Constrained Manipulation Planning

Dmitry Berenson

CMU-RI-TR-11-08

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Robotics.

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

June 20th, 2011

Thesis Committee:
Siddhartha Srinivasa, CMU RI, Co-chair
James Kuffner, CMU RI, Co-chair
Matthew Mason, CMU RI
Thierry Siméon, LAAS-CNRS, France

Abstract

Every planning problem in robotics involves constraints. Whether the robot must avoid collision or joint limits, there are always states that are not permissible. Some constraints are straightforward to satisfy while others can be so stringent that feasible states are very difficult to find. What makes planning with constraints challenging is that, for many constraints, it is impossible or impractical to provide the planning algorithm with the allowed states explicitly; it must discover these states as it plans. The goal of this thesis is to develop a framework for representing and exploring feasible states in the context of manipulation planning.

Planning for manipulation gives rise to a rich variety of tasks that include constraints on collision-avoidance, torque, balance, closed-chain kinematics, and end-effector pose. While many researchers have developed representations and strategies to plan with a specific constraint, the goal of this thesis is to develop a broad representation of constraints on a robot's configuration and identify general strategies to manage these constraints during the planning process. Some of the most important constraints in manipulation planning are functions of the pose of the manipulator's end-effector, so we devote a large part of this thesis to end-effector placement for grasping and transport tasks. We present an efficient approach to generating paths that uses Task Space Regions (TSRs) to specify manipulation tasks which involve end-effector pose goals and/or path constraints. We show how to use TSRs for path planning using the Constrained BiDirectional RRT (CBiRRT2) algorithm and describe several extensions of the TSR representation. Among them are methods to plan with object pose uncertainty, find optimal base placements, and handle more complex pose constraints by chaining TSRs together. We also explore the problem of automatically generating end-effector pose constraints for grasping tasks and present two grasp synthesis algorithms that can generate lists of grasps in extremely cluttered environments. We then describe how to convert these lists of grasps to TSRs so they can be used with CBiRRT2.

We have applied our framework to a wide range of problems for several robots, both in simulation and in the real world. These problems include grasping in cluttered environments, lifting heavy objects, two-armed manipulation, and opening doors, to name a few. These example problems demonstrate our framework's practicality, and our proof of probabilistic completeness gives our approach a theoretical foundation.

In addition to the above framework, we have also developed the Constellation algorithm for finding configurations that satisfy multiple stringent constraints where other constraint-satisfaction strategies fail. We also present the GradienT-RRT algorithm for planning with soft constraints, which outperforms the state-of-the-art approach to high-dimensional path planning with costs.

Acknowledgements

This research was partially supported by Intel Labs Pittsburgh, the Digital Human Research Center (part of AIST Japan), LAAS-CNRS, and the National Science Foundation under Grant No. EEC-0540865. Thanks to Nathan Ratliff, Ross Knepper, Julius Ziegler, Nico Blodow, and David Handron for helpful discussions. Thanks to Mike Vande Weghe, Mehmet Dogar, Alvaro Collet, Anca Dragan, Kyle Strabala, and Rosen Diankov for their collaboration on the Personal Robotics project. Thanks to Joel Chestnut, Koichi Nishiwaki, and Satoshi Kagami for their support at the Digital Human Research Center. Thanks to Jim Mainprice, Romain Iehl, Jean-Philippe Saut, and Thierry Simeon for their support at LAAS-CNRS. Thanks to my collaborators from Karlsruhe Institute of Technology: Tamim Asfour, Nikolaus Vahrenkamp, Felix Messner, and Peter Kaiser. Thank you to my advisors Siddhartha Srinivasa and James Kuffner for their guidance and encouragement and to Matt Mason and Howie Choset for their advice. Finally, thank you to my wife Amy, and to my parents and grandparents, without whom this would not have been possible.

Contents

Chapter 1

Introduction

In the mid 1980's a team of researchers from MIT and several French research institutions developed Handey [1]; an integrated robot system that autonomously perceived its environment, chose how to grasp objects, and constructed plans for pick-and-place manipulation. While previous approaches to manipulation were restricted to very specific tasks [2, 3, 4, 5], Handey sought to be general. Yet the researchers building Handey encountered a fundamental problem:

> The most significant lesson we have drawn from our experience so far with Handey is the need for a systematic and efficient way of dealing with the number of options available while constructing a plan. It is instructive to consider the number of geometrically different ways one could go about stacking two blocks. Consider the block symmetries, the hand symmetries, multiple kinematic solutions, multiple grasp points, and multiple paths. In most cases we don't care which solution is chosen but, unfortunately, many of the possible solutions can be impossible due to the presence of nearby objects or limitations in the robot's joint angles, etc... In earlier work [6], we have considered the use of constraints as a mechanism for making these decisions. Constraint propagation and satisfaction, however, can be extremely difficult and computationally expensive. This area requires a great deal of further work.
>
> Lozano-Perez et al. [1]

More than twenty years later, the field of robotics is still struggling with the same basic questions: How do we plan in a continuous and high-dimensional space of actions with complex constraints on which states are allowed? How do we sort through the infinity of ways an object can be grasped to find only those that are feasible? And, most importantly, how do we produce plans for real-world problems efficiently without sacrificing generality? Like the creators of Handey, we believe the key to answering these questions lies in understanding constraints.

(a) (b) (c) (d) (e) (f)

Figure 1.1: The HERB, HRP2, and HRP3 robots executing grasps and paths planned by our constrained manipulation planning framework.

Every planning problem in robotics involves constraints. Whether the robot must avoid collision or joint limits, there are always states that are not permissible. Some constraints are straightforward to satisfy while others can be so stringent that feasible states are very difficult to find. What makes planning with constraints challenging is that, for many constraints, it is impossible or impractical to provide the planning algorithm with the allowed states explicitly; it must discover these states as it plans. The goal of this thesis is to develop a framework for representing and exploring feasible states and to use this framework to solve real-world manipulation problems.

Manipulation is the study of physical interaction. From towing robots [7] to juggling arms [8] to vibrating tables [9], researchers have explored myriad ways for robots to interact with the physical world. Manipulation planning is the study of *autonomous* physical interaction. A manipulation planning algorithm is responsible for generating the sequence of actions that move the robot and the objects it interacts with from a start state to a goal state.

The techniques presented in this thesis can be applied to manipulation planning tasks where the constraints are evaluated as functions of a robot's configuration. Since our research aims to develop a robot that can assist people in the home, we focus on two important tasks for this application that can be formulated using only constraints on configuration: pick-and-place manipulation and manipulating articulated objects, such as doors (see Figure 1.1). These types of tasks give rise to a rich variety of constraints, including constraints on collision-avoidance, torque, balance, closed-chain kinematics, and end-effector pose.

Some of the most common constraints in manipulation planning involve the pose of a robot's end-effector. These constraints arise in tasks such as reaching to grasp an object, carrying a cup of coffee, or opening a door. As a result, researchers have developed several algorithms capable of planning with end-effector pose constraints [10, 11, 12, 13, 14, 15]. Though often able to solve the problem at hand, these algorithms can be either inefficient [10], probabilistically incomplete [11, 12, 13], or rely on pose constraint representations that are difficult to generalize [14, 15].

This thesis presents a manipulation planning framework that allows robots to plan in the presence of constraints on end-effector pose, as well as others. The framework has three main components: constraint representation, constraint-satisfaction strategies, and a sampling-based approach to planning. These three components come together to create an efficient and probabilistically complete manipulation planning algorithm called the Constrained BiDirectional RRT (CBiRRT2). The underpinning of our framework for pose-related constraints is our Task Space Regions (TSRs) representation. TSRs are intuitive to specify, can be efficiently sampled, and the distance to a TSR can be evaluated very quickly, making them ideal for sampling-based planning. Most importantly, TSRs are a general representation of pose constraints that can fully describe many practical tasks. For more complex tasks, TSRs can be chained together to create more complex end-effector pose constraints, such as those needed to manipulate articulated objects. TSRs can also be used to construct plans that are guaranteed to succeed despite uncertainty in the pose of an object.

Our constrained manipulation planning framework also allows planning with multiple simultaneous constraints. For instance, collision, torque, and balance constraints can be included along with multiple constraints on end-effector pose. Closed-chain kinematics constraints can also be included as a relation between end-effector pose constraints without requiring specialized projection operators [16] or sampling algorithms [17]. In addition, we can plan optimal placements for a robot's base while selecting end-effector poses. An overview of the constraints considered in this thesis and the algorithms that plan with those constraints is shown in Figure 1.2 at the end of this chapter.

We have applied our framework to a wide range of problems for several robots (Figure 1.1), both in simulation and in the real world. These problems include grasping in cluttered environments, lifting heavy objects, two-armed manipulation, and opening doors, to name a few. Despite this wide range of problems, CBiRRT2 only requires three parameters, all of which are constant across many example problems.

These example problems demonstrate our framework's practicality, but it is also important to understand the theoretical properties of manipulation planning. Specifically, we would like to understand whether various sampling methods are able to fully explore the set of feasible configurations. To this end, we provide a proof for the probabilistic completeness of our planning method when planning with constraints on end-effector pose. The proof shows that, given enough time, no part of the constraint manifold corresponding to a pose constraint will be left unexplored, regardless of the dimensionality of the pose constraint. This proof applies to CBiRRT2 as well as other approaches [10, 18], whose probabilistic completeness was previously undetermined.

While the above framework can generate plans for many useful manipulation tasks, there are some cases where none of our constraint-satisfaction strategies apply. These cases often occur when we are trying to find a feasible goal configuration but the set of such configurations is very difficult to find. To address this issue we have developed the *Constellation* algorithm, which searches for a configuration in the intersection of all constraint manifolds using a combination of projection sampling and downhill-simplex. This method is able to find extremely constrained configurations far more reliably than

projection with null-space components [19] or previously-proposed set-intersection methods [20].

Despite their applicability to a wide range of practical problems, the above algorithms are only capable of planning with hard constraints. We would also like to be able to plan with *soft constraints*–i.e. constraints that we would prefer to satisfy but that are not required to be met. Soft constraints can be encoded into a cost function which the planner tries to minimize as it plans. However planning paths for manipulators becomes much more difficult when we wish to optimize the cost of the path in addition to satisfying hard constraints. The high dimensionality of the problem precludes the exhaustive computation necessary to find the globally optimal path. Thus we have developed a practical algorithm called GradienT-RRT, for producing low-cost paths for problems commonly encountered in manipulation planning. This algorithm's effectiveness comes from its ability to navigate narrow low-cost regions in cost-space that are particularly common in manipulation planning problems. We use sampling and gradient methods to explore these regions and compute paths faster than the state-of-the-art algorithm while achieving lower path cost.

Another key component of our work focuses on generating end-effector goal constraints for grasping tasks. Given an object, robot, and environment, the task of a grasp synthesis algorithm is to find poses and configurations of a hand which allow it to grasp the object. The grasp poses generated by such an algorithm can then be used as goal constraints for a planning algorithm. Previous work approaches the grasp synthesis problem from a variety of directions, including machine learning [21, 22, 23], generalizing from demonstrated grasps [24, 25], and maximizing grasp quality metrics through contact-placement [26, 27]. Regardless of the method used for grasp selection, much previous research has focused on finding grasps for a given object/hand pair while ignoring the object's environment. Our approach focuses on finding grasps in cluttered environments, like a kitchen cabinet or a dishwasher. The clutter surrounding an object constrains the set of feasible grasps and makes finding a feasible grasp much more difficult.

We approach grasp synthesis and manipulation planning from a holistic point of view. The goal is not only to find a grasp that is stable for a given object, but also to ensure it is feasible in the current environment and reachable by the robot. Our grasping algorithms take the constraints induced by the surrounding clutter into account when searching for grasps, thus ensuring that the list of grasps generated by our algorithms have a high probability of being valid in the given environment. Since we would like to use TSRs for grasping tasks, we also investigate how to generalize from these grasp lists to TSRs.

In the following chapters, we first discuss the research area of manipulation planning and its role in an autonomous system in Chapter 2 as well as related work in Chapter 3. We then formulate the constrained path planning problem, discuss why it is challenging, and present three strategies for planning with constraints on configuration (Chapter 4). Chapter 5 presents TSRs, which can represent constraints on end-effector poses. We then extend this representation to account for more complex constraints by chaining TSRs together (Chapter 6). Chapter 7 describes the CBiRRT2 planner, which is capable of planning with TSRs and TSR Chains, among other constraints. Chapter 8 describes

several example problems and shows how to formulate constraints for these problems using TSRs. The performance of CBiRRT2 is also evaluated on each example problem. Chapter 9 then discusses the probabilistic completeness of planning with pose constraints.

Chapter 10 describes the process of intersecting TSRs to account for object pose uncertainty and Chapter 11 shows how to use TSRs to find optimal base placements for a mobile manipulator. We then present the Constellation algorithm, which is designed to find extremely constrained robot configurations in Chapter 12. Planning with soft constraints is then discussed in Chapter 13.

Chapters 14 and 15 present two algorithms for generating grasps in cluttered environments. We show how to unify these algorithms with the TSR representation by converting discrete lists of grasps into TSRs in Chapter 16. Finally, we summarize the contributions of this thesis and discuss directions for future work in Chapters 17 and 18.

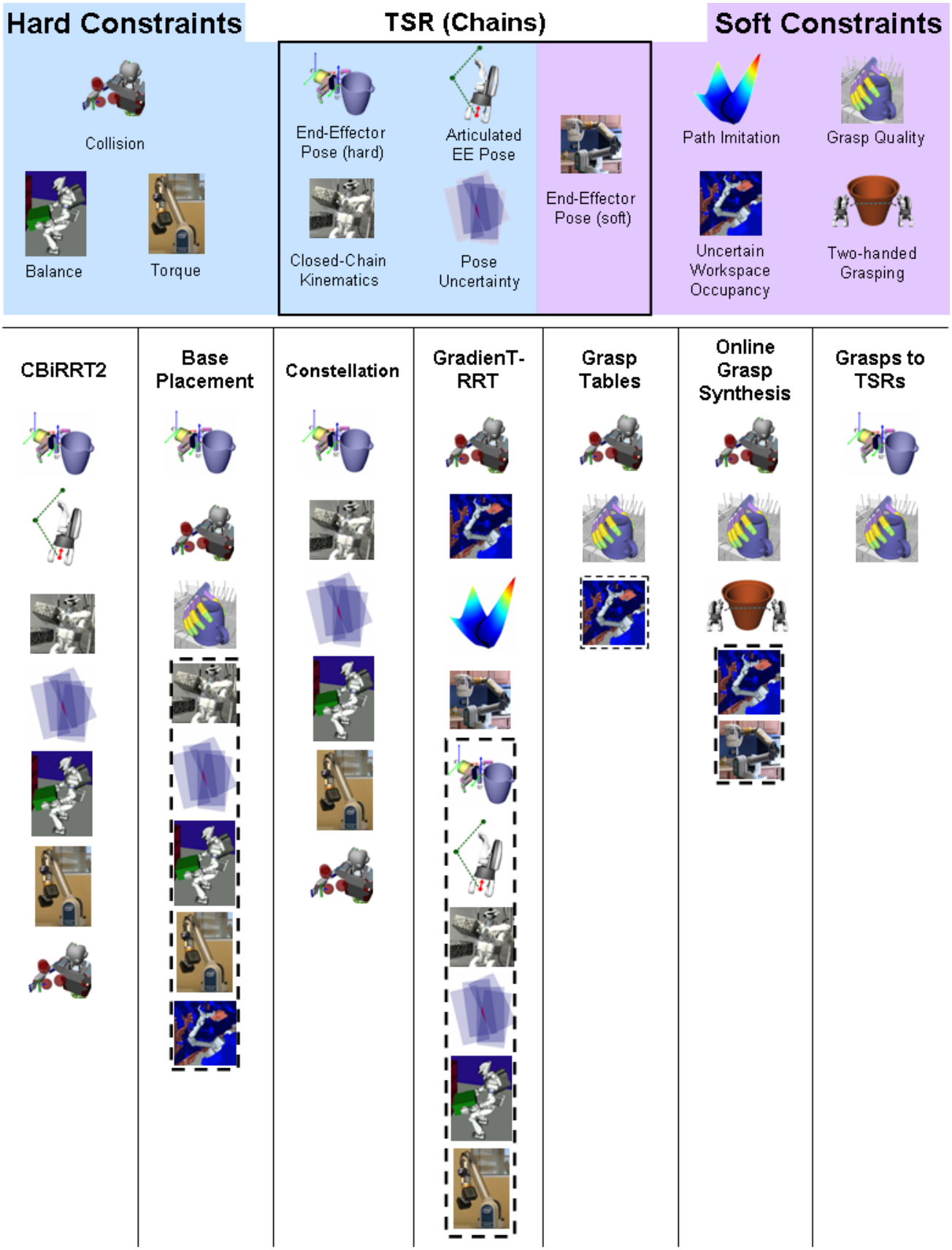


Figure 1.2: The constraints discussed in this thesis and the algorithms that plan with them. Constraints in dotted boxes are possible to use with the above algorithms but we do not provide examples that demonstrate this.

Chapter 2

Manipulation Planning in Context

In order to motivate the techniques described in this thesis, this chapter presents the context in which we see a role for manipulation planning. There are many applications that can make use of manipulation planning. From factory robotics to robots in the home, manipulation capabilities are essential for many useful tasks. Though it is possible to use manipulation planners in a stand-alone fashion for some applications, we see manipulation planning as a part of an integrated stack of modules like the one shown in Figure 2.1.

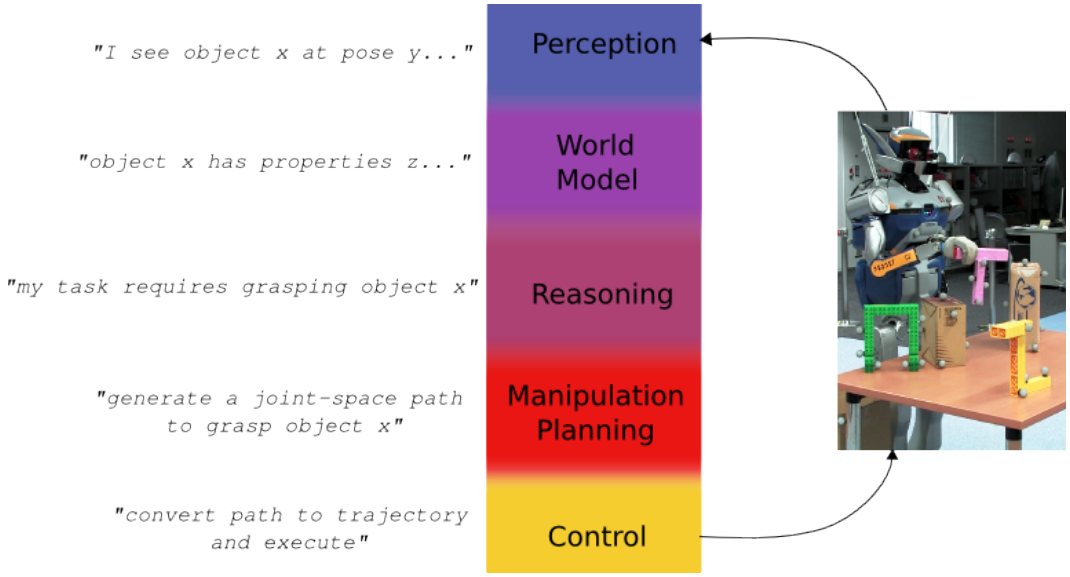


Figure 2.1: An example of an integrated stack of modules for manipulation tasks. The roles performed by each component for the task of grasping a given object are shown on the left.

Since we want to reduce the complexity of such systems as much as possible, one might ask why manipulation planning is necessary at all. Why does the reasoning system need to go through a planner

in order to give commands to a controller? After all, the reasoning system is capable of specifying which objects need to be manipulated and the control system is capable of moving the manipulator to any desired location. Controllers are even able to deal with some constraints on the manipulator's motion, so what is the advantage of having a planner? The answer lies in the different scopes of planning and control for manipulation. For the most part, controllers think locally, while planners think globally.

Some of the most successful methods in control theory for manipulation have come from controllers that seek to minimize a given function in the neighborhood of the robot's current configuration through gradient-descent. For instance, controllers have been developed for balancing two-legged robots [28], placing the end-effector somewhere in task space [29], and collision-avoidance [19]. The technique of recursive null-space projection [19] can satisfy multiple constraints simultaneously by prioritizing the constraints and satisfying lower-priority constraints in the null-space of higher-priority ones. These controllers can succeed or fail depending on the prioritization of constraints and it is unclear which of the multiple simultaneous constraints should be prioritized ahead of which others. Even if the prioritization issue is resolved, controllers based on gradient-descent only guarantee that a local minimum of the function is found, which may not be sufficient to solve the problem.

Researchers in control theory have also pursued global solutions, such as building control policies through dynamic programming [30]. However such approaches do not scale to the high-dimensional configuration spaces of most manipulators because of the exponential cost of computing optimal policies. Another popular approach has been to learn policies from demonstration [31, 32, 33]. However, for our scope of manipulation problems, finding a set of examples that spans the space of tasks the manipulator is expected to perform as well as a robust way to generalize from these examples has proven quite difficult.

Thus control methods alone are not sufficient to provide a practical solution to the problem of global planning for manipulation tasks. It is here that sampling-based planners [34, 35], especially for high-DOF manipulators, are most useful. Sampling-based manipulation planners are designed to explore the space of solutions efficiently, without the exhaustive computation required for dynamic programming and without being trapped by local minima like gradient-descent controllers. This global planning ability is acutely important for manipulation planning because even the most common constraints like collision-avoidance can trap approaches based on gradient-descent. Though sampling-based planners can operate in the state space of the robot to produce trajectories [34], we have found that they are far more efficient and successful when operating in the C-space to produce C-space paths. These paths can then be converted to trajectories [36] and executed by an appropriate controller.

On the other side of manipulation planning is the higher-level reasoning component that commands the planner. In practice the commands can come directly from a user but we will focus on the fully autonomous system exemplified in Figure 2.1. Researchers in artificial intelligence and robotics have been investigating higher-level reasoning methods for many years, with a wealth of different techniques being developed for different applications. The most relevant higher-level reasoning techniques

for manipulation planning are the STRIPS planners [37] and their descendants [38, 39]. The STRIPS framework describes the world in terms of instances, predicates, and operators. The instances are the set of distinct objects in the world. The predicates are the properties of and relations between these instances. The operators are actions that change the state of the world. An operator can be executed if its set of preconditions is valid and the execution of the operator produces a set of changes to the world. STRIPS-like planners are capable of producing a sequence of operators that achieve a desired world state. In the manipulation planning context, an operator can correspond to an action such as "pick up the cup" which requires a robot to perform a sequence of motions that must be planned. Subsuming manipulation planning into higher-level reasoning is far too burdensome because geometric (and other) constraints on robot motion are difficult to express using propositional or first-order logic. Thus a STRIPS-like planner should determine the sequence of operators and allow the manipulation planner to construct paths that perform those operators.

Overall, the contribution of the work presented in this thesis is to expand the frontiers of the manipulation planning component of the system by handling a variety of constrained tasks, i.e. enabling the execution of more types of operators. For instance, consider the task of cleaning up after a party. In such a task, the robot needs to grasp and throw away a series of objects and the order of the objects is unimportant. Traditionally, the reasoning system would need to decide on the order and call the manipulation planner for each object in sequence. However, this requires the reasoning system to understand which objects are reachable and graspable to generate the sequence or else try random objects until one is grasped. Alternatively, using the algorithms presented in this thesis, the reasoning system need only select which set of objects to grasp and allow the manipulation planner to do the rest (see Section 8.2 for an example).

On the controls side, consider the task of transporting a cup full of water. Traditionally, one would create a task-space controller to keep the mug from tilting while moving it directly towards a goal. However, such an approach lacks the power to explore the space of possible C-space paths, which is needed to avoid obstacles and meet other constraints that could be imposed on the robot's motion. Alternatively, the constrained planning algorithms in this thesis are capable of generating paths in the presence of multiple heterogeneous constraints while exploring the C-space. It is still important to fit trajectories to these paths and to track them with appropriate controllers to compensate for errors during execution but the burden of constructing constrained paths can fall to a manipulation planning algorithm instead of a controller.

Chapter 3

Related Work

We now discuss methods in the literature that are relevant to the algorithms presented in this thesis. We first discuss work relevant to our TSR representation for pose constraints, and the CBiRRT2 algorithm, which plans with TSRs. We then describe work related to our method for intersecting TSRs to account for pose uncertainty. Next we discuss related work for our base placement algorithm and Constellation, which finds configurations in the intersection of constraint manifolds. We then describe work relevant to planning in high-dimensional cost-spaces as it relates to our GradienT-RRT algorithm. We conclude with related work for our approach to grasp synthesis in cluttered environments.

