
2. Approach the person
3. Move to a suitable handover position and perform the handover
4. Retreat
5. Navigate back to the waiting position

Besides the successful case, several scenarios of failed handover attempts are evaluated as well. The following cases are considered:

- **Uncooperative Person:** The interaction partner does not cooperate and does not appear to want to participate in the handover. The robot should keep trying for a defined time, and then safely abort the process.
- **Unexpected Obstacles:** If, at any point, unexpected obstacles appear, the robot must not collide with them, and should find a way to complete its task despite the new obstacles, if still possible.
- **Blocked Path:** The path may be blocked completely, making it impossible to reach the person. This case should also be detected and an appropriate reaction performed. In this example, it will simply be the abortion of the process.

For all the failure scenarios, a lot of the details of the best reaction by the robot depend on the real-world application. If, for example, the interaction partner does not seem to participate in the handover, it might be a better reaction to use speech or other forms of communication to attempt to initiate an interaction, or to use workflow models to try to understand the current intentions of the person. These techniques do however form their own research areas and lie outside the scope of this work. The important part, from the perspective of the present thesis, is that the robot moves safely in all situations and different types of reactions to failure cases can be expressed in the robot behaviors. Due to the multitude of research questions arising from human-robot handovers, including topics such as psychological factors, control during physical contact, and grasp planning for optimal handovers, the focus of this work has to be narrowed, and not all the aspects can be considered to the same degree. The main concern of this work is thus put on robust and reactive motion planning and control in the pre- and post-handover phases.

3.1.2 Case Study B: Following and Lighting

In this case study, the robot is used to provide light to a worker moving around a factory floor. This use case is used as an example of a long-running, reactive task. The robot is required to keep reasonably close to the worker, without colliding or standing in the way. To provide the light, the robot is provided with a flashlight in its gripper. For the sake of this evaluation scenario, the flashlight is constantly turned on. Besides the basic ability to follow the person while aiming the flashlight, the following cases have to be considered:

- **Getting Stuck:** The robot can get stuck on obstacles, and no longer be able to follow the person in a purely reactive fashion. The robot should be able to find a way around the obstacles to move into the proximity of the person again.
- **Making Room:** As the robot is following the person, the the robot might block the way of the person. For example, this can happen if the reactive following has moved the robot into a corner, while the person wants to access the same corner. The robot should be able to detect this case and move out of the corner, to make room for the person.

The case of an **uncooperative person** is not relevant here, as the person has no active role in the scenario.

In our evaluation, we assume that the person's location is always known to the robot. This is realized through the use of a Vicon 3D motion-capturing system [85]. In real-world applications, this can of course not always be guaranteed. While it is not problematic to detect the human as long as it is close to the robot in many cases, finding them again after the robot got stuck and couldn't follow them closely can be a challenge. One approach of a solution would be to move to where the person was last seen, and begin a search from there. The use of other, less elaborate tracking systems to find the worker again, at least approximately, is also conceivable [22].

3.2 Description of the Robot Platform

A mobile manipulator consisting of a manipulator arm and an omnidirectional base is used for the case studies of this work. It is additionally equipped with a parallel gripper as its end-effector and various cameras and other sensors. The details of the hardware as well as the kinematic model are described in this section. Besides the actuators, sensors, and computers described in this section, the robot is further equipped with a mount for the flashlight on top of the platform. It is used for holding the flashlight before and after handovers, and to keep it during transport.

3.2.1 Actuators

An omnidirectional base provides the robot with mobility, while a manipulator arm and a gripper provide the ability to manipulate its environment and interact with people.

Mobile Platform

The base of the mobile manipulator is a **Neobotix MPO-700** mobile platform [100]. It is equipped with four powered castor wheels, which allow omnidirectional movements of the platform. The platform can drive at a maximum speed of 0.9 m/s, and the included batteries allow for an uptime of up to 5 h. In total, the platform has a width of 0.658 m, and a length of 0.81 m. We use the terms *platform* and *base* equally to refer to this part of the robot.

Manipulator

The robot is equipped with a **Schunk LWA 4P** [128] manipulator. This robot arm has six degrees of freedom, and a maximum payload of 6 kg. The manipulator is mounted on the platform with its base at a height of 0.348 m above the floor. From its base to its flange, the manipulator has a height of 0.945 m. The manipulator is controlled using a CANopen interface. When the term *mobile manipulator* is used, it refers to the entire robot system consisting of mobile base

and arm. In contrast, if only the term *manipulator* is used, this refers to the arm only. This is following the usual terminology within the field [132]. The terms *arm* and *manipulator* are thus used interchangeably.

Gripper

The robot is equipped with a parallel gripper, specifically the model **Weiss CRG-200-85** [118]. This servo-electric gripper is designed for collaborative applications. The gripper is controlled from the integrated platform computer using an IO-Link interface. The gripper has a gripping force of up to 200 N. Objects with a weight of up to 4.3 kg can be held with this gripper.

Kinematic Model

Strictly speaking, the entire mobile manipulator has 15 controlled degrees of freedom:

- 6 arm joints
- 1 opening width of the gripper
- 4 omnidirectional wheels with two axes each

In planning and control, a high number of degrees of freedom often causes significant increases in the required computational cost. This is known as the *Curse of Dimensionality*. It is therefore desirable to reduce the number of controlled degrees of freedom as much as possible. Firstly, the gripper is excluded from reactive control and planning. While it is of course still used, its available motions are limited to grasping or releasing an object. Further considerations of the available degree of freedom are therefore not needed, and the associated dimension does not need to be considered in planning and control. The six joints of the arm are strictly necessary to consider both for planning and control, as they have a direct and strong influence on the pose of the robot in Cartesian space. They can therefore not be simplified. The degrees of freedom introduced by the omnidirectional wheels can however be reduced significantly. The eight degrees of freedom are used to control only three degrees of freedom in Cartesian space: the position and orientation of the base on the floor. The position on the floor has only two dimensions, the orientation only one. Besides these three degrees of freedom, the effects of the wheels are minor. Due to the holonomic nature of the platform, it is assumed that the platform can move in any direction from any configuration of the wheels. The current heading of the wheels has therefore no relevance for the control of the entire robot. The heading of the wheels has some very minor influence on the shape of the base as a whole: depending on which direction the wheels face, they stand out from the platform at different positions. By using large enough safety distances from any obstacles, this is however not practically relevant. In conclusion, the eight degrees of freedom of the wheels can be reduced to three degrees of freedom for planning and control. These three degrees of freedom are called the **virtual base joints** of the platform.

They are called virtual because they do not correspond to physical joints, but are otherwise treated like joints in the calculations. The planning and control systems generate commands for the virtual joints. A low-level driver is tasked with converting the velocity commands to the commands for the eight wheel axes. The current position of the virtual joints is only partially based on the state of the physical wheels: As the three virtual base joints describe the position of the robot regarding a defined origin, determining the state of the virtual base joints corresponds to the more general problem of localization. This is solved through a combination of data sources from the wheel, laser scanners, and external tracking. Many possible algorithms for robot localization exist, a recent overview is provided by Panigrahi et al. [109].

In conclusion, nine degrees of freedom are considered in the concepts for planning and control described in this work. They are three virtual base joints, that abstract the hardware details of the wheeled base, and six arm joints. This kinematic model of the robot forms a single, open kinematic chain. The three virtual base joints are followed by the six arm joints. No branches or circles in the kinematic structure occur.

3.2.2 Sensors

The platform is equipped with two safety-rated laser scanners of the type **Sick microScan 3** [133]. They are positioned at two diagonally opposite corners of the platform. The scanning angle of 275 ° per sensor thus allows to safely supervise the entire area around the robot. The scanners are mounted so that the scanned area is 0.181 m above the ground. The laser scans are also used for mapping, obstacle detection, and localization.

Besides these laser scanners, three different cameras have been added on a rack on top of the platform. The first is a **Ricoh Theta Z1** 360-degree camera, providing a spherical image of the entire surroundings of the robot. The surround view of the camera thus enables the robot with visual information in every direction, so that the robot can, for example, be aware of any human coming close to it. The nature of 360° cameras means that some form of projection is required to map the spherical image to a two-dimensional image. The vast majority of existing computer vision algorithms, such as algorithms for pose estimation, are designed to work on images using a perspective projection. Projecting the entire spherical image with a perspective projection leads to a strongly warped image. Therefore it is recommended to determine some smaller regions of interest and apply the perspective projection to them individually. The further details of this are described in chapter 4.

Besides this camera, a Basler Time-of-Flight (ToF) camera is used for accurate depth information. The camera is an engineering sample and no official information about it appears to be provided by the manufacturer at this point in time. Some further information is available from other scientific publications [103].

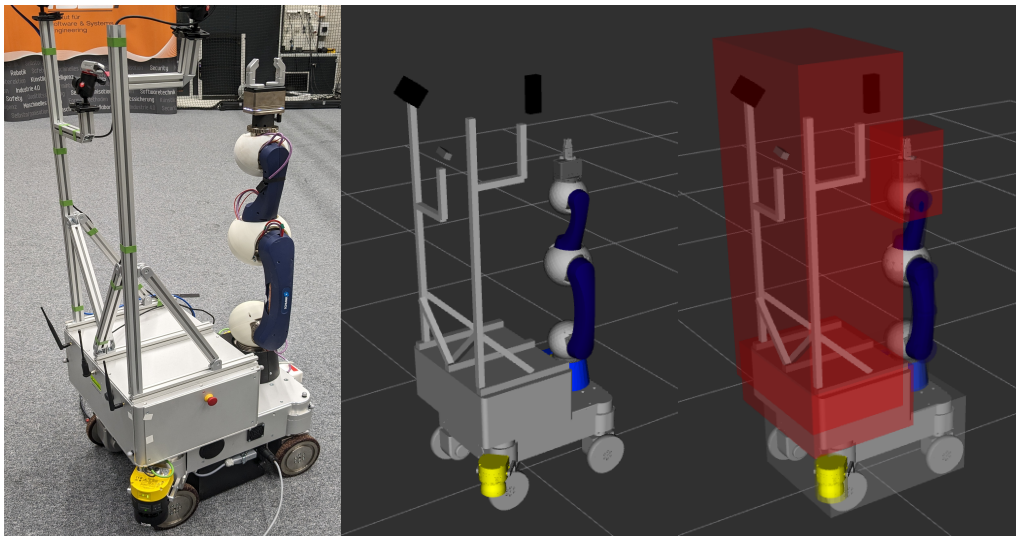
3.2.3 Computing Hardware

The mobile platform is equipped with an on-board computer with an Intel i7 processor and 16 GB of RAM. The computer is running an Ubuntu 22.04 Linux operating system, using a `PREEMPT_RT` real-time kernel [115] to ensure a real-time-capable environment. This computer is used to control the Neobotix platform as well as the Schunk LWA manipulator, and runs all processes related to the control and planning of robot motions.

Besides this computer, an **Nvidia Jetson AGX Orin** [101] embedded computing board has been added to the platform. This is intended to perform most of the sensor data processing of the various sensors. These tasks can be computationally intensive and benefit greatly from GPU acceleration, which the built-in computer of the platform can not provide. It features a 2048-core Nvidia Ampere GPU and a 12-core ARM CPU. This computing board is connected to the platform computer using Gigabit Ethernet for fast data exchange.

3.3 Simulation

In some parts of this work, simulations of the robot are used. These are using the **Gazebo** simulator [45], specifically version 11. Figure 3.1 shows the real robot and its simulation next to each other. Further, the simplified collision geometries of the robots are shown as transparent shapes. The collision geometries are not only used in the Gazebo-simulation, but also for any other calculation that considers collisions, such as planning and control. Since these geometric shapes over-approximate the real robot geometry, the simplification is safe, and can only cause collisions to be detected where none would occur in reality. The opposite, that real collisions are not detected, can not occur. The collision geometry has a much simpler shape than the real robot, which means that collision checks can be performed much more efficiently.



(a) Real robot **(b)** Simulated model **(c)** Collision geometry

Figure 3.1. The real robot compared to its simulated model and the simplified collision geometry.

Summary. In this chapter, all aspects related to the processing and storage of sensor data on the robot platform are described. The first part outlines the necessary steps to detect obstacles, people, and their pose from the available sensors. Next, the environment model and robot model, which encapsulate all relevant data for planning and control, are specified. Lastly, two approaches for the estimation of contact forces based on the measurements of motor currents are presented. One of the approaches is based on an explicit model of dynamical effects, while the other approach uses machine learning. The approaches are then evaluated and compared to each other.

Publication. Parts of this chapter have been published previously [138].

4

Perception and World Modeling

4.1 Image Processing 24
4.1.1 3D Obstacle Map 25
4.1.2 Monitoring of the Surroundings 27
4.1.3 People Detection and Pose Estimation 27
4.2 Environment Model and Robot Model 28
4.3 External Torque Estimation from Motor Torques . . 30
4.3.1 Related Work 30
4.3.2 Model-based methods 31
4.3.3 Learning-based methods 31
4.3.4 Model-based Torque Estimation 32
4.3.5 External Torque Estimation with LSTM Networks . . 37
4.3.6 Validation 39
4.3.7 Evaluation 40

Motion planning and reactive control in dynamic environments require information about the environment. This includes information about the obstacles in the environment, the position of human interaction partners, and the positions of objects to be manipulated. Furthermore, information about the robot itself is necessary as well. The robot's collision geometry, kinematic structure, and joint limits, for example, are essential to motion generation. For safe physical interaction with humans, some method of detecting and measuring contact forces on the robot is required.

This chapter begins with a description of the image processing methods used to estimate the state of the environment. This is followed by a description of the environment robot and robot model, that encapsulate the available data at run-time. To conclude this chapter, the topic of external torque estimation based on motor torques is approached using two different methods.

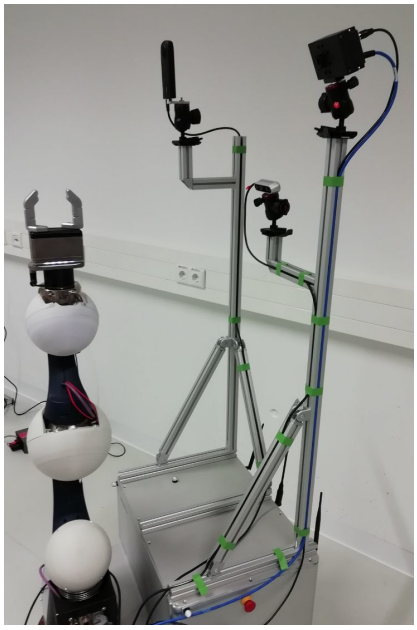


Figure 4.1. Photo of the camera rack and the mounted cameras.

4.1 Image Processing

On the robot platform, three image sources are available:

- **Ricoh Theta Z1**, providing a 360-degree image
- **Intel Realsense D435** camera, capturing color images and depth information (RGB-D)
- **Basler Time-of-Flight** (ToF), also providing depth information.

All three cameras are statically mounted on the camera rack on the robot platform. Their perspective in regards to the robot is thus static. It might be beneficial to have a camera, especially one that provides depth information, installed close to the gripper. This would provide data about close-range objects and objects between the gripper fingers, as well as allow the robot to actively orient the camera towards areas of interest without having to turn the entire platform, which only allows orientation around one axis. However, the camera on the gripper could not be realized due to technical issues. Therefore, all cameras are statically mounted on the platform. They can be seen in figure 4.1. The Ricoh Theta Z1 is shown on the left side of the image, the Intel Realsense D435 is in the middle, the Basler Time-of-Flight camera is the rightmost camera.

Each of the three cameras provides different benefits to the system. The 360-degree camera provides a complete view of the surrounding area but has no depth information. The Realsense D435 camera provides RGB images as well as depth information based on stereoscopic vision. The combination of RGB images with depth information is helpful in the detection and pose estimation

of objects or people. While it is possible to combine the same information from separate cameras, e.g., one depth camera and one RGB camera, this can be a computationally intensive process and prone to errors, as the images are not taken from the same perspective, and the images thus do not necessarily show the same content. Objects that are visible from one perspective might be occluded from another perspective. Matching RGB and depth data is thus considerably simplified by the use of an integrated RGB-D camera.

The depth reconstruction based on stereoscopic matching is, however, highly dependent on the visual features of the scene and can easily produce erroneous results in difficult scenes. This makes it unsuitable as a sole data source for obstacle avoidance. This is a problem that the time-of-flight technology, as used by the Basler ToF camera, does not have. The downsides of this camera, lie in the absent RGB information and comparatively slow frequency of up to 15 Hz, while the Realsense D435 provides depth information at frequencies of up to 90 Hz.

We describe here a concept for the detection of obstacles and people from the available sensors. Further perception abilities, such as object detection, are also possible, but depend on the specific application details and are not described here.

The cameras are used for different purposes according to their capabilities. The time-of-flight camera is used strictly for obstacle detection. The 360-degree camera is used to monitor the surroundings so that the robot can be aware of any human approaching it. The RGB-D camera, lastly, is intended for the perception of interaction objects, where both some segmentation and pose estimation are required.

4.1.1 3D Obstacle Map

The depth information provided by the Basler ToF camera can be used to create a three-dimensional environment map using, for example, the popular OctoMap framework [60]. The resulting map, based on the data structure *octree*, can be queried efficiently as a basis for obstacle avoidance in planning and control. This simple solution has the problem that the robot itself is constantly recorded by the camera. The robot itself should not be registered as an obstacle on not be part of a map. The same goes for human interaction partners. While they must of course not be harmed, the necessary reactions are often different from those of more generic environmental obstacles and should thus be differentiated. For example, in a handover, the robot has to come in direct physical contact with the person, which should not happen with other environmental obstacles. The image in figure 4.2 shows an example of a depth image provided by the camera, with the robot gripper as well as the resulting occluded area behind it clearly visible.

The robot, and possibly the person, thus have to be removed from the depth scans, while accounting for the current robot pose. For the robot, this can be done based on the description of the robot's geometry and the current robot configuration. From this information, the predicted space occupied by the robot

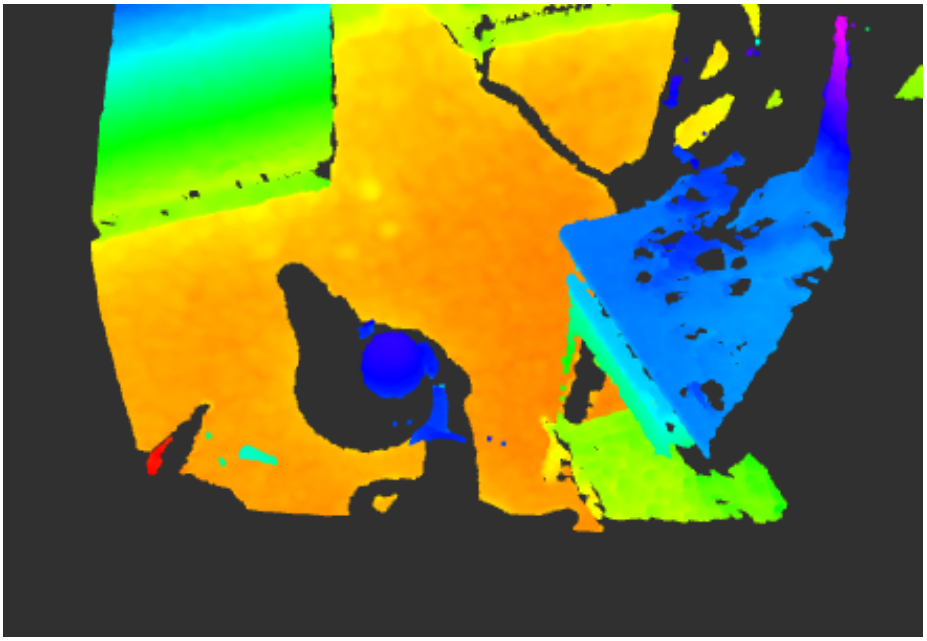


Figure 4.2. Example depth scan from the Basler ToF camera.

in its current configuration can be calculated. Any scanned points that are within a threshold of the calculated robot parts of the image are removed. A similar process is used for the filtering of the person. Once the estimated pose of the person is known, any points within a defined range around the estimated points are removed from the scan. The thresholds are chosen in a way that makes it much more likely that too much is removed rather than not removing enough. Measurements that slip through the filter can disturb the robot controller by being registered as an obstacle, and leading to an abrupt stop. On the other hand, it is presumed to be unlikely that an obstacle newly appears within the direct proximity of the person or the robot, without being registered on the map first, while it was approaching.

The map, as provided by OctoMap, can then be used for planning and control. Distance calculations using octrees are however relatively slow, compared to those using a map based on primitive geometric shapes such as spheres, cubes, and planes. Finding good approximations using geometric primitives based on a depth map is however not trivial and quite computationally intensive [127]. Therefore, it is currently not feasible in real-time. One approach to still make use of geometric primitives can be to use phases where the robot is not in use to perform the necessary calculations and simplify the map. As soon as there are substantial changes to the map, this would have to be repeated.

4.1.2 Monitoring of the Surroundings

The Ricoh Theta Z1 360-degree camera is used to detect any person approaching the robot. While the available image by the camera is not ideal for highly accurate perception tasks, due to perspective projection issues and only containing two-dimensional RGB data, the fact that it records in all directions makes it a valuable sensor.

The image, as provided by the camera, covers a full 360-degree field of view. Displaying it on a two-dimensional image, whether for human viewing or image processing, thus requires some form of projection. The most basic projection is the so-called *equirectangular* projection. This operation causes a curved distortion on the image, which easily disturbs existing algorithms that are typically designed to work with images using a *perspective* projection instead.

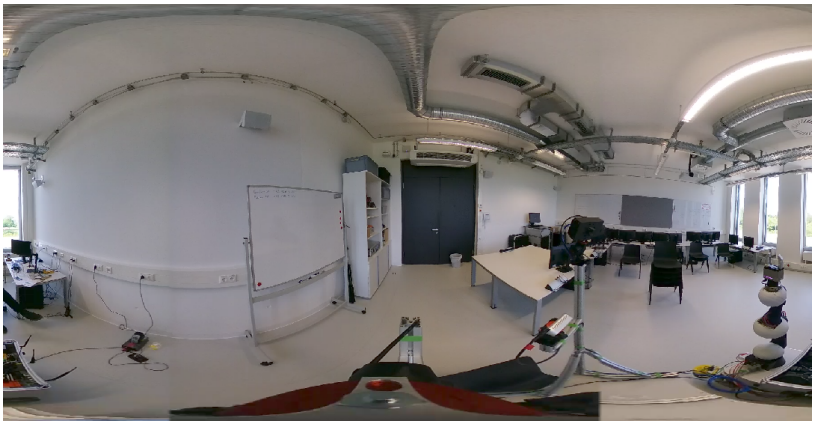
Therefore, the image should be projected to use a perspective projection before further processing. The smaller the field of view, the smaller will be the distortion on the resulting perspective-projected image. Therefore, not the full 360-degree image is projected, but it is divided into four sections. Each section is then projected individually. In order to avoid problems in cases where people are positioned on the dividing line between sections, the sections are chosen to have an angle of 120 degrees each, so that there is a significant overlap between the sections. Each of the sections is then transferred to a perspective projection. Example images showing the curved distortion of equirectangular projection, as well as the resulting image section when using perspective projection, are shown in figure 4.3.

The resulting images can then be used by a people detection method. This is a highly active research area with many competing implementations. We have chosen the popular *peopleNet* [116] for its good results and readily available implementation for the Nvidia Jetson.

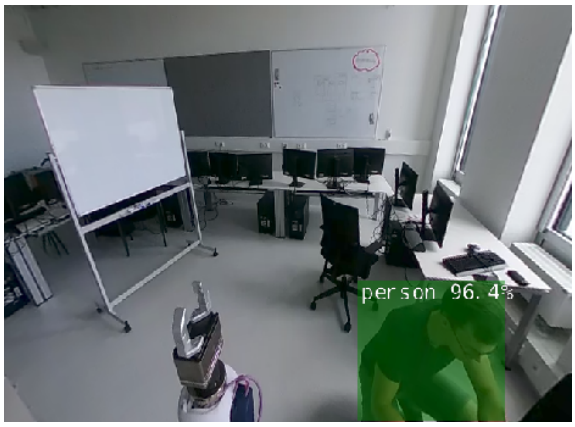
Once a person has been detected, the angle at which it approaches the robot can be calculated from the person's position in the image. Further steps for an interaction are then taken using the Realsense RGB-D camera, as described below.

4.1.3 People Detection and Pose Estimation

Once the person to interact with has been identified using the 360-degree camera, the robot can position itself so that the person is within the field of view of the Realsense D435 RGB-D Camera. Then, existing algorithms [15] based on part-affinity fields are used to estimate the pose of the person from the RGB-D image. The referenced algorithms provide implementations optimized to run on the computing hardware of the Nvidia Jetson AGX computing board, so that the GPU capabilities can be used as much as possible. For now, the system is restricted to a single person within the proximity of the robot. If multiple people are detected approaching the robot, it will stop moving as a safety precaution.



(a) Full 360-degree example image using equirectangular projection.



(b) Image section transformed to perspective projection, with the result of person detection shown.

Figure 4.3. Example images from the 360-degree camera using equirectangular and perspective projection.

To operate the robot in a multi-person scenario, some method of differentiating them and detecting their identity is required. This lies outside of the scope of this work.

4.2 Environment Model and Robot Model

In this section, the internal model used to represent the knowledge about the environment as well as the structure and geometry of the robot itself are described. The information about the robot itself is stored in the robot model. All information about the environment, such as the poses of external frames, obstacles, etc., are stored in the environment model.

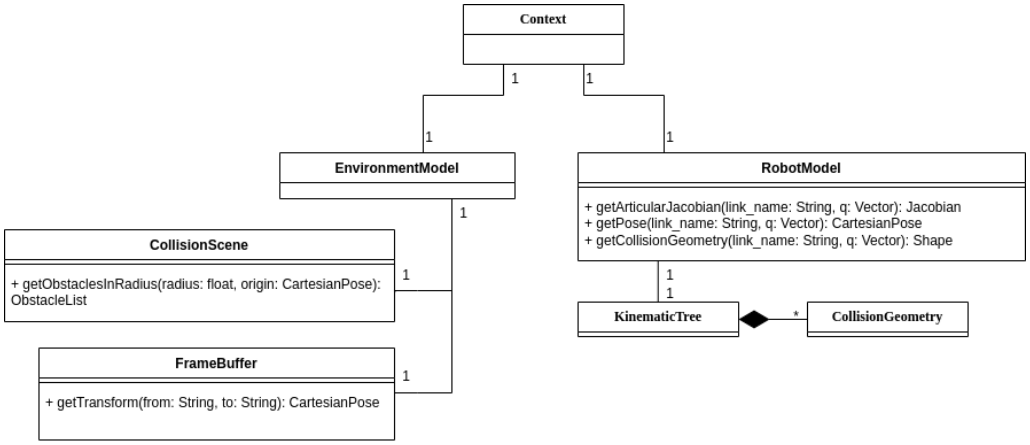


Figure 4.4. Class Diagram of the `Context` model with the most important elements.

The combination of robot model and environment model is called the *context*. Figure 4.4 shows the class structure of the context. The context encapsulates the *environment model* and the *robot model*. It is important to note what are not included in the context, namely the current state of the robot. The robot model only contains the robot information that does not change over time, such as the kinematic structure and the collision geometries of the robot's links. The robot's current configuration, velocity, and other time-dependent information is not included. This will be important later for planning, where multiple samples of robot configurations are considered in the same context.

The environment model contains collision geometries of external obstacles, the robot model collision geometries of the robot itself. In both cases, the geometries can be described through different means. The currently supported types of collision geometries, both for external obstacles and the robot itself, are:

- Spheres
- Cylinders
- Cones
- Boxes
- Convex Meshes
- Octrees

Distance computations using arbitrary meshes or octrees are generally much more computationally intensive than calculations using geometric primitives, such as spheres. It is advisable to simplify the geometries to geometric primitives as far as possible. For the use of unstructured online sensor data, such as depth sensors, octrees, or similar data structures might nonetheless be the only suitable choice.

4.3 External Torque Estimation from Motor Torques

For robots working in close proximity and interacting with humans, it is essential to detect when humans are coming close or even touching the robot. Cameras, including depth cameras, are not reliable enough for these scenarios. A main factor is occlusion, which is almost unavoidable when in direct contact with the robot.

Previous works have attempted to detect approaching human limbs reliably using capacitive sensors on stationary robots [112]. However, capacitive sensing is highly dependent on various environmental factors, including temperature and humidity. While the sensors can mostly be calibrated to account for this variability in the stationary case, this is not possible for mobile robots in changing environments. Therefore, other solutions are needed. A big advantage of capacitive sensing is its ability to detect human limbs before they come in contact with the robot. Without some other form of specialized sensor hardware, this is not possible otherwise. However, detecting touch and pressure can still have large benefits in collaborative scenarios. This can be possible even without additional sensors.

The basic concept of the approach presented here is to compare the measured motor torques to the expected torques without any contact. An important question is then how the expected torques are found. We assume that environmental contact is only relevant with the arm, not with the base. Collision avoidance of the mobile base can be ensured through the laser scanners, and physical interaction is typically not expected or required. The arm, on the other hand, is used to interact with the environment and humans. It can thus not be restricted in this way. Both methods presented here calculate the estimated external joint torques $\hat{\boldsymbol{\tau}}_{ext} \in \mathbb{R}^6$ of the six arm joints.

4.3.1 Related Work

The problem of torque estimation is strongly related to the problem of inverse dynamics, where torques required to reach a given robot state are calculated. The problem of inverse dynamics is to find the joint torques τ that are necessary to reach a desired robot state, described by its position, velocity and acceleration $(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$. As a function, it can be expressed as follows: $f(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) = \tau$. In external torque estimation, it is assumed that the problem is already solved within the internal robot controllers. Nevertheless, the expected torques given the current robot state need to be estimated to compare them to the measured values.

Recent works on mobile manipulator dynamics have applied inverse kinematics to the problem of trajectory tracking, where both model-based [142, 114] and learning-based methods [68, 28] are employed. The referenced works are only examples from a multitude of publications. One publication attempted to identify the full dynamics of a holonomic mobile manipulator with powered castor wheels and a six-joint manipulator [65], similar to the one used in the present work. However, the authors had to reduce the complexity by restricting the robot to simple motions and excluding curves and self-rotations. This makes it unsuitable

for our intended application in human-robot interaction, where the full range of robot motions is required. To our knowledge, no recent publication deals with the estimation of external torques from measured joint torques on mobile manipulators. Below, we give an overview of existing methods for fixed-base serial manipulators.

4.3.2 Model-based methods

Modeling the dynamics of a serial manipulator by identifying the dynamic parameters is an established approach and has been successfully used for a range of manipulators [44, 147], including the Schunk LWA manipulator [95, 84]. The modified hardware we use, as well as the added influence of the mobile base, make the identified parameters not immediately usable for our case. Related methods have been used for many years and a wide range of techniques has been proposed. There are too many to list here, recent surveys are available [79, 141]. Model-based torque estimation has also been studied extensively, the work of Mamedov et al. [91] gives an overview.

4.3.3 Learning-based methods

The idea of using machine learning for inverse robot dynamics has been explored in several publications. It is attractive for this use case as it does not require an explicit model of the physical effects, and the relations can instead be automatically inferred from recorded data. Machine learning approaches dealing with time series data are often preferred for robot dynamics applications, because the temporal relations of the data can be exploited and the results do not have to rely on a single measurement. *Long short-term memory* networks (LSTM) [57] are a particularly successful method used with great success in a wide variety of applications dealing with time series, including robot dynamics. LSTM networks are a type of recurrent neural network, that are especially designed to handle long-term dependencies. They have proved successful in a large range of applications and are widely used today.

Rueckert et al. [121] applied the LSTM technique to the problem of fixed-base manipulator inverse dynamics prediction. They found a two-layer LSTM architecture with 10 hidden neurons per layer and no dropout layer to perform best for their use case. Yilmaz et al. [148] have successfully applied a set of dense neural networks to external force estimation on the da Vinci surgical robot. Similar to our approach, they use only joint positions and velocities as input, however merely use dense neural networks and apply it to surgical robotics. Another application of learning in robot dynamics is to allow the model to adapt to changes that occur during the robots' lifetime, for example due to wear [113]. In these cases, the learning process continues during the operation of the robot. Lim et al. [81] have combined a model-based momentum observer with an LSTM for uncertainty learning. Liu et al. [83] extensively evaluate LSTM techniques for the inverse dynamics problem. Their results show that using only joint position

a	α	d	θ
0	q_x	0	0
0	q_y	0	$\pi/2$
0	0.21	0.5753	q_θ-$\pi/2$
$\pi/2$	0	0	q_1
π	0.35	0	$q_2 + \pi/2$
$\pi/2$	0	0	$q_3 + \pi/2$
$-\pi/2$	0	0.305	q_4
$\pi/2$	0	0	q_5
0	0	0.075	q_6

Table 4.1. Denavit-Hartenberg parameters of our robot model

as input for torque prediction leads to much lower precision compared to using position, velocity and acceleration. They do not evaluate whether using position and velocity is feasible. This is what is attempted by the work described in this chapter.

4.3.4 Model-based Torque Estimation

Explicit modeling of the robot dynamics requires exact knowledge of its physical properties. The required values are called the *dynamic parameters*. These are usually not available from the manufacturer and difficult to measure directly, even if the robot were to be dismantled into its parts. Instead, it is an established technique to identify the dynamic parameters using a linearized dynamic model and least-squares optimization on the measurements of a specifically designed excitation trajectory. Our process for the identification of the dynamic parameters is based on the six-step identification procedure outlined by Swevers et al. [140]. An additional seventh step describes the online estimation of external torques. For each of the steps, our method is described below.

Modeling

A full dynamic model of a mobile manipulator with castor wheels would have to consider the robot as a floating-base kinematic tree, where each wheel contributes two controlled degrees of freedom and interactions between the wheels and the ground are modeled explicitly. Since we are only interested in the torques occurring at the arm joints, we can reduce the model significantly by only considering the motions of the omnidirectional base in the horizontal plane as a whole. The base is thus modeled by three virtual joints, corresponding to the linear x position, linear y position, and yaw, respectively. Hence we model the robot system as a single kinematic chain with 9 joints described by the standard Denavit-Hartenberg parameters shown in table 4.1. Notationally, indices 1 through 6 refer to the respective arm joints, while indices x, y, and θ are used for the virtual base joints.

The model of robot dynamics is based on the classical dynamics equation for serial manipulators:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{F}(\dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q}) + \boldsymbol{\tau}_{ext} = \boldsymbol{\tau}, \tag{4.1}$$

where $\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}} \in \mathbb{R}^n$ are the generalized coordinates, velocities and accelerations, n is the number of joints and has the value 9 in our case (3 virtual base joints and 6 arm joints). $\mathbf{M}(\mathbf{q}) \in \mathbb{R}^{n \times n}$ is the inertia matrix, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) \in \mathbb{R}^{n \times n}$ is the matrix of Coriolis and centrifugal forces. Vectors $\mathbf{F}(\dot{\mathbf{q}}) \in \mathbb{R}^n$ and $\mathbf{g}(\mathbf{q}) \in \mathbb{R}^n$ describe the influence of friction and gravity, respectively. $\boldsymbol{\tau} \in \mathbb{R}^n$ and $\boldsymbol{\tau}_{ext} \in \mathbb{R}^n$ are the vectors of actuator and external torques. $\boldsymbol{\tau}_{ext}$ should be zero as long as the robot is not in contact with the environment. Torques and forces in the virtual base joints are not measured and they have no influence on friction or gravity, but they do exert an influence on the inertial and Coriolis/centrifugal forces on the arm joints through their movements. We do not distinguish this in the equations to keep the notation simple. The identification process relies on the reformulation of the dynamics equation into a form that is linear in the dynamic parameters (assuming $\boldsymbol{\tau}_{ext} = 0$):

$$\mathbf{Y}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})\boldsymbol{\pi} = \boldsymbol{\tau} \tag{4.2}$$

$\mathbf{Y}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$ is the so-called regressor matrix and $\boldsymbol{\pi}$ is the vector of dynamic parameters. The dynamic parameters that are considered vary in the related literature, according to the considered effects. We have used a total of 117 elements in $\boldsymbol{\pi}$, 13 for each link i:

$$\begin{aligned} \boldsymbol{\pi}_i = [\quad & m_i \quad l_{ix} \quad l_{iy} \quad l_{iz} \quad I_{ixx} \quad I_{ixy} \quad I_{ixz} \\ & I_{iyy} \quad I_{iyz} \quad I_{izz} \quad I_{mi} \quad f_{ci} \quad f_{vi} \quad]^T \end{aligned} \tag{4.3}$$

m_i is the mass of the link, l_{ix}, l_{iy}, l_{iz} describe the center of mass, I_{mi} models the rotor inertia and $I_{ixx}, I_{ixy}, I_{ixz}, I_{iyy}, I_{iyz}, I_{izz}$ are the relevant entries of the inertia tensor. We have modeled the effects of friction as part of the dynamic parameters, using the values f_{ci} and f_{vi}. The influence of friction is modeled as the sum of Coulomb and viscous friction according to the following equation:

$$\mathbf{F}_i(\dot{\mathbf{q}}_i) = f_{ci}\,\mathrm{sign}(\dot{\mathbf{q}}_i) + f_{vi}\dot{\mathbf{q}}_i \tag{4.4}$$

The friction parameters can thus be estimated using the same process as used for the other dynamic parameters. Other works have used different friction models that are not linear in the dynamic parameters and thus cannot be handled in the same way. A wide variety of friction models has been created [5, 102].

For the Schunk LWA manipulator, for example, load-dependent friction models [84] and hyperbolic friction models [95] have been used. The former requires an explicit gravity model, the latter produces results that are not consistent with our measurements and uses a procedure that requires mounting the manipulator in different orientations to isolate individual joints from the effects of gravity. For the sake of simplicity, we have decided to use a linear friction model. More intricate friction models might improve accuracy. In most cases, not all dynamic parameters affect the joint torques. The number of required parameters can be reduced by eliminating and linearly combining parameters. The remaining minimum set of required parameters is known as the *base parameters*. Here, the word *base* does not refer to the mobile base of the robot, but is used in the sense that these parameters are the most important parameters.

The technique that we have used to estimate the values of the base parameters is based on QR decomposition and was first described by Gautier et al. [43]. With this technique, the dynamic parameters were reduced from the original 117 to 55 parameters. The resulting base parameters are shown in table 4.4 at the end of this chapter. As expected, all dynamic parameters of the mobile base vanish, as they do not influence the torques at the arm joints. For example, adding additional weight to the platform would not change the effects that platform motions have on the arm.

Experiment Design

The base parameters are identified from data recorded as the robot is moving along a given trajectory. To find accurate parameters, it is important to use a well-designed trajectory. An unfortunately chosen trajectory would not include all relevant information to make a good estimate of the base parameters. If, for example, each joint was only moved in isolation, the compound effects could not be estimated from the recording of the trajectory. Thus, it is important to choose a suitable trajectory for the experiments. A property that can measure the suitability of a trajectory has been proposed. This property is called the persistence of excitation [140]. This resulting trajectory is thus called the *excitation trajectory*. In order to find an excitation trajectory, the trajectory is modeled as a Fourier series:

$$\mathbf{q}_i(t) = \sum_{l=1}^{6} \left(\frac{a_l^i \sin(\omega l t)}{\omega l} - \frac{b_l^i \cos(\omega l t)}{\omega l} \right) + \mathbf{q}_{i,0} \tag{4.5}$$

$$\dot{\mathbf{q}}_i(t) = \sum_{l=1}^{6} \Big(a_l^i \cos(\omega l t) + b_l^i \sin(\omega l t) \Big) \tag{4.6}$$

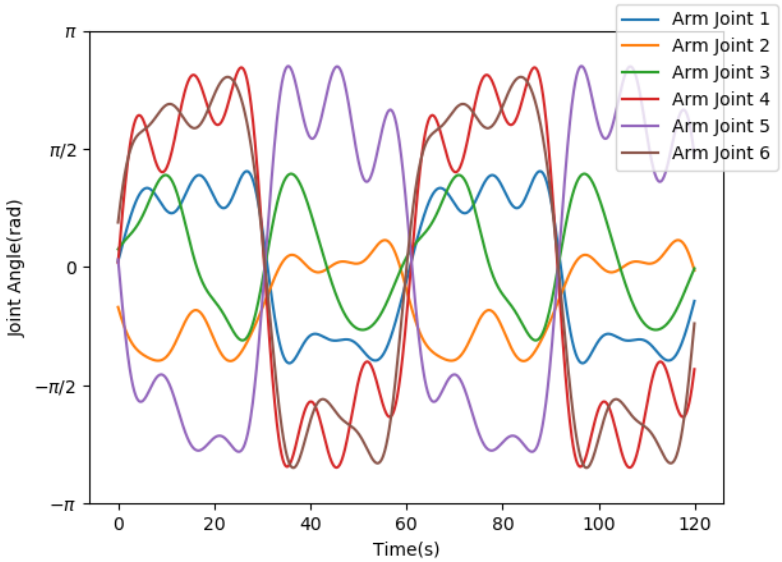


Figure 4.5. The excitation trajectory for the arm joints.

Here, ω is the fundamental frequency and was set to $2\pi/60$, so the trajectory repeats after 60s. $\mathbf{q}_{i,0}$ is a static position offset. i stands for the index of the respective joint. a and b are the optimization variables influencing the shape of the trajectory. Using a Nelder-Mead method, they were chosen to minimize the cost function $\text{cond}(\mathbf{Y}) + \frac{1}{\sigma(\mathbf{Y})}$ over all trajectory points, where $\text{cond}(\mathbf{Y})$ is the condition number of the regressor matrix, and $\sigma(\mathbf{Y})$ is its smallest singular value. The mobile manipulator we use has a relatively high risk of self-collision, due to the structure of the mobile base being very close to the arm links. This forced us to use strict joint position constraints during the optimization. The trajectory we have used has a total duration of 120s. The positions of the arm joints in the trajectory are shown in figure 4.5, and the velocities of the virtual base joints are shown in figure 4.6.

Data Acquisition

The controller used to control the robot along the excitation trajectory is running at a rate of 250Hz. Positions, velocities, and torques of the arm joints are read from internal sensors at the same rate. The base position and velocity are being tracked with a rate of 120Hz by a Vicon Vantage infrared tracking system, to avoid errors due to inaccurate odometry or localization. The manipulator is controlled in interpolated position mode. The setup described here also applies to all other recordings in this chapter.

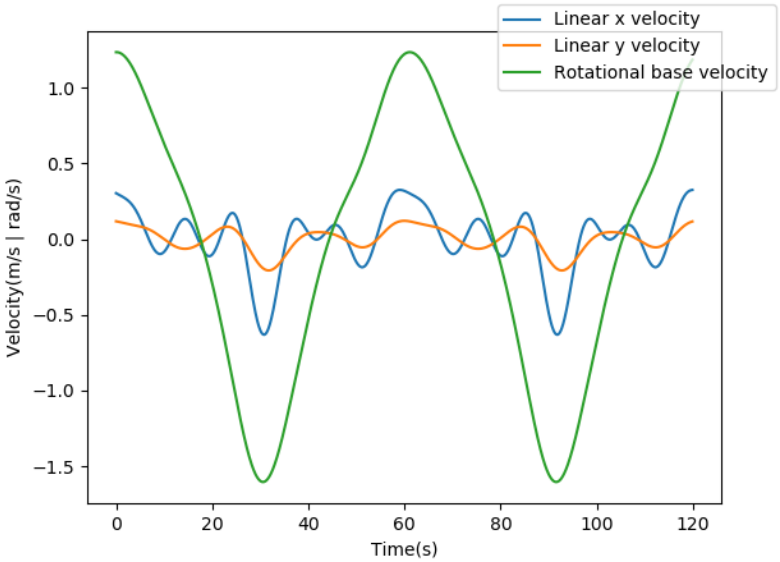


Figure 4.6. Velocities of the base in the excitation trajectory.

Signal Processing

The dynamic model strongly depends on the joint accelerations $\ddot{\mathbf{q}}$. However, these are not measured directly and are subsequently not provided by the driver interfaces. Therefore, they need to be estimated from the recorded data, namely the joint velocities $\dot{\mathbf{q}}$. Since the accelerations are only used for the offline parameter estimation, and not during online operation, the accelerations can be estimated relatively well from the velocity measurements. Mathematically, the accelerations can be found by simply differentiating the velocity measurements. However, the presence of noise in the velocity measurements means that simple differentiation is not suitable. The noise present in the velocity is increased substantially during differentiation and the resulting signal would not be usable anymore. To circumvent this, first, the recorded velocities have been filtered with with a zero-phase bandpass Butterworth filter. A zero-phase filter is a filter that does not produce phase shift on any frequency. After the velocities have been filtered thusly, numerical differentiation is used to estimate the joint accelerations. These are again filtered by a zero-phase lowpass filter to remove high-frequency noise.

Parameter Estimation

After the data has been recorded and processed, the joint positions $\mathbf{q}$, velocities $\dot{\mathbf{q}}$, accelerations $\ddot{\mathbf{q}}$, and torques $\boldsymbol{\tau}$ are available for each recorded trajectory point. The regressor equation $\mathbf{Y}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})\boldsymbol{\pi} = \boldsymbol{\tau}$ can thus be set up for each trajectory point. Still missing are the values of the base parameters $\boldsymbol{\pi}$.

Since each trajectory point provides an equation depending on $\boldsymbol{\pi}$, with all other variables filled in, the values of $\boldsymbol{\pi}$ can be found using an optimization procedure to find values that best fit the recorded data. A least-squares method has been used for each recorded trajectory point of the excitation method to estimate the 55 base parameters. The resulting 55 values are listed in table 4.4.

Validation

The approach described above has been validated on a reference trajectory. For the sake of brevity and in order to provide a better comparison, the validation and its results are described together with the validation of the LSTM approach in section 4.3.6.

External Torque Estimation

After the necessary base parameters have been estimated, still the expected torques have to be found. Directly using equation (4.1) is not feasible for online torque estimation. This is due to the fact that joint accelerations $\ddot{\mathbf{q}}$ are not available at runtime. Direct differentiation of the joint velocities $\dot{\mathbf{q}}$ is not applicable here. Post-processing methods that were used in the offline parameter estimation step are not applicable online. All available methods would either strongly amplify the noise, rendering the signal nearly useless, or introduce a delay that is unacceptable for collision detection.

Several methods have been developed to overcome this problem. The works of Haddadin et al.[51] and Mamedov et al. [91] provide overviews. We decided to use a well-established method, the so-called momentum observer [87, 86]. This method estimates $\boldsymbol{\tau}_{ext}$ by observing the rate of change of the generalized momentum $\mathbf{p}$:

$$\mathbf{p} = \mathbf{M}(\mathbf{q})\dot{\mathbf{q}} \qquad \dot{\hat{\boldsymbol{\tau}}}_{ext} = \mathbf{L}(\dot{\mathbf{p}} - \dot{\hat{\mathbf{p}}}) \tag{4.7}$$

$\mathbf{L}$ is a gain matrix, whose entries were tuned by hand in our case. Integration over time then leads to the estimate of the external torques. Further details about the momentum observer, including the handling of the friction model, can be found in the referenced publications.

4.3.5 External Torque Estimation with LSTM Networks

The second method we employed is fundamentally different. Here, the torque estimation is based on machine learning. No dynamic effects and their parameters need to be modeled explicitly, only a learning architecture and the training data need to be defined. Learning-based approaches have already been used to study

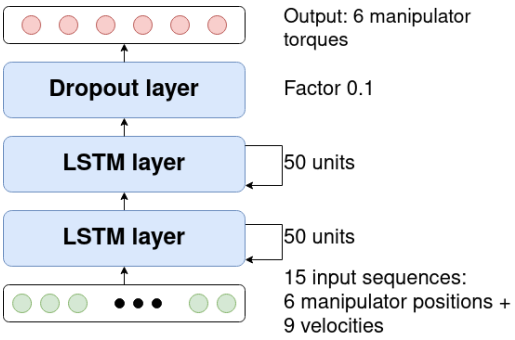


Figure 4.7. Our LSTM network architecture.

the classical inverse dynamics problem $f(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) = \tau$ [39]. As in the model-based approach, the reliance on $\ddot{\mathbf{q}}$ is problematic for online torque estimation.

In the model-based approach, a momentum observer has been used to circumvent the need for accurate measurements of $\ddot{\mathbf{q}}$ at run-time. During offline signal processing, filtering methods have been used to find a less noisy acceleration estimate.

Here, for the learning-based approach, we study the hypothesis that the neural network will be able to find the necessary information by itself by providing it with time-series information of the velocities $\dot{\mathbf{q}}$. Instead of explicitly estimating $\ddot{\mathbf{q}}$ from noisy measurements of $\dot{\mathbf{q}}$, we are moving this task to the neural network. Clearly, it is impossible to estimate $\ddot{\mathbf{q}}$ from a single value of $\dot{\mathbf{q}}$. Instead, the neural network is given a sequence of the last measurements of $\dot{\mathbf{q}}$. LSTMs are a type of neural network that has been specifically designed as a variant of recurrent neural networks (RNN) to work with sequential data points [56], which makes them suitable for our application.

Network architecture

Figure 4.7 illustrates our network architecture. We achieved the best results with two LSTM layers with 50 units each, followed by a dropout layer. The dropout layer serves as a regularization method. Connections to the units in the LSTM network are excluded from updates with a given probability, called the dropout factor, during the training phase. This has the effect of reducing over-fitting. Experiments with additional units and dense layers did not lead to improvements in our case, but a systematic exploration may be worthwhile to further improve the network performance. The input sequences consist of the last five measurements, corresponding to 20 ms of measurements.

The input layer has 15 units, six manipulator joint positions, and nine joint velocities including the virtual joints. The output layer has six units, which are the six estimated torques of the arm joints. Results were improved by filtering the torques used in training with a zero-phase lowpass filter. Training was performed with a batch size of 128 and over 300 epochs.

A downside of neural networks, compared to explicit dynamic models, is that generalization to unseen data points can be less predictable. With an explicit model, the calculated values will always follow the physical laws of the dynamic equation. Neural networks, on the other hand, make it much harder to predict how they generalize to inputs that were not contained in the training set. Therefore it is important that the training data covers as wide a range of data as possible, contrary to the excitation trajectory for parameter identification, where numerical stability of the estimate is important, but generalization is generally not a problem. As training data, we used the excitation trajectory as described in section 4.3.4, and added recordings of another two different 120 s trajectories, as well as 100 static positions and the motions between them. In total, eleven minutes of recorded robot motions were used as training data for the LSTM.

External Force Estimation

The neural network is trained to estimate the actuator torques $\hat{\boldsymbol{\tau}}$ under the assumption that the robot is not in contact with the environment, i.e. $\boldsymbol{\tau}_{ext} = 0$. The estimated external torques $\hat{\boldsymbol{\tau}}_{ext}$ can then be found by comparing the predicted torques $\hat{\boldsymbol{\tau}}$ to the actual measurement $\boldsymbol{\tau}$:

$$\hat{\boldsymbol{\tau}}_{ext} = \boldsymbol{\tau} - \hat{\boldsymbol{\tau}} \tag{4.8}$$

A downside of this method is that noise found in the measurements of $\boldsymbol{\tau}$ extends directly to $\hat{\boldsymbol{\tau}}_{ext}$. In order to alleviate this noise as much as possible, an additional Kalman filter is used on the signal.

4.3.6 Validation

The two models are validated by comparing the model estimates to recorded torques on a reference trajectory. We used a trajectory with a total duration of 120 s, formed according to the Fourier series shown in equation (4.5). The parameters are chosen differently than those for the excitation trajectory and the LSTM training. The trajectory used for validation is plotted in figure 4.8 and figure 4.9, showing the arm and base trajectories, respectively.

Here, we only compare the results for the estimated joint torques $\hat{\boldsymbol{\tau}}$ and do not consider the estimated external torques $\boldsymbol{\tau}_{ext}$. As the validation trajectory does not include any contact with the environment, $\boldsymbol{\tau}_{ext}$ is assumed to be zero over the entire trajectory.

