Universität Bremen

Fachbereich 3: Mathematik und Informatik

Master's Thesis

Kineverse: A Framework for Maintaining and Exchanging Articulation Models for Mobile Manipulation Robots

Kineverse: Ein Framework für die Verwaltung und den Austausch von Artikulationsmodellen in mobilen Manipulatoren

Adrian Röfer

Matriculation No.4012444

March 20, 2020

Adrian Röfer

Kineverse: A Framework for Maintaining and Exchanging Articulation Models for Mobile Manipulation Robots

Kineverse: Ein Framework für die Verwaltung und den Austausch von Artikulationsmodellen in mobilen Manipulatoren

Master's Thesis, Fachbereich 3: Mathematik und Informatik

Universität Bremen, März 2020

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 20.03.2020

_____

Adrian Röfer

Abstract

As robots become more and more ubiquitous in every-day life, they will need to interact with unfamiliar environments. These interactions require flexible and expandable articulation models, which can change and expand as the robot makes new observations. The current frameworks for expressing articulation models are hard to expand, due to their abstract descriptions of models, which require human implementation work to integrate expansions into existing software. In the environment of the *Robot Operating System* (ROS), the most popular robotics application middleware today, there are standards for exchanging many types of data, but there is currently no such standard for exchanging articulation models at runtime.

In this thesis, I propose expressing articulation models as symbolic mathematical expressions and constraints. I argue that this formulation is easier to expand than the existing frameworks and is thus more scalable. I implement a proof-of-concept framework using this formulation and showcase its features given multiple use cases. The framework is integrated with the ROS environment and includes a networked system for managing and exchanging articulation models in a running system. I evaluate the framework by modeling a number of different articulations, including some that cannot be expressed in URDF, the current standard framework in ROS. I also use the implementation in a few small tasks, demonstrating its applicability in an online system.

Contents

Chapter 1

Introduction

1.1 Motivation

When people envision the robots of the future, many imagine a personal assistant who helps them with their daily chores. Cleaning the house, putting things away, as the robot does in Fig. 1.1, maybe even go out grocery shopping. All of these tasks require the robot to interact robustly with its environment.

Figure 1.1   A PR2 robot stowing groceries in a kitchen.

Much like people do, robots will need to be able to look at their environment and determine how they can interact with it. Is the thing they are looking at the front of a drawer, or a door to a cupboard? Is the lid of the teapot attached to its body, or will it fall if the pot is tilted too much? Will a switch control the oven or the stove top?
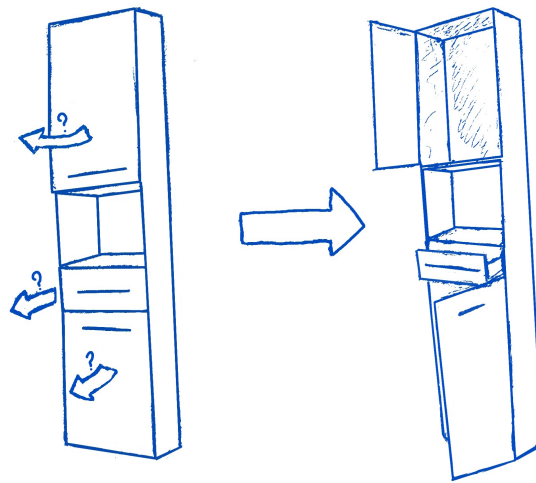
Figure 1.2 Examples of unpredictable environments: How do the parts of the cupboard move?

People seem to be able to answer all of these questions intuitively, as they, for instance, start making an omelette in an AirBnB's kitchen. But even more surprisingly, people are completely unfazed when the object they are trying to use turns out to work differently from how they imagined it, even if it is as unintuitive as the cupboard shown in Fig. 1.2.

While we do not know how it works exactly, we can say that people, in general, have models of how their surroundings might move or be movable, which enable them to interact with objects to achieve some goal, like pressing the foot pedal of the trash can to open the lid, to dispose of a banana peel.

Models that allow these kinds of predictions are commonly called articulation models. Formally, these models describe the possible relative motion of, usually rigid, bodies, parameterized over a number of degrees of freedom (DoF). More colloquially, an articulation model of a door would describe how the door is hinged to its frame, how the handle is bolted to the door and that pushing the handle downwards will unlock the door.

In robotics today, such models are often hand-coded for the environment a robot is going to operate in. Unfortunately, it is easy to see how this approach does not scale. Every time a customer buys a robot to help them around the house, a highly skilled technician would have to come in and build a detailed model of the customer's house so the robot can actually do its job. And when something about the house changes, the technician would have to come back and document the change. Luckily, there has already been progress made on estimating articulation models of an unfamiliar environment [15, 31, 21], but so far there is no standardized framework for expressing these models which is easy to integrate into the various software modules of a robotic system and easy to expand.
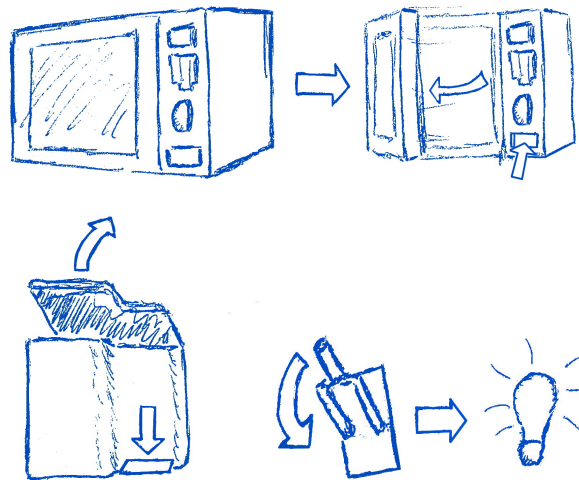
Figure 1.3　Mechanisms of household appliances that robots will need to interact with.

The current standards for modeling articulated objects are very descriptive formats, like URDF[1], SDF[2], COLLADA [2], MuJoCo [33] to name a few examples. They describe connections between objects in an abstract way, e. g. *the door is connected to the frame through a hinge*, and leave it up to every user to translate this description to their internal model. This makes the overall standards very rigid. If one wanted to add the currently uncovered models depicted in Fig. 1.3 to these standards, they would have to be added by hand to every library, inviting implementation errors and making it overall unlikely that these extensions would ever be supported universally.

However, when looking at what is expressed through the description it is very basic: A model defines forward kinematic expressions for frames which are parameterized over a number of variables which are, in turn, limited by constraints. Instead of wrapping this in a high-level description, which then gets expanded as new challenges arise, such a model could also be expressed directly as its mathematical expression. The advantage of a direct expression in terms of math is that mathematical operations are supported much more universally than abstract descriptions and are also interpreted unambiguously. Unless a new mathematical operator or function comes along, no model will ever be encountered which cannot be parsed.

Modern robotics systems are made up of many smaller runtime components which communicate amongst each other, organizing their individual capabilities into a big, coherent behavior. Software platforms like the popular *Robotics Operating System* (ROS) [24] have a lot of established standards for exchanging data between these many components. There are standard ways of exchanging information about locations and rotations of objects, standards for exchanging images and data from laser scans. But even though an articulation model is bound to change, as the

---

[1]URDF: *https://wiki.ros.org/urdf*
[2]SDF: *http://sdformat.org/spec*

robot interacts with an unfamiliar environment, there is so far no standard for exchanging these models at run-time. The lack of such a standard hinders progress in the robotics community. Promising methods are tried in small examples, but rarely updated or expanded to work for an entire problem space. This is not because they would not work in these spaces, it is just not feasible for most researchers to do so, because of the time commitment it currently requires.

1.2 Problem Formulation

In this thesis I set out to investigate the problem of building an unambiguous, scalable framework for describing articulation models on the basis of symbolic mathematical expressions.

The problem is relevant, as future robotic systems interacting with unfamiliar environments will need dynamic articulation models that can be modified and extended as they observe their environment. The existing frameworks popularly used in the robotics and computer graphics world, use a high level of abstraction for describing articulation models. This abstraction means that background knowledge is needed to interpret a model and to construct the underlying mathematical structures. On the one hand, this invites errors in every implementation parsing a model description, on the other hand, it requires a programmer to extend their individual implementation whenever the description language is updated. Especially the implementation overhead impedes the scaling of these frameworks when a new mode of articulation – a new type of connection – needs to be added to the description.

No such framework exists today because of one of the core requirements of these frameworks, which is, especially in the ROS context, the availability across multiple programming languages. The common solution – defining an abstract description language and implementing parsers for all programming languages – is the root cause of the lack of scalability we see in current frameworks. This could be less of an issue, if the descriptions contained all the data users needed for their computations and enabled them to read it blindly from the parsed results. However, it is neither clear, which data users need, nor what a useful representation of that data is.

Expressing articulation models directly as their underlying math is a promising solution to these problems. Wherever articulation models are used, be it for calculating the position of a robot's hand, or for tracking a person in a motion capture studio, they are converted into some representation of a mathematical expression for computation. Describing models directly through sets of expressions leads to unambiguous interpretations and it is unlikely that there will ever be a model which cannot be expressed as such a set of expressions. In this thesis I investigate whether describing articulation models as sets of expressions is actually feasible.

I view the challenge of scalability due to implementation overhead as solved if the new articulation framework enables me to write algorithms which are generic with respect to the articulation model

that they operate on. If this is possible, developers will not have to invest time into updating their implementations to accommodate new articulations that might be added to the framework's repertoire. This will enable the framework to grow and invite others to contribute new modes of articulation and use existing algorithms on them.

I think it is impossible – or at least infeasible for me - to show, that there will never be an articulation model which cannot be expressed in the new framework, since that would require one to know all the possible models that can exist. However, it can at least be shown that progress has been made on the modeling front. I view it as progress being made if it is possible to express articulation models in the new framework which could not be expressed before.

1.3   Contributions

In this thesis, I argue for a new approach of building articulation models based on mathematical expressions and propose a structured way of building an articulation model as a collection of such expressions. As a practical contribution, I introduce the ROS-package *Kineverse*, which contains an example implementation of the articulation model framework and functions for exchanging them in a ROS environment at run-time. I demonstrate the merit of the implementation with regards to the modeling challenge by building models of mechanisms which cannot be modeled in the existing frameworks. With regards to the implementation overhead challenge, I demonstrate the efficacy of the implementation by implementing algorithms for solving problems in articulated systems which are fully generic - not using any high-level knowledge about the models they operate on.

1.4 Structure of the Thesis

This section provides a brief overview of the individual chapters in this thesis. In general, the thesis is structured to be read sequentially.

Chapter 2 reviews the existing works on estimating articulation models and using them to generate motion. Finally, it gives an overview of three popular descriptions from the ROS and computer graphics environment for modeling articulated objects.

Chapter 3 introduces the concept of symbolic math and the formal description of the articulation model. It then presents the implemented software from the perspective of a user and gives insights into the implementation that are useful for developers. It closes by describing two additional practical methods that are used in the evaluation.

Chapter 4 demonstrates the efficacy of the implementation and framework in a number of example scenarios. Among these are the model of a robot, a simple configuration space tracker, a robot interacting with a kitchen and an algorithm generating a motion problem for unlocking a lockbox, as well as some other isolated examples.

Chapter 5 reviews the work presented in the thesis, reflects on the results acquired from the examples, and draws a conclusion with regards to the initial question.

Chapter 2

Related Work

2.1 Existing Descriptions of Articulation Models

In this section three existing description formats for articulation models are introduced. The first two are *URDF* and its quasi-successor *SDF* which are the standard description formats used in the ROS-ecosystem. The third is the *COLLADA* file format which is widely used in the space of 3D computer graphics.

2.1.1 URDF

The Unified Robot Description Format – short *URDF*[1] – is an XML based description of articulated entities which is native to the ROS ecosystem. Its primary purpose is to describe the structure of robots, however it is also commonly used to model other articulated structures like furniture or tools. The format's two core concepts are *Link* and *Joint* with links being the rigid bodies making up the robot and joints connecting them. The format requires the described structure to be a tree, disallowing kinematic loops. Links share their pose with the joint connecting them to their parent. The location of the center of mass and inertial frame are defined separately, relative to that pose. Fig. 2.1 shows the structure of a link in URDF.

Joints encode different degrees of freedom (DoF) between pairs of objects. They can be of six different types:

Fixed: A static connection between two links, defining their relative transformation to be constant.

Prismatic: A 1-DoF connection between links, allowing the child link to be translated freely along an axis that is defined relative to the parent's pose. Bounds for the translation can be specified.

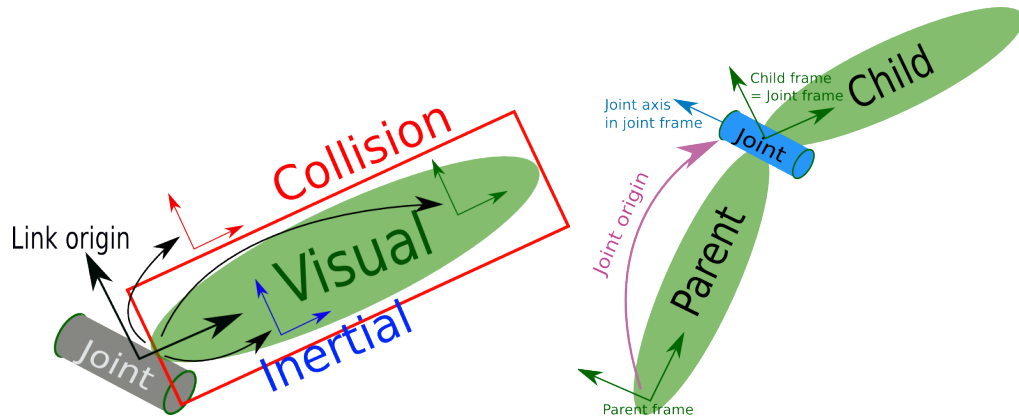[1]URDF: *https://wiki.ros.org/urdf*

Figure 2.1   Schematics of the structure of links and joints in URDF. Originally from: *https:// wiki.ros.org/urdf/XML/link*, *https://wiki.ros.org/urdf/XML/joint*

Continuous: A 1-DoF connection between links, allowing the child link to be rotated freely around an axis that is defined relative to the parent's pose.

Revolute: A bounded version of the *Continuous* connection type.

Planar: A 2-DoF connection between links, allowing the child link to be translated freely in a 2d-plane defined relative to the parent's pose.

Floating: A 6-DoF connection between links, allowing the child link to move freely relative to its parent's pose.

All movable joints can store information about their static friction and damping.

In URDF, joints cannot be connected to other joints, they can only be used to connect two links. To model a joint that is not one of the atomic joint types, it is common practice to use empty links, containing no geometry, to chain multiple atomic joints. This way joints like a planar connection with one degree of rotational freedom around the plane's normal can be created.

URDF offers a *mimic* rule with that the value of a joint's position can be defined as a function of another joint's position. The function is limited to being a linear mapping of the form $f(x) = mx + b$ with m scaling the referenced position x and b applying an offset.

Lastly, joints can store data relating to their calibration on a physical robot and so called *soft* limits that should be enforced by a motor controller.

2.1.2   SDF

SDF is an XML-based description format developed as a part of the Gazebo robot simulator[2]. It can be used to model individual objects, as well as entire simulated worlds, including lighting

---

[2]Gazebo simulator: *http://gazebosim.org/*

settings and physical forces like gravity and winds. For the purpose of this thesis, the following overview is limited to the features relating to articulation.

Articulated objects are referred to as *models*. Models define *Links*, *Joints*, and *Frames*. Frames are defined in relation to other frames, or their defining parent, and can thus be used to create static transformations without using fixed joints as helpers.

Different from URDF, links in SDF can specify their own poses relative to a frame or joint. They can additionally define a pose for their inertia tensor and center of mass.

Like URDF, SDF defines a set of atomic joint types[3]:

Fixed: A static connection between two links, defining their relative transformation to be constant. Analogous to URDF's fixed joint type.

Prismatic: A 1-DoF connection between links, allowing the child link to be translated freely along an axis which is defined relative to the parent's pose. Bounds for the translation can be specified. Analogous to URDF's prismatic joint type.

Revolute: A 1-DoF connection between links, allowing the child link to be rotated around an axis which is defined relative to the parent's pose. The rotation can be bounded, making this joint type a combination of URDF's continuous and revolute type.

Ball: A 3-DoF rotational joint, imitating a ball and socket connection between parent and child link.

Screw: A 1-DoF connection, coupling a prismatic joint with a rotation about the axis of translation.

Universal: A 2-DoF rotational joint, constraining one axis of rotation.

As in URDF, joints cannot be connected directly to other joints. For the types with higher degrees of freedom, a second axis is specified in the joint. The pose of a joint is determined in reference to the child link and not the parent. The pose of the joint relative to its parent is determined from the initial pose of the links in the model. Fig. 2.2 shows the difference in the models. In addition to static friction and damping, a spring stiffness and reference point can be set per axis.

2.1.3   COLLADA

COLLADA [2] is an open XML format for exchanging assets among 3D graphics applications and is widely supported in that field. Originally created at Sony Computer Entertainment, most of its functionality focuses on visual assets. It provides methods for describing articulated systems in terms of their kinematics and dynamics, and since version 1.4 also descriptions of rigid body physics and constraints. Collada is an extremely rich format, making it impossible to fully explore

---

[3]SDF version 1.6 defines additional types, but it is not clear what they do exactly as they are undocumented.
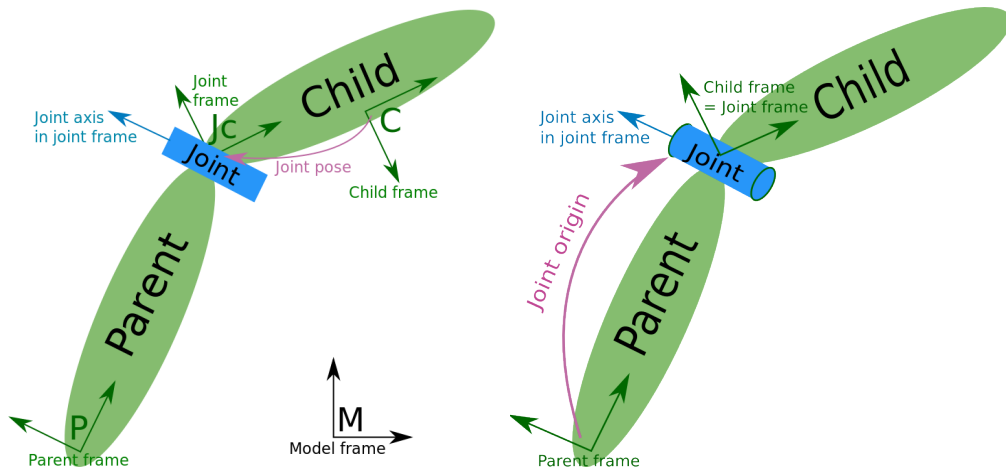
Figure 2.2 Differing joint models in SDF (left) and URDF (right). SDF schematic: *http://sdformat.org/tutorials?tut=spec_model_kinematics&cat=specification&* (Apache 2.0)

here, however, a few basics need introducing before looking at the descriptions of articulated systems in this format.

At its core, Collada separates between libraries and instances of data. As an analogy to object oriented programming: Libraries define classes and instances are objects of these classes. Each Collada file consists of a number of libraries, e. g. geometry, animations, articulated systems, etc. The data from these libraries is then instantiated within the context of a scene. Collada supports scoped and non-scoped identification of objects and allows for the definition and binding of parameters to either expressions, or other parameters, or constant values.

Like the previous models, Collada's kinematics model consists of links and joints with links, again, being rigid bodies and joints connecting links. Unlike the previous joint descriptions, Collada's description always models joints as compounded degrees of freedom, which are combinations of rotations and translations. Collada supports parameterized structures, with parameters that can be bound to constant values, other parameters or expressions specified in MathML [4]. These parameters can be used to recreate the mimic-relation of URDF joints, or SDF's *Screw* type. However, not all elements can be controlled this way. The axis specification of an individual degree of freedom, for example, does not support this mechanism. Collada's kinematic model affords the creation of closed-loop systems, which are impossible to create in the other two formats. The beginnings and ends of such a closed loop need to be marked explicitly as such and cannot be defined implicitly by the kinematic structure.

Collada defines a second joint model specifically for physics simulation which exists separately from the main articulation model. Instead of stacking degrees of freedom, it defines only one type

---

[4]MathML: *https://developer.mozilla.org/en-US/docs/Web/MathML*

of constraint which can connect one rigid body to another, or to a frame. Different joint types are modeled by limiting the degrees of freedom of a connection. The limits come in the form of bounds on the translation between two objects and bounds on the objects' rotation around the axes of the reference frame. This model is shared by the popular PhysX and Bullet physics libraries. It is easy to see how this formalism can be used to model all joint types defined in URDF and most joints defined in SDF, except for the *Screw* type, since this type requires a model of dependence between motions in different dimensions, e.g how a rotation around the screw axis always has to be matched by a proportional translation along the same axis.

To accommodate special functionality needed by its users, Collada provides the concept of *techniques*. Techniques provide additional information needed to interpret a file correctly. Aside from a set of common techniques which need to be implemented to conform with the Collada standard, users can define their own techniques, which others have to then implement in their system to load a given file correctly. In this way a technique can be used to communicate about a missing implementation, though it does not have to provide a description of what is actually missing.

2.1.4 Conclusion

There are multiple description frameworks in existence today, which can be used to express articulated systems. The aforementioned three take a descriptive approach to expressing the types of connections between two objects. URDF is the best supported and most widely used format in the ROS environment. While SDF extends the modeling capabilities of URDF, its use is mostly limited to the Gazebo robot simulator. In the world of ROS, Collada is mostly used as container for exchanging 3D geometry. It does have the ability to express articulated systems, and the inclusion of mathematical expressions gives more freedom to its users than the other two formats offer. However, excluding expressions from the axes of its joints also limits the kinds of models that can be expressed.

2.2 Practical Uses of Articulation Models

In this section I will give an overview of the different uses of articulation models in robotics today, to give an impression of typical use cases and algorithmic requirements for these cases. The examples are used in part to motivate the practical examples in Chapter 4. I group the usages of articulation models into the fields of perception and robotic manipulation.

2.2.1   Articulation Models in Perception

There are many tasks in the space of perception for robotics. There are more general computer vision tasks like recognizing people and individual objects and there exist more specialized tasks which are specifically relevant for robotic manipulation. One of these problems is estimating the pose of an observed object. While much object manipulation work in robotics today is working with loose items, there are cases in which articulated objects, typically doors and drawers need to be manipulated. The interaction requires the poses of the objects to be estimated before they can be manipulated. Articulation models can help with solving these problems, as they restrict the plausible poses of objects.

Using articulation models to track the motions of multiple connected bodies has a long history in the field of human pose and motion tracking. Using some articulation model of the limbs, helps to reduce the search space of a tracker down from more than 50 degrees of freedom to 32 or less, making the problem more tractable.

In recent times, not much emphasis has been placed on exploiting articulation models to locate objects in the field of perception for robotic manipulation. Older works like Drummond *et al.* [8] and Comport *et al.* [6] use articulation models in combination with non-linear optimization techniques to track articulated objects. More recent work by Desingh *et al.* [7] models the task of extracting the configuration of articulated objects as a probabilistic graph problem, similar to previous work on human motion tracking by Sigal *et al.* [29]. The nodes in their graph model the links of the object, while the edges model the object's joints. The joints are used to define potential functions that rate the likelihood of two object poses given the type of the connection. The potential functions are used to filter implausible object poses, improving the overall estimate.