3.1 Representing Pose Constraints as TSRs and TSR Chains

TSRs are a straightforward pose constraint representation that can capture many useful tasks. For more complex tasks, we have also developed TSR Chains, which are defined by linking a series of TSRs. TSRs build on a long history of constraint-based problem specification. Seminal theoretical work in this area was done by Ambler and Popplestone [40], who specify geometric constraints between features of two objects and then solve for the pose of a robot which assembles these objects using symbolic methods. The AL system [41, 42] encoded pose constraints on object placement as inequalities of position and rotation variables, which is similar to the bounds of a TSR. The AUTOPASS system [43] allowed specifying pose constraints for primitive motions of a manipulator.

More recent work in sampling-based planning involves planning to a goal pose [14, 44] or set of goal poses [15] for the end-effector. The representations used in this work are subsumed by TSRs. Stilman [10] presented a representation for pose constraints on the robot's path, which is also subsumed by TSRs. Finally, a representation similar to TSRs was used by De Schutter et al. [45] for a controller that maintained the pose of a frame on the robot in a set defined relative to a frame in the environment.

3.2 CBiRRT2: Planning with Pose Constraints

Our framework for planning with pose constraints exploits the strengths of control theory and sampling-based planning to produce an algorithm that searches globally while satisfying constraints locally. The CBiRRT2 planner uses a bi-directional RRT to explore the constraint manifold while enforcing constraints via rejection sampling and projection methods based on gradient-descent. Although CBiRRT2 is based on the Rapidly-exploring Random Tree (RRT) algorithm [34], it is possible to adapt some of the constraint-satisfaction strategies used by CBiRRT2 to other search algorithms such as the PRM [35]. We selected RRTs for their ability to explore C-space while retaining an element of "greediness" in their search for a solution. The greedy element is most evident in the bidirectional version of the RRT algorithm (BiRRT), where two trees, one grown from the start configuration and one grown from the goal configuration, take turns exploring the space and attempting to connect to each other.

The task of the CBiRRT2 planner is to construct a C-space path that lies on the constraint manifolds induced by pose constraints (as well as constraints like collision-avoidance, balance, and torque). We distinguish this task from the task of tracking pre-scripted end-effector paths [46, 47, 48] because we do not assume an end-effector path is given. CBiRRT2 uses gradient-descent inverse-kinematics techniques [19, 49] to meet pose constraints and sample goal configurations. The algorithm plans in the full C-space of the robot, which implicitly allows it to search the null-space of pose constraints, unlike task-space planners [11, 12, 13], which assign a single configuration to each task-space point (from a potentially infinite number of possible configurations). Exploration of the null-space is necessary for probabilistic completeness (discussed in Chapter 9) and can be useful for satisfying other constraints, such as avoiding obstacles or maintaining balance, though our constraint representation could be incorporated into task-space planners as well.

Algorithms similar to CBiRRT2 have been proposed by Yakey et al. [16] and Stilman [10]. Yakey et al. proposed using Randomized Gradient Descent (RGD) to meet closed-chain kinematics constraints. RGD uses random-sampling of the C-space to iteratively project a sample towards an arbitrary constraint [13]. Though Yakey et al. showed how to incorporate RGD into a sampling-based planner and their method is quite general, it requires significant parameter-tuning and they dealt only with closed-chain kinematic constraints, which are a special case of the pose constraints used in this thesis. Furthermore, Stilman showed that when RGD is extended to work with more general pose constraints it is significantly less efficient than Jacobian pseudo-inverse projection and it is sometimes unable to meet more stringent constraints [10]. Our approach is similar to Stilman's in that we use an RRT-based planner with Jacobian pseudo-inverse projection, however we differ in the specifics of the planning algorithm and use TSRs, which are a more general constraint representation.

Another key feature of CBiRRT2 is that it can sample TSRs to produce goal configurations. Other researchers have approached the problem of ambiguous goal specification by sampling some number of goals before running the planner [50, 51], which limits the planner to a small set of solutions from a region which is really continuous. Another approach is to bias a single-tree planner toward the

goal regions, however this approach usually considers single points [14, 44] in the task space or is hand-tuned for specific goal regions [15].

3.3 Accounting for Pose Uncertainty with TSRs

To account for pose uncertainty in the objects we wish to grasp, we have developed a method that intersects TSRs corresponding to the pose hypotheses of the object. We then sample from that volume of intersection to generate end-effector poses that are guaranteed to meet task specifications despite uncertainty. The idea of motion planning in the presence of uncertainty dates back to the seminal work of Lozano-Perez et al. [52] on *preimage backchaining*. The concept of a preimage—a region of configuration space from which a motion command is guaranteed to attain a given goal recognizably—was used as a building block to compose a planner that produced actions guaranteed to succeed under pose and action uncertainty. However, it was shown that constructing preimages incurred a prohibitive computational cost [53, 54]. We show in Chapter 10 that, for the case of manipulation planning, TSRs provide an efficient representation for computing the preimage of hand poses that are guaranteed to provide an acceptable grasp or object placement under object pose uncertainty.

Another area of related work is quality metrics that characterize the robustness of grasps to pose uncertainty. Techniques for computing such metrics have ranged from first and second-order analysis of the perturbation of grasp contact points [55, 56, 57, 58] to generating contact patches [58, 59, 60]. These metrics could be used to construct robust TSRs for grasping under uncertainty.

Our approach requires that we have a set of hypotheses of pose for every object in the environment other than the robot. These hypothesis sets can come from sensors that are on-board the robot, sensors that are fixed in the environment, or a mixture of both. In general, pose estimates can be propagated through different frames using the methods of Smith and Cheeseman [61]. Though Smith and Cheeseman focus mainly on 2D poses, their methods have been extended to six-dimensional poses in 3D. See Sallinen's thesis [62] for an overview.

3.4 Base Placement

Our algorithm for base placement considers several criteria to determine optimal base placements along with optimal grasps (according to a grasp quality metric [26, 27, 60]) for pick-and-place tasks. Instead of greedily choosing the best grasp at the initial configuration of the object based solely on grasp quality, our algorithm considers the object in both its starting and goal configurations as well as the placement of the robot's base. This differs from the work of Hsu et al. [63], who study the problem

of where to place a fixed base manipulator in a factory environment for optimal task execution because we consider a mobile base.

The base placement algorithm uses a co-evolutionary genetic algorithm to optimize the start and final base placements along with the grasp. Other researchers have also used genetic algorithms for mobile manipulation tasks. Zhao et al. [64] use a genetic algorithm to plan a path for the mobile base between a discrete set of feasible base-placements. The set of placements is determined by exhaustive search but obstacles and grasping are not considered. Chen and Zalzala [65] use a genetic algorithm to plan a path through a gridded environment using distance to obstacles as one of the criteria for optimization. Vannoy and Xiao [66] use a genetic algorithm to create a population of trajectories for a mobile manipulator that allow it to avoid dynamic obstacles while maintaining good manipulability. Similarly, our base placement algorithm considers both distance to obstacles and manipulability when determining the fitness of a robot configuration.

Stilman et al. [50] and Cambon et al. [67] both outline manipulation planning frameworks that could make use of our base placement algorithm. Stilman et al. examine the problem of manipulation planning among movable obstacles. Here the object is initially inaccessible and movable obstacles must be displaced in order to pick up the object and place it in its goal configuration. Cambon et al. have developed a motion planning framework where high-level action plans and low-level path plans are searched in parallel. The high level plans can involve regrasping and moving obstructing obstacles out of the way. Our algorithm can be integrated with these frameworks to choose the optimal base-placements and grasps for all low-level path plans. Likewise if the object must be slid through a narrow space as in [68] and [69].

3.5 Constellation: Finding configurations that satisfy multiple constraints

The Constellation algorithm approaches the problem of generating configurations that satisfy multiple constraints as a set-intersection problem. The core operation used by Constellation is *direct projection*—projecting a guess configuration to one of the constraint manifolds while projecting to the others in the null-space of the first constraint [19]. To generate a configuration near the intersection of all constraint manifolds Constellation uses direct projection and downhill-simplex methods [70] to iteratively build a graph in the C-space. At each iteration of Constellation, we find the shortest cycle in the graph that contains a node on each constraint manifold and use that cycle to generate a new guess configuration, which is then projected to all the constraint manifolds. The nodes resulting from these projections are added to the graph and the process is repeated until a node satisfying all constraints is found. Since Constellation uses direct projection as its underlying projection operator, all problems that can be solved by a single direct projection will be solved by Constellation.

The projection methods in Constellation are derived from iterative inverse-kinematics techniques ([19, 49]) and gradient descent controllers [28, 29, 19]. RGD [16] can also be used to iteratively project a sample towards an arbitrary constraint [13]. Though RGD is quite general, it requires significant parameter-tuning and can take quite a long time to converge to within a small tolerance of the constraint, especially in high-dimensional spaces, thus it is not appropriate for use in Constellation.

Another approach to finding configurations that satisfy multiple constraints is to frame the task as a general optimization problem and to use a solver like FSQP, as in [71]. An overview of such global optimization methods can be found in [72]. While global optimization algorithms can be applied to many problems, they are often quite time-consuming and require that constraints meet certain criteria such as differentiability. Constraints such as collision-avoidance for complex geometries are difficult to encode in a way that is consistent with such criteria.

A method that is related to Constellation is cyclic projection [20], which iteratively projects a configuration to a repeating sequence of constraints. This algorithm does not take advantage of the null-space of a constraint, which we show is quite useful for guiding the search toward the intersection of the constraints. We compare Constellation to cyclic projection in Section 12.2.

3.6 GradienT-RRT: Planning with Soft Constraints

Our algorithm for planning with soft constraints, GradienT-RRT, builds on both the Transition-based RRT (T-RRT) [73] and gradient methods often used in control [28, 29, 19]. Related planners that plan in the C-space while optimizing cost are the heuristically-guided RRT [74] and the Anytime RRT [75]. Though they perform well in mobile-robotics domains, these planners have difficulty in high-dimensional manipulation problems with continuous cost functions because an adequate heuristic is not readily available. Other planners that consider the *obstacleness* of a configuration [76] have difficulty with arbitrary cost functions because the cost threshold growth rate parameter is highly problem-dependent. Another related planner is Conformational Roadmaps [77], which uses a transition test similar to the T-RRT to explore molecular energy landscapes.

A common approach in motion planning is to attempt optimization of soft constraints only after a path satisfying the hard constraints has been found. Methods like shortcut smoothing [78] or partial-shortcut [79] can be used to optimize the length and distance from obstacles of a feasible path. However, such methods only improve a given path locally. In many problems (like the ones presented in Section 13.5), shortcutting will not produce an adequately low-cost path from an arbitrary path because it is unlikely to find a *cost-space chasm*—a narrow low-cost region in the cost space.

The problem of navigating cost-space chasms is related to that of exploring narrow passages in sampling-based planning. The integration of a gradient method and sampling-based planner in the GradienT-

RRT is related to retraction-based methods [80][81] designed to explore narrow passages. Other methods of addressing narrow passages, such as bridge-sampling [82], diffusion control [83], and dilation-based approaches [84] are also widely studied in motion planning. Our problem domain differs because we consider continuous-valued cost functions, which are not narrow passages in terms of feasibility. It is unclear how to generalize the methods cited above to this cost-space domain.

3.7 Grasp Synthesis

In the area of grasping, our approach to generating feasible grasping poses and configurations builds on sampling and executing grasps in simulation [85, 23, 86]. Many researchers have also approached the problem of grasping from a machine learning perspective where the goal is to find grasps of novel objects using information about grasps of already-known objects [21, 22] or to generalize from demonstrated grasps [24, 25]. Another area of grasping research focuses on finding a placement of contact points on an object's surface to maximize a certain grasp metric [26, 27, 60].

Regardless of the method used for grasp selection, much previous research has focused on finding grasps for an object when it is alone in the environment. Furthermore, the hand is often assumed to be disembodied when approaching the object [23], so the kinematics of the robot are not taken into account. While the above assumptions may be valid in certain situations, they are certainly not true for cluttered environments. Thus we approach grasp selection and manipulation planning from a holistic point of view. The goal is not only to select a grasp that is stable for a given object, but also to ensure it is feasible.

Chapter 4

Constraints

Depending on the robot and the task, many types of constraints can limit a robot's motion. One of the most common distinctions in the robotics literature is between *holonomic* and *nonholonomic* constraints. A *holonomic* constraint is one that can be expressed as a function of the configuration of the robot q (and possibly time t) and has the form $F(q,t) = 0$. A *nonholonomic* constraint is one that cannot be expressed in this way. It is important to note that the definition of *holonomic* above does not allow inequalities, i.e. it must be a bilateral constraint. This fact implies that constraints that are typically thought of as holonomic in robotics literature such as collision-avoidance are in fact nonholonomic. To preserve consistency with the robotics literature, we will relax the definition of holonomic constraints to include inequality constraints for the purposes of this thesis.

Nonholonomic constraints are ones that are impossible to represent as a function of only the configuration of the robot and time. A classical example of such a constraint is the kinematics of the unicycle, which can move forward and back and rotate about the center of the wheel but cannot move sideways. This constraint can be expressed as

$$F(\dot{x}, \dot{y}, \theta) = \dot{x} \sin\theta - \dot{y} \cos\theta. \tag{4.1}$$

Some constraints that depend on the derivatives of configuration variables can be integrated into holonomic constraints but the above one can not, thus it is nonholonomic. This is the reason that nonholonomic constraints are sometimes referred to as *nonintegrable* constraints.

There is also a distinction between constraints with respect to their dependence on time. If a constraint depends on time (among other variables), it is referred to as *rheonomic*, otherwise it is referred to as *scleronomic*.

4.1 Defining Constraints on Configuration

This thesis focuses on scleronomic holonomic constraints, which are time-invariant constraints evaluated at a given configuration of the robot. Let the configuration space of the robot be $\mathcal{Q}$. A path in that space is defined by $\tau : [0, 1] \rightarrow \mathcal{Q}$. We consider constraints evaluated as a function of a configuration $q \in \mathcal{Q}$ in τ. The location of q in τ determines which constraints are active at that configuration. Thus a constraint is defined as the pair $\{C(q), s\}$, where $C(q) \in \mathbb{R} \geq 0$ is the *constraint-evaluation function* and $s \subseteq [0, 1]$ is the *domain* of the constraint. $C(q)$ determines whether the constraint is met at that q and s specifies where in the path τ the constraint is active. To say that a given constraint is satisfied we require that $C(q) = 0 \ \ \forall q \in \tau(s)$. For instance, we may require that τ start at a given configuration q_{start}:

$$C(q) = \begin{cases} 0 & \text{if } q = q_{start} \\ 1 & \text{otherwise} \end{cases} \quad \text{for } q = \tau(0) \tag{4.2}$$

We may also require that τ be collision-free everywhere along the path. The collision-avoidance constraint is then defined as

$$C(q) = \begin{cases} 1 & \text{if InCollision}(q) \\ 0 & \text{otherwise} \end{cases} \quad \forall q \in \tau(\cdot). \tag{4.3}$$

Each constraint defined in this way implicitly defines a manifold in $\mathcal{Q}$ where $\tau(s)$ is allowed to exist. Given a constraint, the manifold of configurations that meet this constraint $\mathcal{M}_C \subseteq \mathcal{Q}$ is defined as

$$\mathcal{M}_C = \{q \in \mathcal{Q} : C(q) = 0\}. \tag{4.4}$$

In order for τ to satisfy a constraint, all the elements of $\tau(s)$ must lie within $\mathcal{M}_C$. If $\exists q \notin \mathcal{M}_C$ for $q \in \tau(s)$ then τ is said to violate the constraint.

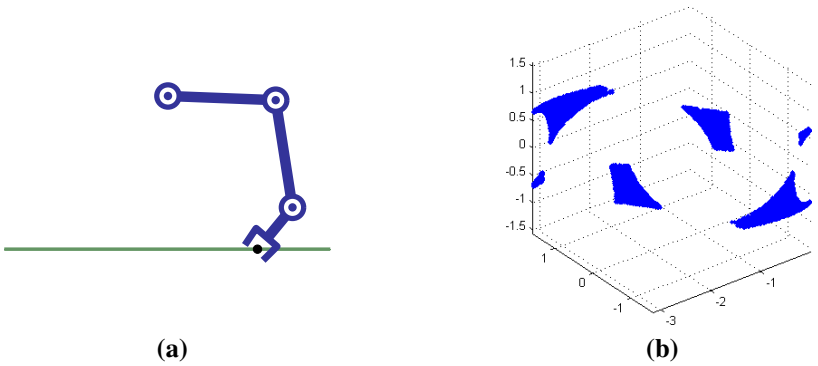
In general, we can define any number of constraints for a given task, each with their own domain. Let a set of n constraint-evaluation functions be $\mathcal{C}$ and the set of domains corresponding to those functions be $\mathcal{S}$. Then we define the constrained path planning problem as

$$\begin{aligned} \text{find } \tau : q \in \mathcal{M}_{\mathcal{C}_i} \quad & \forall q \in \tau(\mathcal{S}_i) \\ & \forall i \in \{1...n\}. \end{aligned} \tag{4.5}$$

Note that the domains of two or more constraints may overlap in which case an element of τ may need to lie within two or more constraint manifolds.

4.2 Challenges of Constrained Path Planning

Two main issues make solving the constrained path planning problem difficult. First, constraint manifolds are difficult to represent. There is no known analytical representation for many types of constraint manifolds (including pose constraints) and the high dimensional C-spaces of most practical robots make representing the manifold through exhaustive sampling prohibitively expensive. It is possible to parameterize some constraint manifolds, however this can be insufficient for planning paths because the mapping from the parameter space to the manifold can be non-smooth (see Figure 4.1). Thus, although we can construct a smooth path in the parameter space, its image on the constraint manifold may be disjoint. Restrictions imposed on the mapping to render it smooth, like imposing a one-to-one mapping from pose to configuration, compromise on completeness.



(a) **(b)**

Figure 4.1: (a) Pose constraint for a 3-link manipulator: The end-effector must be on the line with an orientation within ± 0.7rad of downward. (b) The manifold induced by this constraint in the C-space of this robot.

Second, and acutely important for pose constraints, is the fact that constraint manifolds can be of a lower dimension than the ambient C-space. Lower-dimensional manifolds cannot be sampled using rejection sampling (the sampling technique employed by most sampling-based planners) and thus more sophisticated sampling techniques are required. A key challenge is to demonstrate that the distribution of samples produced by these techniques densely covers the constraint manifold, which is necessary for probabilistic completeness.

We address the first issue by using a sampling-based planner that explores the constraint manifold in the C-space (not in the parameter space). This planner uses a variety of sampling techniques to generate samples on constraint manifolds (Section 4.3). One of these techniques is able to sample

lower-dimensional constraint manifolds. We validate the probabilistic completeness of this approach in Chapter 9, thus addressing the second issue.

4.3 Sampling on Constraint Manifolds

In order to solve the constrained path planning problem, a sampling-based planning algorithm must be able to generate configurations that lie on constraint manifolds. We describe three general strategies for generating these configurations: rejection, projection, and direct sampling (see Figure 4.2).

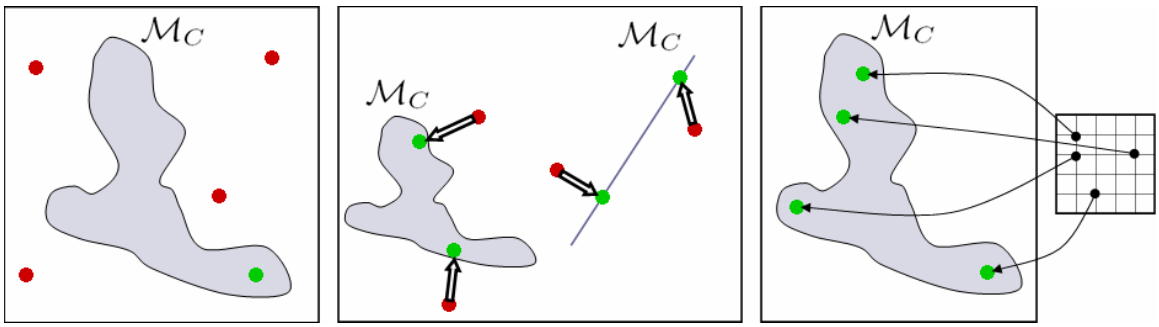


Figure 4.2: The three sampling strategies used in our framework. Red dots represent invalid samples and green dots represent valid ones. (Left) Rejection sampling (Center) Projection sampling (Right) Direct sampling from a parameterization of the constraint

In the *rejection* strategy, we simply generate a random sample $q \in \mathcal{Q}$ and check if $C(q) = 0$, if this is not the case, we deem q invalid. This strategy is effective when there is a high probability of randomly sampling configurations that satisfy this constraint, in other words, $\mathcal{M}_C$ occupies some significant volume in $\mathcal{Q}$. This strategy is used to satisfy torque, balance, and collision constraints, among others.

The *projection* strategy is robust to more stringent constraints, namely ones whose manifolds do not occupy a significant volume of the C-space. However this robustness comes at the price of requiring a function to evaluate how close a given configuration is to the constraint manifold, i.e. $C(q)$ needs to encode some measure of distance to the manifold. The projection strategy first generates a $q_0 \in \mathcal{Q}$ then moves that q_0 onto $\mathcal{M}_C$. The most common type of projection operator relevant for our application is an iterative gradient-descent process. Starting at q_0, the projection operator iteratively moves the configuration closer to the constraint manifold so that $C(q_{i+1}) < C(q_i)$. This process terminates when the gradient-descent reaches a configuration on $\mathcal{M}_C$, i.e. when $C(q_i) = 0$. A key advantage of the projection strategy is that it is able to generate valid configurations near other configurations on $\mathcal{M}_C$, which allows us to use it in algorithms based on the RRT. This strategy is used to sample on lower-dimensional constraint manifolds, such as those induced by end-effector pose or closed-chain kinematics constraints.

Finally, the *direct sampling* strategy uses a parameterization of the constraint to generate samples on $\mathcal{M}_C$. This strategy is specific to the constraint representation and the mapping from the parameterization to $\mathcal{M}_C$ can be arbitrarily complex. Though this strategy can produce valid samples, it can be difficult to generate samples in a desired region of $\mathcal{M}_C$, for instance generating a sample near other samples (a key requirement for building paths). Thus we will use this strategy only when sampling goals for our planner, not to build paths. We will describe how to do direct sampling with our constraint representation in Chapter 5.

A given constraint may be sampled using one or more of these strategies. The choice of strategy depends on the definition of the constraint as well as the path planning algorithm. Sometimes a mix of strategies may be appropriate. For instance, a PRM planning with pose constraints may use the direct sampling strategy to generate a set of map nodes but may switch to the projection strategy when constructing edges between those nodes.

Chapter 5

Task Space Regions

We now focus on a specific constraint representation that we have developed for planning paths for manipulators with end-effector pose constraints. The pose of a manipulator's end-effector is represented as a point in $SE(3)$, the six-dimensional space of rigid spatial transformations. Many practical manipulation tasks, like moving a large box or opening a refrigerator door, impose constraints on the motion of a robot's end-effector(s) as well as allowing freedom in the acceptable goal pose of the end-effector. For example consider a humanoid robot placing a large box onto a table (see Figure 1.1d). Although the humanoid's hands are constrained to grasp the box during manipulation, the task of placing the box on the table affords a wide range of box placements and robot configurations that achieve the goal. We propose a framework for pose-constrained manipulation planning which is capable of trading off constraints and affordances to produce manipulation plans for high degree of freedom robots, like humanoids or mobile manipulators.

Our constrained manipulation planning framework uses a novel unifying representation of constraints and affordances which we term Task Space Regions (TSRs). TSRs describe end-effector constraint sets as subsets of $SE(3)$. These subsets are particularly useful for specifying manipulation tasks ranging from reaching to grasp an object and placing it on a surface or in a volume, to manipulating objects with constraints on their pose such as transporting a glass of water without spilling or sliding a milk jug on a table.

TSRs are specifically designed to be used with sampling-based planners. As such, it is straightforward to specify TSRs for common tasks, to compute distance from a given pose to a TSR (necessary for the projection strategy), and to sample from a TSR using direct sampling. Furthermore, multiple TSRs can be defined for a given task, which allows the specification of multiple simultaneous constraints and affordances.

The work discussed in this chapter was published in [87] and [88].

TSRs are not intended to capture every conceivable constraint on pose. Instead they are meant to be simple descriptions of common manipulation tasks that are useful for planning. We have also developed a more complex representation for articulated constraints called TSR Chains, which is discussed in Chapter 6. Finally, we discuss the limitations of these representations in Chapter 18.