For the model-based approach, we calculate $\hat{\boldsymbol{\tau}}$ using the base parameters according to the regressor equation as defined in equation (4.2). The momentum observer is not yet used. For the LSTM approach, we use the output of the network, without comparing it to the recorded torques nor using a Kalman filter. The results are shown in figure 4.10. While it can be seen that both methods generally follow the shape of the recorded torques, some errors remain regardless in the outputs of both methods.

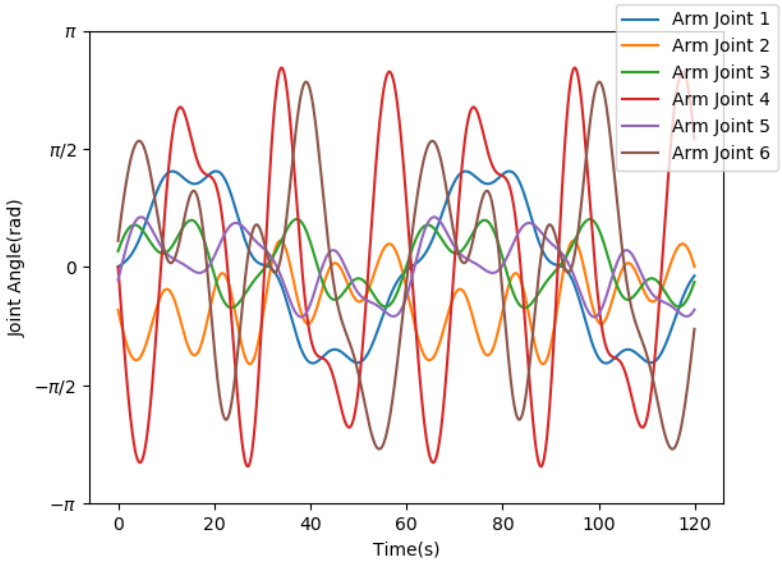


Figure 4.8. The evaluation trajectory of the manipulator joints.

		Added Weight				
		0 kg	0.8 kg	1.3 kg	2 kg	3.25 kg
Joint	1	0.353	0.636	0.454	0.551	0.478
	2	32.066	22.497	29.737	25.634	32.142
	3	14.704	4.370	5.017	3.702	5.686
	4	0.548	1.527	2.342	1.751	2.822
	5	0.041	0.017	0.023	0.024	0.042
	6	0.002	0.004	0.002	0.003	0.004

Table 4.2. Mean squared error of the momentum observer estimations.

4.3.7 Evaluation

From the perspective of practical experiment design, applying accurate reference forces to a moving mobile robot is challenging. As the influence of the moving mobile base is a core question of this research, the robot should not be stationary. The applied forces must be known exactly in order to evaluate the performance of our approaches. Letting a person be in physical contact with the robot is thus unsafe, as the robot should move at different speeds, including higher ones, and can not yet safely react to collisions. Furthermore, this type of experiment would also be unproductive, as the values of the applied forces would not be known and thus no reference values for the model outputs would be available.

Our solution consists of attaching different weights to the robot's end effector. The robot then executed the trajectory shown in figure 4.8 and figure 4.9 with and without the added weight. Any difference between the recorded torques with and

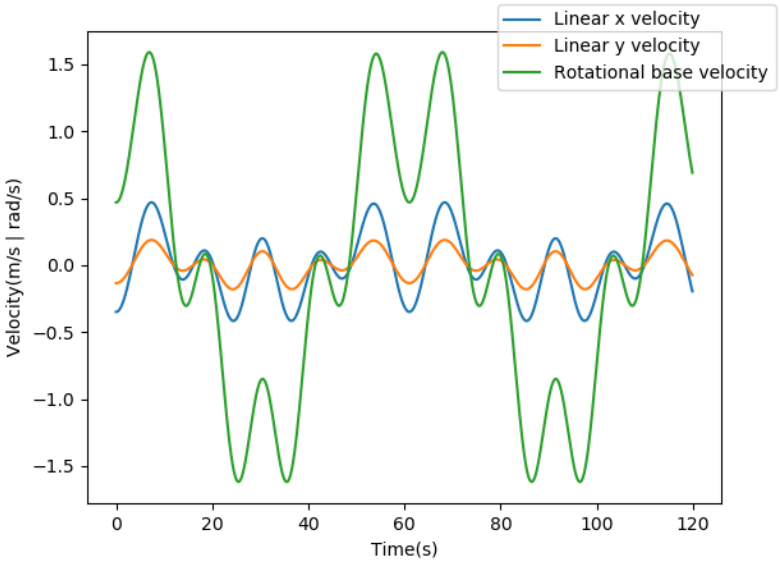


Figure 4.9. The evaluation trajectory of the base.

		Added Weight				
		0 kg	0.8 kg	1.3 kg	2 kg	3.25 kg
Joint	1	0.041	0.458	0.405	0.482	0.358
	2	14.317	16.948	11.991	11.950	13.903
	3	4.942	5.042	4.638	4.162	3.863
	4	1.087	2.624	3.780	2.977	3.492
	5	0.042	0.033	0.044	0.046	0.047
	6	0.025	0.025	0.030	0.027	0.028

Table 4.3. Mean squared error of the LSTM estimations.

without the added weights at the end-effector can then be used as the reference external torque. As the recording of the reference torques also includes some noise, they are filtered offline with a zero-phase low-pass filter. We evaluated our method with four different weights with 0.8 kg, 1.3 kg, 2.0 kg and 3.25 kg, as well as with the empty manipulator. The trajectory used for the evaluation has a duration of 120 s and was calculated as a Fourier series as described in equation (4.5).

The resulting mean squared error (MSE) of the momentum observer and the LSTM are shown in table 4.2 and table 4.3, respectively. Additionally, the results for the case with an added weight of 3.25 kg are plotted in figure 4.11. This method of evaluation creates only very small external torques on joints 1 and 6. Joint 1 is rotating around an axis perpendicular to the ground, as a consequence physical effects related to gravity of the attached weight are almost absent for this joint. Joint 6 rotates around an axis that crosses through the center of mass of the attached weights, and it is the closest joint to the weight. The effects

caused by the additional weights are consequently very small for joint 6. The measurements for joints 1 and 6 are thus dominated by noise and accurate tuning of filters and gain parameters was not possible. Further evaluation with a different experimental setup is needed to accurately assess the accuracy for these joints.

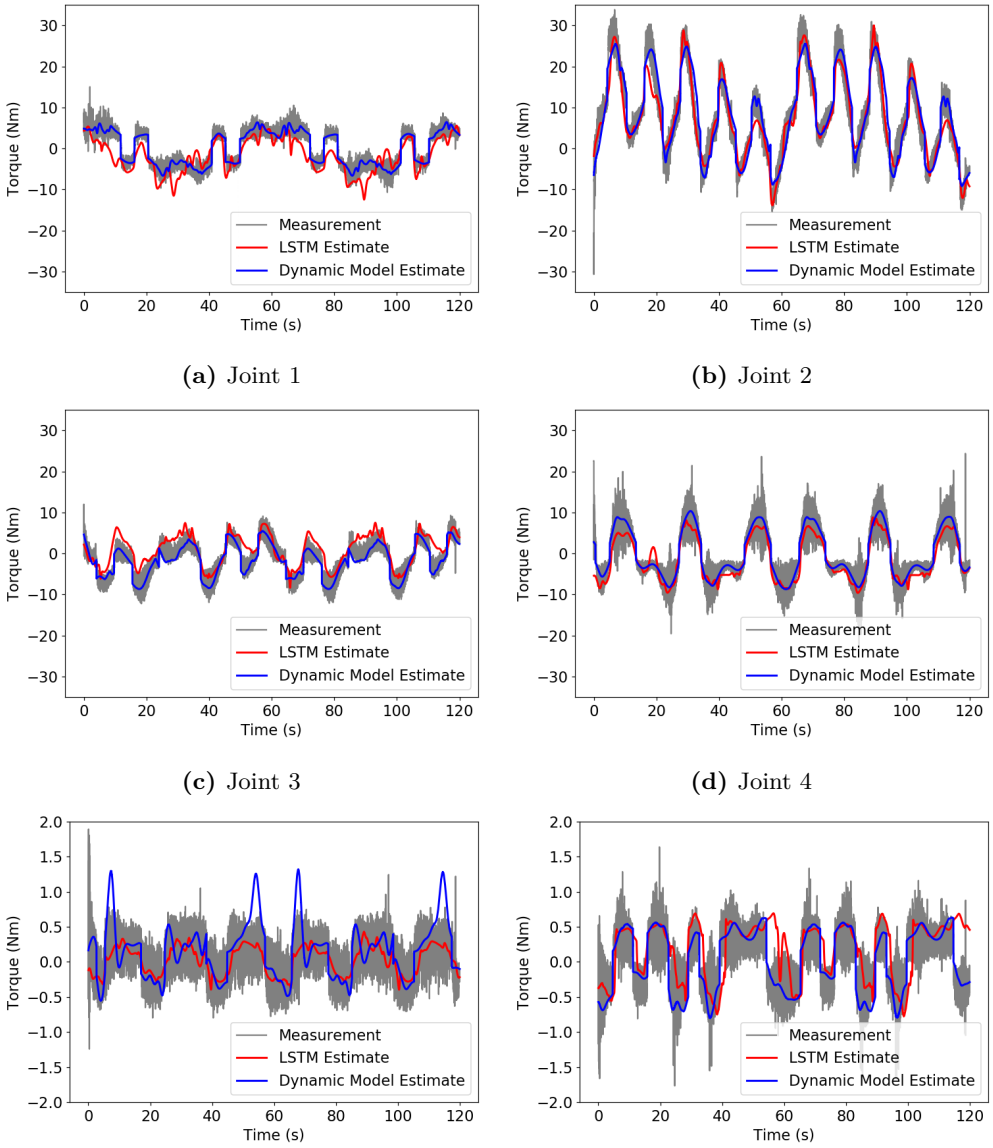
Overall, the amount of added weight does not seem to have a clear influence on the accuracy. On joint 2, the joint with the largest external torques acting on it, the LSTM performs significantly better than the momentum observer. There is no clear preference for joint 3, although the momentum observer shows large errors in the unweighted case. The momentum observer gives better results on joints 4, 5, and 6.

While both methods give estimates close to the correct value for many data points, significant errors remain as well. Further work is required to make the methods accurate enough for use in precise control. At the moment, it must be concluded that the method is not yet accurate enough for practical use and does not yet allow for safe monitoring of external torques.

Parameter	Value	Parameter	Value
I_{a1}	$-7.42e^{-2}$	L_{4xy}	$-1.17e^{-1}$
I_{a3}	-6.66	L_{4xz}	$-9.72e^{-2}$
I_{a4}	$-5.18e^{-1}$	$L_{4yy} + L_{5zz}$	$-5.59e^{-1}$
I_{a5}	$2.64e^{-1}$	L_{4yz}	$1.06e^{-1}$
I_{a6}	$2.60e^{-1}$	L_{5xy}	$-1.42e^{-1}$
L_{2xy}	$3.19e^{-1}$	L_{5xz}	$3.57e^{-2}$
L_{2yz}	$-4.13e^{-1}$	L_{5yz}	$-3.90e^{-2}$
L_{3xy}	$1.96e^{-1}$	$L_{6xx} - L_{6yy}$	$1.27e^{-1}$
L_{3xz}	$-2.95e^{-1}$	L_{6xy}	$5.42e^{-4}$
L_{3yz}	$6.20e^{-1}$	L_{6xz}	$-3.93e^{-2}$
$L_{4xx} - L_{4zz} + L_{5zz}$	1.79	L_{6yz}	$-9.10e^{-2}$
L_{6zz}	$-8.36e^{-2}$	f_{v5}	$-2.07e^{-1}$
f_{c1}	2.87	f_{v6}	$-3.17e^{-1}$
f_{c2}	3.90	l_{1x}	$-4.37e^{-1}$
f_{c3}	2.27	$l_{1z} - l_{2z} - l_{3y}$	$-1.75e^{-2}$
f_{c4}	2.19	l_{2y}	$3.96e^{-2}$
f_{c5}	$-2.13e^{-1}$	l_{3x}	$3.48e^{-2}$
f_{c6}	$-3.10e^{-1}$	l_{4x}	$1.48e^{-2}$
f_{v1}	$1.18e^{-1}$	$l_{4z} + l_{5y}$	$-1.67e^{-2}$
f_{v2}	$2.53e^{-1}$	l_{5x}	$-6.50e^{-3}$
f_{v3}	9.65	l_{6x}	$1.23e^{-2}$
f_{v4}	7.63	l_{6y}	$-1.30e^{-2}$

Parameter	Value
$I_{a2} + L_{2zz} - 0.1225m_2 - 0.1225m_3 - 0.1225m_4 - 0.1225m_5 - 0.1225m_6$	$-1.33e^{-1}$
$L_{1yy} + L_{2yy} + L_{3zz} - 0.1225m_2 - 0.1225m_3 - 0.1225m_1 - 0.1225m_2 - 0.1225m_3$	-1.05
$L_{2xx} - L_{2yy} + 0.1225m_2 + 0.1225m_3 + 0.1225m_1 + 0.1225m_2 + 0.1225m_3$	$-5.65e^{-1}$
$L_{2xz} - 0.35l_{2z} - 0.35l_{3y}$	$2.41e^{-1}$
$L_{3xx} - L_{3zz} + L_{4zz} - 0.61l_{4y} + 0.093025m_1 + 0.093025m_2 + 0.093025m_3$	-1.99
$L_{3yy} + L_{4zz} - 0.61l_{4y} + 0.093025m_1 + 0.093025m_2 + 0.093025m_3$	6.01
$L_{5xx} - L_{5zz} + L_{6yy} + 0.15l_{6z} + 0.005625m_3$	$5.21e^{-1}$
$L_{5yy} + L_{6yy} + 0.15l_{6z} + 0.005625m_3$	$-2.17e^{-1}$
$l_{2x} + 0.35m_2 + 0.35m_3 + 0.35m_4 + 0.35m_5 + 0.35m_6$	$-8.22e^{-1}$
$l_{3z} - l_{4y} + 0.305m_4 + 0.305m_2 + 0.305m_6$	-1.62
$l_{5z} + l_{6z} + 0.075m_6$	$-6.66e^{-2}$

Table 4.4. Base parameters and their identified values

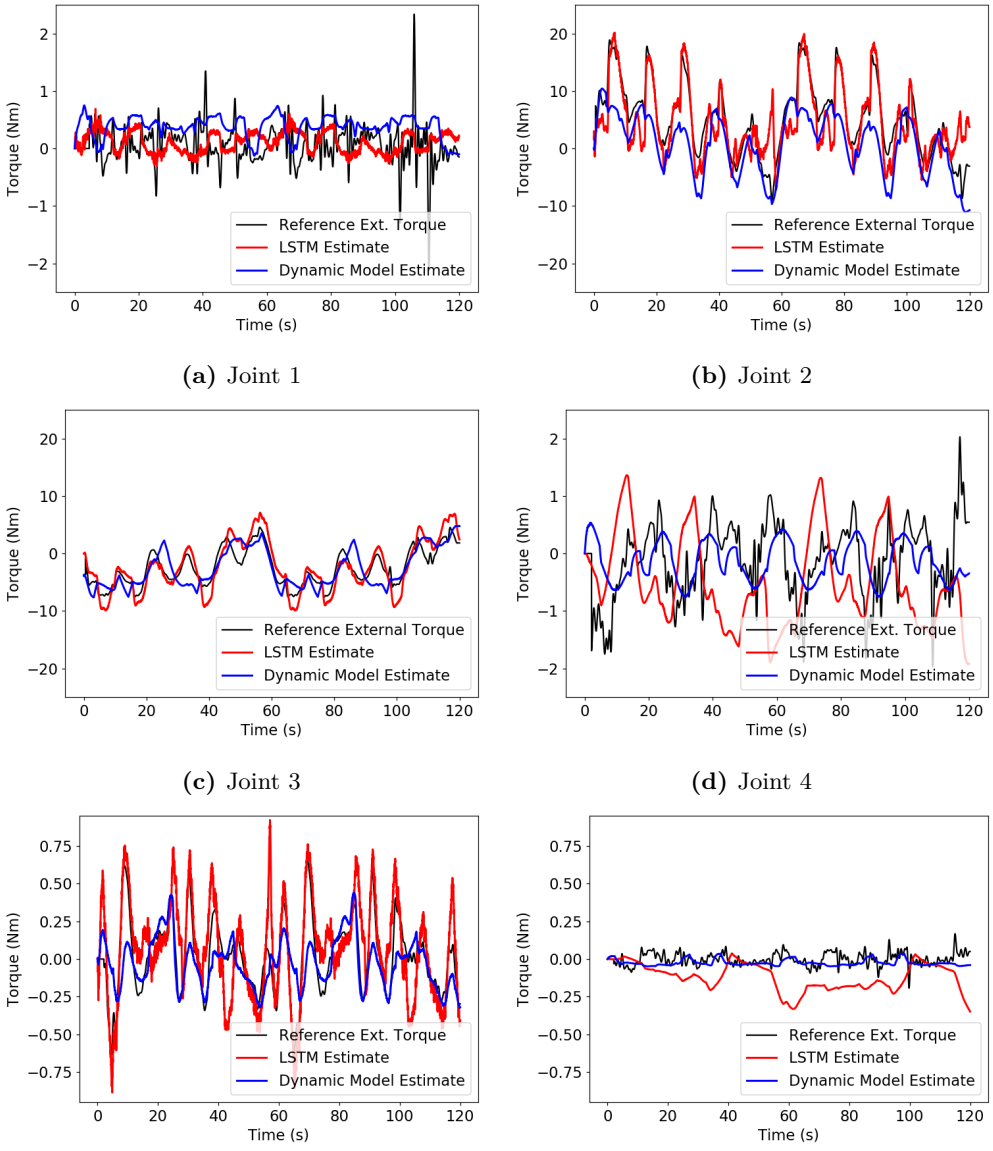


(a) Joint 1 **(b)** Joint 2

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

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

Figure 4.10. Validation of the estimated torques of the dynamic model and the LSTM network on a 120 s test trajectory.



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

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

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

Figure 4.11. Estimated and reference external torques with an attached weight of 3.25 kg.

Summary. This chapter describes how constraints are used to define requirements for reactive robot control, and how a control signal is found during the execution using an optimization procedure.

Publication. Parts of this chapter have been published previously [137].

5

Constraint-based Robot Control

5.1 Related Work **48**
5.2 Definition of Constraints **50**
5.3 Handling Dependencies on the Environment: Constraint Rules and Tasks **54**
5.3.1 Inputs to Constraint Rules 55
5.3.2 Types of Constraint Rules 57
5.4 From Constraints to Velocity Bounds: Constraint Controllers **67**
5.4.1 The `Follow`-Controller 68
5.4.2 The `Limit`-Controller 69
5.4.3 The `Stopping`-Controller 71
5.4.4 The `Hybrid`-Controller 72
5.5 Tasks and Actions **73**
5.5.1 Specification of Tasks 73
5.5.2 Definition of Actions 75
5.6 Finding an Optimal Control Signal **80**
5.6.1 Formulation of a Quadratic Optimization Problem . . 80
5.7 Evaluation **85**
5.7.1 Linear Motions to Cartesian Targets 85
5.7.2 Null-space Motion 86
5.7.3 Reaction to Obstacles 88
5.7.4 Real-time Requirements 94

In order to safely and efficiently perform its required tasks in a dynamic environment, the mobile manipulator must be able to react to the changes in the environment. Besides the ability to perceive the environment, this also requires the control system to generate online motions that adapt to the changing environment. In the use case example of an object handover to a person, the robot should continuously adapt his own position to that of the person, which can not be completely planned in advance. Similarly, the gripper pose has to take the current

pose of the person's hand into account. While continuously adapting to these dynamic elements, it is of even higher importance that the robot does not cause any safety hazards: it is strictly necessary that the robot does not collide with himself, the person, or other obstacles in the environment. This illustrates the need for a method of introducing priorities between requirements: the requirement for collision avoidance should not be circumvented by the requirement to perform an object handover, as a failed handover is much less severe than a collision. Therefore, the relative importance of the individual requirements needs to be expressed and respected by the controller. The specification of an intended robot behavior in a dynamic environment can become complex, especially for robots with many degrees of freedom. Geometric constraints are a way in which the joint positions as well as the motion in Cartesian space can be specified relatively intuitively and can be easily extended with priorities and weights to resolve conflicting constraints.

This chapter is structured as follows: It begins with an overview of related approaches in section 5.1. Afterward, section 5.2 details the specification of robot tasks based on constraints. Section 5.3 introduces *constraint rules*, which serve to parametrize the geometric constraints to the current state of the dynamic environment. The constraints still need to be converted into robot motions, therefore section 5.4 describes the concept of *Constraint Controllers*, which calculate velocity bounds for their respective controlled robot links based on the current constraint values. The velocity bounds are then used to formulate an optimization problem which is solved in every control cycle to find an optimal control signal. The details of this are described in section 5.6. To conclude the chapter, the approach is evaluated with various experiments on the robot in section 5.7.

5.1 Related Work

Reactive control of robots in human-robot-collaboration scenarios or other dynamic environments is an active area of research. Formalizing the requirements for safe robot motions as a set of constraints with different priorities, that are used in an optimization process is a common approach to this problem.

Often, publications on robot control in this area consider the safe execution of plans, rather than purely reactive control. For example, Faroni et al. use velocity scaling and redundancy-optimizing inverse kinematics [31] for safe online modification of plans. This means that the planning problem and the execution of the resulting plans are treated as two separate problems. However, if the design of the planner and the controller are not compatible, the results may be suboptimal. There is also the added user effort of creating two separate specifications: one for the planning problem and another for the controller executing the plan. The present work considers this topic in a later chapter, namely chapter 7. At this point in this thesis, the focus lies on the related work for generating reactive motions without a pre-existing motion plan.

Many research works with this approach design their control architecture specifically for the respective robot kinematics, environment, and robot task of their respective scenario. Furthermore, most works in this area are focused purely on the controller performance with no regard for flexibility, modularity, extensibility, or ease of specification. Often, the different constraints are statically defined for the singular use case of the respective work [145, 41].

The research into the topic of the design of well-performing and safe controllers is the foundation that the present work builds on. In order to bring these concepts into practice, a software architecture to facilitate the design of new robot behaviors is required.

Related work that deals specifically with the control of wheeled mobile manipulators often simplifies the problem into control of the arm and control of the base [64, 16]. This makes the control considerably easier but limits the ability to make use of the robot's redundancy, as well as restricting the fluidity of human-robot-interactions. Considering the example scenario of an object handover, temporal separation of arm and base motions means that the robot would have to stop the arm from moving as soon as the interaction partner moves to a position that the arm can no longer reach, move the base closer, and begin moving the arm again.

Therefore, whole-body-control (WBC) methods have been developed, for mobile manipulators and other types of redundant robots. These methods control all degrees of freedom at the same time. A prominent work that controls redundant robots reactively by considering a set of sub-tasks is the iTaSC framework [25, 26], with a Jacobian-pseudoinverse-based control scheme. Other works have adopted methods inspired by this, e.g. [99]. The Stack-of-Tasks [92], OpenSoT [58] and eTaSL/eTC [1] are other prominent implementations, that use a quadratic programming (QP) formulation to find control signals instead. Our method of calculating joint velocities is based on these works.

A lot of work on constraint-based robot control is focused on legged, free-floating systems such as walking humanoids. These are typically force-controlled systems that deal with challenging dynamics problems, such as walking and balance. These are not relevant for a dynamically stable, velocity-controlled wheeled mobile manipulator such as ours. The downside is, of course, the limitation of wheeled robots regarding uneven terrain, such as stairs. These complex requirements and formal foundations of most works on constraint-based control have had the side effect of making the solutions hard to use for practical applications.

In 2020, a workshop at the International Conference on Intelligent Robots and Systems (IROS) was organized with the specific aim of closing the gap between academic experimental frameworks for constraint-based control on the one hand, and industry practice on the other hand [61]. In the years since, real-world adoption of constraint-based approaches has not seen a significant increase, and the aforementioned gap still exists. The goal of the present work is therefore not focused on furthering the academic understanding of constraint-based control, but

more on the facilitation of the adoption of constraint-based control methods for application developers using popular middlewares such as ROS, as well as keeping the functionality extensible and modular to adapt it to different needs.

The Framework ControlIt! [36, 37] is similar to the presented work in the regard that it attempts a modular and easy-to-use architecture for various whole-body-control algorithms in a ROS environment. However, ControlIt! does not currently have the ability to handle geometric primitives and dynamic environments, instead being focused on explicit Cartesian poses and joint positions that have to be determined through external means. It also does not seem to be actively maintained at the time of writing.

Constraint-based control has also been used in the specification of manipulation tasks, i. e. handling objects. Constraint specifications offer the benefit of compact specifications of complex manipulation tasks. The research works do mostly not consider dynamic environments and interaction.

Phoon et al. [111, 126, 125] present a framework for constraint-based control of mobile manipulators with the goal of easy specification. Their focus lies on sequential manipulation tasks, which they manage to execute in a time-optimal way while smoothly interpolating between sequence steps. Contrary to our approach, they consider manipulation in a static environment, with speed being the main aim. Their work does not consider human-robot interaction or robustness in dynamic environments. In a similar work, Halt et al. [52] built upon iTaSC to present an intuitive system of programming manufacturing tasks.

In conclusion, our work in this chapter differentiates itself from the related work on this topic in the following ways:

- The principles of constraint-based robot control are applied to human-robot interaction in dynamic environments. Instead of defining constraints between defined parts of a workspace, the specification focuses on the reactions to dynamic environment changes.
- Its modular, extensible, and ROS-enabled architecture makes customization and specification simple.
- The design enables the use of the same specification for reactive control, sampling-based planning, and the safe execution of plans. Details of this are described in the following chapter.

5.2 Definition of Constraints

Before describing the use of constraints in more detail, this section defines what is understood as a constraint in the context of this work. We consider only *geometric* constraints. Geometric constraints are constraints that can be evaluated on a robot configuration $\mathbf{q}$. This excludes constraints on the robot's velocity, acceleration, torques, et cetera, but allows expressing constraints on the robot's joint positions and link positions in Cartesian space. It should be noted that the fact that constraints on velocities are not considered does not mean that

the control framework does not have the ability to generate robot motions with well-defined and user-specified velocities. However, the velocities are computed by the *constraint controllers* based on the current value of the geometric constraints. For more details on this, see section 5.4 below. In this section, the focus lies on the geometric constraints themselves.

A (geometric) constraint is formally described by its corresponding *constraint function:*

$$\begin{aligned} &f : \mathbb{R}^n \to \mathbb{R}^m \\ &f(\mathbf{q}) = \mathbf{c} \end{aligned} \tag{5.1}$$

We refer to the value $\mathbf{c}$ as the *constraint value* for the robot configuration $\mathbf{q}$. The constraint is considered satisfied if and only if $f(\mathbf{q}) = 0$. Consequently, any value $\mathbf{c} \neq 0$ means that the constraint is not satisfied. In general, the constraint functions are assumed to be continuous and encode a distance metric for the cases where $f(\mathbf{q}) \neq 0$, i. e. the value of $f(\mathbf{q})$ expresses how far from a satisfying configuration $\mathbf{q}$ is. Therefore, the value *constraint value* $\mathbf{c}$ is alternatively called the constraint's *error value*.

In equation (5.1), n is the robots' number of degrees of freedom. $m \in \mathbb{N} \setminus \{0\}$ is a positive natural number called the *constraint dimension*. If $m > n$, the system the constraint applies to is called *over-constrained*: There are more constraint dimensions than available degrees of freedom. Conversely, a constrained system with $m < n$ is *under-constrained*. An under-constrained system has redundant degrees of freedom that are not determined by the constraint. A classical example of an under-constrained robot system is a seven-joint manipulator, whose end-effector is constrained to a Cartesian pose. This corresponds to a 6-dimensional constraint $m = 6$, while the robot has seven degrees of freedom $n = 7$. Therefore, the robot has one redundant degree of freedom. Methods of *redundancy resolution* are then required to determine the most suitable values for the redundant degree of freedom. On the other hand, over-constrained systems might not have solutions that satisfy the constraint completely and a method to resolve the conflicts is required.

As all constraints are functions on the robot configurations, it is simple to combine two constraints into one, as defined in equation (5.2) for two constraints f_1 and f_2. The inverse operation, splitting a constraint with $m > 1$ into two, is equally simple.

$$\begin{aligned} f_1 &: \mathbb{R}^n \to \mathbb{R}^{m_1} \\ f_2 &: \mathbb{R}^n \to \mathbb{R}^{m_2} \\ f_{1,2} &: \mathbb{R}^n \to \mathbb{R}^{m_1+m_2} \\ f_{1,2}(\mathbf{q}) &= \left(f_1(\mathbf{q})^T, f_2(\mathbf{q})^T\right)^T \end{aligned} \tag{5.2}$$

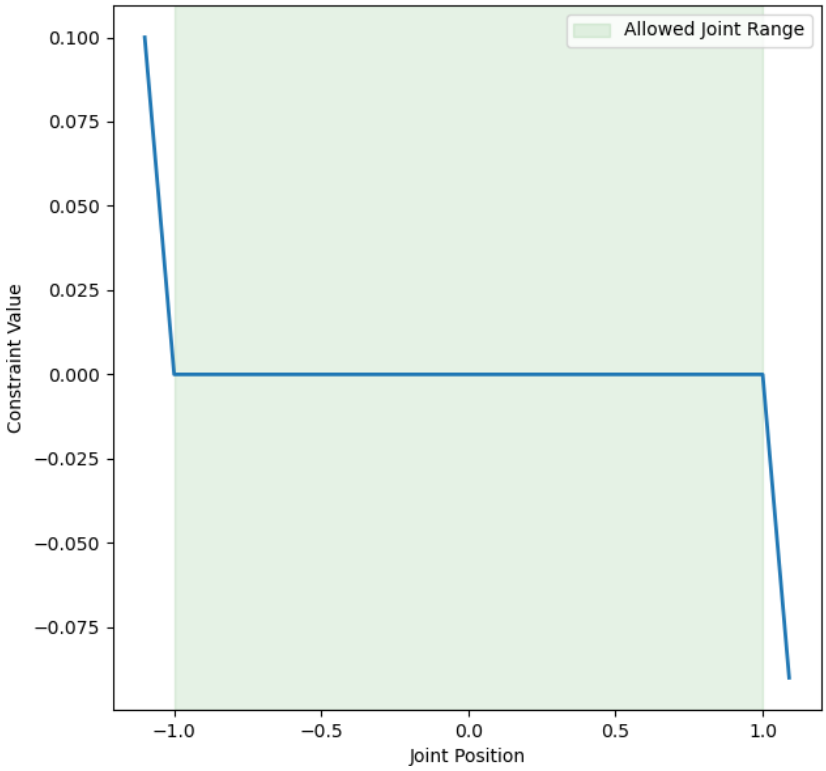


Figure 5.1. Plot of a joint limit constraint for a single joint with limits at -1 and 1.

As a simple, concrete, example, a constraint to keep the first arm joint $\mathbf{q}_1$ at position 0 can be expressed by the following constraint function, whose value will be the distance of the current position to the target 0:

$$\begin{aligned} &f_0 : \mathbb{R}^n \to \mathbb{R} \\ &f_0(\mathbf{q}) = 0 - \mathbf{q}_1 \end{aligned} \tag{5.3}$$

A constraint to keep the joint in given limits $\mathbf{q_{max}}, \mathbf{q_{min}} \in \mathbb{R}^n$ can be expressed by the following constraint function:

$$\begin{aligned} &f_{limits} : \mathbb{R}^n \to \mathbb{R}^n \\ &f_{limits}(\mathbf{q}) = \min(\mathbf{q_{max}} - \mathbf{q}, 0) + \max(\mathbf{q_{min}} - \mathbf{q}, 0) \end{aligned} \tag{5.4}$$

Here, min and max are the element-wise minimum and maximum functions. The function will have the value 0 as long as all joints are within their limits. If a joint moves outside the limits, the corresponding constraint value will be the signed distance to the nearest position within in the limits. Figure 5.1 shows a plot of the function, for the example case of a single joint with limits at -1.0 and 1.0.

As a further example, this time in Cartesian space, a constraint to keep the end-effector at a given position p_{target} is given in equation (5.5). The orientation is not considered here, so the constraint is three-dimensional with the three dimensions x, y, and z. In this equation, $p_{EE}(\mathbf{q})$ is the position of the end effector given the joint positions $\mathbf{q}$, as calculated by the direct kinematics function.

$$\begin{aligned} &f_{cartesian} : \mathbb{R}^n \to \mathbb{R}^3 \\ &f_{cartesian}(\mathbf{q}) = p_{target} - p_{EE}(\mathbf{q}) \end{aligned} \tag{5.5}$$

Other geometric constraints can be defined in similar ways. Given that constraints can easily be combined or split into two, it can be ambiguous what is meant when talking about "multiple constraints" or "a single constraint with multiple dimensions". The terminology is subject to some interpretation, and the related literature is inconsistent in that regard. In the most basic case, each constraint dimension can be understood as a single constraint. With this interpretation, the example for equation (5.5) would define three constraints, one for each axis of Cartesian space. Other interpretations assume that all applied constraints are combined into one function, so that in effect, the system is always understood to have only one single constraint.

While both interpretations are mathematically equivalent, both of the above interpretations can lead to some counter-intuitiveness when describing and specifying constraints. If each degree of freedom is considered as an individual constraint, the description of the constraints is too fine-grained to be intuitive. For example, keeping within the joint limits while moving the end-effector to a pose in Cartesian space would already lead to $9 + 6 = 15$ individual constraints, in the case of our mobile manipulator. It is assumed that it is more intuitive to most people to describe this as 2 constraints, the joint limits and the Cartesian pose. Interpreting this as a single constraint is similarly counter-intuitive.

In conclusion, we generally use the following convention when describing constraints: Individual constraints can be either constraints in joint space, or constraints in Cartesian space. A Cartesian constraint can have up to six dimensions, all of which refer to the same robot link and the same reference coordinate system. Each constraint dimension represents a different degree of freedom of Cartesian space. Joint-space constraints can have up to n dimensions, each of which represents a different robot joint. Furthermore, each individual constraint is expected to represent a shared requirement on all of its degrees of freedom. Consider an example with two requirements: orienting the gripper toward the person, and keeping the gripper at a certain height. This could be represented as a single Cartesian constraint with four degrees of freedom, three for the orientation and one for the height. This should however be formulated as two constraints, one for the orientation and one for the height. The reasoning behind this is that constraints will, later on, be used with different priorities, and it should be avoided to have to split constraints after their definition in order to provide the different dimensions with different priorities.

The definition of constraints provided here is still missing several key elements to generate sensible robot motions. The rest of this chapter deals with the following concepts: *Constraint rules* set the parameters of the constraints to reflect the state of the dynamic environment. *Constraint controllers* define how a constraint value influences the allowed motions of the robot. They are brought together in the concept of *tasks*. Finally, the *solver* calculates an optimal control signal from the active tasks and the current robot state.

5.3 Handling Dependencies on the Environment: Constraint Rules and Tasks

The definition of constraints from the above section 5.2 defines constraints as functions on the robot configuration $\mathbf{q}$. Therefore, the constraints are evaluated only on the current positions of the robot's degrees of freedom. However, as soon as the robot is supposed to react to a dynamic environment, this definition falls short, as the constraint functions do not contain any dependency on the current environment. In the example of an object handover, the robot must move its gripper toward the hand of the person. The information about the pose of the hand of the person can, of course, not be statically encoded at the design time of the constraints, as it is only known online during the execution of the handover, and continuously changing. This means that some method of reflecting the changing environment in the constraint functions is needed.

Some of the first constraint-based robot control frameworks, such as eTasL/eTC [1] and iTasC [25], use an approach based on *virtual kinematic chains*. All relevant information from the environment is encoded in geometric frames, and the transformations between the frames are connected by virtual kinematic chains. In effect, the definition of a *robot configuration* is extended to contain all relevant information from the environment as well, and the constraint functions are defined on this extended configuration. They provide a strict procedure to be followed to create a sound constraint specification. This strict procedure allows various types of mathematical analyses to be performed on the constraint model and, among other things, allows controlling force-controlled, velocity-controlled and position-controlled robots with the same constraint specification.

However, the formal rigor of these methods comes at the cost of flexibility and intuitiveness of the specification. Since everything has to be broken down into explicit relationships between geometrical frames, constraints on complex geometrical shapes, unknown numbers of obstacles, or unstructured sensor data are difficult to define. Furthermore, this blending of environment data and robot configuration complicates path planning with a constraint specification. This is explained in more detail in chapter 6.

For these reasons, we chose another approach to introduce the dependencies on the environment to the constraint functions. Instead of expanding the constraint functions with virtual kinematic chains, we introduce a layer above the constraint functions. This layer updates the parameters of the constraint functions based

on the current state of the environment. This is also executed in every control cycle, to ensure a fast reaction to environmental changes. Using this architecture, the environment can be separated from the constraints on the robot. Once the parameters are set, the constraints themselves fully encapsulate the requirements of the robot motion, and no further knowledge of the environment is required. Among other advantages, this facilitates planning for a given environment, which will be described in the later chapter 6. Figure 5.2 shows a basic overview of the control architecture.

Constraint rules use the current `Context` as described in section 4.2 to set the parameters of the constraint functions. The `Context` contains the environment data and the static information about the robot. The constraint functions calculate the constraint error values from the current robot state. The constraint values are then used by the *constraint controllers* to calculate velocity bounds. These velocity bounds are used in the formulation of a *quadratic programming* (QP) problem. The *QP Solver* finds an optimal control command while attempting to respect all velocity bounds according to their weights and priorities. The control command is finally sent to the hardware controllers, which realize it on the robot hardware and report back the current robot state.

Thus, creating robot motions from an abstract requirement needs the following elements:

- The constraint function
- The constraint rule
- The constraint controller
- The necessary inputs

An instance that brings these elements together is called a `Task`. They are described in more detail in section 5.5.

5.3.1 Inputs to Constraint Rules

The *inputs* encapsulate all the dynamic information that is needed to set the parameters of the constraint function. Inputs can be taken from the `Context`, or be defined as separate command inputs provided by an external mechanism or configuration. Typically, these inputs are used to define the targets of the constraint rule. For example, a constraint rule moving the gripper to a pose in Cartesian space needs to know the target Cartesian pose. This target pose might be determined online from a perception system, might be statically pre-defined in a configuration file, or might correspond to a frame in the current `Context`. To separate the definition of a constraint rule from the definition of the targets, these are encapsulated in the `ConstraintRuleInput` class. Before a constraint rule can be used, all of its inputs must be *ready*, e.g. the Cartesian target pose needs to be known. This is checked to avoid unexpected behavior due to missing data or invalid configuration.

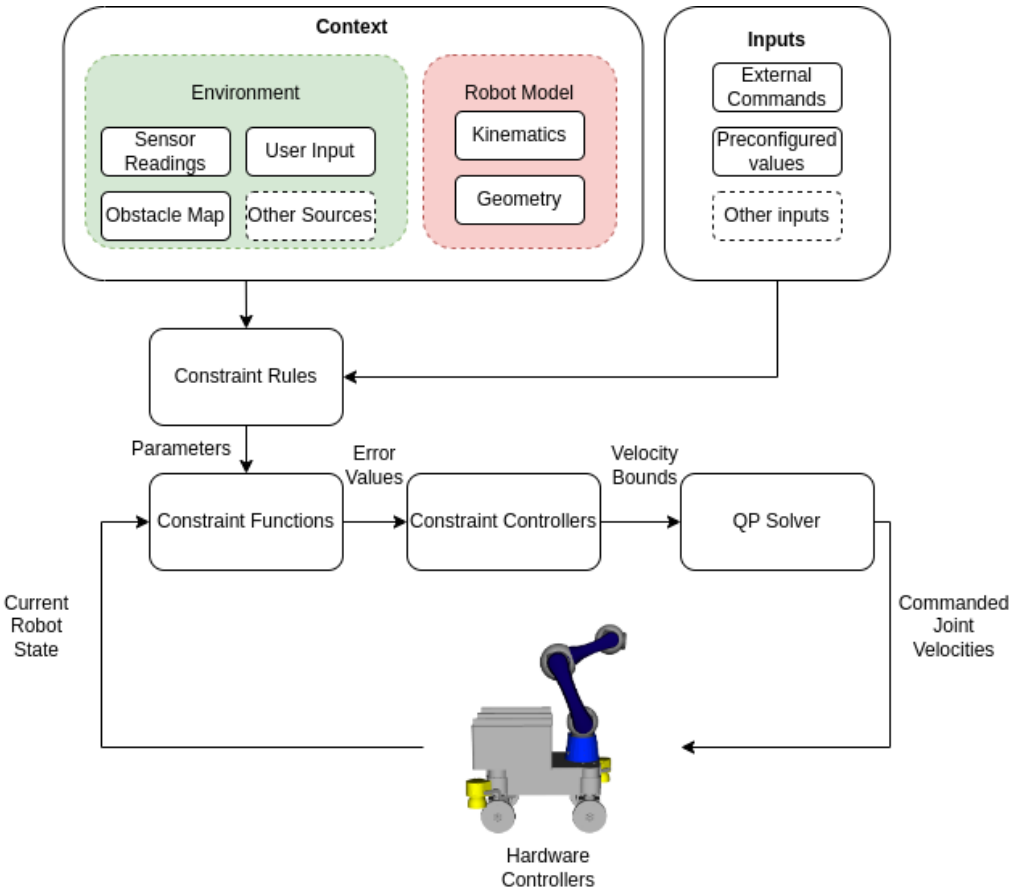


Figure 5.2. Overview of the concepts in our control architecture.

The method `isReady()` checks whether all inputs are able to produce the required data and that the data passes initial validity checks. For example, joint limits are checked to define a valid range. The method `update()` sets the input value to the newest available value. This encapsulates any dependencies on the used middleware. The different types of input that have been implemented are shown in figure 5.3 and briefly explained below.

Sub-types of the `CartesianPoseInput` are used for rules that are dependent on a pose in Cartesian space. The pose can be statically configured using the `StaticCartesianPoseInput` type. This can also be specified to be relative to a moving frame and thus has some flexibility. For example, it can be used to specify a pose relative to the robot's base. The `TopicCartesianPoseInput` type is updated through a ROS topic. The `FrameCartesianPose` takes the Cartesian pose from the pose of a frame, which must be known at run-time, the frame must exist in the `FrameBuffer` of the `Context`. As a further specialization, the `CollisionGeometryCartesianPoseInput` requires a frame with an associated collision geometry.

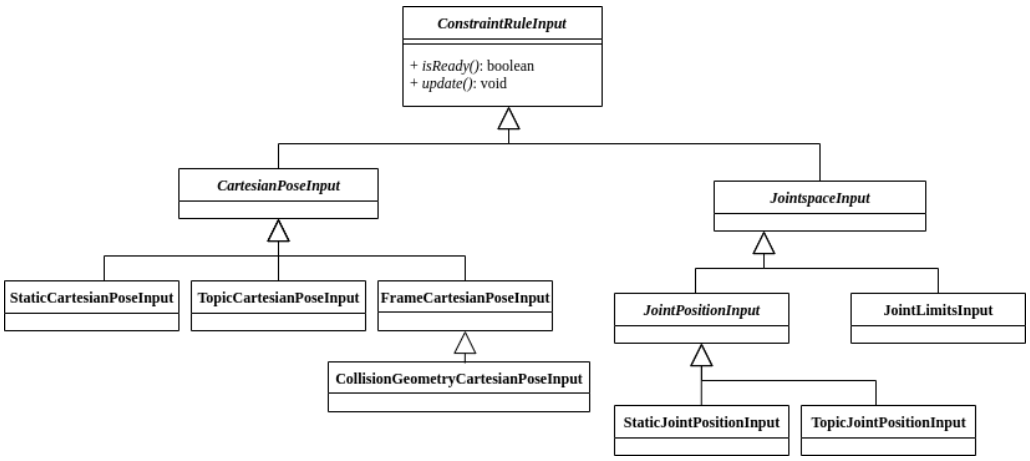


Figure 5.3. Class diagram of the implemented input types rules.

The abstract type `JointSpaceInput` stands for input in the joint space and is extended by the type `JointPositionInput`, which can, in turn be statically configured (`StaticJointPositionInput`) or updated through a ROS topic (`TopicJointPosition`). Lastly, the `JointLimitsInput` type is used to specify a range of joint positions, using an upper and lower limit.

Besides these explicitly specified inputs, the robot's kinematics and geometry, and the obstacles of the environment are included in the context model and globally available to all constraint rules. Moreover, the current robot configuration $\mathbf{q}$ is also available globally, as it forms the basis of the controller calculations.

5.3.2 Types of Constraint Rules

In this section, the different available types of constraint rules are presented. The implemented types are shown in a class diagram in figure 5.4. In the following, each of them is described in detail. The definition of a constraint rule generally consists of four parts:

- **Configuration Parameters**, used to adapt the definition to different use cases. The values are specified in the task configuration file.
- **Required `Inputs`**, that provide all necessary dynamic data, that might only be available online.
- **Constraint Function**, defined using the inputs and configuration parameters.
- **Constraint Jacobian**, the matrix defining how movements of the joints will affect the constraint. Used, among other things, to define the relationship between the calculated velocity bounds and the joint velocities.

Each constraint rule is either a Cartesian rule, or a joint space rule, and consequently each rule is a subtype of one of the abstract classes `CartesianConstraintRule` or `JointSpaceConstraintRule`.

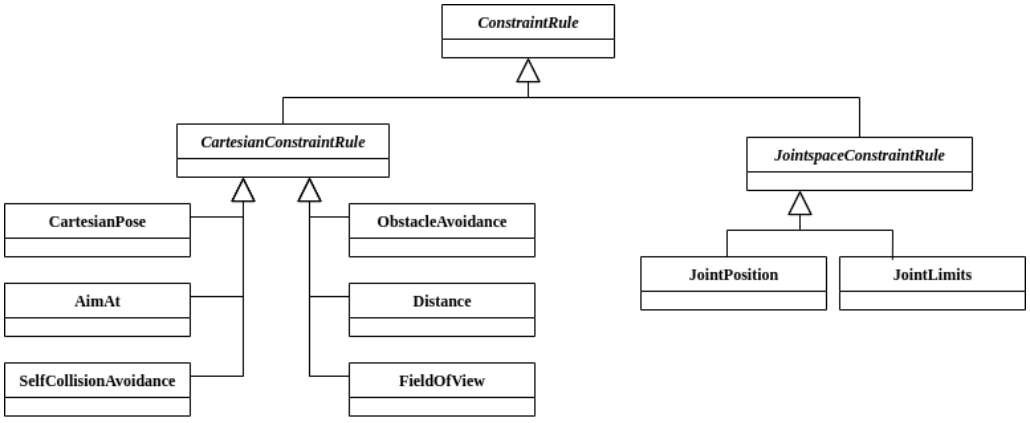


Figure 5.4. Class diagram of the implemented constraint rules.

Constraint Rules in Joint Space

All constraint rules that are sub-types of `JointSpaceConstraintRule` have in common that the corresponding constraint Jacobian is simply the identity matrix, $\mathbf{J}_{JointSpace} = \mathbf{I}_n \in \mathbb{R}^{n \times n}$, where n is the number of controlled joints. For constraints in joint space, no conversion between velocities is necessary, and the Jacobian does not have to specify a further relationship between joint velocities and the constraint. The currently implemented constraint rules in the joint space are the following:

JointPosition This constraint rule type expresses the requirement to move the robot's joints to a target position. As such, it is the simplest example of a constraint rule. An example of a practical use is to move the arm to a pre-defined transport position. With a lower priority, it is also useful to guide the robot toward desired solutions, for example, to prefer elbow-up positions over elbow-down positions. This task requires a single input of type `JointPositionInput`. It specifies the target joint positions $\mathbf{q}_{target}$. The constraint function calculates the error between the target and the current configuration: $f_{JointPosition}(\mathbf{q}) = \mathbf{q}_{target} - \mathbf{q}$. This type of constraint rule has no further configuration parameters, all the possible customization is done through parameters in the task and the controller.

JointLimits This constraint rule keeps the joint positions in the range of an interval of joint values. The range is given as an input of type `JointLimitInput`, specifying the upper ($\mathbf{q_{max}} \in \mathbb{R}^n$) and lower ($\mathbf{q_{min}} \in \mathbb{R}^n$) limits. Besides the obvious example of always keeping the joints within the hardware limits, it can also be used to define preferred ranges with a lower priority. This type of constraint rule has the joint limits as a single input, and no configuration parameters. The constraint function is given as the following function:

$$f_{limits}(\mathbf{q}) = \min(\mathbf{q_{max}} - \mathbf{q}, 0) + \max(\mathbf{q_{min}} - \mathbf{q}, 0)$$

In the above equation, min and max are the element-wise minimum and maximum functions. If the current joint position is within the limits, the constraint value is zero, otherwise the signed distance to the limits is returned. The case that a single joint is both above and below the valid range at the same time can not occur for a valid `JointLimitInput`.

Constraint Rules in Cartesian Space

All Cartesian constraint rules have in common that they require configuration parameters to define which link of the robot they control. The controlled link is referred to as ϕ_{control}. They also require a definition of the *reference frame* $\phi_{\text{reference}}$. The Cartesian requirements are then assumed to be relative to this frame. In general, all Cartesian constraint rules use the following form for the constraint Jacobian:

$$\mathbf{J}_{\text{Cartesian}}(\mathbf{q}) = \mathbf{J}^{\text{geom}}_{\phi_{\text{control}}}(\mathbf{q}) - \mathbf{J}^{\text{geom}}_{\phi_{\text{reference}}}(\mathbf{q})$$

In the case that $\phi_{\text{reference}}$ is not a part of the robot, but an uncontrolled frame in the environment, its geometric Jacobian $\mathbf{J}^{\text{geom}}_{\phi_{\text{reference}}}(\mathbf{q})$ will be zero, as the joints do not influence it. In this case, the constraint Jacobian is just equal to the geometric Jacobian of the controlled frame, $\mathbf{J}^{\text{geom}}_{\phi_{\text{control}}}(\mathbf{q})$. The controlled link is necessarily a controlled robot frame, and, consequently, this Jacobian can never be zero. In the case that both $\phi_{\text{reference}}$ and ϕ_{control} are controlled links of the robot, both their Jacobians must be considered to model the relative motions to each other. An example where this is used is self-collision avoidance between two links of the robot. Here, it is not the absolute movement of the links that is relevant, but only their movement in relation to each other. Besides this, the relative movements can also be used to specify other types of position requirements, for example, to keep the gripper above the platform, or to avoid moving the elbow below the end-effector. Configurations where $\phi_{\text{reference}} = \phi_{\text{control}}$ are invalid. The subtypes of Cartesian constraint rules are further described below.

Cartesian Pose This constraint rule expresses the requirement of a given robot link to be positioned at a target Cartesian pose. It requires an `Input` of type `CartesianPoseInput`, defining the target pose p_{target}.

The constraint function is concisely written as follows:

$$f_{CartesianPose}(\mathbf{q}) = \begin{pmatrix} x_t - x \\ y_t - y \\ z_t - z \\ \sigma_x \\ \sigma_y \\ \sigma_z \end{pmatrix}$$

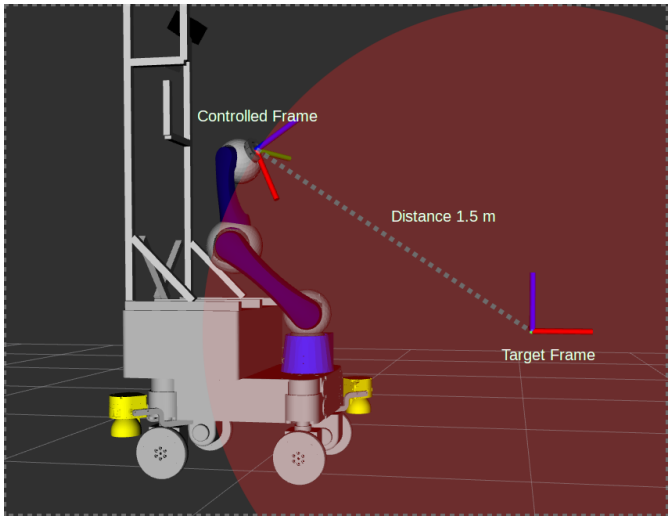


Figure 5.5. Illustration of the Cartesian distance constraint rule. Here, the robot's flange is the controlled frame, which is required to stay at least 1.5 m away from the target frame.