The main focus in the field of robotic perception has been on estimating an articulation model from, mostly visual, observations. As Katz *et al.* [15] point out in 2008, previous methods in the areas of robotic manipulation and perception assume an articulation model as given. In this early work they, estimate the articulation of rigid bodies on a plane from a 2D-image stream. They do so by clustering image features into rigid structures and then matching the relative trajectories of these structures to either a revolving, or prismatic connection.

Sturm *et al.* [31] present a comprehensive formulation for the problem of estimating an articulated structure consisting of multiple bodies. Their approach solves both the problem of determining the kinematic structure as a graph $\hat{G}$, as well as the type of the individual connections. They do so by finding a connection model $\mathcal{M}$ with a parameterization θ to explain a series $\mathcal{D}_{ij}$ of observed relative poses of objects i and j, as in Eq. 2.1. The connection models $\mathcal{M}$ are either selected from a predetermined set of models, which includes rigid, prismatic, or rotational connections, or by a model that is learned from observations.

$$\hat{\mathcal{M}}_{ij}, \hat{\theta}_{ij} = \underset{\mathcal{M}_{ij}, \theta_{ij}}{\arg\max}\, p\left(\mathcal{D}_{ij} \mid \mathcal{M}_{ij}, \theta_{ij}\right) \tag{2.1}$$

The probability of a model is assessed using an inlier function which uses inverse kinematic projection $f^{-1} : SE(3) \rightarrow \mathbf{q}$, projecting a pose into a connection's configuration space. The projected configuration is used to compute the forward kinematic transformation, which is compared to the observed transformation, to judge the model's accuracy. Once the most likely pair-wise connection is found, the most likely kinematic tree can be found by maximizing the joint probability of all edges in the tree. As an extension to their model, they introduce a method for determining the number of degrees of freedom of an observed model, which enables them to detect closed kinematic chains.

While Katz *et al.* do not explicitly state all the details of their method, it does stand to reason, that they, like Sturm *et al.* , require an inverse kinematics function with which they can project an observation into the configuration space.

Both of the described approaches rely on observing motions to estimate the articulation model and cannot act before they do so. Katz *et al.* are able to refine their articulation model by interacting with the object as they demonstrate in [16], but do rely on observations to construct the initial model.

Jain *et al.* have investigated the robotic opening of unfamiliar doors and drawers using *Equilibrium Point Control* [14, 13]. They use a simple Cartesian space controller and a hypothesis of the underlying articulation model to operate doors and drawers using a robot. Incrementally, they calculate new equilibrium point – a point where the endeffector (e. g. gripper of the robot) would settle if no external force is applied – and use the Cartesian controller to move the endeffector to the point. They record the Cartesian path of the endeffector and fit a rotational joint to that path.

Sturm *et al.* utilize Jain *et al.*'s equilibrium point control approach to manipulate objects [30]. They start the interaction by manually specifying a grasp point and endeffector orientation in a 3D point cloud and giving a direction for an initial motion. As the motion is executed, the manipulated object is observed and the observations are used to update the articulation model. Rühr *et al.* [27] present a generalized framework for operating doors and drawers in a kitchen environment. They combine the estimation framework by Sturm *et al.* and impedance control with a detector for handles. Using theses systems, their robot is able to grasp handles, pull them towards itself, and estimate an articulation model from the observation. Once the model has been estimated, the framework stores that information in a semantic map.

Martín *et al.* [19] present an online algorithm for estimating articulation models and tracking object poses at the same time. While Sturm *et al.* [31] required markers to detect their objects, Martín *et al.* use multiple recursive Bayesian filters to cluster features into objects, determine

these objects' motions and estimate the articulation model from this estimate. Like in the previous approaches, the articulation model is estimated on the basis of relative poses of object pairs. Per object pair and supported connection type (*fixed*, *prismatic*, *revolute*), one recursive Bayesian filter is maintained. If none of the filters are consistent with the observed motion, the object pair is assumed to be disconnected. Their articulation model supports closed kinematic chains, as it does not build a complete kinematic graph, unlike the previous approaches. In a number of experiments, Martín *et al.* demonstrate their approach's ability to accurately predict and track the articulation model of books, doors, drawers, a globe and a human head. As a limitation of their system, Martín *et al.* point out that it still needs to observe an initial motion, before it can act. Later, they combine this visual online algorithm with a compliant control scheme to use both visual and proprioceptive observations to estimate single articulations interactively [20].

In the works showcased so far, interaction strategies for articulated objects are quite limited. Höfer *et al.* [12] focus on extracting *kinematic background knowledge* from interactions. Their system generates rules which predict what kind of articulation will be observed given an interaction and a set of object properties. For example: If X is a handle which is attached to object Y, which is attached to frame Z, then pulling X away from Y is likely to reveal that Y and Z are connected with a revolving joint. By learning these rules, Höfer *et al.* implicitly generate a prior on the likely articulation model of an object and can use this information to interact with the object in a way which will help them gather data on their hypothesis. In this initial work, the actions that can be chosen are atomic interactions like pushing towards an object or pulling away from it, or rotating it in a particular direction. In a follow-up work [11] this set is simplified to be either a push, or a pull with a vector denoting the direction of the applied force. In both works, the experiments are carried out in a simulation environment, in which the actions can be carried out directly on the objects in question. Initially introduced in [12], the model cannot only express rigid, prismatic, or revolving connections, but also conditional degrees of freedom e. g. the locking relationship between doors and door handles.

Works like the ones by Höfer *et al.* are focused on determining the complete articulation model of a scene. Strategies for examining larger articulated structures were first proposed by Otte *et al.* [23] who formulate a belief over object types, joint types and limits, and dynamics properties and then propose different action strategies for interacting with objects in the environment. Similar to Höfer *et al.*, their interactions are limited to being a simple push, and the joints can be revolving or prismatic. The effect of the action is observed as 3D object trajectory and integrated into the belief state using the method by Sturm *et al.* [31]. The two best performing methods select an interaction based on either the expected decrease in belief entropy, or the entropy in the belief about an individual object.

Hausman *et al.* [10] focus explicitly on selecting actions to reduce the posterior of their belief over an individual articulation. They model their articulations similar to Sturm *et al.* and use their approach to integrate observations into their belief over the articulation model. For their

interactions, Hausman *et al.* sample a number of directions in which their robot could push or pull an object. For each direction they asses the potential change in the posterior and select an action according to this metric.

Recently the interest in structures with conditional degrees of freedom has increased, e. g. systems like doors which are locked by a handle. Kulick *et al.* [17] explore the space of conditional joint structure estimation by formulating a probabilistic model of dependency structures and proposing a strategy to explore them efficiently. They assume the articulation model to be given and focus only on estimating the dependency structure. In a later work by Baum *et al.* [3] this assumption is relaxed to only knowing about movable parts, but not their individual articulation.

Conclusion

In this section I have presented an overview of methods that perceive articulation models in one way or the other. Some, like [6, 8, 29, 7] exploit the spatial restrictions created by these models to solve high-dimensional problems efficiently. Others, like [15, 14, 31, 27, 12, 19, 23, 10] focus on estimating articulation models from observations. Independent of visual observations, e. g. [15, 31, 12, 19], or proprioceptive feedback [14, 27, 20], all of these approaches rely on the existence of an inlier function, which can rate the compliance of different articulations with the made observations. This inlier typically uses an inverse kinematic function and a projection function, that can project a relative pose of two objects into the model's configuration space and then determine the model's relative pose based on this projection. With the exception of [31, 12, 17, 3], all introduced methods are evaluated exclusively on static, revolving, and prismatic connections. Sturm *et al.* [31] introduced a learnable connection, but that type is not present in the other works, despite the wide spread use of their method. The models in [12, 17, 3] add conditional degrees of freedom to the common connection set.

All in all, there seem to be robust methods for estimating the articulation model of a set of observed objects. There is a chance these methods can be generalized to work for arbitrary articulation models, as long as there is a way of solving the inverse kinematics of the model.

2.2.2   Articulation Models in Motion Generation

Though every robotic interaction with a door or drawer is a manipulation of an articulated objects, there is not much work in this field that uses more general articulation models to generate motion. In this section I will collect a number of the existing works, and then look at some motion generation methods and how they could relate to generating interactions with articulated objects.

To interact with an articulated object, sampling based trajectory planning methods tend to sample a number of endeffector poses for the gripper of the robot, given an established grasp. They then use an inverse kinematics solver to generate a configuration space trajectory for operating the object. This approach is used in [4, 5, 27]. Commonly, when the state space of the system increases and the constraint manifolds become more narrow, sampling based approaches tend to partition the problem. Chitta *et al.* [5] use a PR2 robot to operate a door while utilizing both its omnidirectional base and arm. They partition the problem into finding locations for the base using a graph structure and configurations for the arm in those locations to solve the task. In a more recent work, Burget *et al.* [4] use a bipedal robot, the Aldebaran Nao IV, to operate a cabinet door and a drawer. The robot's 21 DoF and stability requirements prevent them from directly sampling the configuration space of the robot, since most samples would not satisfy these constraints. To get around this problem, they part the problem by generating a set of stable poses and then only generating a trajectory for the 5-DoF arm. While sampling algorithms perform quite well in high-dimensional configuration spaces, they do rely on a successful sampling strategy for the configuration space trajectory. The narrowness of the constraint manifold created by stability, constraints and articulation constraints make sampling largely unsuccessful. In general, the trajectory planning methods require an existing model of an object's articulation.

When lacking an articulation model, a control approach is needed to interact carefully with objects. A very simple, but surprisingly successful method is the equilibrium point control by Jain *et al.* [13], as mentioned in the previous section. They build on the Equilibrium Point Hypothesis, which posits motion to be controlled by an ever shifting Equilibrium Point, a point where an endeffector would settle in the absence of external forces. In line with this hypothesis, they incrementally generate a new Cartesian Equilibrium Point (CEP) at every control cycle, find a matching point in joint/configuration space and command their robot to move to that configuration. As it moves, the position of the endeffector is monitored and its trajectory matched to one either a revolute, or a prismatic joint, informing the generation of the next CEP. As pointed out before, this method of control is used by Rühr *et al.* [27] for the initial interaction with an object when the articulation model is not yet know. After estimating the model, they rely on planning methods for further interactions.

A commonality of both of these strategies is the use of a forward-model mapping a joint position to a pose in Cartesian space, which is then either used as next CEP, or as an endeffector trajectory in combination with other poses. In a control approach like CEP, the forward model is not strictly necessary, as the motion could also be generated by using velocity control where the endeffector velocity $\dot{\mathbf{x}}$ is generated as $\dot{\mathbf{x}} = \mathbf{J}(q)\dot{q}$ with $q \in \mathbb{R}$ being the articulated joint's position and $\mathbf{J}$ being the jacobian of the grasped point. This formulation is utilized by Martín *et al.* [20]. The Jacobian can also be used to compute an endeffector trajectory by integrating $\dot{\mathbf{x}}$.

While sampling based methods tend to struggle with tightly constrained motion problems, optimization based trajectory planning methods have turned out to be quite successful in planning

motions under a multitude of constraints. Although on paper optimization based methods should lend themselves to manipulating articulated objects, there seem to be no works explicitly tackling this problem. In the following I will give a brief overview of existing optimization based motion generation methods, since they should work well for manipulating articulated objects and since I use one of them in the evaluation chapter of this thesis.

Ratliff *et al.* present *CHOMP* [26], the first motion planner in this family of methods. They optimize the cost $\mathcal{U}$ of a trajectory of n configuration space points. The cost term is composed of a cost for closeness to obstacles and a cost for trajectory smoothness. They use the gradient of this cost function to formulate a simple update rule for their trajectory, satisfying both smoothness and obstacle avoidance at the same time. In sampling based planners trajectory smoothing is typically a post-processing step. Burget *et al.* [4] point out that this is problematic when manipulating articulated objects, as the smoothing does not maintain the desired endeffector trajectory. One of the core problems of the optimization approach is that it requires the cost and constraint functions to be smooth and differentiable. This is not the case for object collisions, as they are binary events. Ratliff *et al.* work around this problem, by pre-computing a singed distance field for their environments which they can then query during the optimization. Schulman *et al.* [28] present *TrajOpt*, a successor to CHOMP and do a quantitative comparison with existing sampling based planners. While CHOMP uses an unconstrained gradient descent and models all constraints as high costs, TrajOpt formulates motion planning as non-convex constrained optimization problem which it solves using convex approximation and quadratic programming. Schulman *et al.* incorporate collision avoidance by computing minimal translations between object pairs and using these as penalties. They also include a formulation for continuous time collisions, which is based on swept volumes. Their method uses the collision checking functionality from the *Bullet*[5] physics engine. Their comparison with sampling based methods from *OMPL* [32] and *MoveIt!*[6] shows their method to be applicable to a wide variety of highly constrained problems with many degrees of freedom (up to 28), while being at least twice as fast as the competing methods.

The querying of an external collision detection library, as done by *TrajOpt*, yields only a poor approximation of the geometry in the environment, since the query only returns a direction and a signed distance. Ratliff *et al.* [25] try to solve this problem by viewing obstacles as a part of the problem domain and not as gaps in it. To do so, they discuss the relationship of gradient based trajectory optimization and Riemannian manifolds and propose a cost function which integrates obstacles into the gradient, naturally making the generated trajectory collision free. They demonstrate that their method safely navigates around thin obstacles, which is a problem in query based approaches. However, the method relies on local coordinate spaces which follow obstacles' surfaces, which are easy to find for geometric primitives, but hard to find for arbitrary shapes.

---

[5]Bullet physics engine: *https://pyBullet.org/wordpress/*
[6]MoveIt: *https://moveit.ros.org/*

Most works using optimization methods for motion generation seem to focus on pick and place item manipulation and obstacle avoidance. The work most closely related to using articulation models in this field, seems to be the work done by Toussaint on *Linear-Geometric-Programming* (LGP). Toussaint first introduces the concept in [34] as a method for integrated task- and motion planning (TAMP). LGP optimizes trajectories over switching kinematic conditions, producing a feasible solution for a complex high-level goal. In the specific publication, the goal is given as a function rating height and stability of a tower constructed from pillars and planes. Planning the individual motions, requires the ability to model varying kinematic conditions affecting the trajectory. The kinematic conditions are modeled as additional constraints on the underlying optimization problem, e.g. a *grasp* is defined by equality constraints, forcing endeffector and object pose to be equal. In a later work [35] the LGP approach is expanded to include a dynamics model and more kinematic conditions are added. However, these recent additions venture more into the space of inverse or goal directed dynamics, than they do into articulated object interactions.

As a final work in the space of optimization based motion generation, I would like to point out *Giskard*, an optimal control based approach. The approach is used to generate motions in Chapter 4 of this thesis.

In its first introduction [9], Giskard is used to learn pouring motions from demonstrations. The task is described as constraint-based motion control problem, which uses a simplified version of the task specification language introduced by Aertbeliën *et al.* [1]. Using this specification language, a task can be expressed as a set of hard and soft constraints. The control problem is given as a function $f : \mathbb{R}^n \to \mathbb{R}^m$ which computes joint velocities $\dot{\mathbf{q}}_{des}$ given a vector of observed variables $\mathbf{o}$:

$$\dot{\mathbf{q}}_{des} = f(\mathbf{o}) \tag{2.2}$$

It is assumed, that the joint positions $\mathbf{q}$ are a subset of the observed variables $\mathbf{o}$. The constraints of the task are applied to the following optimization problem that lies at the heart of f:

$$\begin{aligned} \min_{\mathbf{s}} \quad & \mathbf{s}^T \mathbf{H} \mathbf{s} \\ \text{s.t.} \quad & \mathbf{l}_A \leq \mathbf{A}\mathbf{s} \leq \mathbf{u}_A, \\ & \mathbf{l} \leq \mathbf{s} \quad \leq \mathbf{u} \end{aligned} \tag{2.3}$$

With $\mathbf{H}$ being a diagonal weighting matrix and $\mathbf{A}$ holding the Jacobian of the constrained expressions. The vectors $\mathbf{l}_A, \mathbf{u}_A$ hold the lower and upper bounds of the constraints, $\mathbf{l}, \mathbf{u}$ directly constraining the vector $\mathbf{s}$. The soft constraints of the task are encoded by extending $\mathbf{s}$ to include a set of slack variables $\epsilon \in \mathbb{R}^p$ for p soft constraints. Using these variables to satisfy the problem is penalized using the weighting matrix $\mathbf{H}$. The slack variables are incorporated into the task

optimization by adding an identity mapping to $\mathbf{A}$: $\mathbf{A} = \begin{bmatrix} \mathbf{J} & \mathbf{I}_{p \times p} \end{bmatrix}$. All parts of this problem are functions of the vector of observed variables $\mathbf{o}$ and thus are recomputed at every control cycle.

Conclusion

In this section I have given a brief overview over a few methods for generating robotic motion with a special focus on manipulating articulated objects. Sampling based methods rely primarily on projections from an object's configuration space to a pose in Cartesian space, for which they can then solve an inverse kinematics problem. Optimization based methods rely primarily on the availability of differential information for their task space, to generate motions. While most of the showcased control methods use a projection to Cartesian space to define the next goal for a positional controller, the same behavior could be achieved by using the gradient of the articulation to locate a next point in space. All in all, it should be possible to generalize these methods to arbitrary articulation models, as long as the models define a forward kinematic or a gradient with respect to their configuration variables.

Chapter 3

Methods

In this chapter I introduce the implemented *Kineverse* framework. I do so from two different perspectives. The first perspective is that of a user who wants to use Kineverse to create and use articulation models. The second perspective is that of a developer, who wants to understand and extend the framework.

Before the individual perspectives can be explored, an understanding of Kinverse's backbone – the symbolic math framework – is needed.

3.1 The Backbone: Symbolic Math

3.1.1 Handling Symbolic Expressions

All articulation models in Kineverse are assembled from symbolic, mathematical expressions. These expressions can contain free variables that can be substituted for numerical values, to model function evaluation for a set of parameters.

In this thesis, expressions are denoted by greek letters φ, ψ and free variables are denoted by latin letters a, b, c. To illustrate:

$$\varphi = a + \sin b$$
$$\psi = b \cdot c \tag{3.1}$$

In the example above φ, ψ are two expressions with two free variables each. The sets of free variables are denoted by $V(\alpha)$. In the case of φ the set of free variables is $V(\varphi) = \{a, b\}$, in the case of ψ it is $V(\psi) = \{b, c\}$. The free variables in φ, ψ can be substituted for other expressions[1]

---

[1]Free variables are also expressions.

or constant values.

In the space of robotics, the position of all the robot's joints would be denoted by a vector $\mathbf{q} \in \mathbb{R}^n$. In this thesis, I keep to this notation, even though $\mathbf{q}$ is understood more accurately as a mapping, which maps elements from the set of all variables $\mathbb{V}$ to real numbers $\mathbb{R}$, i. e. $\mathbf{q} : \mathbb{V}^n \to \mathbb{R}^n$. Assume a given $\mathbf{q}_1$, which maps $b \to 4$, this $\mathbf{q}_1$ can be applied to the two expressions:

$$\varphi(\mathbf{q}_1) = a + \sin 4$$
$$\psi(\mathbf{q}_1) = 4 \cdot c$$

$$(3.2)$$

Applying $\mathbf{q}_1$ to the two expressions yields two new expressions $\varphi(\mathbf{q}_1), \psi(\mathbf{q}_1)$, both of which have one free variable $V(\varphi(\mathbf{q}_1)) = \{a\}, V(\psi(\mathbf{q}_1)) = \{c\}$. Applying a second mapping $\mathbf{q}_2 = \{a \to 2, c \to 3\}$ to both expressions, yields

$$\varphi(\mathbf{q}_1)(\mathbf{q}_2) = 2 + \sin 4$$
$$\psi(\mathbf{q}_1)(\mathbf{q}_2) = 4 \cdot 3$$

$$(3.3)$$

Now both expressions have no more free variables and are thus constant. When $V(\varphi) = \emptyset$, then I call φ *grounded*. To make it explicit, that an expression is grounded, I mark it by an underscore e. g. $\underline{\varphi(\mathbf{q}_1)(\mathbf{q}_2)}$.

Note, that the order in which different $\mathbf{q}$ substitutions are applied matters. If $\mathbf{q}_1 = \{b \to 4\}$ and $\mathbf{q}_2 = \{a \to 2, b \to 5\}$, then $\varphi(\mathbf{q}_1)(\mathbf{q}_2) \neq \varphi(\mathbf{q}_2)(\mathbf{q}_1)$.

3.1.2 Variable Typing

Articulation models describe relative motions of bodies as a function of a set of parameters, which model the state of the degrees of freedom (DoF) of the model. Given a robot arm with five joints, the model will express the position of the arms' rigid parts - its links - as a function of five variables $q_1, \ldots, q_5$. In addition to this function, the model might also include limits for the value range of these variables, since the joints might have a fixed positional range. All of this can be encoded using the expressions that I introduced in the previous section. However, a robot does not only have positional limits, but usually also limits for the velocity with which joints can change their positions, and also further limits constraining joint accelerations and effort. To express these different aspects of a joint, I add the notion of variable types to the logic of Kineverse. Types do not change how the expressions behave, but they are used to interpret the expressions. In this thesis a can refer to the DoF/joint a as a whole or to its position. The other types are noted as in Eq. 3.4.[2]

---

[2]The actual Kineverse implementation also defines *snap*, but it is never used in practice.

$$
\begin{aligned}
a &:= \text{position of a} \\
\dot{a} &:= \text{velocity of a} \\
\ddot{a} &:= \text{acceleration of a} \\
\dddot{a} &:= \text{jerk of a}
\end{aligned}
\tag{3.4}
$$

These types are used in two different ways. First, they are used to define constraints on different aspects of a DoF, like position, velocity, and acceleration constraints. Second, they are used when creating derivatives of expressions, which will be covered in more detail in the next section.

For this thesis, it is important to note that the $\dot{\varphi}$ does not enforce type homogeneity. An expression φ can contain variables of arbitrary types, but it is important that its velocity expression $\dot{\varphi}$ then changes all variable types accordingly. An example:

$$
\begin{aligned}
\varphi &= a - 2\dot{b} + c^2 \\
\dot{\varphi} &= \dot{a} - 2\ddot{b} + 2c\dot{c} \\
\ddot{\varphi} &= \ddot{a} - 2\dddot{b} + 2c\ddot{c}
\end{aligned}
\tag{3.5}
$$

3.1.3   Augmented Gradients

A feature of symbolic math frameworks is their ability to differentiate expressions with respect to the variables contained in them. This is useful for methods like trajectory optimization or optimal control since they rely on the availability of gradients for problem solving.

The gradient $\nabla\varphi$ of expression φ is a column vector of derivatives towards the parameters of φ. Given φ with $V(\varphi) = \{q_1, \ldots, q_n\}$, its gradient is

$$
\nabla\varphi = \begin{bmatrix} \frac{\partial\varphi}{\partial q_1} \\ \vdots \\ \frac{\partial\varphi}{\partial q_n} \end{bmatrix}
\tag{3.6}
$$

However, when doing a normal differentiation, the information about with respect to which variable was differentiated gets discarded:

$$
\begin{aligned}
\varphi &= a \\
\frac{\partial\varphi}{\partial a} &= 1
\end{aligned}
\tag{3.7}
$$

Given Kineverse's variable typing, this becomes a problem, as the type information gets erased. To prevent this, I introduce a dictionary-like data structure for storing the gradient of an expression φ. The structure is called *augmented gradient* and can, again, best be understood as a mapping. Given an expression φ with $V(\varphi) = \{a, b, c\}$, the augmented gradient $\nabla\varphi$ is

$$\nabla\varphi = \begin{matrix} \dot{a} & \dot{b} & \dot{c} \\ \left[\begin{matrix} \frac{\partial\varphi}{\partial a} & \frac{\partial\varphi}{\partial b} & \frac{\partial\varphi}{\partial c} \end{matrix}\right] \end{matrix} \tag{3.8}$$

Note, the representation as a matrix is an aesthetic choice, since the actual mapping of variables to derivative expressions does not have a structure. The set of variables for which derivative expressions are defined is denoted by $K(\nabla\varphi)$. This set is different from $V(\nabla\varphi)$, which contains all variables of all derivatives contained in the gradient. Formally:

$$V(\nabla\varphi) = \bigcup_{x}^{x \in V(\varphi)} V\left(\frac{\partial\varphi}{\partial x}\right) \tag{3.9}$$

The set $K(\nabla\varphi)$ is a superset of the set of first-order derivatives of the free variables of φ, i.e. $\{\dot{x} \mid x \in V(\varphi)\} \subseteq K(\nabla\varphi)$. The sets not being equal has some utility for modeling DoF that cannot define a forward kinematic expression. An example of such a DoF follows shortly.