5.1 TSR Definition

Throughout this chapter, we will be using transformation matrices of the form $\mathbf{T}_b^a$, which specifies the pose of b in the coordinates of frame a. $\mathbf{T}_b^a$, written in homogeneous coordinates, consists of a 3×3 rotation matrix $\mathbf{R}_b^a$ and a 3×1 translation vector $\mathbf{t}_b^a$.

$$\mathbf{T}_b^a = \begin{bmatrix} \mathbf{R}_b^a & \mathbf{t}_b^a \\ \mathbf{0} & 1 \end{bmatrix} \tag{5.1}$$

A TSR consists of three parts:

- $\mathbf{T}_w^0$: transform from the origin to the TSR frame w
- $\mathbf{T}_e^w$: end-effector offset transform in the coordinates of w
- $\mathbf{B}^w$: 6×2 matrix of bounds in the coordinates of w:

$$\mathbf{B}^w = \begin{bmatrix} x_{min} & x_{max} \\ y_{min} & y_{max} \\ z_{min} & z_{max} \\ \psi_{min} & \psi_{max} \\ \theta_{min} & \theta_{max} \\ \phi_{min} & \phi_{max} \end{bmatrix} \tag{5.2}$$

The first three rows of $\mathbf{B}^w$ bound the allowable translation along the x, y, and z axes (in meters) and the last three bound the allowable rotation about those axes (in radians), all in the w frame. Note that this assumes the Roll-Pitch-Yaw (RPY) Euler angle convention, which is used because it allows bounds on rotation to be intuitively specified.

In practice, the w frame is usually centered at the origin of an object held by the hand or at a location on an object that is useful for grasping. We use an end-effector offset transform $\mathbf{T}_e^w$, because we do not assume that w directly encodes the pose of the end-effector. $\mathbf{T}_e^w$ allows the user to specify an offset

from w to the origin of the end-effector e, which is extremely useful when we wish to specify a TSR for an object held by the hand or a grasping location which is offset from e; for instance in between the fingers. For some example $\mathbf{T}_e^w$ transforms, see Figure 5.1.

5.2 Distance to TSRs

When using the projection strategy with TSRs, it will be necessary to find the distance from a given configuration q_s to a TSR (please follow the explanation below in Figure 5.2). Because we do not have an analytical representation of the constraint manifold corresponding to a TSR, we compute this distance in task space. Given a q_s, we use forward kinematics to get the position of the end-effector at this configuration $\mathbf{T}_s^0$. We then apply the inverse of the offset $\mathbf{T}_e^w$ to get $\mathbf{T}_{s'}^0$, which is the pose of the grasp location or the pose of the object held by the hand in world coordinates.

$$\mathbf{T}_{s'}^0 = \mathbf{T}_s^0(\mathbf{T}_e^w)^{-1} \tag{5.3}$$

We then convert this pose from world coordinates to the coordinates of w.

$$\mathbf{T}_{s'}^w = (\mathbf{T}_w^0)^{-1}\mathbf{T}_{s'}^0 \tag{5.4}$$

Now we convert the transform $\mathbf{T}_{s'}^w$ into a 6×1 displacement vector from the origin of the w frame. This displacement represents rotation in the RPY convention so it is consistent with the definition of $\mathbf{B}^w$.

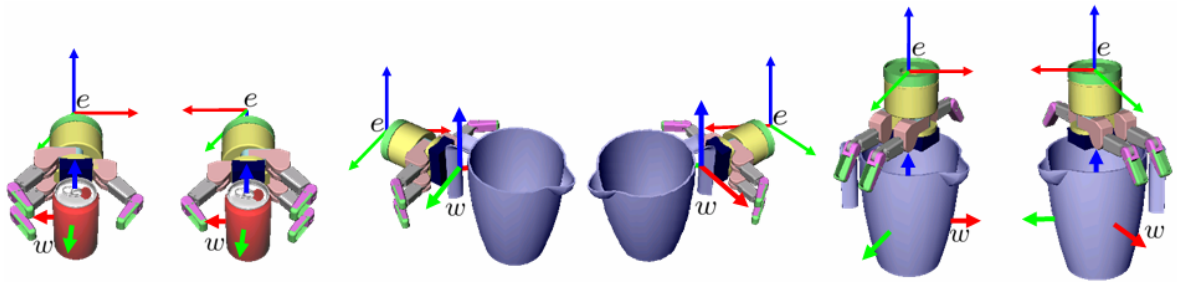


Figure 5.1: The w and e frames used to define end-effector goal TSRs for a soda can and a pitcher.

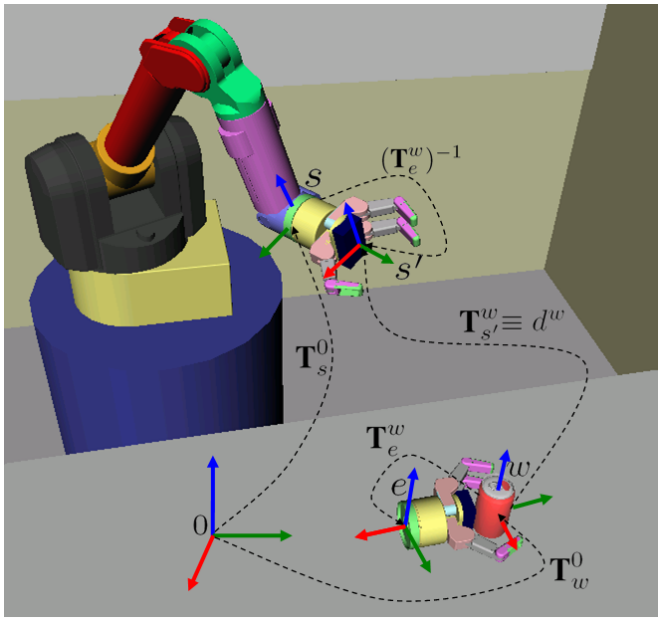


Figure 5.2: Transforms and coordinate frames involved in computing the distance to TSRs. The robot is in a sample configuration which has end-effector transform s and the hand near the soda can at transform e represents the $\mathbf{T}_e^w$ defined by the TSR.

$$d^w = \begin{bmatrix} \mathbf{t}_{s'}^w \\ \arctan 2(\mathbf{R}_{s'_{32}}^w, \mathbf{R}_{s'_{33}}^w) \\ -\arcsin(\mathbf{R}_{s'_{31}}^w) \\ \arctan 2(\mathbf{R}_{s'_{21}}^w, \mathbf{R}_{s'_{11}}^w) \end{bmatrix} \tag{5.5}$$

Taking into account the bounds of $\mathbf{B}^w$, we get the 6×1 displacement vector to the TSR $\Delta\mathbf{x}$

$$\Delta\mathbf{x}_i = \begin{cases} d_i^w - \mathbf{B}_{i,1}^w & \text{if } d_i^w < \mathbf{B}_{i,1}^w \\ d_i^w - \mathbf{B}_{i,2}^w & \text{if } d_i^w > \mathbf{B}_{i,2}^w \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

where i indexes through the six rows of $\mathbf{B}^w$ and six elements of $\Delta\mathbf{x}$ and d^w. $\|\Delta\mathbf{x}\|$ is the distance to the TSR. Note that we implicitly weigh rotation in radians and translation in meters equally when computing $\|\Delta\mathbf{x}\|$ but the two types of units can be weighed in an arbitrary way to produce a distance metric that considers one or the other more important. Because of the inherent redundancy of the RPY Euler angle representation, there are several sets of angles that represent the same rotation. To find the minimal distance by our metric, we evaluate the norm of each of the possible RPY angle sets capable of yielding the minimum displacement. This set consists of the $\{\Delta\mathbf{x}_4, \Delta\mathbf{x}_5, \Delta\mathbf{x}_6\}$ defined above as well as the eight equivalent rotations $\{\Delta\mathbf{x}_4 \pm \pi, -\Delta\mathbf{x}_5 \pm \pi, \Delta\mathbf{x}_6 \pm \pi\}$.

If we define multiple TSRs for a given manipulator, we extend our distance computation to evaluate distance to all relevant TSRs and return the smallest.

5.3 Direct Sampling of TSRs

When using TSRs to specify goal end-effector poses, it will be necessary to sample poses from TSRs. Sampling from a single TSR is done by first sampling a random value between each of the bounds defined by $\mathbf{B}^w$ with uniform probability. These values are then compiled in a displacement d^w_{sample} and converted into the transformation $\mathbf{T}^w_{sample}$. We can then convert this sample into world coordinates after applying the end-effector transform.

$$\mathbf{T}^0_{sample'} = \mathbf{T}^0_w \mathbf{T}^w_{sample} \mathbf{T}^w_e \tag{5.7}$$

We observe that while our method ensures a uniform sampling in the bounds of $\mathbf{B}^w$, it could produce a biased sampling in the subspace of constrained spatial displacements $SE(3)$ that $\mathbf{B}^w$ parameterizes. However this bias has not had a significant impact on the runtime or success-rate of our algorithms.

In the case of multiple TSRs specified for a single task, we must first decide which TSR to sample from. If the bounds of all TSRs enclose six-dimensional volumes, we can choose among TSRs in proportion to their volume. However a volume-proportional sampling will ignore TSRs that encompass volumes of less than six dimensions because they have no volume in the six-dimensional space. To address this issue we use a weighted sampling scheme that samples TSRs proportional to the *sum* of the differences between their bounds.

$$\zeta_i = \sum_{j=1}^{6} \left(\mathbf{B}^{w_i}_{j,2} - \mathbf{B}^{w_i}_{j,1} \right) \tag{5.8}$$

where ζ_i and $\mathbf{B}^{w_i}$ are the weight and bounds of the ith TSR, respectively. Sampling proportional to ζ_i allows us to sample from TSRs of any dimension except 0 while giving preference to TSRs that encompass more volume. TSRs of dimension 0, i.e. points, are given a fixed probability of being sampled. In general, any sampling scheme for selecting a TSR can be used as long as there is a non-zero probability of selecting any TSR.

5.4 Planning with TSRs as Goal Sets

TSRs can be used to sample goal end-effector placements of a manipulator, as would be necessary in a grasping or object-placement task. The constraint for using TSRs in this way is

$$\{C(q) = \text{DistanceToTSR}(q), \quad s = [1]\}. \tag{5.9}$$

Where the DistanceToTSR function implements the method of Section 5.2 and s refers to the domain of the constraint (Chapter 4).

To generate valid configurations in the $\mathcal{M}_C$ corresponding to this constraint, we can use direct sampling of TSRs (Section 5.3) and pass the sampled pose to an IK solver to generate a valid configuration. In order to ensure that we don't exclude any part of the constraint manifold, the IK solver used should not exclude any configurations from consideration. This can be achieved using an analytical IK solver for manipulators with six or fewer DOF. For manipulators with more than six DOF, we can use a pseudo-analytical IK solver, which discretizes or samples all but six joints.

Alternatively, we can use the projection strategy to sample the manifold. This would take the form of an iterative IK solver, which starts at some initial configuration. This configuration should be randomized to ensure exploration of the constraint manifold. Note that this strategy is prone to local minima and can be relatively slow to compute, so we use it only when an analytical or pseudo-analytical IK solver is not available (for instance with a humanoid).

Of course the same definition and strategies apply to sampling starting configurations as well as goal configurations.

5.5 Planning with TSRs as Pose Constraints

TSRs can also be used for planning with constraints on end-effector pose for the entire path. The constraint definition for such a use of TSRs differs from Equation 5.9 in the domain of the constraint

$$\{C(q) = \text{DistanceToTSR}(q), \quad s = [0, 1]\}. \tag{5.10}$$

Since the domain of this constraint spans the entire path, the planning algorithm must ensure that each configuration it deems valid lies within the constraint manifold. While the rejection strategy can be

Algorithm 1: $\mathbf{J}^{+}$Projection(q)

```
while true do
    Δx ← DisplacementFromTSR(q);
    if ‖Δx‖ < ε then
        return q;
    end
    J ← GetJacobian(q);
    Δq_error ← J^T (JJ^T)^{-1} Δx;
    q ← (q − Δq_error);
end
```

used to generate valid configurations for TSRs whose bounds encompass a six-dimensional volume, the projection strategy can be used for all TSRs.

One method of projection for TSRs is shown in Algorithm 1. This method uses the Jacobian pseudo-inverse ($\mathbf{J}^{+}$) [49] to iteratively move a given configuration to the constraint manifold defined by a TSR.

The DisplacementFromTSR function returns the displacement from q to a TSR, i.e. the result of Equation 5.6. The GetJacobian function computes the Jacobian of the manipulator at q. Though Algorithm 1 describes the projection conceptually, in practice we must also take into account the issues of step-size, singularity avoidance, and joint limits when projecting configurations. We will show, in Chapter 9, that the distribution of samples generated on the constraint manifold by this projection operator covers the manifold, which is a necessary property for probabilistic completeness.

It is important to note that we can also use the method of Section 5.3 to generate samples directly from TSRs and then compute IK to obtain configurations that place the end-effector at those samples. Such a strategy would be especially useful when planning in task space, i.e. the parameter space of pose constraints, instead of C-space because it would allow the task space to be explored while providing configurations for each task-space point (similar to [13]). However, we prefer the completeness properties of C-space planners, so we focus on those in this thesis.

Chapter 6

Task Space Region Chains

While we showed that TSRs are intuitive to specify, can be quickly sampled, and the distance to TSRs can be evaluated efficiently, a single TSR, or even a finite set of TSRs, is sometimes insufficient to capture the pose constraints of a given task. To describe more complex constraints such as manipulating articulated objects, this chapter introduces the concept of TSR Chains, which are defined by linking a series of TSRs. Though direct sampling of TSR Chains follows clearly from that of TSRs, the distance metric for TSR Chains is extremely different.

To motivate the need for a more complex representation consider the task of opening a door while allowing the end-effector to rotate about the door handle (see Figure 6.1). It is straightforward to specify the rotation of the door about its hinge as a single TSR and to specify the rotation of the end-effector about the door's handle as a single TSR if the door's position is fixed. However, the product of these two constraints (allowing the end-effector to rotate about the handle for any angle of the hinge) cannot be completely specified with a finite set of TSRs. In order to allow more complex constraint representations in the TSR framework, we present TSR Chains, which are constructed by linking a series of TSRs.

6.1 TSR Chain Definition

A TSR chain $\mathbf{C} = \{\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_n\}$ consists of a set of n TSRs with the following additional property

$$\mathbf{C}_i.\mathbf{T}^0_w = (\mathbf{C}_{i-1}.\mathbf{T}^0_w)(\mathbf{C}_{i-1}.\mathbf{T}^w_{sample})(\mathbf{C}_{i-1}.\mathbf{T}^e_w) \tag{6.1}$$

The work discussed in this chapter was published in [89] and [88].

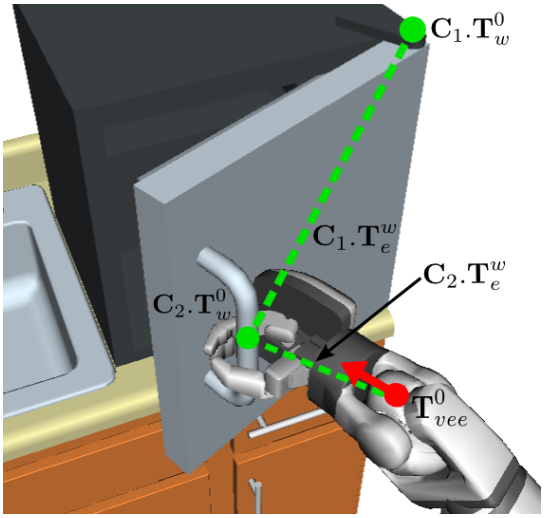


Figure 6.1: The virtual manipulator for the door example. The green dotted lines represent the links of the virtual manipulator and the red dot and arrow represent the virtual end-effector, which is at transform $\mathbf{T}_{vee}^0$.

for $i = \{2...n\}$ where $\mathbf{C}_i$ corresponds to the ith TSR in the chain and $\mathbf{C}_i.\{\cdot\}$ refers to an element of the ith TSR. Of course a TSR Chain can consist of only one TSR, in which case it is identical to a normal TSR. $\mathbf{C}_i.\mathbf{T}_{sample}^w$ can be any transform obtained by sampling from inside the bounds of $\mathbf{C}_i.\mathbf{B}^w$. Thus we do not know $\mathbf{C}_i.\mathbf{T}_w^0$ until we have determined $\mathbf{T}_{sample}^w$ values for all previous TSRs in the chain. By coupling TSRs in this way the TSR Chain structure can represent constraints that would otherwise require an infinite number of TSRs to specify.

A TSR chain can also be thought of as a virtual serial-chain manipulator. Again consider the door example. To define the TSR chain for this example, we can imagine a virtual manipulator that is rooted at the door's hinge. The first link of the virtual manipulator rotates about the hinge and extends from the hinge to the handle. At the handle, we define another link that rotates about the handle and extends to where a robot's end-effector would be if the robot were grasping the handle (see Figure 6.1). $\mathbf{C}_1.\mathbf{T}_{sample}^w$ would be a rotation about the door's hinge corresponding to how much the door had been opened. In this way, we could see the $\mathbf{T}_{sample}^w$ values for each TSR as transforms induced by the "joint angles" of the virtual manipulator. The joint limits of these virtual joints are defined by the values in $\mathbf{B}^w$.

6.2 Direct Sampling From TSR Chains

To directly sample a TSR Chain we first sample from within $\mathbf{C}_1.\mathbf{B}^w$ to obtain $\mathbf{C}_1.\mathbf{T}_{sample}^w$. This is done by sampling uniformly between the bounds in $\mathbf{B}^w$, compiling the sampled values into a displacement $d_{sample}^w = [x\ y\ z\ \psi\ \theta\ \phi]$ and converting that displacement to the transform $\mathbf{C}_1.\mathbf{T}_{sample}^w$. We then use this

sample to determine $\mathbf{C}_2.\mathbf{T}_w^0$ via Equation 6.1. We repeat this process for each TSR in the chain until we reach the nth TSR. We then obtain a sample in the world frame

$$\mathbf{T}_{sample'}^0 = (\mathbf{C}_n.\mathbf{T}_w^0)(\mathbf{C}_n.\mathbf{T}_{sample}^w)(\mathbf{C}_n.\mathbf{T}_e^w). \tag{6.2}$$

Note that the sampling of TSR chains in this way is biased but the sampling will cover the entire set. To see this, imagine a virtual manipulator with many links. It can be readily seen that many sets of different joint values (essentially $\mathbf{T}_{sample}^w$ values) of the virtual manipulator will map to the same end-effector transform. However, if the virtual manipulator's end-effector is at the boundary of the virtual manipulator's reachability, only one set of joint values maps to the end-effector pose (when the manipulator is fully outstretched). Thus some $\mathbf{T}_{sample'}^0$ values can have a higher chance of being sampled than others, depending on the definition of the TSR Chain. Clearly a uniform sampling would be ideal but we have found that this biased sampling is sufficient for the practical tasks we consider.

If there is more than one TSR Chain defined for a single manipulator, this means that we have the option of drawing a sample from any of these TSR Chains. We choose a TSR Chain for sampling with probability proportional to the sum of the differences between the bounds of all TSRs in that chain.

6.3 Distance to TSR Chains

Though the sampling method for TSR Chains follows directly from the sampling method for TSRs, evaluating distance to a TSR Chain is fundamentally different from evaluating distance to a TSR. This is because we do not know which $\mathbf{T}_{sample}^w$ values for each TSR in the chain yield the minimum distance to a query transform $\mathbf{T}_s^0$ (derived from a query configuration q_s using forward kinematics).

To approach this problem, it is again useful to think of the TSR chain as a virtual manipulator (See Figure 6.2a). Finding the correct $\mathbf{T}_{sample}^w$ values for each TSR is equivalent to finding the joint angles of the virtual manipulator that bring its virtual end-effector as close to $\mathbf{T}_s^0$ as possible. Thus we can see this distance-checking problem as a form of the standard IK problem, which is to find the set of joint angles that places an end-effector at a given transform. Depending on the TSR Chain definition and $\mathbf{T}_s^0$, the virtual manipulator may not be able to reach the desired transform, in which case we want the virtual end-effector to get as close as possible. Thus we can apply standard iterative IK techniques based on the Jacobian pseudo-inverse to move the virtual end-effector to a transform that is as close as possible to $\mathbf{T}_s^0$ (see Figure 6.2b). Once we obtain the joint angles of the virtual manipulator, we convert them to $\mathbf{T}_{sample}^w$ values and forward-chain to obtain the virtual end-effector position $\mathbf{T}_{vee}^0$. We then convert $\mathbf{T}_s^0$ to the virtual end-effector's frame

$$\mathbf{T}_s^{vee} = (\mathbf{T}_{vee}^0)^{-1}\mathbf{T}_s^0 \tag{6.3}$$

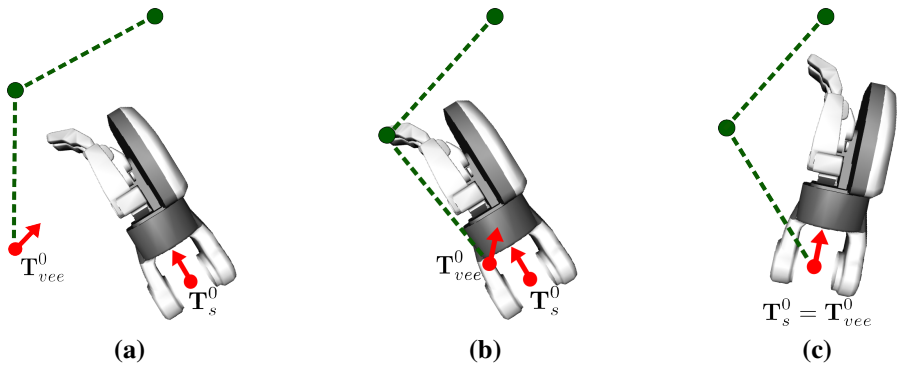


Figure 6.2: Depiction of the IK handshaking procedure. (a) The virtual manipulator starts in some configuration. (b) Finding the closest configuration of the virtual manipulator. (c) The robot's manipulator moves to meet the constraint.

and then convert to the displacement form

$$d_s^{vee} = \begin{bmatrix} \mathbf{t}_s^{vee} \\ \arctan 2(\mathbf{R}_{s32}^{vee}, \mathbf{R}_{s33}^{vee}) \\ -\arcsin(\mathbf{R}_{s31}^{vee}) \\ \arctan 2(\mathbf{R}_{s21}^{vee}, \mathbf{R}_{s11}^{vee}) \end{bmatrix}. \tag{6.4}$$

$\|d_s^{vee}\|$ is the distance between $\mathbf{T}_s^0$ and $\mathbf{T}_{vee}^0$.

Once the distance is evaluated we can employ the projection strategy by calling the IK algorithm for the robot's manipulator to move the robot's end-effector to $\mathbf{T}_{vee}^0$ to meet the constraint specified by this TSR Chain (Figure 6.2c). We term this process of calling IK for the virtual manipulator and the robot in sequence *IK handshaking*.

Just as with TSR Chains used for sampling, we may define more than one TSR Chain as a constraint for a single manipulator. This means that we have the option of satisfying any of these TSR Chains to produce a valid configuration. To find which chain to satisfy, we perform the distance check from our current configuration to each chain and choose the one that has the smallest distance.

6.4 Physical Constraints

In the door example, the first TSR corresponds to a physical joint of a body in the environment but the second one is purely virtual; i.e. defining a relation between two frames that is not enforced by a joint

in the environment (in this case the relation is between the robot's end-effector and the handle of the door). It is important to note that TSR Chains inherently accommodate such mixing of real and virtual constraints. In fact a TSR Chain can consist of purely virtual or purely physical constraints. However, when planning with TSR Chains, special care must be taken to ensure that any physical joints (such as the door's hinge) be synchronized with their TSR Chain counterparts. This is done by including the configuration of any physical joints corresponding to elements of TSR Chains in the configuration space searched by the planner (see Section 7.3).

In the case that the physical constraints included in the TSR Chain form a redundant manipulator, the inverse-kinematics algorithm for the TSR Chain should be modified to account for the physical properties of the chain. For instance, if the chain is completely passive, a term that that minimizes the potential energy of the chain should be applied in the null-space of the Jacobian pseudo-inverse to find a local minimum-energy configuration of the chain. In general, chains can have various physical properties that may not be easy to account for using an IK solver. In that case, we recommend a physical simulation of the movement of the end-effector from it's initial pose to $\mathbf{T}^0_{vee}$ as it is being pulled by the robot to find the resting configuration of the chain.

6.5 Notes on Implementation

Whenever we create a TSR Chain, we also create its virtual manipulator in simulation so that we can perform IK on this manipulator and get the location of the virtual end-effector. When we refer to the joint values of a TSR Chain, we are actually referring to the joint values of that TSR Chain's virtual manipulator. Also, to differentiate whether a TSR Chain should be used for sampling goals or constraining configurations or both, we specify how the chain should be used in its definition. When inputing TSR Chains into our planner, we specify which manipulator of the robot they correspond to as well as any physical DOF that correspond to elements of the chain.