The constraint value expresses the error in Cartesian space between the target and the current pose. Instead of a rotation matrix, the error in orientation is expressed as a rotation vector to simplify the conversion of a rotation into velocities in the constraint controllers. The vector $(x, y, z)^T$ represents the current position of the controlled frame, $(x_t, y_t, z_t)^T$ is the position component of the target pose. $\sigma = (\sigma_x, \sigma_y, \sigma_z)^T$ is the vector expressing the rotational error. It is formed by first calculating the difference rotation matrix $\mathbf{R}_e$ between the current and target orientation: $\mathbf{R}_e = \mathbf{R}^{-1}\mathbf{R}_t$. $\mathbf{R}_e$ is converted to a three-dimensional vector by converting the rotation matrix to a rotation vector σ_e. Lastly, this vector has to be rotated to be relative to the reference frame $\sigma = \mathbf{R}\sigma_e$

Cartesian Distance This rule has the intention of keeping the controlled frame at a minimum distance to the reference frame. It controls only the frame's position, not its orientation. Thus, the constraint function has three dimensions. Also, it takes only the distance between the origins of the frames into account, not any collision geometry they might have. If the distance between geometrical shapes is required instead, the `ObstacleAvoidance` or `SelfCollisionAvoidance` constraint rules defined below should be used instead. Figure 5.5 shows an illustration of this rule. The required distance d is given as a configuration parameter by the name `distance`. The constraint function has the form of the following equation:

$$f_{\text{distance}}(\mathbf{q}) = \mathbf{c} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

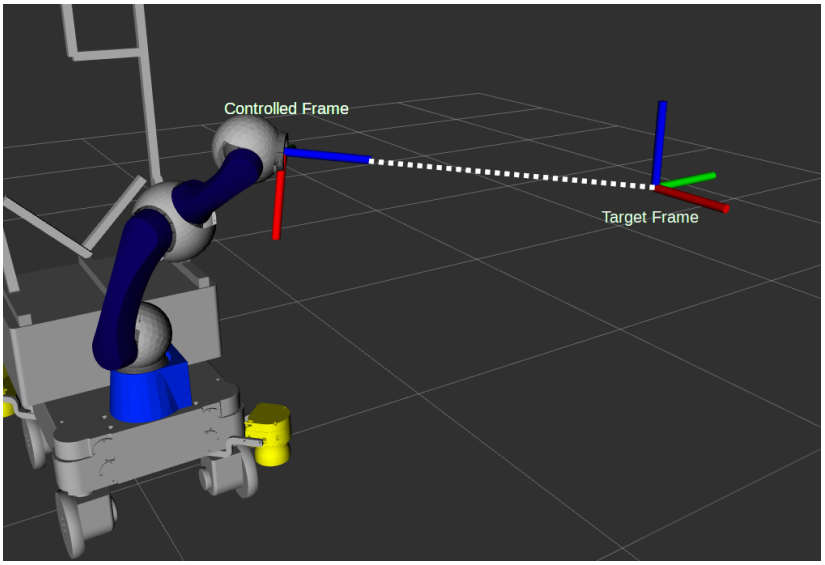


Figure 5.6. Illustration of the aiming constraint rule. The z-Axis of the controlled frame at the robot's flange is aiming at the shown target frame.

The direction of the vector $\mathbf{c}$ is along the line between ϕ_{control} and $\phi_{\text{reference}}$. Its magnitude depends on the distance between ϕ_{control} and $\phi_{\text{reference}}$. $\hat{d} = \|\phi_{\text{reference}} - \phi_{\text{control}}\|$. In conclusion, the constraint function is defined by the following equation:

$$f_{\text{distance}}(\mathbf{q}) = \mathbf{c} = \begin{cases} 0, & \hat{d} \leq d \\ (\hat{d} - d) * \frac{\phi_{\text{reference}} - \phi_{\text{control}}}{\hat{d}}, & \text{otherwise} \end{cases}$$

Aiming This constraint rule is used to let a robot frame point toward a target Cartesian pose, either in the environment or as a part of the robot itself. For example, it can be used to rotate the platform toward the person during an interaction. If the robot is supposed to shine a light from a flashlight in its gripper, this rule can be used to shine the light toward the target. This rule, on its own, influences only the rotation of the controlled frame, never its position. Therefore, only the lower three rows of the Jacobian are required here, and the constraint function has only three dimensions. Figure 5.6 illustrates this.

It requires the `axis` configuration parameter, which can be set to one of the values **x**, **y**, or **z**. This parameter determines which axis of the controlled frame should point toward the target position p_{target}. Besides, p_{target} is required as an input of type `CartesianPoseTarget` and specifies the target pose that the controlled link should aim toward.

The constraint function looks as follows:

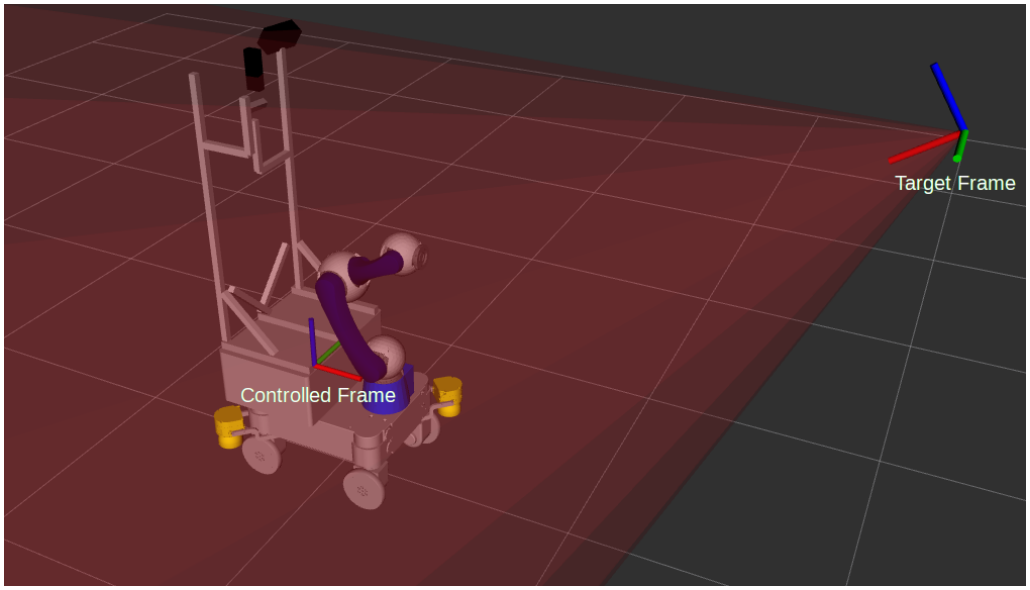


Figure 5.7. Illustration of the field-of-view constraint rule.

$$f_{\text{aiming}}(\mathbf{q}) = \begin{pmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \end{pmatrix}$$

The constraint value consists of the orientation error $\sigma = (\sigma_x, \sigma_y, \sigma_z)^T$. The target orientation is defined by the line between ϕ_{control} and $\phi_{\text{reference}}$. p_t is the normalized vector in the direction of this line:

$$p_t = \frac{\phi_{\text{control}} - \phi_{\text{reference}}}{\|\phi_{\text{control}} - \phi_{\text{reference}}\|}$$

p_c is of the normalized axis vector of ϕ_{control}, as specified in the `axis` configuration parameter. The cosine of the angle α between p_t and p_c can then be calculated as $\cos(\alpha) = p_c \cdot p_t$. An orthogonal vector to p_t and p_c is defined by $v = p_c \times p_t$. Using these values, a quaternion r is defined as $r = (v^T, \cos(\alpha))^T$. This quaternion is converted to a rotation vector σ_e. Lastly, σ_e is rotated to be relative to $\phi_{\text{reference}}$: $\sigma = \mathbf{R}\phi_{\text{control}}\sigma_e$.

Field of View This constraint rule is used to express that the controlled frame should be in the field of view of the reference frame. The field of view is specified through configuration parameters that define a cone, with its tip at the origin of the target pose. This constraint rule is typically used to have the controlled link in the field of view of a person, to create a natural interaction. The opposite, that

some external object has to be kept in the view of a camera on the robot, is also possible. The field of view here is described as a cone, with the parameters of the opening angle and its height. The cone is oriented along one of the axes of the target frame, which is given as a configuration parameter. What is not included is the requirement that the line between the frames is not occluded. This would require a model of visibility and occlusion, and can not be solved in a purely reactive fashion.

The following configuration parameters are needed for this constraint rule:

- `axis`: The axis along which the cone is oriented.
- `length`: The length of the cone L.
- `angle`: The opening angle of the cone α.

Figure 5.7 shows an example of a cone, here oriented along the x-axis. For the sake of simplicity, the calculations described here always assume the cone to be oriented along the x-axis. The other cases can be calculated in analogous ways.

The constraint function has the following form:

$$f_{\text{FoV}}(\mathbf{q}) = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The calculation of the constraint function is designed from two requirements that arise in order to be within the cone:

- The x-position of ϕ_{control} relative to $\phi_{\text{reference}}$ must be positive, but not larger than L.
- On the y-z-plane, the distance of ϕ_{control} to the cone's center line must be no larger than the radius of the conic section at the x-position. The further ϕ_{control} is from $\phi_{\text{reference}}$, the larger the cone becomes.

The first requirement is simple to express in the following form:

$$x = \begin{cases} x_c, & x_c < 0 \\ 0, & 0 \leq x_c \leq L \\ x_c - L, & x_c > L \end{cases}$$

In the above equation, x_c is the x-position of ϕ_{control} relative to $\phi_{\text{reference}}$. For the second requirement, the applicable radius of the cone section can be calculated as $r = \max(0, \min(x_c, l)) \sin(\frac{\alpha}{2})$. As long as x_c is within the length of the cone, r is the radius of the cone at that point, otherwise, it is the radius of the cone at the closest end. If the ϕ_{control} is within the cone radius, i. e. $\sqrt{y_c^2 + z_c^2} \leq r$, the constraint values in y and z dimensions are zero.

$$y = \begin{cases} 0, & \sqrt{y_c^2 + z_c^2} \leq r \\ \frac{r y_c}{\sqrt{y_c^2 + z_c^2}}, & \text{otherwise} \end{cases}$$

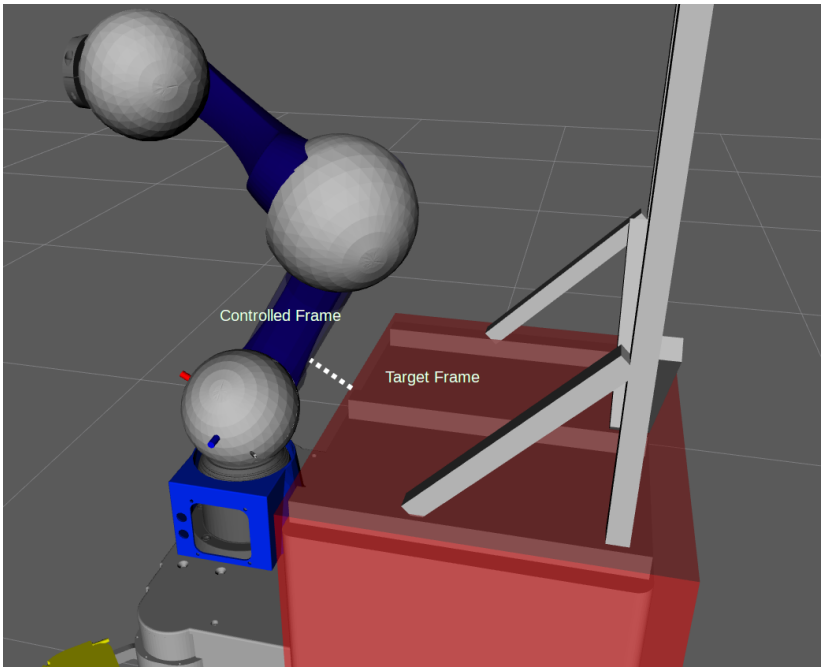


Figure 5.8. Illustration of the self-collision avoidance constraint rule. Here, the distance between the collisions geometries of a box and a cylinder is constrained.

$$z = \begin{cases} 0, & \sqrt{y_c^2 + z_c^2} \leq r \\ \frac{r z_c}{\sqrt{y_c^2 + z_c^2}}, & \text{otherwise} \end{cases}$$

Self-collision Avoidance Self-collision avoidance is realized by controlling the distance between two robot links. Each `SelfCollisionAvoidance` constraint rule is defined between exactly two frames with collision geometries on the robot. Hence, multiple instances will often be required to fully avoid self-collisions. The specified frames $\phi_{\text{reference}}$ and ϕ_{control} must have defined collision geometries. The constraint value is calculated as the distance between the closest points on the shapes. This constraint rule has one configuration parameter, `minimum_distance`. It defines the distance threshold: if the distance between the two frames falls below this distance, the constraint is no longer satisfied. In formulas, this value is written is d_{min}.

The method for calculating the constraint value begins by calculating the closest points on the collision geometries of the two robot links. To find the closest points on the two convex shapes of the collision geometries of the relevant robot links, the Gilbert-Johnson-Keerthi (GJK) algorithm [47] is used. The resulting closest points are referred to as p_c, the point on the controlled frame, and p_t, which is the point on the reference frame.

After the closest points and their distance have been calculated, it is still not clear in which ways the motion of the controlled link should be constrained. To determine which dimensions are relevant to keep the distance, the distance gradient is computed:

$$\nabla d(p_c, p_t) = \begin{pmatrix} \nabla_{\text{lin}} d(p_c, p_t) \\ \nabla_{\text{rot}} d(p_c, p_t) \end{pmatrix}$$

The linear part of the gradient is given as follows:

$$\nabla_{\text{lin}} d(p_c, p_t) = \frac{p_c - p_t}{\|p_c - p_t\|}$$

The rotational part of the gradient is defined by the following equation:

$$\nabla_{\text{rot}} d(p_c, p_t) = \begin{pmatrix} ((1 & 0 & 0)^T \times r) \circ \nabla_{\text{lin}} d(p_c, p_t) \\ ((0 & 1 & 0)^T \times r) \circ \nabla_{\text{lin}} d(p_c, p_t) \\ ((0 & 0 & 1)^T \times r) \circ \nabla_{\text{lin}} d(p_c, p_t) \end{pmatrix}$$

In the above equation, r is the radius of rotation of the controlled link. If p_c is the closest point on the geometry of ϕ_{control}, and p_{c0} is the center of rotation of the frame, the vector r is defined as $r = p_c - p_{c0}$. It must be noted that this gradient only models the distance between the points that are currently closest to each other. This does not always accurately reflect the total distance between the collision geometries. Since only the currently closest points on the shapes are considered, a rotation that increases the distance between these points might not actually increase the distance between the obstacles. For example, the closest points on two spheres can be moved away from each other by simply rotating the spheres in any direction. Nevertheless, the distance between the spheres would, of course, stay the same. Similar effects can happen with other shapes, for example in the case of parallel cubes, where a rotation moves away one point, but moves another one closer in the process. If this causes problems, more accurate methods of computing the distance gradient are also available [30]. In our evaluations, this does however not seem to cause practical problems.

The distance value d is calculated as the difference between the current distance and the configured minimum distance: $d = \max(d_{\min} - \|p_o - p_c\|, 0)$. Finally, the constraint function is defined using this value and the gradient:

$$f_{\text{SelfCollision}}(\mathbf{q}) = \frac{\nabla d(p_c, p_t)}{d}$$

ObstacleAvoidance This constraint rule is based on the idea of the repulsive vector taking multiple obstacles into account as presented by Flacco et al. [35]. All obstacles within a configuration-defined radius are considered. Obstacles that are further away than the radius are excluded to avoid wasting computation time on obstacles with negligible effects.

ϕ_{control} is required to have a defined collision geometry. This constraint rule is the only Cartesian constraint rule that does not require an explicit target frame, and uses simply all obstacles in the surrounding area instead. The following parameters are required from the configuration file:

- `radius`: Obstacles further away than this value are not considered.
- `minimum_distance`: The minimum distance d_{min} to be kept to obstacles. If all obstacles have a larger distance, the constraint value is 0, i. e.] the constraint is satisfied for the current robot configuration.

The set of obstacles O is taken from the *environment model*. For each contained obstacle $o \in O$, the nearest points on o and on the collision geometry of ϕ_{control} are calculated. As above, the method to calculate the closest points between the two shapes is the GJK algorithm [47]. Let p_o be the nearest point on the obstacle, and p_c the nearest point on the controlled link. Next, a repulsive vector is generated from them. The direction of the total repulsive vector takes all obstacles into account.

$$V(O) = \sum_{o \in O} (p_o - p_c)$$

The magnitude of the resulting repulsive vector is based only on the distance to the closest obstacle o_{min} and the minimum allowed distance d_{min}. The distance is calculated as $d = \max(d_{\text{min}} - \|p_o - p_c\|, 0)$. The resulting constraint function then looks as follows:

$$f_{Obstacle}(\mathbf{q}) = d\frac{V(O)}{\|V(O)\|}$$

In its current form, this constraint rule uses only linear motion and thus three dimensions. While it is possible to consider the rotational velocities as well, it is currently unclear how to define the rotational gradient for multiple obstacles in a suitable way. Since it does not seem to cause issues in practical experiments, this is currently left out. The issue can also largely be circumvented by defining the collision geometries of the controlled links in suitable ways. It is however a possible future extension.

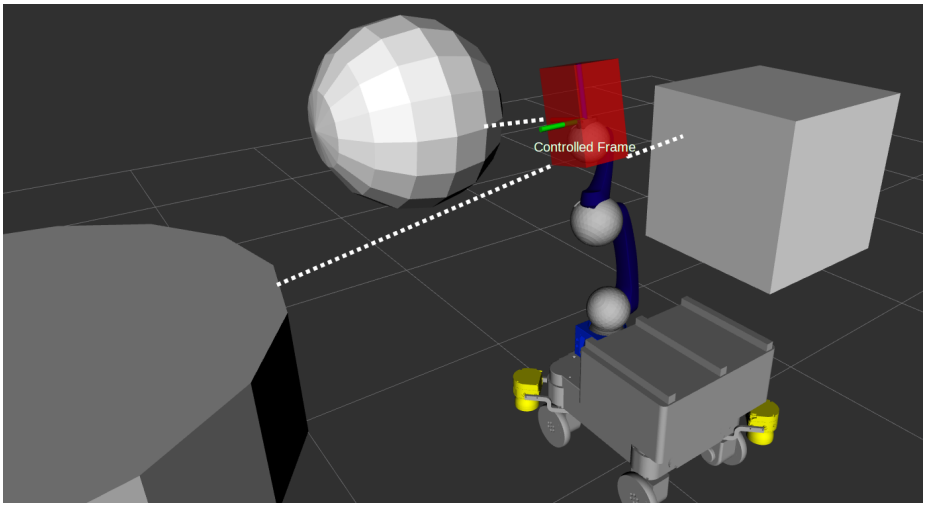


Figure 5.9. Illustration of the obstacle avoidance constraint rule.

5.4 From Constraints to Velocity Bounds: Constraint Controllers

After constraint rules determine the constraint parameters based on the current *inputs* and *context*, the *constraint value* can be calculated from the current robot state $\mathbf{q}$. For a single constraint f, this can be expressed by the formula $f(\mathbf{q}) = \mathbf{c}$. $\mathbf{c}$ names the constraint value, also known as the current *error value* of the constraint. This value, by itself, only contains the information whether the constraint is satisfied (iff $\mathbf{c} = 0$), and how large the error to a satisfying configuration is. To define how the values should influence the robot's motion, constraint controllers are used. Various different types of reactions are possible. For example, if the robot is supposed to move to a target Cartesian pose, the robot should move in the direction opposite of the error, i. e. toward the target pose. If the error is zero, i. e. the robot has already reached the target, the robot should stay there, unless a higher-priority task requires it to move away.

Stopping the robot from driving into obstacles, for example, requires a completely different reaction: Here, the error is zero when the robot has sufficient distance from all obstacles. Not allowing the robot to move in this case would be highly impractical. Instead, the robot should be stopped from moving closer to the obstacles only as the error gets larger. In the case of $\mathbf{c} = 0$, the robot should however be able to move freely. Further, even if close to an obstacle, the robot should not have to stop completely, which would mean that the robot is stuck unless the obstacles move away. Instead, the robot should be free to move away from the obstacles, but not be allowed to move any closer. Additionally, it should be possible to specify different behaviors based on the same constraint rule, for example not moving into obstacles or actively moving away from obstacles.

Thus, different reaction strategies for the constraint rules are needed. These reaction strategies are defined by the *constraint controllers*. The same constraint rules can be combined with different constraint controllers to different effects. As an example, the `ObstacleAvoidance` rule can be used to move away from obstacles, or to just slow down when approaching obstacles, depending on the controller applied to it. It should be noted that, in this section, we are only concerned with one single constraint at any point. How multiple, possibly conflicting, constraints are resolved will be described in the following section 5.6. Furthermore, often the controllers will be presented as if only a single degree of freedom is controlled, for example in function plots. Of course, the controllers usually control more degrees of freedom, but they are left out here for easier understanding. In practice, multiple constraint controllers will be active during any robot action.

At first, it might be the most intuitive concept that each controller generates one exact velocity as its output, which should then be applied to the joints or the controlled link. However, this would be overly restrictive. Often, the task does not require an exact motion, but it can be possible to express the requirements of the task as a range of permitted velocities. In the example of obstacle avoidance, it can be allowed to move away from the obstacle with a high velocity, but only move toward the obstacle with a velocity that decreases until a minimum distance is reached, when no further movement in the direction of the obstacle is permitted. Therefore, constraint controllers do not generate a single velocity, but instead calculate upper and lower bounds on the velocity. Depending on the type of the constraint, these can be bounds on the Cartesian velocity of a robot link, or bounds on the joint velocities of the robot's joints. If the upper and lower bounds are identical, the output corresponds to an exact value. The output of the constraint controllers, the functions for the upper and lower bounds, must be continuous. Discontinuities can lead to various undesirable effects, such as oscillation. The exact behavior of the controllers can be adapted through configuration parameters, for example, to influence the slope of the generated velocity curve. The output of the controllers itself should be bounded, i. e. even a very far away target should not lead to overly large velocities. In this section, the details of the different types of controllers are presented.

In the remainder of this section, the different types of constraint controllers that are used are described in detail. In the equations, b_+ refers to the generated upper bounds on the velocity, b_- to the lower bounds. Both of them are functions that depend on the constraint value $\mathbf{c}$ of the associated constraint.

5.4.1 The `Follow`-Controller

This controller has the intention of letting the robot follow the constraints exactly. Therefore, the upper and lower bounds are set to the same value. The generated velocity is moving in the opposite direction of the constraint value, in order to decrease the error and move toward constraint satisfaction.

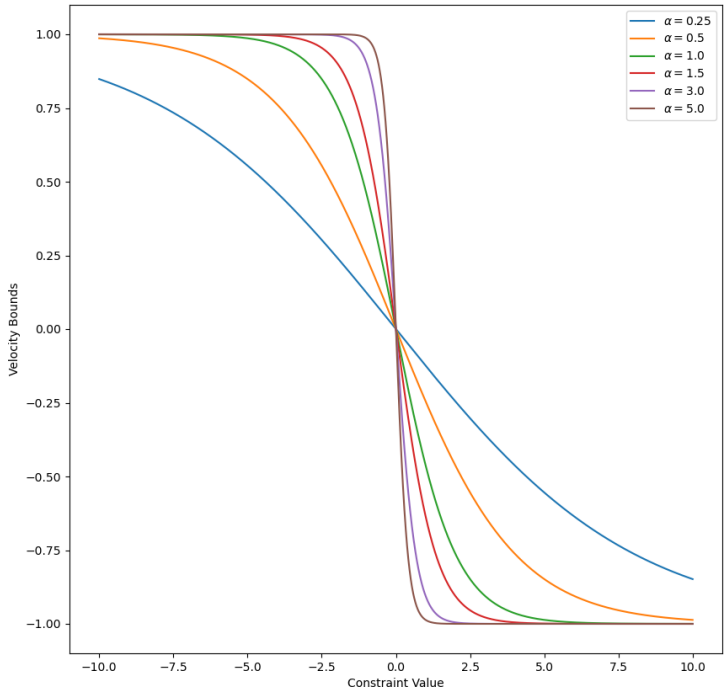


Figure 5.10. Plot of the output of the `Follow`-Controller (equation (5.6)) with $L = 1.0$ and various values for α.

In a very basic way, it could therefore just be defined as $b_+(c) = b_-(c) = -c$. However, to handle the velocity limits smoothly, it is instead modeled as a sigmoid function:

$$b_+(\mathbf{c}) = b_-(\mathbf{c}) = \frac{-2L}{1 + e^{-\alpha \mathbf{c}}} - L \tag{5.6}$$

In equation (5.6), L refers to the maximum velocity output for this degree of freedom. The resulting value is limited to the interval $[-L, L]$. The parameter $\alpha \in \mathbb{R}^+$ can be set by the user to influence the slope of the curve, i. e. how sharply the controller reacts to an error value. Larger values of α lead to a steeper slope. figure 5.10 shows the output of this controller for different values of α.

5.4.2 The `Limit`-Controller

This controller is used for scenarios where the robot should be stopped from increasing the error value any further, after the error has reached a given threshold. Before this threshold is reached, velocities that increase the error are increasingly limited, i. e. the robot has to slow down. It is not enforced that the robot actively moves to decrease the error. This type of controller is often used for safety requirements, such as avoiding self-collisions.

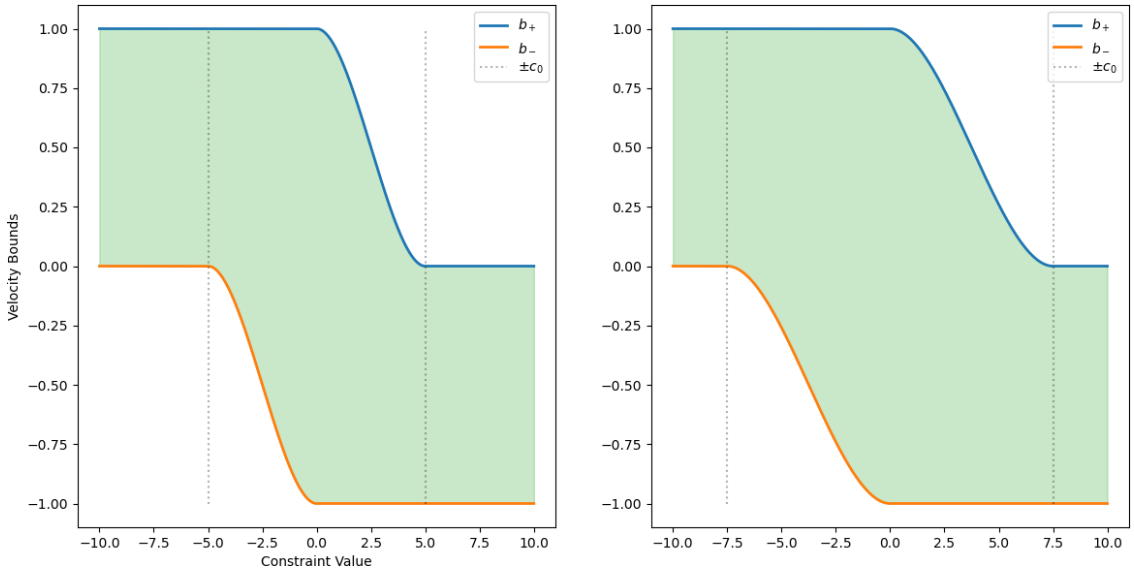


Figure 5.11. Plots of the output of the `Limit`-controller (equation (5.8)) with $L = 1.0$. Left side: $\mathbf{c}_0 = 5.0$, right side: $\mathbf{c}_0 = 7.5$.

This controller is based on the *smoothstep* function [110]. This function is a popular choice for smooth interpolation and was originally designed for the field of computer graphics. It is defined by the following equation:

$$\text{smoothstep}\,(x) = \begin{cases} 0, & x \leq 0 \\ 3x^2 - 2x^3, & 0 < x \leq 1 \\ 1, & x < 1 \end{cases} \tag{5.7}$$

Defining the controller based on the *smoothstep* function has the following advantages, compared to other functions that might be considered, such as the logistic function:

- The function does actually reach its limits, instead of just converging toward them. It does this at clearly defined points. These are 0 and 1 for the basic *smoothstep* function. In the controller, these are 0 and $\pm\mathbf{c}_0$. Therefore, the user can be sure at which point the robot will no longer be allowed to move against the constraint. With a function that is only converging toward its limits, this behavior is harder to predict.
- The function is, at the same time, (once) continuously differentiable.
- From a user perspective, the function can be adapted by simply setting the parameter $\mathbf{c}_0$, the point at which the robot must no longer continue to move in this direction. This is conceptually much simpler to choose, compared to a more abstract slope parameter.

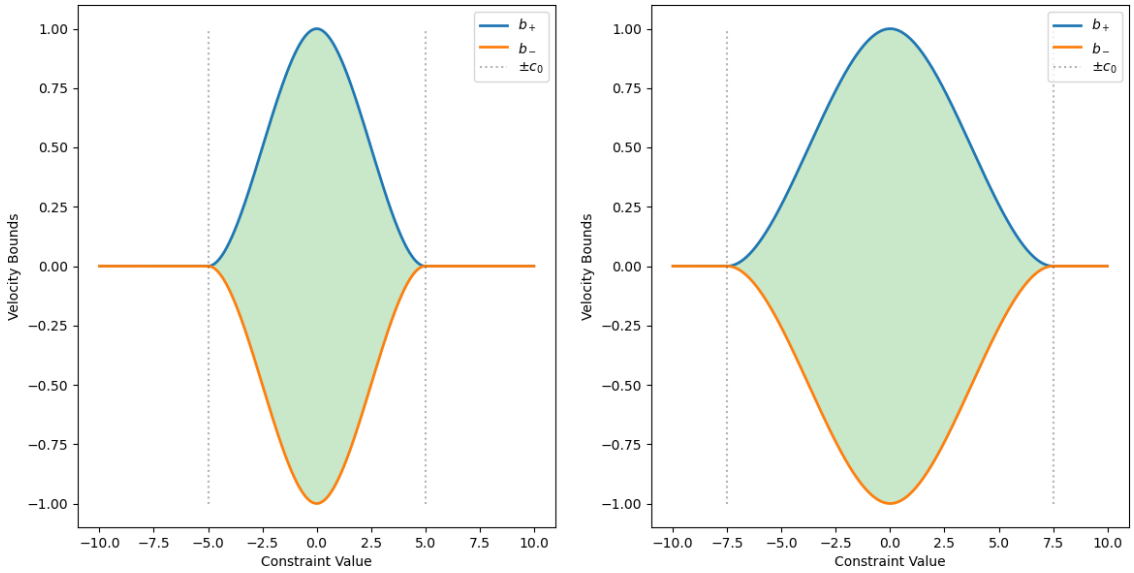


Figure 5.12. Plots of the output of the `Stopping`-controller (equation (5.9)) with $L = 1.0$. Left side: $\mathbf{c}_0 = 5.0$, right side: $\mathbf{c}_0 = 7.5$.

Using the *smoothstep* function, the controller is defined by the functions for b_+ and b_-, as defined in equation (5.8).

$$\begin{aligned} b_+(\mathbf{c}) &= L(\text{smoothstep}\left(\frac{-\mathbf{c}}{\mathbf{c}_0}\right) - 1) \\ b_-(\mathbf{c}) &= -L(\text{smoothstep}\left(\frac{\mathbf{c}}{\mathbf{c}_0}\right) - 1) \end{aligned} \tag{5.8}$$

figure 5.11 shows the output of the controller for two different values of $\mathbf{c}_0$. The permitted velocities are highlighted in green. It can be seen that for $\mathbf{c} = 0$, the velocity is only bound to the interval $[-L, L]$. As the error value $\mathbf{c}$ increases, the permitted velocity in the same direction smoothly get smaller, until it reaches 0 at $\mathbf{c}_0$. Moving against the direction of the error is always possible.

5.4.3 The `Stopping`-Controller

The `Stopping`-controller does not allow any motion when the error gets too large. The velocity bounds at $\mathbf{c} = 0$ are $[-L, L]$ and gradually move toward zero as $\mathbf{c}$ increases. Therefore, this controller does not actively attempt to fulfill the constraint, but has the role of a monitor. Since the velocity is fixed to zero for high values of $\mathbf{c}$, this controller requires that external circumstances change in order to allow motion again. A typical use case for this controller is slowing and eventually stopping the robot when a person comes close. There are cases where it is intended that the robot has to slow down and eventually come to a standstill when a person comes close, and is not allowed to move away on its own either.

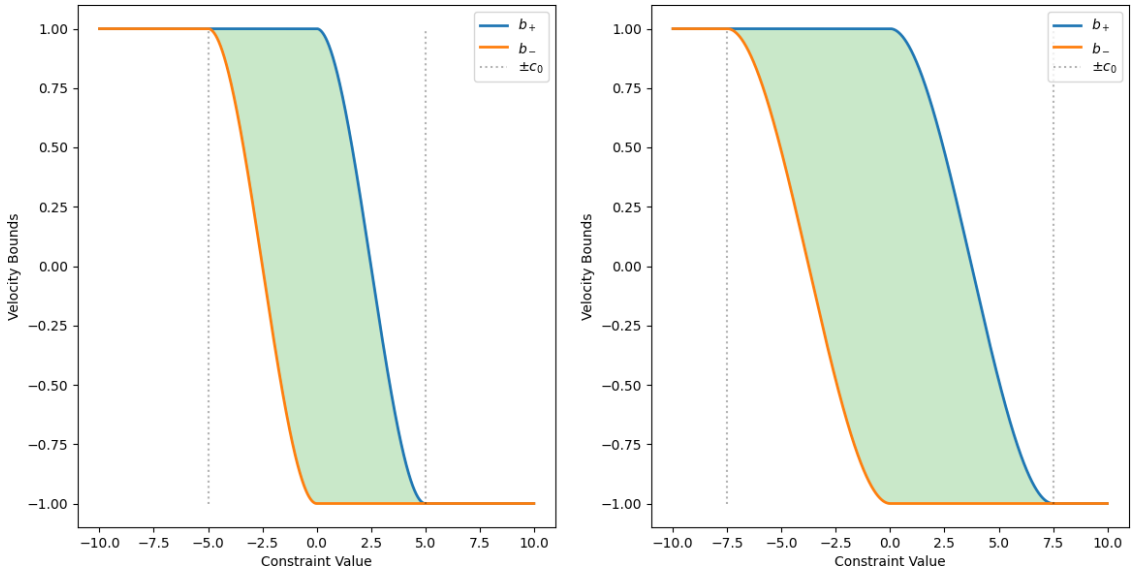


Figure 5.13. Plots of the output of the `Hybrid`-controller with $L = 1.0$. Left side: $x_0 = 5.0$, right side: $x_0 = 7.5$.

The limiting controller is therefore not a valid solution for this case. Otherwise, the calculation is very similar to the `Limit`-controller. The stopping controller is described by the equations given in equation (5.9). Plots of the output for some example values can be seen in figure 5.12.

$$\begin{aligned} b_+(\mathbf{c}) &= -L\left(\text{smoothstep}\left(\frac{|\mathbf{c}|}{\mathbf{c}_0}\right) - 1\right) \\ b_-(\mathbf{c}) &= -b_+(\mathbf{c}) \end{aligned} \tag{5.9}$$

5.4.4 The `Hybrid`-Controller

This controller is similar to `Limit`-controller, but while the `Limit`-controller always allows the robot to stand still, i. e. a velocity of zero, this controller enforces a velocity in the opposite direction of the error if the error gets too large. In contrast to the `Follow`-controller, it permits a range of velocities, especially the entire range $[-L, L]$ at $c = 0$. However, if $|c| > c_0$, the upper and lower bounds converge toward the same value. figure 5.13 shows a plot of this controller. The controller is named *hybrid* because it combines the enforcement of velocities that decrease the error of the `Follow`-controller with the unconstrained motion in low-error cases of the `Limit`-Controller. An example of the practical use of this controller is keeping the robot base close to a person. If the base is close enough, it is free to perform its other tasks, even if they slightly move the robot further away. If, however, the distance increases further, the robot has to move toward the person again.

The control law is again based on the *smoothstep* function:

$$\begin{aligned} b_+(\mathbf{c}) &= -L\left(2\text{smoothstep}\left(\frac{\mathbf{c}}{\mathbf{c}_0}\right)+1\right) \\ b_-(\mathbf{c}) &= L\left(2\text{smoothstep}\left(\frac{-\mathbf{c}}{\mathbf{c}_0}\right)-1\right) \end{aligned} \tag{5.10}$$

The different available controller types are summarized below:

- `Follow`: Enforces velocities that reduce the error value.
- `Limit`: Prevents velocities that increase the error value beyond a specified value.
- `Stopping`: Slows down and eventually stops as the error increases.
- `Hybrid`: A combination of `Follow` and `Limit`. Velocities are unconstrained for small error values, but for higher error values velocities that reduce the error are enforced.

5.5 Tasks and Actions

In the preceding sections of this chapter, the basic concepts of constraint-based control of robots were introduced:

- *Constraints* describe geometric requirements on the robot.
- *Constraint rules* update the constraint parameters to a dynamic environment.
- *Constraint controllers* calculate velocity bounds based on the current constraint values.

After the above definitions of constraints, constraint controllers, and constraint rules, they can be used to define *tasks*. Tasks, in turn, can be used to define actions. This section starts by describing the definition of tasks, followed by the description of how tasks are combined to define actions. Furthermore, concepts for defining the relative importance of tasks and their required accuracy are introduced.

5.5.1 Specification of Tasks

A task encapsulates a requirement on the robot's behavior in regard to its environment and how this requirement should influence the robot's motion. As such, it combines a constraint function and its associated constraint rule with a controller and the required inputs.

The structure of a `Task` is shown in figure 5.14. Besides the above elements, the task also contains other relevant parameters, namely tolerances and weights. Their meaning is described below.

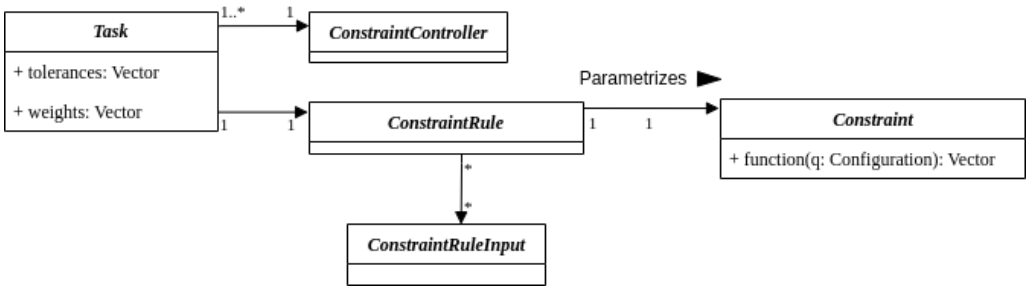


Figure 5.14. Class diagram showing the fundamental structure of `Task`s

Tolerances

According to the fundamental definition of constraints in section 5.2, a constraint is only fulfilled if its value is exactly zero. In many cases, this it not a practicable method of deciding whether a task has successfully reached its goal. If, for example, the goal contains a Cartesian pose for the end-effector, it is practically almost impossible to position the gripper at the pose without any measurable error. Therefore, this requirement often needs to be relaxed. Each task can define a vector of tolerances, $t \in \mathbb{R}^m_{\geq 0}$. If no tolerances are defined, 0 is assumed as the default value.

A task is said to be ***satisfied*** iff, for its constraint function f, the vector of tolerances t, and the constraint value $f(\mathbf{q}) = c$, the following holds:

$$\forall i \in \{1, \ldots, m\} : |c_i| \leq t_i$$

It is important to note that changing the tolerances will not change the generated robot motion. The tolerance is only used to specify how much deviation is allowed to consider the task as satisfied. The constraint controllers will still react in the same way. For example, a `Follow`-controller will attempt to fulfill the constraint as exactly as possible, regardless of the tolerances.

Weights

While tolerances define how exactly a constraint and its components must be satisfied, without influencing the resulting motions, *weights* are used to specify the relative importance of the constraint. They are specified as a vector $w \in \mathbb{R}^m_{\geq 0}$. If an element of w is set to 0, the corresponding dimension is removed from the task. For example, a Cartesian pose task with the weights $w = (0.5, 1.0, 0, 0, 0, 0)^T$ would only affect the x- and y- position of the controlled link. The $z-$ position as well as the three rotational axes are ignored. In the implementation, the task is automatically reduced to a two-dimensional case, so that the implementation is not weighed down by unnecessary zero-entries in the solver and other places. These dimensions are then also no longer considered in the decision whether the task is satisfied.

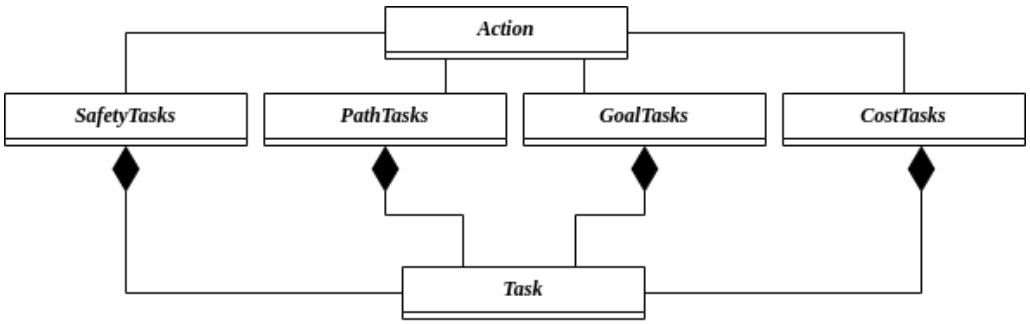


Figure 5.15. Class diagram of an action.

Any weights ≥ 0 do however have the same influence on whether the task is considered satisfied or not, the specific value has no effect on the decision whether the task is satisfied.

In the above example, the x-dimension also has a lower weight (0.5) than the y-dimension, which has a weight of 1.0. This has the effect that the constraint value for the x-dimensions only grows half as fast as that for the y-direction. The controller would thus react more strongly to an error in y-direction.

5.5.2 Definition of Actions

Actions combine multiple tasks into a single unit that can easily be interacted with by the user. They include different tasks with different priorities to define their relative importance. They do not include discrete state changes, such as a change between navigating to a room and picking up an object. Such discrete changes are introduced later in the concept of a *behavior*. Since actions already provide a simple interface with a clear result, it is then simple to combine them, e.g. in a state machine. Figure 5.15 shows the structure of an action. An action simply contains four collections of *Tasks*, which are named *Safety Tasks*, *Path Tasks*, *Goal Tasks* and *Cost Tasks*. For further specification of what constitutes a *Task*, refer to figure 5.14. In the context of reactive control, the difference between the four collections of tasks lies mostly in their priorities. The meaning and definition of the priorities is described in the remainder of this subsection.

Priorities

Priorities describe the relative importance of a task over another. The execution of a task with a lower priority should never disturb the execution of another task with a higher priority. In theory, an unlimited amount of priority levels is possible. The priority is often expressed as a natural number $p \in \mathbb{N}$, with 0 being the most important priority, and higher values becoming less important. This is contrary to how priorities are labeled in natural language, where a *high* priority is the most important, but has the advantage of having an obvious *most important* value (0), and more levels can be introduced simply by following the natural numbers.

However, simply assigning tasks a numerical value is not an intuitive method of specification. The user has to have the priority values for all other tasks in mind when assigning the values, in order to not make any mistakes. In our experiments, no more than four priority levels were needed, and is unlikely to be required in practice. Furthermore, some classification of tasks beyond numeric values is required to express the different semantics of tasks in the context of planning (see chapter 6). Therefore, the following method of classifying tasks instead of simply assigning priority numbers is proposed:

- **Safety Tasks:** Safety tasks have the highest priority ($p = 0$) and are generally required for the safe operation of the robot. Typical examples of safety tasks are keeping the joints within their limits, self-collision avoidance, and collision prevention. Safety tasks are specified globally and automatically added to each action. In some cases, the global safety tasks can, however, conflict with the requirements of an action. For example, this is the case when the robot puts the flashlight from its gripper into the flashlight mount on the base. This would normally be prevented by the tasks for self-collision avoidance. To still allow such motions to be possible, action specifications can provide explicit overrides, that disable safety tasks.
- **Path Tasks:** Tasks that are required to be satisfied at any point during the action, with priority $p = 1$. The difference to safety tasks is mainly the fact that they are not specified globally, but for each action individually. An example of a typical path task is the requirement to hold a glass of water upright.
- **Goal Tasks:** These have a priority lower than safety and path tasks ($p = 2$). Tasks in this class express the main intended goal of the action that should be reached as long as it is compatible with the safety and path constraints. An example would be to place the glass on a table, which might be the main task, but should not override keeping the glass upright or the joint limits.
- **Cost Tasks:** These tasks have the lowest priority ($p = 3$), and are intended to add a specification of optional criteria to the action, that are not critical to the success of the action. Cost tasks are not considered when determining whether the action has succeeded. In the example of placing a glass on a table, this could be a task to keep the platform close to the table, in order to leave more free space behind the robot.

The constraints found from safety, path, goal, and cost tasks are correspondingly referred to as the safety, path, goal, and cost *constraints* of the action. Since safety tasks are defined globally, a new action is completely defined by its **path**, **goal**, and **cost** tasks, as well as optionally the safety overrides. Much of the details of the action then lies in the definition of the tasks. In this way, a library of tasks can be created that can be recombined in different ways to form new action types. Figure 5.16 shows an example of how an action can be defined, using the form of an object diagram.

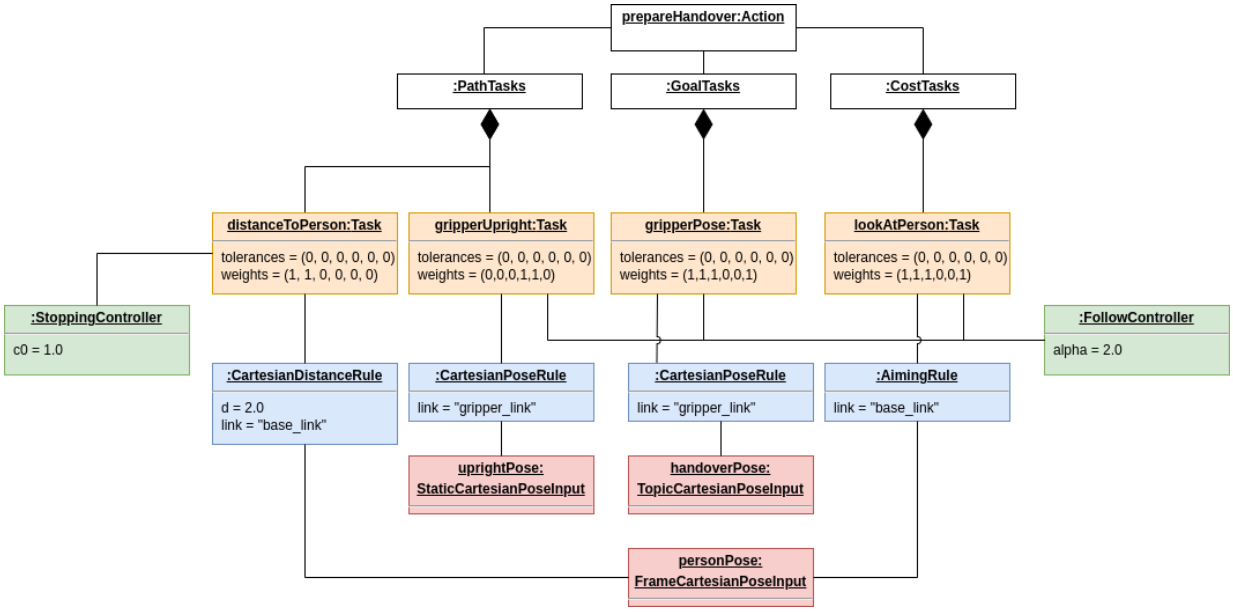


Figure 5.16. Object diagram of an example action. The global safety tasks are omitted here, and shown in figure 5.17 instead.

The safety tasks are omitted for the sake of clarity. They are instead shown in figure 5.17. Unless especially noted otherwise, these are globally defined for all actions and will be the same for all actions. The safety tasks contain nine tasks. One for keeping within the joint limits, four for obstacle avoidance with four different controlled links, and another four for self-collision avoidance with four controlled links. Weights and Tolerances of most of the tasks are not specified, in consequence, they have their default values of 1 for weights, and 0 for the tolerance. The only exception is the task `jointLimits`, which has the weights for the three virtual platform joints set to zero, as they do not have position limits. All tasks use a `LimitController`. Since the safety tasks are merely intended to provide a baseline for safety, not to create motion in itself, this is appropriate here. Two different controllers are instantiated with two different parameter sets. Both will stop the robot relatively quickly after violating the constraints, with c_0 set to 0.01 and 0.1, respectively. On the other hand, the `minimum_distance` parameter in all the collision-related tasks is set to relatively low values. For example, the `SelfCollisionRules` will only be violated if the link moves closer than 0.05 m. The robot will always be rather close to its other links, so a more gradual slowing would cause almost a general slowdown of the robot at any configuration. Instead, the robot will come to a relatively quick stop if the lower-priority tasks move it too close to a collision, whether with an obstacle or itself.

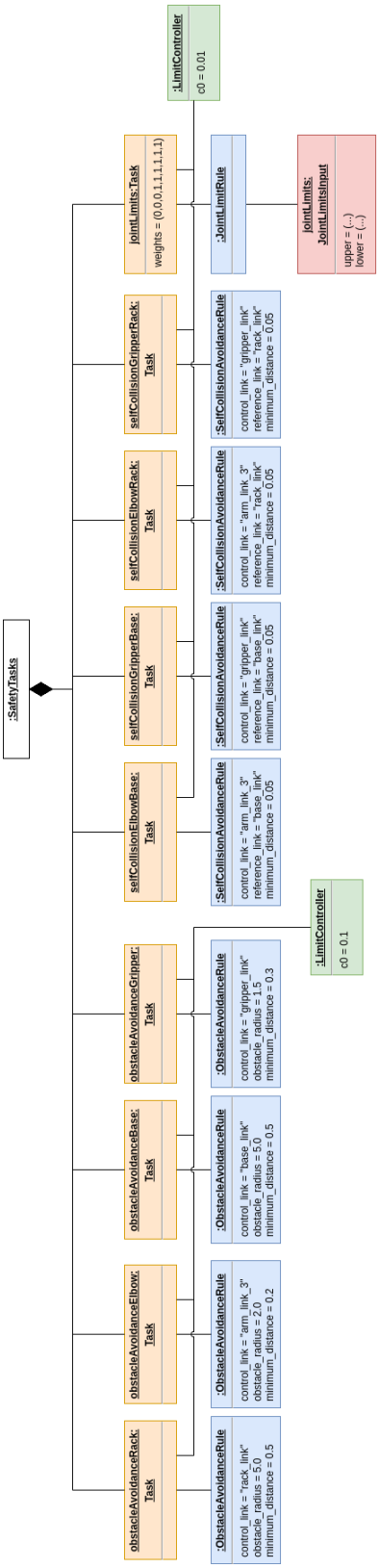


Figure 5.17. Object diagram of the safety tasks.