The fact that these augmented gradients store an association of variables and expressions is used for different purposes. The first is the generation of velocity expressions. Given φ and its augmented gradient $\nabla\varphi$

$$\nabla\varphi = \begin{matrix} \dot{a} & \dot{b} & \dot{c} \\ \left[\begin{matrix} 2 & 3 & 4 \end{matrix}\right] \end{matrix} \tag{3.10}$$

then $\nabla^{\dot{a}}\varphi$ denotes the expression that the gradient associates with $\dot{a}$, i.e. $\nabla^{\dot{a}}\varphi = 2$ in the given example. The velocity expression $\dot{\varphi}$ can be created by multiplying every element in $K(\nabla\varphi)$ with the expression it is associated with:

$$\dot{\varphi} = \sum_{x}^{x \in K(\nabla\varphi)} x\nabla^{x}\varphi$$

$$\dot{\varphi} = 2\dot{a} + 3\dot{b} + 4\dot{c} \tag{3.11}$$

This process motivates the second use of these gradients, to express aspects of a degree of freedom, which do not define a forward kinematic relationship, i.e. direct correspondences on a

positional level. A common way of driving a robot around a space is using a *differential drive*. A differential drive consists of two individually actuated wheels that are mounted along one axis. The mechanism is simple but effective: To move in a straight line, both wheels are turned in the same direction at the same speed. To turn, one wheel is turned faster than the other. While the relationships of wheel velocity and linear and angular robot velocity can be expressed easily, there is no mapping from wheel positions to a robot location in the world. The augmented gradients can be used to include this relationship in an articulation model. Given the equation for differential drives in chapter 13 of *Planning Algorithms* [18], let $\dot{u}_l, \dot{u}_r$ be the velocities of the left and right wheel and L, r the distance between them and their radius. Let x, y, θ be the robot's current linear and angular position and $\mathbf{p} = \begin{bmatrix} x, y, \theta \end{bmatrix}^T$ the complete position of the robot. Then the gradient $\nabla \mathbf{p}$ can be extended to include additional expressions for $\dot{u}_l, \dot{u}_r$:

$$\nabla \mathbf{p} = \begin{matrix} \dot{x} & \dot{y} & \dot{\theta} & \dot{u}_l & \dot{u}_r \\ \begin{bmatrix} 1 & & & \frac{r}{2}\cos\theta & \frac{r}{2}\cos\theta \\ & 1 & & \frac{r}{2}\sin\theta & \frac{r}{2}\sin\theta \\ & & 1 & -\frac{r}{L} & \frac{r}{L} \end{bmatrix} \end{matrix} \tag{3.12}$$

Given that it is possible to add expressions to the gradient for variables that are not included in the original expression, it is important to point out that these expressions do propagate to other gradients. When two gradients $\nabla\varphi, \nabla\psi$

$$\nabla\varphi = \begin{matrix} \dot{a} & \dot{b} \\ \begin{bmatrix} x & y \end{bmatrix} \end{matrix}, \nabla\psi = \begin{matrix} \dot{b} & \dot{c} \\ \begin{bmatrix} u & z \end{bmatrix} \end{matrix} \tag{3.13}$$

are combined using mathematical operators, their stored expressions are combined to recreate the effect the operator would have on regular derivatives, as in:
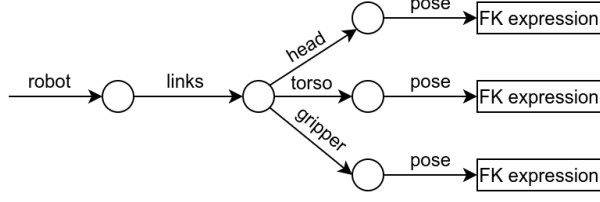
Figure 3.1 Example of three robot links stored within Kineverse's data tree.

$$
\nabla\varphi + \nabla\psi = \begin{bmatrix} \overset{\dot a}{x} & \overset{\dot b}{y+u} & \overset{\dot c}{z} \end{bmatrix} \qquad \nabla\varphi - \nabla\psi = \begin{bmatrix} \overset{\dot a}{x} & \overset{\dot b}{y+u} & \overset{\dot c}{z} \end{bmatrix}
$$

$$
\nabla\varphi \cdot \nabla\psi = \begin{bmatrix} \overset{\dot a}{\psi x} & \overset{\dot b}{\psi y + \varphi u} & \overset{\dot c}{\varphi z} \end{bmatrix} \qquad \frac{\nabla\varphi}{\nabla\psi} = \begin{bmatrix} \overset{\dot a}{\frac{\psi x}{\psi^2}} & \overset{\dot b}{\frac{\psi y - \varphi u}{\psi^2}} & \overset{\dot c}{\frac{\varphi z}{\psi^2}} \end{bmatrix} \tag{3.14}
$$

$$
\nabla\varphi \cdot v = \begin{bmatrix} \overset{\dot a}{v \cdot x} & \overset{\dot b}{v \cdot y} \end{bmatrix}
$$

This list is not complete. In Kineverse the differentiation rules for all common functions and operations are implemented.

3.2 The Articulation Model

Independent of the kind of user, whether they want to just use Kineverse or extend it, they must understand how articulation models are represented and understood in Kineverse. In this section, I describe the articulation model and how it is interpreted.

An articulation model, denoted as $\mathcal{A}$ in this thesis, is a tuple of a data tree $\mathcal{D}$ and a set of constraints $\mathcal{C}$, i. e. $\mathcal{A} = (\mathcal{D}, \mathcal{C})$. The data tree stores the data, e. g. expressions, of the articulation model at its leaf nodes. The edges of the tree are named, so that each piece of data is uniquely identified by the path leading to its node. Fig. 3.1 shows an example of three robot links stored in a data tree.

The set of constraints $\mathcal{C}$ encodes the configuration limits of the expressions stored in $\mathcal{D}$. Each constraint is a three-tuple $c = (\iota_c, \upsilon_c, \varphi_c)$, which expresses the two inequality constraints $\iota_c \leq \varphi_c$ and $\varphi_c \leq \upsilon_c$. The tuple can be used to express both inequality and equality constraints. An
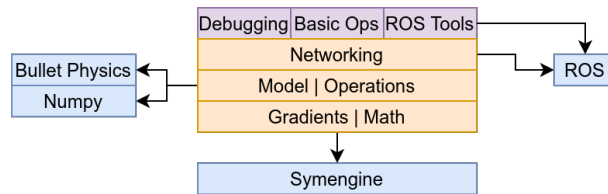
Figure 3.2 Visual representation of Kineverse's layered structure.

equality constraint $\varphi = \psi$ can be expressed as $c = (\psi, \psi, \varphi)$, an inequality constraint $\varphi \geq \psi$ can be expressed as $c = (\psi, \infty, \varphi)$.

The constraints become relevant, when a configuration $\mathbf{q}$ is supposed to be applied to the model $\mathcal{A}$. When a configuration $\mathbf{q}$ is applied to $\mathcal{A}$, i.e. $\mathcal{A}(\mathbf{q}) = (\mathcal{D}(\mathbf{q}), \mathcal{C}(\mathbf{q}))$, the resulting model is valid if all constraint bounds are satisfied, unless they have free variables. This requirement is expressed formally in Eq. 3.15.

$$
\begin{aligned}
\forall c \in \mathcal{C}(\mathbf{q}) : & (V\left(\iota_c(\mathbf{q})\right) \cup V\left(\upsilon_c(\mathbf{q})\right) \neq \emptyset \vee \iota_c(\mathbf{q}) \leq \upsilon_c(\mathbf{q})) \wedge \\
& (V\left(\iota_c(\mathbf{q})\right) \cup V\left(\varphi_c(\mathbf{q})\right) \neq \emptyset \vee \iota_c(\mathbf{q}) \leq \varphi_c(\mathbf{q})) \wedge \\
& (V\left(\upsilon_c(\mathbf{q})\right) \cup V\left(\varphi_c(\mathbf{q})\right) \neq \emptyset \vee \varphi_c(\mathbf{q}) \leq \upsilon_c(\mathbf{q}))
\end{aligned}
\tag{3.15}
$$

3.3 System Overview

With the understanding of the conceptual basis of Kineverse - its symbolic math system and its articulation model - I will now give an overview of the structure of Kineverse, before illustrating how a user might integrate Kineverse into their project.

The Kineverse framework is structured into multiple layers that build on each other.[3]

Gradients, Math: The lowest layer of Kineverse is the *Gradient* layer. It implements the fundamental logic to Kineverse's math system, which is used in all other layers.

Model, Operations: The second lowest layer is the model and operations layer. The model stores expressions and constraints which form a complete articulation model. Models can be built up through the sequential execution of operations that read data from the model and apply changes to it.

Networking: The third layer implements the exchange of Kineverse models within a ROS environment. It contains a model server, model clients and serialization functionality.

---

[3]Unofficially there is a *motion* module, which contains a rudimentary, QP-programming-based solver for motion generation that was forked from GiskardPy (*https://github.com/SemRoCo/giskardpy*).

	Local	Networked
Static	`KinematicModel`	`ModelClient`
Dynamic	`Operation`	`OperationsClient`

Table 3.1 Two main dimensions for use cases and the classes that are most relevant for the user.

Tools, Basic Operations: The fourth and last layer implements a variety of tools for visualizing and debugging of models and for integrating them in ROS environments. It also implements operations for creating common kinematics and for loading articulation models from URDF.

Kineverse uses a number of external libraries.

Symengine: The Symengine library is a computational algebra system (CAS) which provides a vast set of symbolic algebraic data structures and operations on them. It is used as the mathematical backbone for Kineverse.

Bullet Physics: The Bullet Physics Engine[4] is a library for simulating rigid- and soft body physics. It comes with an integrated collision detection library, which is used for geometric queries by Kineverse.

NumPy: NumPy is one of the most popular math libraries for the Python programming language. It adds array data types and fast mathematical operations on them. In Kineverse it is mostly used to quickly exchange large quantities of data in an out of Bullet.

ROS: The Robotics Operating System (ROS) is one of the most commonly used middleware frameworks in robotics today. It enables developers to create robotics applications as a network of encapsulated processes, all of which communicate using one common serialization interface. Kineverse is developed as an application for the ROS ecosystem.

A visual representation of the layered structure of Kineverse and of which layers depend on which library, can be found in Fig. 3.2.

3.4 Kineverse from a User's Perspective

In this section I will describe the different components of Kineverse from a user's perspective, by going through a number of use cases. Depending on the use case, a user can use more or less of Kineverse's functionality. There are two main dimensions to a use case: Local or networked use, and static or dynamic model. Tab. 3.1 shows which classes in Kineverse are most relevant for the use case depending on the dimension.

---

[4]https://pyBullet.org/wordpress/

I will start the exploration of the user perspective with use cases focused on using a static model locally. Whenever programming is mentioned, it is always assumed that the user is using Python.

3.4.1 Local Static Models: Just Math

The simplest use case is the one in which the users just want to create arbitrary expressions for which they want to generate augmented gradients. In this case, they can import the `gradient_math` module, which contains an implementation of augmented gradients and matrices storing augmented gradients. It also contains implementations for common mathematical functions, like `abs`, `sqrt`, `log`, and trigonometric functions. Further, it implements constructors for typed variables, common spatial transformations and 3D entities like vectors and points. All these functions are implemented to handle augmented gradient expressions as parameters and update their stored gradients accordingly.

3.4.2 Local Static Models: Loading A URDF

A common first use case for any user is going to be loading an existing URDF into a Kineverse model. This use case is great for exploring the different articulation models that are implemented in Kineverse.

Kineverse offers a URDF loading function which takes a URDF model as input and adds it to a Kineverse model. There are three different models implemented in Kineverse: `ArticulationModel`, `EventModel`, and `GeometryModel`. These three models build upon each other as shown in the inheritance diagram in Fig. 3.3.

Independent of the model implementation the user uses, the URDF loader always loads a URDF as flattened structure, collecting all links under a subpath `links` and all joints under a subpath `joints`. In the Python implementation, these subpaths are members of the `ArticulatedObject` class, which serves as a container for all the joints and links of an articulated object in Kineverse. The joints are created as instances of the `KinematicJoint` class, while links are instances of `RigidBody`. All these classes are convenience features for using Kineverse in Python and do not affect the underlying concept of the articulation model.
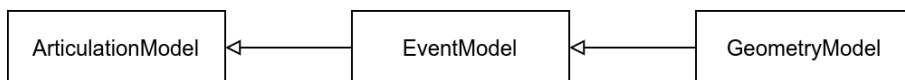

Figure 3.3 Inheritance of the three articulation model implementations in Kineverse.
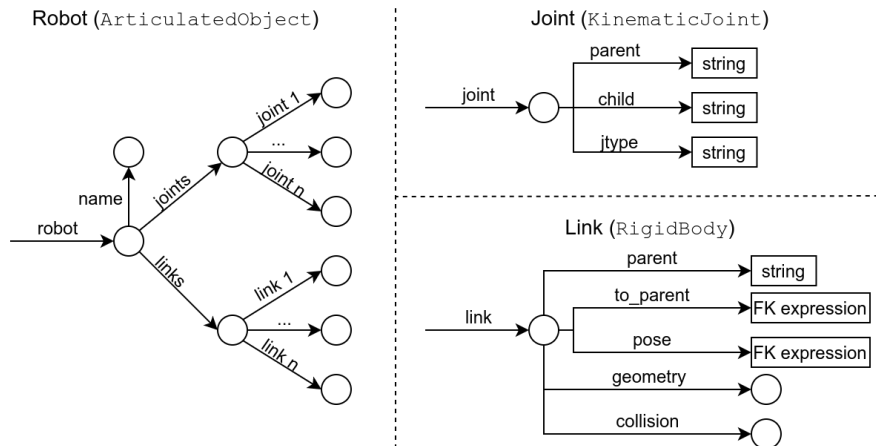
Figure 3.4   Subtree generated by the URDF loader. Names in typewriter-font are the class names of the created data structures.

Fig. 3.4 visualizes the structure of the `ArticulatedObject`, `KinematicJoint`, and `RigidBody`. The URDF loader does not just add mathematical data to the model, but also some metadata and information about object geometry, as the figure shows.

I will cover how this additional data is used, as I describe the models and the functionality, each one adds in the following sections.

ArticulationModel

The `ArticulationModel` is the basic implementation of articulation models in Kineverse. It stores data in a tree structure and can retrieve the stored data for a given path. The data retrieval is implemented using Python's reflection system, so that any Python object can be inserted into the model and its members are accessible using path syntax without any implementation overhead. To complete the formal description of an articulation model, the `ArticulationModel` also implements a constraint storage and a query functionality to retrieve stored constraints by the variables they affect.

The query is useful in scenarios like this: The user queries the `pose` of their robot's gripper from the model. They use the pose in some expression φ and then want to compute a configuration $\mathbf{q}$ for φ. Using the constraint query functionality they can retrieve the constraint set C of constraints that affect the free variables of φ.

EventModel

The `EventModel` extends the `ArticulationModel` by adding a listener functionality to the model, where the user can register callbacks for model changes. In the case of loading a single URDF model, however, this functionality is not needed, which is why I will return to it later, when I discuss dynamic models.

GeometryModel

The `GeometryModel` extends the `EventModel`. This model uses the geometry data added to the Kineverse model to build and maintain a scene in the *Bullet Physics Engine*[5]. This scene can be used to query the closest distance between objects, or query contact points of objects. These types of queries are common in motion planning tasks, as well as perception approaches that do physical plausibility checking, such as the one by Mitash *et al.* [22].

The `GeometryModel` monitors the type of the inserted data. If a `ArticulatedObject`, or `RigidBody` get inserted into the model, a geometric equivalent is added to the Bullet collision scene.

As with the constraints in the `ArticulationModel` the `GeometryModel` implements a query function for bodies affected by a set of variables.

An example using the familiar scenario: A user queries their robot's gripper's pose from the model, uses it in some expression φ, and wants to compute a configuration $\mathbf{q}$ for φ. They want to ensure that $\mathbf{q}$ is not in collision, so they query the articulation model for the geometry that is affected by $V(\varphi)$ and static geometry. The articulation model returns the relevant geometry wrapped as a `CollisionSubworld`, which offers a simple function for updating the collision world using a configuration $\mathbf{q}$ and offers two query functions for computing contact points and closest distances between objects. Using these functions, the user can check whether their chosen $\mathbf{q}$ puts the objects in the scene in collision.

3.4.3   Local Dynamic Models: Changing a Loaded URDF

After loading their URDF file, the users might want to change the loaded model. For instance, they might want to add a kinematic for connecting the robot's base to a global frame of reference. To do so, they need to understand how Kineverse models are built.
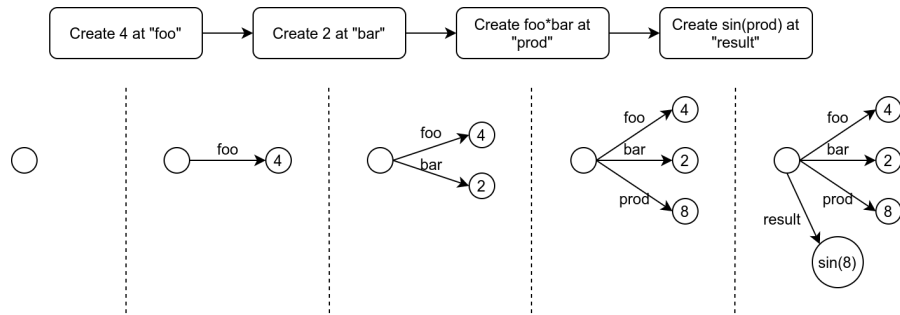
---

[5]https://pyBullet.org/wordpress/

Figure 3.5   Incremental building of a model.

A Kineverse model is the result of a sequential application of operations that transform an original model. The following small example should help to illustrate how models are built by applying operations. In the example, I store two numbers under paths foo, bar and then combine them in two expressions. I construct the model in four steps, the process is represented graphically in Fig. 3.5:

1. Create expression 4 under path *foo*.
2. Create expression 2 under path *bar*.
3. Create expression $foo \cdot bar$ under path *prod*.
4. Create expression $\sin(prod)$ under path *result*.

Note that steps 1 and 2 can be swapped, without changing kinematics of the model. Also note, that changing the data stored under path *foo* after step 3 has no effect on *prod*, or *result*. The application of operations is comparable to imperative programming, in that it matters what the system state is at the time the operation is applied.

These operations are applied to the model and associated with a tag, which becomes their unique identifier. Tags are used to communicate in which order operations should be applied. By default, all operations are applied at the end of the sequence. However, the user can also specify that a new operation should be inserted before or after a specific tag.

The URDF loader creates a model in three phases. In the first it adds the `ArticulatedObject` to the model. In the second it creates all the model's links and tags all the operations as `create link name`. In the third it creates all the forward kinematic connections and tags them `connect parent child`.

To add a new connection for the base of their robot, the user needs to insert the matching operation after the base of the robot has been created. Since they know the name of their base, they know that the tag will be `create my_robot/links/my_robot_base`. Thus they can tell the model to apply the operation after the robot base has been created. In keeping with the convention used by the URDF loader, the new operation should be tagged `connect world my_robot/links/my_robot_base`. Fig. 3.6 shows the phases of loading a URDF file and also
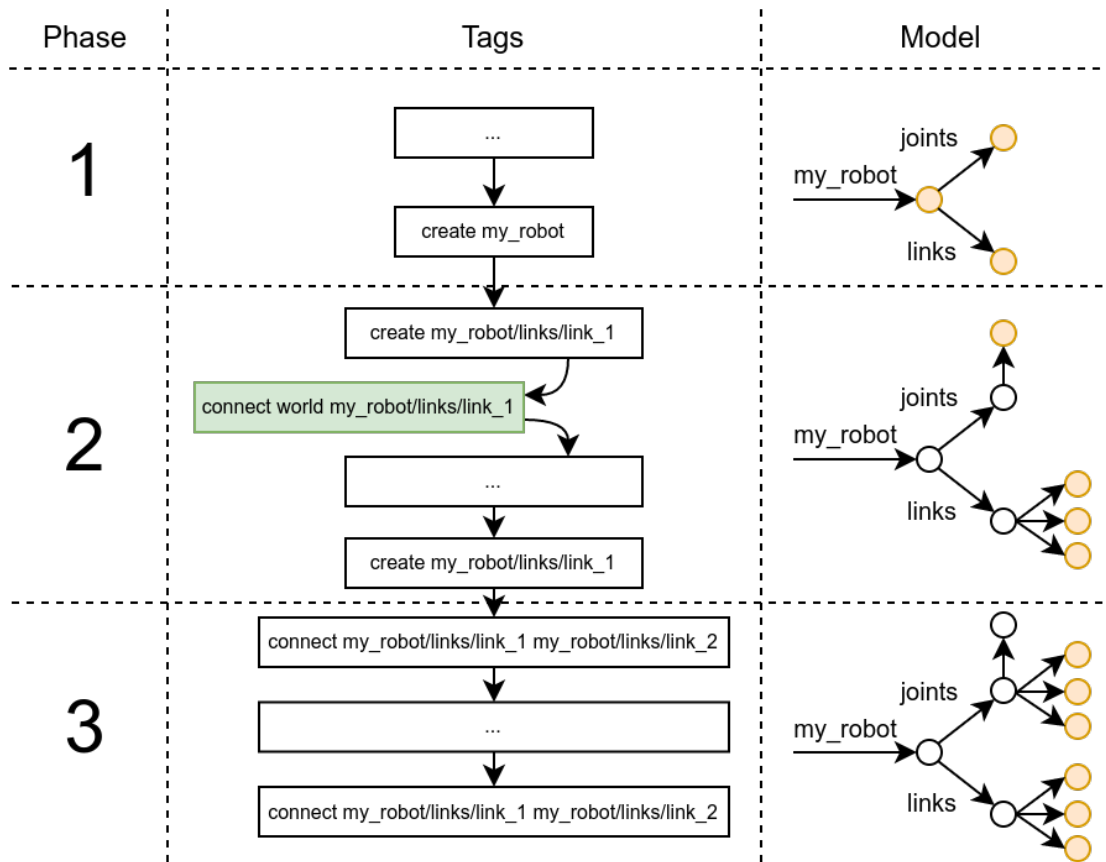
Figure 3.6   Visualization of the process of loading a URDF. Yellow: Parts of the model changed in a phase. Green: Newly inserted operation by the user.

displays where the user's operation would get inserted. By default, Kineverse provides two operations for adding robot base kinematics: `SetOmnibaseJoint` for an omnidirectional base and `SetDiffDriveJoint` for a base using a differential drive.

3.4.4 Networked Models

For any bigger application, a user will want to share their model across different programs in the ROS system. Kineverse comes with a server application that is wrapped in a ROS node. The server stores an articulation model and publishes updates when the model is changed.