Chapter 7

The CBiRRT2 Algorithm

This chapter describes the Constrained BiDirectional RRT (CBiRRT2) planner, which is capable of planning with TSR Chains among other constraints. Since the representation of TSR Chains subsumes that of TSRs, the planner can incorporate the uses of TSRs already described. In this chapter we describe the operations of the planner. Several example problems as well as CBiRRT2's performance on these problems are shown in Chapter 8. We prove the probabilistic completeness of CBiRRT2 when planning with pose constraints in Chapter 9. For some notes on implementing several components of CBiRRT2, see Appendix A.

7.1 Planner Operation

CBiRRT2 takes into account constraints on the configuration of the robot during its path as well as constraints on the goal configuration of the robot. Constraints on the poses and goal locations of the robot's end-effectors are specified as TSR Chains.

CBiRRT2 operates by growing two trees in the C-space of the robot (please follow the explanation below in Algorithm 2). At each iteration, CBiRRT2 chooses between one of two modes: exploration of the C-space using the two trees or direct sampling from a set of TSR Chains. The probability of choosing to sample is defined by the parameter P_{sample}.

If the algorithm chooses to sample, it calls the AddRoot function, which tries to inject a goal configuration into the backward tree T_{goal}. If the algorithm chooses to explore the C-space, one of the trees grows a branch toward a randomly-sampled configuration q_{rand} using the ConstrainedExtend function.

The work discussed in this chapter was published in [89] and [88].

The branch grows as far as possible toward q_{rand} but may be stalled due to collision or constraint violation and will terminate at q^a_{reach}. The other tree then grows a branch toward q^a_{reach}, again growing as far as possible toward this configuration. If the other tree reaches q^a_{reach}, the trees have connected and a path has been found. If not, the trees are swapped and the above process is repeated.

The ConstrainedExtend function (see Algorithm 3) iteratively moves from a configuration q_{near} toward a configuration q_{target} with a step size of Δq_{step}. After each step toward q_{target}, the function checks if the new configuration q_s has reached q_{target} or if it is moving farther from q_{target}, in either case the function terminates. If the above conditions are not true then the algorithm takes a step toward q_{target} and passes the new q_s to the ConstrainConfig function, which is problem-specific. If ConstrainConfig is able to project q_s to a constraint manifold, the new q_s is added to the tree and the stepping process is repeated. Otherwise, ConstrainedExtend terminates (see Figure 7.1 for an illustration). ConstrainedExtend always returns the last configuration reached by the extension operation. The c vector is a vector of TSR Chain joint values of all TSR Chains. Every q_s has a corresponding c which is stored along with q_s in the tree. We store the c vector so that the ConstrainConfig function has a good initial guess of the TSR chain joint values when taking subsequent steps. This greatly decreases the time used by the inverse-kinematics solver inside ConstrainConfig.

Algorithm 2: CBiRRT2(Q_s, Q_g)

```
1  T_a.Init(Q_s); T_b.Init(Q_g);
2  while TimeRemaining() do
3      T_goal = GetBackwardTree(T_a, T_b);
4      if size(T_goal) = 0 or rand(0,1) < P_sample then
5          AddRoot(T_goal);
6      else
7          q_rand ← RandomConfig();
8          q^a_near ← NearestNeighbor(T_a, q_rand);
9          q^a_reach ← ConstrainedExtend(T_a, q^a_near, q_rand);
10         q^b_near ← NearestNeighbor(T_b, q^a_reached);
11         q^b_reach ← ConstrainedExtend(T_b, q^b_near, q^a_reach);
12         if q^a_reach = q^b_reach then
13             P ← ExtractPath(T_a, q^a_reach, T_b, q^b_reach);
14             return ShortenPath(P);
15         else
16             Swap(T_a, T_b);
17         end
18     end
19 end
20 return ∅;
```

Algorithm 3: ConstrainedExtend(T, q_{near}, q_{target})

$q_s \leftarrow q_{near}$; $q_s^{old} \leftarrow q_{near}$;
while *true* **do**
  if $q_{target} = q_s$ **then**
    return q_s;
  else if $\|q_{target} - q_s\| > \|q_s^{old} - q_{target}\|$ **then**
    return q_s^{old};
  end
  $q_s^{old} \leftarrow q_s$;
  $q_s \leftarrow q_s + \min(\Delta q_{step}, \|q_{target} - q_s\|) \frac{(q_{target} - q_s)}{\|q_{target} - q_s\|}$;
  $c \leftarrow \text{GetConstraintValues}(T, q_s^{old})$;
  $\{q_s, c\} \leftarrow \text{ConstrainConfig}(q_s^{old}, q_s, c, \emptyset)$;
  if $q_s \neq \emptyset$ **then**
    T.AddVertex(q_s, c);
    T.AddEdge(q_s^{old}, q_s);
  else
    return q_s^{old};
  end
end

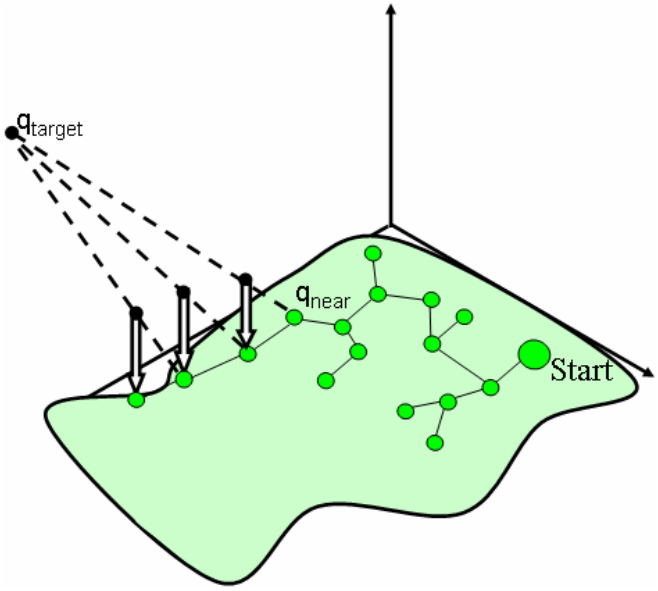


Figure 7.1: Depiction of one **ConstrainedExtend** operation. The operation starts at q_{near}, which is a node of a search tree on the constraint manifold and iteratively moves toward q_{target}, which is a configuration sampled from the C-space. Each step toward q_{target} is constrained using the ConstrainConfig function to lie on the constraint manifold.

After a path is found, we shorten it using the ShortenPath function. This function implements the

Algorithm 4: ShortenPath(P)

```
while TimeRemaining() do
    T_shortcut ← {};
    i ← RandomInt(1, size(P) − 1);
    j ← RandomInt(i, size(P));
    q_reach ← ConstrainedExtend(T_shortcut, P_i, P_j);
    if q_reach = P_j and Length(T_shortcut) < Length(P_i ··· P_j) then
        P ← [P_1 ··· P_i, T_shortcut, P_{j+1} ··· P.size];
    end
end
return P;
```

Algorithm 5: AddRoot(T)

```
for i = 1...m do
    ℂ ← GetTSRChainsForManipulator(i);
    {T^0_targ, c} ← SampleFromTSRChains(ℂ);
    Targets.AddTarget(T^0_targ, i);
end
{q_s, c} ← GetInitialGuess();
{q_s, c} ← ConstrainConfig(∅, q_s, c, Targets);
if q_s ≠ ∅ then
    T.AddVertex(q_s, c);
end
```

popular "short-cut" method to iteratively shorten the path (Algorithm 4). However, instead of using straight lines which would violate constraints, we use the ConstrainedExtend function (Algorithm 3) for each short-cut. Using ConstrainedExtend guarantees that constraints will be met along the shortened path. Also, it is important to note that a short-cut generated by ConstrainedExtend between two nodes is not necessarily the shortest path between them because the nodes may have been projected in an arbitrary way. This necessitates checking whether $\mathrm{Length}(P_{shortcut})$ is shorter than the original path between i and j.

Note that CBiRRT2 can also be seeded with multiple start and goal configurations (Q_s/Q_g). If no goals are specified, the AddRoot function will insert the first goal into the backward tree. In fact, the AddRoot function can be called for both the start and the goal trees, if this is desired.

7.2 Planning with TSR Chains

Accounting for TSR Chains is done in the AddRoot and ConstrainConfig functions. When CBiRRT2 chooses to sample a goal configuration, it calls the AddRoot function (see Algorithm 5). This function retrieves the relevant set of TSR Chains for each manipulator and samples a target transform for each manipulator using the SampleFromTSRChain function, which is an implementation of the methods described in Section 6.2. It then forms an initial guess of the robot's joint values and c and calls the ConstrainConfig function. In practice we usually use the initial configuration of the robot and vector of zeros for c as the guess but these can be randomized as well. If the ConstrainConfig does not return

Algorithm 6: ConstrainConfig(q_s^{old}, q_s, c, Targets)

```
1  CheckDist = False;
2  if Targets = ∅ then
3  |   CheckDist = True;
4  |   for i = 1...m do
5  |   |   ℂ ← GetTSRChainsForManipulator(i);
6  |   |   T_s^0 ← GetEndEffectorTransform(q_s, i);
7  |   |   {T_targ^0, c} ← GetClosestTransform(ℂ, T_s^0, c);
8  |   |   Targets.AddTarget(T_targ^0, i);
9  |   end
10 end
11 q_s ← UpdatePhysicalConstraintDOF(q_s, c);
12 q_s ← ProjectConfig(q_s, Targets);
13 if q_s = ∅ or
14    (CheckDist and |q_s − q_s^old| > 2Δq_step) then
15 |   return ∅;
16 end
17 return {q_s, c};
```

$\emptyset$, the resulting q_s and corresponding c are added to the tree.

The ConstrainConfig function is problem-specific, an example of a ConstrainConfig function that considers *only* TSR Chains is given in Algorithm 6. If ConstrainConfig is not passed a set of targets (i.e. it is called from ConstrainedExtend instead of AddRoot), then it generates a set of targets for each manipulator using the GetClosestTransform function, which is an implementation of the methods described in Section 6.3. Note that this function also updates the c vector with the joint values of the TSR Chain that generated the closest transform. The c values for the TSR Chains that did not yield the closest transform to $\mathbf{T}_s^0$ are not updated. After the target transforms for each manipulator are obtained, ProjectConfig projects the configuration of the robot using standard inverse-kinematics algorithms based on the Jacobian pseudo-inverse to produce a q_s which meets the constraints represented by the TSR Chains. This completes the IK handshaking process described in Section 6.3.

If ConstrainConfig was called by AddRoot, the distance between q_s^{old} and q_s is unimportant. However, we do not wish for q_s to be too far from q_s^{old} when extending using ConstrainedExtend because the intermediate configurations are not likely to meet the constraints. Thus we enforce a small step size to reduce deviation from constraints between nodes.

In most situations, we are also interested in satisfying other constrains such as balance and collision using the rejection strategy. Checks for these constraints should be inserted at line 14 of ConstrainConfig.

7.3 Augmenting Configuration with States of Physical DOF

Because TSR chains can specify constraints corresponding to physical degrees of freedom of objects in the world (such as the hinge of a door) as well as purely virtual constraints, we have to account for physical DOF when checking collision and measuring distances in the C-space. To achieve this, we include the configuration of all physical DOF in the configuration vector q. We set these DOF by extracting their values from the vector of all the TSRChains' virtual manipulator joint values c using the UpdatePhysicalConstraintDOF function. This is done on line 11 of the ConstrainConfig function. Note that these DOF are not affected by the ProjectConfig function.

7.4 Parameters

One of the strengths of CBiRRT2 is that it uses only three parameters, all of which require minimal tuning. The first parameter encodes the RRT step size Δq_{step}. Δq_{step} can be increased to speed up planning or decreased to allow finer motions but we have found that tuning this parameter is rarely necessary for the manipulation tasks we consider.

The numerical error allowed in meeting a pose constraint ϵ (in Algorithm 1) is necessitated by the numerical nature of our projection operator. Our projection method is quite accurate, so CBiRRT2 performs well even for small values of ϵ.

Finally, the third parameter P_{sample} is only used when goal sampling is required. A higher P_{sample} biases CBiRRT2 toward goal sampling, a lower one biases it toward building paths. We showed in [87] that the algorithm performs well for a wide range of values for P_{sample}, though we recommend setting the value to be low because building paths usually requires more computation than sampling an adequate goal in our problem domain.

Chapter 8

Example Problems

This chapter describes several example problems and the constraints specified for those problems as well as results for running CBiRRT2 in simulation and experiments on physical hardware. The first six examples are implemented on a 7DOF Barrett arm and the last three on the 28DOF of the HRP3 robot. The first three examples describe how to use TSRs for goal pose specification. The next three examples show how to use TSRs as pose constraints. The last three examples show how to mix goal and pose TSRs and TSR Chains. At the end of this chapter we analyze the computational cost of the operations of CBiRRT2 on a door-opening task for both robots.

Since the TSR Chain representation subsumes the TSR representation, each example problem can be implemented using TSR Chains. However we do not describe a chained implementation when only chains of length 1 are used so that the explanation is clearer. All paths were planned to be collision-free using the rejection strategy. All experiments were performed on a 2.4 GHz Intel CPU with 4 GB of RAM using the OpenRAVE simulation and planning environment [91]. The numerical error allowed in meeting a pose constraint was $\epsilon = 0.001$.

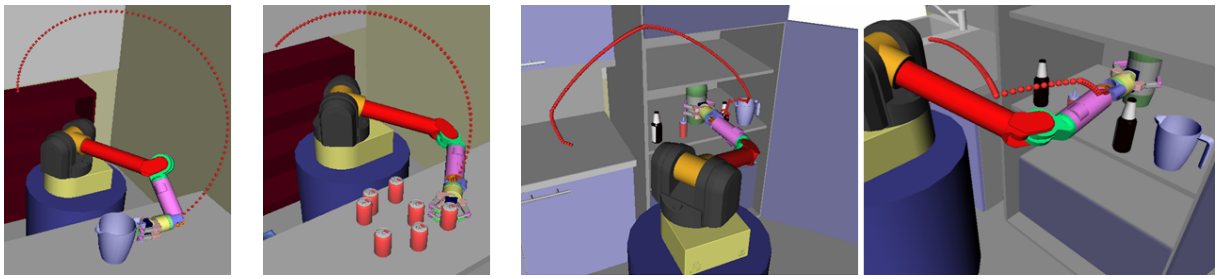
8.1 Reaching to Grasp an Object

Our goal in this problem is to grasp an object for which we can define a continuum of acceptable grasp poses. These grasp poses can be encoded into TSRs and passed to our planner. We define four TSRs for the pitcher we wish to grasp (see Figure 8.1(a)): two for the top of the pitcher and two for the handle. The $\mathbf{T}_w^0$ and $\mathbf{T}_e^w$ transforms of these TSRs are shown in Figure 5.1. The two bounds for the top TSRs are identical, as are the two bounds for the handle TSRs.

The work discussed in this chapter was published in [90], [87], [89], and [88].

$$\mathbf{B}^w_{top} = \begin{bmatrix} \mathbf{0}_{5\times 2} \\ -0.3 \quad 0.3 \end{bmatrix} \qquad \mathbf{B}^w_{handle} = \begin{bmatrix} \mathbf{0}_{2\times 2} \\ -0.03 \quad 0.02 \\ \mathbf{0}_{3\times 2} \end{bmatrix} \tag{8.1}$$

The top TSRs allow the robot to grasp the pitcher from the top with limited hand rotation about the z-axis. The handle TSRs allow the robot to grasp the pitcher anywhere along the handle but do not allow any offset in hand rotation. Trajectories produced by our planner are shown in Figure 8.1(a). The RRT step-size Δq_{step} was set to 0.05 and $P_{sample} = 0.1$. The average planner runtime for 15 trials was 0.04s.



(a) **(b)** **(c)**

Figure 8.1: Paths of the end-effector produced by CBiRRT2 for the three goal TSR examples. (a) Reaching to grasp a pitcher. (b) Reaching to grasp one of many soda cans. (d) Placing a bottle into the refrigerator left: fixed base, right: mobile base. The paths shown have been smoothed with 500 iterations of the shortcut smoothing algorithm.

8.2 Reaching to Grasp Multiple Objects

In this problem the robot's task is to reach and grasp one of seven randomly-placed soda cans on a table (see Figure 8.1(b)). Each soda can is treated as a cylinder and two TSRs are defined for each can. The $\mathbf{T}^0_w$ and $\mathbf{T}^w_e$ transforms are shown in Figure 5.1. Both TSRs for each can have identical bounds

$$\mathbf{B}^w = \begin{bmatrix} \mathbf{0}_{5\times 2} \\ -\pi \quad \pi \end{bmatrix}. \tag{8.2}$$

These bounds allow the grasp to rotate about the z-axis of the can, thus allowing it to grasp the can from any direction in the plane defined by the x and y coordinates of the can's center. Note that we do not specify which soda can to grasp, this choice is made within the planner when sampling from the TSRs. Trajectories produced by our planner are shown in Figure 8.1(b). $\Delta q_{step} = 0.05$ and $P_{sample} = 0.1$. The average planner runtime for 15 trials was 0.21s.

8.3 Placing an Object into a Cluttered Space

The task in this problem is for the robot to place the bottle it is holding into a very cluttered location (see Figure 8.1(d)). The bottles in the refrigerator and the upper refrigerator shelf make it difficult for the robot to find a path that places the large bottle it is holding onto the middle refrigerator shelf. $\mathbf{T}_w^0$ is defined at the center of the middle shelf and $\mathbf{T}_e^w$ is defined as an end-effector position pointing along the y axis (away from the robot) that is holding the bottle at $\mathbf{T}_w^0$.

$$\mathbf{B}^w = \begin{bmatrix} -0.24 & 0.24 \\ -0.34 & 0.34 \\ \mathbf{0}_{4\times 2} \end{bmatrix} \tag{8.3}$$

This $\mathbf{B}^w$ defines a plane on the shelf where the bottle can be placed. $\Delta q_{step} = 0.05$ and $P_{sample} = 0.1$. The average planner runtime for 15 trials was 93s.

8.4 The Maze Puzzle

In this problem, the robot arm must solve a maze puzzle by drawing a path through the maze with a pen (see Figure 8.2). The constraint is that the pen must always be touching the table however the pen is allowed to pivot about the contact point up to an angle of α in both roll and pitch. We define the end-effector to be at the tip of the pen with no rotation relative to the world frame. To specify the constraint in this problem, we define one pose constraint TSR with $\mathbf{T}_w^0$ to be at the center of the maze with no rotation relative to the world frame (z being up). $\mathbf{T}_e^w$ is identity and

$$\mathbf{B}^w = \begin{bmatrix} -\infty & \infty \\ -\infty & \infty \\ 0 & 0 \\ -\alpha & \alpha \\ -\alpha & \alpha \\ -\pi & \pi \end{bmatrix}. \tag{8.4}$$

This example is meant to demonstrate that CBiRRT2 is capable of solving multiple narrow passage problems while still moving on a constraint manifold. It is also meant to demonstrate the generality of the CBiRRT2; no special-purpose planner is needed even for such a specialized task.

IK solutions were generated for both the start and goal position of the pen using the given grasp and input as Q_s and Q_g. The values in Table 8.1 represent the average of 10 runs for different α values.

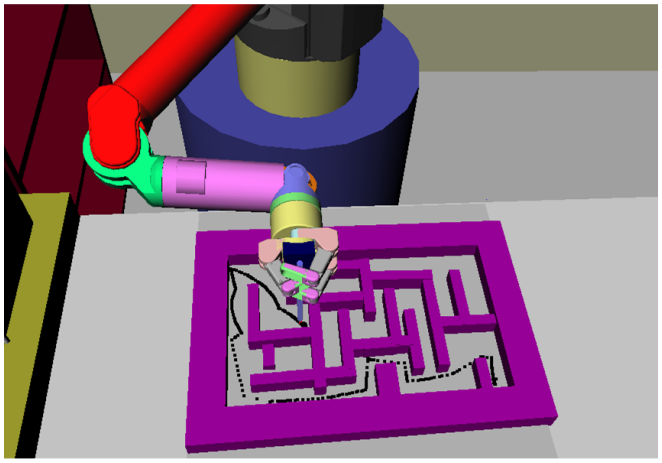


Figure 8.2: A path found for the maze puzzle using $\alpha = 0.4$rad. The black points represent positions of the tip of the pen along the path.

Runtimes with a "$>$" denote that there was at least one run that did not terminate before 120 seconds. For such runs, 120 was used in computing the average. The RRT step-size Δq_{step} was set to 0.05. No goal sampling is performed in this example.

α(rad.)	0.0	0.1	0.2	0.3	0.4	0.5
Avg. Runtime	$>$83.5s	$>$58.8s	$>$49.0s	19.5s	14.3s	15.2s
Success Rate	40%	60%	90%	100%	100%	100%

Table 8.1: Simulation Results for Maze Puzzle

The shorter runtimes and high success rates for larger α values demonstrate that the more freedom we allow for the task, the easier it is for the algorithm to solve it. This shows a key advantage of formulating the constraints as bounds on allowable pose as opposed to requiring the pose of the object to conform exactly to a specified value, as in [10]. For problems where we do not need to maintain an exact pose for an object we can allow more freedom, which makes the problem easier. See Figure 8.2 for an example path of the tip of the pen.

8.5 Heavy Object with Sliding Surfaces

In this problem the task is to move a heavy dumbbell from a start position to a given goal position (see Figure 8.3). The weight of the object is known to the planner but the planner does not know what configurations entail acceptable torques a priori. Sliding surfaces are also provided so that the planner may use these to support the object if necessary. The planner is allowed to slide the object along a

sliding surface or to hold the object if the torques in the holding configuration are within torque limits. The constraint on torque is formulated as

$$\left\{ C(q) = \begin{cases} 1 & \text{if InTorqueLimits}(q) \\ 0 & \text{otherwise} \end{cases}, \forall q \in \tau(\cdot) \right\}. \tag{8.5}$$

We will be employing the rejection strategy with respect to torque constraints, so we need to calculate the torques on the joints in a given q. This is done using standard Recursive Newton-Euler techniques described by Walker and Orin [92]. To incorporate the object into the robot model, we take a weighted average of the centers of mass of the end-effector and the object and set that as the mass and center of mass of the end-effector. We will refer to the combined mass as m. Note that this formulation only takes into account the torque necessary to maintain a given q, i.e. it assumes the robot's motion is quasi-static. The end-effector frame is defined to be at the bottom of the dumbbell.

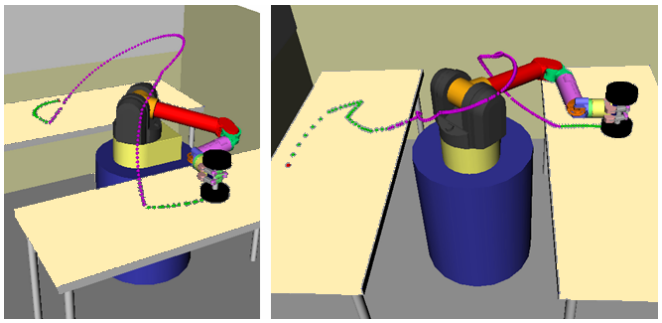


Figure 8.3: Two trajectories for heavy object sliding using a fixed base. Left: Trajectory found for $m = 7$kg, Right: Trajectory found for $m = 12$kg. Green nodes represent points in the trajectory where the dumbbell is sliding, purple nodes represent points where the dumbbell is not sliding. In the 7kg case, completing the task does not require much sliding of the dumbbell and it can be lifted high above the robot. In the 12kg case the weight of the dumbbell prohibits lifting it above the robot and necessitates more sliding along the tables.

Each sliding surface is a rectangle of known width and length with an associated surface normal. In general, the surfaces may be slanted so they may only support part of the object's weight, which is taken into account when calculating joint torques. Each sliding surface gives rise to a constraint manifold and there can be any number of sliding surfaces. To represent the sliding surfaces, we define pose constraint TSRs at the centers of each sliding surface with $\mathbf{T}_w^0$ at the center with the z axis oriented normal to the surface and $\mathbf{T}_e^w$ being identity. For each of these TSRs

$$\mathbf{B}^w = \begin{bmatrix} -\text{length}/2 & \text{length}/2 \\ -\text{width}/2 & \text{width}/2 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ -\pi & \pi \end{bmatrix}. \tag{8.6}$$

The ConstrainConfig function for this example differs slightly from the one in Algorithm 6. Instead of always satisfying one of the pose constraint TSR, the ConstrainConfig function for this example first checks if the configuration meets the torque constraint and if it does not, attempts to satisfy the closest pose constraint TSR in the same way as Algorithm 6 (see Figure 8.4). Finally, ConstrainConfig for this example checks if the projected configuration supports enough weight to ensure the torque constraint is met.