To summarize, the following concepts are used to define the requirements of robot motions:

- A *constraint* (also *constraint function*) is a function that maps from a robot configuration $\mathbf{q}$ to a vector of error values: $f : \mathbb{R}^n \to \mathbb{R}^m$. A constraint is satisfied for a configuration $\mathbf{q}$ if $f(\mathbf{q}) = 0$. n is the number of controlled degrees of freedom, m is the dimension of the constraint.
- A *constraint rule* updates the parameters of its associated *constraint function* to reflect the influence of the current *context* the robot is situated in, as well as the value of the *inputs*. The *context* encapsulates data such as obstacles, collision geometries, and sensor data. *Constraint rules* can thus be seen as functions mapping from a context $\in \mathcal{C}$ to a constraint: $\mathcal{C} \to (\mathbb{R}^n \to \mathbb{R}^m)$.
- A *constraint controller* is used to define how the robot reacts to the constraint values. Given a current constraint value $c \in \mathbb{R}^m$, upper and lower bounds on the velocity are calculated. Formally, a controller can thus be seen as two functions $b_+ : \mathbb{R}^m \to R^m$ and $b_- : \mathbb{R}^m \to R^m$. The velocities can be joint velocities or Cartesian velocities. Different types of *constraint controllers* are available to define different reactions.
- A *task* combines a constraint rule with a controller, defines tolerances and weights, and additional parameters that might be required by the components. *Tasks* thus form a unit that expresses a requirement on the robot's motion. *Tasks* can, for example, express concepts such as "Keep the gripper upright", "Avoid obstacles with the base", or "Move the arm to a given position".
- *Actions* bring together multiple *tasks* on different levels of priority to describe a type of motion. An action contains *path tasks*, *goal tasks*, and *cost tasks*. Furthermore, *safety tasks* are globally defined and apply to all actions. For purely reactive control, these categories just correspond to different levels of priority. In the later chapters on planning, there or semantic differences to be aware of for the different sets of tasks.

Execution of Actions

Once an action is fully specified, all the user has to do is start the action. The action will then begin to execute the robot motions as determined by the QP solver and the velocity bounds calculated from the individual tasks. Of course, the execution of an action has to end at some point. To decide on the following actions to take, each action will end with a *result*, that is either *failure* or *success*. The execution of an action ends with *success* if all safety, path, and goal tasks are satisfied. An action is called *satisfied* if all of its safety, path, and goal constraints are satisfied.

Several options are available to the caller of an action to specify the execution:

- **Success Duration:** The action only *succeeds* if it is continuously satisfied for the specified duration. If not specified, the action succeeds as soon as it satisfied for a single instant.
- **Timeout:** If the action did not *succeed* after this amount of time, the execution will *fail*. If this is not specified, the action will keep running until it succeeds or is canceled.
- **Cancel Currently Execution Action:** If set to true, any currently executing action will be canceled as soon as the new action is called. Otherwise, the call of the new action will be rejected.

Besides these options, it is also always possible to cancel an action using an external signal. This can be used for choices that need to be made using external mechanisms and can not be decided using the action definition. This is the case, for example, with an action that lets the robot follow a person. This action has no natural successful ending that can be expressed only through the state of the robot and its environment. The end of the action instead has to be determined by an external decision-making procedure that selects the currently appropriate action. For example, this could be a voice command transmitted to the robot, or the robot's low battery state forcing it to move to a charging station.

5.6 Finding an Optimal Control Signal

In the previous section, it was described how *actions* are defined and how they are used for generating velocity bounds based on the robot state and the environment state. In the next step, the velocity bounds, weights, and priorities need to be used to find a concrete control signal. In our architecture, these are joint velocities $\dot{\mathbf{q}}$.

5.6.1 Formulation of a Quadratic Optimization Problem

The search for an optimal joint velocity $\dot{\mathbf{q}}$ can be formulated as a quadratic programming (QP) problem. First, we describe a simple formulation to introduce the concept. Afterwards, the formulation is extended, until finally the full formulation used to control the robot is given at the end of this section.

Simplified Formulation

$$\begin{aligned} \min_{\dot{\mathbf{q}}} \quad & \dot{\mathbf{q}}^T \dot{\mathbf{q}} \\ \text{s. t.} \quad & \mathbf{b}_- \leq \mathbf{J}\dot{\mathbf{q}} \leq \mathbf{b}_+ \\ & \dot{\mathbf{q}}_{\mathbf{min}} \leq \dot{\mathbf{q}} \leq \dot{\mathbf{q}}_{\mathbf{max}} \end{aligned} \tag{5.11}$$

equation (5.11) shows the QP formulation in its most basic form. This optimization problem aims to determine the vector of joint velocities $\dot{\mathbf{q}} \in \mathbb{R}^n$, so that the term $\dot{\mathbf{q}}^T \dot{\mathbf{q}}$, i. e. the squared norm of $\dot{\mathbf{q}}$, is minimal.

While this requirement that $\dot{\mathbf{q}}$ is minimal is conventionally written as the first line in the specification, the core of the problem lies much more in the side conditions. The minimization of $\dot{\mathbf{q}}$ only matters in case that there are multiple equally valid solutions to the constraints, in which the one with the lowest velocities should be chosen. As an extreme example, the robot should stand still rather than move arbitrarily if there are no constraints at all.

The side condition on the last line specifies that the joint velocities must lie within their defined ranges ($\dot{\mathbf{q}}_{\mathbf{min}} \leq \dot{\mathbf{q}} \leq \dot{\mathbf{q}}_{\mathbf{max}}$), where $\dot{\mathbf{q}}_{\mathbf{min}} \in \mathbb{R}^n$ and $\dot{\mathbf{q}}_{\mathbf{max}} \in \mathbb{R}^n$ are the vectors of minimum and maximum joint velocities, respectively. The core of the problem, the execution of the tasks, is specified in the second line of the equation (5.11): the resulting velocities must respect the bounds given by the constraint controllers: $\mathbf{b}_- \leq \mathbf{J}\dot{\mathbf{q}} \leq \mathbf{b}_+$. To keep the equations as concise as possible, we write b_- and b_+ to refer to the *values* of these functions $b_-(\mathbf{c})$ and $b_+(\mathbf{c})$ given the current constraint values $\mathbf{c}$.

$\mathbf{J} = [\mathbf{J}_1^T \dots \mathbf{J}_s^T]^T$ is the matrix formed by vertically stacking all constraint Jacobians of the active tasks. s is the number of tasks in the active action. Similarly, $b_- = (b_{-1}, \dots, b_{-s})^T$ and $b_- = (b_{+1}^T, \dots, b_{+s})^T$ are the stacked vectors of all the velocity bounds generated by the constraint controllers of the tasks.

Full formulation

The above formulation of equation (5.11) is simplified for easier understanding and is not yet adequate for the practical control of a robot. Several other factors have to be considered in the QP formulation:

- No solution to equation (5.11) can be found if the outputs of the constraint controllers are conflicting. Weights and priorities have to be added to the formulation find the best possible solution even in this case.
- Weighting of joints: In many cases, some of the controlled joints are considered preferable to move over others. In the example of the mobile manipulator, rotating the arm with its first joint is generally preferred to rotating the entire platform. This should also be possible to specify in the formulation.
- The platform should not be used with very small control signals, as it is not able to accurately realize them.

The first two of these shortcomings can be resolved by extending the QP formulation. The details of this are presented in the rest of this section. The last shortcoming can not be handled by a pure QP formulation. The description of our solution to this problem can be found in the following section 5.6.1.

The extended version of the QP formulation is given below:

$$\begin{aligned} \min_{\gamma} \quad & \boldsymbol{\gamma}^T \boldsymbol{\eta} \boldsymbol{\gamma} \\ \text{s. t.} \quad & \mathbf{b}_- - \boldsymbol{\chi} \leq \mathbf{J}\dot{\mathbf{q}} \leq \mathbf{b}_+ + \boldsymbol{\chi} \\ & \dot{\mathbf{q}}_{\mathbf{min}} \leq \dot{\mathbf{q}} \leq \dot{\mathbf{q}}_{\mathbf{max}} \end{aligned} \tag{5.12}$$

The last line, specifying the maximum joint velocities, is unchanged. To allow for tasks to deviate from their velocity bounds, if necessary, slack variables $\chi = (\chi_1^T, \ldots, \chi_s^T)^T$ are introduced for each task. These variables capture how far the task is outside its prescribed velocity bounds. Of course, this deviation should be kept as small as possible. Therefore, the optimization variable that is minimized is no longer just $\dot{\mathbf{q}}$, but extended to $\gamma = (\dot{\mathbf{q}}^T, \chi^T)$. In the second line of equation (5.12), the velocity bounds are extended by χ to let deviating solutions become valid.

The matrix $\boldsymbol{\eta} = \text{diag}((\omega^T, w^T)^T)$ is a diagonal matrix containing the specification of the weights of the individual joints $\omega \in \mathbb{R}^n_{\geq 0}$, and the stacked weights of all active tasks $w^T = (\epsilon^{p_1} w_1^T, \ldots, \epsilon^{p_s} w_s^T)$. The task weight vectors are multiplied with the term ϵ^{p_t}, where p_t is the priority of the task. This is used to realize task priorities, based on the observation that a lower priority is equal to taking the limit of the task weight toward zero [26]. In practice, a *very small* value ϵ is a sufficient approximation and can be generalized to more levels of priority by weighting a task of priority p with ϵ^p. Other researchers have found this approximation to have comparable accuracy to specialized solvers with explicit handling of priorities [31].

To solve the QP problem, we have evaluated two different open-source solvers: qpOASES [33] and qpmad [131]. They implement different solving algorithms: qpOASES implements the *online active set strategy* [34], while qpmad implements the Goldfarb-Idnani dual active set algorithm [48]. A third implementation called *eiquadprog* [14] was assessed. Like qpmad, it implements the Goldfarb-Idnani dual active set algorithm. However, in preliminary tests its efficiency was not competitive with the other two implementations. Furthermore, the mathematical formulation of the QP problem it uses is different from the other implementations, for example, it does not explicitly support simple bounds on the optimization variables. Therefore, this implementation was not investigated further. A comparison of the solving performance of qpOASES and qpmad is given below in section 5.7

Both qpOASES and qpmad require the problem to be positive definite. Since the matrices in our formulation can often include zero-rows, making the problem only positive semi-definite, some regularization is required. qpOASES offers an integrated regularization scheme, which we have used. For qpmad, we have explicitly added a regularization term $1 \times 10^{-8} \|\gamma\|$ to the QPproblem of equation (5.12).

Handling minimum velocities of the platform

Modeling the mobile manipulator as a single kinematic chain, with the platform abstracted as three virtual joints, greatly simplifies the formulation of the control problem in Cartesian space. However, this abstraction can not hide the differences in the underlying hardware systems and their consequences for robot motions.

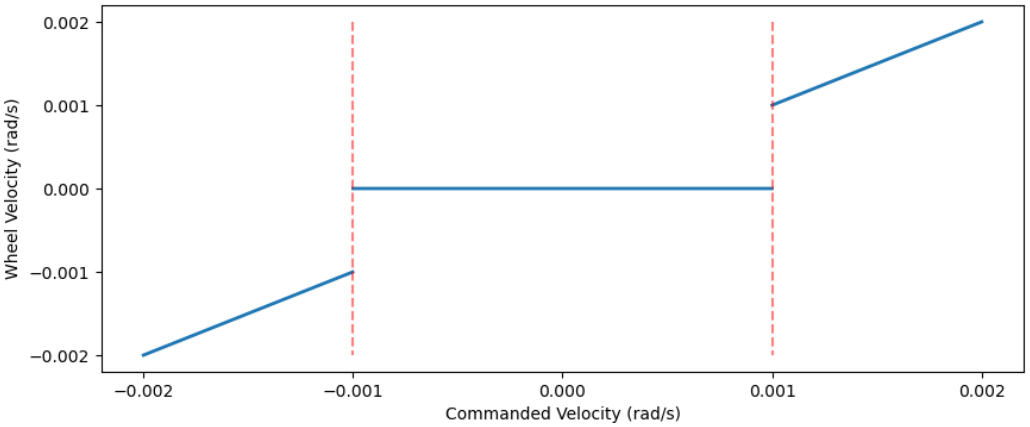


Figure 5.18. Behavior of the platform's hardware controllers at small velocities.

The main problem that occurs is that the platform can not be controlled as exactly as the arm. Since the platform is controlled by commanded velocities, this means that very small velocities can not be used in practice. An exception is, of course, the case when the commanded velocity is zero. In fact, the case of zero velocity is the only case where the platform is controlled as exactly as the arm. Thus, in all use cases that require exact control of the end-effector, the platform should stand still.

If the hardware controller of the platform is commanded a small, but non-zero velocity it is set to zero instead. For the Neobotix MPO-700 platform, the hardware controller does not move the individual wheels if their commanded velocity is below 0.001 rad/s. This is illustrated in figure 5.18. Given the wheel radius of 0.09 m, this corresponds to a linear velocity of roughly 0.09 m/s, assuming that all wheels are moving linearly in the same direction. This threshold is referred to as τ. In our practical implementation, we added some buffer to account for small inaccuracies, thus we use $\tau = 0.1$ m/s. For the control of the entire mobile manipulator, using the method of equation (5.12), these discontinuities at $\pm\tau$ can cause two types of problems:

1. **Inconsistent motions**. Since the calculation is not aware that small velocities are ignored by the platform, the resulting motions do not take this into account. For example, consider a calculated motion in which the end-effector stays still in Cartesian space, and the platform is slowly moving backward. The arm is intended to compensate for the motion of the platform to keep the end-effector still. However, the platform will not move as intended, but the arm will still execute the compensation motion. Therefore, the end-effector would actually move forward in Cartesian space. Thus, both requirements to the keep end-effector still, and to move the platform backward, are violated.

2. **Oscillation.** Often we observed the case that the commanded velocity would oscillate around the threshold value. This leads to a behavior in which the platform rapidly alternates between standing still and driving. It is caused, among other cases, when the commanded value is smaller than the cutoff value. Since the platform will not move, the error will grow until the cutoff is exceeded again. If the error then shrinks again, the platform will stop again, repeating the cycle.

Due to the above problems, this needs to be handled. The discontinuities that are visible in figure 5.18 are characteristic of the hardware and can not be avoided completely. However, some strategies can be used to mitigate their effects.

In order to mitigate the first problem mentioned above, the fact that small velocities are ignored by the platform has to be encoded in the optimization problem. The total velocity v_{base} of the platform is calculated as the Euclidean norm of the three virtual base joints: $v_{base} = \|(\dot{\mathbf{q}}_x, \dot{\mathbf{q}}_y, \dot{\mathbf{q}}_{theta})^T\|$. To generate velocity commands that can be executed accurately, v_{base} should be **either** zero **or** above the threshold τ. This **either-or** problem is, however, not possible to express in a simple QP problem, as it can not be expressed as a (linear or quadratic) constraint on real numbers. Instead, a discrete decision variable has to be introduced. This type of problem, QP with additional discrete decision variables, is called *mixed integer quadratic programming* (MIQP) and requires special solving strategies [78].

In the absence of suitable solver implementations with real-time guarantees, we decided to employ a simple strategy based on two solving attempts of the QP solver. Since the only discrete variable in the problem is the two-state decision variable $X \in \{0, 1\}$, which determines whether the platform stands still or moves with a velocity above the threshold, only the two options need to be evaluated. Each of these options can be formulated as a QP problem. Our strategy then is to solve both of these QP problems sequentially. This is equivalent to two instances of equation (5.12), only that in one of the instances the velocity limits $\dot{\mathbf{q}}_{\mathbf{max}}$ and $\dot{\mathbf{q}}_{\mathbf{min}}$ for the virtual base joints are set to 0. Since the QP solving makes up only a relatively small part of the calculation time required in each control cycle, the increase in calculation time is not critical. For further details and a comparison of the required solving times, see section 5.7 below.

The two solving attempts thus lead to two different results, and a decision still has to be made. Let $\dot{\mathbf{q}}_{full}$ be the result of equation (5.12) without further restrictions, and $\dot{\mathbf{q}}_{arm}$ the result of equation (5.12) with disabled platform motions. If the result of the original formulation $\dot{\mathbf{q}}_{full}$ already fulfills the requirement $v_{base} = 0 \vee v_{base} > \tau$, it is returned as the final result. Otherwise, $\dot{\mathbf{q}}_{arm}$ is used. This solution does by definition fulfill $v_{base} = 0$ and the other velocities have been calculated to be consistent with this. In this way, it is ensured that a solution is used that is aware of the hardware limitations and platform and arm will move synchronized to each other.

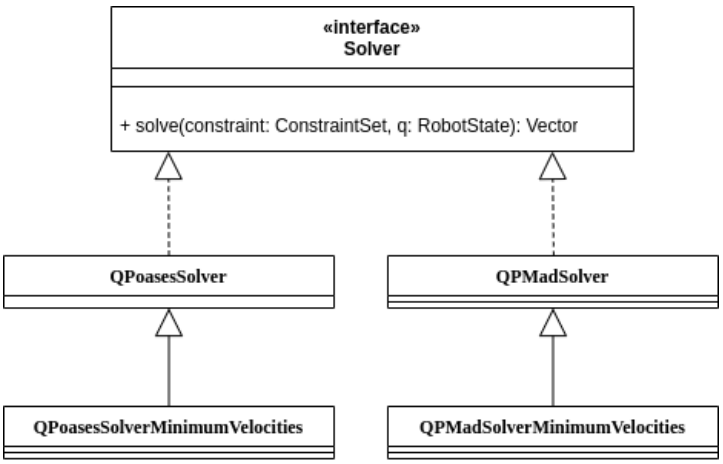


Figure 5.19. Class diagram of the available solver types.

Given the two QP solver implementations qpmad and qpOASES, as well as the option to use the specialized solving method considering the minimum velocities of the platform, four solving methods are currently available in our framework. They all implement the same interface and can easily be switched through a setting in a configuration file. While for the scenarios in the present work, only one of them will be chosen according to the evaluation results, using a different approach can be useful when using the software on a different robot, for example, one without a mobile base with minimum velocities, or other circumstances. The available classes of solvers and their structure are shown in figure 5.19.

5.7 Evaluation

The basic functionality of the reactive control method is evaluated in three different scenarios. The first consists of a motion moving the end-effector to a predefined Cartesian pose. The next scenario performs a null-space motion: here, the robot should keep the end-effector still in Cartesian space, while moving the mobile base to the side. Lastly, the third scenario begins with the same setting as the first, but now the way is blocked by an obstacle. This is intended to evaluate the ability to react to obstacles and handle conflicting tasks. Throughout the evaluations, the required calculation time is measured to evaluate the real-time capabilities of our implementation. The safety tasks that are used in the evaluation scenarios, as well as all following executions, are the ones illustrated in figure 5.17.

5.7.1 Linear Motions to Cartesian Targets

In the first evaluation scenario, we evaluate the basic ability to perform accurate Cartesian motions. The robot is moving its end-effector toward a point in Cartesian space. The only active tasks are the global safety tasks as described above, and the `CartesianPose` task to move toward the target point.

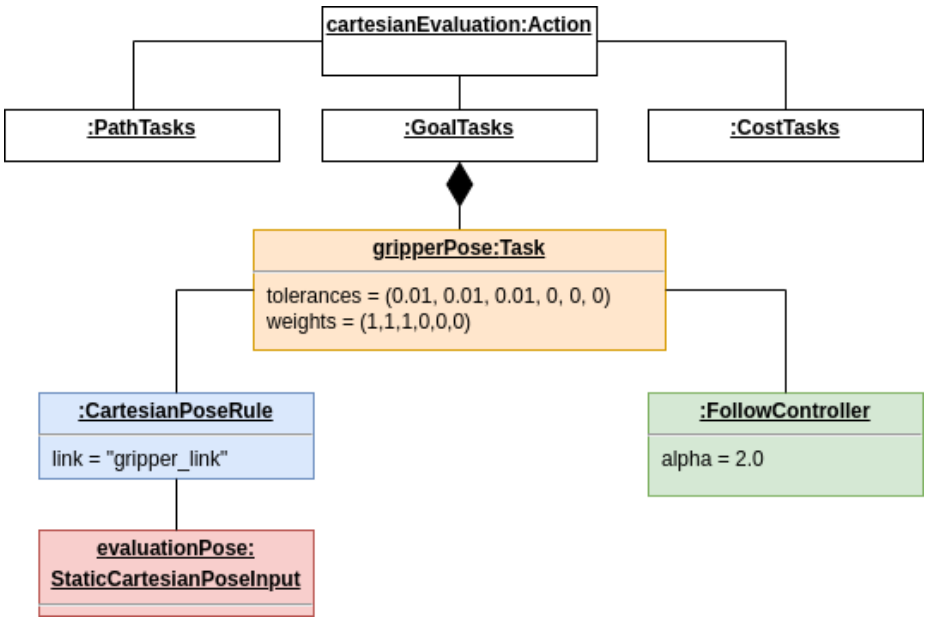


Figure 5.20. Object diagram of the action used in the Cartesian motion evaluation. The global safety tasks are omitted here, and shown in figure 5.17 instead.

The definition of the action is thus simple, and consists only of the specification of the single *goal* task for the Cartesian pose. The corresponding object diagram can be seen in figure 5.20. Besides the safety tasks, the action consists of only a single goal task, while cost and path tasks are empty. The goal task is to move the gripper to the specified target pose. As only the first three weights of the task are non-zero, the orientation is not enforced. A `Follow`-Controller is used to reach the target, the target pose is provided as an input named `evaluationPose`, whose value is statically defined in the configuration file. Tolerances for the three relevant dimensions are set to 0.01 m.

Figure 5.23 shows the results for the solver without consideration of minimum velocities, figure 5.24 with the minimum-velocity-aware solver. In both cases, the scenario described here is shown in the first, lightly colored section of the plot. The plots clearly show that the solver not considering minimum velocities leaves a considerable Cartesian error to the target, which is reduced to almost zero when using the improved solver.

5.7.2 Null-space Motion

In the next evaluation scenario, we introduce conflicting constraints to evaluate their resolution. The robot is holding its end effector still at a given pose in Cartesian space, while the base is moving to the side. The corresponding action is shown in figure 5.21. As shown there, the action is very similar to the previously defined one, but using an additional task for a target pose of the platform as a cost task.

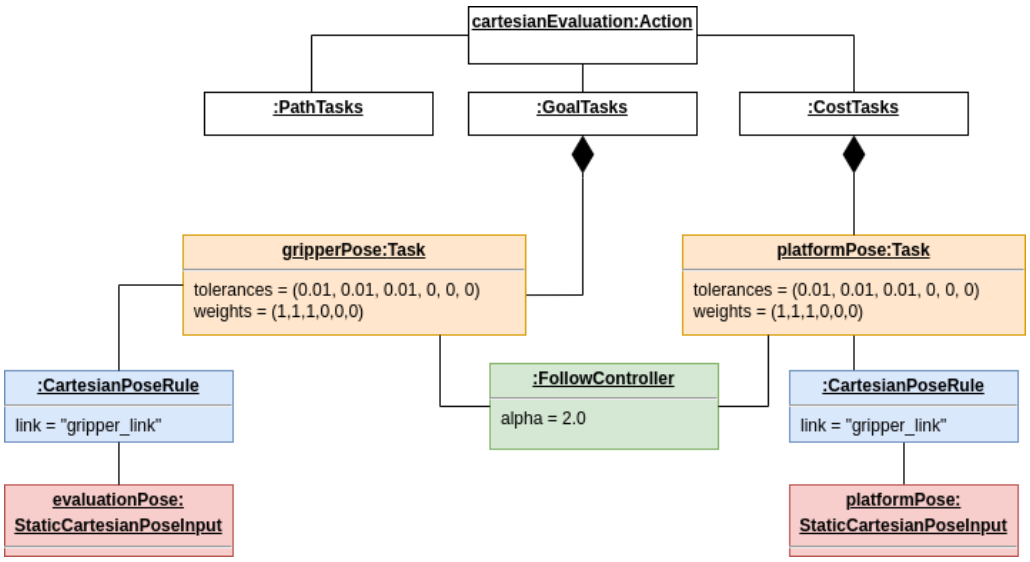


Figure 5.21. Object diagram of the action used in the nullspace motion evaluation. The global safety tasks are omitted here, and shown in figure 5.17 instead.

Therefore, the platform should be moved to its target but without disturbing the gripper pose. The evaluation is started after the previous action has been executed, so the gripper is assumed to already be positioned at its target. Both tasks use the same `Follow`-controller. The photos of the robot poses are shown in figure 5.22

Again, the global safety tasks specified in figure 5.17 are also active. In this scenario, the most relevant metric for this evaluation is the Cartesian end-effector pose error: Since the end effector pose is of the highest priority in this scenario, it should ideally not move from its pose. However, inaccuracies in the control, especially of the platform, can still lead to deviations from the pose.

Figure 5.23 shows the Cartesian position error of the end-effector when executing the Cartesian and null-space motions described above, using the solver without minimum velocities. The Cartesian position error is defined as $\|\vec{p}_{EE} - \vec{p}_t\|$, where $\vec{p}_t$ is the target position, and $\vec{p}_{EE}$ is the target position defined in the task definition. The plots of the two repeated experiments show that a significant error remains even when only tasked with reaching the Cartesian pose. This can be explained by the fact that the remaining error leads to a commanded platform velocity below the minimum velocity threshold, which the platform will simply not execute. Since the solver is not aware of this fact, the arm does not compensate for it either. As soon as the active action switches to the null-space motion experiment, the error shrinks substantially in the beginning. Since the platform is commanded velocities that it can execute, the coordination of arm and platform is working more accurately again. However, toward the end of the motion, when the commanded velocities decrease below the minimum velocity of the base again, the error increases once more, as arm and platform are no longer

(a) Home position **(b)** Reaching the target pose **(c)** After nullspace motion

Figure 5.22. Example photos of the evaluation experiments.

acting consistently. The equivalent plots for the same repeated experiment with the specialized solver can be seen in figure 5.24. Here, the Cartesian position is reached much more accurately. During the execution of the null-space motion, some amount of error in the position of the Cartesian pose occurs, however, it is much smaller than the one in figure 5.23.

5.7.3 Reaction to Obstacles

In this third evaluation scenario, the ability of the method to react safely to obstacles is evaluated, especially the case where the obstacles are conflicting with the targets of the task. Figure 5.25 illustrates the scenario.

As before, the task is to move the end-effector to a given target position. The action definition thus is the same as already presented in figure 5.20. The difference in this scenario lies only in the environment: a box obstacle is in the way to the target pose, preventing the robot from reaching the Cartesian goal. A collision with the obstacle will be prevented on account of the global safety tasks, which include obstacle avoidance for four different robot links. Figure 5.25 illustrates the scenario. With this scenario, we evaluate the handling of conflicting constraints. The collision prevention has higher priority than the Cartesian target, and should not be violated. Still, the Cartesian pose of the end-effector should behave smoothly and in a predictable way.

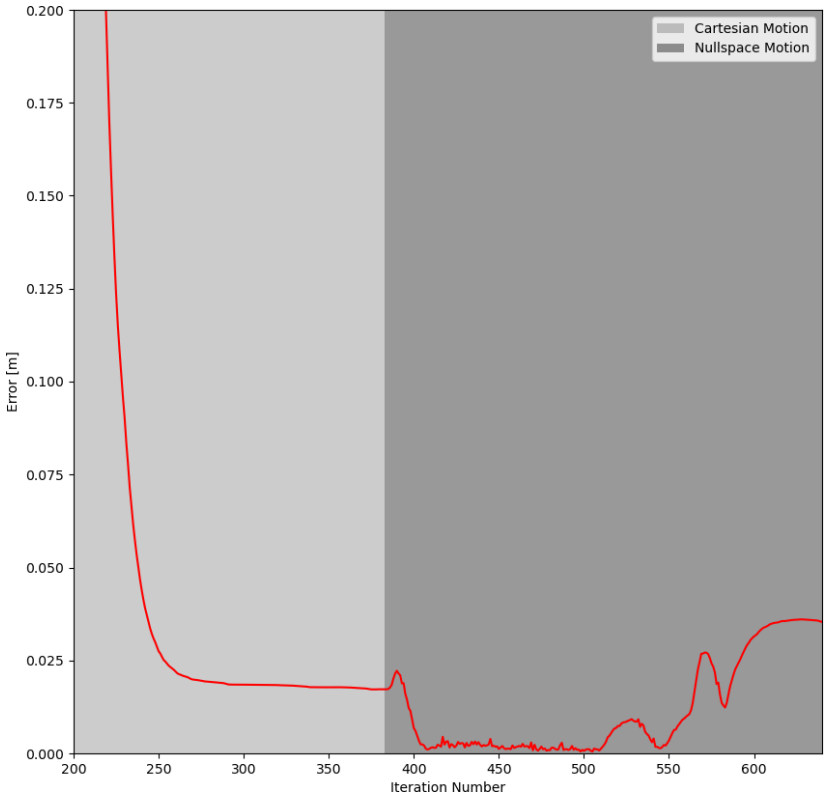
Figure 5.26 shows a plot of the results. The end-effector does not reach its target, but remains with a distance of about 0.2 m, when the platform can not move any further. It stops smoothly as soon as no further progress is possible and remains still at this position. The platform slows down as soon as the distance to the obstacle falls below 0.5 m. The controller stops the platform as soon as the

distance to the obstacle falls below 0.49 m, which is consistent with the parameters of the task. It can be seen that the platform reaches a distance that is slightly below 0.49 m. Besides general control inaccuracies of the platform, this can be explained by three causes:

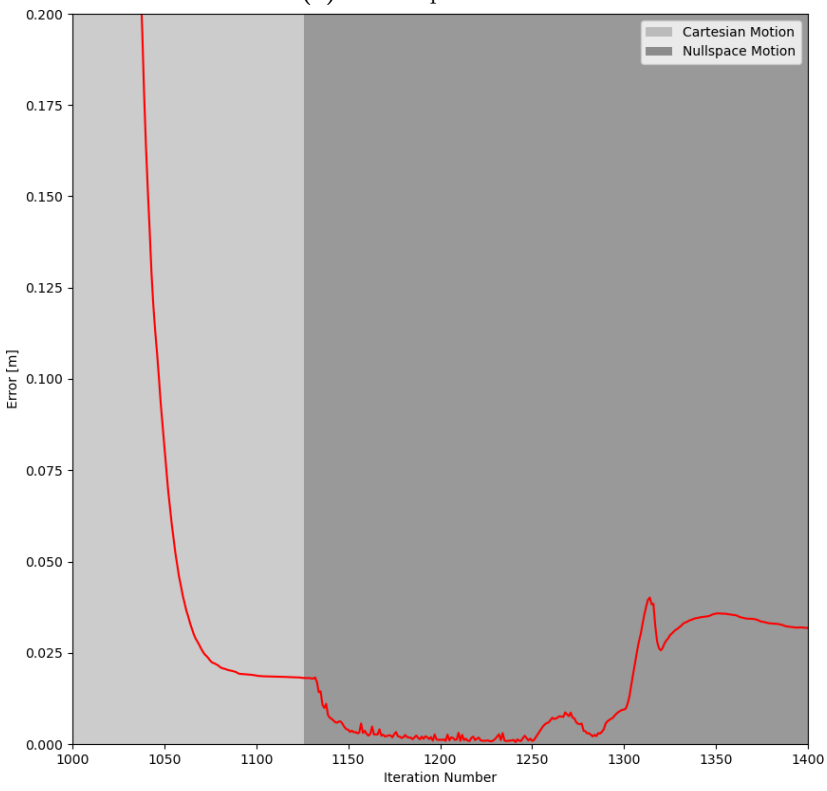
1. Constraints only cause a reaction by the controllers when they are not satisfied, i. e. the constraint value is not 0. In this example, this means that the platform only begins to slow down when the specified obstacle distance falls below 0.5 m.
2. The `limit` constraint controller only stops the constraint value from growing when it reaches the specified maximum value c_0. Stopping the robot instantly as soon as the distance falls below 0.5 m, which would be equivalent to $c_0 = 0$, would lead to a discontinuity in the robot's velocity and a very sudden stop. It is not desirable that the robot comes to a sudden and complete stop whenever an obstacle is within the area where obstacles are considered at all. Instead, the robot slows down smoothly, which necessarily can not be an instant reaction.
3. Some timing delay needs to be taken into account. When a state is reached where the controller stops the platform, the platform is still executing the previous command and thus can not stop in the same instant.

This is all expected behavior. It is however important that the user is aware of these properties of the system. Creating a task to keep the obstacle distance to, for example, 0.5 m, will not mean this distance will always be kept, but that the robot will *start to react* when this distance is no longer satisfied. For the specifics of the reaction, the used constraint controller and its parameters must be considered. This evaluation shows, however, that our control method is able to deal with conflicting constraints and respects their priorities. The resulting motions are still smooth and predictable. It also shows that, when configuring minimum distances and other parameters, the chain of reactions must be considered. The constraint controller only begins to react when the constraint is no longer satisfied, so it is important to choose appropriate limits in the parameters.

This evaluation scenario also illustrates that *planning* can be necessary, as it is clearly not intended that a simple box in the way can indefinitely keep the robot from moving forward. Instead, it should be able to find a way around the box. This can however not be solved using a purely reactive approach, but requires planning. The topic of planning and how an action specification can be used for planning is described in the following chapter 6.

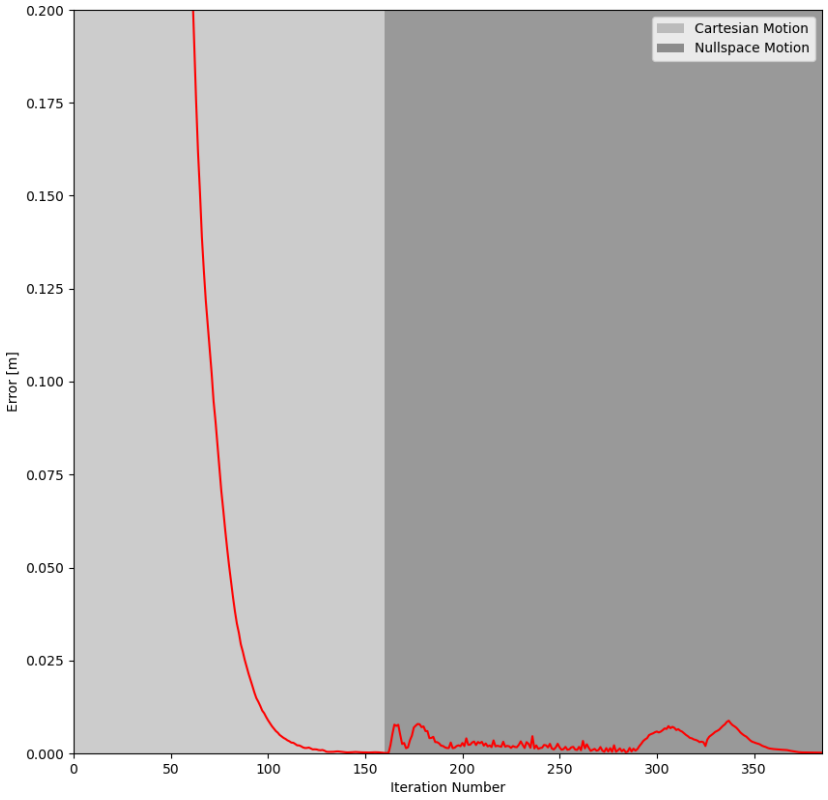


(a) First experiment

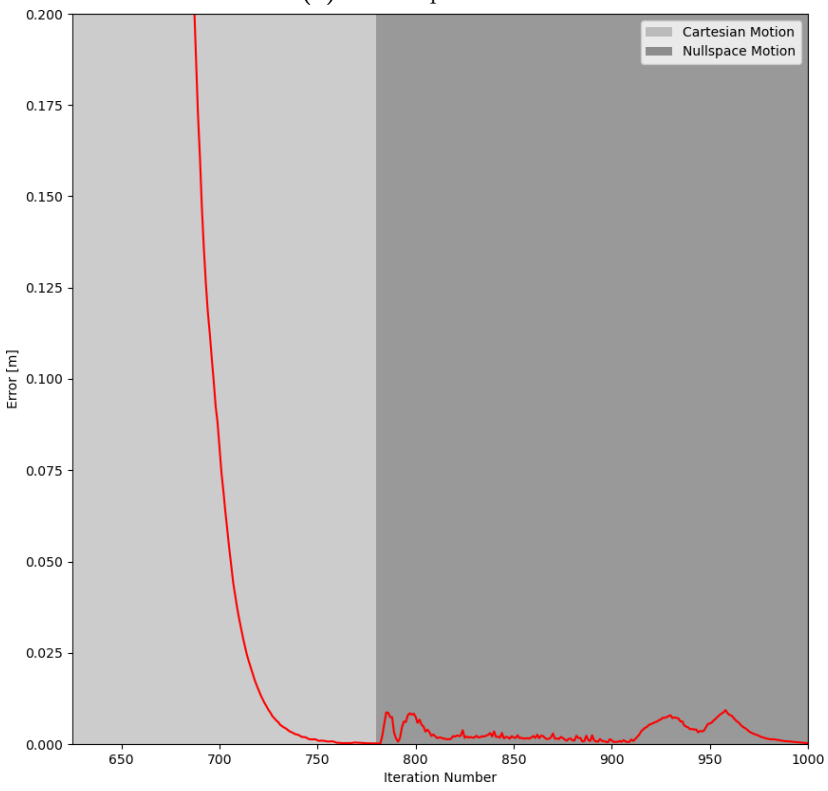


(b) Second experiment

Figure 5.23. Cartesian end-effector error without platform minimum velocities.



(a) First experiment



(b) Second experiment

Figure 5.24. Cartesian end-effector error with platform minimum velocities.

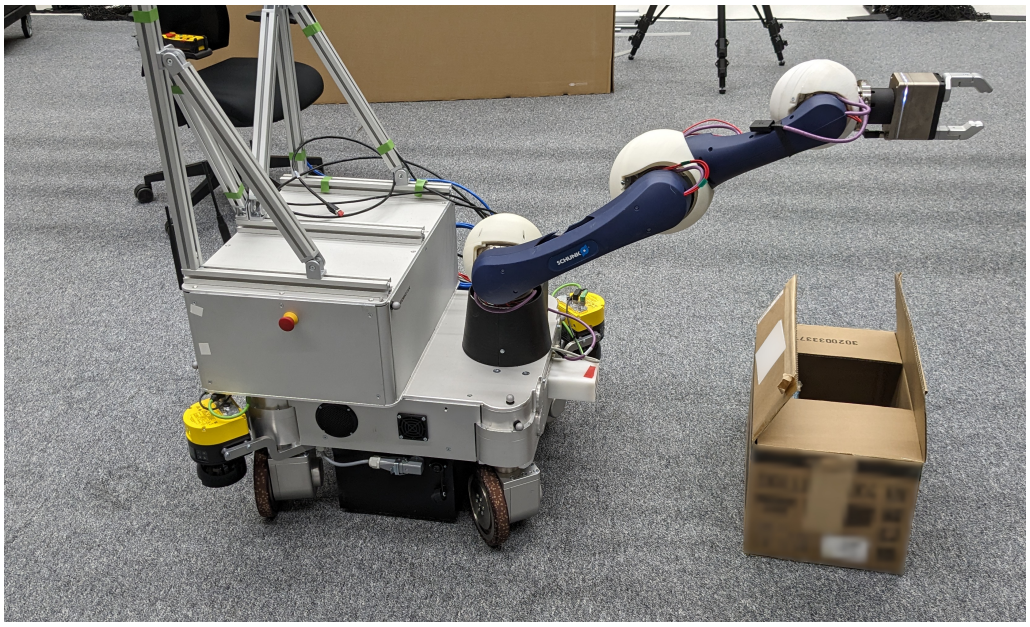
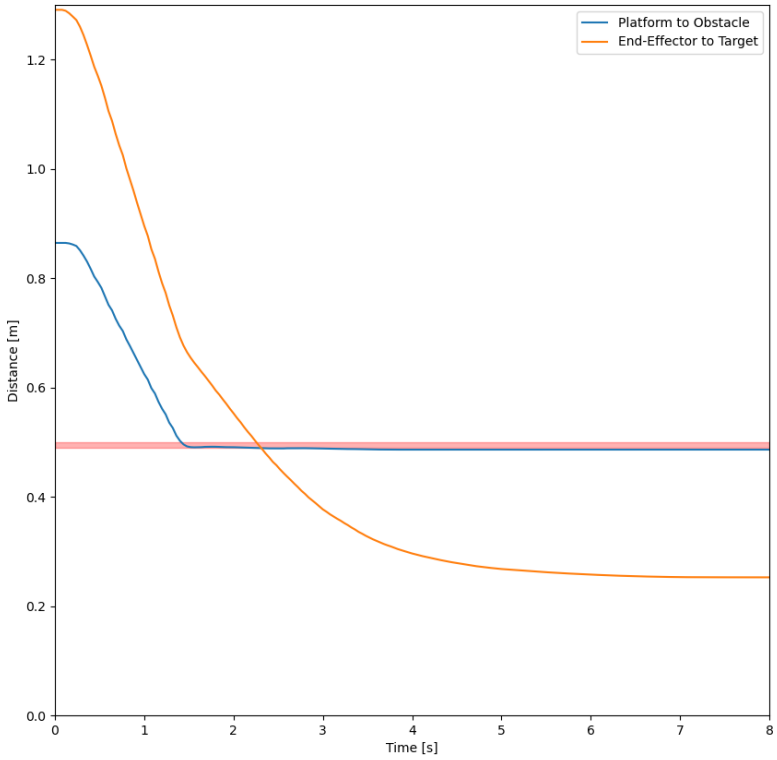
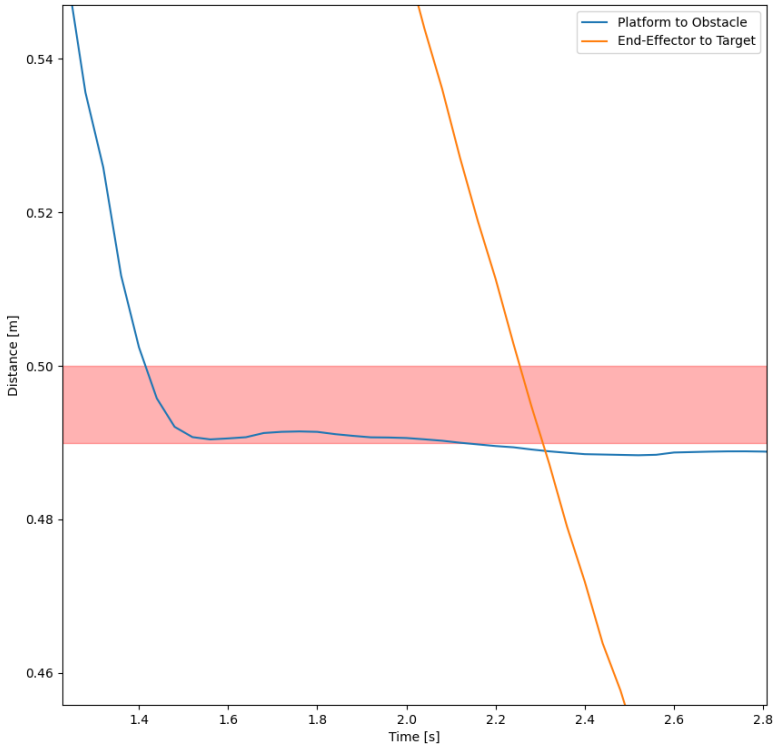


Figure 5.25. Photograph of the evaluation scenario with a box obstacle.



(a) End-effector error and platform-to-obstacle distance.



(b) Detail view

Figure 5.26. Results of the evaluation with an obstacle.

Solver	Total		QP	
	Avg. (ms)	Max. (ms)	Avg. (ms)	Max. (ms)
QPMadSolver	1.30	4.12	0.94	3.12
QPMadSolverMinimumVelocities	2.03	7.57	1.08	7.25
QPoasesSolver	17.38	111.66	16.66	105.92
QPoasesSolverMinimumVelocities	23.09	84.15	22.50	83.38

Table 5.1. Average and maximum calculation times for a full control cycle and for solving the QP problem using the different solvers.

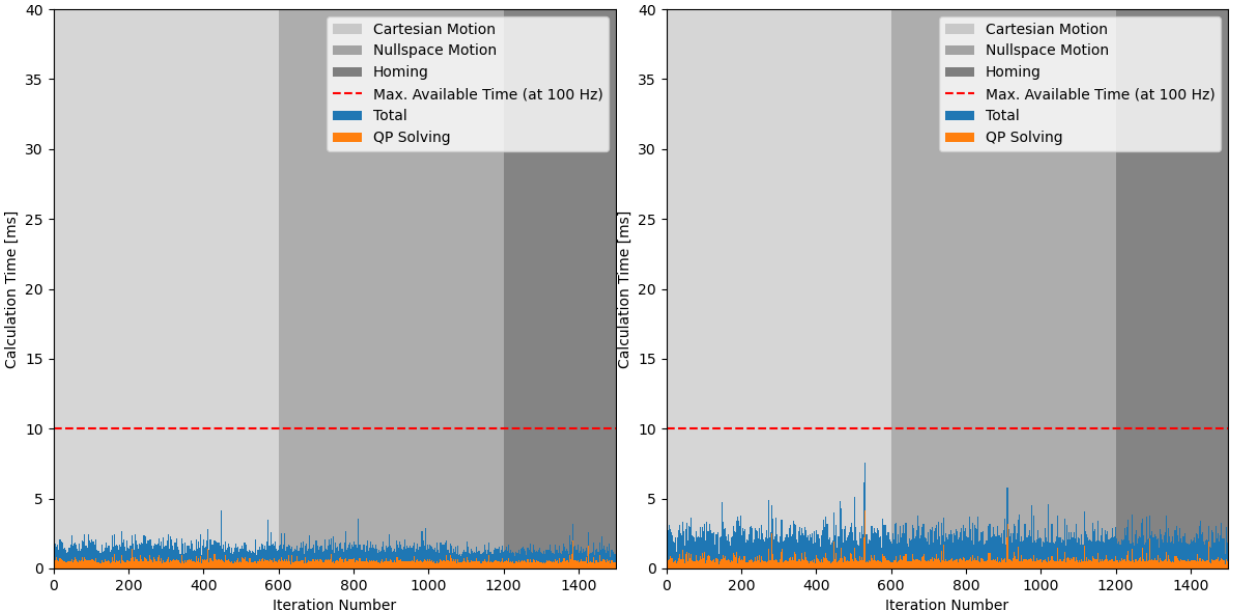
5.7.4 Real-time Requirements

It is of high importance for robot control that the control signal is found within predefined time limits. Especially since the method requires the solving of an optimization problem in each control cycle, low cycle times can be a challenge. If the control signal is only sent with unpredictable frequency, the robot motions, especially regarding safety, can become equally unpredictable.

To evaluate the real-time behavior of our method and its implementation, we have recorded the time that is necessary for each control cycle. The time that is used only to solve the QP problem has been recorded separately, to provide information about how the calculation time is spent. The evaluation is executed on the internal robot computer as described in section 3.2.3. The four solvers shown in figure 5.19 have been evaluated during 1500 control cycles each, while performing the linear Cartesian and null-space motion described above, followed by a homing motion.

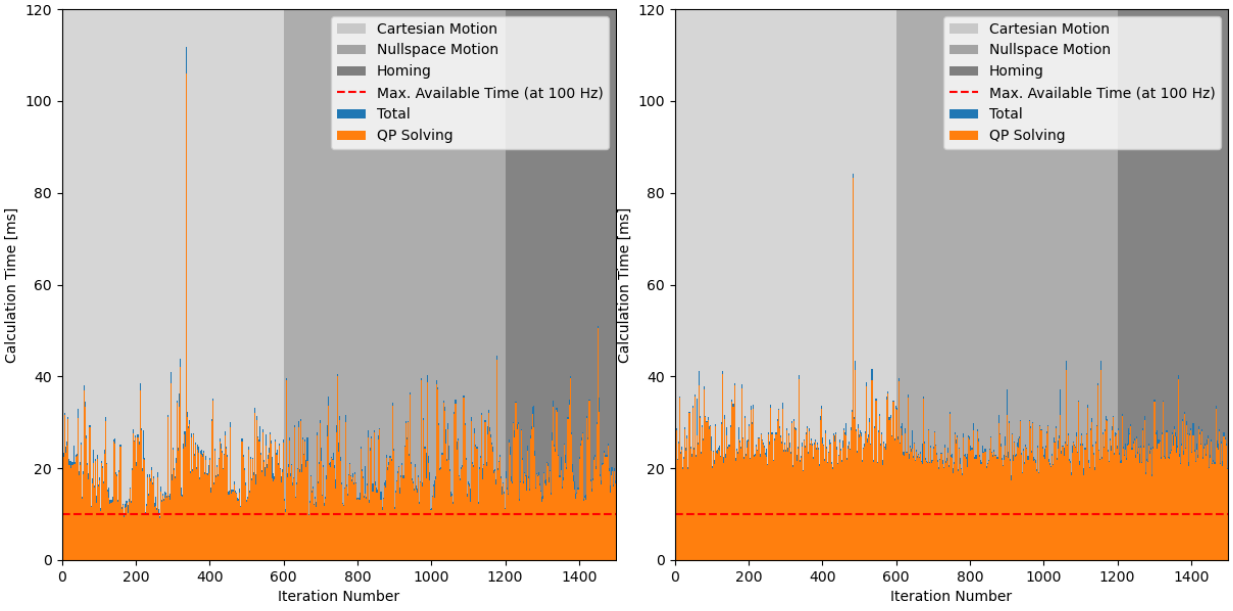
Table 5.1 shows the statistics of the recorded measurements for the four different solver implementations. Figure 5.27 and figure 5.28 show a plot of the required solving times during the different motion types. The qpmad-based implementations consistently perform faster than the ones employing qpOASES. The solver variants that consider the minimum velocities generally take longer, as expected, because they have to solve two QP instances. However, the effect on the total solving time is relatively small and does not fully double the calculation time. Using the qpmad implementations, the calculation time consistently stays below 10 ms, so that a control frequency of 100 Hz can be achieved. Using qpOASES, the solving time is not only longer on average, but also shows a higher variation, with some spikes reaching more than 100 ms required calculation time.

Combined with the increased accuracy of the minimum velocities as evident in the comparison of figure 5.23 and figure 5.24, `QPoasesSolverMinimumVelocities` is the preferred implementation for our use case. If used with another robot, without the minimum velocity requirement of the platform used here, the `QPMadSolver` might be preferred. Judging by the available data, qpOASES is not competitive for this use case. The results of our evaluation are consistent with other benchmarks [130].



(a) QPMadSolver **(b)** QPMadSolverMinimumVelocities

Figure 5.27. Required calculations times of the qpmad-based solvers.



(a) QPoasesSolver **(b)** QPoasesSolverMinimumVelocities

Figure 5.28. Required calculations times of the qpOASES-based solvers.

Summary. Path planning can be used to circumvent some of the limitations of reactive control. The same action specification as used for reactive control can be used to define a planning problem. Solutions to find suitable goals, a valid start, and to respect hard and soft constraints during the path are presented. Different algorithms are evaluated in three evaluation scenarios in order to find the most suitable method.

6

Robotic Path Planning with Constraints

6.1 Related Work **98**
6.1.1 Basics of Path Planning 98
6.1.2 Multi-level Planning 101
6.1.3 Planning with Constraints 103
6.1.4 Planning in Dynamic Environments 109
6.2 Planning Pipeline **110**
6.3 Handling Path Constraints **111**
6.3.1 Path constraints in multi-level planning 112
6.4 Finding Goals from Constraints **113**
6.5 Initial configuration **114**
6.6 Planning with Soft Constraints **116**
6.7 Evaluation and Results **117**
6.7.1 Cartesian Goal with Obstacle 119
6.7.2 Narrow Passage 122
6.7.3 Making room 126
6.7.4 Conclusions 128

Purely reactive control has several limitations: Its lack of foresight means that it tends to get stuck in local minima and can fail to converge towards globally optimal positions. Planning alleviates these problems by considering the entire motion from start to finish, instead of just the current state. Usually, planning begins with the creation of a planning problem, consisting of start configuration, goal configuration, cost metrics, and a definition of valid states, among possibly other elements. In a dynamic environment and with varying action specifications, it is impractical to manually define all of these elements manually whenever planning is required. Instead, we aim to use a given action specification as also used for reactive control, and automatically use it to define and solve a planning problem.