Uploading a URDF to a model server does not require any coding. There are multiple small scripts in Kineverse that upload URDF models to the model server, optionally adding a new transform connecting the robot's base to a global world frame.

	`ModelClient`	`OperationsClient`
Listen for data changes	✓	✓
Listen for constraint changes	✓	✓
Publish model changes	✗	✓

Table 3.2   Comparison of the features of the two Kineverse clients.

3.4.5   Using Networked Models

Kineverse implements two different clients for interactions with the model server. A simple client for only subscribing to the model (or a part of it), and a more complicated client for making changes to the model. A brief comparison of their features is shown in Tab. 3.2.

The simpler client implementation `ModelClient` is useful for users who are not interested in changing the model, but only in using it within their applications. The client only processes updates to the model and does not store the sequence of operations that created it. The client relies on the event handling functionality implemented in the `EventModel`. The user can register callbacks for changes to the model. The first type of callback is for changes to the data tree. When the user wants to receive updates whenever the model of their robot's gripper is changed, they would register a callback for the path `my_robot/links/gripper`. The second type of callback is for changes to the set of model constraints. If the user is working with an expression φ and they want to be notified about changes to the constraints affecting φ, they can register a callback that gets triggered whenever the set of constraints affecting $V(\varphi)$ changes.

The more complicated client implementation `OperationsClient` offers the same kind of callbacks as the `ModelClient`, but it processes different server updates. It listens to updates about operations and reproduces the entire model building process on the client's side. If a user is trying to estimate the articulation of two objects, they can use the full sequence of operations to create model hypotheses, evaluate their observation against these and then publish the best fitting model to the server again. While the client is functionally complete as opposed to the `ModelClient`, it also comes with its own significant computational cost, when rebuilding a model from the sequence of operations.

3.4.6   Visualization and ROS Integration

Independent of local or networked model, and independent of static or dynamic model, the user will want to integrate their model with the ROS environment and visualize it given a configuration **q**. Kineverse comes with tools to do both. First I will cover Kineverse's integration with TF, then I will describe the two avenues of visualization, one of which relies on the TF integration.

TF Integration

One of the major infrastructure systems in the ROS environment is *TF*[6]. TF is used to collect 3D frames of reference in the robot's environment and exchange them among nodes in the ROS environment. TF comes with a client module for every major programming language, which can be used to query or update the stored frames. The clients also provide common transformation functions, e. g. for determining the relative pose of the robot's endeffector to an object that was located in the camera's frame of reference. Due to its availability and utility, TF is the backbone of many ROS applications and anything that is related to spatial models should be integrated with it.

Kineverse comes with a TF integration, which enables users to publish the frames in their model given a configuration **q**. As with the articulation models, there are three implementations for TF integration which build upon each other, each one adding a bit more functionality. The basic implementation `ModelTFBroadcaster` is relevant for a user who has a model instance and wants to publish its contained frames. They can instantiate a `ModelTFBroadcaster` and pass some data from the model to it. The broadcaster uses Python's reflection feature to locate all `Frame` objects, which is a standard datatype in Kineverse. If the model was loaded with the URDF loader, the robot links satisfy this requirement, as the `RigidBody` type is an extension of the `Frame` type.

The broadcaster provides a function to publish all the found frames given a configuration **q**. The relative forward kinematics of all frames is evaluated by the function and published to TF. The names of the frames in TF is determined by the paths the frames were located at in the model, e. g. the frame for the robot's head located at `my_robot/links/head` will be published with that path as its name.

In the default ROS installations there is a convenient ROS node called `robot_state_publisher` for publishing the frames contained in a URDF file. The node reads a URDF description from the ROS parameter server and listens for joint state updates on a ROS topic. It keeps re-publishing the frames to TF at a regular interval.

Kineverse comes with its own version of the `robot_state_publisher` called the `model_tf_publisher`, a ROS node that subscribes to a model on a Kineverse server, listens to state updates on a ROS topic, and publishes the frames resulting from the last received configuration to TF at a fixed frequency.

Visualization

Independent of the articulation model users choose or whether they use a networked or local model, they will want to visualize it. Kineverse offers two main avenues for visualization.

---

[6]TF-library: *https://wiki.ros.org/tf*

The first visualization builds on the `ModelTFBroadcaster` from the previous section. Specifically it is implemented in the class `ModelTFBroadcaster_URDF`, which extends the original broadcaster. As with the original implementation users can import the class into their environment and pass a model to it. Building on the frames found by the base class, the new broadcaster generates a URDF description of the frames by trying to identify a common root frame. If it can find any instances of `RigidBody` among the found frames, it generates matching URDF links containing the visual geometry of the rigid body. The URDF description is uploaded to the ROS parameter server and can be visualized in RVIZ using a *RobotModel* display.

The second method of visualization does not depend on TF, but does require the objects that should be visualized to be contained in a `CollisionSubworld`. Given such a world, it turns all the contained objects into ROS' default visualization markers, that can be displayed in RVIZ using a *MarkerArray* display. This second approach is more flexible than the first one, as RVIZ does not automatically reload the generated URDF description from the parameter server when it changes. At the same time it does cause more network traffic and limits users to using the `GeometryModel` implementation for their articulation model.

Terminal Tools

One of ROS' strong features is its varied set of terminal tools ,which can be used to get a quick and easy look into the running system. Kineverse provides three very small tools to interact with a running Kineverse server from a terminal. All of them are implemented in the `ros_tools` ROS node.

list Lists all paths that are defined in the current model, starting at a given root path. The depth to which paths are listed can be limited to avoid outputting a superfluous amount of information.
save Saves the sequence of operations that created the current model to a file in JSON syntax. The sequence can be limited to include only the operations that modified a given path in the model.
load Loads sequence of operations from a JSON file onto the server.

3.5   Kineverse from a Developer's Perspective

In this section I will share information about Kineverse that is aimed at developers who either want to extend Kineverse, or understand more clearly how it actually works. Initially I will spend a moment introducing the symbolic math framework that Kineverse is built on. Then I will look

at the use case of adding new operations to Kineverse which will dive deeper into the workings of the operation system. Following this, I will describe how models are actually exchanged between server and clients, because this is no trivial task. Finally I will describe the bridge between the Kineverse model and the Bullet Physics engine.

3.5.1   The Symbolic Math Backend: Symengine

Kineverse uses the computational algebra system *Symengine*[7] as its symbolic math implementation. The library provides an adequate set of mathematical functions and contains a matrix implementation, making it possible to describe articulation models with it.

The Symengine library is the successor of *SymPy*[8], a symbolic math framework implemented in Python with severe performance issues. The goal of Symengine is to provide the same functionality as SymPy without the performance problems. To enable this, it is implemented in C++ and has bindings for other programming languages e.g. Python, LISP, Java, etc., making it widely compatible. To further increase performance, Symengine enables its users to compile their expressions to native byte code using LLVM[9], making function evaluation acceptably fast.

Symengine is not the only existing symbolic math engine that could be used for the purposes of Kineverse. *Casadi*[10] is another framework that seems to be a strong competitor in this space. However, Kineverse was started with a special focus on being compatible to the *giskardpy*[11] motion library, which was using Symengine at the time.

3.5.2   Adding new Operations to Kineverse

To add new articulations to Kineverse, a new operation needs to be created, which applies the articulation to a model. In this section I cover the operations mechanism in more detail, to enable an understanding of the limitations of the operations API.

Existing articulation model frameworks define their models as a sequence of links and joints, enforcing that the pose of an object is always the result of a chain of transformations with a single starting point. One of the goals of Kineverse is not to enforce a specific structure like this. However, parts of the model still need to be able to build on other parts, e.g. the gripper of a robot needs the be able to build upon the robot's shoulder.

---

[7]Symengine: *https://github.com/symengine/symengine*
[8]SymPy: *https://www.sympy.org/en/index.html*
[9]LLVM: *https://llvm.org/*
[10]Casadi symbolic math framework: *https://web.casadi.org/*
[11]Giskardpy: *https://github.com/SemRoCo/giskardpy*

The solution to this requirement is for Kineverse to produce a model from a sequence of operations, as mentioned before in Section 3.4.3. Such a sequence of n operations o is denoted as $\mathcal{S} = (o_1, \ldots, o_n)$. The model that results from a sequential application of the operations in $\mathcal{S}$ is denoted as $\mathcal{A}^{\mathcal{S}}$. The application of an operation is denoted by the $\vdash$ operator, with $\mathcal{A} \vdash_{o_i}$ expressing that operation o_i is applied to model $\mathcal{A}$. A complete model creation from $\mathcal{S}$ is described formally as

$$\mathcal{A}^{\mathcal{S}} = \mathcal{E} \Vdash_{\mathcal{S}} = \mathcal{E} \vdash_{o_1} \mathcal{A}^1 \vdash_{o_2} \mathcal{A}^2 \ldots \mathcal{A}^{n-1} \vdash_{o_n} \tag{3.16}$$

with $\mathcal{E}$ being the empty model. Operations do two things, they read data from an existing model and write data to the new model. The data is addressed by a set of paths: D_o for the set of paths the operation o reads from, M_o for the paths o modifies. Operations cannot remove any paths from the model.

The dependency on the D_o makes it possible to create sequences which cannot generate a full model, because an operation is missing a dependency. To check whether a sequence is (syntactically) correct, it can be checked whether the modified paths of all operations preceding operation o_i form a superset for the paths that o_i depends on. This condition is expressed formally in Eq. 3.17.

$$\mathrm{valid}(\mathcal{S}) = \bigwedge_i^n \left(D_{o_i} \subseteq \bigcup_{k=1}^{i-1} M_{o_k} \right) \tag{3.17}$$

Implementing a New Operation

To implement a new operation, the user has to subclass the `Operation` type that is contained in Kineverse. This type automatically wraps the reading of the dependencies from the articulation model and the writing to the model. A subclassed implementation has to specify which paths it will depend on, and which ones it will modify. Both sets need to be known before the operation is actually applied to the model. A challenge is that the set of modified paths needs to be complete, meaning it has to include all members of the objects that are being inserted into the model.

The reason for this requirement is a larger dependency checking system, which operates under the hood of the Kineverse `ArticulationModel` implementation. While the formal description of applying an operation to a model describes it as creating a new model, this is not the case in the actual implementation. The primary reason for this is efficiency. An implementation of the formal description would require the entire model to be copied every time an operation is applied to it. This is both expensive and needless, since most of these copies would only be used to produce another copy. Instead, the existing `ArticulationModel` is modified in-place, with the
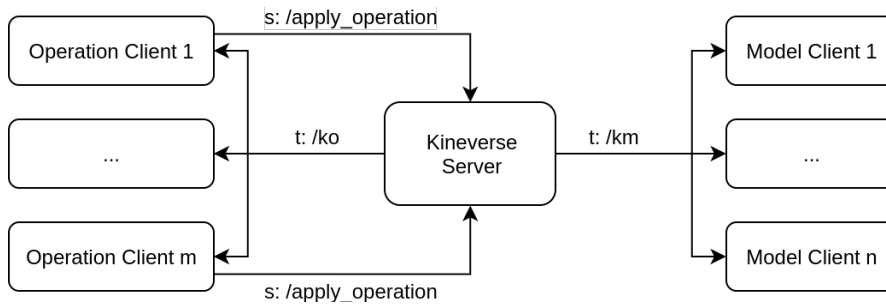
Figure 3.7   Main channels of communication in the kineverse network. ROS services marked as `s`; ROS topics marked as `t`.

operations implementing a memoization pattern so the reversion of an operation application can be performed.

The sequence of operations that produce an `ArticulationModel` is managed by an instance of a `History`. This special class maintains a dependency graph of the stored operations, to enable partial model updates. If an operation is inserted or removed at index i in a sequence $\mathcal{S}$, the straightforward way of updating the model is to re-apply all operations after i to produce the new model. However, this can potentially waste a lot of time, as not all parts of the model have to depend on the removed operation. If robots A, B, C are inserted into $\mathcal{A}$ in that order it does not make sense to rebuild B and C, just because the arm of A was changed.

The `History` class enables partial updates by creating a dependency graph which stores for every operation which other operations immediately depend on it. The history also maintains ordered list structures for all paths, to quickly determine which operations a newly inserted operation depends on given a timestamp and set of path dependencies.

3.5.3   Understanding the Network Exchange

Kineverse is implemented to be used in the context of the ROS ecosystem. Much like the widely used ROS module TF, Kinverse consists of one server, storing a model and multiple clients which connect to it. The server publishes two ROS topics **/ko**, **/km** which distribute updates about the sequence of operations and the built model, respectively. The communication between the server and the clients is depicted in Fig. 3.7.

The updates distributed on the **/km** topic come in the form of `kineverse/ModelUpdate` messages. They contain a timestamp, a model and a list of deleted paths and constraints. The model is of type `kineverse/Model` and only represents a partial model when it is included in the model update message. It contains two aligned lists, one of paths and one of data. The path list determines where the aligned data belongs in the data tree of the larger model. The

data itself is encoded as JSON-strings. This is done to work around the static type system of ROS messages, which does not allow for any kind of inheritance. In addition to the serialized data model, the model message also contains a list of constraints, which are encoded using the `kineverse/Constraint` message type, which structures a constraint as lower and upper bound, and a bounded expression. All these three fields are, again, encoded as JSON-strings. Since the model message only states what the model contains and not what it does not, the model update message adds the list of deleted paths and constraints to communicate what has changed since the last update. It should be noted that the processing of model updates does not have a strict Python dependency. The JSON serializations do include identifiers for Python types, namely as `__type__` attributes in JSON-dictionaries, but these are not necessary to parse the data contained in the model.

Merging model updates into an existing model is straightforward. The data transmitted in the update is placed in the existing data tree and constraint set, and the paths and constraints listed as deleted are removed.

The updates distributed on the **/ko** topic come in the form of `kineverse/OperationsUpdate` messages. These messages contain a timestamp, three aligned lists and a list of tags of deleted operations. The three aligned lists contain tags, stamps, and the associated operations as `kineverse/Operation` messages. The operations consist of a string naming their Python type and a list of JSON-serialized parameters that are required to instantiate the operation. The string naming the operation's Python type is used to load the Python module that contains the operation at runtime and to instantiate the actual implementation.

Merging operations is more difficult. As mentioned in Section 3.4.5, the operations client also allows to track the operations on the basis of a set of paths the user cares about. This requires the client to determine which paths the operations in an update actually affect. It does so by instantiating the operations and checking the paths they modify against a set of paths that are known to be relevant to the paths tracked by the user. This checking is performed in reverse temporal order, to enlarge the set of relevant paths as operations require new dependencies. Once the relevant operations are identified, they are added using a merging algorithm that is implemented by `ArticulationModel`.

While the `ModelClient` cannot make any changes to the model stored on the server, the `OperationsClient` can. The changes the client wants to make are communicated to the server using a ROS service called `apply_operations`. Since the client is not required to maintain a complete model and since different clients can suggest changes based on their local models, clients do not determine the timestamps of operations. Rather, they communicate the relative placement instructions given by the user to the server, who merges them into one model. The resulting model and order of operations is then broadcast by the server using the **/km**, **/ko** topics. The request to the server is transmitted in the form of a `kineverse/OperationCall` message, which contains the operation, itself as `kineverse/Operation`, an instruction, a tag for the operation

and a reference tag for relative insertion. The instruction can be to:

Add: Appending the operation at the end of the sequence of operations, or replacing it if its tag is already known.

Insert Before: Inserts the operation immediately before the operation identified by the reference tag.

Insert After: Inserts the operation immediately after the operation identified by the reference tag.

Remove: Removes the operation associated with the given tag from the model.

Internally, both clients use an instance of `EventModel` to notify their users of changes that have been made to the model. Like the `EventModel`, they provide functions where callbacks for path and constraint updates can be registered.

3.5.4   Understanding Kineverse's Connection to Bullet

After covering the details of Kineverse's operation model, I would like to take a moment to describe Kineverse's relationship to the Bullet Physics Engine. While the bridge to Bullet is not necessary for realizing the conceptual core of Kineverse, it is a potential solution to the problem of connecting arbitrary articulation models with other libraries which perform operations on spatial models. This bridge is also used extensively in a part of the evaluation, which makes it worth mentioning.

As stated before in Section 3.4.2, the `GeometryModel` articulation model implementation automatically populates a Bullet collision scene in the background, as new instances of `RigidBody` get inserted into the model. Given a set of variables, the model can assemble a `CollisionSubworld`, which contains the bodies affected by these variables and provides the user with simple functions for updating the bodies' poses using a $\mathbf{q}$ configuration, and for querying the scene about contact events and smallest distances between objects.

All of these functions are provided by a custom Python binding for the collision detection module of the Bullet physics engine. This custom wrapper is aimed at fast interoperability of Kineverse and Bullet, while offloading as much work as possible into the C++ side of the wrapper. The `CollisionSubworld` implementation makes use of Symengine's LLVM compiler to evaluate all object poses in a single, fast call. To do so, the poses are stacked vertically in one combined matrix, which is compiled into LLVM. The NumPy matrix that results from the evaluation can be passed to the custom Bullet wrapper along with a list of objects to update. The C++ implementation extracts the homogeneous transformations from the stacked matrix and updates the collision scene.

The query functions do not rely on NumPy for their communication, but they do return structured Python objects which is not the case in Bullet's original Python bindings.

The bindings are implemented using *pybind11*[12].

3.6 Practical Methods

In this section I will introduce two methods I developed in the context of Kineverse. The first is an exploitation of the augmented gradient model to express Boolean logic as real-valued functions. The second is a short description of translating Kineverse constraints into a Giskard motion problem. Both of these methods will be used in Chapter 4 during the later stages of the evaluation.

3.6.1 Approximating Boolean Logic

Many complex mechanisms rely on binary switches which constrain the degrees of freedom of their parts. A prime example are doors, whose mobility is dependent on the pose of the handle, if they are in their frames.

Storing the gradient $\nabla^{\dot{x}}\varphi$ explicitly, makes it possible to define arbitrary gradients, which are not related to the definition of f. These gradients can be utilized to define an approximation of boolean logic in a continuous value space, which can, in turn, aid the search for a satisfying solution.

The approximation starts with the definition of *True* as 1.0 and *False* as 0.0. These truth values are grounded in $\mathbb{R}$ by defining the functions $\prec: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and $\succ: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as in Eq. 3.18 with R being a large real number.

$$\begin{aligned}
\prec (a,b) &= \frac{1 + \tanh(b-a)R}{2} \\
\succ (a,b) &= \frac{1 + \tanh(a-b)R}{2} \\
\nabla \prec_{a,b} &= \nabla b - \nabla a \\
\nabla \succ_{a,b} &= \nabla a - \nabla b
\end{aligned} \tag{3.18}$$

This definition causes the functions to behave analogously to the binary predicates $<, >$. The analytical gradients of these functions are practically 0 unless $a \approx b$. The gradients of the functions are redefined to encode the general logic of the functions. In the case of $\prec$ decreasing a,

---

[12]Pybind11: *https://pybind11.readthedocs.io/en/master/*

or increasing b generally moves the evaluation closer to being true, while the opposite moves the evaluation closer to being false. While these gradients are not an accurate model of the change in the value of the function, they do still capture the relative effect that variables in terms a, b have on the value on the function.

The basic boolean operations $\tilde{\neg}, \curlywedge, \curlyvee$ are defined as

$$
\begin{aligned}
\tilde{\neg}(a) &= 1 - a \\
\curlywedge(a, b) &= a \cdot b \\
\curlyvee(a, b) &= \tilde{\neg}(\curlywedge(\tilde{\neg}(a), \tilde{\neg}(b))) \\
\nabla\tilde{\neg}_a &= -\nabla a \\
\nabla\curlywedge_{a,b} &= \nabla a + \nabla b
\end{aligned}
\tag{3.19}
$$

$\curlyvee$ is defined using the $\neg(\neg a \wedge \neg b) \equiv a \vee b$ equivalence to avoid adding another saturation function and defining another gradient.

With these functions it is now possible to define boolean expressions and embed them into a continuous model.

3.6.2 Building a Giskard Motion Problem

For practical experiments, a way of converting Kineverse constraints and expressions into a motion problem is needed. This section describes a strategy for converting Kineverse constraints into a Giskard motion problem. The initial strategy is very simple but unable to cover all cases of constraints and variables. The later strategies are more complicated and add requirements to Giskard's state vector **o** and update rules for generating trajectories, but manage to cover the entirety of the Kineverse features.

Giskard Motion Problem Formulation

The Giskard framework was briefly introduced before in Section 2.2.2. In this section I will re-iterate the problem formulation in greater detail and then describe how to construct a Giskard motion problem from Kineverse constraints.

As mentioned before, a Giskard motion problem is formulated as minimization problem which is constrained by m hard and p soft constraints. The cost function is defined over a vector of n controllable variables and p slack variables. The constraints are defined as follows:

Controllable Constraints define limits on the vector of controlled variables $\dot{\mathbf{c}}$. They are accumulated in boundary vectors $\mathbf{l}, \mathbf{u}$ that are used to enforce $\mathbf{l} \leq \dot{\mathbf{c}} \leq \mathbf{u}$. They also assign a weight scaling the cost of utilizing the controllable variables. For this section, the set of controllable constraints will be denoted as set Q containing constraint tuples $(l, u, \dot{c}, w)$ where $l, u \in \mathbb{R}$ are the bounding expressions, $\dot{c}$ is the controlled variable and $w \in \mathbb{R}$ is the weighting expression.

Soft Constraints define task objectives which can be disobeyed at a cost. In this section, they will be accumulated in the set S as tuples (l, u, e, w) with all components being expressions in $\mathbb{R}$. Each of these tuples expresses the following relationship $l \leq \dot{e} + \epsilon \leq u$, where ϵ is a slack variable that can be used to satisfy the constraint at the cost of $w\epsilon^2$.

Hard Constraints express non-negotiable constraints for the motion, like the robot's joint limits. In this section they are collected in the set Z as tuples (l, b, e) that express the constraint $l \leq \dot{e} \leq u$.

These constraint sets are assembled into the following minimization problem:

$$\begin{aligned} \min_{\mathbf{s}} \quad & \mathbf{s}^T \mathbf{H} \mathbf{s} \\ \text{s.t.} \quad & \mathbf{l}_A \leq \mathbf{A}\mathbf{s} \leq \mathbf{u}_A, \\ & \mathbf{l} \leq \mathbf{s} \quad \leq \mathbf{u} \end{aligned} \tag{3.20}$$

with

$$\begin{aligned} \mathbf{s} &= \left[\dot{c}_1, \ldots, \dot{c}_n, \epsilon_1, \ldots, \epsilon_p \right]^T \\ \mathbf{H} &= diag(w_{q,1} \ldots w_{q,n}, w_{s,1}, \ldots, w_{s,p}) \\ \mathbf{A} &= \begin{bmatrix} \nabla^{\dot{\mathbf{c}}} Z & 0 \\ \nabla^{\dot{\mathbf{c}}} S & \mathbf{I}^{p \times p} \end{bmatrix} \\ \mathbf{l} &= \left[l_{q,1}, \ldots, l_{q,n}, -\infty_1, \ldots, -\infty_p \right]^T \\ \mathbf{u} &= \left[u_{q,1}, \ldots, u_{q,n}, \infty_1, \ldots, \infty_p \right]^T \\ \mathbf{l}_A &= \left[l_{z,1}, \ldots, l_{z,m}, l_{s,1}, \ldots, l_{s,p} \right]^T \\ \mathbf{u}_A &= \left[u_{z,1}, \ldots, u_{z,m}, u_{s,1}, \ldots, u_{s,p} \right]^T \end{aligned} \tag{3.21}$$

The components of $\mathbf{H}, \mathbf{A}, \mathbf{l}, \mathbf{u}, \mathbf{l}_A, \mathbf{u}_A$ can all be parameterized over a vector $\mathbf{o}$, a subset of which is the current position of the controlled variables $\mathbf{c} \subseteq \mathbf{o}$. The generation of the motion is wrapped as a function calculating a desired velocity vector for the controlled variables:

$$\dot{\mathbf{c}}_{des} = f(\mathbf{o}) \tag{3.22}$$

Conversion Strategy

There are a couple limiting factors that have to be considered when converting Kineverse constraints to Giskard constraints:

- Giskard can only define constraints for the first derivative of a constrained equations, not for the expression itself.
- Giskard has no built-in notion of time.
- Giskard does not require the value of $\dot{\mathbf{c}}$ to be part of $\mathbf{o}$.