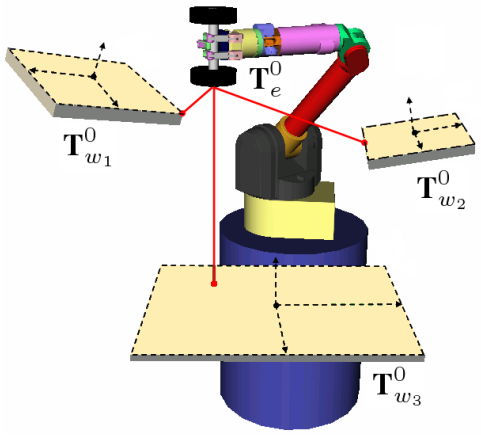


Figure 8.4: Depiction of finding the closest pose constraint TSR within ConstrainConfig for the heavy object sliding example. The shortest distance from $\mathbf{T}^0_e$ to $\mathbf{T}^0_{w_i}$ (computed using the DistanceToTSR function of Section 5.2) determines which TSR is chosen for projection.

We ran this example for both the fixed and mobile base cases. When planning with a mobile base, we allow translation of the base in x and y to be considered as two additional DOF of the robot. No non-holonomic constraints are placed on the base's motion. For the fixed base mode, we generate Q_s and Q_g the same way as in the Maze Puzzle. For the mobile base mode, we sampled 200 random base positions in a circle around the start and goal of the dumbbell and computed all IK solutions (to a 0.05rad discretization of the first joint) for each base position. All the collision-free IK solutions were input as Q_s and Q_g. The values in Table 8.2 represent the average of 10 runs for different weights of the dumbbell. The weight of the dumbbell was increased until the algorithm could not find a path within 120 seconds in any of the 10 runs. $\Delta q_{step} = 0.05$. No goal sampling is performed in this example.

The shorter runtimes and higher success rates for lower weights of the dumbbell match our expectations about the constraints induced by torque limits. As the dumbbell becomes heavier, the manifold of configurations with valid torque becomes smaller and thus finding a path through this manifold becomes more difficult. See Figure 8.3 for two sample trajectories illustrating this concept. The mobile base tends to not do as well as the fixed base in this example because the addition of the base's DOF expands the size of the C-space exponentially, thus making the problem more difficult.

We also implemented this problem on our physical WAM robot. Snapshots from three trajectories for three different weights are shown in Figure 8.5. As with the simulation environment, the robot

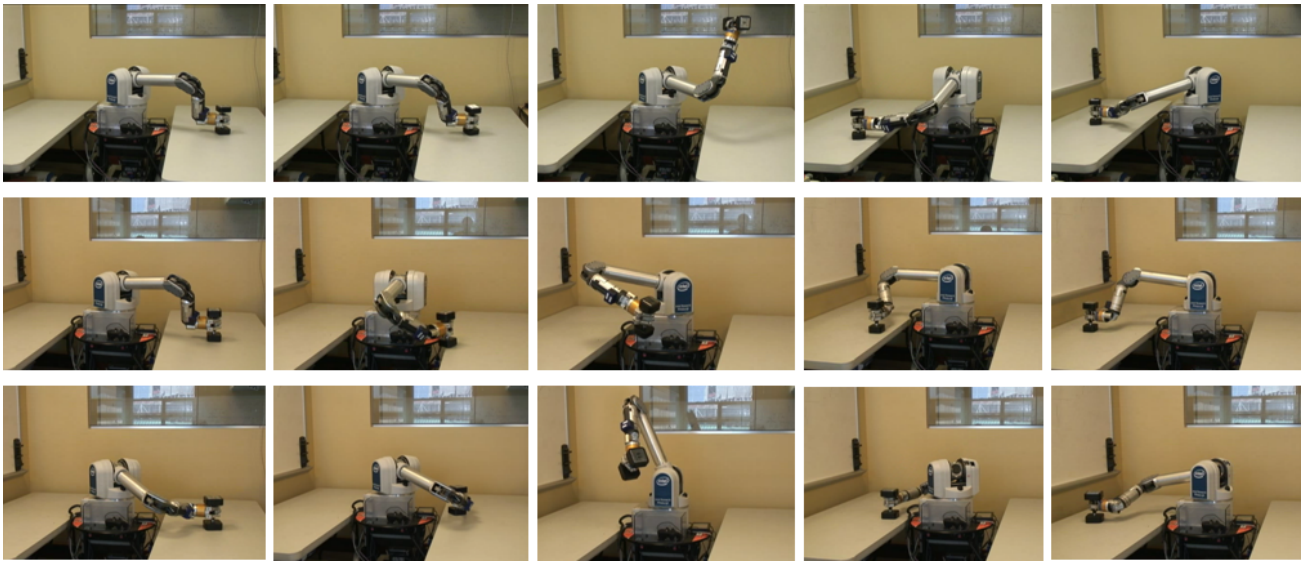


Figure 8.5: Experiments on the 7DOF WAM arm for three different dumbbells. Top Row: $m = 4.98$kg. Middle Row: $m = 5.90$kg. Bottom Row: $m = 8.17$kg. The trajectory for the lightest dumbbell requires almost no sliding, where as the trajectories for the heavier dumbbells slide the dumbbell to the edge of the table. Time proceeds from left to right.

Weight	7kg	8kg	9kg	10kg	11kg	12kg	13kg	14kg
Fixed Base								
Avg. Runtime	1.89s	2.06s	3.84s	5.51s	7.29s	12.4s	27.5s	>53.9s
Success Rate	100%	100%	100%	100%	100%	100%	100%	80%
Mobile Base								
Avg. Runtime	12.9s	22.1s	17.5s	33.5s	57.3s	>105s	>110s	>120s
Success Rate	100%	100%	100%	100%	100%	40%	40%	0%

Table 8.2: Simulation Results for Heavy Object Sliding

slid the dumbbell more when the weight was heavier and sometimes picked up the weight without any sliding for the mass of 4.98kg. Note that we take advantage of the compliance of our robot to help execute these trajectories but in general such trajectories should be executed using an appropriate force-feedback controller.

8.6 Heavy Object with Sliding Surfaces and Pose Constraint

This problem is similar to the previous one except that there is a constraint on the pose of the object throughout the task. The example we use for this kind of task is getting a pitcher of water out of a refrigerator and placing it on a counter (see Figure 8.6). Since the top of the pitcher is open, we must impose a constraint on the pose of the pitcher so that the water does not spill out. Again, we do not know a priori whether the pitcher is light enough to simply lift out of its start configuration or to place directly in its goal position without sliding. While this task is more complex than the previous one, it only requires one additional pose constraint TSR to enforce the no-spilling constraint. For this TSR $\mathbf{T}_w^0$ and $\mathbf{T}_e^w$ are set to identity and

$$\mathbf{B}^w = \begin{bmatrix} -\infty & \infty \\ -\infty & \infty \\ -\infty & \infty \\ 0 & 0 \\ 0 & 0 \\ -\pi & \pi \end{bmatrix}. \tag{8.7}$$

The ConstrainConfig function first attempts to meet the no-spilling constraint and then, if it does so successfully, executes the ConstrainConfig function of the previous example. Since we do not want to spill the water while sliding, only non-tilted sliding surfaces are considered for projection in this problem.

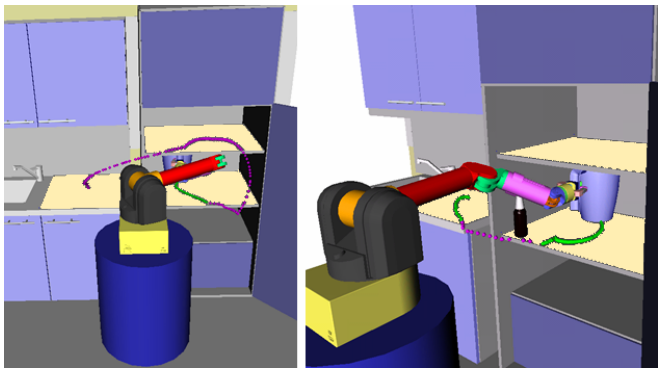


Figure 8.6: Left: Trajectory found for $m = 8$kg using a fixed base. Right: Trajectory found for $m = 12$kg using a mobile base. Green nodes represent points in the trajectory where the pitcher is sliding, purple nodes represent points where the pitcher is not sliding. When $m = 8$kg, the pitcher can be lifted over the body of the robot to avoid collision but when $m = 12$kg, the fixed base mode fails because the pitcher cannot be lifted over the body of the robot.

This example was also run for the fixed base and mobile base cases. Q_s and Q_g are generated the same way as in the previous example. The weight of the pitcher is incremented and runtimes are averaged

as with the previous example. The results are summarized in Table 8.3. The center of gravity of the pitcher was set to be the same as the center of gravity of the weight to make the results of the two problems comparable. $\Delta q_{step} = 0.05$. No goal sampling is performed in this example.

Weight	5kg	6kg	7kg	8kg	9kg	10kg	11kg	12kg
Fixed Base								
Avg. Runtime	2.79s	15.8s	18.1s	>39.1s	>120s	>120s	>120s	>120s
Success Rate	100%	100%	100%	90%	0%	0%	0%	0%
Mobile Base								
Avg. Runtime	31.2s	54.9s	57.8s	>78.3s	>104s	>80.9s	>90.7s	>115s
Success Rate	100%	100%	100%	90%	60%	80%	60%	20%

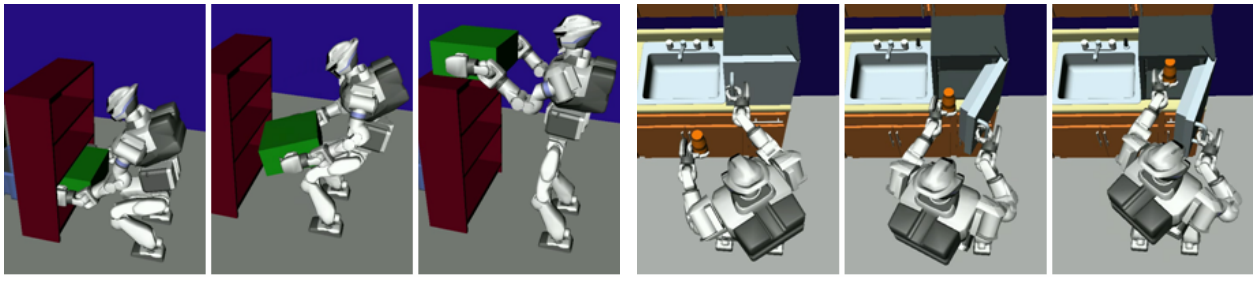
Table 8.3: Simulation Results for Heavy Object Sliding with Pose Constraint

The results of this problem demonstrate the advantages of having a mobile base in cluttered environments. The fixed base is placed close to both the start and goal locations of the object so that it can pull the object close to its body while keeping it on a sliding surface, thus maintaining a low torque when it lifts the object. However, in order to get from the refrigerator to the counter, the robot must lift the pitcher over its own body to avoid collisions, which requires more torque. This make it difficult to pick a position for the base that will work with larger weights because the base must be close enough to the refrigerator and counter to lift the object off of the sliding surfaces but must be far enough so that the robot can move the pitcher between the refrigerator and the counter without requiring a lot of torque. The mobile base is preferable in this situation because it can access both the refrigerator and the counter by moving the base closer to them and maintaining low torques for the arm. It can then drive from the refrigerator to the counter while keeping the arm in roughly the same position, thus requiring no increase in torque. See Figure 8.6 for sample trajectories both with and without the mobile base.

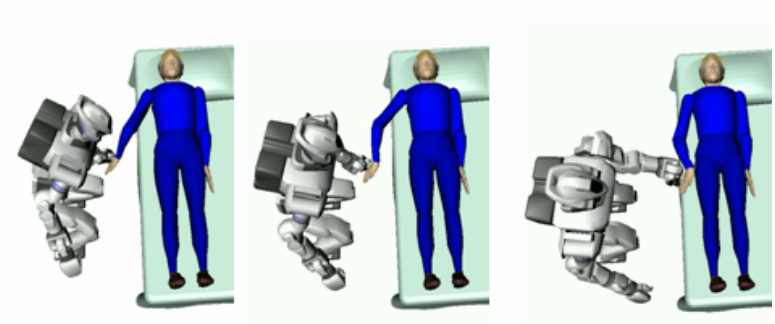
8.7 Closed Chain Kinematics

Some researchers approach the problem of planning with closed-chain kinematics by implementing specialized projection operators [16] or sampling algorithms [17]. However, in the TSR framework no special additions are required. In fact, closed chain kinematics are inherently enforced by TSR definitions.

Consider the problem shown in Figure 8.7a. The task is for the HRP3 humanoid to pick up the box from the bottom of the bookshelf and place it on top. There are two closed chains which must be enforced by the planner; the legs and arms form two separate loops.



(a) (b)



(c)

Figure 8.7: Snapshots from paths produced by our planner for the three examples using HRP3 in simulation. (a) Closed chain kinematics example. (b) Simultaneous constraints and goal sampling example. (c) Manipulating a passive chain example.

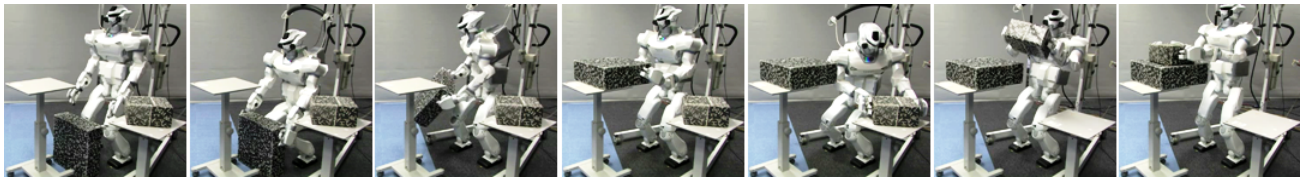


Figure 8.8: Snapshots from the execution of the box stacking task on the HRP3 robot.

We define three TSRs. The first TSR is assigned to the left leg of the robot and allows no deviation from the current left-foot location (i.e. $\mathbf{B}^w = \mathbf{0}_{6\times 2}$). The second and third TSRs are assigned to the left and right arms and are defined relative to the location of the box (i.e. the 0 frame of $\mathbf{T}_w^0$ is the frame of the box). The bounds are defined such that the hands will always be holding the sides of the box at the same locations ($\mathbf{B}^w = \mathbf{0}_{6\times 2}$). The geometry of the box is "attached" to the right hand.

We use the ConstrainConfig function of Algorithm 6 and include balance and collision constraints on line 14 so all paths produced by the planner are guaranteed to be collision-free and quasi-statically balanced. $\Delta q_{step} = 0.05$. In this problem we get the goal configuration of the robot from inverse kinematics on the box target; no goal sampling is performed in this example.

The result of this construction is the following: When ConstrainedExtend generates a new q_s, the box moves with the right hand and the frame of the box changes thus breaking the closed-chain constraint. This q_s is passed to ConstrainConfig, which projects q_s to meet the constraint (i.e. moving the left arm). Meanwhile, the TSR for the right hand ensures the box does not move during the projection. The same process happens simultaneously for the left leg of the robot as well since the kinematic chain is rooted at the right leg.

We implemented this example in simulation and on the physical HRP3 robot. Runtimes for 30 runs of this problem in simulation can be seen in Table 8.4. On the physical robot, the task was to stack two boxes in succession, snapshots from the execution of the plan can be seen in Figure 8.8. The experiments on the robot show that we can enforce stringent closed-chain constraints using the CBiRRT2 planner and TSRs.

8.8 Simultaneous Constraints and Goal Sampling

The task in this problem is to place a bottle held by the robot into a refrigerator (see Figure 8.7b). Usually, such a task is separated into two parts: first open the refrigerator and then place the bottle inside. However, with TSRs, there is no need for this separation because we can implicitly sample how much to open the refrigerator and where to put the bottle at the same time. The use of TSR Chains is important here, because it allows the right arm of the robot to rotate about the handle of the refrigerator, which gives the robot more freedom when opening the door. We assume that the grasp cages the door handle (as in [93]) so the end-effector can rotate about the handle without the door escaping.

There are four TSR Chains defined for this problem. The first is the TSR Chain (1 element) for the left leg, which is the same as in the previous example. This TSR Chain is marked for both sampling goals constraining pose. The second TSR Chain (2 element) is defined for the right arm and is described in Section 6.1. This chain is also marked for both sampling goals and constraining pose. The third TSR Chain (1 element) is defined for the left arm and constrains the robot to disallow tilting of the bottle during the robot's motion. This chain is used only as a pose constraint. Its bounds are

$$\mathbf{B}^w = \begin{bmatrix} -\infty & \infty \\ -\infty & \infty \\ -\infty & \infty \\ 0 & 0 \\ 0 & 0 \\ -\pi & \pi \end{bmatrix}. \tag{8.8}$$

The final TSR Chain (1 element) is also defined for the left arm and represents the allowable place-

ments of the bottle inside the refrigerator. Its $\mathbf{B}^w$ has freedom in x and y corresponding to the refrigerator width and length, and no freedom in any other dimension. This chain is only used for sampling goal configurations.

We use the ConstrainConfig function of Algorithm 6 and include balance and collision constraints on line 14 so all paths produced by the planner are guaranteed to be collision-free and quasi-statically balanced. $\Delta q_{step} = 0.05$. P_{sample} was set to 0.1 for this example.

The result of this construction is that the robot simultaneously samples a target bottle location and wrist position for its right arm when sampling goal configurations, thus it can perform the task in one motion instead of in sequence. Another important point is that we can be rather sloppy when defining TSRs for goal sampling. Observe that many samples from the right arm's TSR chain will leave the door closed or marginally open, thus placing the left arm into collision if it is reaching inside the refrigerator. However, this is not an issue for the planner because it can always sample more goal configurations and the collision constraint is included in the ConstrainConfig function. Theoretically, a TSR Chain defined for goal sampling need only be a super-set of the goal configurations that meet all constraints. However, as the probability of sampling a goal from this TSR chain which meets all constraints decreases, the planner will usually require more time to generate a feasible goal configuration, thus slowing down the algorithm.

Runtimes for 30 runs of this problem in simulation can be seen in Table 8.4.

8.9 Manipulating a Passive Chain

The task in this problem is for the robot to assist in placing a disabled person into bed (see Figure 8.7c). The robot's task is to move the person's right hand to a specified point near his body. The person's arm is assumed to be completely passive and the kinematics of the arm (as well as joint limits) are assumed to be known. The robot's grasp of the person's hand is assumed to be rigid.

There are two TSR Chains defined for this problem, both of which are used as pose constraints. The first is the TSR Chain(1 element) for the left leg, which is the same as the previous example. The second is a TSR Chain(6 element) defined for the person's arm. Every element of this chain corresponds to a physical DOF of the person's arm. Note that since the arm is not redundant, we do not need to perform any special IK to ensure that the configuration of the person matches what it would be in the real world.

We use the ConstrainConfig function of Algorithm 6 and include balance and collision constraints on line 14 so all paths produced by the planner are guaranteed to be collision-free and quasi-statically balanced. $\Delta q_{step} = 0.05$. In this problem we get the goal-configuration of the robot from inverse

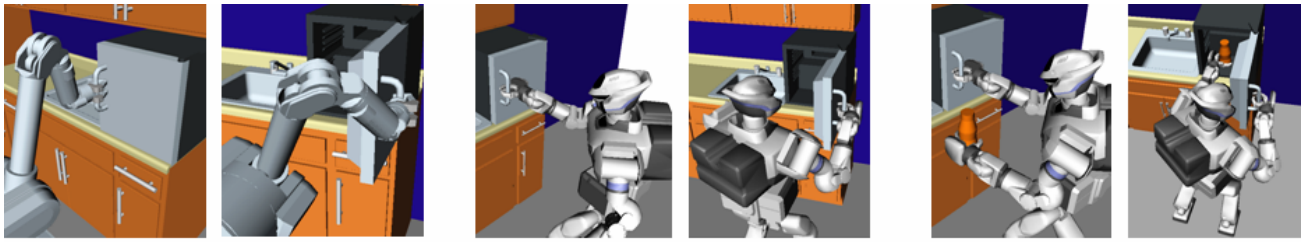


Figure 8.9: Start and goal configurations for the door-opening task used for the timing experiments. (a) WAM (7DOF) (b) HRP3 (9DOF) (c) HRP3 (28DOF)

kinematics on the target pose of the person's hand; no goal sampling is performed.

The result of this construction is that the person's arm will follow the robot's left hand. Since the configuration of the person's arm is included in q, there cannot be any significant discontinuities in the person's arm configuration (i.e. elbow-up to elbow-down) because such configurations are distant in the C-space. Runtimes for 30 runs of this example ins simulation can be seen in Table 8.4.

	Mean	Std. Dev
Closed Chain Kinematics	4.21s	2.00s
Simultaneous Constraints and Goal Sampling	1.54s	0.841s
Manipulating a Passive Chain	1.03s	0.696s

Table 8.4: Runtimes for Example Problems using HRP3

8.10 Profiling CBiRRT2

To evaluate the computational cost of the operations of CBiRRT2, we performed a runtime comparison on a door-opening task for the WAM and HRP3 (see Figure 8.9). We performed three experiments to gauge how the computation times of the main components of the algorithm (projection, collision-checking, and nearest neighbor queries) scaled with the DOF of the robot.

In the first experiment, we use a 7DOF WAM to open a refrigerator door 90 degrees. The constraint in this problem is formulated as a TSR Chain, similar to the one described in Section 8.8. The TSR Chain has two elements and allows the robot to rotate its hand about the door handle as well as allowing the door to rotate about its hinge.

The second experiment is identical to the first, except that we use the HRP3 robot instead of the WAM. We allow the robot to use its waist and right arm joints, for a total of 9DOF.

Finally, the third experiment is the same as the one described in Section 8.8 except that we have a pre-determined goal configuration where the door is opened to 90 degrees and the bottle is placed in the refrigerator, so there is no goal sampling. In this experiment we use the arms, legs, and waist of the HRP3, for a total of 28DOF.

We ran each experiment 50 times and the total computation times averaged over all runs are shown in Table 8.5. We also show the average time needed to do a single projection, collision check, and nearest neighbor query averaged over all runs[1].

	Projection (total)	Col. Check (total)	NN Query (total)	Projection (avg)	Col. Check (avg)	NN Query (avg)
WAM (7DOF)	0.1324s	0.3650s	7.2×10^{-6}s	0.0008s	0.0037s	2.1×10^{-6}s
HRP3 (9DOF)	0.1372s	0.5276s	1.2×10^{-5}s	0.0009s	0.0048s	2.5×10^{-6}s
HRP3 (28DOF)	0.8469s	2.049s	0.0017s	0.0018s	0.0052s	2.3×10^{-5}s

Table 8.5: Total and average computation times for the main components of CBiRRT2

The results in Table 8.5 show that collision-checking is the most time-consuming operation of the algorithm. However, as the number of DOF increases, the average projection time increases significantly. This is because the size of the matrices involved in the computation of the Jacobian pseudo-inverse, which is used to perform the projection, increases with the number of DOF. The projection also involves calling the forward kinematics function of the robot to obtain the robot's end-effector pose as well as computing the Jacobian of the end-effector, both of which become slower with increasing DOF.

[1] Note that these timing tests were run with a different version of OpenRAVE than the previous examples in this chapter.

Chapter 9

Probabilistic Completeness and Pose Constraints

We now discuss a proof of probabilistic completeness for our approach to planning with end-effector pose constraints. We emphasize that this proof is valid for a class of algorithms that plan with constraints on end-effector pose, not only CBiRRT2. We also note that, while we focus on the TSR representation in the rest of this thesis, this chapter makes no assumptions about the representation of pose constraints. Our only restriction is that the dimensionality of the manifold described by the constraint is fixed for a given problem.

Depending on the definition of an end-effector pose constraint, it can induce a variety of manifolds in the robot's configuration space. If these manifolds have non-zero volume in the C-space (see Figure 9.1d) it is straightforward to show that an RRT-based algorithm is probabilistically complete because rejection sampling in the C-space will eventually place samples inside of the manifold. However, if a pose constraint induces a lower-dimensional manifold, i.e. one that has zero volume in the C-space (see Figures 9.1e and 9.1f), rejection sampling in the C-space will not generate a sample on the constraint manifold.

RRT-based algorithms can overcome this problem by using the projection strategy: sampling coupled with a projection operator to move configuration space samples onto the constraint manifold. However, it is not clear whether the projection strategy produces adequate coverage of the constraint manifold to guarantee probabilistic completeness. The proof presented in this chapter guarantees probabilistic completeness for a class of RRT-based algorithms given an appropriate projection operator. This proof is valid for constraint manifolds of any fixed dimensionality.