This chapter begins with an overview of existing techniques of path planning, especially those with constraints. Afterwards, a method of planning for an action definition as defined in the previous chapter is presented. It includes finding suitable goals as well as the path to the goals, while respecting hard and soft constraints. To conclude the chapter, different planning algorithms are evaluated and compared in multiple scenarios to find the ones most suitable for the use cases of this work.

6.1 Related Work

In this section, an overview of the state of the art of path planning is given. It will be helpful for the rest of this chapter to provide both the reasoning and notation of the foundations of path planning, as well as state-of-the-art algorithms, before describing our own work. After a general introduction to path planning in section 6.1.1, related works for the particular challenges of the use cases of this work are presented. These are mainly the consideration of constraints in planning, which is presented in section 6.1.3; and planning in dynamic environments, which is described in section 6.1.4.

6.1.1 Basics of Path Planning

Path planning is an essential tool for autonomous robots. The notion of the *configuration space* is important in the context of planning. It describes the space containing all possible robot configurations, and is typically partitioned into the *free space* and the *obstacle space*. The free space contains all configurations that are not in collision with any obstacles, the obstacle space is its complement. The size of the configuration space and the difficulties of calculating the free space and the obstacle space are often at the core of path planning problems. In general, the problem of path planning is PSPACE-hard [108], and the complexity grows with the number of degrees of freedom to be considered.

Potential field methods [69] can be seen as the first approaches for robotic path planning. These methods work by generating a potential that draws the robot towards the goal and pushes it away from undesirable configurations. The robot follows the gradient of the potential to reach the goal. The potential is defined as a function of the configurations, and therefore it is enough to know the configurations that are part of the path and the potential at these configurations; the configuration space does not need to be modeled explicitly. Two large downsides of this approach exist, however: it can easily get stuck in local minima, and it is difficult to design the potential functions so that the resulting path has all the desired characteristics. While methods to avoid local minima have been proposed, it is hard to find general solutions that are applicable to a wide range of problems. The problem of local minima thus remains an obstacle in the adoption of potential field methods.

Path planning can also be framed as a graph search problem, and the well-known methods for these problems, such as the A*-algorithm [55], can be applied. However, this requires a discretization of the configuration space.

With fine discretization, the search space gets very large, and the algorithms are no longer efficient enough, especially in high-dimensional configuration spaces. With a coarser discretization, the algorithms are no longer complete, i. e. they might not find a solution, even though one exists. Due to these problems, these methods are generally not used for robot path planning, except for low-dimensional problems such as two-dimensional navigation.

The most popular approach for path planning is *sampling-based planning*. The basic idea is to avoid computing the entire configuration space, and instead work with randomly generated samples of configurations. While this can also be seen as a discretization of the configuration space, it is different from static discretization of the configuration space by the fact that new samples will continuously be added. They are thus not limited to an initial discretization, and for most methods, the probability of finding a solution (given that it exists) converges towards 1 with an increasing number of samples. This is called *probabilistic completeness*. Some methods are also *asymptotically optimal:* As the number of generated samples goes towards infinity, the probability of finding an *optimal* path goes towards 1.

Sampling-based planners can be classified into two broad categories: graph-based and tree-based. Graph-based methods, such as the probabilistic road map method [67], create a map of the environment in the form of a connectivity graph that can be used for all future queries in that environment. This is often efficient for environments that do not change. Since our focus lies on dynamic environments, we concentrate on the second category of sampling-based planners: the tree-based planners. For each new query, a tree of configurations is grown from the start configuration. The edges of the tree stand for possible connections between the configurations at the nodes. When the tree contains both the goal and the starting position, a path has been found.

The first tree-based planner is the method of rapidly exploring random trees (RRT) [77]. A prototypical description of tree-based planning is described in algorithm 6.1. Given a start configuration q_{start} and a goal configuration q_{goal}, it begins by creating a tree containing only the start configuration as its root. Then, it enters a loop that terminates only once the tree contains a path from q_{start} to q_{goal}. This illustrates the lack of completeness of typical sampling-based planners: there is no simple way to detect cases where no solution exists. Practical implementations thus have to add other termination criteria, such as upper bounds on the number of iterations. Moreover, it is often desirable to continue building the tree even after a first solution has been found to find better paths.

Algorithm 6.1 Basic procedure of tree-based planning.

Input: q_{start}, q_{goal}
$T \leftarrow initTree(q_{start})$;
while not $T.containsPath(q_{start}, q_{goal})$ **do**
  $q_{rand} \leftarrow sample()$;
  $q_{near} \leftarrow select(T, q_{rand})$;
  $q_{new} \leftarrow extend(T, q_{rand})$;
  if $connect(q_{new}, q_{near})$ **then**
    $T.insert(q_{near}, q_{new}$
  end if
end while
return $T.findPath(q_{start}, q_{goal})$

As long as it is running, the algorithm will perform the following steps: A random configuration q_{rand} is created using the procedure $sample()$. This configuration can usually be anywhere within the configuration space. Next, the nearest node of the tree is determined using the procedure $select(T, q_{rand})$. This is the node where it will be attempted to extend the tree. Next, the configuration to be added q_{new} is created by the procedure $extend(T, q_{rand})$. Usually, this procedure creates the new configuration by moving in the direction of q_{rand} starting from q_{near} with a defined step length. It is then checked whether it is possible to connect q_{near} and q_{new}, using $connect(q_{new}, q_{near})$. This usually tests whether the robot can move directly from q_{near} to q_{new} without collisions. If the check succeeds, q_{new} is inserted into the tree.

Using the basic method of algorithm 6.1, sampling-based planning provides a framework that is easy to adapt for many kinds of specialized applications. Many variants can be defined just by changing the details of the procedures *sample*, *select*, *extend*, *connect*, and *insert*. For example, the RRT can be used for non-holonomic robots by using specialized *connect* and *extend* procedures that consider the robot's steering abilities. One method of handling constraints in an RRT works by using a *sample* procedure that is already able to generate only constraint-satisfying configurations. Further detail of this is given in section 6.1.3. Other variants of tree-based planning use slight variations of the above algorithm, for example by building multiple trees in parallel.

The RRT* [66] is an especially noteworthy variant of the RRT. It can be shown to be asymptotically optimal and thus is able to find optimal paths, given enough time. The main difference to the RRT is that the tree will be *re-wired* if shorter paths are found. The operation of re-wiring the tree can be costly in itself, however.

The transition-based RRT (TRRT) [62] is another noteworthy variant of the RRT that attempts to produce paths that optimize a cost function. In contrast to the RRT*, it is not guaranteed to converge towards an optimal solution. It aims to produce low-cost paths that follow valleys and saddle points on the cost map of

the configuration space. It uses the concept of transition tests from stochastic optimization methods to accept or reject new potential states based on how much they change the current cost.

In practical applications, RRTConnect [75] is a popular variant. Instead of just growing one tree from start to goal, this method grows two trees from both ends and attempts to connect them.

A large variety of variants of the RRT have been developed over the years, with many specialized applications. There are too many to list exhaustively here. The most relevant specializations will be described in the following sections. First, multi-level planning, a specialized planning approach for systems with high numbers of degrees of freedom, is described. This is followed by an overview of methods for planning with constraints and planning in dynamic environments.

6.1.2 Multi-level Planning

Motion planning for a mobile manipulator in general has to consider all available degrees of freedom. Even in scenarios that mostly focus on either the mobile base or the arm, the rest of the robot can generally not be ignored. When navigating only with the robot base, it might be necessary to move the arm to avoid obstacles. On the other hand, moving only the arm limits the robot to a small work-space that is immediately reachable from the current base position.

However, planning for all degrees of freedom at the same time is costly in terms of efficiency. In many cases, it is also sufficient to keep either the base or the arm still and only plan for the remaining degrees of freedom. Whether it is necessary to consider all degrees of freedom or not is however not possible to decide in advance, especially when dealing with unpredictable and dynamic environments. Reducing the number of degrees of freedom can critically reduce the solution space, but always planning with all available degrees of freedom is inefficient and often unnecessary.

Multi-level planning is a recently proposed technique that attempts to resolve this dilemma in a soundly formulated way [106, 105]. Instead of planning directly in the full configuration space, these planning algorithms plan in a sequence of simplified state spaces. For example, the simplified state spaces can be defined by removing some degrees of freedom of the robot from the planning problem. The proposed planning algorithm *quotient-space rapidly exploring random tree* (QRRT) is growing trees, analogous to the original RRT, both sequentially and simultaneously on each of the simplified configuration spaces. The final resulting plan considers all degrees of freedom and solutions that require movement with all degrees of freedom are not precluded. The sequential simplifications can be exploited by focusing the exploration of the random trees on configurations that are feasible in the lower-dimensional configuration spaces, which can be explored much more quickly. The QRRT algorithm has been shown to be probabilistically complete, but no optimality guarantees can be given.

To give an example with the mobile manipulator used in this thesis, the full configuration space is the nine-dimensional configuration space of the six arm joints and the three virtual base joints. A possible simplification can be formed by removing the arm joints, so that the configuration space is projected onto a three-dimensional, simplified configuration space. The hope of using multi-level planning is that the simplifications can be used to explore the state space in more sensible ways, without precluding solutions that require all degrees of freedom. Considering a navigation scenario that mostly requires the platform to move, with little to no arm motion required. Using a standard RRT would waste a lot of computation time on exploring different arm configurations. With multi-level planning, the motions of the platform would be explored first, and only feasible motions there completed by extending them with the arm again. The inverse type of scenario, where only the arm is required to move, should equally be able to be handled efficiently by multi-level planning: If the initial configuration of the base already satisfies the goal requirements of the base, the exploration can focus on exploring arm motions.

In general, the sequence of simplified configuration spaces is defined through the projection operators between them. Projection in the direction of the simpler state space is modeled as a mapping $\pi : X \to Y$ from the original configuration space X to a lower-dimensional configuration space Y. The inverse mapping is not unique and thus describes a set of solutions for each configuration y: $\pi^{-1}(y) = \{x \in X | \pi(x) = y\}$. This set is called the *fiber* of y in X.

The definition of suitable projections between state spaces, called *admissible lower-dimensional simplifications* (ALDS), is very general and quite abstract. The most important case of the so-called *canonical* ALDS is more intuitive: Given a configuration space X that is the product of two sub-spaces Y and Z, $X = Y \times Z$, the canonical projection π is defined as $\pi : X \to Y$. In effect, Z is removed from the configuration space. In practical terms, this corresponds to removing joints of the robot.

Besides the configuration space itself, the validity checks of the configurations must also be considered. Invalid configurations are, for example, configurations in collision with obstacles, or other constraint-violating cases. In the case of a *canonical* ALDS, the configurations in the projected state space Y are defined to be invalid if and only if each element of its fiber $\pi^{-1}(y)$ is invalid in X. This means that an invalid configuration in a simplified configuration space does not need to be considered further, as it can never lead to a valid configuration in the original configuration space.

However, this general definition of invalid configurations can lead to inefficient computations, as every element of $\pi^{-1}(y)$ possibly has to be checked to determine the validity of y. Thus, the *canonical* ALDS are further restricted to the *efficient* ALDS. These are, roughly speaking, the ALDS where y can be checked for validity with a single check. This is the case when leaving out parts of the robot to form a simplification: If, for example, the mobile base of the robot is already in collision

with an obstacle, then no possible configuration of the arm can create a valid configuration.

In the evaluations of the creators of multi-level planners [105], they show remarkable results. In most of the scenarios they considered, which have 21 or more degrees of freedom, the multi-level planners are often faster by orders of magnitude when compared to classical planning algorithms. Therefore, we consider multi-level planning a promising approach for our applications and have evaluated it below.

6.1.3 Planning with Constraints

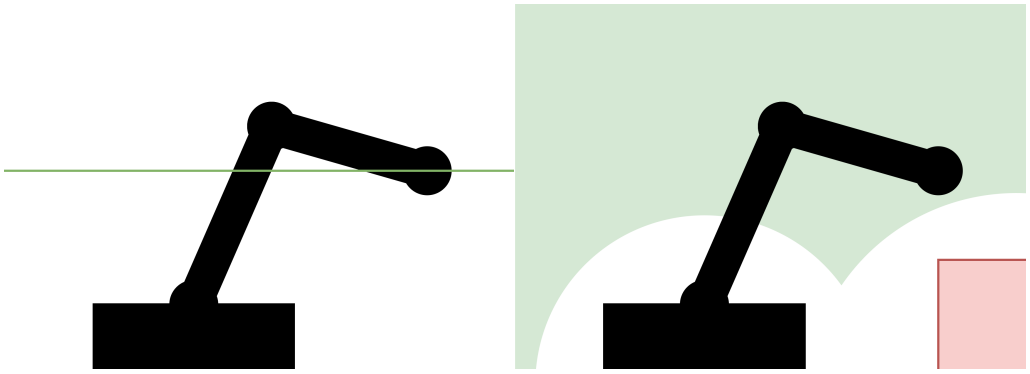
In planning problems with constraints, it is not only required to find a path for the robot from start to goal without collisions, but also to satisfy given constraints during the entirety of the path. The first instances of constrained path planning arose from industrial robotics. In this area, constraints are used mainly to describe constraints on the end-effector. Example applications include welding and painting. To apply paint, the robot has to constantly keep in touch with the surface to be painted; in welding the end-effector has to closely follow a prescribed path. Many other industrial applications similarly require geometric constraints of the end-effector. For typical industrial manipulators with six degrees of freedom, these constraints are relatively easy to handle. If the end-effector is constrained in all six dimensions of Cartesian space, only up to eight satisfying joint configurations exist (outside of singularities) for each pose. Planning constraint-satisfying paths can thus be reduced to determining the constraint-satisfying configurations and connecting them to form a path. For robots with more degrees of freedom, the problem can become much more complex.

Equality and Inequality Constraints

In order to understand constrained path planning, this section introduces some fundamental concepts. An often-used concept is that of the *constraint manifold* [71]. Given that the configuration space of the complete planning problem is Q, the constraint manifold X_f of a constraint f is defined as the set of all configurations of Q that satisfy f:

$$X_f = \{\mathbf{q} \in Q | f(\mathbf{q}) = 0\} \tag{6.1}$$

The volume of the constraint manifold is important for analyzing the complexity of handling the constraint. The case where X_f has a positive volume in Q is the less problematic one: elements of X_f can be found efficiently by sampling uniformly from Q. This is the case, for example, with obstacle avoidance constraints. The subspace of configurations not in collision with an obstacle has, except in rare circumstances, a positive volume. Configurations in X_f can be found by sampling and checking in Q, as samples of X_f are generated with a positive probability.



(a) Equality constraint **(b)** Inequality constraint

Figure 6.1. Constraint manifold of a two-dimensional robot for equality and inequality constraints.

More difficult to handle is the case where X_f has zero volume in Q. This is the case, for example, when the end-effector is constrained to move in a line. X_f thus has the shape of the line, which has zero volume in the ambient space Q. Similarly, constraining the end-effector to a table surface induces a manifold in the shape of a two-dimensional plane. Constraining the end-effector to have a fixed distance to some target point means that X_f has the shape of a sphere, with the end-effector moving on its surface. Constraint-satisfying configurations can no longer be found through sampling Q as the probability of generating a sample that lies in X_f is also zero. Generating samples directly from X_f is possible in some special cases, but not generally applicable.

Figure 6.1 shows an illustration of the constraint manifold of an equality and an inequality constraint of a two-dimensional robot. In the first image, the end-effector is constrained to a given height, and the constraint manifold, shown in green, forms a one-dimensional line. The second picture shows an obstacle avoidance constraint as an example of an inequality constraint. The end-effector is constrained not to move too closely to the robot's base and not too close to the obstacle shown in red. While the green constraint manifold is of course smaller than the full configuration space, it still has positive volume. Randomly sampling robot configurations will have a high probability of fulfilling the constraint.

Depending on the volume of the constraint manifold, constraints often require different procedures to enforce them. Thus, we categorize constraints into these two types:

- **Equality constraints** have a constraint manifold of zero volume in the surrounding ambient space, i. e. the full configuration space of the planning problem. Therefore, they effectively reduce the available degrees of freedom in the planning problem. While this may sound like a simplification at first, it often makes the problem more difficult. This is due to the facts that the constraint manifold can often not be directly sampled from, and that a path on the constraint manifold is not necessarily executable in the full configuration space. Therefore, planning has to consider both the full configuration space, as well as the constraint manifold.

  An example is a constraint constraining the end-effector to a certain height. The constraint manifold embedded in Cartesian space thus has one less dimension than the surrounding space as the height is no longer allowed to vary. A similar example is a constraint enforcing that the end-effector has a fixed distance to a target point. The constraint manifold of the Cartesian positions thus has the shape of a sphere, and positions on the sphere can be described as two-dimensional coordinates.

- **Inequality constraints** are constraints where the constraint manifold has a positive volume. Examples include obstacle avoidance and joint limits. It should be noted that the term *inequality constraint* do not directly correspond to an inequality in our formulation of constraints. As defined in section 5.2, we only consider constraints to be satisfied if $f(\mathbf{q}) = 0$. In other formulations, constraints requiring only the inequality $f(\mathbf{q}) \leq 0$ are used, which inherently have a constraint manifold of positive volume. As such, the term *inequality constraint* has become widespread for constraints with a positive constraint manifold. We follow this terminology, even though they are not actually described by an inequality in our case.

In the related literature, constraints in robot motion planning are often implicitly assumed to be equality constraints [71], as inequality constraints are relatively simple to handle and already included in standard algorithms such as the RRT. The constraints are further seen as a secondary condition to be fulfilled, while the main task of moving the robot to a goal is typically not expressed through constraints. It is generally assumed that a constraint-satisfying starting configuration and goal are provided. In our work, this is not the case: when an action is called, it can not be guaranteed that the robot already fulfills its constraints, and goal configurations are not available either. Moreover, constraints include both equality and inequality constraints. Thus, our planning method requires the following additional steps, which are typically not considered in constrained planning research:

1. Find a suitable starting configuration, and move to it.
2. Find suitable goals.
3. Create a planning problem from the action definition, that handles both equality and inequality constraints in efficient ways.

Soft Constraints

Another classification of constraints, that is often encountered in fields dealing with constraints, is that of *hard* and *soft* constraints. Hard constraints are constraints that have to be satisfied for a solution to be valid. Soft constraints should be satisfied if possible, and if this is not possible, the solution should still be as close as possible to a satisfying solution. The question by which metric this closeness to a satisfying solution is measured has no simple answer in the general research area of constraint solving. Choices include the total number of violated constraints, possibly weighted, or constraint preferences, which define an order of importance on the constraints [123].

Soft constraints in sampling-based planning have not received a lot of research attention, although some works have been published on the topic [76, 143]. Soft constraints with a well-defined formulation of constraint importance add considerable amount of complexity to the planning problem, and the existing approaches are quite restrictive to their respective applications. Given the probabilistic nature of sampling-based planning, optimal results for soft constraints are a challenge. As even optimizing sampling-based planners such as the RRT^* can only achieve asymptotic optimality, it can not be decided in finite time whether it is *impossible* to fulfill a soft constraint. The requirement that soft constraints should only be deviated from if they are impossible to fulfill is thus generally not practically realizable.

As we are only considering geometric constraints, we assume that distance to the closest satisfying configuration is the most appropriate metric, rather than counting the number of violated constraints. In practical path planning, soft constraints are usually approximated as hard constraints with a permissible tolerance for deviation, or as path costs. When using path costs, the error values of the soft constraints are included in the calculation of the path costs that is optimized by optimizing planners. While this technique can not guarantee that the soft constraints will be satisfied if possible, it will attempt to keep the deviation small. Using hard constraints with a tolerance interval, on the other hand, means that any generated solution will lie within the tolerance around the soft constraint, but no attempt is made to exactly fulfill the soft constraint. Our method of planning, which is presented below, is able to express soft constraints both as path costs to be optimized, and as permissible tolerances on hard constraints.

Planning with hard equality constraints

We continue this section by giving an overview of the existing research on constrained path planning with hard equality constraints. In section 6.3, we describe the planning algorithms we have used and how the constraint model of chapter 5, which is generally over-constrained and uses constraints with priorities, has been adapted to make use of existing work on constrained planning. Equality constraints are generally a challenge for sampling-based planners, because of the difficulty of generating samples that lie on the lower-dimensional constraint manifold. Uniform

sampling only creates constraint-satisfying samples with a probability of zero. Equality constraints thus need special treatment for sampling-based planning and are the focus of most related work on constrained sampling-based planning. Methods of handling constraints in sampling-based planning methods can be classified into different categories:

1. **Sample Rejection:** Whenever a sample configuration is drawn, it is checked whether it satisfies all relevant constraints. If not, the sample is rejected and another sample will be drawn [10, 11, 119]. This is the simplest, yet often inefficient method of enforcing path constraints. A big advantage of this method is that it only requires the ability to check whether the constraints are satisfied or not, and no further knowledge about the constraints is needed. Sample Rejection is typically used for inequality constraints, such as obstacle avoidance in classical planning algorithms. This method is not recommended for handling equality constraints.

2. **Constraint Relaxation:** The constraint requirements are *relaxed*, so that a constraint is considered satisfied not only if $f(\mathbf{q}) = 0$, but also if $\|f(\mathbf{q})\| < \epsilon$, where $\epsilon \in \mathbb{R}_{\geq 0}$ is a real number that expresses the amount of relaxation. With this change, constraint-satisfying configurations of equality constraints can be sampled with a probability larger than zero and sampling-based planners can be used without further changes. However, there are two downsides to this method: First, even the relaxed constraints can cause problems for planning methods. The relaxed constraints are in effect equivalent to narrow passages, which are a well-known problem for classical planners [8]. The second downside is that the resulting plan will not satisfy the constraints exactly. Depending on the details of the application, this can be acceptable. Often, exact enforcement of the constraints is delegated to the controller executing the plan instead.

3. **Sample Projection:** The idea of this method is to use (possibly) constraint-violating samples to create constraint-satisfying samples. After a random sample is generated, an iterative optimization procedure is used to attempt to find a constraint-satisfying sample from the original sample [9, 96]. Usually, this is based on various forms of gradient descent. Mathematically, this mapping is expressed as the *projection operator* $P(q) : \mathbb{R}^n \to Q$, with $Q \subseteq \mathbb{R}^n$ and $f(q) = 0$ for all $q \in Q$. This method can be used with most sampling-based algorithms simply by replacing the *sample*() procedure with a procedure that uses the projection operator. It is then said that the algorithm is using a *projected state space*. While this method is among the most widely used methods for constrained planning, it does have downsides as well. In order to perform a gradient descent, the constraint functions must not only decide whether a configuration is satisfying or not, but also encode the distance to a satisfying configuration. Furthermore, the gradient of the constraints must be available. Lastly, the calculation of the projection can be computationally expensive.

4. **Tangent Spaces:** Using a known constraint-satisfying configuration (beginning with the start of the plan), new constraint-satisfying configurations are found by constructing the tangent space of the constraint functions, a locally linear approximation of the constraint manifold [70]. The computation of tangent spaces requires computing multiple matrix decompositions and can be computationally intensive. This requires that the constraint functions define a manifold. Methods based on tangent spaces do not work well if the constraint manifolds are not easily approximated by linear functions, for example when the constraint manifolds are highly curved. These methods can also not be used if the Jacobian does not have full rank. So-called *Atlas*-based methods [63] store the tangent spaces to avoid re-computations. While tangent-space-based methods have been shown to be computationally efficient, they require comparatively strict mathematical preconditions, which restrict them to specific use cases.
5. **Direct Sampling with Reparametrization:** This type of method aims to create new configuration spaces, of which constraint-satisfying configuration can be directly sampled without requiring further processing of the samples. The robot's geometry and knowledge about the constraints are used to create a reparameterized configuration space, with a mapping to the original configuration space [94, 53]. On top of this new configuration space, any traditional (sampling-based) planner can be used. This approach is in principle very elegant, as the constrained planning problem is reduced to unconstrained planning. It does have serious downsides though, which prevent general use. The reparametrization has to be calculated anew for each type of robot and each constraint specification. In many cases, reparametrization is not possible at all. Hence, this method can not be used as a general method and heavily depends on the specific properties of the robot and the constraints.

To summarize, constraints in path planning are an active field of research with various different approaches being developed. However, none of the currently available methods are generally usable independent of the characteristics of the robot and the constraints. Therefore, the user has to be aware of this and make appropriate choices in the selection of algorithms. Since there is no method suitable for all types of constraints, multiple approaches are implemented in our framework, drawing from sample projection, constraint relaxation, and sample rejection methods. They are described in detail in section 6.3. Methods based on tangent spaces and reparametrization have not been implemented due to their highly specific requirements.

6.1.4 Planning in Dynamic Environments

Classical path planning methods compute a single solution for a given, static environment. In dynamic environments, changes in the environment need to be considered as well. When the environment changes, a previous solution might become invalid, a better solution might become available, or the goal could move somewhere else. The simplest solution to changing environments is to stop the execution of the plan as soon as something changes and create a new plan. This is a computationally expensive solution, however, as the previous results are completely discarded. Various algorithms have been proposed to re-plan in a new environment while keeping some of the previously calculated results.

The execution-extended RRT [12] stores waypoints and uses them to generate new trees. The dynamic RRT [32] places the root of the tree at the goal, so that the same tree can be used to find alternate paths to the goal if some nodes become invalid. This assumes that the goal itself will not change. The multipartite RRT [149] keeps multiple subtrees in memory that are dynamically pruned and reconnected. RRT^X [107] is an approach that keeps a graph that is re-wired and repaired whenever an obstacle or the robot moves. More recently, EB-RRT [144] uses two hierarchical planners with dynamic replanning based on the elastic band method.

All of the above approaches have in common that they are focused on mobile robots navigating on a plane, with typically only three degrees of freedom. The computational cost increases substantially when applying them to nine-dimensional problems. The following assumptions about the use case of our work largely prevent the application of the existing algorithms:

- We assume no information about the future development of the environment, i. e. the constraints. Neither in the form of concrete predictions nor probabilities.
- Constraints are used to define the cost of a path, the goal of the plan, and the invalid configurations (for example, obstacles). Any of these can change at any time. Many of the existing algorithms presuppose that, for example, the goal stays the same, of *only* the goal can change.
- Replanning has to be done for all nine degrees of freedom of the robot, considerably more than the three of planar mobile robots, which are mostly considered in the related work.

To summarize, while various methods for sampling-based planning in dynamic environments have been created, they are typically restricted to low-dimensional problems and make restrictive assumptions about the nature of the dynamic environment that do not hold in our use case. Therefore, we are not using specific algorithms for planning in dynamic environments. Instead, our method relies on the safe execution using the reactive controller, and possibly planning anew if necessary.

Model-predictive Control

A completely different approach to handling dynamic obstacles is model-predictive control. This approach does not create a path that is then to be executed by the controller, but instead extends the formulation of the control problem to include optimization over a prediction of the future as well. This method has been explored for obstacle avoidance in ground and flying vehicles [82, 40]. Model-predictive control has also been applied to mobile manipulators, although in very limited use cases that only contain basic obstacle avoidance while tracking given trajectories [13, 80].

However, we do not consider it suitable for our use case due to two reasons: First, the optimization over multiple steps into the future becomes costly very quickly with increasing time horizons. Since we are interested in larger-scale plans, such as navigation between rooms, rather than quick reactions in a driving vehicle, this is problematic for our application. Second, model-predictive control requires an exact model of the robot as well as its environment that can be predicted in to the future. This is not available in our use case, and is difficult to produce for human interaction partners.

6.2 Planning Pipeline

A planning method for the system described here needs to be able to produce a complete motion path given only a high-level action definition, as well as the current environment and robot state. Additionally, the plan needs to be executed safely in a dynamic environment. These requirements mean that more steps are necessary than the core planning algorithm itself. The pipeline of our planning approach is graphically illustrated in figure 6.2.

The pipeline starts by determining suitable start configurations. This is only necessary in cases where the planner is requested to produce a path satisfying constraints that are not fulfilled in the current configuration.

In the next step, suitable goals for the action definition have to be determined. Here, the action definition needs to be converted to concrete joint positions to use as goal configurations in the planning process. If the action definition is only a Cartesian pose, this is the well-known problem of inverse kinematics. Our action definitions are much more general and do not necessarily define a target Cartesian pose, but various constraints. This step is thus referred to as the *generalized* inverse kinematics.

The next step in the pipeline is the core of the planning process. This is where classical planning algorithms, such as rapidly-exploring random trees, are used to find a path from the starting configuration to a goal. Since the resulting paths can contain unnecessary detours and are often not smooth, this step is followed by a post-processing step that attempts to simplify and smoothen the path. All the previous steps are calculated using the environment state as it was when the planning request was received. To ensure that the plan is still applicable in a

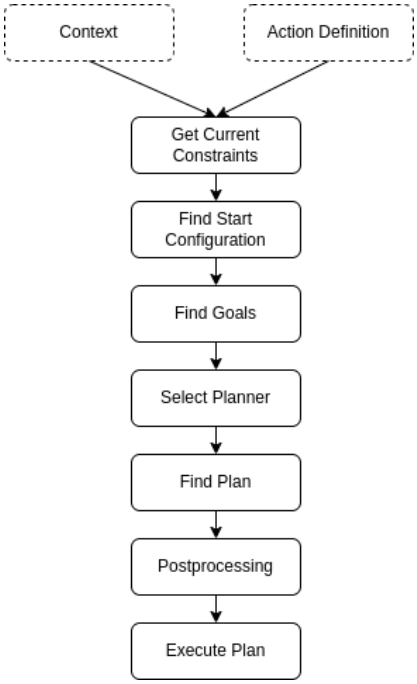


Figure 6.2. The pipeline of our planning method.

possibly changed environment, the plan is checked using the updated environment state. If the plan is no longer safe, the pipeline starts from the beginning, if there is still available time.

Lastly, the plan is executed safely, using the reactive control scheme described in the previous chapter to safely react to any unforeseen circumstances. Each of the steps in the pipeline is described in more detail in the following sections.

6.3 Handling Path Constraints

As has been described in the previous section, not everything that is expressed as constraint in the control system described in chapter 5 can be handled in the same way during planning.

Goal constraints describe only the goal of the plan, as the name already suggests, and have no further relevance to the path leading to them. Inequality constraints such as obstacle avoidance are classically implemented through a simple sample rejection process. This a well-studied and understood method for collision-free path planning, and should not be altered without good reason. However, equality constraints such as holding a glass of water upright or keeping in touch with a table surface, for example for cleaning or drawing, can not be efficiently handled in this way.

However, this problem has also been studied and methods to handle such constraints have been developed, such as sample projection. For constraints of this type, these methods should hence be used.

Lastly, *cost* constraints should be considered during the path as far as possible, but it is assumed that a costly path is still preferred over no path. These constraints should thus be considered when optimizing a path, but not lead to full rejection of samples.

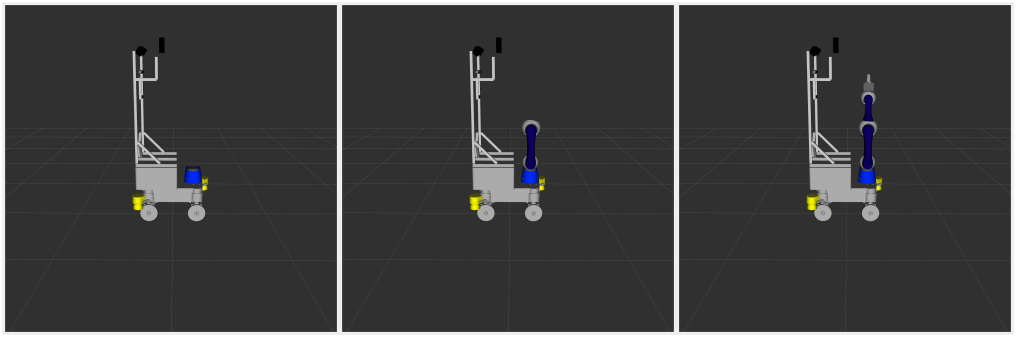
6.3.1 Path constraints in multi-level planning

To make use of multi-level planning, such as the QRRT algorithm, the validity of configurations has to be decided for simplified configuration spaces, which are in our case formed by leaving out links of the robot. This is easy to do for simple collision checking, as just the remaining parts can be checked for collision. For the more general constraint specifications that we use, this is not as simple.

For example, constraints on the orientation of the end-effector can not be decided if only the base configuration is known in the simplified configuration space. For these types of constraints, their validity check can thus only be performed in the full joint space. If possible, constraints should be evaluated as early as possible however to exploit the simplified configuration space the most. For example, if the platform orientation is constrained, this should already be checked in the simplified configuration space so that unfeasible configurations are pruned as early as possible. Hence, an automatic method to decide which constraints can be evaluated on each simplified configuration space is required.

We consider only the case where degrees of freedom and their associated links are removed from the kinematic tree to form simplified configuration spaces. Only leaves or subtrees whose leaves are leaves of the original tree can be removed, so that the resulting structure still forms a tree. We assume that the simplified spaces are pre-defined by the user. Given the set of active tasks of the active action, it has to be determined for each task whether its constraint can be decided on each configuration space or not. Since the simplified configuration spaces are formed by removing joints, the configuration spaces are identified by the set of included joints.

For joint-space tasks, providing constraints directly on the joints with the Jacobian being the identity matrix, this is a simple procedure. Generally, all joints are required to evaluate these constraints, unless the weights of some joints are explicitly set to zero. Thus, the task can be evaluated on the joints of a simplified configuration space if and only if all joints with non-zero weight are included in the configuration space. For Cartesian tasks, this is slightly more involved. All joints that have an effect on the controlled link as well as the reference frame have to be determined. This is done by a traversal of the kinematic tree. If all the resulting joints are included in the configuration space, the task can be evaluated. Given a sequence of configuration spaces with ALDS, the constraints are evaluated with the QRRT algorithm on each configuration space where it is possible to evaluate them.



(a) 3 Joints **(b)** 6 Joints **(c)** 9 Joints

Figure 6.3. Visualization of the three configuration spaces used with multi-level planning.

In the practical evaluations, we have used two simplified configuration spaces, formed by removing the first three and all six of the arm joints. Figure 6.3 shows a visualization of the remaining robot links. In the evaluation, we refer to the QRRT algorithm applied to all three of the configuration spaces as $QRRT_3$. If using only the first and the last configuration space, we refer to the algorithm as $QRRT_2$.

6.4 Finding Goals from Constraints

Most planning algorithms assume that the goal configuration is known explicitly. In our case, only the action definition is available, and goals need to be determined from this definition. For some planners, multiple goals can be used. If a goal is given in Cartesian space, inverse kinematics are used to calculate a configuration. For non-redundant robots, this is trivial. For redundant robots, however, the redundancy needs to be resolved appropriately.

Specifically for mobile manipulators, inverse kinematics methods have been developed that express the available redundancy as explicit *redundancy parameters*, which can then be chosen in optimal ways [129, 3].

However, this still assumes that the goal is explicitly given as a Cartesian pose, with the rest of the available redundancy being only considered afterthoughts. In contrast, when planning for the satisfaction of a set of arbitrary geometric constraints, In our framework, it can not be assumed that the goal is given as a Cartesian pose or a complete configuration. Instead, only a high-level description of the goal in the form of geometric constraints is available.

For some planners, such as the basic RRT and RRT*, being able to decide whether a configuration satisfies the goal constraint is enough: The tree is simply expanded until a goal-satisfying configuration has been added to the tree. Many of the most successful variants of the RRT require explicitly known goals though. This includes planning with a goal bias, bidirectional planning, and multi-level planning.

Algorithm 6.2 Algorithm for the calculation of goal configurations.

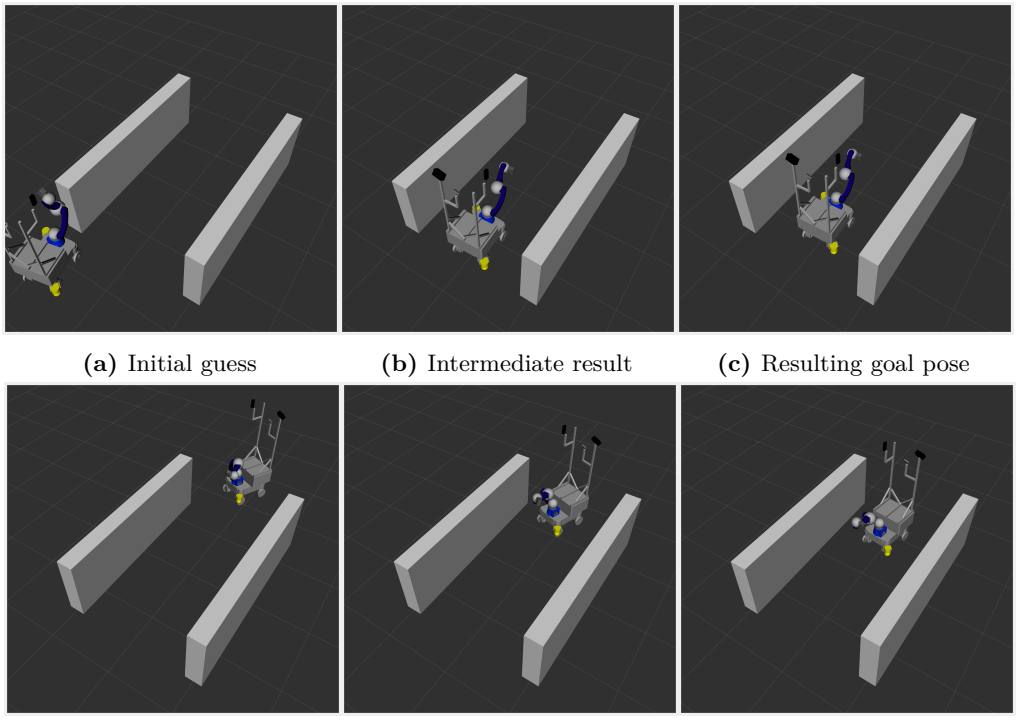
Input:
Constraint function f,
Constraint Jacobian $\mathbf{J}$,
Initial Configuration $\mathbf{q}$
Maximum number of iterations k
Output: Set of goal configurations Q
$Q \leftarrow \emptyset$;
$i \leftarrow 0$;
repeat
  $\mathbf{q} \leftarrow \mathbf{q} + \mathbf{J}^{\dagger} f(\mathbf{q})$
  if $f(\mathbf{q}) = 0$ **then**
    $Q \leftarrow Q \cup \{\mathbf{q}\}$
    $q \leftarrow randomConfiguration()$;
  end if
  $i \leftarrow i + 1$;
until $i = k$
return Q

Therefore, a method to find goal-satisfying configurations from a specification of goal constraints is required. Our method is using the pseudoinverse of the constraint Jacobian, notated as $\mathbf{J}^{\dagger}$, which can be found through singular value decomposition.

The procedure $randomConfiguration()$ computes a configuration within the joint limits $[\mathbf{q_{min}}, \mathbf{q_{max}}]$ with a uniform distribution. In our implementation, one second is given as the maximum duration for the process of finding goals. If no valid goals can be found in this time, the request has failed. If more than 10 goal configurations are found, the process terminates early. Figure 6.4 shows illustrations of two goal-finding processes and their intermediate states.

6.5 Initial configuration

During path planning, constraints from both *safety* and *path* tasks are initially seen as hard constraints for the entire path. There is however one case where they are handled in different ways: It arises when, in the current configuration, i. e. the start of the plan, the hard constraints are not satisfied. If the safety constraints are not fulfilled, the planning fails, as there is no valid initial state. This is seen as sensible behavior, as the robot should not be able to move from a position that violated safety constraints. However, for path constraints, this can be an impractical assumption. Using path constraints would require the user to ensure that the path constraints are fully satisfied before requesting a plan. Often, the behavior that the user intends will be to generate a motion that satisfies the path constraints as quickly as possible, and then proceeds with a constraint-satisfying path from there.



(a) Initial guess **(b)** Intermediate result **(c)** Resulting goal pose

(d) Second initial guess **(e)** Intermediate result **(f)** Resulting goal pose

Figure 6.4. Visualization of the goal-finding process.

Of course, there are limits to the ability to find a path constraint-satisfying initial configuration. Both in terms of computational resources, and in terms of what is considered sensible behavior. If the user requests a plan with the path constraint of keeping the gripper upright, but the gripper is currently not exactly upright, it is usually intended that the motion starts by moving the gripper to a fully upright position. If, however, moving the gripper to an upright position would require a larger motion, making room around obstacles etc., it might be more sensible to report this to the user instead of autonomously executing larger motions that do not conform to the constraints.

Thus, parameters can be set by the user to limit the number of iterations, and the maximum distance between the current configuration and the newly found solution. Finding a constraint-satisfying initial configuration is done using a projection method based on the pseudo-inverse of the constraint Jacobian. The algorithm is shown in algorithm 6.3. Since this algorithm is simply descending along the direction defined by the constraint Jacobians, it can be assumed that it can easily be reached from the current configuration using the reactive control scheme. For more details about the execution of plans using the reactive controller, see the following chapter 7.

Algorithm 6.3 Algorithm for the calculation of a valid initial configuration.

Input: Combined Constraint function of safety and path constraints f,
Constraint Jacobian $\mathbf{J}$,
Current Configuration $\mathbf{q}$,
Maximum number of iterations k
Maximum distance d_{max}
Output: Initial configuration q_{init}
$i \leftarrow 0$;
$q_{init} \leftarrow q$;
while $i < k$ and $f(\mathbf{q}_{init}) \neq 0$ and $\|q_{init} - q\| \leq d_{max}$ **do**
  $\mathbf{q} \leftarrow \mathbf{q} + \mathbf{J}^{\dagger} f(\mathbf{q})$
  if $f(\mathbf{q}) = 0$ **then**
    $Q \leftarrow Q \cup \{\mathbf{q}\}$
    $q \leftarrow RandomConfiguration()$;
  end if
  $i \leftarrow i + 1$;
end while
return q_{init}

6.6 Planning with Soft Constraints

As explained in section 6.1.3, soft constraints are approximated through two different ways: Either by defining a cost function to be optimized over the path, or by defining tolerances around hard constraints. The latter is simply specified by defining the vector of tolerances in the task definition, which is then automatically used when evaluating the task. Thus, no further steps are needed to use this method in conjunction with constrained planning algorithms.

Planning with a cost metric is also simple to add to most planning algorithms. The most important part is the definition of a suitable cost function. The cost of a sampled configuration $\mathbf{q}$ is calculated by the following equation, given a set of constraint functions F:

$$\text{cost}_{state}(\mathbf{q}) = \sum_{f \in F} \|f(\mathbf{q})\|$$

Many planning algorithms use the notion of a *motion cost* between two states instead of a state cost for individual states. The cost for a motion from $\mathbf{q}_1$ to $\mathbf{q}_2$ is defined as follows:

$$\text{cost}_{motion}(\mathbf{q}_1, \mathbf{q}_2) = \max\left(0, \alpha\left(\text{cost}_{state}(\mathbf{q}_2) - \text{cost}_{state}(\mathbf{q}_1)\right)\right) + (1-\alpha)\|\mathbf{q}_2 - \mathbf{q}_1\|$$

The cost of a motion thus considers the difference in the constraint costs, as well as the distance between the two configurations. The parameter $\alpha \in [0, 1]$ defines the relative importance of the two cost factors. In our experiments, $\alpha = 0.5$ has been used. The difference in state costs has a lower bound at 0 put on it, as negative cost is not compatible with the definition of most planning algorithms.

The definitions for the cost of a motion between two configurations can be extended to define the cost of a path, consisting of a sequence of configurations. The cost of a path P, consisting of a sequence of configuration waypoints $(\mathbf{q}_1, \dots \mathbf{q}_e)$ is defined as follows:

$$\text{cost}_{path}(P) = \sum_{i=2}^{e} \text{cost}_{motion}(\mathbf{q}_{i-1}, \mathbf{q}_i)$$

In the evaluations, the notion of path *length* is also used. It is defined as follows:

$$\text{length}(P) = \sum_{i=2}^{e} \|\mathbf{q}_i - \mathbf{q}_{i-1}\|$$

6.7 Evaluation and Results

In the preceding parts of this chapter, the state of the art in planning with geometric constraints has been described, as well as the approach for using an action specification to determine goal configurations, and find a plan for the goal while respecting soft- and hard constraints. Different approaches are compared regarding their efficiency and the quality of the results. The following algorithms have been evaluated:

- *RRT*
- RRT^*
- *TRRT*
- *RRTConnect*
- $QRRT_2$
- $QRRT_3$

The first four listed algorithms can be used for equality-constrained problems, simply by changing the underlying state space to a projected state space. For the last two, the multi-level RRT variants on quotient spaces, it is currently not clear how to apply them to constrained state spaces. The quotient space method uses a sequence of lower-dimensional spaces leading up to the original state space, while the projected state space method projects the original state space to the constraint manifold. Both methods thus build their own separate sub-state-spaces of the original state space. Combining them is not possible using their current formulations. It could be a fruitful direction for future research. Currently, the algorithms $QRRT_2$ and $QRRT_3$ can not be used for problems with equality constraints.

Normally, a planning process in our system starts by finding possible goals, after which the actual path planning starts. Since the nature of the goals has a large influence on planner performance, for this comparison we have pre-determined the goals and used the same goals for all planners. Giving statistics for the goal-finding procedure is difficult because it highly depends on random chance, the environment, and the given tasks. In the two evaluation scenarios we described, three valid goals were found within two seconds. Since the planner evaluation shows that a planning time under three seconds is well achievable, with some algorithms staying below one second, we assume that a duration of under five seconds that it takes for the robot to find a plan once it gets stuck is a realistic expectation. After planning, the resulting paths are simplified using standard smoothing and short-cutting algorithms. This is independent of the planning algorithm in use. Our implementation of the algorithms is based on the *Open Motion Planning Library* (OMPL) [139, 72]. The algorithms are evaluated in three different scenarios, described below.

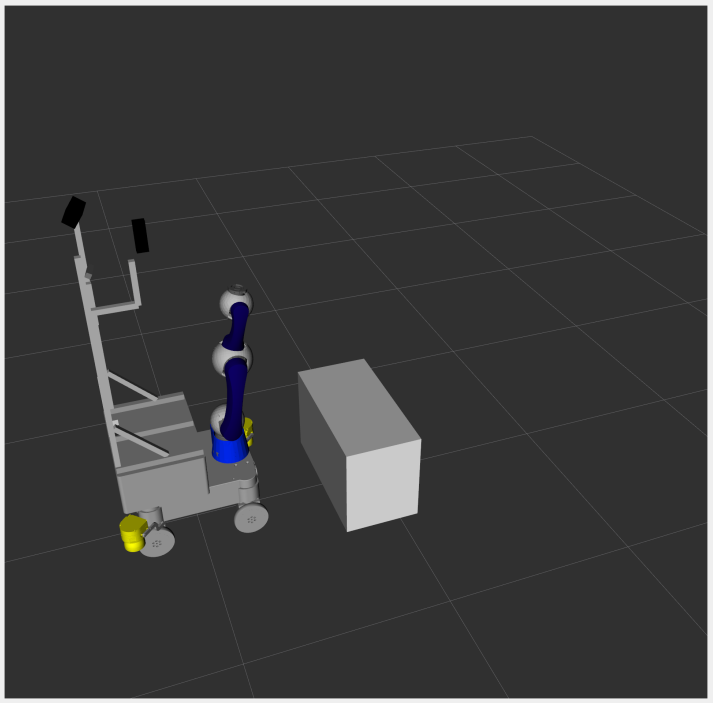


Figure 6.5. Visualization of the first evaluated planning scenario, *Cartesian Goal with Obstacle*.

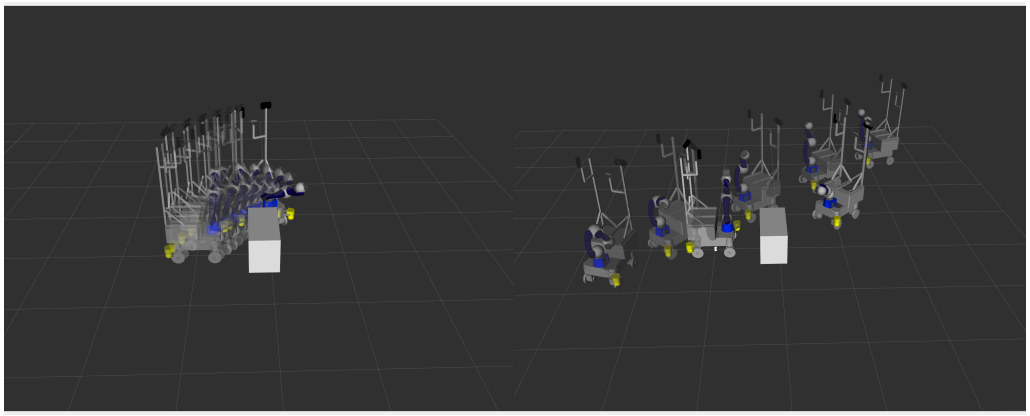
6.7.1 Cartesian Goal with Obstacle

The first evaluation scenario has the intention of testing the basic functionality of the different planning algorithms, without particular challenges. The action definition is the same as in section 5.7.3. The planners thus have to find a path to the Cartesian goal pose while avoiding the box obstacle. For this evaluation, projected state spaces are not used, as there a no applicable equality constraints in this scenario.

	RRT	*RRT**	*RRT-Connect*	*TRRT*	$QRRT_2$	$QRRT_3$
Failure rate	0%	0%	0%	0%	44%	71%
Avg. path cost	7.04	5.24	7.09	5.73	7.40	6.90
Avg. path length	7.04	5.24	7.09	5.73	7.40	6.90
Avg. time (s)	0.02	2.55	0.01	0.04	0.38	0.61
Max. path cost	11.28	7.35	11.42	8.72	13.56	9.88
Max. path length	11.28	7.35	11.42	8.72	13.56	9.88
Max. time (s)	0.05	3.0	0.03	0.06	2.67	1.90

Table 6.1. Planner results for the scenario *Cartesian Goal with Obstacle*

In this scenario, no cost constraints are specified. Path cost is thus equivalent to path length. Planning time is limited to three seconds to keep the planners usable



(a) Best Plan **(b)** Worst Plan

Figure 6.6. Visualization of the best and worst generated paths for the scenario *Cartesian Goal with Obstacle*.

in interactive scenarios. If the planner has not found a valid plan within three seconds, it is counted as a failure. The results are shown in table 6.1 The failure rate of all planners is zero, except for the multi-level planners. RRT^* takes by far the longest time. This can however be trivially explained by the fact that, as a optimizing planner, it uses all the time available to it. The $TRRT$ produces on average paths with cost almost as low as the RRT^*, in a much shorter time. RRT^* produced both the best paths on average, as well as the least bad paths, i. e. the maximum cost paths it produced were not as costly as those of the other planners. $RRT - Connect$ takes, both on average and maximally, the least time, but also produces relatively long paths. The classic RRT is quite similar. The multilevel planners appear to produce results that are not able to keep up with the other planners. In almost all metrics, they are considerably worse than the other planners. It can thus be concluded that the overhead of multi-level planning is not worth its benefits in this relatively low-dimensional state space of nine dimensions. In figure 6.7, the average path cost and planning time of the evaluated algorithms are compared. Judging only by the two criteria in the plot, the $TRRT$ appears to be the overall best algorithm in this scenario, considering the trade-off of cost and time.