The first observation is relevant to translating positional constraints from Kineverse to Giskard. If there is a constraint in Kineverse $(-2 \leq a \leq 2)$, then this constrains the value of a to be within the interval $[-2, 2]$. If copied directly to Giskard, this constraint would define $(-2 \leq \dot{a} \leq 2)$, which leaves the value of a unconstrained. A simple conversion is to subtract the constrained expression from the boundaries $(-2 - a \leq a \leq 2 - a)$, decreasing the value range of $\dot{a}$ as it approaches the boundaries of a.

This simple conversion has two problems: It is over constraining $\dot{a}$ and simultaneously does not enforce the boundaries of a reliably. Both of these problems are caused by the lack of a concept of time. Given the Kineverse constraint set $\mathcal{C} = \{(-0.1 \leq a \leq 0.1), (-4 \leq \dot{a} \leq 4)\}$, the Giskard equivalent would be $Z = \{(-0.1 - a \leq \dot{a} \leq 0.1 - a), (-4 \leq \dot{a} \leq 4)\}$. Given Z, $\dot{a}$ will always be limited to the interval $[-0.2, 0.2]$ even tough the original model allows for $[-4, 4]$. At the same time, there is no guarantee that a will actually remain within $[-0.1, 0.1]$, since it is not clear of how long the integration step Δt will be. This can be rectified by adding Δt to the converted constraints:

$$\mathcal{C} \Rightarrow Z$$
$$\{(-0.1 \leq a \leq 0.1), (-4 \leq \dot{a} \leq 4)\} \Rightarrow \left\{ \left(\frac{-0.1 - a}{\Delta t} \leq \dot{a} \leq \frac{0.1 - a}{\Delta t} \right), (-4 \leq \dot{a} \leq 4) \right\} \tag{3.23}$$

For a discrete timestep integration scheme, including Δt guarantees that a will remain within its configuration space and make it possible to utilize the full range of $\dot{a}$, given that Δt is small enough. If the motion is generated as a part of a control loop, Δt has to be chosen so that it matches the execution frequency of the loop.

Chapter 4

Evaluation

In this chapter, the Kinverse articulation model is evaluated. This is done in two stages: First, its modelling capabilities are showcased in a number of small examples. Then a simulated robotic system is shown, performing a simple, simulated task in a kitchen environment.

4.1 Modeling Capabilities

In this section I showcase the modeling capabilities of Kineverse. I compare Kineverse's capabilities to those of URDF, even though URDF is the most limited out of the three available modeling languages: URDF, SDF and COLLADA. I do so because URDF is the de-facto standard in the ROS environment and Kineverse was developed with that environment in mind.

I identify a number of different types of articulation models by which I compare the two frameworks:

Kinematic Chain The simplest type is the kinematic chain, which is a group of bodies connected by joints as a single, open chain. A practical example of a kinematic chain is a robot arm.

Kinematic Tree The branching, loop-free version of the kinematic chain is a kinematic tree. A kinematic tree can be used to model a robot with multiple arms, or more generally a system with multiple endpoints.

Kinematic Loop A kinematic loop is a group of bodies connected by joints in a looping fashion. When a two-armed robot lifts a tray, the arms and the tray define a loop in which the arms limit each other's joint positions.

Linear Dependent Joints A joint is linearly dependent on another joint if its position can be calculated from the other joint's position by applying some linear scaling and adding an

Articulated System	Kinematic Chain	Kinematic Tree	Kinematic Loop	Linear Dependent Joints	Non-Linear Dependent Joints	Multidependent Joints	Conditional Kinematics	Differential Kinematics
URDF	✓	✓	✗	✓	✗	✗	✗	✗
Kineverse	✓	✓	✗	✓	✓	✓	✓	✓

Table 4.1   Comparison of the modeling capabilities of Kineverse and URDF.

offset. Linear dependence is often used to model the behavior of two-fingered pinch grippers, which have two moving fingers, but only one actual degree of freedom.

Non-Linear Dependent Joints Similar to linear dependence, with the only difference being that the transformation function does not need to be linear. Such a joint can be used to express the position of a piston that is being driven by a rotating wheel.

Multi-Dependent Joints A joint is multi-dependent if its position can be determined as a function of the positions of a group of other joints. This model can be used to model connections using multiple tendons to control the pose of a link.

Conditional Kinematics A conditional kinematic models the mobility of a DoF based on some switch conditions. Locking mechanisms like door handles can be understood as conditional kinematics.

Differential Kinematics A lot of mechanisms encountered in everyday life do not have a forward kinematic, but do define functions that can be used to predict the velocity of a property. A common example of such a mechanism in robotics are differential drives, which do not define a forward kinematic for the position and rotation of a robot, but do define functions mapping the velocity of a robot's wheels to its linear and angular velocity.

I show that it is possible to express these different types of articulations by building up a fetch robot in Kineverse in multiple steps. For the remaining capabilities I build small, self-contained examples which demonstrate that it is possible to express these kinematics in Kineverse. The full comparison of Kineverse and URDF is collected in Table 4.1.

Across all steps of building up the fetch robot, I also demonstrate the successful integration of Kineverse into ROS, by using the kineverse server and its clients to manage the model. To show that Kineverse is integrated with the central TF-library and the visualization tool RVIZ, I use Kineverse's TF-publisher to publish the model's frames and visualize the complete model in RVIZ. The dataflow between the different components is depicted in Fig. 4.1.
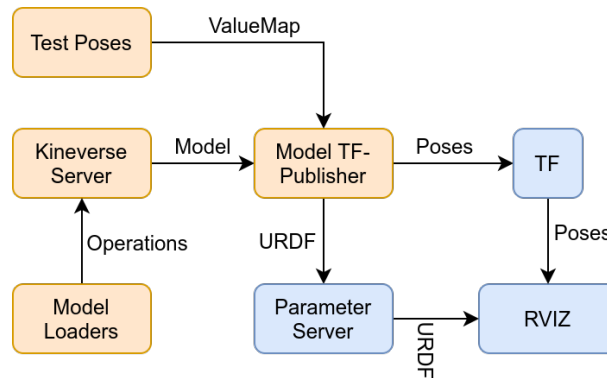
Figure 4.1   Components and their dataflow used in the modeling evaluation.

4.1.1   Kinematic Chains

The fetch robot contains multiple kinematic chains. I choose to start at the `shoulder_pan_link` and load it up until the `gripper_link`, excluding the fingers because they split the chain of the arm into a tree.

I set a number of different joint positions to check that the model was loaded correctly. The resulting visualization and TF-tree can be seen in Fig. 4.2.

4.1.2   Kinematic Trees

I expand the loaded model of the fetch to include its torso and head, which, together with the arm, form a kinematic tree. The fingers of the gripper are still excluded, as they involve a linearly dependent kinematic.

I set a number of different joint positions to check that the model was loaded correctly. The resulting visualization and TF-tree can be seen in Fig. 4.3.

4.1.3   Kinematic Loops

There are no ways of modeling kinematic loops in either URDF or Kineverse. A URDF file that includes a kinematic loop will raise an exception in all standard parsers. Kineverse does not restrict the user from building articulation models with loops, but they will simply override the forwards kinematic that was previously defined. In a small example:
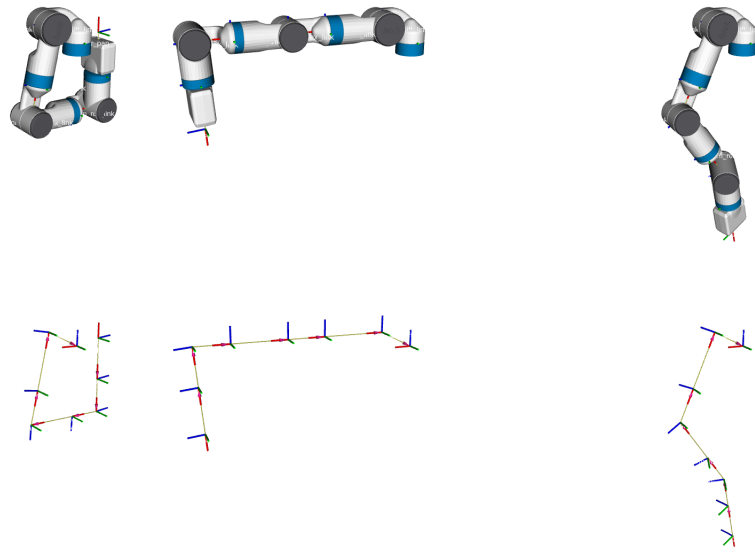
Figure 4.2 Visualization of fetch's arm loaded into Kineverse. First row: Mesh visualization; Second row: Display of corresponding TF-tree.
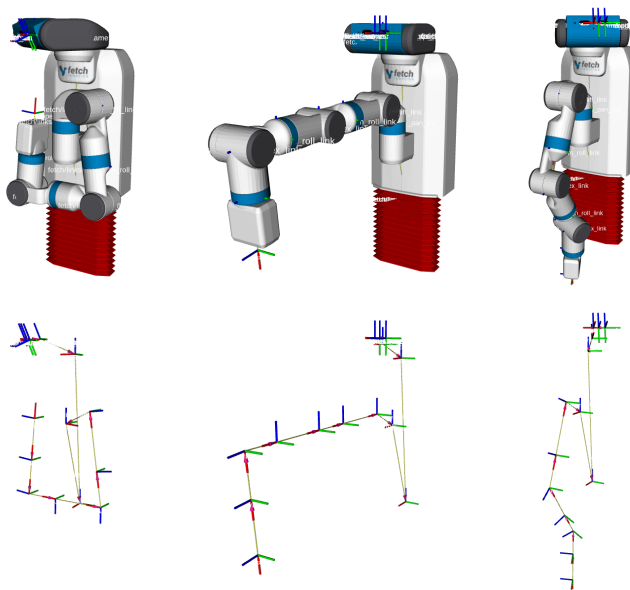
Figure 4.3 Visualization of fetch's torso, arm, and head loaded into Kineverse. First row: Mesh visualization; Second row: Display of corresponding TF-tree.
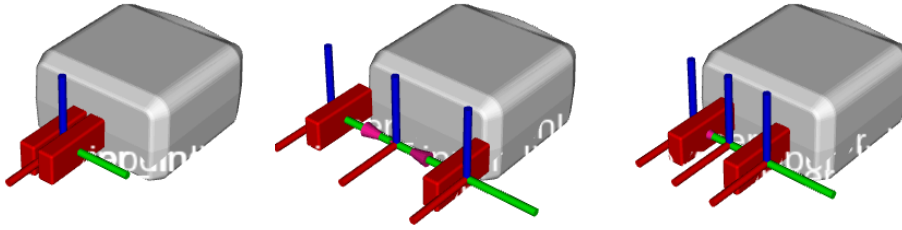
Figure 4.4   Visualization of fetch's pinch gripper loaded into Kineverse.

If A, B, C, D are frames which are connected by some forward kinematic expression as $A \rightarrow B \rightarrow C \rightarrow D$, then applying that expression to $D \rightarrow A$ will only override the transform of A and not affect any other object in the loop. This is the case because Kineverse models are built sequentially.

It is possible to simulate kinematic loops in Kineverse applications by adding constraints to the model, but then a solver is needed to generate a valid model configuration.[1]

4.1.4 Linear Dependent Joints

A simple example of a linearly dependent joint can be found in fetch's gripper. It is a two-fingered pinch gripper which can only open symmetrically. I load the gripper in isolation and use the Kineverse infrastructure to set it to a number of different poses and visualize the results. Fig. 4.4 shows the visualization.

4.1.5 Non-Linear Dependent Joints

For an example of non-linear dependent joints, I use an example from the world of mechanics. Specifically, I use the rod and piston construction from a car engine to demonstrate that it is possible, to model this very constrained, non-linearly dependent kinematic chain. Fig. 4.5 shows such an assembly and its parts.

Given a crank shaft with a crank pin distance of r and a connecting rod length of d, the kinematic of the shaft transformation $\mathbf{T}_S$, the rod transformation $\mathbf{T}_R$ and the piston transformation $\mathbf{T}_P$ can be expressed as a function of a single DoF θ as in Eq. 4.1, where $R(\mathbf{x}, \alpha)$ expresses a rotation

---

[1]COLLADA has a way of explicitly stating that a system is a loop. This is not afforded as a single operation by Kineverse's operational model.
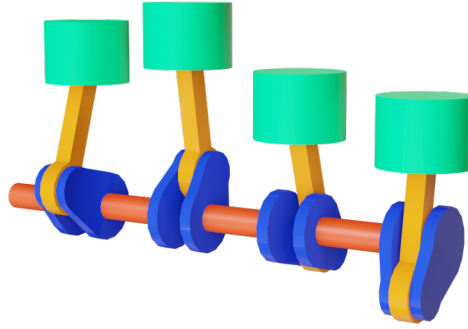
Figure 4.5   A crankshaft from a car's combustion engine. Red: Crankshaft; Blue: Crank pin; Green: Piston; Yellow: Rod connecting crank pin and piston.

around the X-axis by α radians. The result is once again visualized using RVIZ - as can be seen in Fig. 4.6.

$$
\begin{aligned}
\mathbf{T}_S &= \mathbf{T}(\mathbf{v}) \cdot R(\mathbf{x}, \theta) \\
\mathbf{T}_R &= \mathbf{T}_S \cdot \mathbf{T}(0, 0, r) \cdot R\left(\mathbf{x}, -\operatorname{asin}\left(\sin(\theta)\frac{r}{d}\right) - \theta\right) \\
\mathbf{T}_P &= \mathbf{T}_R \cdot \mathbf{T}(0, 0, d) \cdot R\left(\mathbf{x}, \operatorname{asin}\left(\sin(\theta)\frac{r}{d}\right)\right)
\end{aligned}
\tag{4.1}
$$

4.1.6   Multidependent Kinematics

Some robots, like *Charlie* in Fig. 4.7, use a tendon-like combination of linear actuators to control their links. These constructions make the pose of the link a function of the positions of these multiple actuators. I demonstrate that it is possible to create such models in Kineverse by building a small, artificial example that is comparatively easy to follow.

The example consists of a platform that is connected to a static frame in one point. The connection allows the platform to pitch and roll over two intersecting axes. The platform is actuated using two prismatic actuators that are mounted to the same static frame. A physical build of the example is displayed in Fig. 4.8.

Since the two actuators can only rotate about one axis, both define a triangle in a plane that is perpendicular the the actuator's axis of rotation (cf. Fig. 4.8). Each triangle is formed between the mounting point of the platform $\mathbf{a}$, mounting point of the actuator $\mathbf{b}$, and the point at which the actuator's end is hinged to the platform $\mathbf{c}$. The points $\mathbf{a}, \mathbf{b}$ are constant, while $\mathbf{c}$ is not. However, the distances between the pairs of points are known, which enables the determination of the triangles' angles.
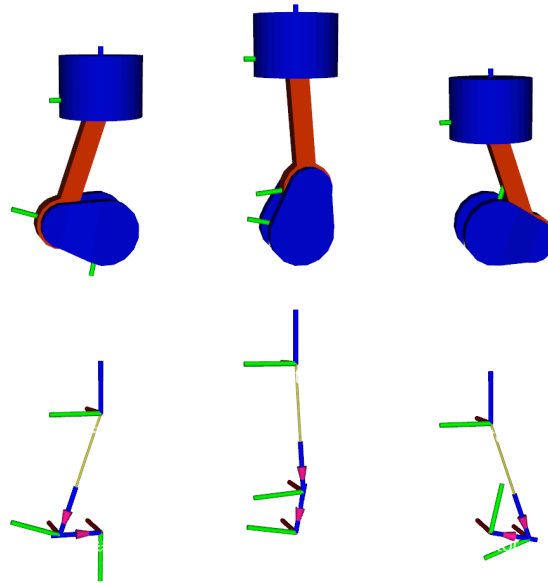
Figure 4.6 A car engine's piston as an example of non-linear dependence in an articulation model.
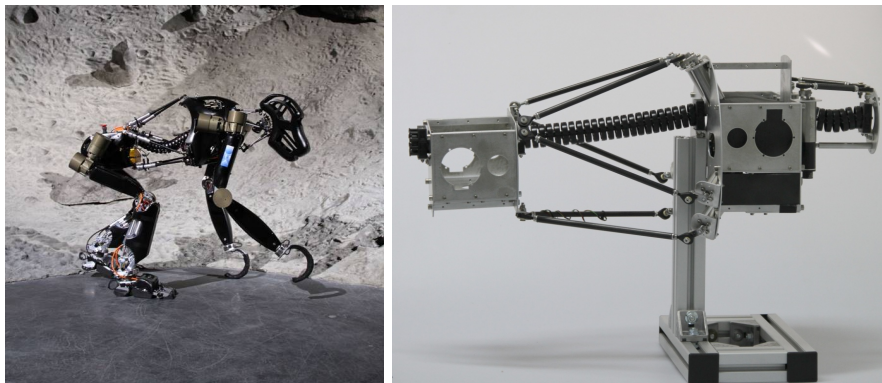

Figure 4.7 Robot *Charlie* as an example of a biologically inspired robotic structure. Multiple linear actuators connect the hip and upper body of the robot, creating a multi-dependent kinematic. *Charlie* is a creation of the DFKI Robotics Innovation Center in Bremen. (Source: Daniel Kühn, DFKI GmbH)
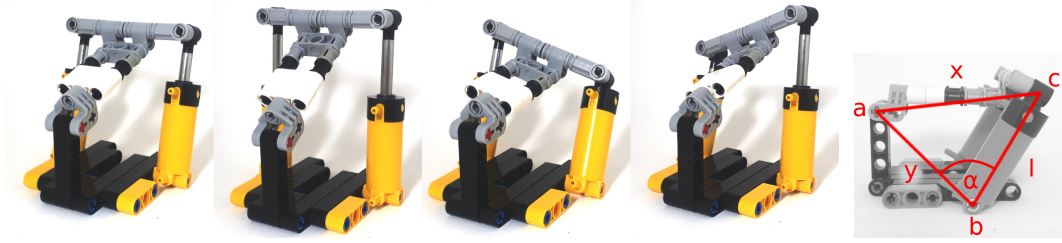
Figure 4.8 Physical equivalent of modeled, multidependent kinematic. The two pistons on the side of the construction control the pitch and roll of the platform. Schematic on the right shows the triangle problem formed by the actuators.

For the kinematic of the platform, only the angles α are relevant:

$$\begin{aligned} \alpha_l &= \cos^{-1}\left(\frac{l^2 + y^2 - x^2}{2ly}\right) \\ \alpha_r &= \cos^{-1}\left(\frac{r^2 + y^2 - x^2}{2ry}\right) \end{aligned} \tag{4.2}$$

where x and y are the lengths of the triangle sides, cf. Fig. 4.8. Using these angles the forward kinematic of the actuators can be constructed as

$$\begin{aligned} \mathbf{T}_l &= \mathbf{T}_M \mathbf{R}_M \mathbf{R}(\Pi + \alpha_l) \\ \mathbf{T}_r &= \mathbf{T}_M \mathbf{R}_M \mathbf{R}(\Pi + \alpha_r) \end{aligned} \tag{4.3}$$

where $\mathbf{R}_M$ is the rotation of the lower side of the triangle in the common frame of reference of the actuators and the platform. These two transformations can be used to determine the endpoints of the prismatic actuators $\mathbf{p}_l, \mathbf{p}_r$. The pitch β of the platform is determined from the relative location of a to the center point $\frac{1}{2}(\mathbf{p}_l + \mathbf{p}_r)$. The roll γ of the platform is determined by the slope of the vector from one piston endpoint to the other.

$$\begin{aligned} \mathbf{v} &= \frac{1}{2}(\mathbf{p}_l + \mathbf{p}_r) \\ \beta &= \arctan2(v_y, v_x) \\ \mathbf{w} &= (\mathbf{R}(\beta)\mathbf{Y}) \cdot (\mathbf{p}_l - \mathbf{p}_r) \\ &+ (\mathbf{R}(\beta)\mathbf{Z}) \cdot (\mathbf{p}_l - \mathbf{p}_r) \\ \gamma &= \arctan2(w_y, w_x) \end{aligned} \tag{4.4}$$

These results are visualized using RVIZ and can be seen in Fig. 4.9.
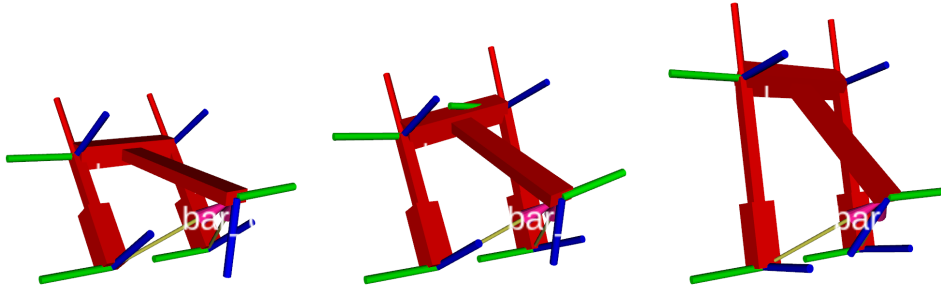
Figure 4.9   A model of a multidependent kinematic. The prismatic actuators on the left and right control pitch and roll of the T-shaped link.
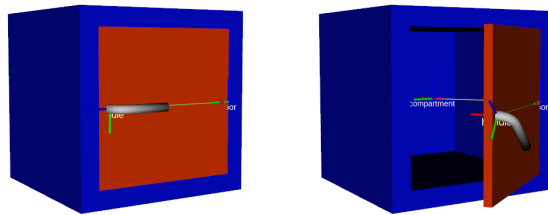

Figure 4.10   Visualization of the compartment modeled in this section.

4.1.7   Conditional Kinematics

Conditional kinematics can be used to express the switching degrees of freedom of a system based on some logical conditions. To show that it is possible to model conditional kinematics in Kineverse, I model a shelf compartment which consists of three parts: The compartment itself, a door, and a handle. The handle is mounted to the door, which is, in turn, mounted to the compartment. Fig. 4.10 gives an impression of this setup. The configuration space consists of two variables d, h, with d being the rotation angle of the door and h being the angle of rotation of the handle. Each of these objects is associated with a transformation $\mathbf{T}_C, \mathbf{T}_D, \mathbf{T}_H$. The door is setup to rotate around the Z-axis of the compartment, while the handle is set to rotate around the X-axis of the door:

$$\mathbf{T}_D = \mathbf{T}(\mathbf{v}) \cdot R(\mathbf{z}, d)$$
$$\mathbf{T}_H = \mathbf{T}_D \cdot \mathbf{T}(\mathbf{u}) \cdot R(\mathbf{x}, h) \tag{4.5}$$

The function $\mathbf{T}(\mathbf{v}), \mathbf{T}(\mathbf{u})$ in the definition above signifies some constant translations that are used to move the mounting points of the objects in their parent frames.