The work discussed in this chapter was published in [94] and [88].

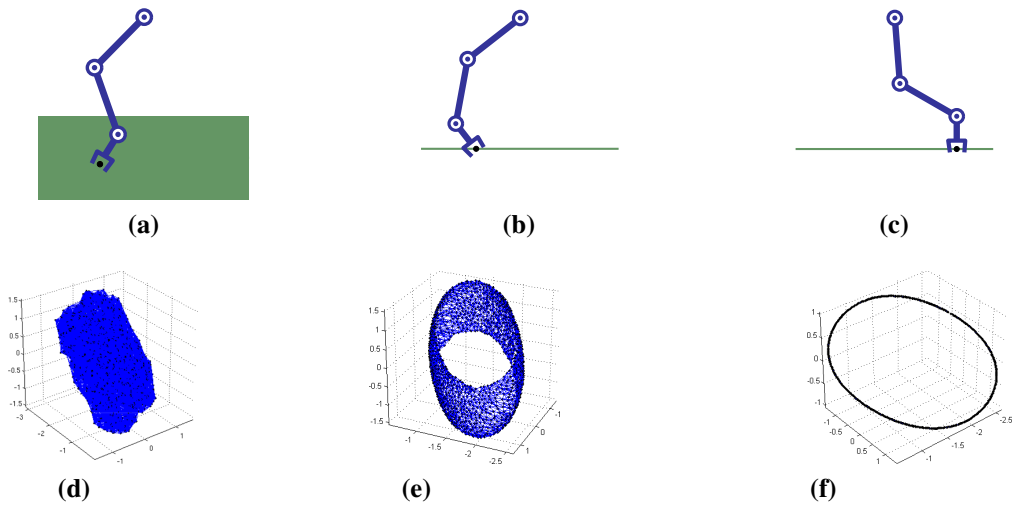


Figure 9.1: Three pose constraints and their corresponding C-space constraint manifolds for a 3-link manipulator. (a) The end-effector must be in the green rectangle with an orientation ± 0.7rad of downward. (b) The end-effector must be on the line with an orientation ± 0.7rad of downward. (c) The end-effector must be on the line pointing downward. (d-f) show graphs created from sampling on the manifolds corresponding to the constraints in (a-c), respectively. Black points are nodes and blue lines are edges. The manifold in (d) has non-zero volume in the C-space, the manifolds in (e) and (f) do not.

9.1 General Properties

Before we begin the proof for probabilistic completeness, we discuss some general properties of manifolds that will be useful in the proof. For brevity, we will only state parts of definitions that are relevant to our proof.

A topological *manifold* is a second countable Hausdorff space where every point has a neighborhood homeomorphic to an open Euclidean n-ball, where n is allowed to vary. Manifolds can be disjoint, with each piece of the manifold called a *connected component*. Manifolds that have a fixed n (i.e. n-dimensional manifolds) are called *pure* manifolds.

Lemma 1. *Consider manifolds $\mathcal{A}$ and $\mathcal{B}$. $(\mathcal{A} \cap \mathcal{B})$ is n-dimensional if the following conditions are true:*

1. *$\mathcal{A}$ and $\mathcal{B}$ are both n-dimensional*
2. *$\mathcal{A}$ and $\mathcal{B}$ are both submanifolds of the same n-dimensional manifold.*
3. *$(\mathcal{A} \cap \mathcal{B}) \neq \emptyset$*

Name	Symbol	Dimension
C-space	$\mathcal{Q}$	n
Configuration	q	0
Constraint Manifold	$\mathcal{M}$	$m = n - (r - d)$
Reachability	$\mathcal{R}$	r
Pose	$x(q)$	0
Task Constraint	$\mathcal{T}$	$d \leq r$

Table 9.1: Definitions used throughout the proof

Proof. Consider a point $p \in (\mathcal{A} \cap \mathcal{B})$. $\mathcal{A}$ and $\mathcal{B}$ both contain the same n-dimensional ball centered at p by the first and second conditions. $(\mathcal{A} \cap \mathcal{B})$ is the union of all such n-dimensional balls for all p. This union is n-dimensional. □

Let μ_n be a measure of volume in an n-dimensional space. If the volume of a manifold in an embedding space is zero, then the probability of generating a sample on the manifold by rejection sampling in the embedding space is zero. Conversely if the volume of a manifold in an embedding space is *not* zero, then the probability of generating a sample on the manifold by rejection sampling in the embedding space will go to 1 as the number of samples goes to infinity.

Definition 1. *A sampling method covers a manifold if it generates a set of samples such that any open n-dimensional ball contained in the manifold contains at least one sample.*

9.2 Definitions

Let the reachable manifold of end-effector poses of the given manipulator be $\mathcal{R} \subseteq SE(3)$. $\mathcal{R}$ is defined by the Forward Kinematics function of the robot

$$x : \mathcal{Q} \to \mathcal{R}. \tag{9.1}$$

where $\mathcal{Q}$ is the C-space. x is always surjective and can be one-to-one or many-to-one, depending on the manipulator.

We restrict our proof to manipulators whose $\mathcal{Q}$ and $\mathcal{R}$ are both pure manifolds. This restriction is necessary to meet the conditions of Lemma 1, yet allows many manipulators commonly used today,

such as serial-chain manipulators and humanoids. In this chapter, we also restrict our focus to manipulators with no non-holonomic constraints, though we hypothesize that our proof can extend to the non-holonomic case as well.

Let $\mathcal{Q}$ be n-dimensional, where n is the number of DOF of the manipulator. We will parameterize $SE(3)$ locally using three variables for translation and three for rotation, i.e. a pose will be a vector in $\mathbb{R}^6$. Let $\mathcal{R}$ be r-dimensional, where $r \leq 6$.

Let the manifold of end-effector poses allowable by the task be $\mathcal{T} \subseteq \mathcal{R}$. Let this manifold have dimensionality $d \leq r$. We will assume that d is fixed, though we will discuss the implications of allowing d to vary in Section 9.5. Let the manifold of configurations that place the robot's end-effector in $\mathcal{T}$ be $\mathcal{M} \subseteq \mathcal{Q}$. $\mathcal{M}$ is the union of all self-motion manifolds that map to a pose in $\mathcal{T}$, i.e.

$$\mathcal{M} = \bigcup_{t \in \mathcal{T}} \{q \in \mathcal{Q} \mid x(q) = t\}. \tag{9.2}$$

$\mathcal{M}$ has dimensionality $m = n - (r - d)$.

We will be using the (weighted-)Euclidian distance metric on $SE(3)$, which we will denote as dist. This distance metric has the property that each pose in $\mathcal{T}$ has at least an $(r - d)$-dimensional *Voronoi cell* in $\mathcal{R}$. A Voronoi cell is the set of points that are closer to a certain point than to any other given a distance metric [95].

Finally, our proof of manifold coverage hinges on the concept of *self-motion manifolds*, which were described by Burdick [96]. A self-motion manifold is the set of configurations in C-space which place the end-effector of the robot in a certain pose.

Table 9.1 summarizes the definitions and dimensionalities of manifolds used in the proof.

9.3 Proof of manifold coverage by projection sampling

When the manifold of allowable configurations in the C-space is of the same dimensionality as the C-space, samples on the manifold can be generated by *rejection sampling* in the C-space. Rejection sampling is simply the process of generating a sample in the C-space and then checking if it is an allowable configuration. However, when $\mathcal{M}$ is of a lower dimension than the C-space, the probability of generating a sample on the manifold through rejection sampling is 0.

The projection sampling method is an approach which can generate samples on manifolds of a lower

dimension. This method first produces a sample in the C-space and then projects that sample onto $\mathcal{M}$ using a *projection operator* $P : \mathcal{Q} \to \mathcal{M}$. In order to show that an algorithm using projection sampling is probabilistically complete, we must first show that this method covers $\mathcal{M}$.

This section will show that projection sampling does indeed cover $\mathcal{M}$ if the projection operator has the following properties:

1. $P(q) = q$ if and only if $x(q) \in \mathcal{T}$
2. If $x(q_1)$ is closer to $x(q_2) \in \mathcal{T}$ than to any other point in $\mathcal{T}$ and $\text{dist}(x(q_1), x(q_2)) < \epsilon$ for an infinitesimal $\epsilon > 0$, then $x(P(q_1)) = x(q_2)$.

The first property guarantees that a configuration that is already in $\mathcal{M}$ will project to itself. The second property ensures that any pose in $\mathcal{T}$ can be chosen for projection. We will describe the underlying mechanics of the projection operator in Section 9.3.2.

If $d = r$ then $\mu_n(\mathcal{M}) > 0$. By the first property of P, a sample placed in $\mathcal{M}$ will project to itself. Thus projection sampling will cover $\mathcal{M}$ by the same principle as rejection sampling.

The remainder of this section will focus on proving coverage when $d < r$, i.e. when $\mu_n(\mathcal{M}) = 0$. Consider an open m-dimensional ball $B_m(q) \subseteq \mathcal{M}$ for any $q \in \mathcal{M}$. Note that *open* in this context refers to the openness of the set with respect to $\mathcal{M}$. For notational simplicity, let B_m represent any $B_m(q)$ for any such q. We will show that projection sampling places a sample in any B_m as the number of iterations goes to infinity, thus covering of $\mathcal{M}$.

Consider an n-dimensional manifold $\mathcal{C}(B_m) = \{q : P(q) \in B_m,\ q \in \mathcal{Q}\}$. If such a $\mathcal{C}$ exists for any B_m, projection sampling will place a sample inside $\mathcal{C}$ with probability greater than 0 (because $\mathcal{C}$ is n-dimensional) and that sample will project into B_m. This will guarantee coverage of $\mathcal{M}$ as the number of iterations goes to infinity. But how do we guarantee that such a $\mathcal{C}$ exists for any B_m?

We will show that $\mathcal{C}$ can be defined as the intersection of two n-dimensional manifolds, $x^{-1}(\mathcal{N})$ and $\mathcal{UH}$, both of which must exist for any B_m (notation will be explained in subsequent subsections). The following subsections describe each of these manifolds and show that $(x^{-1}(\mathcal{N}) \cap \mathcal{UH})$ must be n-dimensional and all configurations in $(x^{-1}(\mathcal{N}) \cap \mathcal{UH})$ must project into B_m. Thus $(x^{-1}(\mathcal{N}) \cap \mathcal{UH})$ meets the requirements of $\mathcal{C}$, which completes the proof of coverage.

9.3.1 To Task Space and Back Again: Defining $x^{-1}(\mathcal{N})$

In this subsection we will define a manifold $x^{-1}(\mathcal{N})$, which projects into a set of self-motion manifolds $\mathcal{S} \subseteq \mathcal{M}$ that intersects B_m. We will do this by mapping B_m into task space, constructing a manifold

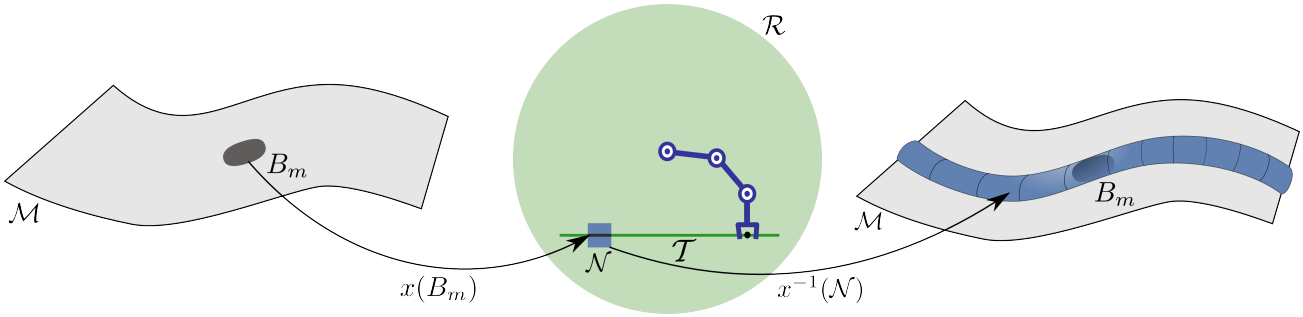


Figure 9.2: An example showing the process used to define $x^{-1}(\mathcal{N})$. $d = 1$, $r = 2$, and $n = 3$.

of poses that project into $x(B_m)$ and then mapping that manifold back into C-space (see Figure 9.2).

Mapping B_m into task space yields a d-dimensional manifold $x(B_m) \subseteq \mathcal{T}$. Let us define a task-space manifold $\mathcal{N}(x(B_m)) \subseteq \mathcal{R}$. $\mathcal{N}$ is constructed in two steps. First, take the union of a set of r-dimensional open balls centered at every $x(q) \in x(B_m)$. Second, remove all poses that are closer to a $x(q) \in (\mathcal{T} - x(B_m))$ than to any $x(q) \in x(B_m)$ and remove all equidistant poses. Formally, $\mathcal{N}$ is defined as

$$\begin{aligned} \mathcal{U} &= \bigcup_{x(q) \in x(B_m)} B_r(x(q)) \\ \mathcal{N} &= \mathcal{U} - \{p \in \mathcal{U} \mid \exists p_1 \in (\mathcal{T} - x(B_m)) \\ &\qquad \operatorname{dist}(p, p_1) \leq \inf_{p_2 \in x(B_m)} \operatorname{dist}(p, p_2)\}. \end{aligned} \tag{9.3}$$

where $B_r(x(q))$ is an open r-dimensional ball centered at $x(q)$ with radius ϵ. $\mathcal{N}$ has the following properties:

1. $\mathcal{N}$ is r-dimensional.
2. $\mathcal{N}$ contains $x(B_m)$.
3. $x(P(q)) \in x(B_m)$ for all $\{q \in \mathcal{Q} \mid x(q) \in \mathcal{N}\}$

The first property follows from the fact that the Voronoi cell of any $x(q) \in \mathcal{T}$ is $(r - d)$-dimensional and the fact that $x(B_m)$ is d-dimensional. The second property is clear by construction. The third property is guaranteed by the second property of the projection operator.

Define the C-space manifold $x^{-1}(\mathcal{N}) = \{q \in \mathcal{Q} \mid x(q) \in \mathcal{N}\}$. Since $\mathcal{N}$ is r-dimensional, x^{-1} will map it to an n-dimensional manifold of configurations in C-space (recall that x is surjective). $x^{-1}(\mathcal{N})$ has the following properties:

1. $x^{-1}(\mathcal{N})$ is n-dimensional.

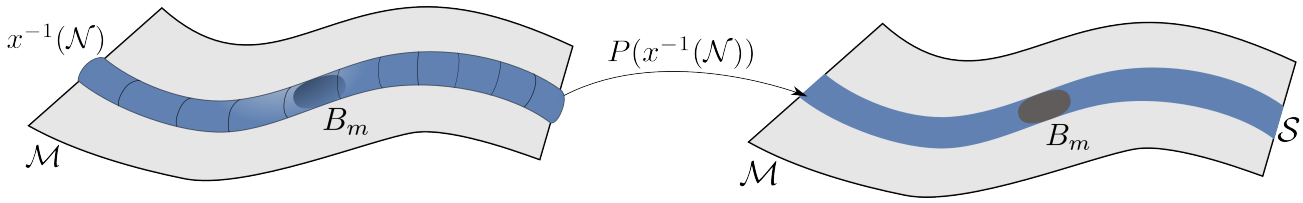


Figure 9.3: An example of $P(x^{-1}(\mathcal{N})) = \mathcal{S}$. $d = 1$, $r = 2$, and $n = 3$.

2. $x^{-1}(\mathcal{N})$ contains B_m

Now the question becomes, where do configurations in $x^{-1}(\mathcal{N})$ project to? It would be convenient if $P(x^{-1}(\mathcal{N})) = B_m$, however this is not always the case. The reason is that a single pose can correspond to more than one configuration in C-space, i.e. to a self-motion manifold. This occurs when the manipulator is redundant, for instance.

Let us now define where $x^{-1}(\mathcal{N})$ projects to. By the third property of $\mathcal{N}$, we know that all poses in $\mathcal{N}$ will project into $x(B_m)$. The analog of $x(B_m)$ in C-space will be a manifold $\mathcal{S} \subseteq \mathcal{M}$. $\mathcal{S}$ is defined as the manifold of configurations which map into $x(B_m)$; i.e. $\mathcal{S} = \{q \in \mathcal{M} \mid x(q) \in x(B_m)\}$. Thus we can state a third property of $x^{-1}(\mathcal{N})$ (see Figure 9.3):

3) $P(x^{-1}(\mathcal{N})) = \mathcal{S}$.

We will use $\mathcal{S}$ to show coverage of $\mathcal{M}$ in Section 9.3.3, but first we must show that $\mathcal{S}$ contains all self-motion manifolds that intersect B_m and only those self-motion manifolds.

Lemma 2. *$\mathcal{S}$ has the following properties:*

1. *If a self-motion manifold intersects B_m, it is a subset of $\mathcal{S}$.*
2. *If a self-motion manifold does not intersect B_m, it is not a subset of $\mathcal{S}$.*
3. *$\mathcal{S} \neq \emptyset$.*

Proof of 1st Property: Recall that, for any self-motion manifold $s \subseteq \mathcal{M}$, $x(s) = x(q)$ for any $q \in s$; i.e. all configurations on the self-motion manifold map to the same pose. If there exists $q \in (s \cap B_m)$, then $x(q) \in x(B_m)$, which entails $x(s) \in x(B_m)$. Thus by definition of $\mathcal{S}$, $s \subseteq \mathcal{S}$. □

Proof of 2nd Property: We will show this by contradiction. Suppose there is a self-motion manifold $s \subseteq S$ that does not intersect B_m. By definition of $\mathcal{S}$, $x(s)$ would have to be in $x(B_m)$. However, if $x(s) \in x(B_m)$, there is some configuration $q \in B_m$ such that $x(q) = x(s)$. This entails that $q \in s$ and thus $q \in (s \cap B_m)$, but this contradicts the assumption that s does not intersect B_m. □

Proof of 3rd Property: $\mathcal{M}$ is composed of all self-motion manifolds for all poses in $\mathcal{T}$. Since $\mu_m(B_m) > 0$, B_m must intersect at least one self-motion manifold and by the first property of $\mathcal{S}$, $\mathcal{S} \neq \emptyset$. □

9.3.2 Projection onto a ball on a self-motion manifold

We have shown that $P(x^{-1}(\mathcal{N})) = \mathcal{S}$ and described some properties of $\mathcal{S}$, however we have not stated where on $\mathcal{S}$ a projected configuration will go. It is possible that a $q \in x^{-1}(\mathcal{N})$ will project to a configuration outside of B_m because $\mathcal{S}$ may contain configurations outside of B_m.

In order to show that the probability of placing a sample inside B_m using projection sampling is greater than 0, we need to show that there exists an n-dimensional manifold around a ball on a self-motion manifold that projects into that ball. The purpose of this subsection is to prove this property of self-motion manifolds. The remainder of this subsection will consider a self-motion manifold in isolation in order to show this property.

Let us now look closer at the mechanism of projection used by P. In this chapter, we focus on projection operators based on Jacobian pseudo-inverse or Jacobian transpose. These kinds of projection operators step towards a pose target and this process can be written as a differential equation:

$$\frac{dq}{dt} = f(q(t)), t \in [0, \infty). \tag{9.4}$$

f satisfies the Lipschitz condition

$$\| f(q_1) - f(q_2) \| \leq L \| q_1 - q_2 \| \tag{9.5}$$

where $\| \cdot \|$ denotes any *p*-norm and L is the Lipschitz constant [97].

An *equilibrium point* $\bar{q}$ of Eqn.9.4 satisfies $f(\bar{q}) = 0$.

Definition 2. *The equilibrium point* $\bar{q}$ *is*

- *stable if, for each* $\epsilon > 0$*, there is a* $\delta = \delta(\epsilon) > 0$ *such that*

$$\| q(0) - \bar{q} \| < \delta \Rightarrow \| q(t) - \bar{q} \| < \epsilon, \forall t \geq 0$$

- *unstable if it is not stable*
- *asymptotically stable if it is stable and* δ *can be chosen such that*

$$\| q(0) - \bar{q} \| < \delta \Rightarrow \lim_{t \to \infty} q(t) = \bar{q}.$$

Asymptotic stability guarantees that any point inside a δ neighborhood of $\bar{q}$ will converge to $\bar{q}$. We will use Lyapunov's stability theorem to define a domain around $\bar{q}$ where asymptotic stability is valid.

Theorem 3. *Let* $\bar{q}$ *be an equilibrium point for Eqn.9.4 and* D *be a domain containing* $\bar{q}$*. Let* $V : D \to \mathbb{R}$ *be a continuously differentiable function such that*

$$V(\bar{q}) = 0 \tag{9.6}$$

$$V(q) > 0 \text{ for } q \in (D - \{\bar{q}\}) \tag{9.7}$$

$$\frac{dV(q)}{dt} < 0 \text{ for } q \in (D - \{\bar{q}\}) \tag{9.8}$$

then $\bar{q}$ *is asymptotically stable.*

In this chapter, we focus on systems of the form

$$\frac{dq}{dt} = A(q)(x_o - x(q)) = Ae \tag{9.9}$$

where x_o is a target pose and $A(q) : \mathbb{R}^r \to \mathbb{R}^n$ is a configuration-dependent linear map. The error e decreases linearly as the pose approaches the target.

Equilibrium points of Eqn.9.9 occur at

$$A(q)(x_o - x(q)) = Ae = 0. \tag{9.10}$$

If rank$(A) = r$, these occur at $x(q) = x_o$. We define the set of equilibrium points as the $(n - r)$-dimensional self-motion manifold

$$\bar{\mathcal{Q}} = \{q \in \mathcal{Q} \mid x(q) = x_o\}. \tag{9.11}$$

Consider an equilibrium point $\bar{q} \in \bar{\mathcal{Q}}$. We define a candidate Lyapunov function of the form

$$V(q) = \frac{1}{2} e'(Ae). \tag{9.12}$$

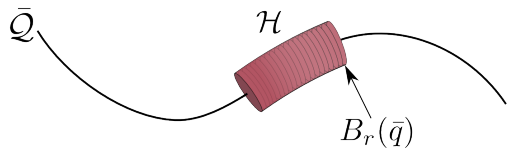


Figure 9.4: A self-motion manifold in C-space. $r = 2$ and $n = 3$.

We know from Eqn.9.10 that $V(q) = 0$ at $q = \bar{q}$, thus satisfying the condition of Eqn.9.6.

However, if we define the domain D to be $\mathcal{Q}$, the requirements of Eqn.9.7 and Eqn.9.8 are *not* met: V is not uniquely 0 at $\bar{q}$. Rather, there is an entire set of equilibrium points $\bar{\mathcal{Q}} \subseteq \mathcal{Q}$ where $V = 0$. We address this issue by restricting V to a domain where $\bar{q}$ is the exclusive minimum.

For illustration, we first consider the case where A has no functional dependence on q. We restrict the domain D of V to the r-dimensional hyperplane

$$D = \{q \in \mathcal{Q} \mid q = \bar{q} + Ae,\ e \in \mathbb{R}^r\}. \tag{9.13}$$

Revisiting Eqn.9.7 and Eqn.9.8, we get $V(q) = 0$ and $\frac{dV(q)}{dt} = 0$ at $Ae = 0$ which occurs uniquely at $q = \bar{q}$ when V is restricted to D, thereby completing all the requirements for Lyapunov's theorem.

For the case where A does have a functional dependence on q, the Taylor series approximation of $A(q)$ in an n-dimensional ball $B_n(\bar{q})$ around $\bar{q}$ gives

$$A(q) \approx A(\bar{q}) + \frac{dA}{dq}(q - \bar{q}) + \cdots . \tag{9.14}$$

We can then restrict the domain of V to the intersection of the r-dimensional hyperplane with $B_n(\bar{q})$

$$D = \{q \in \mathcal{Q} \mid q = \bar{q} + A(\bar{q})e,\ q \in B_n(\bar{q})\}. \tag{9.15}$$

The intersection restricts the domain D to an r-dimensional ball $B_r(\bar{q})$ around $\bar{q}$ (see Figure 9.4). Once restricted, the situation is identical to the previous case.

To summarize, we have shown that, for any point $\bar{q} \in \bar{\mathcal{Q}}$, there exists an r-dimensional ball $B_r(\bar{q})$ within which the conditions for Lyapunov's stability theorem hold.

The evolution of Eqn.9.4 results in the projection $P(q_1) = \bar{q}$ for all $\bar{q} \in \bar{\mathcal{Q}}$, for all $q_1 \in B_r(\bar{q})$ if $q(0) = q_1$. Note that this is only valid for a self-motion manifold in isolation.