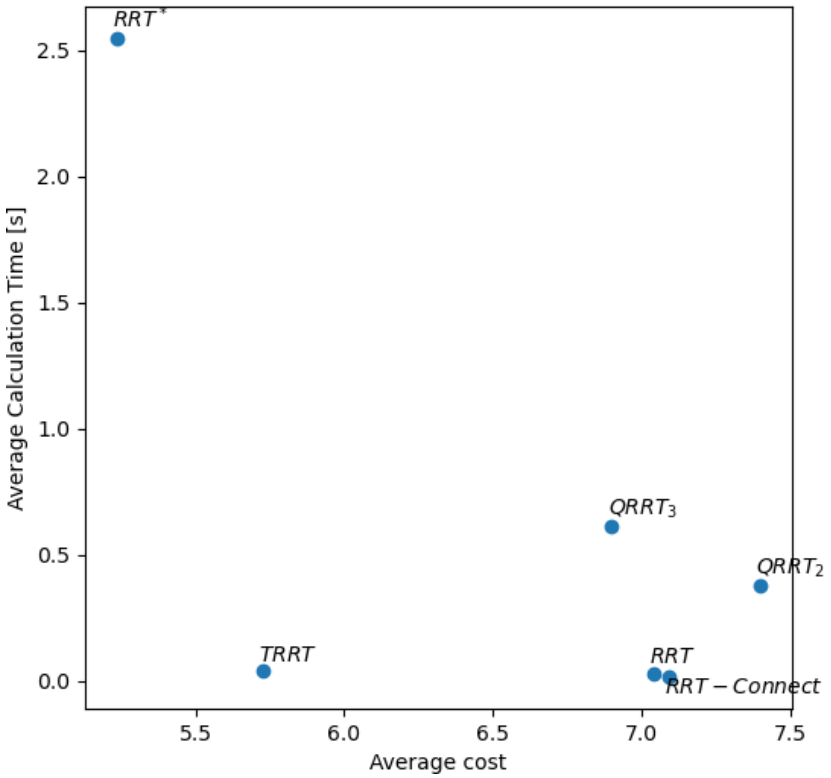


Figure 6.7. Average path cost and required planning time of the planners for the scenario *Cartesian Goal with Obstacle*.

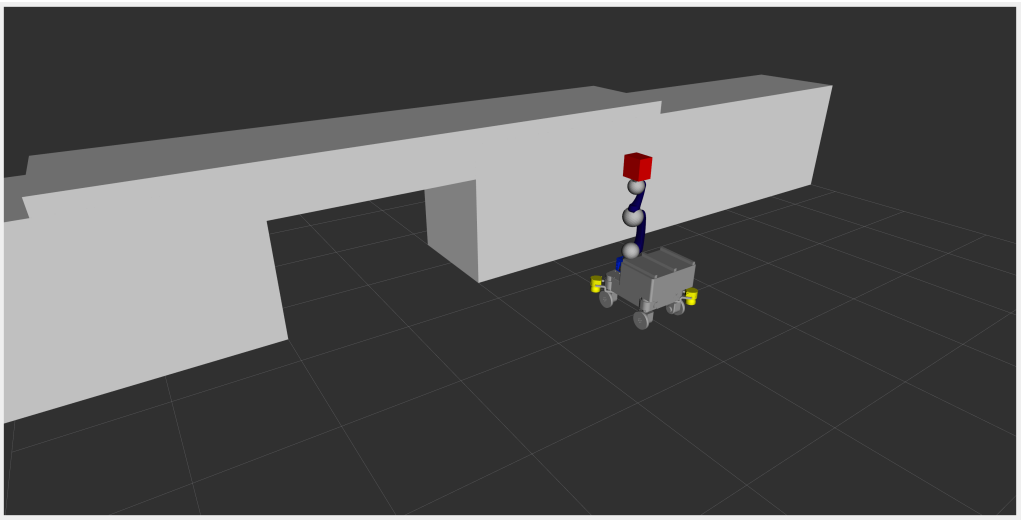


Figure 6.8. The *Narrow Passage* planning scenario. The collision geometry of the gripper is highlighted in red.

6.7.2 Narrow Passage

This scenario is designed to evaluate the limits of the algorithms in challenging conditions, even if they might not be realistic. The robot is tasked to move its base to a position that is behind a wall. The wall has only a narrow passage leading through it. Furthermore, the passage has a low ceiling, forcing the robot to lower its gripper. The constraint to keep the gripper pointing upwards must be respected at all times, making the lowering of the gripper more challenging. Even though the goal does not specify any further requirements for the arm, the planner has to fold the arm in order to fit through the narrow passage.

With the full robot hardware, there would be no feasible solutions due to the camera rack being taller than the arm. Therefore, the rack has been removed for this evaluation scenario. The planners have again been evaluated over 100 attempts each. The available time for each attempt was set to 20 seconds. The pipeline begins by trying to determine a constraint-satisfying start configuration. Figure 6.10 shows the resulting configuration. The action specification used for planning is shown in figure 6.9. The controllers are omitted for the sake of brevity, as they have no influence on planning. All the inputs are statically defined in configuration files. They are omitted from the diagram as well. The safety tasks are again the tasks shown in figure 5.17. Figure 6.13 shows a plot comparing the average required calculation time and average resulting path cost. All the planners are evaluated on a projected state space, with the projecting constraints being the one to hold the gripper in an upright orientation.

For comparison, the planners were evaluated without projection as well. In this case, the constraints were enforced only through sample rejection. None of them were able to produce any constraint-satisfying paths in this case. This is

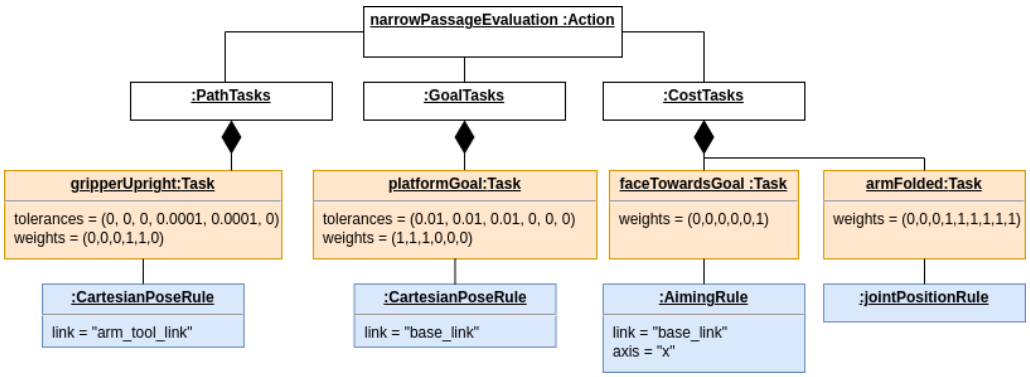
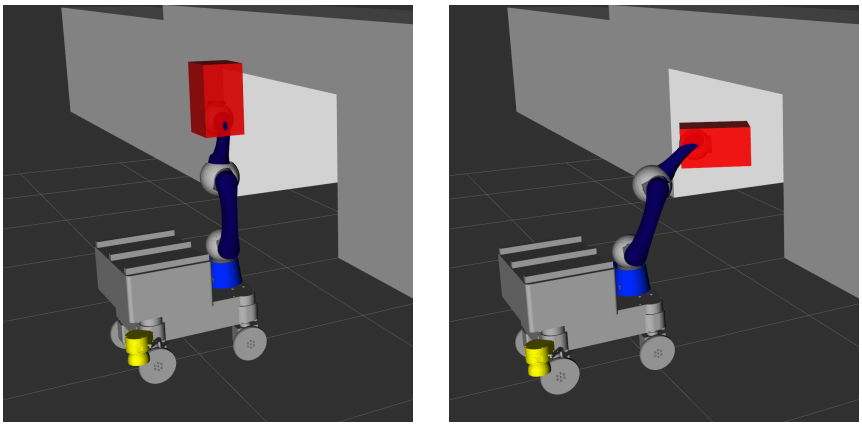


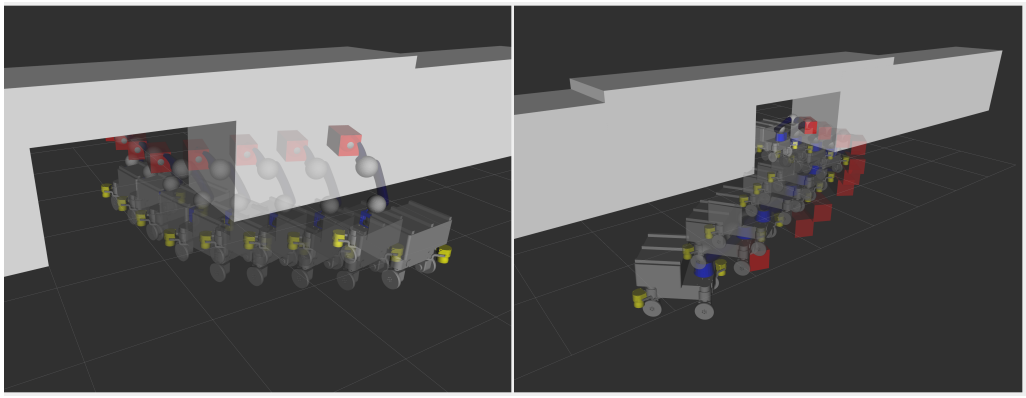
Figure 6.9. Object diagram of the action used in the *narrow passage* evaluation scenario. The global safety tasks are omitted here, and shown in figure 5.17 instead.



(a) Actual starting configuration **(b)** Result

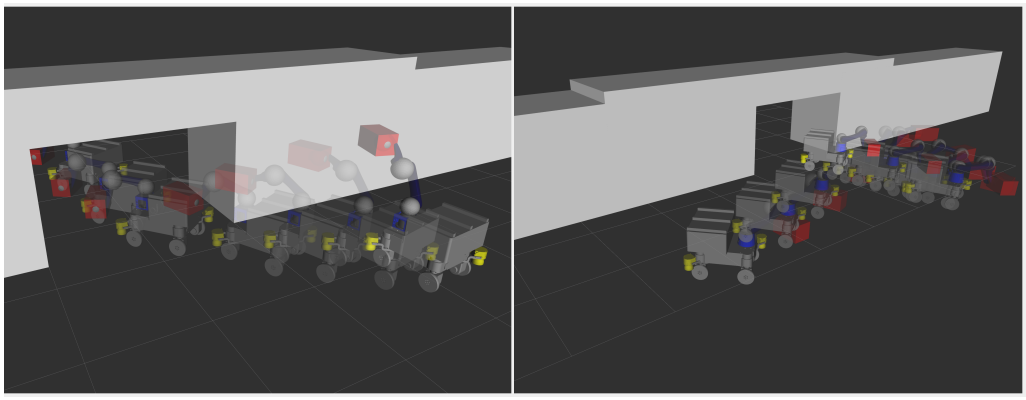
Figure 6.10. Finding a constraint-satisfying initial configuration in the *Narrow Passage* scenario.

expected, due to the very low probability of sampling suitable configurations. As in the previous example (figure 6.7), the TRRT algorithm appears to be the best candidate according to the presented metrics. It showed both faster calculation times and better resulting paths than the RRT and RRT^*. Only the $RRT - Connect$ produced faster, but also considerably worse results.



(a) **(b)**

Figure 6.11. Best path generated for the *Narrow Passage* scenario.



(a) **(b)**

Figure 6.12. Worst path generated for the *Narrow Passage* scenario.

	RRT	*RRT**	*RRTConnect*	*TRRT*
Failure rate	18%	19%	1%	3%
Avg. path cost	24.03	24.21	24.52	23.48
Avg. path length	10.78	10.11	10.96	10.60
Avg. time (s)	9.84	20.00	5.78	7.98
Max. path cost	31.84	29.52	46.98	31.04
Max. path length	12.54	11.61	19.96	14.89
Max. time (s)	19.23	20.00	14.61	18.57

Table 6.2. Planner results for the scenario *Narrow Passage*.

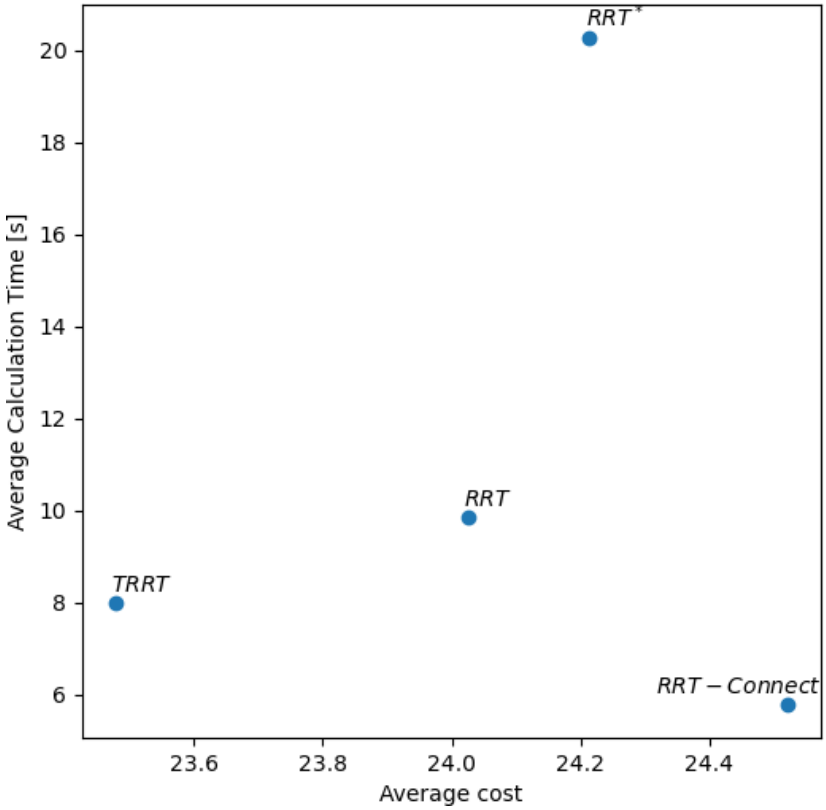


Figure 6.13. Average path cost and required planning time of the planners in the *Narrow Passage* evaluation scenario.

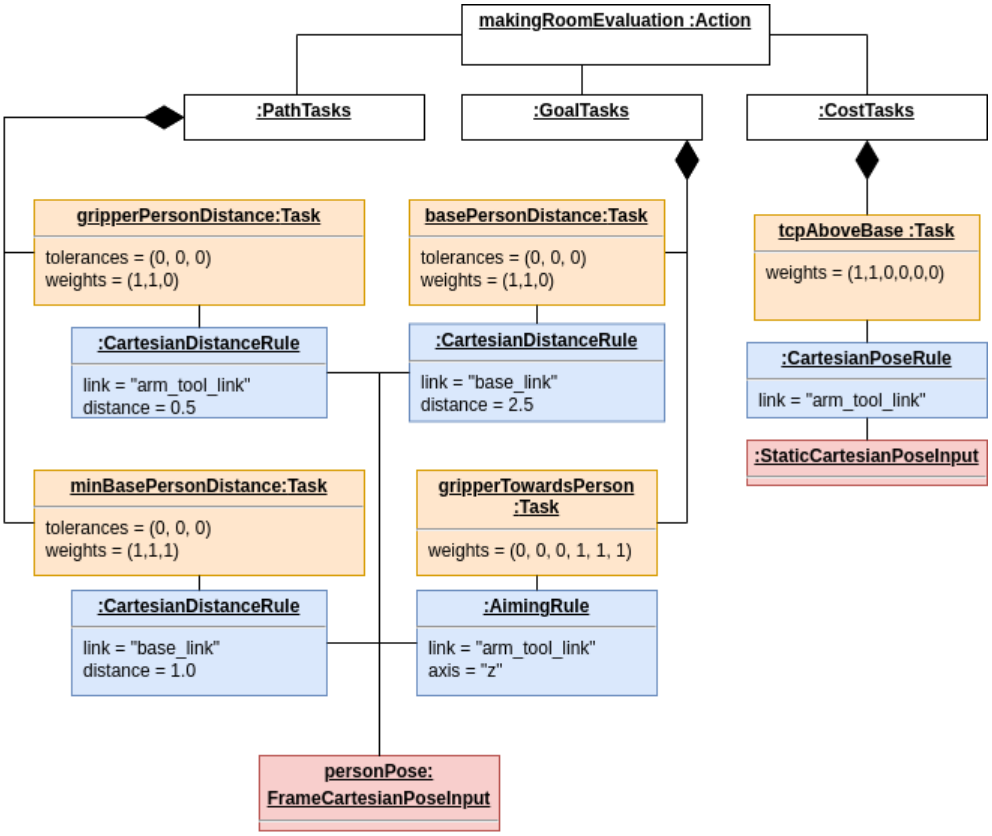


Figure 6.14. Object diagram of the action used in the *Making Room* evaluation scenario. The global safety tasks are omitted here, and shown in figure 5.17 instead.

6.7.3 Making room

This scenario is an example of a case where purely reactive control gets stuck in a local minimum. This can happen, for example, when the robot is given the specification to stay out of the way of a person, but ends up in a corner. If now the person is trying to move into the corner, the robot should still make room for the person. However, reactive control can not achieve this, as every possible motion leads it either closer to the walls, or closer to the person. Both go against the specified constraints. The aim of planning here is to make the robot realize that a temporary cost (moving closer to the person) will lead to a larger benefit (successfully making room) at the end of the plan.

The action definition is shown in figure 6.14. Controllers are omitted, considering that they have no influence in planning. Besides the shown tasks, the safety tasks as per figure 5.17 are also active. Two path tasks are used: both of them use a `CartesianDistanceRule`. The first is controlling the distance of the gripper to the person, the second is controlling the distance of the base to the person. In this path task, the target distance to the person, which has to be observed at all times, is set to the relatively low 1.0 m. In the goal tasks, however, there is

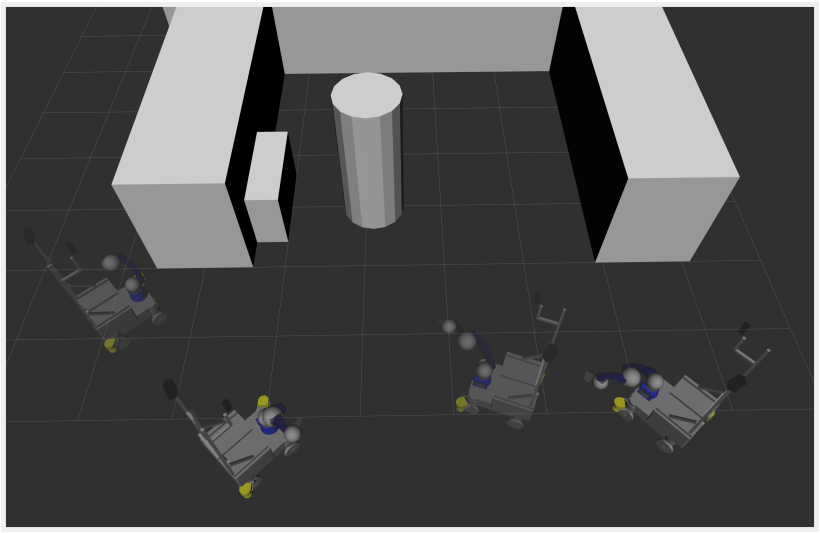


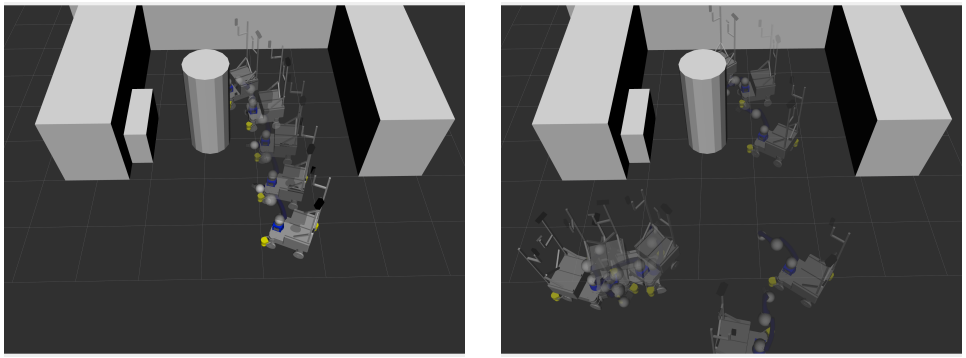
Figure 6.15. Some of the goal configurations determined for the *Making Room* scenario.

a second `CartesianDistanceRule` controlling the distance between person and base. This time, the target distance is set to 2.5. By specifying two distance values, once as a path task and once as a goal task, it can be expressed that the distance should never fall below 1.0 m, but ideally be above 2.5 m. Another goal task aims the z-axis of the gripper towards the person. This is used to let the robot shine its flashlight towards the person. The flashlight is not shown in the pictures of the simulation. Lastly, a single cost task is used to have the robot keep the gripper above the base as much as possible. This is intended to avoid configurations where the arm is extended far away from the base.

From a path planning perspective, the path to be found is not particularly challenging. The challenge lies much more in finding an appropriate goal. Classical planning approaches, which rely on a joint space or Cartesian target definition, can not easily express the goal of "making room". With our approach, it can be elegantly expressed. As this scenario requires equality constraints, the multilevel planners are not used here.

Some examples of the generated goal configurations are shown in figure 6.15. Ten goal configurations were found in 0.453 s, after which the goal-finding procedure ended. The figure also shows one weakness that currently exists in our method. The left-most goal configuration correctly points its end-effector towards the person, but the obstacles in the way prevent the light (not simulated) from reaching the person. To resolve this, an additional constraint rule taking visibility and occlusion of obstacles into account would have to be defined.

A plot of the average calculation times and resulting path costs is shown in Figure 6.17. In this scenario, choosing the *best* planner is less obvious than in the previous scenarios. Here, there appears to be a clear correlation between longer calculation times and path cost. No planner can be said to clearly outperform



(a) Best Plan **(b)** Worst Plan

Figure 6.16. Visualization of the best and worst generated paths for the *Making Room* scenario.

another. The choice must thus be made according to the criteria of how much planning time is available, and how important path costs are in the scenario. Figure 6.16 shows the best and worst plans that have been generated as examples.

	RRT	*RRT**	*RRTConnect*	*TRRT*
Failure rate	1%	0%	4%	2%
Avg. path cost	23.16	21.74	28.70	26.49
Avg. path length	11.24	9.93	13.66	12.35
Avg. time (s)	6.54	20.00	4.17	4.71
Max. path cost	63.94	43.77	71.54	54.95
Max. path length	29.32	22.22	34.12	26.31
Max. time (s)	16.64	20.00	17.64	14.00

Table 6.3. Planner results for the scenario *Making Room*.

6.7.4 Conclusions

Choosing planner algorithm and their respective parameters can be an arduous task, even for a clearly defined and static scenario. In the present case, very few assumptions can be made in advance about the nature of the planning scene. Important factors such as the ratio of invalid configurations, the distance of the goal and the properties of state costs depend critically on the action definition as well as the context they are applied in. Different planners have been evaluated in three different, exemplary planning scenarios. Multilevel planning did not outperform any of the other considered planners and will thus not be used further. The topic of selecting planner parameters has not been touched upon, instead, we rely on the defaults and partially automatic parameter selection provided by the OMPL. It is possible that manually tuning the parameters can lead to performance boosts. However, the unpredictability of the problem makes this difficult. Since the *TRRT* algorithm performed among the best in all evaluation scenarios, it will be used as the default algorithm.

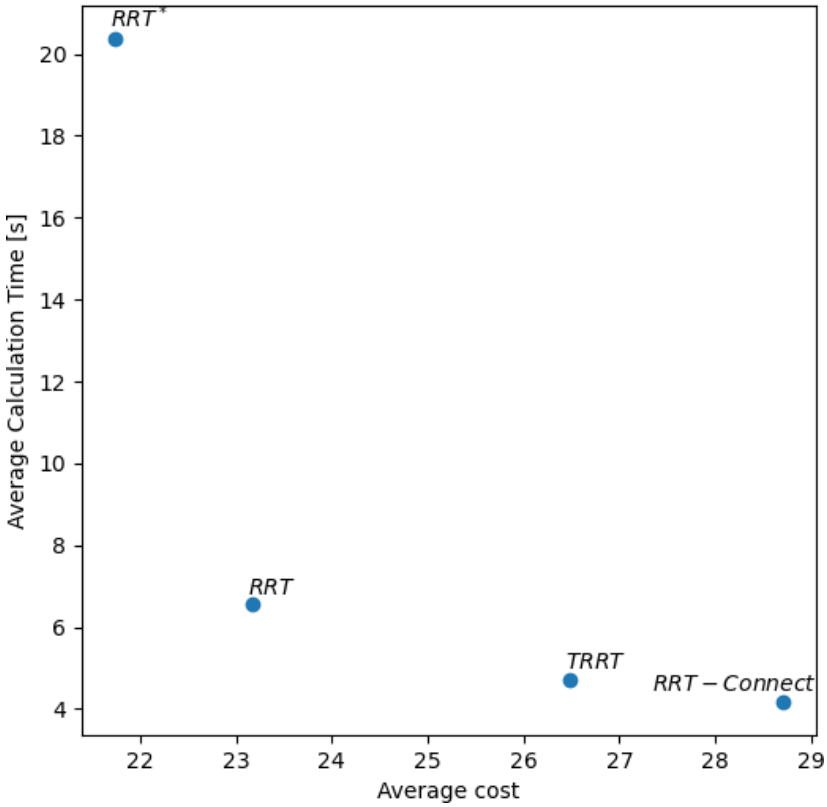


Figure 6.17. Average path cost and required planning time of the planners in the *Making Room* evaluation scenario.

Summary. This chapter explains how plans are executed safely in dynamic environments by using the reactive control method. Additionaly, action execution modes are presented that carry out a specified action while optionally using planning in different conditions.

7

Connecting Planning and Reactive Control

7.1 Execution of Plans in Dynamic Environments 131
7.1.1 State of the Art 132
7.1.2 Execution of plans in reactive control 133
7.2 Execution Modes 134
7.2.1 Reactive Action Execution 134
7.2.2 Planned Action Execution 135
7.2.3 Autoplanning Action Execution 137
7.2.4 Evaluation 138
7.3 Combining Actions to Behaviors 138

The previous chapter has described how plans can be created using an action specification. Still, it is not clear how the resulting plans as well as the process of planning are connected with the reactive control scheme. The two main questions to answer are how the created plans can be executed safely in a dynamic environment, and when planning is initiated at all. Our solutions to these questions are described in this chapter.

7.1 Execution of Plans in Dynamic Environments

The combination of motion plans and reactive control in dynamic environments causes a conflict between the static plan and the possible changes in the environment. Even after a plan exists, its execution takes some amount of time, during which the environment will possibly change. Blindly executing the plan is therefore not a safe approach, nor is it guaranteed that the plan will still be able to fulfill its original goals.

7.1.1 State of the Art

Widespread software solutions for manipulator motion planning, such as the ones typically used in ROS [19], do not include solutions for the safe execution of the resulting plans in dynamic environments. Only basic types of monitoring, such as an upper bound on the position error and execution types, are supported. These errors are not typically caused by a changing environment, but more likely due to failures in the robot hardware.

Frameworks for two-dimensional navigation typically include some methods for adapting the plan due unforeseen environments [88]. A classical solution is the use of *global* and *local* planning: The global planner creates the full plan for the goal using the data available at planning time. The local planner is tasked with executing the global plan. Therefore, it cyclically creates a short-term plan that followsthe waypoints provided by the global plan while taking the new information about the changed environment into account.

Since navigation deals only with movements on the ground plane, the control space is relatively small and solutions can be found much more easily, compared to the nine-dimensional case. A classical example of an algorithm for two-dimensional local planning is the dynamic window approach [38]. In each local planning cycle, control commands are randomly sampled and scored based on the currently available information. The scoring includes factors such as distance to obstacles and distance to the next waypoint. A defined number of commands are sampled and scored, and the best-scoring command is then sent to the robot. For a nine-dimensional robot, it is not possible to efficiently search the available motions by random sampling in a reasonable time. Hence, such approaches are not applicable to our case.

Another, more modern approach to the problem is the timed elastic band [120, 146]. The local planner is described as an elastic band, whose initial shape follows the path generated by the global planner. Artificial forces are formulated that deform the elastic band in real-time, to create a short and smooth local path that maintains distance from the obstacles. The elastic band continues to deform as the environment changes. Thus, the robot can handle uncertainties and react to unexpected dynamic obstacles.

While this method has been proven to be highly successful for navigation scenarios, it can not easily be generalized to robots with a larger number of degrees of freedom. Magyar et al. [90] have used it for manipulation planning, this still only works for control of a single frame in Cartesian space. More general control of multiple Cartesian and joint-space tasks is more challenging to express as an elastic band. The types of tasks that we consider can not be simply considered as artificial forces on the robot. Instead, the QP problem would have to be solved at least once for every point on the elastic band to find a suitable deformation. This, however, would be far too computationally intensive for practical application.

Our approach instead is simply using the waypoints of the plan as intermediate goals for the reactive control. The safety of the plan and reaction to obstacles are maintained by the reactive controller. What this method can not achieve at the moment, compared to the popular local planners used in navigation, is the avoidance of local minima that can occur during the execution of the plan. This happens, for example, if a dynamic obstacle blocks the way to the next waypoint on the path. Local planners would be able to plan around this obstacle to still follow the plan, at least to the extent that their range of motion forecasting allows. Our approach, on the other hand, would first get stuck, before realizing this and creating a new plan. This is of course less ideal, but the only method currently feasible for the type of complex control formulations that we use. It could be a possible direction of future research to explore local planning for the control problem of the system described here.

7.1.2 Execution of plans in reactive control

As explained in the previous subsection, the plans are executed using the reactive control scheme as described in chapter 5. The waypoints of the plan are given as intermediate goals to the reactive controller. The execution should be based on the original action that the plan was created for, and consider the specified tasks as well as possible, even if the environment changes. On the other hand, to actually execute the plan, the action that is executed has to be modified when a plan is to be executed. The most obvious modification is the introduction of a new task that lets the robot follow the waypoints of the plan. This new task has to be integrated into the action so that the waypoints are followed, as long as it is safe to do so, while still monitoring and trying to fulfill the other tasks present in the action. Often, the reason why planning was used at all is that the robot got stuck using the action in a purely reactive fashion. Consequently, it is necessary that following the plan has a higher priority than the previous tasks, otherwise, the robot might just continue being stuck. Considering the example of section 6.7.3, the robot has to be allowed to move closer to the person in order to make room. Safety tasks should however not be overridden by the plan, as they are required for example to react to dynamic obstacles safely. The solution we use is to add the task for following the waypoints as a task with priority that is between the safety tasks and path tasks. In the optimization problem of section 5.6.1, the following changes are thus performed whenever a plan is executed:

- A new `JointPositionTask` is added to the action. It is not considered to be a part of the existing task categories (safety, goal, path, or cost), but instead forms its own category. It has priority $p = 1$.
- All other tasks except safety tasks have their priority value increased by one in order to ensure that they do not override the execution of the plan.
- The input to the newly introduced task is the current waypoint of the plan that is being followed, beginning with the first, and moving forward whenever the robot has come within in defined distance to the current waypoint.

Using this method, plans can be executed while still respecting the specified safety tasks and thus being able to safely react to a dynamic environment. The other pre-existing tasks are still considered with lower priority and thus are only executed as far as they do not disturb the execution of the plan. Since the waypoints of the plan specify positions for all joints, there usually is no room to consider the lower-priority tasks. It is however possible to exclude joints from following the plan. This can be used, for example, to follow a plan only with the base, while still controlling the arm reactively.

7.2 Execution Modes

So far, the question of *when* plans are created has not been touched upon. In some cases, it is clear that an action should be executed completely without planning, or conversely only by following a plan. In the latter case, still it is required that the execution of the plan can react to the dynamic environment. In cases where it is not clear beforehand whether planning is necessary, a typical use case of planning is the case when the robot gets stuck. Therefore, it should also be possible to automatically detect whether planning is required during the execution of an action.

To meet these requirements, we propose three different *execution modes* that can be used when executing an action. Actions can be started in three different execution modes: *reactive*, *planned*, and *autoplanning*. *Reactive* actions only start the reactive control and will never use planning. *Planned* actions will always start planning as soon as they get activated, and not move until a plan is found. Reactive control is still used in *planned* mode, but only to execute the plan. If it is detected that the robot deviates too far from the plan, the plan is aborted and the action is counted as having failed; the robot will not move until a new action is started. Lastly, *autoplanning* combines the two execution modes. This execution mode starts the same as *Reactive* mode, but will try to detect conditions that require planning, and automatically start planning and following the plan then. If the plan execution fails, control goes back into reactive mode. The subsections below describe the different action modes in more detail.

7.2.1 Reactive Action Execution

The *reactive* mode of executing an action is the simplest one. It is only using the reactive control as described in chapter 5. The only elements that are newly introduced here are the termination criteria. It based on the condition *goalReached*:

$$\text{goalReached} := \bigwedge_{f \in C} |f(\mathbf{q})| \leq t_f \tag{7.1}$$

In the equation above, C is the set of all safety, goal, and path constraints. Consequently, an action is considered to have reached its goal once all of its

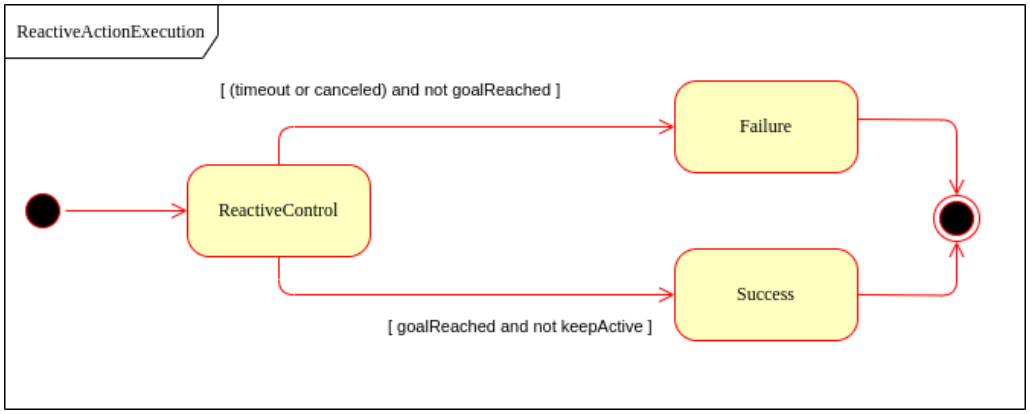


Figure 7.1. State machine of the execution of an action in `reactive` mode.

safety, goal, and path constraints have errors below their defined tolerances. f is the corresponding constraint function and $\mathbf{q}$ the current robot configuration. The specified tolerances of the action are given in the vector t_f. If they are not specified explicitly, they will have the value 0.

Figure 7.1 shows a state machine specifying the behavior of a reactive action. The included condition *timeout* is true if a specified time limit since the start of the action has been exceeded. The condition *canceled* becomes true when an external cancellation request has been received. This is the basic execution scheme when an action is executed as a `ReactiveActionExecution`. To further customize the execution, the following parameters can be set by the caller:

- `timeout:` The time in seconds after which the action is counted as having failed.
- `successDuration:` The time in seconds that `goalReached` must continuously be true in order for the action to succeed.
- `keepActive:` If this is set to true, the execution will not end after `goalReached` becomes true. In this case, the action can only end through external cancellation or a timeout.

7.2.2 Planned Action Execution

Planned action executions are intended for cases where it is clear to the user that a plan is preferred for the execution of the user, and it should not be attempted to execute it without planning beforehand. The behavior of planned actions follows the state machine shown in figure 7.2. Like reactive actions, planned actions can also be canceled at any point, which is seen as a failed action. This has been omitted in the diagram for the sake of compactness.

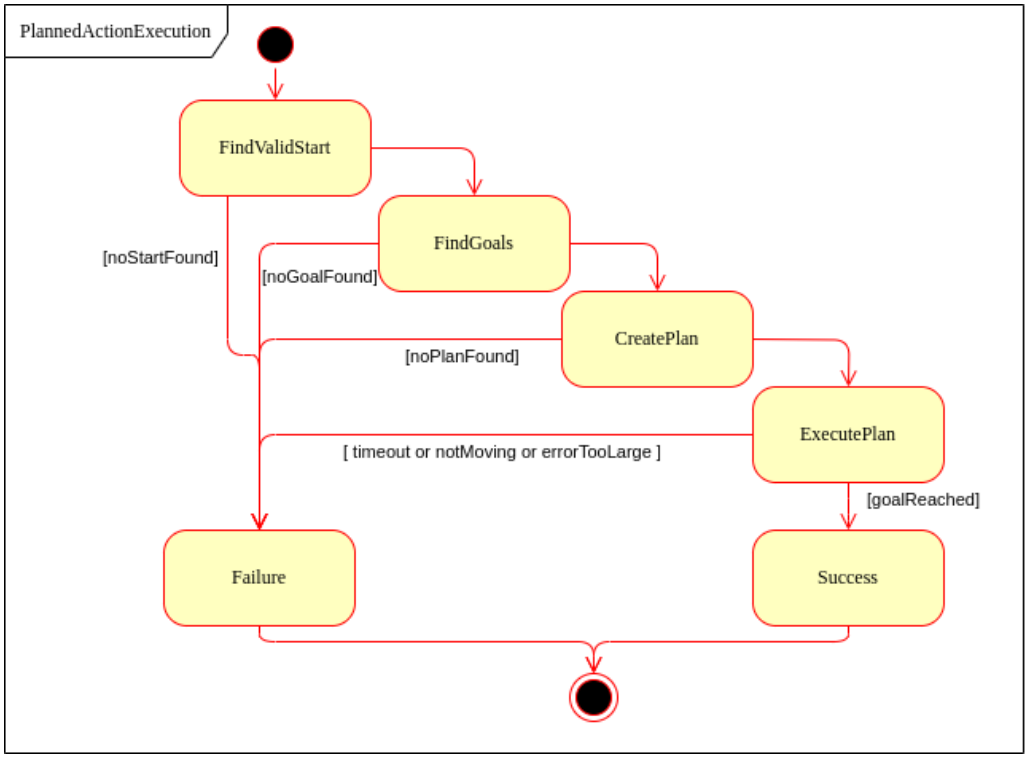


Figure 7.2. State machine of a planned action.

During the execution of plans, the plan will be aborted if at least one of the following conditions becomes true:

- The robot does not move anymore. In this case, it is assumed that the robot got stuck and can not follow the plan anymore. In some applications, it can be acceptable to wait for some time before aborting the plan, for example to wait for an obstacle to clear.
- The robot deviates too far from the waypoints of the plan. This usually only occurs if the safety tasks do not allow for the execution of the plan, and the robot had to move in other directions.

The condition *errorTooLarge* indicates when the second condition is fulfilled. It is defined in equation (7.2). Here, $\mathbf{q}_P$ stands for the next waypoint as given by the plan that is currently being followed. η is the user-definable parameter indicating the maximum allowed deviation from the plan.

$$\text{errorTooLarge} := \bigwedge_{i=1}^{n} \|\mathbf{q} - \mathbf{q}_P\| \geq \eta \tag{7.2}$$

The first condition is expressed in the condition *notMoving*. This condition monitors the joint velocities $\dot{\mathbf{q}}$ over a moving window of w control cycles.

If the commanded joint velocities have not exceeded a defined threshold μ in this window of time, the robot is considered stuck. This condition is defined in equation (7.3). $\dot{\mathbf{q}}_i$ refers to the commanded velocities from i cycles ago. The definition of *goalReached* used here is the same one as given in equation (7.1).

$$\text{notMoving} := \bigwedge_{i=1}^{w} \|\dot{\mathbf{q}}_i\| < \mu \tag{7.3}$$

This execution allows the same parameters as the `ReactiveActionExecution` described above, as well as additional ones listed below:

- `planner:` The name of the planning algorithm to use. If not specified, *TRRT* is used as a default.
- `maximumWaypointError:` The maximum amount that the robot is allowed to deviate from the waypoints of the plan, before the plan execution fails. Corresponds to η in Equation (7.2).
- `movingThreshold:` Determines the minimal joint velocities the robot has to show in order not to be considered stuck. Corresponds to the value μ in Equation (7.3).
- `slidingWindowSize:` The amount of control cycles contained in the sliding window to determine whether the robot is not moving anymore. Corresponds to the value w in Equation (7.3).

A further difference also exists in the meaning of the `keepActive` parameter: The planned execution will switch to a reactive execution after a plan has been successfully found and executed, and then behave as if executed as a *ReactiveActionExecution* with `keepActive` set to true. If any of the states `FindValidStart`, `FindGoals`, `CreatePlan`, or `ExecutePlan` fail, the action will terminate and count as a failure.

7.2.3 Autoplanning Action Execution

This execution mode begins with a simple reactive execution, but will automatically initiate planning when it is deemed necessary. Planning is initiated if it is detected that the robot has not moved significantly for a defined time window, and the goal is not reached at the same time.

$$\text{planningRequired} := \text{notMoving} \wedge \neg\text{goalReached} \tag{7.4}$$

If *planningRequired* (equation (7.4)) is fulfilled while the goal is not satisfied, `PlannedActionExection` is automatically started for the active action. This will result in either *failure* or *success*. Depending on the parameter `keepActive`, the *AutoplanningActionExecution* will then terminate with the same result, or continue execution using reactive control.

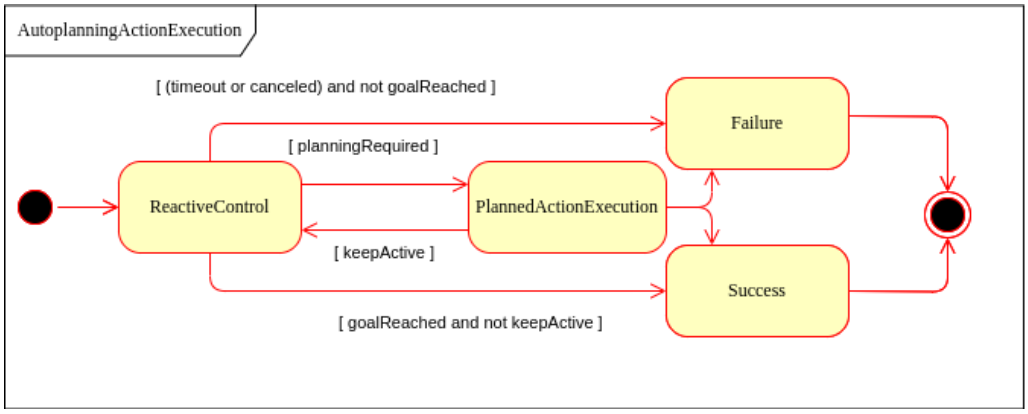


Figure 7.3. State machine of a planned action.

Other criteria for the initiation of planning are possible. We have experimented with definitions taking the current constraint costs into account, but have not found satisfactory definitions for the general case. For example, monitoring whether the constraint cost is increasing over time has not been found suitable, because this can also be the case just when the environment is changing, which should not necessarily trigger planning. Therefore, this simple formulation considering only whether the robot is moving and whether the goal is fulfilled is used.

This execution mode supports all parameters of the other execution modes. In addition, the following parameters are available only for this execution mode:

- `startWithPlanning`: If this is set to true, the execution will begin by creating by attempting to plan. Otherwise, the execution starts purely reactive and will only begin to plan if it is detected that the robot got stuck.

7.2.4 Evaluation

Both the reactive control method and the planning methods have already been evaluated in isolation in Section 5.7 and section 6.7, respectively. The combination of the various actions into behaviors will be evaluated as part of the case studies in section 9.2. Therefore, only an example of an action definition being executed in each of the different execution modes is presented here.

7.3 Combining Actions to Behaviors

Using the action execution modes presented in the previous section, actions can be used with a simple interface and will always result in either *failure* or *success*. Due to this simple interface, it is now simple to combine specified actions to form more complex behaviors. In particular, state machines have been chosen as the formalism to combine actions. The resulting state machines are referred to as *behaviors*. Further examples and the modeling software used are presented in the following chapter.

Summary. The existing components of the system, such as controllers and constraint rules, can be configured in many different ways to achieve new types of robot motions. The required configuration methods are described in this chapter. If the existing components do not provide enough flexibility, the framework can easily be extended through the use of a plugin mechanism. The use of standard interfaces makes the integration of new robot types simple as well. The necessary steps are also described in this chapter.

8 Implementation and Software Architecture

8.1 Integration into the ROS2 environment 139
8.1.1 Defining actions 141
8.1.2 Defining Behaviors 144
8.2 Extending the Framework 145
8.2.1 Adding new Types of Rules, Controllers, Inputs, Solvers 145
8.2.2 Integration of a new robot 145

In this chapter, the software implementation of the previously described concepts and their integration in the ROS2 environment are described. After an explanation of the ROS2 interfaces, it is described how a user can configure new actions and tasks using configuration files. To conclude this chapter, an outline of how the system can be extended is given. It includes implementing new task types as well as integrating new robots into the system.

8.1 Integration into the ROS2 environment

The approaches described in this thesis are implemented and integrated into a ROS2 environment. The use of standard interfaces allows easy interaction with other software components and extension of the system through the introduction of new elements.

The interfaces of the system in ROS2 are graphically illustrated in figure 8.1. The node `constraint_control_node` forms the central part of the constraint control system. The node interacts with the hardware interfaces provided by `ros2_control`, shown on the left side in green, to send hardware commands and receive the current joint states. The `constraint_control_node` calculates the joint velocity command for all controlled joints.

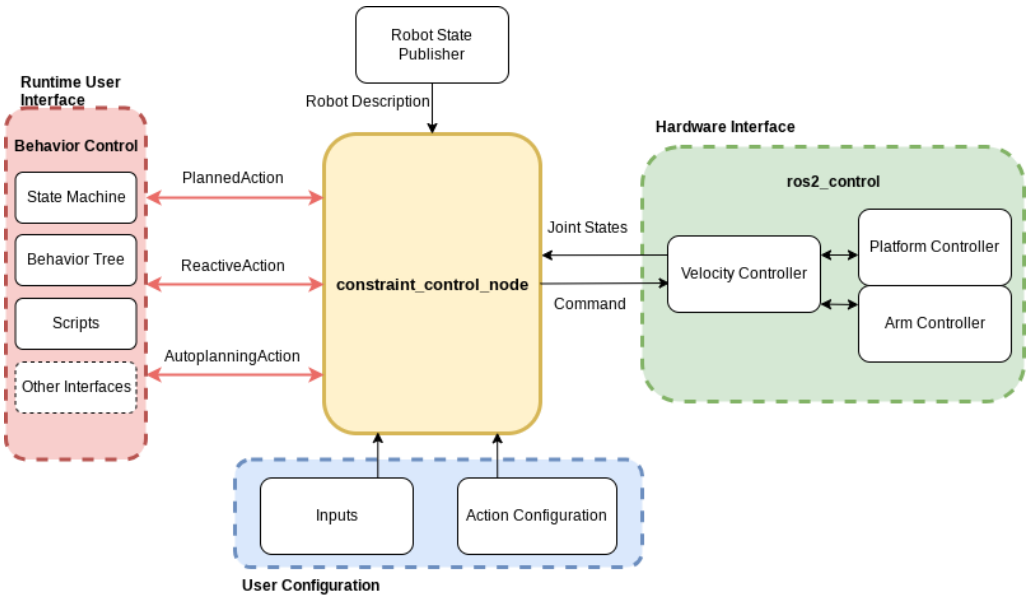


Figure 8.1. Overview of the ROS2 interfaces.

In the case of the mobile manipulator used in the case studies, these are the nine joints corresponding to the three virtual base joints and the six arm joints. These are sent to a velocity controller, which controls the arm and the base combined. Data transmission happens through ROS topics.

The interfaces available to the user are shown on the right side of the diagram, colored in red. The three different execution modes of actions are available as different types of *ROS actions*: The ROS action type `PlannedAction` executes an action as a `PlannedActionExecution`; the ROS action types `ReactiveAction` and `AutplanningAction` analogously use the respective action execution modes `ReactiveActionExecution` and `AutoplanningActionExecution`. The parameters of the action execution types can be specified in the parameters of the goal definition of the ROS action. During the execution, the actions will continuously provide feedback on the current state of the action. This includes the currently active tasks, the corresponding constraint values

They can either be called manually, from a script, or from behavior modeling frameworks such as state machines or behavior trees. As usual in a ROS2 system, the robot description is provided by the `Robot State Publisher`. Offline, the user can define the available actions, tasks, inputs, and controllers in various configuration files. The inputs are continuously updated to the current state of the environment. This information can be accessed through various ROS2 mechanisms, depending on the type of the input, for example, topics and frames.

8.1.1 Defining actions

The user can define actions as well as the underlying tasks through configuration files in YAML format. Similarly, inputs, constraint controllers and global parameters are also configured in YAML files. In the action configuration file, only the tasks included in the action are listed, which completely specifies the action.

The respective components of the actions, such as controllers, inputs and rules, are defined in their respective configuration files. In summary, there are thus five configuration files for an application, which are explained in the following subsections. In addition, behaviors can be built out of the defined actions. However, they are not specified in the configuration files of the system described here. Instead, the flexible interfaces provided allow them to be defined in various different tools that can work with ROS actions.

The global configuration file

This file defines global parameters pertaining to the robot itself and other globally relevant parameters. First, the names and the order of the controlled joints are defined. This is required to interact with other ROS nodes. If the names of the joints are not specified, the joint states and other joint data provided by other ROS components can not be reliably interpreted. Additionally, this file can be used to specify joints that are present in the robot specification, but should not be actively controlled. In our use cases, this used for the gripper: The control system should be aware of the gripper position, but should not attempt to control the fingers of the gripper itself.

Furthermore, this file then also defines the velocity and position limits for each joint. Lastly, the QP solver implementation is selected, and the minimum platform velocity defined, if used.

The action configuration file

Actions are defined in this file by specifying the contained tasks. An example of a definition of an action in YAML is shown in Listing 8.1. This corresponds to the action shown in the object diagram of figure 5.16. The safety tasks are defined globally at the top of the configuration file. Each action definition then only needs to provide a list of path, goal, and cost tasks. The tasks referenced in the action definition of listing 8.1 have to be defined in the task configuration file, explained in the following section.

```
- id: prepareHandover
  tasks:
    path: [gripperUpright, distanceToPerson]
    goal: [gripperPose]
    cost: [lookAtPerson]
```

Listing 8.1. Example specification of an action in a YAML configuration file.

The task configuration file

This configuration file consists of a list of task definitions. Each task definition begins by giving the defined task a name. Next, the *constraint rule* to be used by the task is referenced. This implicitly defines the constraint function that will be used. The `weights` and `tolerances` of the task are also specified here, in the form of arrays of floating-point numbers. Each task is also associated with a `controller` and an `input`. Cartesian tasks are further provided with `controlled_link`. For joint space tasks, this is not applicable.

As an example, the task with `id: gripperUpright` is shown as it appears in the YAML configuration in listing 8.2. The definition shows that the task named `gripperUpright` is defined to use a constraint rule of the type `cartesianPose`, and it controls the link `gripper_link` to move to the pose of the input `base_link` from the input configuration file. In principle, any input providing a Cartesian pose can be used here. Since all weights except those corresponding to the orientation in x and y direction are set to zero, only these two will be considered. These tolerances for these dimensions are also the only relevant ones. They are set to 0.01 rad here, corresponding to roughly 0.5 °. As only x and y orientations are considered, this task will not move the gripper to the base link, but only keep the gripper in the same orientation as the base link. The rotation around the z-axis is still free to move.

The constraint controller is defined to be the `controller: genericFollowController` which is defined in the configuration file for controllers. The details of this are described below.

```yaml
- id: gripperUpright
  rule: cartesianPose
  controlled_link: gripper_link
  input: base_link
  weights: [0, 0, 0, 1, 1, 0]
  tolerance: [0, 0, 0, 0.01, 0.01, 0.0]
  controller: genericFollowController
```

Listing 8.2. Example specification of a task in a YAML configuration file.