To model that the door is locked as long as the handle is not turned at least to a certain angle, I add constraints on the velocity $\dot{d}$ using $\prec$ to check the unlocking condition:

$$-\infty \leq \dot{d} \quad \leq (d \succ \epsilon) \curlyvee (h \prec x)\infty \qquad (4.6)$$

This constraint can be read as: $\dot{d}$ can be positive while $d > \epsilon$, or $h > x$, where ϵ is the threshold angle at which it the locking mechanism will not catch anymore and x is the angle at which the handle unlocks the door. This locking constraint is combined with two other constraints which enforce the positional limits of the door and handle:

$$
\begin{aligned}
0 \leq d \quad &\leq \frac{1}{2}\pi \\
0 \leq h \quad &\leq \frac{1}{4}\pi
\end{aligned}
\qquad (4.7)
$$

4.1.8 Differential Kinematics

One of the most popular ways of moving a robot is the differential drive: Two wheels which are mounted along one axis and can be controlled independently. The differential drive is cheap and easy to build and allows a robot to go forwards, backwards and turn. Though it is a very common kinematic, it cannot be expressed in URDF. While URDF can model the existence and the DoFs of the wheels, it cannot express the effect these are going to have on the motion of the base with respect to the world.

The kinematic equations of differential drives are well known - e. g. described in chapter 13 of *Planning Algorithms* [18].

The velocity equations can be added to the pose of the base by extending its gradient information. If $\dot{u}_l, \dot{u}_r$ are the rotation speeds for the left and right wheel, and x, y, θ are the X, Y coordinates of the base's pose and θ is its rotation around the Z-axis, then their gradient is extended by the following:

$$
\begin{array}{c}
\begin{array}{cc} \dot{u}_l & \dot{u}_r \end{array} \\
\begin{array}{c} \nabla x \\ \nabla y \\ \nabla \theta \end{array}
\left[
\begin{array}{cc}
\frac{r}{2}\cos\theta & \frac{r}{2}\cos\theta \\
\frac{r}{2}\sin\theta & \frac{r}{2}\sin\theta \\
-\frac{r}{L} & \frac{r}{L}
\end{array}
\right]
\end{array}
\qquad (4.8)
$$

where r is the radius of the wheels and L is the distance between them. There is no way of directly visualizing the differential drive model, since it does not effect the robot's position. The drive
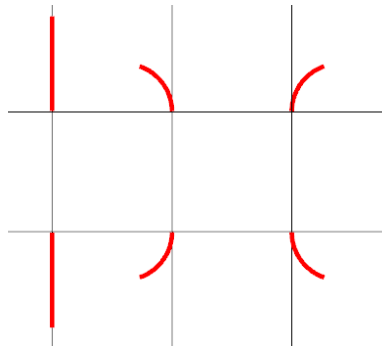
Figure 4.11   Simple test of the differential drive's gradients. Wheel speeds are set to constant values and integrated over time. In order starting from top left: Both forwards, Right forward, Left forward, both backwards, right backwards, left backwards.

Figure 4.12   A look onto the fetch's differential drive system.

is evaluated by setting different constant wheel speeds and integrating the evaluated gradients. Fig. 4.11 shows the resulting trajectories.

4.1.9 Robots

To demonstrate compatibility with existing robot models, I import two robots into Kineverse.

Fetch

Over the course of the previous modeling exercises, the fetch robot has been loaded partially into Kineverse. As a final step, I load the entire fetch using the default URDF loader. All of the robot's constraints can be queried from the articulation model. When the robot is loaded into
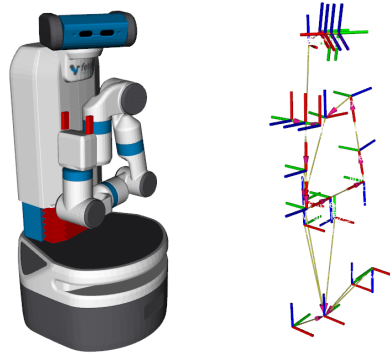
Figure 4.13   Full fetch model loaded into Kineverse. Left: Mesh visualization; Right: TF tree of the loaded model.

an instance of the `GeometryModel` class, a representation is automatically created in a Bullet collision world as the robot is loaded. Setting different variable assignments and rendering the resulting robot pose to RVIZ, shows that the robot has been loaded correctly.

To improve upon the articulation model of the fetch that can be captured using URDF, I use the operations' history feature of Kineverse to connect fetch's base to the origin frame of the world using a differential drive kinematic (see Fig. 4.12 for a look at fetch's drive system). To check that the gradients are propagated properly to the other links of the fetch, I retrieve the gradient for the position of the endeffector $\mathbf{p}_G$ and check that the expressions from the base have propagated as:

$$
\nabla \mathbf{p}_G =
\begin{matrix}
\dot{u}_l & \dot{u}_r & \ldots \\
\begin{bmatrix}
\neq 0 & \neq 0 & \ldots \\
\neq 0 & \neq 0 & \ldots \\
0 & 0 & \ldots
\end{bmatrix}
\end{matrix}
\tag{4.9}
$$

The exact gradient for the X and Y coordinates have been omitted, since they include the complete forwards kinematic of the arm. The fully loaded, visualized fetch is depicted in Fig. 4.13.

PR2

To show that the fetch is not the only robot that can be loaded into Kineverse, I load a PR2 from a URDF file. As before, all constraints for the PR2 can be queried from the articulation model
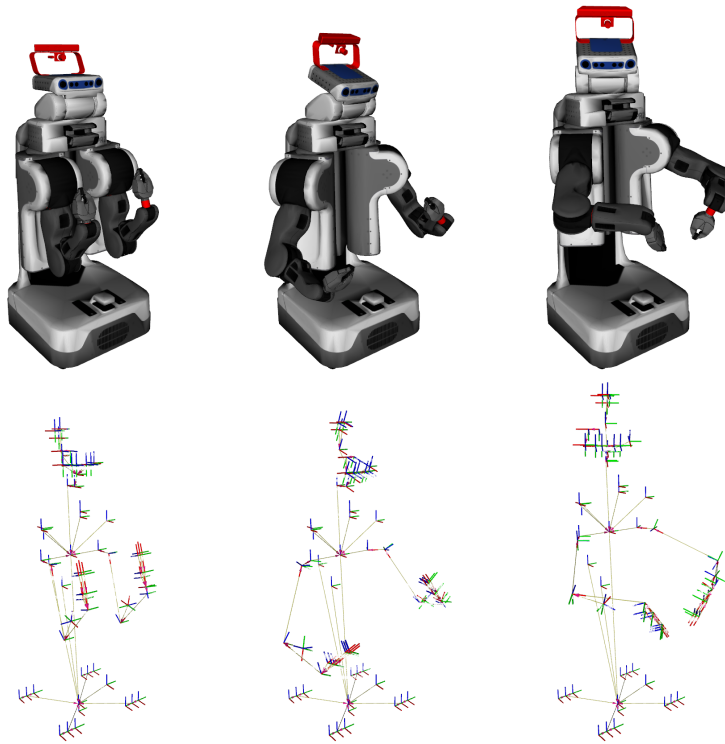
Figure 4.14   A PR2 loaded into Kineverse and visualized using RVIZ and TF. First row: Mesh visualization; Second row: TF tree.

and a collision representation is created automatically when the robot is loaded into an instance of the `GeometryModel`. Setting different variable assignments and rendering the resulting robot pose to RVIZ, shows that the robot has been loaded correctly. Fig. 4.14 shows the loaded PR2.

The PR2 has an omni-directional base, which makes it unnecessary to modify the base transform.

4.1.10   Faucet

As a final example in this modeling section, I will look at linking spatial and non-spatial properties and how they can support reasoning. Consider a simple, modern faucet like the one depicted in Fig. 4.15. Lifting the handle upwards opens the tap, turning it to the left will make the water hotter, turning it to the right will make the water colder.

Normal articulation models are limited to expressing facts about the relative motion of objects. In Kineverse, however, it is possible to model mechanisms which do not only affect the spatial relations of objects - mostly because Kineverse is largely agnostic about models.
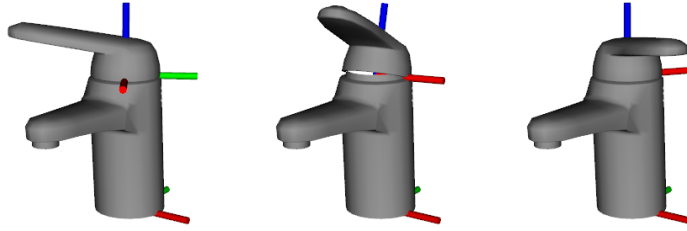
Figure 4.15   A simple, modern faucet as seen in most bathrooms today.

To model a simple faucet, I define the frame of the handle $\mathbf{T}_H$ relative to the frame of the faucet's base $\mathbf{T}_B$:

$$\mathbf{T}_H = \mathbf{T}_B \cdot \mathbf{T}(\mathbf{v}) \cdot R(\mathbf{z}, \alpha) \cdot R(\mathbf{y}, \beta) \tag{4.10}$$

Using the two angles of rotation α, β, I define the logic of the water flow w and water temperature t as

$$
\begin{aligned}
w &= -\beta \cdot m_1 + a \\
t &= \alpha \cdot m_2 + b
\end{aligned}
\tag{4.11}
$$

where m_1, m_2 are some scaling factors that map the angles to real-world quantities, b is an offset to set a neutral temperature, and a is an offset to model a leaky tap. Adding the frames with associated geometry to a `GeometryModel` $\mathbf{T}_B, \mathbf{T}_H$ and the expressions w, t to the same model, enables me to query the model for information. I can query from it the parameter sets of w, t: $V(w), V(t)$ using these sets I can query the model for geometry that is affected by them. Once I have that geometry, I can infer from the gradient of the associated pose how the geometry needs to be moved to change w, t to a goal configuration. Since this is venturing into the space of solving tasks in Kineverse, I will leave the example at this and let Sections 4.2.1 and 4.2.3 give fuller pictures.

4.2   Task Integration

The experiments in the previous section were focused on examining the modeling capabilities of Kineverse. In this section, I present three example applications using Kineverse. The first is a task, which shows how the symbolic expressions in Kinverse can be used to do inference and reasoning. The other two tasks are real-world inspired, and executed in a simulated environment.

The goal of this evaluation is to see if Kineverse can be utilized in real robotic problems and how it performs in a running system.

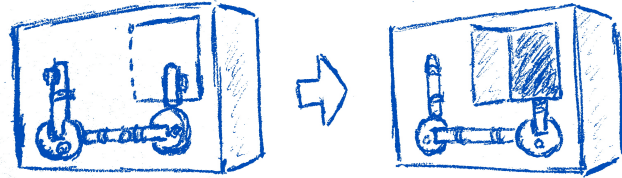## 4.2.1   Conditional Degrees of Freedom

Figure 4.16   An example of a lockbox. Left: Closed; Right: Opened

More complex mechanisms require boolean switches to be modeled accurately. To showcase that such models can be expressed in Kineverse, a model of a lockbox is created and a simple algorithm exploring the box and opening it is defined. The lockbox is chosen due to the interest in the problem in recent research [17, 3]. A depiction of a typical lockbox can be seen in Fig. 4.16. It should be noted, that the model presented in this section is definitely not the ideal representation of a lockbox. Neither is the algorithm the best algorithm to devise a sequence of interactions to open such a box. This example exists as a proof of concept, that there is a way of constructing such mechanisms in the world of Kineverse.

Model

The lockbox has six degrees of freedom a, b, c, d, e, f, which lock each other in the following way:

- a unlocks when $b > 0.4$ and $c > 0.6$.
- b unlocks when $d > 0.8$.
- c unlocks when $d > 0.5$ and $e > 1.0$.
- d unlocks when $e > 0.9$.
- e unlocks when $f > 1.2$.
- f is free.

A graphical representation of this mechanism can be found in Fig. 4.17.

To express this mechanism in Kineverse, the operations $\prec, \succ, \curlywedge$ from Section 3.6.1 are required.

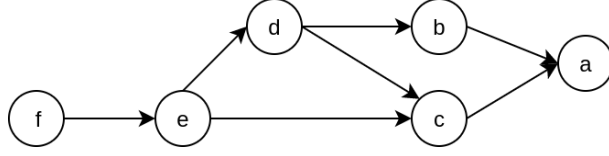The locking mechanism is encoded by constraining the DoFs' velocities.

Figure 4.17 Locking Mechanism. Each arrow is read as *locks*, e.g *f locks e*

$$
\begin{aligned}
-0.4(b \succ 0.4 \curlywedge c \succ 0.6) \le \dot{a} &\le \quad 0.4(b \succ 0.4 \curlywedge c \succ 0.6)\\
-0.1 \le \dot{b} &\le \qquad\qquad\quad 0.1(d \succ 0.8)\\
-0.1 \le \dot{c} &\le \quad 0.1(d \succ 0.5 \curlywedge e \succ 1.0)\\
-0.1 \le \dot{d} &\le \qquad\qquad\quad 0.1(e \succ 0.9)\\
-0.1 \le \dot{e} &\le \qquad\qquad\quad 0.1(f \succ 1.2)\\
-0.4 \le \dot{f} &\le \qquad\qquad\qquad\qquad 0.4
\end{aligned}
\tag{4.12}
$$

At first glance, these constraints can be confusing to understand. Looking at the constraint of $\dot{e}$ for an example, it can be read as follows: *The velocity of e is always greater than* -0.1. *It is at most* 0.1 *if f is in a position greater than* 1.2 *otherwise it is at most* 0. Note that this constraint model encodes both a hard limit on the velocity, and the locking mechanism. In addition to being velocity constrained, the DoF are also constrained in their positions as follows:

$$
\begin{aligned}
-\infty \le a &\le \quad \infty\\
0 \le b &\le \quad 0.7\\
0 \le c &\le \quad 0.8\\
0 \le d &\le \quad 0.9\\
0 \le e &\le \quad 1.1\\
-\infty \le f &\le \quad \infty
\end{aligned}
\tag{4.13}
$$

It should be noted that all locking constraints, with the one of a being the exception, lock their DoF asymmetrically. These asymmetrically locked DoF can always be moved in the negative direction of their configuration space.[2]

These constraints are added to a Kineverse model $\mathcal{A}$ from where they can be queried by the lock opening algorithm.

---

[2]The decreasing value of a DoF's unlocking position is chosen to for aesthetic reasons only.

Lock Opening Algorithm

The input to the algorithm is an articulation model $\mathcal{A}$, a state β which can completely ground the model ($V(\mathcal{A}(\beta)) = \emptyset$), and a set of goal constraints G. The goal of the algorithm is to build a new goal constraint set so that the original problem becomes *movable*. The problem is considered movable, when the velocities of the DoF are not limited in the direction of the goal. If the goal is $a = 4$ and $a(\beta) = 2$ then the problem is considered movable if there is no constraint enforcing $\dot{a} \leq 0$. The constraint set generated by the algorithm can be used to build a Giskard motion problem (cf. Section 3.6.2) which will then generate a trajectory to open the lockbox. The exact definition of the algorithm is laid out in Algorithm 1.

The set of goal constraints is built by iterating over the constraints in the initial goal set G. If all constraints are met, the algorithm is done. If a constraint c is not satisfied, it is determined in which *direction* d the value of the constrained expression e_c needs to be changed to satisfy c. The direction is positive if $e_c(\beta) < \iota_c(\beta)$ and negative if $e_c(\beta) > \upsilon_c(\beta)$. Due to the requirement $\iota_c(\beta) \leq \upsilon_c(\beta)$, the direction can be determined as $d = \text{sgn}(\iota_c(\beta) - e_c(\beta) + \upsilon_c(\beta) - e_c(\beta))$.

To determine, whether e_c can change in direction d, the constraint set $C \subseteq \mathcal{C}$ is queried from $\mathcal{A}$ so that it is the set of all constraints affecting any parameter of gradient ∇e_c: $C = \{c' \mid K(\nabla e_c) \cap V(e_{c'}) \neq \emptyset\}$. For each $x \in K(\nabla e_c)$ a direction d_x needs to be determined in which x needs to change in order for e_c to change in direction d. This is done using the helper function D, which is defined as:

$$D(x) = \text{sgn} \, \nabla_{e_c}^{x}(\beta) \cdot d \tag{4.14}$$

For each x all constraints C_x are collected, which limit x from changing in direction d_x. If there is an x with $C_x = \emptyset$ nothing needs to be done for constraint c since it has at least one open DoF. Otherwise, new goal constraints N are defined, which enforce an opening of the binding limits. This new set is then expanded recursively.

Result

Using the simple conversion strategy described in Section 3.6.2, the constraint set returned by the algorithm can be converted into a Giskard motion problem. The initial goal in the scenario is $a = 1.2$, which is encoded as $G = \{(1.2 \leq a \leq 1.2)\}$. The final set G' is going to be viewed as soft, equally weighted task constraints. As a simplifying assumption, all variables in the constrained expressions of set G' are assumed to be controllable and weighted equally. The hard constraints Z of the motion problem are queried from model $\mathcal{A}$ as the constraints affecting the controlled degrees of freedom.

71

Algorithm 1 Simple Lockbox Exploration

---

function EXPLORECONSTRAINTS($\mathcal{A} = (\mathcal{D}, \mathcal{C}), \beta, G, H$)
    for all $c \in G$ **do**
        $d \leftarrow \text{sgn}(\underline{\iota_c(\beta)} - \underline{e_c(\beta)} + \underline{v_c(\beta)} - \underline{e_c(\beta)})$
        if $d \neq 0$ **then**
            $D \leftarrow D(x) = d \cdot \text{sgn} \, \nabla^x \underline{e_c(\beta)}$ ▷ Directionality function for gradient components.
            $M \leftarrow$ GETLIMITINGCONSTRAINTS($D, K(\nabla e_c), \mathcal{C}$)
            if $\neg\exists(x, \emptyset) \in M$ **then**   ▷ All DoF are blocked
                $N, H \leftarrow$ GENERATESUBGOALS(D, M, H)
                if $N \neq \emptyset$ **then**
                    $G \leftarrow G \cup$ EXPLORECONSTRAINTS($\mathcal{A}, \beta, N, H$)
                else
                    return Problem is unsolvable.
                end if
            end if
        end if
    end for
    return G
end function

function GETLIMITINGCONSTRAINTS($D, X, \beta, \mathcal{C}$)
    $M \leftarrow \emptyset$
    for all $x \in X$ **do**
        $C \leftarrow \{c \mid x = e_c \wedge c \in \mathcal{C}\}$ ▷ Get all constraints directly affecting x
        $M' \leftarrow \emptyset$
        for all $c \in C$ **do**
            if $D(x) < 0 \wedge \underline{\iota_c(\beta)} \geq 0$ **then**   ▷ Desired change is negative and blocked
                $M' \leftarrow M' \cup \{c\}$
            else if $D(x) > 0 \wedge \overline{v_c(\beta)} \leq 0$ **then**   ▷ Desired change is positive and blocked
                $M' \leftarrow M' \cup \{c\}$
            end if
        end for
        $M \leftarrow M \cup \{(x, M')\}$
    end for
    return M
end function

function GENERATESUBGOALS(D, M, H)
    $N \leftarrow \emptyset$
    for all $(x, C) \in M$ **do**
        for all $t \in C$ **do**
            if $D(x) > 0 \wedge V(v_t) \neq \emptyset$ **then**   ▷ Bound is variable
                $N \leftarrow N \cup \{(v_t \prec 0), \infty, v_t)\}$ ▷ Add new goal making u_t positive
            else if $D(x) < 0 \wedge V(l_t) \neq \emptyset$ **then**   ▷ Bound is variable
                $N \leftarrow N \cup \{(-\infty, -(\iota_t \succ 0), \iota_t)\}$ ▷ Add new goal making ι_t negative.
            end if
            $H \leftarrow H \cup \{t\}$ ▷ Add t to touched constraint set
        end for
    end for
    return N, H
end function

The algorithm generates the following Giskard constraints:

$$1.2 - a \leq \dot{a} \qquad\qquad\qquad \leq 1.2 - a \qquad (4.15)$$

$$(f \succ 1.2) \prec 0 \leq 0.1 \cdot \dot{f} \qquad\qquad \leq \infty \qquad (4.16)$$

$$(e \succ 0.9) \prec 0 \leq 0.1 \cdot \dot{e} \qquad\qquad \leq \infty \qquad (4.17)$$

$$(d \succ 0.5) \cdot (e \succ 1.0) \prec 0 \leq 0.1 \cdot (\dot{d} + \dot{e}) \qquad \leq \infty \qquad (4.18)$$

$$(d \succ 0.8) \prec 0 \leq 0.1 \cdot \dot{d} \qquad\qquad \leq \infty \qquad (4.19)$$

$$(b \succ 0.4) \cdot (c \succ 0.6) \prec 0 \leq 0.4 \cdot (\dot{b} + \dot{c}) \qquad \leq \infty \qquad (4.20)$$

The constraints relate to the original problem in the following way: Eq. 4.15 is the original goal, driving a towards 1.2. Eq. 4.16 is the first unlocking goal in the unlocking process. Its goal is to unlock e by moving f. The constrained term is the derivative of the upper bound placed on $\dot{e}$, which needs to be positive to move f towards the position in which it unlocks e. At first glance, the factor of 0.1 scaling the velocity is somewhat puzzling. It originates from the original constraint placed on $\dot{e}$ through which the locking mechanism is encoded. In that original constraint, a hard velocity limit of 0.1 is set, which carries over to the derivative. The left-hand side of the constraint is the driving term, which forces the controlled term, and by extension $\dot{f}$, to be positive. The condition $f \succ 1.2$ is the original unlocking condition from the $\dot{e}$ constraint. The comparison $\prec 0$ serves to invert the truth value of the condition. Read in natural language, the mechanics of this constraint can be read as *While f is not greater* 1.2, $\dot{f}$ *must be positive.*

The remaining constraints are structured the same way, with Eq. 4.17 unlocking d, Eq. 4.18 unlocking c, Eq. 4.19 unlocking b, and Eq. 4.20 unlocking a.

A discrete time stepping integration of the resulting motion problem $f(\mathbf{q})$ is performed until G is satisfied, or for a maximum of 200 iterations. The velocities generated by $f(\mathbf{q})$ are integrated using a time step of $\Delta t = 0.1$.

The generated motion problem successfully opens the lockbox, by sequentially unlocking the individual degrees of freedom, as can be seen in the plot in Fig. 4.18. The approximation of boolean expressions works as expected, switching directly between 0.0 and 1.0.