Let us now consider an open $(n - r)$-dimensional ball $B_{\bar{\mathcal{Q}}}(\bar{q}) \subseteq \bar{\mathcal{Q}}$. Define $\mathcal{H}(B_{\bar{\mathcal{Q}}}(\bar{q}))$ as the manifold $B_r(\bar{q}) \times B_{\bar{\mathcal{Q}}}(\bar{q})$ embedded in $\mathcal{Q}$ (see Figure 9.4).

$\mathcal{H}$ has the following properties:

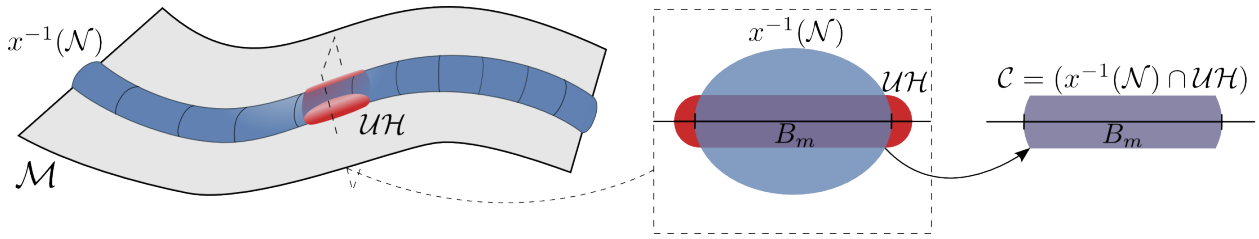


Figure 9.5: An example of $\mathcal{C} = (x^{-1}(\mathcal{N}) \cap \mathcal{UH})$. $d = 1$, $r = 2$, and $n = 3$.

1. $\mathcal{H}$ is n-dimensional.
2. $\mathcal{H}$ contains $B_{\bar{\mathcal{Q}}}(\bar{q})$.
3. $P(q) \in B_{\bar{\mathcal{Q}}}(\bar{q})$ for all $q \in \mathcal{H}$ (conditional).

It is important to note that we have only described $\mathcal{H}$ for a self-motion manifold in isolation. The third property of $\mathcal{H}$ is only valid when $x(q)$ is closer to $x(\bar{\mathcal{Q}})$ than to any $x(k)$, for a self-motion manifold $k \subseteq (\mathcal{M} - \bar{\mathcal{Q}})$. I.e. the third property holds only when $\bar{\mathcal{Q}}$ is "chosen" by the projection operator.

9.3.3 Putting it all together

We will now bring the concepts developed thus far together to show coverage of $\mathcal{M}$ by projection sampling. Recall that, to show coverage of $\mathcal{M}$, we must show that projection sampling places a sample inside any $B_m \subseteq \mathcal{M}$.

Section 9.3.1 showed that a configuration inside $x^{-1}(\mathcal{N})$ will project into $\mathcal{S}$, which consists of all self-motion manifolds that intersect B_m. Let us construct an n-dimensional manifold $\mathcal{UH}(B_m)$ by taking the union of the n-dimensional $\mathcal{H}$ manifolds around every $\bar{\mathcal{Q}} \cap B_m$ for all $\bar{\mathcal{Q}} \subseteq \mathcal{S}$

$$\mathcal{UH}(B_m) = \bigcup_{\bar{\mathcal{Q}} \subseteq \mathcal{S}} \mathcal{H}(\bar{\mathcal{Q}} \cap B_m). \tag{9.16}$$

$\mathcal{UH}$ inherits the properties of $\mathcal{H}$, as well as the condition on the third property: A configuration $q \in \mathcal{UH}$ will project to a configuration inside B_m by the third property of $\mathcal{H}$ if, for some $\bar{\mathcal{Q}} \subseteq \mathcal{S}$, $x(q)$ is closer to $x(\bar{\mathcal{Q}})$ than to any $x(k)$, for a self-motion manifold $k \subseteq (\mathcal{M} - \mathcal{S})$.

Lemma 4. *Let $\mathcal{C} = (x^{-1}(\mathcal{N}) \cap \mathcal{UH})$. $\mathcal{C}$ has the following properties:*

1. *C is n-dimensional.*

2. $P(q) \in B_m$ *for all* $q \in \mathcal{C}$.

Proof of 1st Property: $B_m \subseteq x^{-1}(\mathcal{N})$ by the second property of $x^{-1}(\mathcal{N})$. $B_m \subseteq \mathcal{UH}$ by the second property of $\mathcal{H}$. Both $x^{-1}(\mathcal{N})$ and $\mathcal{UH}$ are n-dimensional. By Lemma 1, $x^{-1}(\mathcal{N}) \cap \mathcal{UH}$ is n-dimensional (see Figure 9.5). □

Proof of 2nd Property: A $q \in (x^{-1}(\mathcal{N}) \cap \mathcal{UH})$ will project into $\mathcal{S}$ because $q \in x^{-1}(\mathcal{N})$. Thus there exists a self-motion manifold $\bar{\mathcal{Q}} \subseteq \mathcal{S}$ such that $x(q)$ is closer to $x(\bar{\mathcal{Q}})$ than to any $x(k)$, for a self-motion manifold $k \subseteq (\mathcal{M} - \mathcal{S})$. This fact meets the condition required by the third property of $\mathcal{H}$. Because $q \in \mathcal{UH}$, and the third property of $\mathcal{H}$ is valid, $P(q)$ must be in B_m. □

Theorem 5. *Projection sampling places a sample inside any* B_m *as the number of samples goes to infinity.*

Proof. By the first property of $\mathcal{C}$, $\mu_n(\mathcal{C}) > 0$. Thus the probability of sampling a $q \in \mathcal{C}$ is greater than 0 and, as the number of samples goes to infinity, the probability of sampling a $q \in \mathcal{C}$ goes to 1. $P(q) \in B_m$ by the second property of $\mathcal{C}$. Thus we have shown that projection sampling places a sample inside any B_m as the number of samples goes to infinity, which entails that projection sampling covers $\mathcal{M}$. □

9.4 Probabilistic completeness of RRT-Based algorithms that use projection sampling

We will use the fact that projection sampling covers $\mathcal{M}$ to prove that RRT-based methods that plan paths on $\mathcal{M}$ are probabilistically complete. We focus on a class of RRT-based algorithms that plan with end-effector pose constraints, for example CBiRRT [90], TCRRT [10], and the algorithm of Dalibard et al. [18]. These algorithms grow trees on $\mathcal{M}$ by sampling near an existing node on $\mathcal{M}$ and then projecting that sample to $\mathcal{M}$ (see Figure 9.6). We will show that an RRT-based algorithm with the following properties is probabilistically complete.

1. Given a node of the existing tree, the probability of sampling in an n-dimensional ball centered at that node is greater than 0 and the sampling covers this ball.

2. The algorithm uses a projection operator with the properties of P to project samples to $\mathcal{M}$.

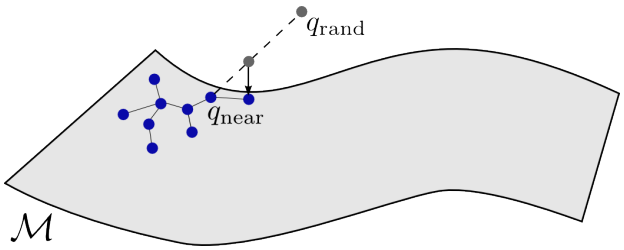


Figure 9.6: An example of an RRT-based algorithm using projection sampling. First a random sample q_{rand} is generated in the C-space and the closest node of the tree q_{near} is calculated. The algorithm then computes an intermediate configuration by stepping from q_{near} toward q_{rand}. If the distance between q_{rand} and q_{near} is smaller than the step size, q_{rand} is used as the intermediate configuration. The algorithm then projects this configuration to the manifold and adds the resulting node to the tree.

Define $\mathcal{M}_c \subseteq \mathcal{M}$ as the connected component of $\mathcal{M}$ which contains the initial configuration. In order to be probabilistically complete, we must show that the algorithm will place a node in any m-dimensional ball in $\mathcal{M}_c$. Let the set of RRT nodes already generated by the algorithm (including the start and goal) be $N \subseteq \mathcal{M}_c$. A node of the tree is associated with a configuration q_n; nodes will be referenced by their configuration. Let $B_n(q)$ for a configuration $q \in \mathcal{M}_c$ be an n-dimensional open ball centered at q.

Lemma 6. *The algorithm covers $B_n(q_n) \cap \mathcal{M}_c$ as the number of samples goes to infinity.*

Proof. Consider a $B'_m \subseteq (B_n(q_n) \cap \mathcal{M}_c)$. From Theorem 5, we know that any B'_m has an associated n-dimensional manifold $\mathcal{C}$ such that all samples in $\mathcal{C}$ project into B'_m. Both $\mathcal{C}$ and $B_n(q_n)$ are n-dimensional and both contain B'_m. Thus, by Lemma 1, $\mathcal{C} \cap B_n(q_n)$ is n-dimensional. Because the algorithm samples in $B_n(q_n)$ with probability greater than 0 and the samples cover $B_n(q_n)$, it will generate a sample inside $\mathcal{C} \cap B_n(q_n)$ as the number of samples goes to infinity with probability 1. This sample will then project into B'_m by the second property of $\mathcal{C}$. Thus the algorithm projects a sample into any $B'_m \subseteq (B_n(q_n) \cap \mathcal{M}_c)$ as the number of samples goes to infinity. □

We have shown that the algorithm covers $\mathcal{M}_c$ locally, i.e. around existing nodes. We will now show that the nodes of the tree(s) cover $\mathcal{M}_c$ as the number of samples goes to infinity. We will be using the series-of-balls argument in the subsequent proof. For a more detailed explanation of this argument see [98] and [99].

Theorem 7. *The algorithm will place a node in any $B_m \subseteq \mathcal{M}_c$ as the number of samples goes to infinity.*

Proof. For any $B_m \subseteq \mathcal{M}_c$, we can construct a series of open m-dimensional balls starting at some $q_n \in N$ such that subsequent balls overlap and the final ball overlaps with B_m. The overlapping regions between the balls are m-dimensional. By Lemma 6 a sample will be placed in any B'_m in the

overlapping region if we sample in the B_n centered at the center of the previous ball. This sample will then be added to N as a node of the RRT. The algorithm is guaranteed to place a node within each overlap by induction, thus placing a node in any B_m as the number of samples goes to infinity. □

Theorem 8. *The algorithm is probabilistically complete.*

Proof. By Theorem 7, as the number of samples goes to infinity any $B_m \subseteq \mathcal{M}_c$ will be sampled and a corresponding node will be added to N. As the radius of B_m goes to 0, N will approach $\mathcal{M}_c$ in the limit. It follows that a path from start to goal will be found if one exists. □

9.5 Discussion

We now discuss the implications of our proof for projection operators and mixed-dimensional constraint manifolds.

9.5.1 Projection Operators

Section 9.3 describes a projection operator that guarantees probabilistic completeness for RRT-based algorithms. The regularized Jacobian pseudo-inverse or Jacobian transpose iterative inverse kinematics methods can be used to perform the projection because they possess the requisite properties. Unfortunately, there are some common iterative methods which will *not* yield probabilistic completeness.

The null-space projection method [19] operates by using the null-space of the primary task (placing the end-effector in some pose) to satisfy secondary tasks such as collision-avoidance or balancing. For this method, $\frac{dq}{dt} = f(q(t))$ is

$$\frac{dq}{dt} = \mathbf{J}^{\#}\dot{x} + (\mathbf{I} - \mathbf{J}^{\#}\mathbf{J})\dot{q}_{\text{null}}. \tag{9.17}$$

where $\mathbf{J}^{\#}$ is the generalized pseudo-inverse of the Jacobian, $\dot{x}$ is the error in pose, and $\dot{q}_{\text{null}}$ is the error in meeting a secondary objective. This type of projection attains optimal configurations by sliding along a self-motion manifold when $\dot{x} = 0$. The secondary task induces local minima on the self-motion manifold which attract configurations from the rest of the self-motion manifold. Thus, if a ball on the self-motion manifold does not contain a local minimum of the secondary task, configurations projecting to that ball may escape by sliding along the manifold. It follows this projection operator will not cover $\mathcal{M}$. Though using this projection operator in an RRT-based algorithm may yield an effective planner, it will not be probabilistically complete.

9.5.2 Mixed-dimensional Constraint Manifolds

The proof of coverage and probabilistic completeness assumed that the task constraint $\mathcal{T}$ had a fixed dimensionality d. If we allow d to vary, then m (the dimensionality of $\mathcal{M}$) will vary as well. Since our proof of coverage used only local properties of $\mathcal{M}$, we can extend this proof to the case of varying m simply by applying the proof to each m-dimensional component of $\mathcal{M}$ for every m. However, there is the case when a ball around a point on $\mathcal{M}$ contains components of varying dimension. In this case, the ball can be split according to the dimensionality of its components and $\mathcal{C}$ can be shown to exist for one of these components, thus guaranteeing a sample will be placed in this ball.

Though we can show that projection sampling covers $\mathcal{M}$, the proof of probabilistic completeness for RRT-based algorithms only holds when $\mathcal{M}$ is pure. The reason is that mixed-dimensional manifolds can be constructed such that all paths between two configurations must go through a narrow passage, which is of lower dimension than any component of the manifold. For instance, suppose $\mathcal{M}$ were composed of two lines that intersect at some configuration q_p. To get from one line to the other, the algorithm would need to find a path which contained q_p. Yet there is 0 probability of generating q_p exactly, thus a path may never be found, though one exists. This difficulty is not caused by projection sampling and is not limited only to pose constraints. Rather this difficulty arises for all RRT-based planners when they must find a path through a lower-dimensional narrow passage. Coverage of $\mathcal{M}$ does not entail probabilistic completeness in this case. In order to guarantee probabilistic completeness, the algorithm must be able to generate samples in these lower-dimensional narrow passages, as in [100].

Chapter 10

Addressing Pose Uncertainty with TSRs

In an effort to broaden the applicability of our framework to larger classes of real-world problems, we have been studying how to compute safe plans in the presence of uncertainty. A common assumption when planning for robotic manipulation tasks is that the robot has perfect knowledge of the geometry and pose of objects in the environment. For a robot operating in a home environment it may be reasonable to have geometric models of the objects the robot manipulates frequently and/or the robot's work area. However these objects and the robot often move around the environment, introducing uncertainty into the pose of the objects relative to the robot. Laser-scanners, cameras, and sonar sensors can all be used to help resolve the poses of objects in the environment, but these sensors are never perfect and usually localize the objects to be within some hypothetical set of pose estimates. Planning with only one estimate from this set can violate task specifications.

Suppose that a robot arm is to grasp an object by placing its end-effector at a particular pose relative to the object and closing the fingers. If there is any uncertainty in the pose of the object, generally no guarantee can be made that the end-effector will reach a specific point relative to the true pose of the object. Depending on the task, this lack of precision may range from being the source of minor disturbances to being the cause of critical failure.

To address this problem, we can exploit task affordances (encoded as TSRs) to compensate for uncertainty. TSRs allow planning for manipulation tasks in the presence of pose uncertainty by ensuring that the given task requirements are satisfied for all hypotheses of an object's pose. TSRs also provide a way to quickly reject tasks which cannot be guaranteed to be accomplished given the current pose uncertainty estimates. In this chapter, we show how to modify the TSRs of a given task to account for pose uncertainty and guarantee that samples drawn from the modified TSRs will meet task specifications. The methods presented in this chapter apply only to the TSR representation; we have not yet generalized them to account for TSR Chains because we do not yet have a method for intersecting the

The work discussed in this chapter was published in [101] and [88].

implicit sets of poses defined by TSR Chains.

10.1 Intersecting TSRs

Let the set of pose hypotheses for a given object be a set of transformation matrices $\mathcal{H}$. Also, let the set of TSRs defined for this object be $\mathcal{T}$. The process for generating a new set of TSRs $\mathcal{T}_{new}$ that takes into account $\mathcal{H}$ is shown in Algorithm 7. The goal of this process is to find the intersection of copies of each TSR corresponding to each pose hypothesis. Any point sampled from within the volume of intersection is guaranteed to meet the task specification despite pose uncertainty.

This algorithm first splits every TSR $t \in \mathcal{T}$ to take into account the rotation uncertainty in $\mathcal{H}$, generating a set $\mathcal{T}_{split}$ for each t. See Figure 10.1 for an illustration of this process. It then places a duplicate of each $t_s \in \mathcal{T}_{split}$ at every location defined by the transforms in $\mathcal{H}$. Next, it computes the volume of intersection of all duplicates for every t_s (see Figure 10.2). Recall that TSR bounds define a cuboid in pose space. The volume of intersection between multiple cuboids is computed by first converting all faces of all cuboids into linear constraints via the FacesToLinInequalities function and then converting those linear constraints into vertices P of a 6D polytope via the GetVerticesInequalities function. Since TSRs are convex we know that the polytope of intersection must be convex as well. If the uncertainty is too great (i.e. there is no 6D point where all duplicates intersect), P will be empty. If P is not empty, we place an axis-aligned bounding box around P, set this as the new bounds of t_s, and add t_s to $\mathcal{T}_{new}$. Note that it is irrelevant which element of $\mathcal{H}$ is used as $\mathbf{T}^0_{h_0}$ because the results will always be the same in the world frame.

10.2 Direct Sampling from the Volume of Intersection

In order to guarantee that a directly sampled 6D point meets the uncertainty specification of the problem, samples drawn from t_s must lie inside the polytope defined by P. Ideally, we would like to generate uniformly random samples from within P directly. Indeed, this is always possible because the polytope defined by P is convex. Because the polytope is convex, it is divisible into simplices using Delaunay Triangulation. To generate a uniformly random sample from a collection of simplices, we first select a simplex proportional to its area and then sample within that simplex by generating a random linear combination of its vertices [102]. For simple polytopes, this method is quite efficient, however as the polytope defined by P grows more complex, the Delaunay Triangulation becomes more costly, thus this method usually does not scale well with the number of hypotheses in $\mathcal{H}$.

Rejection sampling can also be used to sample from the polytope defined by P. When using rejection sampling, we sample a point x uniformly at random from the bounding-box of P until we find an x

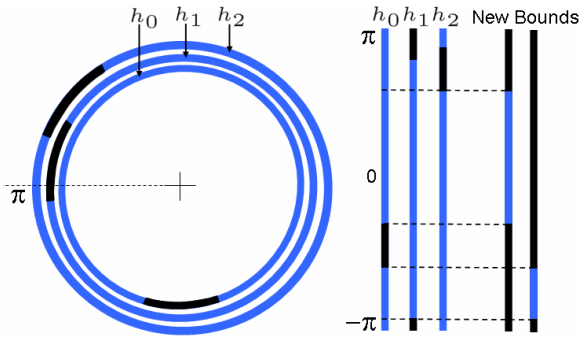


Figure 10.1: Process for splitting TSRs to take into account rotation uncertainty. Only one dimension of rotation is shown here. The three concentric circles correspond to a *single* TSR's bound in Roll that has been rotated by transforms $\mathbf{T}^0_{h_0}$, $\mathbf{T}^0_{h_1}$, and $\mathbf{T}^0_{h_2}$. Blue regions correspond to allowable rotations and black ones to unallowable rotations. The circles are cut at $\pi = -\pi$ and overlaid on the right. The strips where all rotations are valid (there are no black regions) are extracted as new separate bounds for this dimension. This process is identical for Roll, Pitch, and Yaw. The cartesian product of the new bounds for Roll, Pitch, and Yaw along with the original x, y, and z bounds produces a new set of TSRs $\mathcal{T}_{split}$.

Algorithm 7: ApplyUncertainty($\mathcal{T}$,$\mathcal{H}$)

$\mathbf{T}^0_{h_0} \leftarrow$ Any element of $\mathcal{H}$;
$\mathcal{T}_{new} \leftarrow \emptyset$;
for $t \in \mathcal{T}$ **do**
  $\mathcal{T}_{split} \leftarrow$ SplitRotations(t, $\mathbf{T}^0_{h_0}$, $\mathcal{H}$);
  for $t_s \in \mathcal{T}_{split}$ **do**
    $A \leftarrow \emptyset$; $b \leftarrow \emptyset$;
    for $\boldsymbol{T}^0_h \in \mathcal{H}$ **do**
      $V \leftarrow$ GetVertices(t_s);
      $V_{xyz} \leftarrow (\mathbf{T}^0_{h_0})^{-1}\mathbf{T}^0_h V_{xyz}$;
      $F \leftarrow$ GetFaces(V);
      $\{A_{temp}, b_{temp}\} \leftarrow$ FacesToLinInequalities(F);
      $A \leftarrow A \cup A_{temp}$;
      $b \leftarrow b \cup b_{temp}$;
    end
    $P \leftarrow$ GetVerticesFromInequalities(A, b);
    if $P = \emptyset$ **then**
      $t_s.\mathbf{T}^0_w \leftarrow h_0$;
      $t_s.\mathbf{B}^w \leftarrow$ BoundingBox(P);
      $t_s.\mathbf{T}^w_e \leftarrow t.\mathbf{T}^w_e$;
      $t_s.\mathbf{LI} \leftarrow \{A, b\}$;
      $\mathcal{T}_{new} \leftarrow \mathcal{T}_{new} \cup t_s$;
    end
  end
end
return $\mathcal{T}_{new}$;

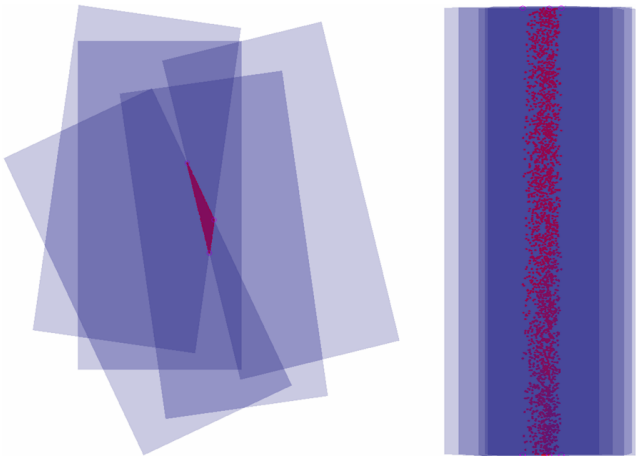


Figure 10.2: Intersection of five instances of a TSR. *Left*: x-y view. *Right*: y-z view. The red points are sampled within the polytope of intersection using rejection sampling.

which satisfies $b - Ax \geq 0$, where the matrix A and the vector b describe the hyperplanes and offsets, respectively, that define the faces of P. This method is quite fast in practice and does not require triangulating the polytope defined by P, thus it is more suitable for use in an online planning scenario.

In order to accommodate rejection sampling with TSRs, we add another element to our TSR definition (Section 5.1) for tasks with pose uncertainty:

- **LI**: Linear inequalities of the form $b - Ax \geq 0$

If the $\mathcal{T}_{new}$ returned by ApplyUncertainty($\mathcal{T}$, $\mathcal{H}$) is empty, then we know that it is impossible to accomplish this task with the uncertainty in $\mathcal{H}$. We can thus reject this task without calling the planner, a key advantage of this approach.

10.3 Results

To evaluate our approach, we conducted experiments to measure the time taken to apply uncertainty to TSRs (Algorithm 7) and show examples of using TSR intersection along with CBiRRT2 in a cluttered kitchen environment. The TSRs in these experiments were defined for the juice bottle (orange bottle) and rice box (red box) shown in Figure 10.3.

The juice bottle has a TSR that allows the hand to rotate about the z-axis of the bottle and also several centimeters of freedom in translation. The $\mathbf{T}_e^w$ is set such that the fingers envelope the bottle. The bounds of this TSR are

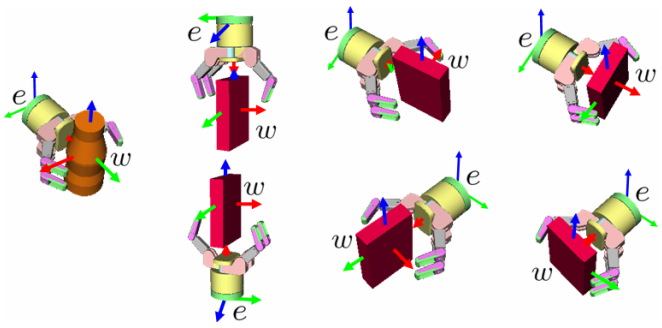


Figure 10.3: Depiction of the w and e frames which are used to get the $\mathbf{T}_w^0$ and $\mathbf{T}_e^w$ transforms for the juice bottle and rice box when reaching to grasp. We also include copies of these TSRs with the hand rotation by πrad around the x axis (*red*).

$$\mathbf{B}^w = \begin{bmatrix} -0.02 & 0.02 \\ -0.02 & 0.02 \\ -0.02 & 0.02 \\ 0 & 0 \\ 0 & 0 \\ -\pi & \pi \end{bmatrix}. \tag{10.1}$$

We also add another TSR with identical bounds, but with $\mathbf{T}_e^w$ defined to flip the hand.