The input configuration file

This file defines all the required inputs, providing online data to the tasks. In the example of listing 8.3, an input of `type: cartesianPose`, called `base_link` is defined. Since `source_type: yaml` is used, its values are statically defined within the YAML file itself. Here, they have a pose of zero in the `reference_frame: base_link`, which means that this input just corresponds to the base link of the robot. Other options are available to set input poses provided by frames or topics at run-time, as well as other types of inputs, as shown in figure 5.3.

```yaml
  - id: base_link
    type: cartesianPose
    source_type: yaml
    reference_frame: base_link
    source:
      pose: [0, 0, 0, 0, 0, 0]
      reference_frame: base_link
```

Listing 8.3. Specification of the `base_link` input in the input configuration file in YAML format.

The controller configuration file

This file defines the available controllers and their parameters. Listing 8.4 shows an example definition of the `genericFollowController`. It is defined to be of `type: Follow`, which means it will be an instance of a `Follow`-Controller, as defined in section 5.4.1.

The gain and `max_output:` for the six dimensions of Cartesian poses are defined. The `max_output:` corresponds to the value L in the formal definition of section 5.4, and the gain corresponds to the value α.

```
1
2 - id: genericFollowController
3   type: Follow
4   gain: [3.1, 3.1, 3.1, 3.2, 3.2, 3.2]
5   max_output: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Listing 8.4. Specification of the `genericFollowController` in the controller configuration file in YAML format.

8.1.2 Defining Behaviors

Action definitions and their different execution modes allow for flexible definitions of various robot motions. Each action does, however, only correspond to one specific motion description and no discrete changes in the type of motion can be expressed within an action. Exceptions to this are the execution modes in which the robot can switch between planned and reactive motion, however, these discrete switches are still realizing the same action specification, just by different approaches. In order to change between different actions during the operation of the robot, a method of behavior modeling is required. The execution of individual actions is implemented as ROS2 actions and always ends in success or failure. This simple interface makes it easy to combine actions into behaviors.

Several tools exist that support behavior modeling based on ROS2 actions. For the case studies, we have decided to use FlexBE [124]. This program enables the graphical creation of state machines. Each state can correspond to an execution of a ROS2 action, and other states are possible as well. For example, other states can be used to control systems that are not directly controlled by our constraint-based controller. In the case studies, we have used this to open and close the gripper at specified times in the execution of a behavior. The fact the FlexBE can be extended and customized through Python scripts has proven highly practical in our implementations. New states can be added, and the available parameters specified in the scripts, which can then be instantiated in the graphical user interface. Besides the specification, FlexBE also enables the execution and graphical monitoring of the execution of the specified state machines. Example behavior specifications in FlexBE are shown in the description of the realization of the case studies in chapter 9.

Besides FlexBE, various alternatives are available to the users. One simple option is to simply call the actions from Python scripts, for which ROS2 and the Python ecosystem provide mature tooling. SMACC2 [6] is an asynchronous, behavioral state machine tool for ROS2, based on the semantics of UML state charts. YASMIN [49] is another state machine tool for ROS2 focussing on ease of use. Recently, FlexBE has been extended to support behavior trees as an alternative to state machines as well [150]. BehaviorTree.ROS2 [7] provides another implementation of behavior trees with ROS2 interfaces.

8.2 Extending the Framework

In the preceding parts of this chapter, the possibilities to create new actions through different configurations of existing types of rules, actions, inputs and controllers have been described. While this configuration allows to express a wide range of different motion types, in some cases a user might want to create specifications that can not be created through the combination of existing types. In these cases, new types of tasks, inputs, and controllers can be added to the software. The details of this are described below. Afterwards, the steps that would be necessary to apply the software to a new type of robot are described.

8.2.1 Adding new Types of Rules, Controllers, Inputs, Solvers

The modular architecture enables easy extension of our framework through the use of ROS2 plugins [117]. New types of inputs, solvers, constraint controllers, and constraint rules can be added by implementing ROS2 plugins.

Constraint controllers, constraint rules, and inputs are implemented as ROS2 plugins. This means that they can be implemented and compiled independently of the code of the main project. During the initialization phase, they are dynamically loaded from dynamically linked libraries. In this way, the functionalities of the constraint control system can be extended by a user without having to touch or even possess the original source code. It is enough to have the interface of the base classes that the new classes will extend, for example, the generic controller interface. Since the plugins are loaded as regular classes into the application, no communication overhead is added, and real-time capabilities are maintained (as long as the loaded classes themselves are real-time capable).

New types of solvers, for example to use a different QP implementation, or with hardware-specific adaptations such as the platform minimum velocity in our case, can also simply be added through a plugin mechanism, and the chosen solver specified in a configuration file.

8.2.2 Integration of a new robot

It is easy to use the framework presented here on different robots. As long as the basics of a ROS2 integration are present, the robot can be used with our framework without major changes.

In particular, we understand the basics of ROS integration to be the following:

- Availability of controllers, that can command the robot using velocity commands and provide at least the current joint positions as feedback.
- Definition of the robot in a URDF file, providing the kinematic structure and collision geometry of the robot.

Some limitations on the structure of the robot apply as well: The robot kinematics have to be described as a kinematic tree (so closed kinematic chains are excluded), and the robot has to be able to be controlled using velocity commands.

Using the new robot than only requires a new global configuration file, where the joint characteristics are defined. Likely, the existing action definitions need to be updated as well, unless the robots are very similar to each other. The framework is written in such a way that all algorithms work on kinematic trees, so even dual-arm robots can be used. This has however not been tested in practice.

Summary. The case studies are implemented using the behaviors that contain various actions executed in different modes. The behaviors are implemented in the graphical tool FlexBE. The case studies evaluated using a real robot. Different metrics are recorded for the successful scenarios as well as different scenarios with unexpected environment changes, to assess the robustness of our approach.

9 Realization and Evaluation of the Case Studies

9.1 Realization **147**
9.1.1 Description of the graphical notation of FlexBE 147
9.1.2 Case Study A: Object Handover 148
9.1.3 Case Study B: Following and Lighting 149
9.2 Evaluation **151**
9.2.1 Case Study A: Object Handover 151
9.2.2 Case Study B: Following and Lighting 161

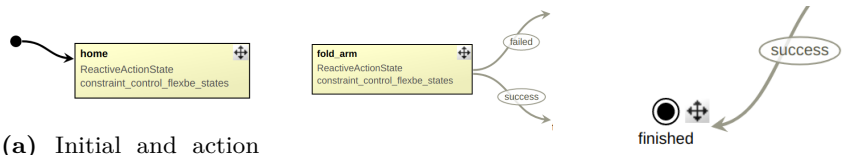
The case studies previously presented in chapter 3 have been realized on a real robot. In this chapter, the details of their realization as well as the metrics recorded during the corresponding experiments are described.

9.1 Realization

In this section, the practical realization of the case studies and the specification of the created robot behaviors are presented. The created behaviors are shown here as they have been created in FlexBE. This section will commence with a brief description of the graphical notation, before describing the realization of the case studies.

9.1.1 Description of the graphical notation of FlexBE

A behavior in FlexBE consists of a state machine. The execution of a behavior begins at the initial state, displayed as a filled black circle. An example is shown in figure 9.1a. An arrow outgoing from it indicates a state transition and points to the next state in the execution. States are shown as yellow or red rectangles. States shown in red stand for the execution of an embedded FlexBE behavior.



(a) Initial and action state

(b) State with transitions

(c) Terminal state

Figure 9.1. Graphical elements in FlexBE

The output of this state corresponds to the terminal state it has reached. States shown in yellow are individual states. In our examples, they mostly correspond to an execution of an action. The name of the action is shown in the first line, the execution type is shown in the second line. A `PlannedActionState` will execute the action as a `PlannedActionExecution`. `ReactionActionState` and `AutoplanningActionState` similarly use the corresponding execution modes. A state in FlexBE is only left after the action of the corresponding action has finished. Parallel execution of actions is not supported at the moment.

Not every state in FlexBE has to be an action execution. Other states, such as a state for opening and closing the gripper, can also be used. Other examples include states that request input from the user, or states that make decisions based on other external events. In this way, flexibility can be added to the behaviors that can not be achieved only by considering the results of action executions.

The outgoing transitions from each state have an attached label, in our case only `success` or `failure`. Figure 9.1b shows an example. Which state is executed next depends on the result of the previous one.

The execution ends as soon as a terminal state is reached. In the examples of this work, this is always one of the two states `finished` or `failed`. Other terminal states can be defined and used if other results of an execution are required. The terminal states are illustrated by a filled black circle surrounded by another circle, and have the name of the result written below them. In figure 9.1c, a terminal state with the result `finished` is shown.

9.1.2 Case Study A: Object Handover

The robot behavior used for this case study is implemented as a state machine and graphically modeled in the tool *FlexBE*. The resulting state machine can be seen in figure 9.2. The behavior execution will begin at the top left. The first activated state then is the state `get_flashlight`. The execution of this state will execute another behavior, correspondingly called `get_flashlight`. It is shown in the figure 9.3 and described below. The red color of the state indicates that the state stands for an embedded behavior. A photograph taken during the execution of the behavior can be seen in figure 9.4. Here, the robot is approaching the flashlight with its gripper. The flashlight is contained in its mount on the robot base.

The `get_flashlight` behavior.

The purpose of this behavior is to pick up the flashlight from the mount on the robot's base with the robot's gripper. This behavior contains no dependencies on the environment, no interaction with a person, and does not require any use of the robot's mobility, using only the arm. All referenced poses are completely static with respect to the robot's base. As such, this behavior also serves as an example of how motions typical of classical industrial robot programming can be realized in the framework presented here.

The behavior starts by moving the arm to its home position (`home`), once this has succeeded, the robot will open its gripper (`open_gripper`). Next, the robot arm is moved to a position directly above the flashlight (`pre_flashlight`), using a point-to-point motion. Afterward, the gripper moves linearly toward the flashlight (`grasp_flashlight`), where the gripper will be closed (`close_gripper`). This is followed by a linear upward motion (`grasp_flashlight_post`). Afterward, the arm is moved to a folded position (`fold_arm`). All of this assumes that each action succeeds. In case any action fails, the entire behavior will terminate with the final state `failed`. This behavior assumes that no external disturbances are occurring during the execution. Any disturbances, such as a person moving their arm in between the robot arm and the robot base, are not part of the expected behavior and thus will simply result in a failure of the behavior. If such failures would have to be considered and the operation continued, an appropriate reaction needs to be defined and can then be added to the state machine of the behavior.

All the actions in this behavior are executed as reactive actions. This might be unexpected, considering they are all executed in a static environment which typically calls for planning rather than reactive control. However, our experiments have shown that the reactive control has fewer possible causes for failure or suboptimal results. The probabilistic nature of the planning algorithms means that an optimal plan is unlikely, and rarely planning can even fail. The chance that the plan will include some unnecessary detours is not to be ignored, while the reactive control is sure to take the direct path. As long as there is no risk of getting stuck in local minima, the reactive execution is thus more robust and predictable.

9.1.3 Case Study B: Following and Lighting

The realization of the behavior used for the second case study is a lot more simple. Considering that most of the behavior is described as a single reactive control application, the need for complex state machines is low. Figure 9.5 shows the behavior definition from FlexBE. Only two actions are used. The first is the action `follow_and_light`, which contains the main success scenario of the case study. The definition of this action is shown separated into the different task types. Figure 9.6 shows the path tasks, figure 9.7 shows the goal tasks, and lastly, figure 9.8 shows the cost tasks.

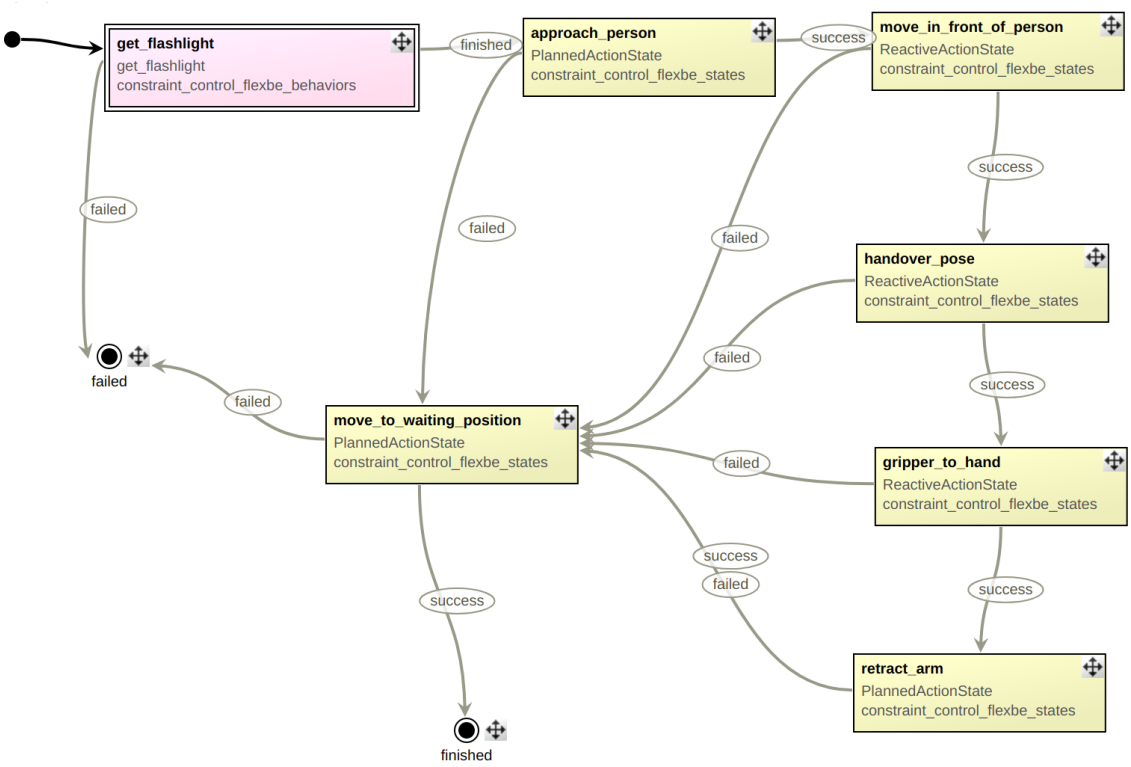


Figure 9.2. State machine of the object handover behavior, as it appears in FlexBE.

Details regarding tolerances, controller parameters, and others have been omitted to keep the graphics concise. The safety tasks are again the global safety tasks defined in chapter 5. There a two path tasks: `stopBaseWhenClose` and `stopEEWhenClose`. Both are controlled using `StoppingControllers`. The effect is that the base and the end-effector will slow down and stop when the hand or the torso of the person come close to the controlled links. Three goal tasks are in use. `toPerson` is intended to keep the robot close to the person, while `keepDistance` ensures that the distance will not become too small either. `toPerson` uses a `FollowController`, while a `HybridController` is used for `keepDistance`. In this way, the robot can move even while `keepDistance` is fulfilled, while not being allowed to move away from the person due to the task `toPerson`. Lastly, the task `shineLight` uses an `AimingRule` to let the robot point the flashlight in its gripper toward the person.

Three cost tasks are specified. These are used to let the robot face its base toward the person, keep the arm in a nicely folded position, and to keep the end-effector above a certain minimum height. Of course, as these are cost tasks, they are only realized as far as they do not disturb the other tasks. All the cost tasks are controlled by `FollowControllers`. Since they are usually overridden by the other tasks anyway, using more lenient controllers has rarely proven useful in our experiments.

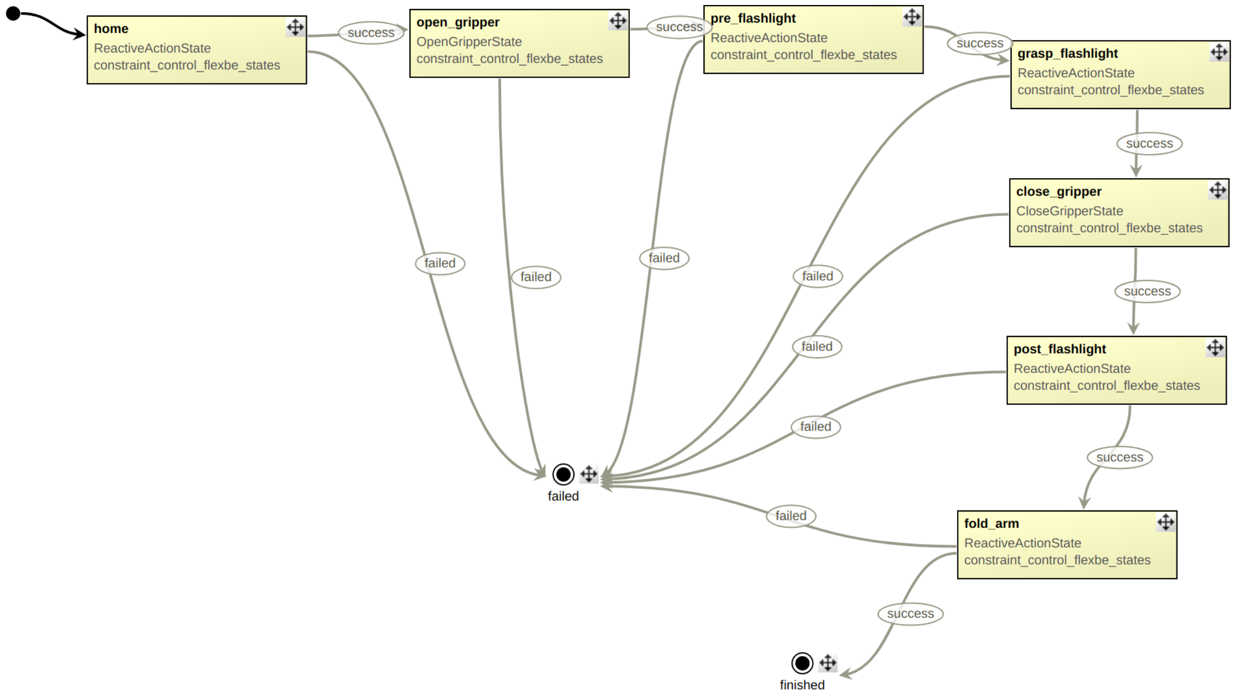


Figure 9.3. State machine of the behavior to pick up the flashlight, as it appears in FlexBE.

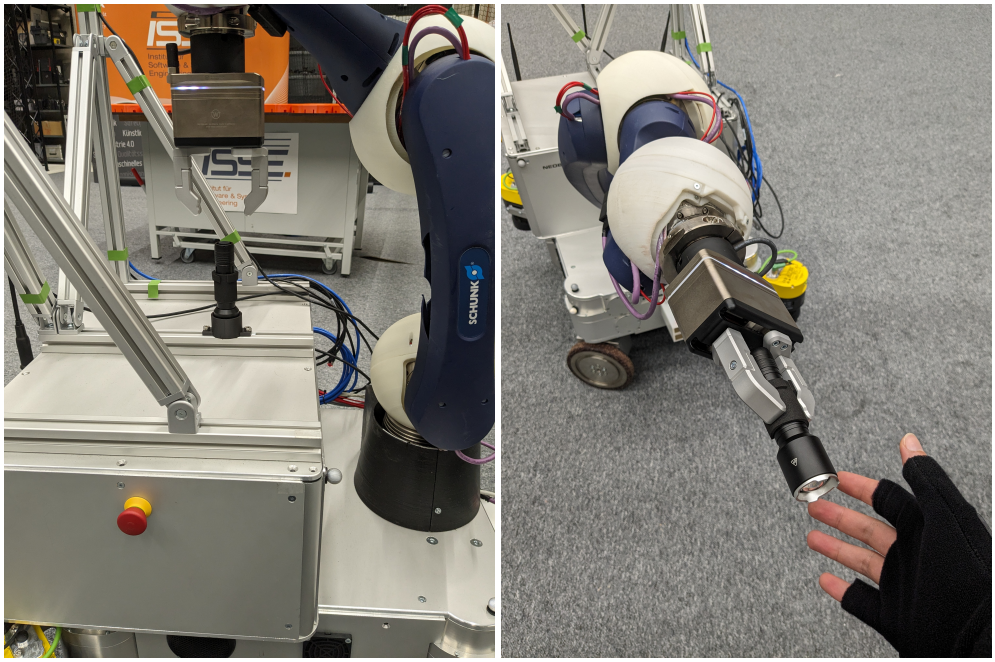
Only if this action `follow_and_light` fails, typically because it can no longer reach the person, is the other action of the behavior, `move_to_waiting_position`, called. This is executed as a `PlannedActionExecution` and has the sole purpose of returning the robot to a defined waiting position. Whether this action succeeds or not, the overall outcome of the behavior is `failed`, as the main action failed.

9.2 Evaluation

The behaviors described above have been executed on the real robot, with different behaviors and different motions performed by the human interaction partner. In this section, different metrics that have been recorded during the experiments are presented and interpreted.

9.2.1 Case Study A: Object Handover

The object handover scenario is evaluated in four different executions. In the first execution, the successful execution case is evaluated, with a person that is cooperative in the handover, and no blocking obstacles present. Next, in order to evaluate the robustness of our method to different challenging scenarios, three failure cases are tested as well. First, the path to the person is blocked by obstacles. Next, the person shows no interest in the handover, does not cooperate with the robot, and does not move their hand toward the handover position.



(a) Picking up the flashlight **(b)** Handover

Figure 9.4. Photographs of the experiments.

Lastly, a scenario is tried where the person attempts to cooperate at first, but then continues with unexpected motions, moving their hand behind the robot's gripper.

Picking up the flashlight

As a prerequisite to the handover, first, the ability to pick up the flashlight from the mount on the robot's base is analyzed. The included actions are shown in figure 9.3. While most of the included actions are simple point-to-point motions of the arm, the approaching motion toward the flashlight is a bit more interesting.

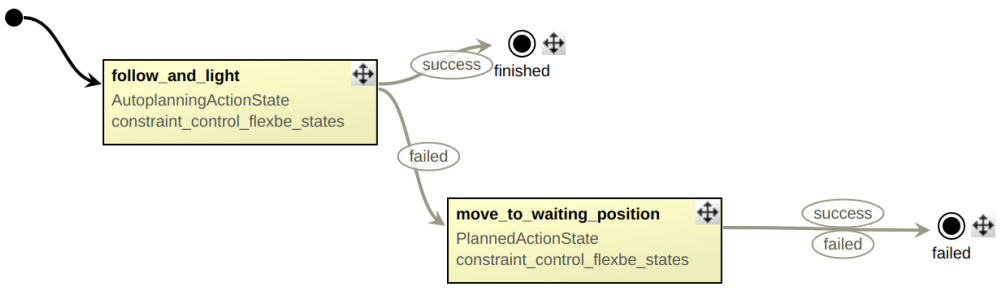


Figure 9.5. State machine of the case study B, as it appears in FlexBE.

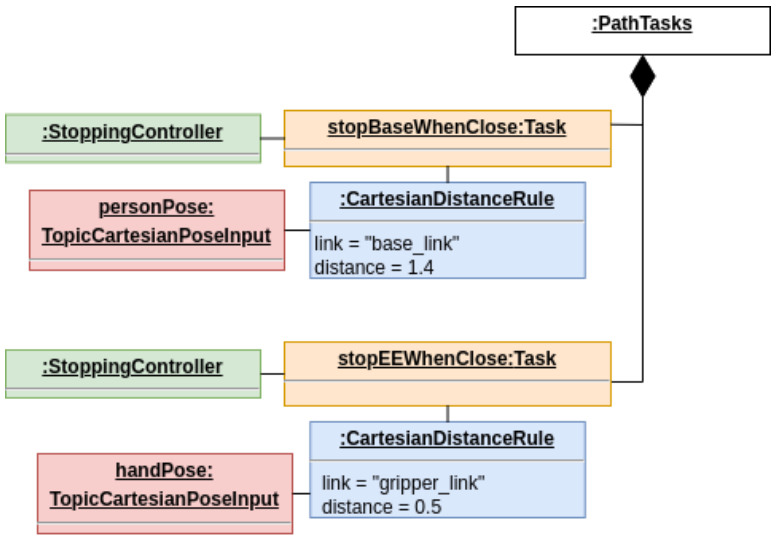


Figure 9.6. Overview of the path tasks in the action `follow_and_light`.

It corresponds to the action `grasp_flashlight`. The name refers to the arm motion to grasp the flashlight, the closing of the gripper is contained in the action `close_gripper`. All actions of the gripper are using the interfaces provided by the gripper's driver directly, and are not controlled by the constraint-based control system.

When moving the gripper fingers around the flashlight, as well as when pulling the flashlight out of its mount, the gripper has to move linearly. Otherwise, the flashlight would collide with the sides of the mount when putting it into the mount, or the gripper fingers would collide with the top of the flashlight when moving them to the grasping position. The action begins from a position above the flashlight. The action is thus described by two main tasks: one to move the gripper downwards toward a suitable (statically defined) grasping position, and a second one that prevents any sideways motion of the gripper. The corresponding object diagram is shown in figure 9.9.

Both tasks, `avoidXYMotion` and `moveDownwards`, use the same input `flashlightPose`, which is statically stored in the configuration, as the pose is static on the robot itself. `avoidXYMotion` uses a `follow`-Controller with a steep response ($\alpha = 10$) to ensure a quick reaction to any deviation, while `moveDownwards` uses $\alpha = 2$ for a gentler approach of the flashlight.

Besides these two tasks, the global safety tasks are active, except for the self-collision avoidance between gripper and base, which has been overridden in order to allow the gripper to move close enough to the flashlight. This evaluation serves as an experiment to analyze the capability of the constraint-based system to realize standard motions in robot programming. For the final grasping motion, the robot has to move the gripper about 0.075 m downwards.

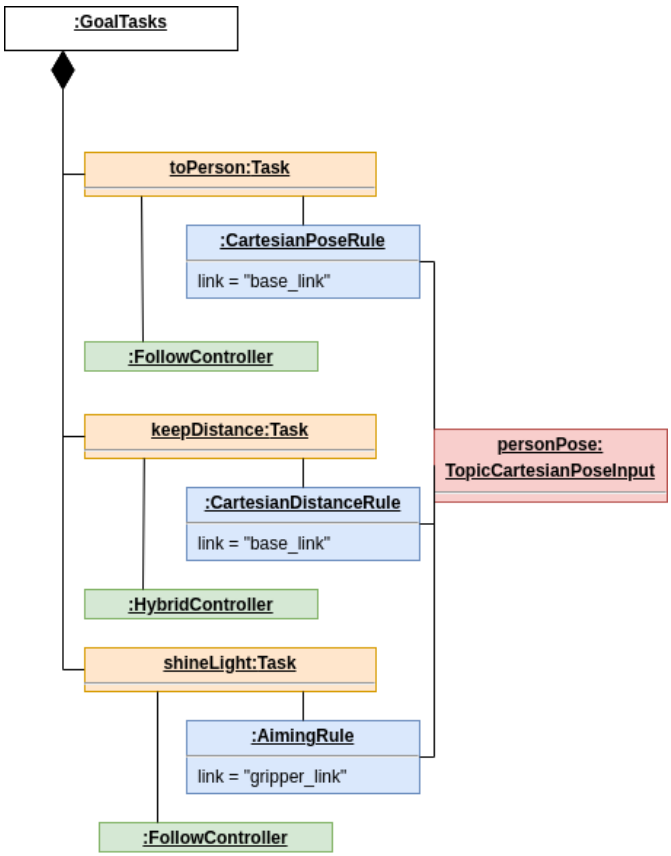


Figure 9.7. Overview of the goal tasks in the action `follow_and_light`.

Figure 9.10 shows a plot of the position errors recorded during the execution. The plot shows the error along the z-axis as the gripper moves downwards, as well as the deviation on the x-y-plane that occurs during the motion. This deviation should ideally be zero. It can be seen that some deviation occurs, with a maximum magnitude of 2.5 mm. Such measurements of path accuracy are difficult to compare to other results, as they are heavily dependent on the robot model, pose and execution speed. To provide some assessment of the recorded values, the value can be compared to those of other robots. For a KUKA KR 5 robot, the path accuracy at 10% speed has been measured to be 1.844 mm on a rectangular path, and 1.493 mm on a figure-eight-shaped path, and decreases up to 4.464 mm at full speed [98]. Thus it can be concluded that the accuracy of our experiments is within an acceptable range. It does not quite reach to accuracy of industrial control systems. It must also be considered that neither the Schunk LWA manipulator nor the constraint-based control method presented here have the same focus on path accuracy as the KUKA KR 5, with welding being one of its main applications. Also highlighted in the plot is the time at which the action terminates and succeeds.

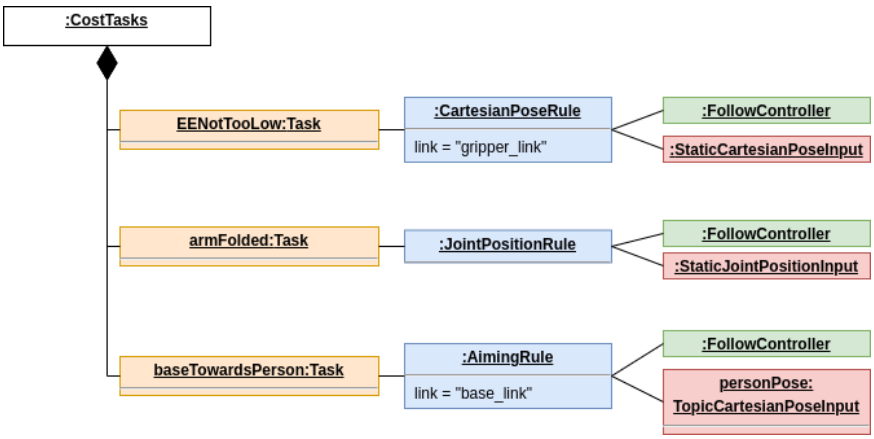


Figure 9.8. Overview of the cost tasks in the action `follow_and_light`.

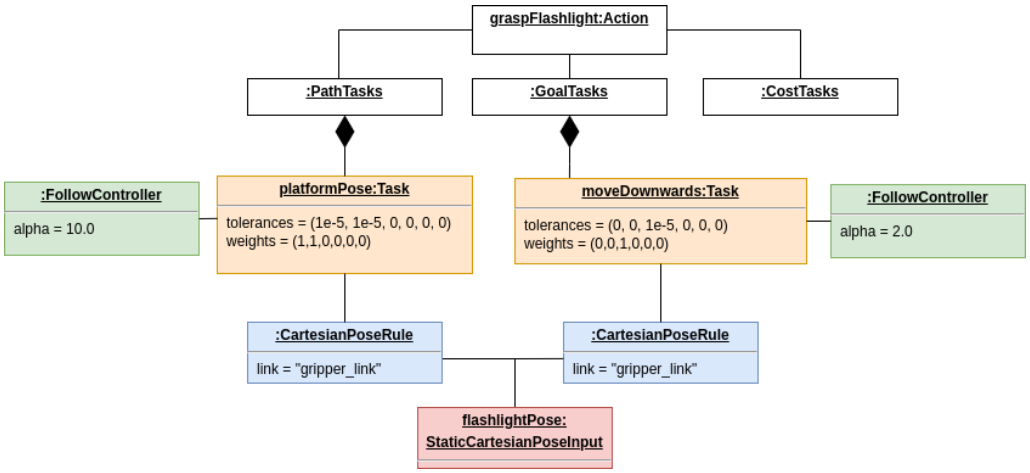


Figure 9.9. Object diagram of the configuration used during the flashlight pickup. Safety tasks are excluded.

As the tolerances are all set to very small values in the example, the action only terminates once the error values reach very small values. The other actions of this behavior are not further evaluated here. The linear motion to pull out the flashlight is the same in the opposite direction. The other motions are either simple point-to-point motions or gripper motions, which use the gripper driver interface directly.

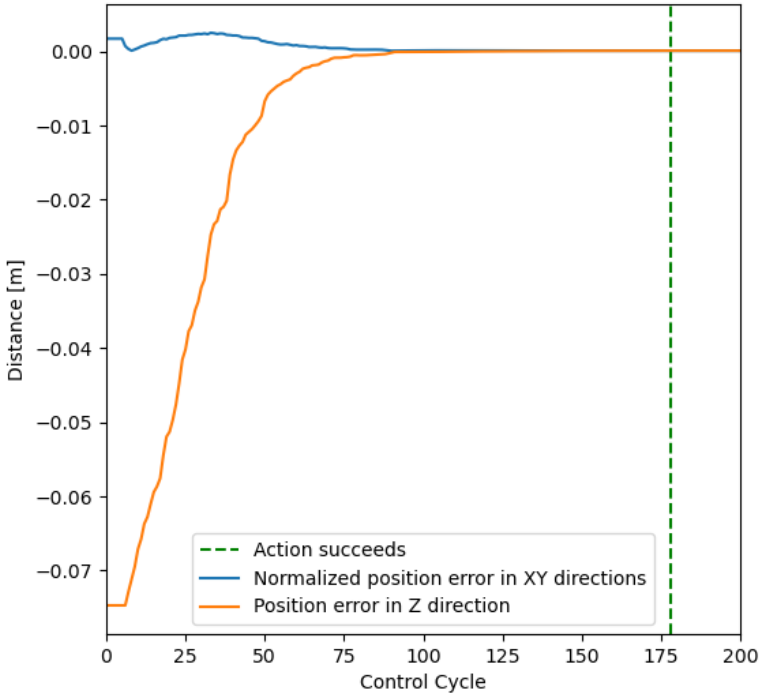


Figure 9.10. Accuracy plot of the linear motion downwards toward the flashlight.

Successful execution

At first, the successful scenario is evaluated, where all steps of the specified behavior are executed successfully. Thus, the following actions are executed in order:

1. `approach_person`
2. `move_in_front_of_person`
3. `handover_pose`
4. `gripper_to_hand`
5. `retract_arm`
6. `move_to_waiting_position`

The results of each of these actions are briefly described below. Excluded are `handover_pose`, `retract_arm`, `move_to_waiting_position`, which are simple motions without explicit dependencies on the outside environment. As the ability to perform basic motions has already been analyzed in section 5.7, they are not evaluated further here.

`approach_person`: The robot has executed the planned action to move to close enough to the person in order to begin the interaction. In this simple, first scenario, there are no further obstacles in between robot and person. This action is thus a simple planning problem. Using the default TRRT planner, a suitable plan has been found in 0.74 s. The plan is then executed successfully in 4.01 s.

`move_in_front_of_person`: This is a reactive action, ensuring that the robot is positioned in front of the person. In the successful scenario where the person is actively engaged in the handover, this is already the case after the previous action and the action succeeds immediately.

`handover_pose`: This is a simple motion, moving the arm from its transport position to a more forward position, more suitable to begin the handover from. In this case, without unexpected obstacles, there are no metrics of interest to record.

`gripper_to_hand`: This action controls the position of the gripper, leading up to the physical handover. The action is shown in figure 9.12. The robot is supposed to adjust the height of the gripper and its orientation to enable a handover. As the person's hand comes closer to the gripper, the gripper should stop moving actively, to ensure safety. The former requirement is expressed through the tasks `gripperHeight` and `faceTowardsHand`. The latter is realized through the task `stopEEWhenClose`. The used `StoppingController` will stop the gripper once the person's hand comes too close to the gripper. No cost tasks are used in this action.

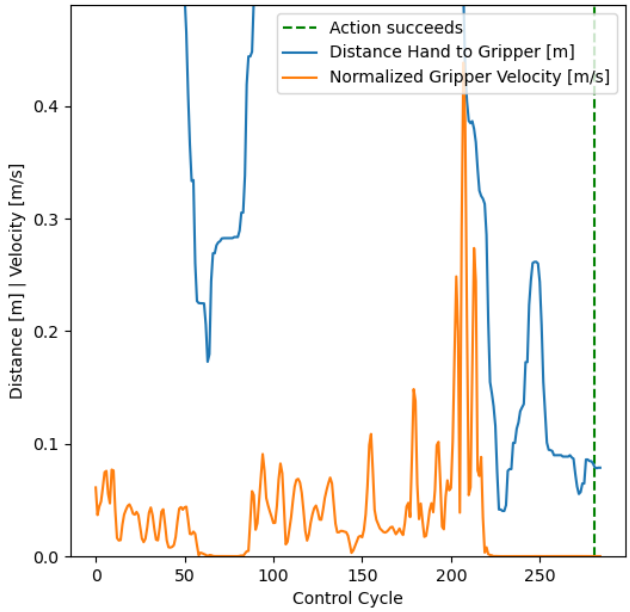


Figure 9.11. Recorded data of gripper velocity and distance to the person's hand

The magnitude of the Cartesian gripper velocity and the distance between gripper and the person's right hand have been recorded. The data is shown in figure 9.11. In the recording, the hand approaches the gripper for a first time, before retreating a bit. Later, it approaches again and fully comes in contact with the gripper. It can be seen that the gripper velocity goes toward zero as the hand approaches the gripper. As soon as the distance between gripper and hand has been below the task's tolerance for long enough, the action succeeds. It can be seen that the slowing of the gripper as the hand comes closer works as intended. Furthermore, the method of determining the success of the action through the state of the individual tasks is successfully used as well.

Scenarios with Obstacles

Two scenarios with additional obstacles have been evaluated. In the first, the path toward the human is blocked completely, and approaching the human is thus not possible. The scenario is shown in figure 9.13. The execution of the behavior is simple in this case: As `approach_person` can not find a valid path, planning is aborted after the configured maximum duration of 10 s, and the execution continues with the action `move_to_waiting_position`. As the robot has not yet moved from its waiting position, this succeeds immediately and the behavior finishes.

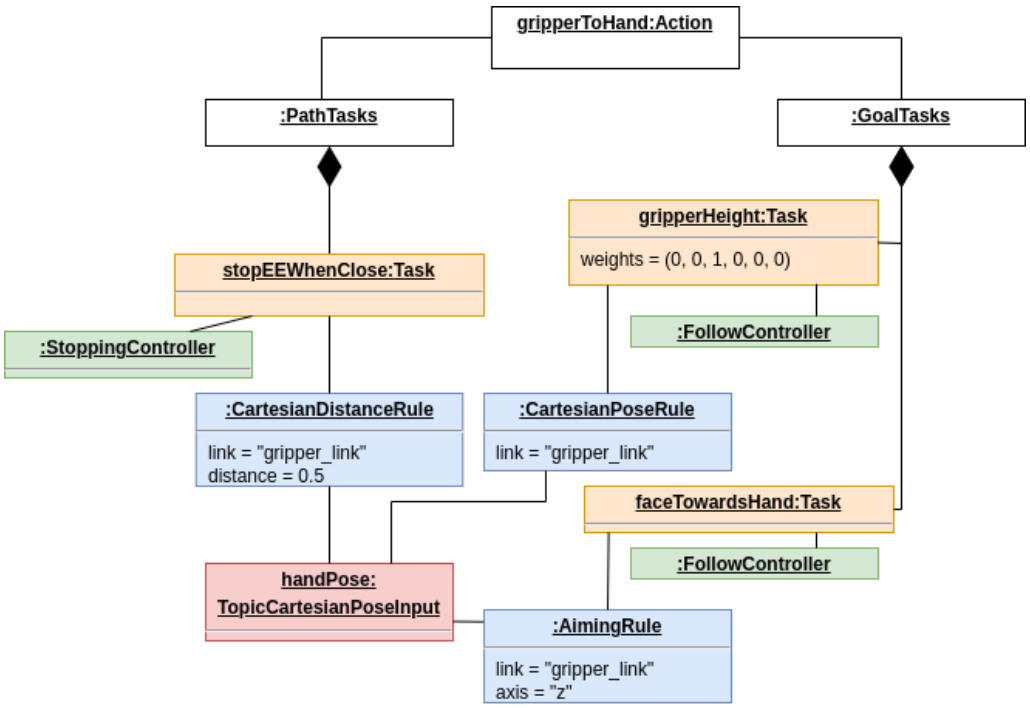


Figure 9.12. Object diagram of the action `gripper_to_hand`

In the second scenario with obstacles, the obstacles leave a narrow, door-like passage between them. This is shown in figure 9.14. In the execution, this is encountered during the execution of the action `approach_person`. As this execution is executed using planning, the effects of the obstacles mean only that the planning problem has become slightly more challenging. As before, the default planner TRRT is used. The planning took 1.34 s, and the plan was successfully executed in 12.10 s. These results show that the approach is robust to obstacles appearing in the described way. Planning times are still relatively low, only the execution takes somewhat longer. Besides the simple fact that the robot has to drive a longer path, the fact that the robot slows down more as it is closer to the obstacles also contributes to the increased execution time.

After this phase, the execution of the handover proceeds as in the scenario without any obstacles, described in the subsection above.

Person ignoring the robot

This evaluation case describes an execution where the person does not have the intention to interact with the robot in any way, and does not move toward it. At the moment, no sophisticated method of estimating the person's intention is used. As long as the person is positioning themselves in a way that allows the robot to fulfill its tasks, such as bringing the gripper close to the person's hand, the robot will perform the actions of the designed behavior in sequence and consider

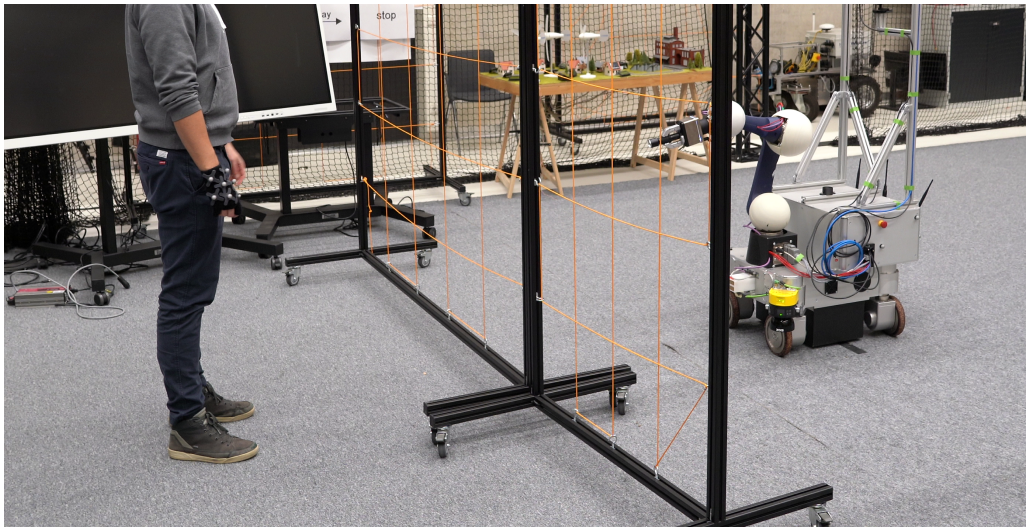


Figure 9.13. Handover scenario with obstacles completely blocking the robot from the person.

it a success. The design of the presented system allows however for a simple integration of such systems through the use of interfaces to start and cancel robot actions at any time. At the moment, the only metric that can be evaluated here is whether the robot behaves safely even if the person is not participating in the interaction. As long as the robot is able to fulfill all the specified actions, it will consider the actions a success. The action `gripper_to_hand` can not be fulfilled without the person actively moving their hand towards the gripper. If the person does not do this, the action will time out and the execution of the behavior will fail. In this way, a simple method of handling a person that does not interact with the robot has been realized. For practical use, more complex methods based on visual cues or other information should be used to recognize the human's intention. The described system allows for the easy integration of such external decisions.

Unexpected motions by the person

In this scenario, the person behaves as expected for the handover at first, but then continues with unexpected motion by moving their hand behind the gripper instead of grasping the object. This is shown in figure 9.15. The requirements on the robot's motion here are that it neither creates any unsafe motions that could hurt the person, nor releases the object from its gripper because it is not held by the person.

A plot of the Cartesian gripper velocity compared to the distance between gripper and hand is shown in figure 9.16. It can be seen that the distance approaches zero, and the gripper safely comes to a stop. As the person does, however, not keep their hand close, and instead increases the distance again, this time behind the gripper, the action does not succeed. The hand is not close enough to the gripper for a long enough duration.

Figure 9.14. Handover scenario with additional obstacles.

It must be said that a potentially unsafe scenario could occur from the given specification, if the person moves their hand even further behind the gripper, so that it is free to move again. As the specification only takes the location of the hand into account, the fact that the person's chest would now be dangerously close to the robot is currently not considered in the specification as it was used in the experiments. While this might be an untypical motion, this also shows the need for extensive testing and verification of all possible human behaviors. The behavior is rather easy to fix by introducing another task that stops the whole robot as soon as the torso of the human approaches. In a real-world application, more sophisticated methods of intention estimation would be desirable, taking more data into account than just the position of the hand. However, such methods are outside the scope of this work.

It can, however, be seen that the robot safely comes to a stop as the human hand approaches. The action failed after the configured 5 s timeout duration, after which the robot returned to its waiting position.

9.2.2 Case Study B: Following and Lighting

Similarly to the evaluation of case study A, we evaluate this case study in a simple success scenario as well as different failure cases with unexpected environments or motions by the interaction partner. First, the basic ability to follow a human around while providing light is evaluated. Afterward, small obstacles are introduced that should still be able to be handled in a purely reactive fashion. Then, larger obstacles are introduced that cause the robot to get stuck when attempting to follow the person and require planning.

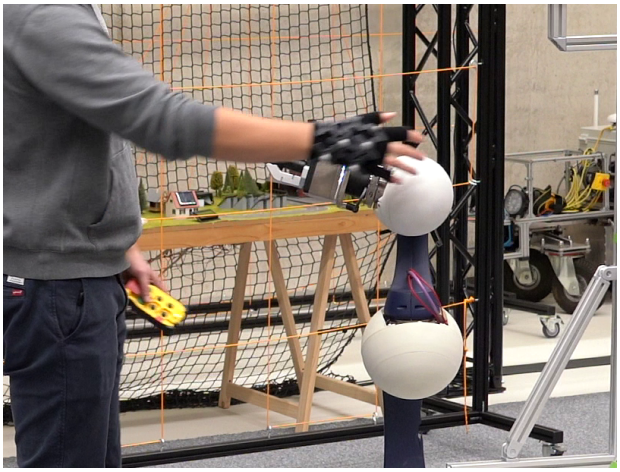


Figure 9.15. Example of an uncooperative person during the handover: Reaching past the gripper.

Successful Execution

Figure 9.18 shows plots of both the distance between the person and the platform, as well as the angular distance between the direction the flashlight is shining in, and the direction of the person. It can be seen that the angular distance is strongly increasing whenever the person comes close to the platform. This can be explained by the fact that the robot is forced to slow down and eventually stop as the human comes closer, as enforced by the task `stopEEWhenClose`. Hence, the robot can no longer move the flashlight in the correct direction, and the angular distance increases. Therefore, this is an expected effect, as specified in the action definition. The robot is actively trying to keep a distance of at most 2.0 m. By itself, the robot will not move closer toward the person and stop at this distance. If the person actively approaches the robot, it will however not retreat.

The green horizontal line in the distance plot illustrates the distance that the robot is attempting to keep between itself and the person, according to the behavior specification. This effect is achieved by the task `toPerson`, moving it toward the person, and the task `keepDistance`, which ensures it does not move too close. Toward the end of the plotted evaluation run, the person is moving away from the robot in a direction that the robot can not follow, due to its given position limits. Thus the distance increases strongly.

The accuracy of the aiming of the flashlight can potentially be increased substantially by fine-tuning the controller parameters and allowing higher joint velocities. In experiments, this however had the unpleasant side effects of a jittery appearance of the light, and a subjectively hectic-looking robot. Therefore, the task parameters have been set to create a slower, more stable reaction at the cost of some accuracy.

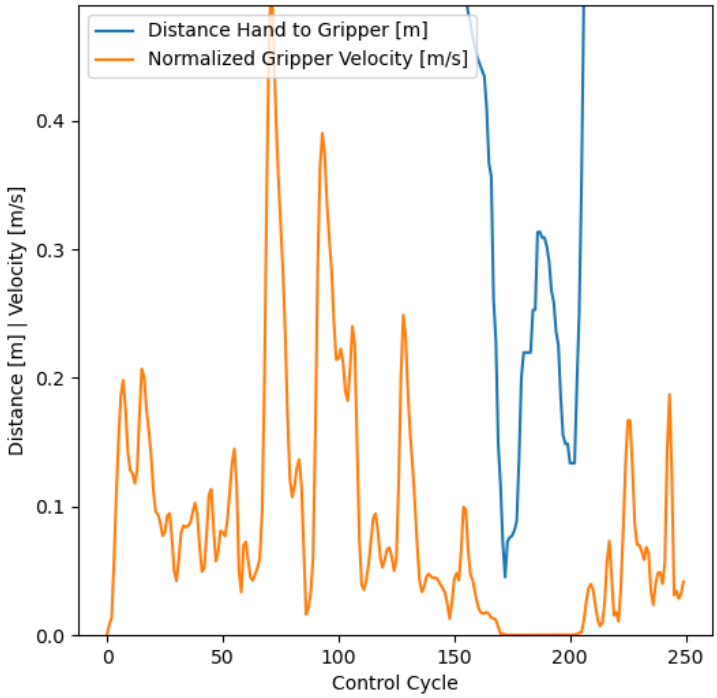


Figure 9.16. Plot of recorded gripper velocity and its distance to the human hand.

Getting stuck on obstacles

In this scenario, the robot is at first following and lighting the human as described above. After a while, obstacles in the environment prevent the robot from following the human any further. The use of autoplanning means that this should be recognized and planning initiated. The execution of the plan should then resolve the problem, and move the robot to a position from which it can fulfill its tasks reactively again.

Figure 9.19 shows plots of the total velocity command magnitude, as well as the constraint costs during the execution. In the first phase, it can be seen that the robot is moving relatively fast, and the cost is mostly kept at a level below three. In this phase, the robot is reactively following the robot. Then, as obstacles appear, the robot is no longer able the follow the human, and the commanded velocities decrease substantially. Consequently, the costs rise. After a while, the high cost in combination with low velocities triggers the autoplanning action to start path planning. During the planning phase, which takes about 2.5 s, the robot still does not move much.

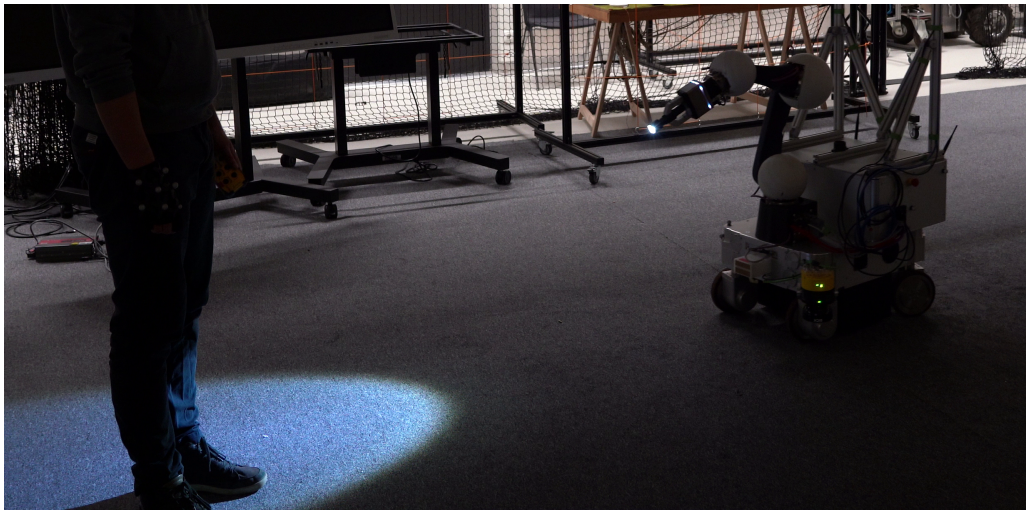
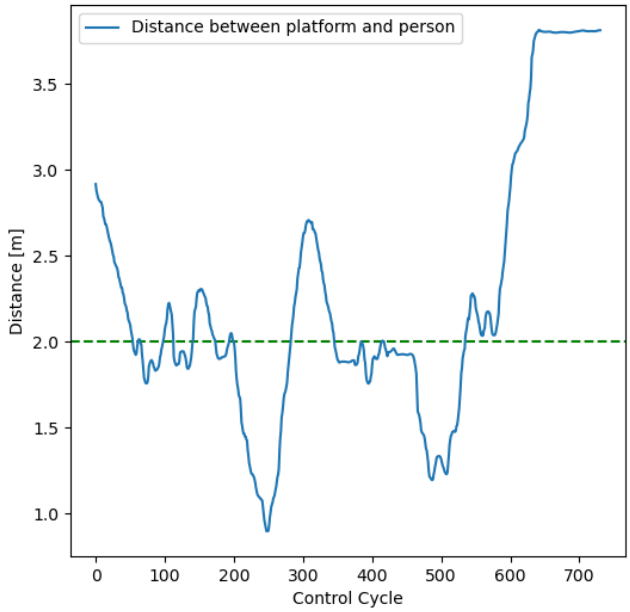


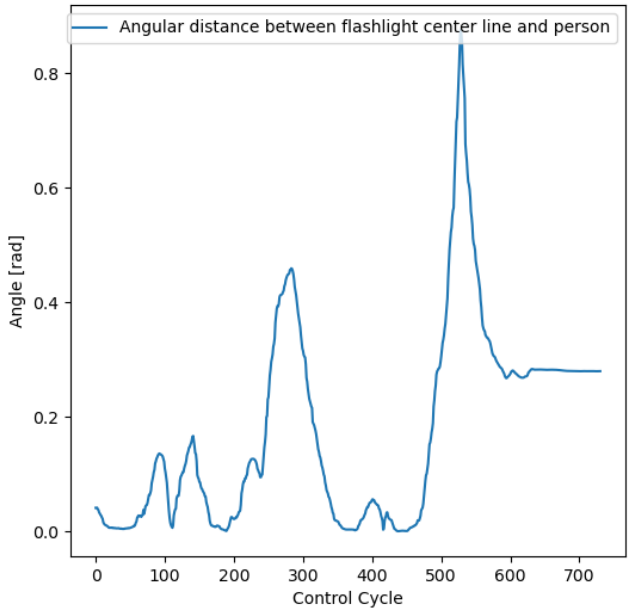
Figure 9.17. Robot shining its flashlight at a person's feet.

Then, however, after a plan has been found, it is executed and the robot is commanded high velocities again. The cost during the execution rises further at first: moving around the obstacles requires the robot to move even further away from the person. After a while, the costs decrease again however as the robot is now able to reach the person again. During the execution of the plan, some oscillation in the velocity commands can be seen. This is caused by the waypoint-based method of executing plans, in combination with the use of a simple proportional position controller. Whenever the robot comes close to a waypoint, the velocities calculated by the controllers decrease and increase sharply again as the next waypoint is used as the next target. This could be alleviated through the use of feed-forward control, which takes not only the position to reach, but also the velocity the robot should have at this point into account. The robot would thus not temporarily attempt to come to a standstill at every waypoint.

All in all, this scenario illustrates the validity of our autoplanning method for detecting when planning is required, and its ability to find plans that lead the robot out of local minima.



(a) Distance between platform and person



(b) Angular distance between flashlight center line and person

Figure 9.18. Plots of the recorded data during following and lighting.

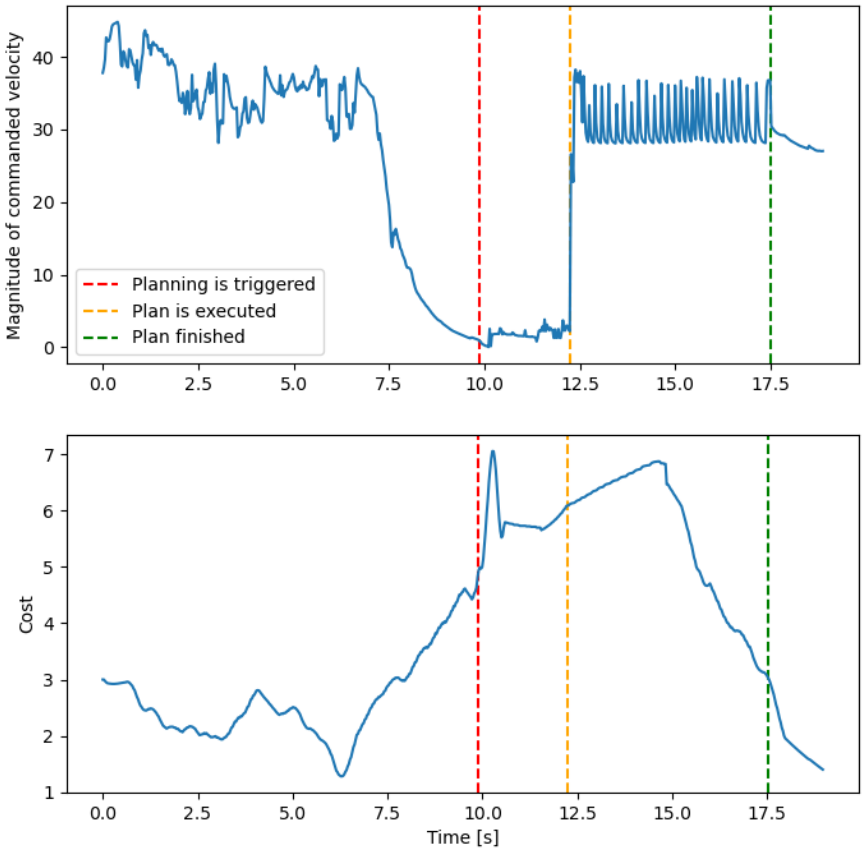


Figure 9.19. Plots of commanded velocity and constraint costs.

10 Conclusion and Outlook

At the moment, assistive robots are still not widely applied despite their large potential. Safety in interaction and dynamic environments is one of the most challenging causes of the current limitations. The generation of robot motions that are safe and robust in unforeseeable environments remains a core challenge. This work offers several research contributions to these questions. Most important is a method for the specification of requirements for robot motions, which can automatically be executed safely and robustly. Both planning and reactive control are supported and can be combined in different ways.

10.1 Summary of Research Contributions and Evaluation Results

Motivated by the case studies of object handovers and lighting using a mobile manipulator, we have developed a method of specifying requirements on robot motions based on geometric constraints. Different methods for specifying their relative importance are supported. These include weights, priorities, and tolerances. Constraints are formulated as simple functions on the robot configuration. This makes them easy to describe formally and enables their simple use for path planning. On the other hand, this means that they do not explicitly depend on the state of the dynamic environment. This dependency is instead handled by constraint rules. Constraint rules adapt the parameters of the constraints to reflect the current state of the environment.

Constraint controllers compute velocity bounds of their respective controlled links or joints based on the current value of the constraint functions. These velocity bounds are then used in conjunction with the specification of weights and priorities to formulate a quadratic programming problem. This problem is solved by a QP solver, and results in an optimal control signal which is then executed by the robot. It has been shown that the mobile manipulator can be controlled at a frequency of 100 Hz in this way.

The use of independent and encapsulated components for the specification of robot motions enables great flexibility in the combination of individual elements to create new robot motions. The same action specification can be executed as a planned, reactive, or autoplanning motion. The same constraint specification can be used for different reactions by applying a different controller to it.

Similarly, the inputs used by the constraints can be easily exchanged, for example using data from ROS topics, statically configured data, or environment frames. Further customization can be achieved through various parameters, which enable a wide range of robot motions through only small configuration changes. At the same time, the architecture allows for the specification of global safety constraints, which are automatically monitored and make it possible to specify actions without having to consider safety anew every time.

Each action uses a simple interface with a clearly defined result, making it easy to combine actions to more complex behaviors, using state machines, behavior trees or other established methods. Furthermore, the framework is easy to extend by implementing new types of constraint rules, solvers, inputs, or controllers, which can be added easily through the use of a plugin mechanism.

The approach was evaluated in the two case studies. Both simulated and real evaluations were used. The results show that the presented method is able to generate safe and robust motions from simple specifications. If the environment and the human interaction partner allow for the successful execution of the specified behavior, it is executed quickly as specified. Various scenarios in which no successful execution is possible have been evaluated to test the robustness of the approach. The results show that the robot behaves safely in all cases. Planning is automatically employed when required to resolve situations when the robot gets stuck using reactive control.

10.2 Open Research Challenges and Future Directions

Many research questions remain to be solved before assistive robots can be widely and robustly applied in human-centered environments. Many of them lie outside the scope of this work. While all of them have received research attention, none of them can be considered as generally solved. Considerable work is required on many topics before the robot presented here can be used as a generally applicable assistive robot. Aspects that are lacking at the moment include human-robot communication and human intention estimation. The robot behaviors are started manually for the evaluation, but for practical application, a better method of understanding what the human partner expects of the robot in the current moment is required. Similarly, better scene understanding, object recognition, and grasp planning are required so that the robot is no longer limited to interactions with a single object. The physical interaction, such as the phase during a handover where both participants are in contact with the object, needs further attention as well. Here, sensing capabilities are most relevant, as visual methods fall short.