When looking at the plot there are a couple details worth a small discussion. Most notably, some plots stop sharply, while others stop gradually. This is caused by the positional limits of the DoF. As e and a approach their positional limits, their velocities get limited proportionally to avoid them overshooting the positional bound. A more subtle detail is the overall plot of c. By the design of the experiment, c locks a while its position is below 0.6, however the plot rises up to ~ 0.7 before it stops. This is caused by the definition of the gradient for $\curlywedge$. As stated in 3.19, the gradient $\nabla(a \curlywedge b)$ is the sum of the gradients of the two arguments $\nabla a + \nabla b$. This definition does
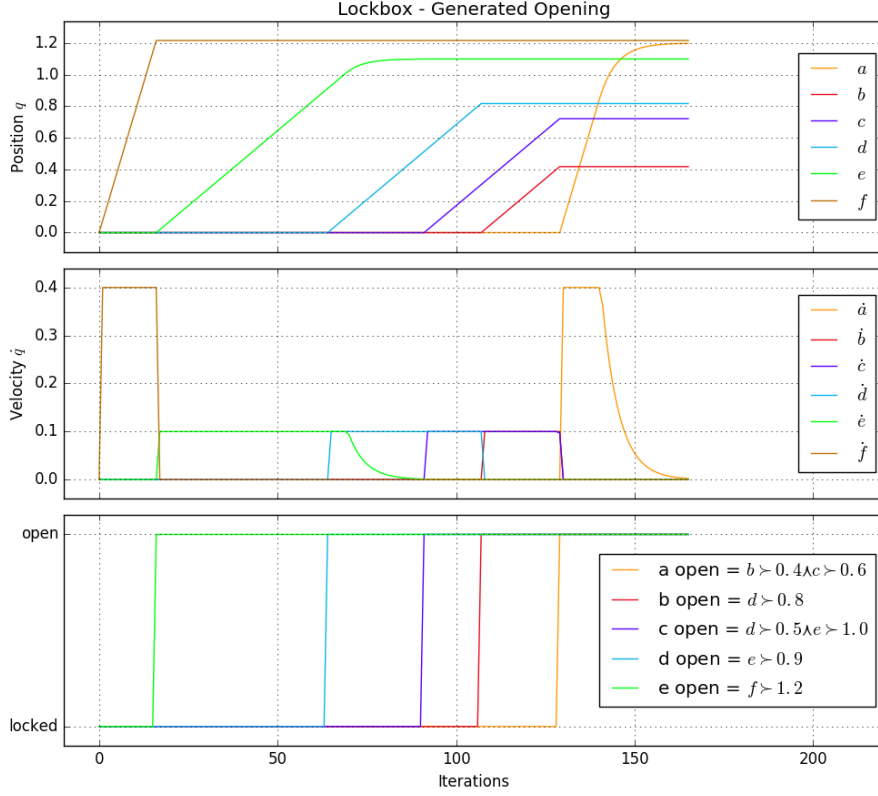
Figure 4.18 Resulting plot from solving the generated control problem. Top: Position of the DoF; Bottom: Value of the locking expression of each DoF.

not cover the dynamics of the conjunction accurately, as it does not express, that an argument has no positive effect on the truth value anymore once it is itself true. There is no simple way of capturing this dynamic because it is asymmetrical. A simple re-definition of $\curlywedge$ integrating the truth value of the two terms:

$$\nabla(a \curlywedge b) = (a \prec 1)\nabla a + (b \prec 1)\nabla b \tag{4.21}$$

does eliminate the effects observed in the plot, however it does not capture that a currently true component can affect the truth value of the overall term negatively, as that term's gradient is simply 0.

While the algorithm works in this setup, it should be noted that it would fail in a scenario which requires a DoF to be in two different positions over the course of the unlocking process. If, for example, d in the given scenario had to be ≥ 0.4 to unlock c but ≤ 0.2 to unlock b, this algorithm would not be guaranteed to produce an unlocking trajectory. This would be the case, because the generated problem does not encode any kind of state about the unlocking process. It only

encodes whether the DoF are unlocked, but does not keep track of the unlocking process as a whole, as in *It is now time to unlock d so b and c can be unlocked.*. Given such an encoding, the driving terms of the problem could change as the unlocking process takes place. Intuitively such an encoding is possible, by expressing the desired sub-goal state as a conjunction, multiplying it with a driving term and summing over all these drivers, however investigating this intuition further is out of the scope of this example.

4.2.2   Simple Configuration Space Tracker

When interacting with an articulation model, it is important to be able to estimate and track the model's current state $\mathbf{q}$. This section describes a simple algorithm tracking the configuration $\mathbf{q}$ of an articulated object based on observations of the parts of its poses in $SE(3)$.

Poses of individual parts of a model can be estimated using markers like AprilTags [36], or increasingly even using non-annotated approaches like PoseCNN [37].

Model

The algorithm requires an articulation model $\mathcal{A}$, a set of frames X from the model and a continuous stream of observations $Y_t \subseteq SE(3)$ of the object's individual parts. It outputs a mapping β minimizing the distance between the projected model state $\underline{X(\beta)}$ and Y_t.

The algorithm formulates the problem of estimating β as an optimization problem. The problem uses two metrics to converge the projected state of the model and the observed state. The first seeks to align the positions $T(x_i)$ of the model parts with their observed positions $T(y_i)$ by minimizing the Euclidean distance d_i:

$$d_i = \|T(x_i) - T(y_i)\| \tag{4.22}$$

The second metric aims at reducing the rotational difference between the projected rotation $R(x_i)$ and the observed rotation $R(y_i)$. There are multiple ways of assessing rotational offset. The most intuitive one might be to measure it as the difference in rotations around the three major axes, however that approach suffers from gimbal lock, a non-continuous configuration space and non-linear convergence. A better approach would be to measure the dot products of the rotation matrices' columns. This metric does not suffer from gimbal lock, or non-linear convergence, however it is difficult to combine the products into a single metric and convergence becomes slow towards the end of the alignment. Even better is to measure the difference between the matrices combined axis-angle representations. This metric has practically no downside, except

for its numerical instability and the rather long expressions it generates. A simple, yet successful metric is to directly assess the difference in the rotation matrices' components:

$$\Delta R_i = \sqrt{\sum_i^3 \sum_j^3 (R(X)_{i,j} - R(Y)_{i,j})^2} \tag{4.23}$$

This metric converges as quickly as the axis-angle metric, but with far less variance in its final result.

The problem is solved iteratively using the Giskard optimization problem formulation. For each $x \in X$ two soft constraints are added, one for each of the two metrics, as shown in Eq.+4.24. The relevant degrees of freedom of the model are identified by the sets of free variables in the two metrics: $V(d_i) \cup V(\Delta R_i)$. Their relevant constraints are queried from $\mathcal{A}$ and added to the problem as velocity constraints while adjusting for the integration factor λ. Note that the actual constraints on velocities are not taken into account. They are left out since this algorithm only operates on a single observation and does not estimate dynamical properties of the model.

$$
\begin{aligned}
-d_1 &\leq \dot{d}_1 & &\leq -d_1 \\
&\vdots & &\vdots \\
d_n &\leq \dot{d}_n & &\leq -d_n \\
-\Delta R_1 &\leq \Delta \dot{R}_i & &\leq -\Delta R_1 \\
&\vdots & &\vdots \\
\Delta R_n &\leq \Delta \dot{R}_n & &\leq -\Delta R_n
\end{aligned}
\tag{4.24}
$$

ROS Integration

The tracker is built to be used in a ROS environment. It uses the Kinverse model client to query the object parts to be tracked from the Kineverse server. The observations are presented to the tracker as an array of stamped poses which are published on a topic. The tracker inserts these poses into the generated optimization problem and iterates on the problem for a maximum of k steps. The result of these iterations is published on another topic as a `ValueMap` message which can be interpreted by the TF publisher described in Section 3.4.6. The number of iterations is fixed to avoid irregular tracking updates. Fig. 4.19 shows the dataflow in the tracking system.
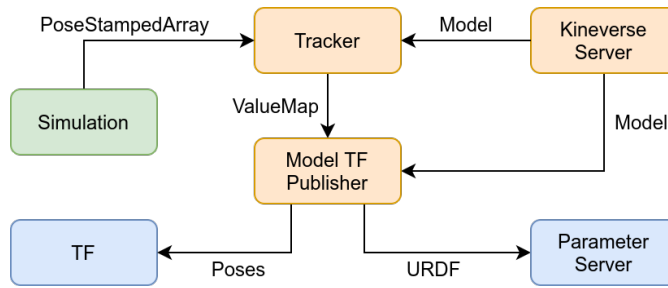
Figure 4.19   Dataflow in the tracking scenario.

Evaluation

The tracker is evaluated on a model of the IAI kitchen[3]. The evaluation is performed in two scenarios. The first scenario evaluates the tracker quantitatively, by sampling configurations for the kitchen, computing the object poses, adding noise and then estimating the original configuration from these noisy poses.

The second evaluation is qualitative. The tracker is run on observation data, generated in a simulated environment [4]. The environment contains a fetch robot in addition to the IAI kitchen. The articulation models of the robot and the kitchen are available on the Kineverse server. Pose observations of the kitchen's parts are generated using a plugin to the simulator. The plugin approximates the field of view of the robot using a view cone which is based on the fetch's horizontal field of view of 54°. If the center of an object is within this cone, an observation for that object is going to be generated.

In both scenarios, noise is added to the observed poses. The noise is assumed to be normally distributed and independent in location and rotation. For both location and rotation a random point is sampled from the surface of a sphere. In the case of the positional noise, the point is interpreted as a translation vector and scaled to a random, normally distributed length with mean 0. In the case of the rotational noise, the point marks the axis of rotation. The angle is, once again, determined by sampling it from a normal distribution with mean 0.

In the second scenario, the deviation of the noise σ is determined using the distance d of an object to the camera. A quadratic function is chosen to model the noise increases with increasing distance:

$$\sigma = 2^{\alpha d} - 1 \tag{4.25}$$

This scaling behavior is added to mimic the increasing noise in depth camera data. The obser-

---

[3]IAI kitchen: *https://github.com/code-iai/iai_maps*
[4]IAI Bullet Sim: *https://github.com/ARoefer/iai_Bullet_sim*
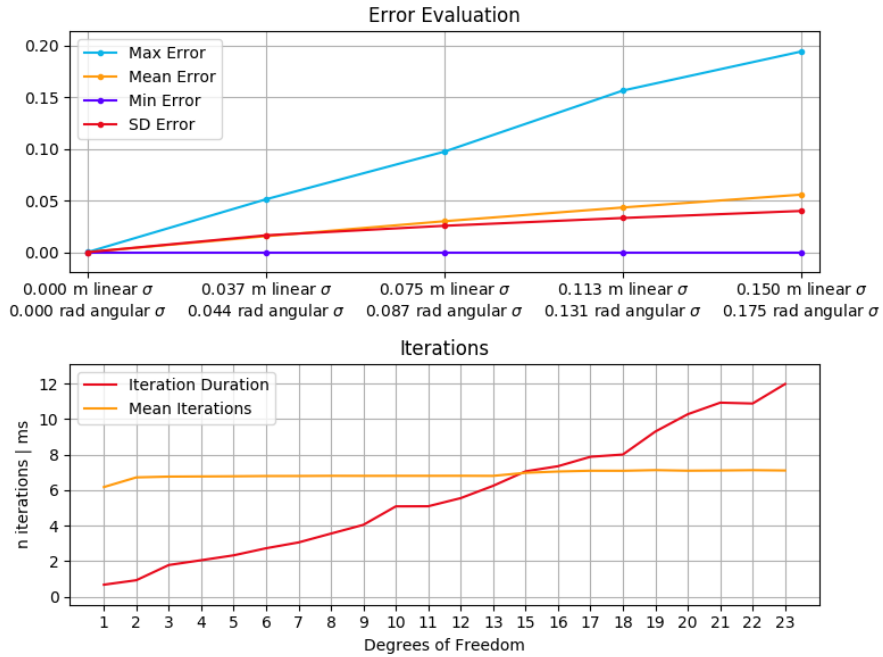
Figure 4.20   Plots summarizing the results from the quantitative evaluation of the tracker. First row: Final error at differing linear and angular noise tracking 23 DoF from observing 60 poses; Second row: Average number of iterations and average iteration duration across all noise levels as a function of DoF.

vations are sent out on a ROS topic as an array of stamped poses.

Results - Quantitative Evaluation

All quantitative experiments are performed on 200 randomly sampled configurations, tracking up to 60 object poses with up to 23 degrees of freedom. The algorithm employs an iterative gradient descent scheme. The iterations are bounded to a maximum of 10 per configuration and the iteration is stopped early if no bound of the optimization problem is violated by more than 0.001. The experiments are executed using a single thread on a desktop style computer with an Intel Core i7-6700, 16GB of memory, running Ubuntu 18.04 and ROS Melodic.

The algorithm is evaluated with an increasing number of poses and DoF to track. For each pose set, varying noise on the linear and angular parts of the observed poses are applied. Over the course of the evaluation the average error in the final configuration is tracked as a measure of accuracy of the final result. Further, the error deviation across the 200 samples is tracked as a measure of reliability, an the maximum final error is recorded to see how bad the worst outliers are. Since the tracker is ideally an online component, the average number of iterations until
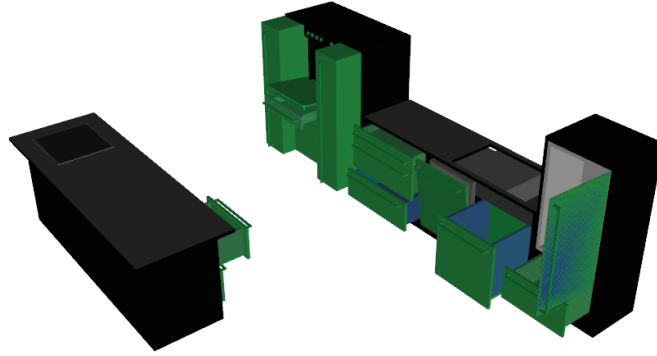
Figure 4.21   Result of tracking algorithm with no noise added to the observations. Green: Ground truth; Blue: Tracked configuration. Since they are so close, they are almost indistinguishable.

convergence and average duration of an iteration are recorded as well. The results from largest experiment configuration, containing the full 60 poses and 23 DoF, collected in Table 4.2. To give a better feel for the trend in the data, the results are also presented graphically in Fig. 4.20. Fig. 4.20 also plots the mean number of iterations per sample and the mean duration of an iteration as a function of DoF.

To prove that the implemented algorithm works under ideal circumstances, it is first tested without adding any noise to the poses computed from the random configurations. As can be seen in Table 4.2, the average final error is negligible with no significant spread or outliers. The iterative process ends early, after an average of 3.65 iterations. The average duration of a step in the iteration is 10 ms, which makes for an average update frequency of 27.4 Hz given the average number of iterations. It is fair to conclude, that the algorithm itself is sound and works as expected. Fig. 4.21 shows an example result of a tracked configuration.

The algorithm is then evaluated under increasing noise in the observation. The noise in position and rotation increases linearly in 4 steps up to a deviation of 0.15 m in position and 0.175 $rad \equiv$ 10 deg in rotation. Fig. 4.22 shows the effect of 0.1 m linear and 10 deg angular noise on the

Linear σ	Angular σ	$\varnothing$ Error	σ Error	Max Error	$\varnothing$ Iterations	Iteration Duration
0.0 m	0.0 rad	2×10^{-5}	2×10^{-5}	1×10^{-4}	3.65	13.0 ms
0.038 m	0.044 rad	0.015	0.016	0.059	6.82	14.8 ms
0.075 m	0.087 rad	0.030	0.026	0.107	7.88	15.3 ms
0.113 m	0.130 rad	0.044	0.034	0.154	8.41	15.6 ms
0.150 m	0.175 rad	0.057	0.041	0.190	8.73	15.7 ms

Table 4.2   Performance of tracking algorithm given different linear and angular noise.
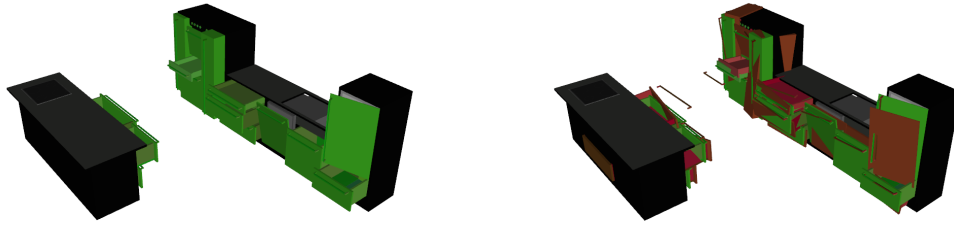
Figure 4.22   Tracking challenge. Left: Ground truth configuration; Right: Poses of objects with added noise.

poses.

As the noise increases, the average final error increases linearly up to 0.057. Note that this error is unit-less as it is measured in the configuration space of the model. While the other metrics increase as well, they do so sublinearly, as can be observed in Fig. 4.20. Even though this is the case, the tracker would not perform well at tracking many moving objects at higher levels of noise. Given the final average iteration duration of 13.0 ms and an average of 8.73 iterations, the tracker would produce updates at only 8.84 Hz. While the this update frequency is very high, the experiment with a varying number of DoF suggests that higher update rates are possible with fewer number of DoF. Looking at Fig. 4.20 again, tracking of 10 DoF seems to be possible at a frequency of up to 38 Hz. Another consideration to take into account when estimating the real-world impact of these results is the aggressive noise model. The noise model that is applied to the poses, does not take the geometry of the objects into account which is a major source of noise in real observations. Overall this should inspire confidence, that the tracker will not perform worse with real observations than it does with the generated ones.

Results - Qualitative Evaluation

The qualitative experiments are performed under different noise levels, much like the quantitative experiments. The experiments are recorded on video, which is then used to assess the tracker's performance when tasked with tracking a model's configuration over time.

The tracking task in this experiment is simpler than it was in the quantitative experiment. Only 12 poses are tracked and the maximum number of iterations is limited to 3 per update. In all experiments these parameters yield a stable update rate of 28 Hz. Each iteration is initialized with the last computed configuration $\mathbf{q}_{t-1}$. The noise is controlled by the factor α in Eq. 4.25. Table 4.3 displays the resulting deviation at different distances from the camera.

The recorded video can be found on YouTube[5], Fig. 4.23 gives an impression of the setup of the

---

[5]Tracker behavior with simulated observations: *https://www.youtube.com/watch?v=__cIsLP-mdvw*
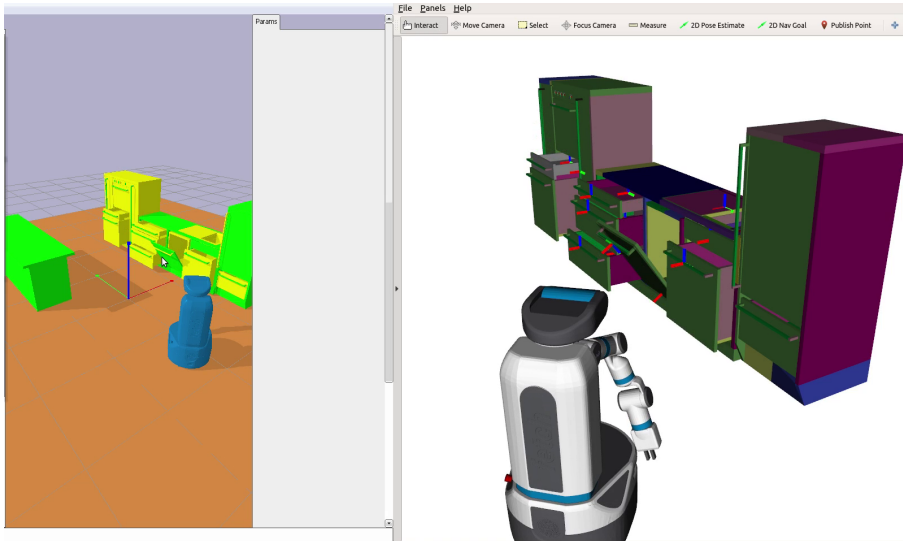
Figure 4.23   Screenshot of the tracker evaluation using simulated observations. Left: Simulated environment; Right: Tracking result.

evaluation. Without any noise added, the tracker performs perfectly, as was to be expected given the quantitative evaluation. There seems to be a slight delay in the updates, which is likely to be caused by the fixed 3 iterations. As the noise increases, it becomes very apparent, that this tracking model lacks a damping term to suppress the worst outliers of the noise. At a noise level of $\alpha = 0.01$ the configuration merely seems somewhat jittery, at a level of $\alpha = 0.03$ the jittering becomes so bad that even objects close to the robot start jumping by multiple centimeters. At a level of $\alpha = 0.06$ the jittering of especially distant objects becomes so strong that it looks like the configuration of the kitchen is determined completely randomly.

All in all this evaluation does show that it is possible to integrate the tracking algorithm in a real-time system, be it on a smaller problem space. There is room for enlarging the problem space while maintaining real-time capability. One way of achieving this could be a very selective update strategy, which only tracks the part of the model based on the objects that are currently in view. In the current implementation no optimizations of this kind exist, forcing the solver to always solve for all DoF, even if there is no new data that affects them.

α	0.0	0.01	0.03	0.06
σ at $d = 1\ m$	0.0	0.0069	0.021	0.04
σ at $d = 2\ m$	0.0	0.014	0.043	0.09
σ at $d = 3\ m$	0.0	0.021	0.064	0.13

Table 4.3   Noise deviation at different distances and α.

4.2.3   Interacting with Articulated Objects - Simple Pushing

In the previous example I applied Kineverse to a perception task and evaluated its performance. In this example, I will use Kineverse to solve a manipulation task which requires the interaction with an articulated object - the IAI kitchen.

The task is simple: The robot needs to keep the kitchen in an orderly state. Using the configuration tracker from the previous section, the robot is able to perceive the current pose of the kitchen. The kitchen is considered to be "in order" when its configuration is $\mathbf{q} = \mathbf{0}$. The robot exploits the fact that it is standing in front of the kitchen and uses pushing to close doors and drawers.

Model

As stated above, the robot is using a pushing strategy to fulfill its task of keeping the state of the kitchen at $\mathbf{0}$. First and foremost, a model of pushing is required to do so.

A fundamental challenge in manipulating an articulated object is the fact, that such an interaction is either causing a kinematic loop, or requires a restructuring of the kinematic tree to avoid such a loop. Restructuring the tree to attach the object being manipulated at the end of the robot's endeffector is only a solution in scenarios in which the object is freely movable. If it is not, constraints are going to be needed to enforce the constrained motion of the object. Given that constraints are going to be needed in most cases, the restructuring can be skipped and constraints can be used to synchronize the motions of the endeffector and the manipulated object directly. Fig. 4.24 the different models that result from applying either strategy in the push scenario.

The goal is specified with respect to the configuration of the kitchen and not connected to the state of the robot. Because of this specification, no gradient towards the robot's DoF can be generated. To still create motions solving the task, a trick is employed: The DoF of the articulated object are added as controllable variables to the motion problem. The constraints connecting the robot's endeffector and the articulated object have to be chosen so that the motion the robot makes can be used to produce the motion the object makes.

The constraints modeling the push require an actuated point $\mathbf{r}$, a point on the object $\mathbf{p}$ and the contact normal $\mathbf{n}$ pointing from the object's surface towards $\mathbf{r}$. The points $\mathbf{r}, \mathbf{p}$ are the closest points on the robot's endeffector and object respectively. They and the contact normal can be queried from the Bullet physics backend. The points and the vector are visualized in Fig. 4.25.