The system has been evaluated in two case studies with various failure scenarios. However, no statements about the coverage of these isolated tests can be given at the moment. Given that both the environment and the human can in principle change in any possible way in any possible way, verification of control methods in dynamic environments remains a challenge. While neither formal methods

nor testing can fully capture the complexity of human-robot interaction at the moment, at least some more systematic methods of evaluating and measuring the robustness and safety of robots would be highly desirable.

Regarding the details of the presented framework, some future directions can be pointed to as well. Object manipulation is not explicitly considered by the specification at the moment. Instead, such motions have to be manually described by geometric constraints. To enable specification at a higher level of abstraction, constraints could be automatically generated from a description of the object geometries. The experiments have also shown that the lack of awareness of visibility and occlusion can be a problem. Tasks to stay in the field of view of a camera or to shine a light at a position only consider the absolute position. This can lead to results where a wall between the light source and the target is not seen as a problem. To fix this, the environment and robot models need to be enriched with information about occlusion.

Practical experience gathered during the evaluations shows that, while the specification is always respected, the emerging motions can be hard to analyze. The large amount of parameters to configure in the controllers, inputs, and rules can make it difficult to achieve the desired motion in some cases. This is further amplified by the unpredictable environment and behavior of the interaction participants. No unsafe motions are created if specified accordingly. Still, the exact cause for the resulting motions can be hard to understand.

The cause for this lies in the underlying complexity of controlling nine degrees of freedom in a dynamic environment and an interactive scenario. While machine learning techniques such as reinforcement learning have become a popular method to deal with problems that are hard to parametrize manually, safely generating enough usable data for use cases such as object handovers is still a challenge.

Bibliography

[1] Erwin Aertbeliën and Joris De Schutter. "eTaSL/eTC: A Constraint-Based Task Specification Language and Robot Controller Using Expression Graphs". eng. In: IEEE, 2014, pp. 1540–1546. ISBN: 9781479969340.

[2] Rachid Alami, Alin Albu-Schäffer, Antonio Bicchi, Rainer Bischoff, Raja Chatila, Alessandro De Luca, Agostino De Santis, Georges Giralt, Jérémie Guiochet, Gerd Hirzinger, et al. "Safe and dependable physical human-robot interaction in anthropic domains: State of the art and challenges". In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2006, pp. 1–16.

[3] Roberto Ancona. "Redundancy modelling and resolution for robotic mobile manipulators: a general approach". In: *Advanced Robotics* 31.13 (2017), pp. 706–715.

[4] Paola Ardón, Maria E Cabrera, Eric Pairet, Ronald PA Petrick, Subramanian Ramamoorthy, Katrin S Lohan, and Maya Cakmak. "Affordance-aware handovers with human arm mobility constraints". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3136–3143.

[5] Brian Armstrong-Hélouvry, Pierre Dupont, and Carlos Canudas De Wit. "A survey of models, analysis tools and compensation methods for the control of machines with friction". In: *Automatica* 30.7 (1994), pp. 1083–1138. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(94)90209-7.

[6] SMACC authors. *SMACC2*. https://smacc.dev. [Available online; accessed 08. March 2024]. 2024.

[7] *BehaviorTree.ROS2*. https://github.com/BehaviorTree/BehaviorTree.ROS2. [Available online; accessed 08. March 2024]. 2024.

[8] Amine Belaid, Boubekeur Mendil, and Ali Djenadi. "Narrow Passage RRT*: A New Variant of RRT*". In: *Int. J. Comput. Vision Robot.* 12.1 (Jan. 2022), pp. 85–100. ISSN: 1752-9131. DOI: 10.1504/ijcvr.2022.119247.

[9] Dmitry Berenson, Siddhartha Srinivasa, Dave Ferguson, and James Kuffner. "Manipulation planning on constraint manifolds". In: May 2009, pp. 625–632. DOI: 10.1109/ROBOT.2009.5152399.

[10] Joshua Bialkowski, Michael Otte, and Emilio Frazzoli. "Free-configuration biased sampling for motion planning". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 1272–1279. DOI: 10.1109/IROS.2013.6696513.

[11] Manuel Bonilla, Lucia Pallottino, and Antonio Bicchi. "Noninteracting constrained motion planning and control for robot manipulators". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 4038–4043. DOI: 10.1109/ICRA.2017.7989463.

[12] James Bruce and Manuela Veloso. "Real-time randomized path planning for robot navigation". In: *IEEE/RSJ international conference on intelligent robots and systems*. Vol. 3. IEEE. 2002, pp. 2383–2388.

[13] Giovanni Buizza Avanzini, Andrea Maria Zanchettin, and Paolo Rocco. "Constrained model predictive control for mobile robotic manipulators". In: *Robotica* 36.1 (2018), pp. 19–38. DOI: 10.1017/S0263574717000133.

[14] Gabriele Buondonno. *eiquadprog*. https://github.com/stack-of-tasks/eiquadprog. [Available online; accessed 01. November 2023]. 2023.

[15] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. *Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*. 2017. arXiv: 1611.08050 [cs.CV].

[16] Jan Carius, Martin Wermelinger, Balasubramanian Rajasekaran, Kai Holtmann, and Marco Hutter. "Deployment of an autonomous mobile manipulator at MBZIRC". In: *Journal of Field Robotics* 35 (Oct. 2018). DOI: 10.1002/rob.21825.

[17] Luca Cavalli, Gianpaolo Di Pietro, and Matteo Matteucci. "Towards affordance prediction with vision via task oriented grasp quality metrics". In: *arXiv preprint arXiv:1907.04761* (2019).

[18] Wesley P Chan, Iori Kumagai, Shunichi Nozawa, Yohei Kakiuchi, Kei Okada, and Masayuki Inaba. "Implementation of a robot-human object handover controller on a compliant underactuated hand using joint position error measurements for grip force and load force estimations". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 1190–1195.

[19] David Coleman, Ioan Alexandru Sucan, Sachin Chitta, and Nikolaus Correll. "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study". In: *CoRR* abs/1404.3785 (2014). arXiv: 1404.3785. URL: http://arxiv.org/abs/1404.3785.

[20] Michele Colledanchise and Lorenzo Natale. "Analysis and Exploitation of Synchronized Parallel Executions in Behavior Trees". In: *CoRR* abs/1908.01539 (2019). arXiv: 1908.01539. URL: http://arxiv.org/abs/1908.01539.

[21] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[22] Kevin Curran, Eoghan Furey, Tom Lunney, Jose Santos, Derek Woods, and Aidan Mccaughey. "An evaluation of indoor location determination technologies". In: *J. Location Based Services* 5 (June 2011), pp. 61–78. DOI: 10.1080/17489725.2011.562927.

[23] Mohammad-Javad Davari, Michael Hegedus, Kamal Gupta, and Mehran Mehrandezh. "Identifying Multiple Interaction Events from Tactile Data during Robot-Human Object Transfer". In: *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. 2019, pp. 1–6. DOI: 10.1109/RO-MAN46459.2019.8956306.

[24] Agostino De Santis, Bruno Siciliano, Alessandro De Luca, and Antonio Bicchi. "An atlas of physical human–robot interaction". In: *Mechanism and Machine Theory* 43.3 (2008), pp. 253–270.

[25] Joris De Schutter, Tinne De Laet, Johan Rutgeerts, Wilm Decré, Ruben Smits, Erwin Aertbeliëen, Kasper Claes, and Herman Bruyninckx. "Constraint-Based Task Specification and Estimation for Sensor-Based Robot Systems in the Presence of Geometric Uncertainty". In: *The International Journal of Robotics Research* 26.5 (2007), pp. 433–455.

[26] Wilm Decré, Ruben Smits, Herman Bruyninckx, and Joris De Schutter. "Extending iTaSC to support inequality constraints and non-instantaneous task specification". In: *Proceedings of the 2009 IEEE International Conference on Robotics and Automation*. Kobe, Japan, 2009, pp. 964–971.

[27] *Robots and robotic devices – Collaborative robots*. Standard. International Organization for Standardization, Mar. 2016.

[28] Ke Dong, Karime Pereida, Florian Shkurti, and Angela P. Schoellig. "Catch the Ball: Accurate High-Speed Motions for Mobile Manipulators via Inverse Dynamics Learning". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 6718–6725. DOI: 10.1109/IROS45743.2020.9341134.

[29] A Gomez Eguiluz, Iñaki Rañó, Sonya A Coleman, and T Martin McGinnity. "Reliable object handover through tactile force sensing and effort control in the shadow robot hand". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 372–377.

[30] Adrien Escande, Sylvain Miossec, Mehdi Benallegue, and Abderrahmane Kheddar. "A Strictly Convex Hull for Computing Proximity Distances With Continuous Gradients". In: *Robotics, IEEE Transactions on* 30 (June 2014), pp. 666–678. DOI: 10.1109/TRO.2013.2296332.

[31] Marco Faroni, Manuel Beschi, and Nicola Pedrocchi. "An MPC Framework for Online Motion Planning in Human-Robot Collaborative Tasks". In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019, pp. 1555–1558. DOI: 10.1109/ETFA.2019.8869047.

[32] Dave Ferguson, Nidhi Kalra, and Anthony Stentz. "Replanning with rrts". In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE. 2006, pp. 1243–1248.

[33] H.J. Ferreau, C. Kirches, A. Potschka, H.G. Bock, and M. Diehl. "qpOASES: A parametric active-set algorithm for quadratic programming". In: *Mathematical Programming Computation* 6.4 (2014), pp. 327–363.

[34] Hans Joachim Ferreau, Peter Ortner, Peter Langthaler, Luigi del Re, and Moritz Diehl. "Predictive control of a real-world Diesel engine using an extended online active set strategy". In: *Annual Reviews in Control* 31.2 (2007), pp. 293–301. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2007.09.001.

[35] Fabrizio Flacco, Torsten Kroger, Alessandro Luca, and Oussama Khatib. "Depth space approach to human-robot collision avoidance". In: *Proceedings - IEEE International Conference on Robotics and Automation* (May 2012), pp. 338–345. DOI: 10.1109/ICRA.2012.6225245.

[36] Chien-Liang Fok, Gwendolyn Johnson, John D Yamokoski, Aloysius Mok, and Luis Sentis. "ControlIt!a software framework for whole-body operational space control". In: *International Journal of Humanoid Robotics* 13.01 (2016), p. 1550040.

[37] Chien-Liang Fok and Luis Sentis. "Integration and usage of a ROS-Based whole body control software framework". In: *Robot Operating System (ROS) The Complete Reference (Volume 1)* (2016), pp. 535–563.

[38] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. "The dynamic window approach to collision avoidance". In: *IEEE Robotics & Automation Magazine* 4.1 (1997), pp. 23–33.

[39] Daichi Furuta, Kyo Kutsuzawa, Sho Sakaino, and Toshiaki Tsuji. "LSTM Learning of Inverse Dynamics with Contact in Various Environments". In: *2018 12th France-Japan and 10th Europe-Asia Congress on Mechatronics*. 2018, pp. 149–154. DOI: 10.1109/MECATRONICS.2018.8495698.

[40] Magnus Gaertner, Marko Bjelonic, Farbod Farshidian, and Marco Hutter. "Collision-free MPC for legged robots in static and dynamic scenes". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 8266–8272.

[41] Mirosaw Galicki. "An adaptive non-linear constraint control of mobile manipulators". In: *Mechanism and Machine Theory* 88 (2015), pp. 63–85. ISSN: 0094-114X. DOI: https://doi.org/10.1016/j.mechmachtheory.2015.02.001. URL: https://www.sciencedirect.com/science/article/pii/S0094114X15000269.

[42] Cipriano Galindo, Juan-Antonio Fernández-Madrigal, Javier González, and Alessandro Saffiotti. "Robot task planning using semantic maps". In: *Robotics and autonomous systems* 56.11 (2008), pp. 955–966.

[43] M. Gautier. "Numerical calculation of the base inertial parameters of robots". In: *Journal of Robotic Systems* 8.4 (1991), pp. 485–506. DOI: https://doi.org/10.1002/rob.4620080405.

[44] Claudio Gaz, Marco Cognetti, Alexander Oliva, Paolo Robuffo Giordano, and Alessandro De Luca. "Dynamic Identification of the Franka Emika Panda Robot With Retrieval of Feasible Parameters Using Penalty-Based Optimization". In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 4147–4154. DOI: 10.1109/LRA.2019.2931248.

[45] *Gazebo*. www.gazebosim.org. [Online; accessed 15-December-2023]. 2023.

[46] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. *Behavior Trees and State Machines in Robotics Applications*. 2023. arXiv: 2208.04211 [cs.RO].

[47] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. "A fast procedure for computing the distance between complex objects in three-dimensional space". In: *IEEE Journal on Robotics and Automation* 4.2 (1988), pp. 193–203. DOI: 10.1109/56.2083.

[48] Donald Goldfarb and Ashok Idnani. "A numerically stable dual method for solving strictly convex quadratic programs". In: *Mathematical programming* 27.1 (1983), pp. 1–33.

[49] Miguel Á. González-Santamarta, Francisco J. Rodríguez-Lera, Vicente Matellán-Olivera, and Camino Fernández-Llamas. "YASMIN: Yet Another State MachINe". In: *ROBOT2022: Fifth Iberian Robotics Conference*. Ed. by Danilo Tardioli, Vicente Matellán, Guillermo Heredia, Manuel F. Silva, and Lino Marques. Cham: Springer International Publishing, 2023, pp. 528–539. ISBN: 978-3-031-21062-4.

[50] Sami Haddadin, Alin Albu-Schäffer, and Gerd Hirzinger. "Requirements for safe robots: Measurements, analysis and new insights". In: *The International Journal of Robotics Research* 28.11-12 (2009), pp. 1507–1527.

[51] Sami Haddadin, Alessandro De Luca, and Alin Albu-Schäffer. "Robot Collisions: A Survey on Detection, Isolation, and Identification". In: *IEEE Transactions on Robotics* 33.6 (2017), pp. 1292–1312. DOI: 10.1109/TRO.2017.2723903.

[52] Lorenz Halt, Frank Nagele, Philipp Tenbrock, and Andreas Pott. "Intuitive Constraint-Based Robot Programming for Robotic Assembly Tasks". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 520–526. DOI: 10.1109/ICRA.2018.8462882.

[53] Li Han, Lee Rudolph, Jonathon Blumenthal, and Ihar Valodzin. "Stratified deformation space and path planning for a planar closed chain with revolute joints". In: *Algorithmic Foundation of Robotics VII: Selected Contributions of the Seventh International Workshop on the Algorithmic Foundations of Robotics*. Springer. 2008, pp. 235–250.

[55] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[56] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[57] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[58] Enrico Mingo Hoffman, Alessio Rocchi, Arturo Laurenzi, and Nikos G. Tsagarakis. “Robot control for dummies: Insights and examples using OpenSoT”. In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. 2017, pp. 736–741. DOI: 10.1109/HUMANOIDS.2017.8246954.

[60] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. “OctoMap: An efficient probabilistic 3D mapping framework based on octrees”. In: *Autonomous robots* 34 (2013), pp. 189–206.

[61] *IROS 2020 Workshop on Bringing Constraint-based Robot Programming to Real-World Applications (IROS CobaRoP)*. https://iros2020-workshop-cobarop.gitlab.io/. Accessed: 2024-01-18. Las Vegas, NV, USA.

[62] Léonard Jaillet, Juan Cortés, and Thierry Siméon. “Transition-based RRT for path planning in continuous cost spaces”. In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2008, pp. 2145–2150.

[63] Léonard Jaillet and Josep M. Porta. “Path Planning Under Kinematic Constraints by Rapidly Exploring Manifolds”. In: *IEEE Transactions on Robotics* 29.1 (2013), pp. 105–117. DOI: 10.1109/TRO.2012.2222272.

[64] Advait Jain and Charles Kemp. “EL-E: An assistive mobile manipulator that autonomously fetches objects from flat surfaces”. In: *Autonomous Robots* 28 (Sept. 2010), pp. 45–64. DOI: 10.1007/s10514-009-9148-5.

[65] Wenji Jia, Guiling Yang, Lefeng Gu, and Tianjiang Zheng. “Dynamics modelling of a mobile manipulator with powered castor wheels”. In: *2017 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. IEEE. 2017, pp. 730–735.

[66] Sertac Karaman and Emilio Frazzoli. *Incremental Sampling-based Algorithms for Optimal Motion Planning*. 2010. arXiv: 1005.0416 [cs.RO].

[67] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. “Probabilistic roadmaps for path planning in high-dimensional configuration spaces”. In: *IEEE transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[68] Ameer Hamza Khan, Shuai Li, Dechao Chen, and Liefa Liao. “Tracking control of redundant mobile manipulator: An RNN based metaheuristic approach”. In: *Neurocomputing* 400 (2020), pp. 272–284. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2020.02.109.

[69] O. Khatib. "Real-time obstacle avoidance for manipulators and mobile robots". In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. 1985, pp. 500–505. DOI: 10.1109/ROBOT.1985.1087247.

[70] Beobkyoon Kim, Terry Taewoong Um, Chansu Suh, and F. C. Park. "Tangent bundle RRT: A randomized algorithm for constrained motion planning". In: *Robotica* 34.1 (2016), pp. 202–225. DOI: 10.1017/S0263574714001234.

[71] Zachary Kingston, Mark Moll, and Lydia E Kavraki. "Sampling-based methods for motion planning with constraints". In: *Annual review of control, robotics, and autonomous systems* 1 (2018), pp. 159–185.

[72] Zachary Kingston, Mark Moll, and Lydia E. Kavraki. "Exploring Implicit Spaces for Constrained Sampling-Based Planning". In: *Intl. J. of Robotics Research* 38.10–11 (Sept. 2019), pp. 1151–1178. DOI: 10.1177/0278364919868530.

[73] Mia Kokic, Danica Kragic, and Jeannette Bohg. "Learning task-oriented grasping from human activity datasets". In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 3352–3359.

[74] Jelizaveta Konstantinova, Senka Krivic, Agostino Stilli, Justus Piater, and Kaspar Althoefer. "Autonomous object handover using wrist tactile information". In: *Towards Autonomous Robotic Systems: 18th Annual Conference, TAROS 2017, Guildford, UK, July 19–21, 2017, Proceedings 18*. Springer. 2017, pp. 450–463.

[75] J.J. Kuffner and S.M. LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 2. 2000, 995–1001 vol.2. DOI: 10.1109/ROBOT.2000.844730.

[76] Tobias Kunz and Mike Stilman. "Manipulation planning with soft task constraints". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 1937–1942. DOI: 10.1109/IROS.2012.6386134.

[77] Steven LaValle. "Rapidly-exploring random trees: A new tool for path planning". In: *Research Report 9811* (1998).

[78] Rafael Lazimy. "Mixed-integer quadratic programming". In: *Mathematical Programming* 22 (1982), pp. 332–349.

[79] Quentin Leboutet, Julien Roux, Alexandre Janot, Julio Rogelio Guadarrama-Olvera, and Gordon Cheng. "Inertial parameter identification in robotics: A survey". In: *Applied Sciences* 11.9 (2021), p. 4303.

[80] Wei Li and Rong Xiong. "Dynamical Obstacle Avoidance of Task-Constrained Mobile Manipulation Using Model Predictive Control". In: 7 (2019), pp. 88301–88311. DOI: 10.1109/ACCESS.2019.2925428.

[81] Daegyu Lim, Donghyeon Kim, and Jaeheung Park. “Momentum Observer-Based Collision Detection Using LSTM for Model Uncertainty Learning”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 4516–4522. DOI: 10.1109/ICRA48506.2021.9561667.

[82] Björn Lindqvist, Sina Sharif Mansouri, Ali-akbar Agha-mohammadi, and George Nikolakopoulos. “Nonlinear MPC for collision avoidance and control of UAVs with dynamic obstacles”. In: *IEEE robotics and automation letters* 5.4 (2020), pp. 6001–6008.

[83] Nan Liu, Liangyu Li, Bing Hao, Liusong Yang, Tonghai Hu, Tao Xue, and Shoujun Wang. “Modeling and Simulation of Robot Inverse Dynamics Using LSTM-based Deep Learning Algorithm for Smart Cities and Factories”. In: *IEEE Access* 7 (2019), pp. 173989–173998. DOI: 10.1109/ACCESS.2019.2957019.

[84] Stefan B. Liu and Matthias Althoff. “Reachset Conformance of Forward Dynamic Models for the Formal Analysis of Robots”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Madrid, Spain: IEEE Press, 2018, pp. 370–376. DOI: 10.1109/IROS.2018.8593975.

[85] Vicon Motion Systems Ltd. *Vicon Motion Capture Systems*. https://www.vicon.com/. [Available online; accessed 11. March 2024]. 2024.

[86] A. de Luca and R. Mattone. “Sensorless Robot Collision Detection and Hybrid Force/Motion Control”. In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2005, pp. 999–1004. DOI: 10.1109/ROBOT.2005.1570247.

[87] Alessandro De Luca and Raffaella Mattone. “Actuator failure detection and isolation using generalized momenta”. In: *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)* 1 (2003), 634–639 vol.1.

[88] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. “The Marathon 2: A Navigation System”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020. URL: https://github.com/ros-planning/navigation2.

[89] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074.

[90] Bence Magyar, Nikolaos Tsiogkas, Jérémie Deray, Sammy Pfeiffer, and David Lane. “Timed-elastic bands for manipulation motion planning”. In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 3513–3520.

[91] Shamil Mamedov and Stanislav Mikhel. “Practical Aspects of Model-Based Collision Detection”. In: *Frontiers in Robotics and AI* 7 (2020). ISSN: 2296-9144. DOI: 10.3389/frobt.2020.571574.

[92] N. Mansard, O. Stasse, P. Evrard, and A. Kheddar. "A Versatile Generalized Inverted Kinematics Implementation for Collaborative Working Humanoid Robots: The Stack of Tasks". In: *International Conference on Advanced Robotics (ICAR)*. June 2009, p. 119.

[93] Nestor Maslej, Loredana Fattorini, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Helen Ngo, Juan Carlos Niebles, Vanessa Parli, Yoav Shoham, Russell Wald, Jack Clark, and Raymond Perrault. *Artificial Intelligence Index Report 2023*. 2023. arXiv: 2310.03715 [cs.AI].

[94] Troy McMahon, Shawna Thomas, and Nancy M. Amato. "Sampling-based motion planning with reachable volumes: Theoretical foundations". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 6514–6521. DOI: 10.1109/ICRA.2014.6907820.

[95] Amirhossein H Memar and Ehsan T Esfahani. "Modeling and dynamic parameter identification of the Schunk Powerball robotic arm". In: *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 57144. American Society of Mechanical Engineers. 2015, V05CT08A024.

[96] Joseph Mirabel, Steve Tonneau, Pierre Fernbach, Anna-Kaarina Seppälä, Mylène Campana, Nicolas Mansard, and Florent Lamiraux. "HPP: A new software for constrained motion planning". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 383–389. DOI: 10.1109/IROS.2016.7759083.

[97] AJung Moon, Daniel M Troniak, Brian Gleeson, Matthew KXJ Pan, Minhua Zheng, Benjamin A Blumer, Karon MacLean, and Elizabeth A Croft. "Meet me where i'm gazing: how shared attention gaze affects human-robot handover timing". In: *Proceedings of the 2014 ACM/IEEE international conference on Human-robot interaction*. 2014, pp. 334–341.

[98] M. Morozov, J. Riise, R. Summan, S.G. Pierce, C. Mineo, C.N. MacLeod, and R.H. Brown. "Assessing the accuracy of industrial robots through metrology for the enhancement of automated non-destructive testing". In: *2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. 2016, pp. 335–340. DOI: 10.1109/MFI.2016.7849510.

[99] Dennis Mronga, Tobias Knobloch, José de Gea Fernández, and Frank Kirchner. "A Constraint-Based Approach for Human-Robot Collision Avoidance". In: *Advanced Robotics* (2020), pp. 1–17.

[100] *Neobotix MPO-700*. www.neobotix-robots.com. [Online; accessed 02-March-2021]. 2021.

[101] *NVIDIA Jetson Orin Modules*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. [Online; accessed 02-February-2024]. 2024.

[102] Henrik Olsson, Karl Johan Åström, Carlos Canudas De Wit, Magnus Gäfvert, and Pablo Lischinsky. “Friction models and friction compensation”. In: *Eur. J. Control* 4.3 (1998), pp. 176–195.

[103] Roberto Opromolla, Giancarmine Fasano, Giancarlo Rufino, and Michele Grassi. “Characterization and Testing of a High-Resolution Time-of-Flight Camera for Autonomous Navigation”. In: *2018 5th IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*. 2018, pp. 380–385. DOI: 10.1109/MetroAeroSpace.2018.8453522.

[104] Valerio Ortenzi, Akansel Cosgun, Tommaso Pardi, Wesley P. Chan, Elizabeth Croft, and Dana Kuli. “Object Handovers: A Review for Robotics”. In: *IEEE Transactions on Robotics* 37.6 (2021), pp. 1855–1873. DOI: 10.1109/TRO.2021.3075365.

[105] Andreas Orthey, Sohaib Akbar, and Marc Toussaint. “Multilevel Motion Planning: A Fiber Bundle Formulation”. In: *CoRR* abs/2007.09435 (2020). arXiv: 2007.09435. URL: https://arxiv.org/abs/2007.09435.

[106] Andreas Orthey and Marc Toussaint. “Rapidly-Exploring Quotient-Space Trees: Motion Planning using Sequential Simplifications”. In: *CoRR* abs/1906.01350 (2019). arXiv: 1906.01350. URL: http://arxiv.org/abs/1906.01350.

[107] Michael Otte and Emilio Frazzoli. “RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning”. In: *The International Journal of Robotics Research* 35.7 (2016), pp. 797–822. DOI: 10.1177/0278364915594679.

[108] Tony Owen. “The Complexity of Robot Motion Planning”. In: *Robotica* 8.3 (1990), pp. 259–260. DOI: 10.1017/S0263574700000151.

[109] Prabin Kumar Panigrahi and Sukant Kishoro Bisoy. “Localization strategies for autonomous mobile robots: A review”. In: *Journal of King Saud University - Computer and Information Sciences* 34.8, Part B (2022), pp. 6019–6039. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2021.02.015. URL: https://www.sciencedirect.com/science/article/pii/S1319157821000550.

[110] Ken Perlin. “An Image Synthesizer”. In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’85. New York, NY, USA: Association for Computing Machinery, 1985, pp. 287–296. ISBN: 0897911660. DOI: 10.1145/325334.325247. URL: https://doi.org/10.1145/325334.325247.

[111] Mun Seng Phoon, Philipp S. Schmitt, and Georg V. Wichert. “Constraint-based Task Specification and Trajectory Optimization for Sequential Manipulation”. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2022, pp. 197–202. DOI: 10.1109/IROS47612.2022.9981909.

[112] Alexander Poeppel, Alwin Hoffmann, Martin Siehler, and Wolfgang Reif. "Robust Distance Estimation of Capacitive Proximity Sensors in HRI using Neural Networks". In: *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*. 2020, pp. 344–351. DOI: 10.1109/IRC.2020.00061.

[113] Athanasios S. Polydoros, Evangelos Boukas, and Lazaros Nalpantidis. "Online multi-target learning of inverse dynamics models for computed-torque control of compliant manipulators". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 4716–4722. DOI: 10.1109/IROS.2017.8206344.

[114] Ronghuai Qi, Amir Khajepour, and William W. Melek. "Modeling, tracking, vibration and balance control of an underactuated mobile manipulator (UMM)". In: *Control Engineering Practice* 93 (2019), p. 104159. ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2019.104159.

[115] *Real-Time Linux*. https://wiki.linuxfoundation.org/realtime/start. [Online; accessed 06-February-2024]. 2024.

[116] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV].

[117] Open Robotics. *Creating and using plugins*. https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Pluginlib.html. [Available online; accessed 08. March 2024]. 2024.

[118] Weiss Robotics. *CRG Series*. https://weiss-robotics.com/servo-electric/crg-series/product/crg/selectVariant/crg-200-085-191/. [Available online; accessed 13. December 2023]. 2023.

[119] Samuel Rodriguez, Shawna Thomas, Roger Pearce, and Nancy M. Amato. "RESAMPL: A Region-Sensitive Adaptive Motion Planner". In: *Algorithmic Foundation of Robotics VII: Selected Contributions of the Seventh International Workshop on the Algorithmic Foundations of Robotics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 285–300. DOI: 10.1007/978-3-540-68405-3_18.

[120] Christoph Roesmann, Wendelin Feiten, Thomas Woesch, Frank Hoffmann, and Torsten Bertram. "Trajectory modification considering dynamic constraints of autonomous robots". In: *ROBOTIK 2012; 7th German Conference on Robotics*. 2012, pp. 1–6.

[121] Elmar Rueckert, Moritz Nakatenus, Samuele Tosatto, and Jan Peters. "Learning inverse dynamics models in O(n) time with LSTM networks". In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. 2017, pp. 811–816. DOI: 10.1109/HUMANOIDS.2017.8246965.

[122] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[123] Alexander Schiendorfer, Alexander Knapp, Gerrit Anders, and Wolfgang Reif. “MiniBrass: soft constraints for MiniZinc”. In: *Constraints* 23 (2018), pp. 403–450.

[124] Philipp Schillinger, Stefan Kohlbrecher, and Oskar von Stryk. “Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics”. In: *IEEE International Conference on Robotics and Automation*. Stockholm, Sweden, May 2016.

[125] Philipp S Schmitt, Florian Wirnshofer, Kai M Wurm, Georg v Wichert, and Wolfram Burgard. “Modeling and planning manipulation in dynamic environments”. In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 176–182.

[126] Philipp S. Schmitt, Florian Wirnshofer, Kai M. Wurm, Georg v. Wichert, and Wolfram Burgard. “Planning Reactive Manipulation in Dynamic Environments”. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 136–143. DOI: 10.1109/IROS40897.2019.8968452.

[127] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. “Efficient RANSAC for point-cloud shape detection”. In: *Computer graphics forum*. Vol. 26. 2. Wiley Online Library. 2007, pp. 214–226.

[128] *Schunk LWA 4P*. www.schunk.com. [Online; accessed 02-March-2021]. 2021.

[129] Shashank Sharma, Gerhard K. Kraetzschmar, Christian Scheurer, and Rainer Bischoff. “Unified Closed Form Inverse Kinematics for the KUKA youBot”. In: *ROBOTIK 2012; 7th German Conference on Robotics*. 2012, pp. 1–6.

[130] Alexander Sherikov. *Comparative benchmark of QP solvers*. https://github.com/asherikov/qpmad_benchmark. [Available online; accessed 04. December 2023]. 2023.

[131] Alexander Sherikov. *qpmad: Eigen-based C++ QP solver*. https://www.sherikov.net/qpmad. [Available online; accessed 01. November 2023]. 2023.

[132] Bruno Siciliano, Oussama Khatib, and Torsten Kröger. *Springer handbook of robotics*. Vol. 200. Springer, 2008.

[133] *Sick microScan 3*. https://www.sick.com/de/en/catalog/products/safety/safety-laser-scanners/microscan3/c/g295657. [Online; accessed 06-February-2024]. 2024.

[134] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[135] Mike Stilman. “Task constrained motion planning in robot joint space”. In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2007, pp. 3074–3081.

[136] Kyle Strabala, Min Kyung Lee, Anca Dragan, Jodi Forlizzi, Siddhartha S. Srinivasa, Maya Cakmak, and Vincenzo Micelli. "Toward seamless human-robot handovers". In: *J. Hum.-Robot Interact.* 2.1 (Feb. 2013), pp. 112–132. DOI: 10.5898/JHRI.2.1.Strabala. URL: https://doi.org/10.5898/JHRI.2.1.Strabala.

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

[140] Jan Swevers, Walter Verdonck, and Joris De Schutter. "Dynamic model identification for industrial robots". In: *IEEE control systems magazine* 27.5 (2007), pp. 58–71.

[141] Claudio Urrea and José Pascal. "Design, simulation, comparison and evaluation of parameter identification methods for an industrial robot". In: *Computers & Electrical Engineering* 67 (2018), pp. 791–806. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2016.09.004.

[142] José Varela-Aldás, Víctor Hugo Andaluz, and Fernando A. Chicaiza. "Modelling and Control of a Mobile Manipulator for Trajectory Tracking". In: *2018 International Conference on Information Systems and Computer Science (INCISCOS)*. 2018, pp. 69–74. DOI: 10.1109/INCISCOS.2018.00018.

[143] Jiangping Wang, Shirong Liu, Botao Zhang, and Changbin Yu. "Manipulation planning with soft constraints by randomized exploration of the composite configuration space". In: *International Journal of Control, Automation and Systems* 19 (2021), pp. 1340–1351.

[144] Jiankun Wang, Max Q.-H. Meng, and Oussama Khatib. "EB-RRT: Optimal Motion Planning for Mobile Robots". In: *IEEE Transactions on Automation Science and Engineering* 17.4 (2020), pp. 2063–2073. DOI: 10.1109/TASE.2020.2987397.

[145] Yuquan Wang and Lihui Wang. "Real-time collision-free multi-objective robot motion generation". In: *Advanced Human-Robot Collaboration in Manufacturing*. Springer, 2021, pp. 115–139.

[146] Jiafeng Wu, Xianghua Ma, Tongrui Peng, and Haojie Wang. "An Improved Timed Elastic Band (TEB) Algorithm of Autonomous Ground Vehicle (AGV) in Complex Environment". In: *Sensors* 21.24 (2021). ISSN: 1424-8220. DOI: 10.3390/s21248312. URL: https://www.mdpi.com/1424-8220/21/24/8312.

[147] Tian Xu, Jizhuang Fan, Yiwen Chen, Xianyao Ng, Marcelo H. Ang, Qianqian Fang, Yanhe Zhu, and Jie Zhao. "Dynamic Identification of the KUKA LBR iiwa Robot With Retrieval of Physical Parameters Using Global Optimization". In: *IEEE Access* 8 (2020), pp. 108018–108031. DOI: 10.1109/ACCESS.2020.3000997.

[148] Nural Yilmaz, Jie Ying Wu, Peter Kazanzides, and Ugur Tumerdem. "Neural Network based Inverse Dynamics Identification and External Force Estimation on the da Vinci Research Kit". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 1387–1393. DOI: 10.1109/ICRA40945.2020.9197445.

[149] Matt Zucker, James Kuffner, and Michael Branicky. "Multipartite RRTs for rapid replanning in dynamic environments". In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE. 2007, pp. 1603–1609.

[150] Joshua M. Zutell, David C. Conner, and Philipp Schillinger. *Flexible Behavior Trees: In search of the mythical HFSMBTH for Collaborative Autonomy in Robotics*. 2022. arXiv: 2203.05389 [cs.RO].

Own Publications

[54] Julian Hanke, Matthias Stueben, Christian Eymüller, Maximilian Enrico Müller, Alexander Poeppel, and Wolfgang Reif. "CASP: Computer Aided Specimen Placement for Robot-Based Component Testing". In: *Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics, ICINCO 2023, Rome, Italy, November 13-15, 2023, Volume 1*. Ed. by Giuseppina Gini, Henk Nijmeijer, and Dimitar P. Filev. SCITEPRESS, 2023, pp. 374–382. DOI: 10.5220/0012155000003543. URL: https://doi.org/10.5220/0012155000003543.

[59] Alwin Hoffmann, Andreas Schierl, Andreas Angerer, Matthias Stueben, Michael Vistein, and Wolfgang Reif. "Robot collision avoidance using an environment model for capacitive sensors". In: *Planning, Control, and Sensing for Safe Human-Robot Interaction, 2015 IEEE International Conference on Robotics and Automation, Seattle, USA, May 26-30, 2015*. 2015. URL: https://cs.stanford.edu/people/tkr/icra2015/index.html.

[137] Matthias Stueben, Alwin Hoffmann, and Wolfgang Reif. "Constraint-based Whole-Body-Control of Mobile Manipulators in Human-Centered Environments". In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2021, pp. 1–8. DOI: 10.1109/ETFA45728.2021.9613281.

[138] Matthias Stueben, Alexander Poeppel, and Wolfgang Reif. "External Torque Estimation for Mobile Manipulators: A Comparison of Model-based and LSTM Methods". In: *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*. 2022, pp. 95–102. DOI: 10.1109/IRC55401.2022.00026.

List of Figures

2.1 Example of a behavior tree. 10

3.1 The real robot compared to its simulated model and the simplified collision geometry. 21

4.1 Photo of the camera rack and the mounted cameras. 24
4.2 Example depth scan from the Basler ToF camera. 26
4.3 Example images from the 360-degree camera using equirectangular and perspective projection. 28
4.4 Class Diagram of the `Context` model with the most important elements. 29
4.5 The excitation trajectory for the arm joints. 35
4.6 Velocities of the base in the excitation trajectory. 36
4.7 Our LSTM network architecture. 38
4.8 The evaluation trajectory of the manipulator joints. 40
4.9 The evaluation trajectory of the base. 41
4.10 Validation of the estimated torques of the dynamic model and the LSTM network on a 120 s test trajectory. 44
4.11 Estimated and reference external torques with an attached weight of 3.25 kg. 45

5.1 Plot of a joint limit constraint for a single joint with limits at -1 and 1. 52
5.2 Overview of the concepts in our control architecture. 56
5.3 Class diagram of the implemented input types rules. 57
5.4 Class diagram of the implemented constraint rules. 58
5.5 Illustration of the Cartesian distance constraint rule. 60
5.6 Illustration of the aiming constraint rule. 61
5.7 Illustration of the field-of-view constraint rule. 62
5.8 Illustration of the self-collision avoidance constraint rule. 64
5.9 Illustration of the obstacle avoidance constraint rule. 67
5.10 Plot of the output of the `Follow`-Controller (equation (5.6)) with $L = 1.0$ and various values for α. 69
5.11 Plots of the output of the `Limit`-controller (equation (5.8)) with $L = 1.0$. Left side: $\mathbf{c}_0 = 5.0$, right side: $\mathbf{c}_0 = 7.5$. 70
5.12 Plots of the output of the `Stopping`-controller (equation (5.9)) with $L = 1.0$. Left side: $\mathbf{c}_0 = 5.0$, right side: $\mathbf{c}_0 = 7.5$. 71
5.13 Plots of the output of the `Hybrid`-controller with $L = 1.0$. Left side: $x_0 = 5.0$, right side: $x_0 = 7.5$. 72
5.14 Class diagram showing the fundamental structure of `Tasks` 74
5.15 Class diagram of an action. 75

5.16 Object diagram of an example action. 77
5.17 Object diagram of the safety tasks. 78
5.18 Behavior of the platform's hardware controllers at small velocities. 83
5.19 Class diagram of the available solver types. 85
5.20 Object diagram of the action used in cartesian motion evaluation. . 86
5.21 Object diagram of the action used in the nullspace motion evaluation. 87
5.22 Example photos of the evaluation experiments. 88
5.23 Cartesian end-effector error without platform minimum velocities. 90
5.24 Cartesian end-effector error with platform minimum velocities. . . 91
5.25 Photograph of the evaluation scenario with a box obstacle. 92
5.26 Results of the evaluation with an obstacle. 93
5.27 Required calculations times of the qpmad-based solvers. 95
5.28 Required calculations times of the qpOASES-based solvers. 95

6.1 Constraint manifold of a two-dimensional robot for equality and inequality constraints. 104
6.2 The pipeline of our planning method. 111
6.3 Visualization of the three configuration spaces used with multi-level planning. 113
6.4 Visualization of the goal-finding process. 115
6.5 Visualization of the first evaluated planning scenario, *Cartesian Goal with Obstacle*. 119
6.6 Visualization of the best and worst generated paths for the scenario *Cartesian Goal with Obstacle*. 120
6.7 Average path cost and required planning time of the planners for the scenario *Cartesian Goal with Obstacle*. 121
6.8 The *Narrow Passage* planning scenario. The collision geometry of the gripper is highlighted in red. 122
6.9 Object diagram of the action used in the *narrow passage* evaluation scenario. 123
6.10 Finding a constraint-satisfying initial configuration in the *Narrow Passage* scenario. 123
6.11 Best path generated for the *Narrow Passage* scenario. 124
6.12 Worst path generated for the *Narrow Passage* scenario. 124
6.13 Average path cost and required planning time of the planners in the *Narrow Passage* evaluation scenario. 125
6.14 Object diagram of the action used in the *Making Room* evaluation scenario. 126
6.15 Some of the goal configurations determined for the *Making Room* scenario. 127
6.16 Visualization of the best and worst generated paths for the *Making Room* scenario. 128
6.17 Average path cost and required planning time of the planners in the *Making Room* evaluation scenario. 129

7.1 State machine of the execution of an action in `reactive` mode. . . 135
7.2 State machine of a planned action. 136
7.3 State machine of a planned action. 138

8.1 Overview of the ROS2 interfaces. 140

9.1 Graphical elements in FlexBE 148
9.2 State machine of the object handover behavior, as it appears in FlexBE. 150
9.3 State machine of the behavior to pick up the flashlight, as it appears in FlexBE. 151
9.4 Photographs of the experiments. 152
9.5 State machine of the case study B, as it appears in FlexBE. 152
9.6 Overview of the path tasks in the action `follow_and_light`. . . . 153
9.7 Overview of the goal tasks in the action `follow_and_light`. 154
9.8 Overview of the cost tasks in the action `follow_and_light`. 155
9.9 Object diagram of the configuration used during the flashlight pickup. Safety tasks are excluded. 155
9.10 Accuracy plot of the linear motion downwards toward the flashlight. 156
9.11 Recorded data of gripper velocity and distance to the person's hand 158
9.12 Object diagram of the action `gripper_to_hand` 159
9.13 Handover scenario with obstacles completely blocking the robot from the person. 160
9.14 Handover scenario with additional obstacles. 161
9.15 Example of an uncooperative person during the handover: Reaching past the gripper. 162
9.16 Plot of recorded gripper velocity and its distance to the human hand. 163
9.17 Robot shining its flashlight at a person's feet. 164
9.18 Plots of the recorded data during following and lighting. 165
9.19 Plots of commanded velocity and constraint costs. 166

List of Tables

4.1 Denavit-Hartenberg parameters of our robot model 32
4.2 Mean squared error of the momentum observer estimations. 40
4.3 Mean squared error of the LSTM estimations. 41
4.4 Base parameters and their identified values 43

5.1 Average and maximum calculation times for a full control cycle and for solving the QP problem using the different solvers. 94

6.1 Planner results for the scenario *Cartesian Goal with Obstacle* . . . 119
6.2 Planner results for the scenario *Narrow Passage*. 125
6.3 Planner results for the scenario *Making Room*. 128

List of Listings

8.1 Example specification of an action in a YAML configuration file. . 142
8.2 Example specification of a task in a YAML configuration file. . . . 143
8.3 Specification of the `base_link` input in the input configuration file in YAML format. 143
8.4 Specification of the `genericFollowController` in the controller configuration file in YAML format. 144

List of Algorithms

6.1 Basic procedure of tree-based planning. 100
6.2 Algorithm for the calculation of goal configurations. 114
6.3 Algorithm for the calculation of a valid initial configuration. 116