The first constraint enforces the general logic of a push: It cannot pull. This is done by limiting the dot product of the velocity vector $\dot{\mathbf{p}}$ and the contact normal $\mathbf{n}$ to be at most 0:
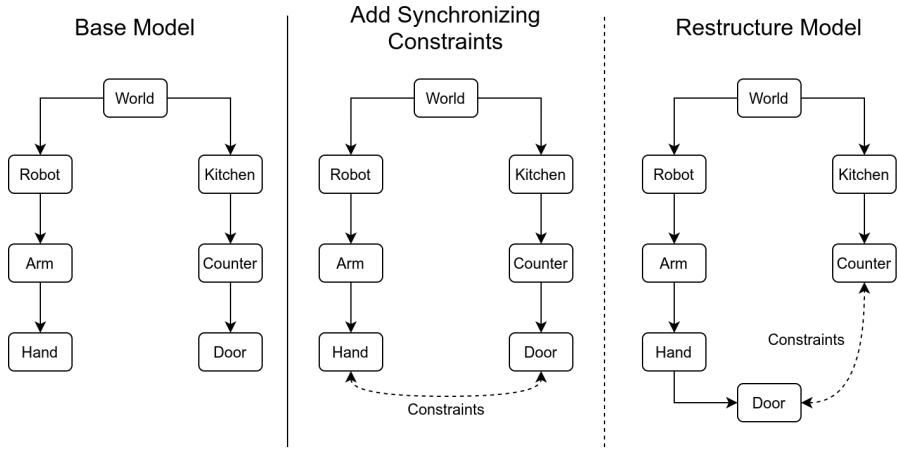
Figure 4.24   Two different strategies for modeling the kinematics of a push in an existing articulation model. Solid arrow: Forward kinematic connection; Dashed arrow: Constraint on relative pose of two objects.
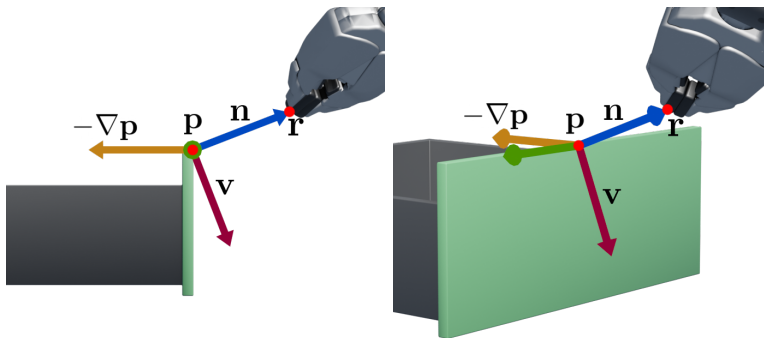

Figure 4.25   Points and vectors used to generate motions for a push. $\mathbf{r}$ (red): Contact point on the robot; $\mathbf{p}$ (red): Contact point on the object; $\mathbf{n}$ (blue): Contact normal; $-\nabla\mathbf{p}$ (yellow): Negative gradient of $\mathbf{p}$; Green arrow: neutral tangent; $\mathbf{v}$ (dark red): Active tangent.

$$-\infty \leq \dot{\mathbf{p}} \cdot \mathbf{n} \ \leq 0 \tag{4.26}$$

The second constraint enforces the impenetrability of the robot and the object, by constraining the signed distance $d = (\mathbf{r} - \mathbf{p}) \cdot \mathbf{n}$ between the two bodies:

$$0 \leq d \ \ \leq \infty \tag{4.27}$$

Lastly, a constraint for the velocity of every $x \in V(\mathbf{p})$ is added to keep the object from moving without any contact being made. This is done using the $\prec$ function to compare d to a tiny threshold value ϵ:

$$
\begin{aligned}
-\infty(d \prec \epsilon) \leq \dot{x}_0 \ &\leq \infty(d \prec \epsilon) \\
\vdots \qquad\qquad &\qquad \vdots \\
-\infty(d \prec \epsilon) \leq \dot{x}_n \ &\leq \infty(d \prec \epsilon)
\end{aligned}
\tag{4.28}
$$

These constraints are combined with two task constraints. The first is the configuration goal for the object, which is $\mathbf{q} = \mathbf{0}$. The second seeks to make contact with the object to enable actually affecting change in the configuration:

$$
\begin{aligned}
0 \leq \mathbf{q} \ &\leq 0 \\
0 \leq d \ &\leq 0
\end{aligned}
\tag{4.29}
$$

When these constraints are converted to Giskard constraints, their bounds are expanded to include an integral term, to avoid persistent state error which would keep the objects from making contact, or reaching their goal.

One other goal constraint is added to the motion problem. The original constraints specify a simple model of a push, however they do not help to move the point of contact to a location from which the push can actually be performed. Fundamentally, the contact normal and the push direction have to be aligned, which would be easy if a gradient for the surface normal existed. Since the normal is queried from the physics backend this is not the case. As a work-around, a heuristic is used to navigate around the object's geometry to a point where the normal of the desired motion and the contact normal align. The heuristic consists of a displacement vector $\mathbf{v}$ and a distance metric d_x which are described in Eq. 4.30. The distance metric d_x replaces d in the task constraint, which drives the robot to make contact with the object. It measures the distance between the actuated point $\mathbf{r}$ and a point which is offset from $\mathbf{p}$ along $\mathbf{v}$. The offsetting vector $\mathbf{v}$ lies in the plane described by the contact normal $\mathbf{n}$ and points in the direction which most strongly decreases the dot product of the actuated point and the direction towards the goal

of $\mathbf{q} = 0$. The definition of $\mathbf{v}$ exploits domain knowledge by assuming, that the goal is always in the negative direction of the gradient. The offset $\mathbf{v}$ is scaled based on the cross product of the contact normal and the gradient $\nabla \mathbf{p}$. This is one of three scaling model that were tried and the most successful one. It drives the motion of $\mathbf{r}$ around the object, while not diverting it too much, while $\mathbf{r}$ is far away.

$$
\begin{aligned}
\mathbf{v} &= (-\nabla \mathbf{p} \times \mathbf{n}) \times \mathbf{n} \\
d_x &= \|\mathbf{p} + \mathbf{v} \cdot \| -\nabla \mathbf{p} \times \mathbf{n}\| - \mathbf{r}\|
\end{aligned}
\tag{4.30}
$$

For practical applications a third constraint is added to keep the robot looking at the object so it can keep tracking it during the interaction.

Results

The motion generation is evaluated by using it to close doors and drawers in the IAI kitchen. 12 parts of the kitchen are interacted with - 9 drawers, 3 doors two of which open vertically. Every object's configuration is set to 0.4 at the beginning of the experiment, while the remaining kitchen's configuration is set to 0. The robot always starts in the same pose at the same location which is centrally located approximately 1.2 m away from the kitchen. The experiment is run with both a PR2 and a fetch. For both robots, the link used to make contact with the object is specified. In the case of the fetch, two modes are explored: In the first, the fetch is equipped with an omnidirectional base like the one the PR2 has, in the second mode, the fetch is equipped with the differential drive that it is manufactured with in real life. These modes are added to isolate whether any failures on the fetch are due to the limited kinematics of fetch's base, or whether they just do not map easily to another robot.

The motion generation is performed using an integration step factor 0.1 and a maximum of 500 iterations. Over the course of the generation, iteration duration is tracked to determine whether the built model can be evaluated fast enough to be integrated in an online control loop.

The recorded videos can be found on the attached CD, or online[6][7][8]. Fig. 4.26 shows a generated motion as a series of images. The iteration times are collected in Table 4.4.

In the case of the robots with an omnidirectional base, the motions look plausible and transfer well between the two different robots. The footage from the planned trajectories shows some jitter in the position of the robot endeffector, when it and the target object are in contact. The jitter is caused by the geometry of the target object, which is always a handle in this scenario.

---

[6]Fetch pushing with omni base: *https://youtu.be/hAuz7nMFNrA*
[7]Fetch pushing with differential drive: *https://youtu.be/WlgcCcJwoBA*
[8]PR2 pushing: *https://youtu.be/q77DOFXee-o*
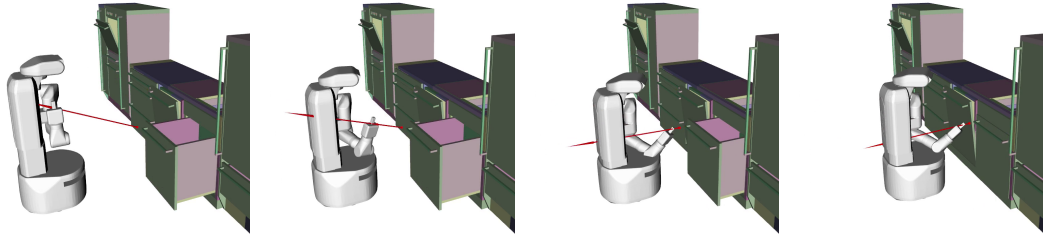
Figure 4.26 Planned motion of the fetch robot closing a drawer. The red arrow marks the closest direction between drawer and robot. Left: Initial pose; Center Left: Robot approaches closest point on object's geometry; Center Right: Contact is made, enabling the use of the drawer's DoF in planning; Right: The drawer is closed, the goal satisfied.

The geometry of the handles is very narrow, which leads to large changes in the contact normal when over correcting the endeffector's position slightly.

The evaluation using the fetch with the differential drive shows that a different navigation function is needed for this kind of base. While the motion problem does work to solve the task, the solutions become less stable towards the far end of the kitchen.

The performance evaluation in Table 4.4 shows that the motion generation can be run at a stable $63 - 74\ Hz$ - depending on the robot. This makes the motion problem suitable for online, closed-loop control.

4.2.4   Full Kitchen Pushing System

In a final evaluation, I combine the tracker from Section 4.2.2 and the pushing motion from Section 4.2.3 in a simple behavior which I deploy in a simulated environment to monitor the kitchen and restore it to order when any drawer or compartment is left open.

	⌀ Iteration Duration	σ Iteration Duration
Fetch w. differential drive (18 DoF)	$14.3\ ms$	$0.77\ ms$
Fetch w. omni-base (19 DoF)	$13.5\ ms$	$0.28\ ms$
PR2 (49 DoF)	$15.7\ ms$	$0.71\ ms$

Table 4.4 Average iteration duration and deviation in the duration in the kitchen pushing task, using different robots.
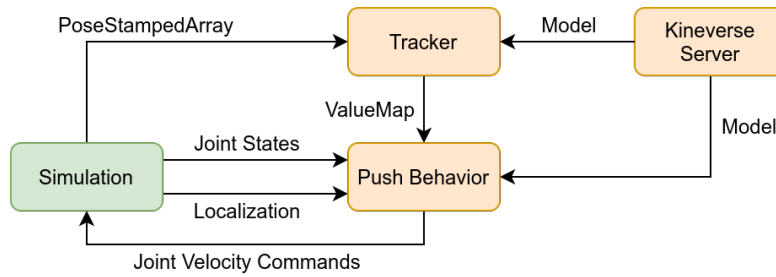
Figure 4.27   Data flow in the combined tracking and pushing system.
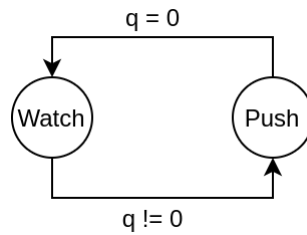
Figure 4.28   State machine of pushing behavior.

System Setup

The system used in the evaluation consists of four components: A Kineverse server, a tracker, a push behavior and a simulation. The Kineverse server provides the articulation models of the kitchen and the robot to the tracker and the push behavior. As in the evaluation in Section 4.2.2, the tracker receives simulated observations and determines the configuration $\mathbf{q}$ of the kitchen. This configuration is sent to the push behavior which combines it with simulated robot state data to compute a motion command for the robot, which it sends to the simulator. The complete data flow is depicted in Fig. 4.27. All these four components are implemented as nodes in a ROS system which communicate over ROS topics.

Behavior

The pushing algorithm is embedded in a behavior consisting of two states. The first simply monitors the value of $\mathbf{q}$. When $\mathbf{q} \neq \mathbf{0}$ the machine transitions to the second state, which generates a pushing problem for the first variable $x \in \mathbf{q}$ which is $x \neq 0$ and then executes the pushing motion until $x = 0$. The pushing problem is generated by querying the articulation model $\mathcal{A}$ for the set G of geometry that is affected by x. The pushing problem is then generated between the endeffector and the DoF x with G being used to compute contact normal and closest point. A practical challenge is created by the Bullet physics engine which is used to compute these points: The closest points are not exactly on the surface of the objects, when the objects are meshes. This

inaccuracy only seems to affect this particular calculation, but not the actual collision detection during the physics calculation. The mismatch in distance calculation and collision detection leads to the motion controller assuming that it has made contact with an object when it actually has not. To combat this, I move the contact point back along the contact normal by 8 *cm*.

To get the robot to look around, I added an additional constraint which makes its gaze wander up and down the kitchen. This behavior is hard-coded and uses neither the articulation nor any other kind of data available in the scenario.

Results

As with the previous test, this experiment is also captured on video and evaluated based on the recording. The evaluation starts with the robot standing at the origin of the scene, sweeping the kitchen with its gaze. To trigger the manipulation, I open a drawer or door of the kitchen by dragging it open in the simulation. If the robots fails at closing the opened object or gets stuck doing so, I close the object manually, which returns the robot to its idle state. The video of the experiment can be found on the CD attached to this thesis, or on online[9].

The experiment shows that it is possible to combine the object tracker and the simple pushing motion in a system to solve this task of keeping the simulated kitchen in order. It also reveals a problem in that is caused by the lack of noise suppression in the tracker and another problem that is caused by the mismatch in Bullet's distance calculations and the collision detection. The lack of noise suppression makes it impossible to run the experiment with any kind of noise added to the perception. The noise in the tracked configuration leads the behavior to constantly believe that some part of the kitchen has been left open and needs fixing. The calculation mismatch in Bullet and the simple fix for, combined with the navigation heuristic introduced in Eq. 4.30, lead to strong oscillating motions in the robot's endeffector in some interactions. When looking at the video, one can observe the contact normal rapidly changing direction, as the robot gets close to the object it is trying to push. This is followed by a counter motion, which tries to navigate around the object for a better point to push at according to this normal.

---

[9]PR2 pushing objects in simulated kitchen environment: *https://youtu.be/95TQbioTAog*

Chapter 5

Conclusion

In this thesis I set out to investigate the feasibility of building an articulation model framework which uses symbolic mathematical expressions to encode models directly as their mathematical mechanisms. I argued for this approach on the basis of two shortcomings I see in existing, descriptive frameworks. As mobile manipulators become more and more ubiquitous and interact with new environments, they will need the ability to incorporate newly observed articulation models into their behaviors. The existing frameworks for describing articulation models take an abstract, descriptive approach. Each of these descriptions need to be translated into an internal representation, which invites implementation errors and ambiguous interpretations. At the same time, this abstract description causes a lot of implementation overhead when the description language is extended, as a human developer will have to integrate the language update into the manipulator's software. Further, I stated that future articulation models needed to be dynamic, to adapt to observations that were made by mobile manipulators as they interact with their environment and should support being exchanged in a networked system.

I posited that the problem of implementation overhead would be solved in a new framework if that framework enabled its users to implement algorithms which use articulation models but are agnostic about the articulations actually present in the model.

In pursuit of answering the primary question of feasibility, I first proposed a formal model in this thesis which represents an articulation model as a tree of expressions with free variables and a set of constraints. As an extension to regular mathematical expressions I introduced the concept of the *Augmented Gradient* which maintains information about with respect to which variables a term was differentiated and allows for additional derivative expressions which enables the model to cover both positional and differential kinematics. To avoid enforcing a linear kinematic structure, I describe the logic behind an articulation model as a sequence of operations that build up the expressions of the model incrementally. I implemented the concept as a ROS package called *Kineverse* with a special focus on its integration into the ROS runtime environment.

In my evaluation I was able to demonstrate that the modeling capabilities of Kineverse can rival

those of URDF, the most widely used articulation model framework in the ROS ecosystem. I demonstrated further that it is possible to implement algorithms which operate successfully on a kineverse model without any further knowledge of the exact articulations. According to the supposition I made before, this is an indicator that the articulation model would be able to scale when extended, without causing implementation overhead.

As it stands, the model has been applied successfully to rudimentary object tracking and motion generation tasks. Its applicability in articulation model estimation has not been evaluated yet. In future work, the model should be applied that problem domain and the integration with more perception and manipulation tasks in general should be attempted.

From a practical perspective I view Kineverse as a successful proof-of-concept. The task-oriented evaluations have shown that it is possible to utilize the implementation in practical applications. From the experience I gained when realizing the tasks practically using Kineverse, I must say that I like the implementation of the articulation model. Using the models is easy and the query functions are helpful when collecting the relevant constraints and geometry for the task. The operations' API, on the other hand, is somewhat cumbersome to use. Adding new operations is not as easy as I would have liked it to be. Improvements to the API would help Kineverse's adoptability.

Publication of the Software

The *Kineverse* software, as well as the code for the experiments is hosted on GitHub. Both repositories are equipped with readme files containing installation instructions and instructions to reproduce the experiments. It is provided under the `MIT` license, allowing non-commercial and commercial use of the software, as well as redistribution and modification. While the code may develop further over time, the final version of the software for this thesis is stored on the branch `masters_thesis` of both repositories. A copy of the software is also included on the CD that is submitted with this thesis.

Package	Repository
`kineverse`	*https://github.com/ARoefer/kineverse.git*
`kineverse_experiment_world`	*https://github.com/ARoefer/kineverse_experiment_world.git*

Appendix A

Appendix

A.1 List of Tables

A.2 List of Figures

A.3   Bibliography

[1]   Erwin Aertbeliën and Joris De Schutter. "eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs". In: *Intelligent Robots and Systems (IROS), 2014 IEEE/RSJ International Conference on.* IEEE. 2014, pp. 1540–1546.

[2]   Remi Arnaud and Mark C Barnes. *COLLADA: sailing the gulf of 3D digital content creation.* AK Peters/CRC Press, 2006.

[3]   Manuel Baum, Matthew Bernstein, Roberto Martin-Martin, Sebastian Höfer, Johannes Kulick, Marc Toussaint, Alex Kacelnik, and Oliver Brock. "Opening a lockbox through physical exploration". In: *Humanoid Robotics (Humanoids), 2017 IEEE-RAS International Conference on.* IEEE. 2017.

[4]   Felix Burget, Armin Hornung, and Maren Bennewitz. "Whole-body motion planning for manipulation of articulated objects". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on.* IEEE. 2013.

[5]   Sachin Chitta, Benjamin Cohen, and Maxim Likhachev. "Planning for autonomous door opening with a mobile manipulator". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on.* IEEE. 2010.

[6]   Andrew Comport, Eric Marchand, and Francois Chaumette. "Object-based visual 3D tracking of articulated objects via kinematic sets". In: *In IEEE Workshop on Articulated and Non-Rigid Motion.* 2004, pp. 2–9.

[7]   Karthik Desingh, Shiyang Lu, Anthony Opipari, and Odest Chadwicke Jenkins. "Efficient nonparametric belief propagation for pose estimation and manipulation of articulated objects". In: *Science Robotics* 4.30 (2019).

[8]   Tom Drummond and Roberto Cipolla. "Real-time visual tracking of complex structures". In: *IEEE Transactions on pattern analysis and machine intelligence* 24.7 (2002), pp. 932–946.

[9]   Zhou Fang, Georg Bartels, and Michael Beetz. "Learning Models for Constraint-Based Motion Parameterization from Interactive Physics-based Simulation". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on.* 2016.

[10]   Karol Hausman, Scott Niekum, Sarah Osentoski, and Gaurav S Sukhatme. "Active articulation model estimation through interactive perception". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on.* IEEE. 2015, pp. 3305–3312.

[11]   Sebastian Höfer and Oliver Brock. "Coupled learning of action parameters and forward models for manipulation". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on.* IEEE. 2016, pp. 3893–3899.

[12]   Sebastian Höfer, Tobias Lang, and Oliver Brock. "Extracting kinematic background knowledge from interactions using task-sensitive relational learning". In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on.* IEEE. 2014, pp. 4342–4347.

[13]   Advait Jain and Charles C Kemp. "Pulling open doors and drawers: Coordinating an omni-directional base and a compliant arm with equilibrium point control". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on.* IEEE. 2010, pp. 1807–1814.

[14]   Advait Jain and Charles C Kemp. "Pulling open novel doors and drawers with equilibrium point control". In: *Humanoid Robotics (Humanoids), 2009 IEEE-RAS International Conference on.* IEEE. 2009, pp. 498–505.

[15]   Dov Katz and Oliver Brock. "Manipulating articulated objects with interactive perception". In: *Robotics and Automation (ICRA), 2008 IEEE International Conference on.* IEEE. 2008, pp. 272–277.

[16]   Dov Katz, Yuri Pyuro, and Oliver Brock. "Learning to manipulate articulated objects in unstructured environments using a grounded relational representation". In: *Robotics: Science and Systems IV* (2009), p. 254.

[17]   Johannes Kulick, Stefan Otte, and Marc Toussaint. "Active exploration of joint dependency structures". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on.* IEEE. 2015, pp. 2598–2604.

[18]   S. M. LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[19]   Roberto Martin Martin and Oliver Brock. "Online interactive perception of articulated objects with multi-level recursive estimation based on task-specific priors". In: *Intelligent Robots and Systems (IROS), 2014 IEEE/RSJ International Conference on.* IEEE. 2014, pp. 2494–2501.

[20]   Roberto Martín-Martín and Oliver Brock. "Cross-modal interpretation of multi-modal sensor streams in interactive perception based on coupled recursion". In: *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on.* IEEE. 2017, pp. 3289–3295.

[21]   Roberto Martín-Martín, Sebastian Höfer, and Oliver Brock. "An integrated approach to visual perception of articulated objects". In: *Robotics and Automation (ICRA), 2016 IEEE International Conference on.* IEEE. 2016, pp. 5091–5097.

[22]   Chaitanya Mitash, Abdeslam Boularias, and Kostas E Bekris. "Improving 6d pose estimation of objects in clutter via physics-aware monte carlo tree search". In: *Robotics and Automation (ICRA), 2018 IEEE International Conference on.* IEEE. 2018, pp. 1–8.

[23]   Stefan Otte, Johannes Kulick, Marc Toussaint, and Oliver Brock. "Entropy-based strategies for physical exploration of the environment's degrees of freedom". In: *Intelligent Robots and Systems (IROS), 2014 IEEE/RSJ International Conference on.* IEEE. 2014, pp. 615–622.

[24]   Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software.* Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[25]   Nathan Ratliff, Marc Toussaint, and Stefan Schaal. "Understanding the geometry of workspace obstacles in motion optimization". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on.* IEEE. 2015, pp. 4202–4209.

[26] Nathan Ratliff, Matt Zucker, J Andrew Bagnell, and Siddhartha Srinivasa. "CHOMP: Gradient optimization techniques for efficient motion planning". In: *Robotics and Automation (ICRA), 2009 IEEE International Conference on.* IEEE. 2009, pp. 489–494.

[27] Thomas Rühr, Jürgen Sturm, Dejan Pangercic, Michael Beetz, and Daniel Cremers. "A generalized framework for opening doors and drawers in kitchen environments". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on.* IEEE. 2012, pp. 3852–3858.

[28] John Schulman, Jonathan Ho, Alex X Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. "Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization." In: *Robotics: science and systems.* Vol. 9. 1. Citeseer. 2013, pp. 1–10.

[29] Leonid Sigal, Sidharth Bhatia, Stefan Roth, Michael J Black, and Michael Isard. "Tracking loose-limbed people". In: *Computer Vision and Pattern Recognition (CVPR), 2004 IEEE Computer Society Conference on.* Vol. 1. IEEE. 2004, pp. I–I.

[30] Jürgen Sturm, Advait Jain, Cyrill Stachniss, Charles C Kemp, and Wolfram Burgard. "Operating articulated objects based on experience". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on.* IEEE. 2010, pp. 2739–2744.

[31] Jürgen Sturm, Cyrill Stachniss, and Wolfram Burgard. "A probabilistic framework for learning kinematic models of articulated objects". In: *Journal of Artificial Intelligence Research* 41 (2011), pp. 477–526.

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

[33] Emanuel Todorov, Tom Erez, and Yuval Tassa. "Mujoco: A physics engine for model-based control". In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on.* IEEE. 2012, pp. 5026–5033.

[34] Marc Toussaint. "Logic-geometric programming: An optimization-based approach to combined task and motion planning". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence.* 2015.

[35] Marc Toussaint, Kelsey Allen, Kevin A Smith, and Joshua B Tenenbaum. "Differentiable Physics and Stable Modes for Tool-Use and Manipulation Planning." In: *Robotics: Science and Systems (RSS).* 2018.

[36] John Wang and Edwin Olson. "AprilTag 2: Efficient and robust fiducial detection". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on.* 2016.

[37] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. "PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes". In: *Robotics: Science and Systems (RSS), Proceedings of.* 2018.