For the rice box, we start with six TSRs, one for each face. The $\mathbf{B}^w$ is set according to the dimension of each face and $\mathbf{T}_e^w$ for each TSR points the hand toward the corresponding face. We also allow ± 0.4rad rotation freedom about the z-axis of the rice box in each $\mathbf{B}^w$. As with the juice bottle, we add 6 more TSRs that are identical to the previous 6 except that $\mathbf{T}_e^w$ flips the hand.

10.4 Applying Uncertainty to TSRs

We conducted several experiments to gauge the performance of our algorithm for applying uncertainty to TSRs with various numbers of pose hypotheses in the scenes shown in Figure 10.4. The pose hypotheses were sampled uniformly from ± 1.5cm in x and y and ± 0.2rad in Yaw. There was no error in the other dimensions because all the objects we detected to lie on a flat surface. In practice, we would project pose hypotheses generated by a sensor onto the flat surface to eliminate error in the other dimensions.

The goal was to see how the necessary runtime scaled with respect to the number of pose hypotheses. The results are shown in Table 10.1. We found that the runtime scaled approximately linearly with the

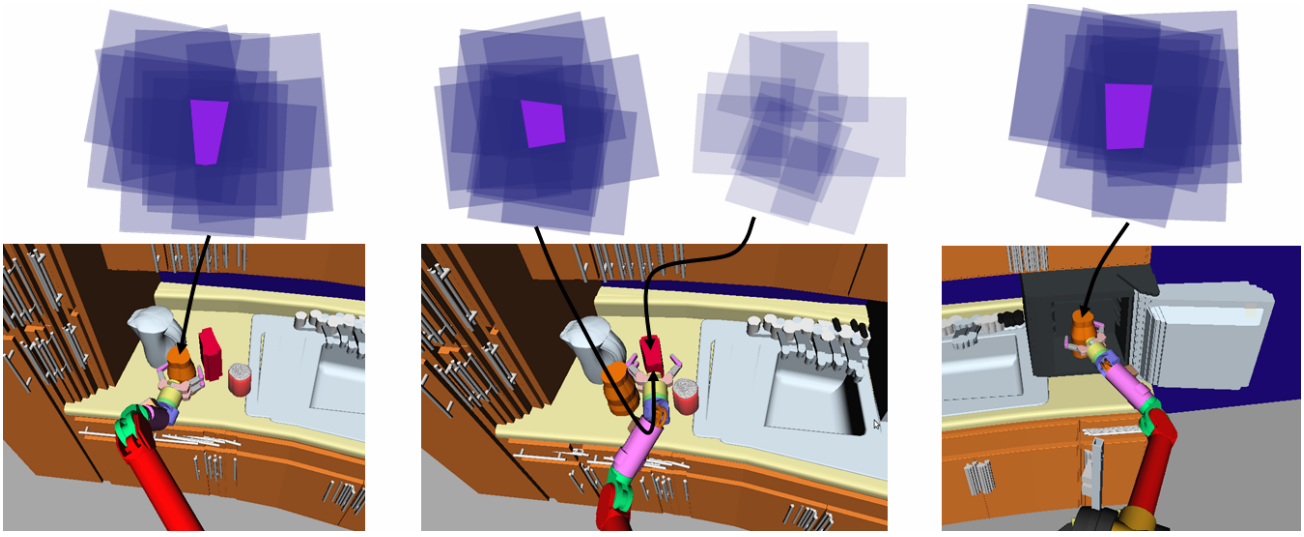


Figure 10.4: Typical results of tasks 1, 2, and 3, from left to right. The intersecting boxes above show several of the intersecting TSRs for these tasks. In the task shown in the center the TSR for grasping the box from the top is eliminated by uncertainty (there is no point where all the boxes intersect) while the one for grasping it from the side is not.

number of hypotheses. The runtime for the rice box was greater because it has 12 associated TSRs whereas the juice bottle only has 2.

No. Hypotheses	1	15	30	45	60
Juice Bottle	0	0.17	0.29	0.44	0.58
Rice Box	0	0.85	1.60	2.40	3.10

Table 10.1: Runtimes(in seconds) for Applying Uncertainty

10.5 Reaching in Cluttered Environments

We performed three experiments of planning with TSR intersection in a cluttered kitchen environment: reaching for the juice bottle in scene 1, reaching for the rice box in scene 1, and reaching for the juice in scene 2 (see Figure 10.4). The scene contains duplicates of every object at its pose estimates to ensure that the path generated by the planner is collision-free for all hypotheses of object pose.

The first task is fairly easy because the juice bottle is in a relatively open area. the second task is harder because the robot's arm must squeeze between two objects to grasp the box. If there were no uncertainty, the robot would simply reach for the box from the top, however the uncertainty in the pose

of the box invalidates the TSR that allows approaching from the top. The remaining TSRs only allow the robot to approach toward the thin edge of the box. The third task is the most difficult because the duplication of the refrigerator leaves very little room for the robot to approach the juice bottle. See Figure 10.4 for typical results of the three experiments.

The object pose hypotheses were sampled uniformly from $\pm$2cm in x and y and ± 0.1rad in Yaw for Scene 1. However, this proved to be too much uncertainty for the task in scene 2 (the duplicates of the refrigerator made the task infeasible). Thus, in scene 2, we used an error of $\pm$1cm in x and y and ± 0.05rad in Yaw. This is an instance of different tasks requiring different degrees of certainty to guarantee that the task is accomplished.

10.6 Summary and Discussion

We have presented an approach to addressing pose uncertainty in manipulation planning problems through the intersection of TSRs. After describing a reaching or object placement task with a set of TSRs, we can modify this set to take into account pose uncertainty by efficiently intersecting copies of each TSR in 6D pose space. If there is no point where all copies of a TSR intersect, that TSR is discarded. If all TSRs for a task are discarded, then we know that the task cannot be guaranteed to be accomplished given the current pose uncertainty. This is a key advantage of our approach because infeasible tasks can be rejected quickly, before a planner is even invoked. If any TSRs remain after intersection, we sample from the volume of intersection to generate goals that are guaranteed to succeed despite pose uncertainty. We also demonstrated the method on the WAM arm in a cluttered kitchen environment for several reaching-to-grasp tasks.

We see this work as an important first step toward addressing pose uncertainty. However the methods in this chapter only address the worst-case approach to managing uncertainty, i.e. the path generated must meet task specification for all estimates of object pose. We took this approach because we observed that an incorrect grasping pose would sometimes lead to complete task failure, such as the object being dropped, damaged, or destroyed.

On the other hand, the worst-case approach can sometimes be too stringent because it labels a task as infeasible if there is no intersection between the TSRs corresponding to *all* pose estimates. Thus, if we have a pose estimate that is far from the other estimates (i.e. and outlier), our current algorithm would label the task infeasible even though there was little probability of the outlier being the true pose. One way to address this issue is to filter out the outliers, but the difficulty there is how to correctly classify these outliers. Another approach is to assign probabilities to different pose estimates and to sample from TSRs corresponding to these pose estimates in proportion to their probability. However the problem with such a scheme is that it may take quite a while to generate samples in volumes of intersection of multiple low-probability TSRs. An approach that focuses more directly on volumes of

intersection between subsets of TSRs would be preferable.

It would also be interesting to explore an approach to sampling goal configurations in the presence of uncertainty that takes into account the degree of certainty assigned to pose estimates. Imagine that the volume between the bounds of each TSR is filled with a certainty value. When the bounds of two or more TSRs overlap, the volume of intersection is the sum of the certainty values of the intersecting TSRs. The planning algorithm can then sample from TSRs and overlapping regions in proportion to their certainty. This would give outlier TSRs little influence because they will not overlap with most TSRs while favoring intersections between high-certainty TSRs.

Chapter 11

Base Placement and TSRs

This chapter focuses on finding base placements along with end-effector poses which allow mobile manipulators and humanoids to complete pick-and-place tasks. Path planning for a mobile manipulator involves multiple levels of planning which are often divided into sub-problems to manage complexity. Consider a pick-and place operation for a mobile manipulator: the task is to move an object from some given starting configuration to some given goal configuration. The problem can be broken down into three sub-problems: 1) move the robot from its initial configuration to a configuration where it is near the object, 2) grasp the object, 3) move the robot (holding the object) to some configuration which places the object into its goal configuration. Breaking the problem into the above subproblems reduces the complexity of the overall task by allowing sub-plans to be generated in series. However, ignoring the coupling between sub-problems can turn feasible problems into infeasible ones and introduce unnecessary difficulty for the path planning algorithm. In this chapter, we will show the importance of coupling the sub-problems and present an algorithm which takes this coupling into account. This algorithm focuses on two issues central to mobile manipulation: choosing a feasible end-effector pose for grasping and choosing the optimal locations for the mobile base.

We address the issues of grasp selection and base placement from the perspective of optimization. We present three metrics for evaluating the quality of a robot configuration: grasp quality, configuration desirability, and configuration clutter. These metrics are evaluated at each candidate configuration where the robot is grasping the object—thus giving the configuration an overall score. We show how to use a co-evolutionary algorithm to search a constrained space of base placements and grasps for the optimal (as defined by the above score) robot configurations for the start and goal configurations of the object we wish to move. These configurations can then be used as the start and goal passed to a planning algorithm like CBiRRT2, which then finds a path between them.

The work discussed in this chapter was published in [103].

11.1 Problem Definition

We define the problem of pick-and-place mobile manipulation as follows:

Define the following configurations:

- qR, the full configuration of the robot, including the base and the arm
- $qArm \subset qR$, a configuration of the robot's arm joints excluding the joints of the hand
- $qBase \subset qR$, a configuration of the robot's base (usually in the XY plane)
- qO, a configuration of the object (usually the 6 DOF pose)
- qO_{start}, the starting configuration of the object
- qO_{goal}, the goal configuration of the object
- qR_{start}, the starting configuration of the robot
- qR_{Ostart}, a collision-free configuration of the robot where the robot is grasping the object in the object's starting configuration
- qR_{Ogoal}, a collision-free configuration of the robot where the robot is grasping the object in the object's goal configuration

We also assume a TSR or TSR Chain is given for grasping the object. Let $\mathbf{T}^0_{sample'}$ be a sample drawn from this TSR or TSR Chain using Equation 5.7 or Equation 6.2.

Our algorithm requires qO_{start}, qO_{goal}, and qR_{start} as input. The problem is to find a collision-free path for the robot and object which takes the object from qO_{start} to qO_{goal}. We do *not* assume that the robot starts in a configuration where it is grasping the object. qR_{Ostart} and qR_{Ogoal} are initially unknown.

We also require an IK algorithm for the given robot. In our context the IK algorithm need not find values for all DOF of the robot, only $qArm$. To illustrate, consider an arm mounted on a mobile base. In this case the IK algorithm would determine $qArm$ given $qBase$ and an end effector position determined by some $\mathbf{T}^0_{sample'}$ as arguments. The IK algorithm is robot-specific and can be iterative or analytical.

11.2 Optimization

Our approach consists of two-phases: an optimization phase and a planning phase. In the optimization phase, we use a co-evolutionary algorithm to find the optimal $qBase \subset qR_{Ostart}$, $qBase \subset qR_{Ogoal}$, and $\mathbf{T}^0_{sample'}$. After the search, we extract the accompanying $qArm$ values using IK, thus fully defining qR_{Ostart} and qR_{Ogoal}. In the path planning phase, we use an algorithm like CBiRRT2 to find a path connecting qR_{Ostart} and qR_{Ogoal}. The optimization phase is described below.

Scoring Function

In order to find the optimal grasp and base placement at both the start and goal configurations of the object we need a scoring function that judges the quality of a given robot configuration. Let $qR = IK(qBase, \mathbf{T}^0_{sample'})$ be a configuration we wish to evaluate. If qR is in collision it receives a score of 0. Otherwise, we consider three criteria to compute the score: *grasp quality, configuration desirability, and configuration clutter*. *Grasp quality* is often defined as force-closure [104], but can be measured with a variety of metrics [26, 27]. For our purposes it will suffice that the there exists some grasp-quality metric that returns a single number as a measure of grasp quality, G, given qR. Note that if the TSR or TSR Chain defined for grasping is defined in such a way that all $\mathbf{T}^0_{sample'}$ values will yield desirable grasps, this measure is unnecessary.

Configuration desirability is robot-specific, referring to the cost of being in a certain configuration of the arm. If we assume that the robot's arm can be easily controlled in any feasible configuration, there is no need to consider configuration desirability. Unfortunately, an arm's configuration space often contains singular configurations which are difficult to deal with in control. The manipulability measure is useful for gauging the desirability of a configuration because it gives better scores to configurations that are farther from singular configurations. Thus we use manipulability as our configuration desirability score, calculated as

$$M = \min(\text{eigs}(\mathbf{J}\mathbf{J}^T)) \tag{11.1}$$

where M is the manipulability score for a certain robot configuration, $\mathbf{J}$ is the Jacobian of the robot's arm evaluated at $qArm \subset qR$, and $\text{eigs}()$ is a function that returns the eigenvalues of a matrix.

Configuration clutter measures the proximity of a given robot configuration to nearby obstacles. For path planning and safety purposes, it is desirable to choose the goal configuration for the robot that is not in a cluttered area, if possible. To measure the clutter in the vicinity of the robot, we compute the distance from each of the robot's links to the nearest obstacle. This distance, d_l, is computed for each link, l, and the total configuration clutter is given by

$$C = \sum w_l d_l \tag{11.2}$$

where w_l is a constant weight for the link l. We use the inverse of the distance between l and the robot base's center of mass in a nominal configuration as w_l.

After calculating the three metrics, G, M, and C, we weigh and sum them into an overall score for the configuration, $Score(qBase, \mathbf{T}^0_{sample'})$. If qR is in collision or does not meet the minimum requirements for grasp stability (as determined by the grasp quality metric), $Score$ is set to 0. It is important to note that these metrics are not the only ones that can be used, they can be replaced by other metrics or other metrics can be used in conjunction with them. All that is required is a function that returns an $Score$ when given a base position and end-effector pose.

Co-Evolutionary Algorithm

Environments with clutter and objects which can be grasped in multiple ways make the problem of finding the optimal base positions and grasp highly non-linear with many local minima. The difficulty of the problem necessitates an optimization approach that can efficiently avoid these local minima. We use a co-evolutionary algorithm (see Mitchell's book [105] for a description of evolutionary algorithms) to search for the optimal base positions and grasp.

The evolutionary structure is set up as follows: There are three populations; a population of $qBase \subset qR_{Ostart}$ individuals, a population of $qBase \subset qR_{Ogoal}$ individuals, and a population of $Grasp$ individuals. The base position individuals are specified in polar coordinates, with the origin of the coordinate system placed at the center of mass of the object in configurations qO_{start} and qO_{goal}, respectively. The individuals also contain genes for an offset in X and Y limited to $\pm$20cm. These offset genes allow the co-evolutionary algorithm to improve on good solutions in later generations via mutation.

A $Grasp$ individual is a string of values between the non-zero bounds of $\mathbf{B}^w$ of the TSR defined for grasping. In the case of a TSR chain, these values are the joint values of the virtual manipulator. These values are converted to $\mathbf{T}^0_{sample'}$ using Equation 5.7 or Equation 6.2.

Each population is associated with its own fitness function, however the fitness of one population depends on the individuals in one or more of the other populations (this is what makes the algorithm *co*-evolutionary). The base position populations use the fitness function

$$F_{Base}(qBase) = \max_{0 \leq i < k} Score(qBase, \mathbf{T}^0_{sample'_i}). \tag{11.3}$$

$\mathbf{T}^0_{sample'_i}$ is generated from the ith individual in the $Grasp$ population in order of the $Grasp$'s fitness. k determines how many $Grasp$ individuals are to be evaluated. The fitness of $Grasp$ individuals is determined by

$$F_{Grasp}(Grasp) = \sum_{i=0}^{k} \left(Score(qBase_i \subset qR_{Ostart}, \mathbf{T}^0_{sample'}) + Score(qBase_i \subset qR_{Ogoal}, \mathbf{T}^0_{sample'}) \right) \tag{11.4}$$

where $\mathbf{T}^0_{sample'_i}$ is generated from the given $Grasp$. In this equation, i indexes over the individuals in the base position populations in order of their fitnesses. We also include a special provision for this fitness function: if *exactly one* of the scoring functions returns 0, (i.e. the resulting configuration is in collision), $F_{Grasp}(Grasp)$ is set to 1.0. This provision ensures that grasps which are valid in exactly one of the start/goal configurations of the base are preserved but not given any advantage over other such grasps. This is important for maintaining a pool of grasps that will eventually be successful in the $Grasp$ population without allowing one such grasp to dominate.

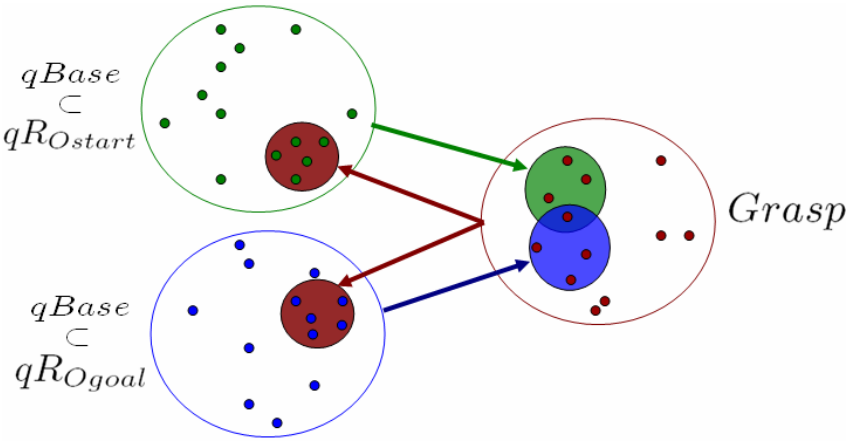


Figure 11.1: Depiction of the three populations. The best individuals in the $qBase \subset qR_{Ostart}$ population prefer the grasps in the green circle. The best individuals in the $qBase \subset qR_{Ogoal}$ population prefer the grasps in the blue circle. The best individuals in the $Grasp$ population prefer the base positions in the red circles. The grasp in the intersection of the green and blue circles is preferred by both base position populations and will receive the highest fitness in the $Grasp$ population.

The effect of the fitness functions is to guide all three populations to a solution that has maximum score (see Figure 11.1). Each base position population "pulls" the grasp population toward what the best base position individuals "want" via the F_{Grasp} fitness function. Likewise, the $Grasp$ population "pulls" both of the base position populations toward what its best individuals want via the F_{Base} fitness function. Thus the most successful individuals in the $Grasp$ population will be the ones which are preferred by *both* base position populations and the most successful individuals in the base position populations will be the ones which are preferred by the $Grasp$ population.

One cycle of evolution is defined as evolving each of the base position populations for one generation and then evolving the $Grasp$ population for one generation. For each generation in the evolution of all three populations, the parents are the top 50% of the population. Children are generated from these parents by randomly choosing two parents, performing two-point crossover, and then mutating the resulting genomes with 20% mutation probability for each gene. Mutation adds a random value to the value of the mutated gene. This random value is uniformly distributed between $\pm 1/4$th of the gene's range.

To initialize the base position populations, we need to sample positions from regions of the space that are likely to yield IK solutions. We define an annulus around the object with inner radius equal to the radius of the robot's base and an outer radius equal to the distance from the robot's base to its end effector when the arm is fully extended. Individuals in the initial start and goal base position populations are sampled from annuluses centered at the center of mass of the object in qO_{start} and qO_{goal}, respectively. Each sample is tested for collision between the base and environment obstacles and any samples in collision are rejected and re-sampled. Initial $Grasp$ individuals are randomly sampled from the TSR or TSR chain.

After running the co-evolutionary algorithm, we extract the best $Grasp$ in the population of grasps and the best $qBase \subset qR_{Ostart}$ and $qBase \subset qR_{Ogoal}$ for that grasp in the base position populations. We then compute the IK and arrive at the fully specified qR_{Ostart} and qR_{Ogoal} configurations.

11.3 Results

We conducted several experiments with a Puma robot on a cylindrical mobile base in simulation. The robot has 8 DOF: 6 DOF in the arm and 2 DOF of translation (X and Y) of the base. We define $qBase$ to be the X and Y translation and define $qArm$ to be the DOF of the arm. We consider two pick-and-place problems: moving a wineglass from a kitchen counter into a dishwasher and moving a plate from the dishwasher onto a cabinet shelf (see Figure 11.2).

The TSRs for both the wineglass and the plate where defined to have $\mathbf{T}_w^0$ at the center of the object with $\mathbf{T}_e^w$ defined to point the hand toward the center of the object along the x axis at a distance that would allow the fingers to grip the object. The $\mathbf{B}^w$ for the wineglass was defined to allow free rotation about the z axis for both objects. For the wineglass, $\mathbf{B}^w$ was also defined to allow vertical translation corresponding to the height of the object.

To gauge our co-evolutionary optimization algorithm, we performed a benchmark test against random sampling of base placement and grasp parameters. Co-evolution was run for 6 cycles with population sizes of 84 for the base position populations and 64 for the grasp population. Each run entailed 6940 scoring function evaluations. To be fair, the random sampling was restricted to base positions within

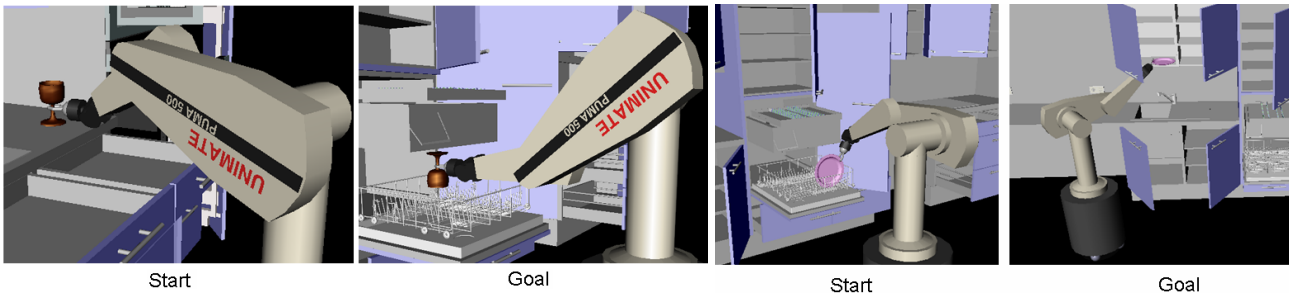


Figure 11.2: An example of the qR_{Ostart} and qR_{Ogoal} found by the optimization phase for the two problems. Note that in the goal configuration in (b) the optimizer prefers to keep the base away from the lower cabinet doors more than it prefers to keep the arm away from the upper cabinet doors. This is because the distance from the *base* link to the nearest obstacle is weighted higher than the distance from the *forearm* link to the nearest obstacle in our scoring function.

	Percent Success	Avg. Score (100 max)	Score Std. Dev.
Wineglass Problem			
Co-Evolution	99.0%	91.74	3.77
Random Sampling	99.0%	72.57	6.13
Plate Problem			
Co Evolution	100%	90.18	3.94
Random Sampling	100%	78.06	4.52

Table 11.1: Results of 200 runs of the wineglass and plate problems

the same annulus as the co-evolutionary algorithm and both random sampling and co-evolution were allowed the same number of scoring function evaluations. Both problems were run 200 times for both random sampling and co-evolution, the statistics are shown in Table 11.1. Both algorithms were able to find a feasible solution in an equal number of runs. However, co-evolution clearly outperforms random sampling in terms of both average score and consistency (standard deviation). Note that the scores are normalized to 100 maximum by the best score found after hundreds of runs of each problem.

The runtime of optimization depends almost entirely on the time needed for scoring function evaluation. In total, co-evolution took an average of 149 and 133 seconds on a 3.0 GHz Pentium 4 with 1GB of memory for the wineglass and plate problems, respectively. Random sampling took an average of 97.4 and 112 seconds on the same computer for the wineglass and plate problems, respectively. While the difference between co-evolution and random sampling runtimes may seem large, it is important to note that random sampling spends most of its time evaluating solutions that result in collision, which are quickly rejected, while co-evolution spends most of its time evaluating high-scoring solutions, which takes more time.

After generating the qR_{Ostart} and qR_{Ogoal} configurations using the co-evolutionary algorithm, we plan a path from qR_{start} to qR_{Ostart}. We then grasp the object by closing the gripper at the target point. After the grasp is complete, we compute a plan to move from qR_{Ostart} to qR_{Ogoal} with the exception that the fingers are not moved so that the grasp is maintained. Planning these two trajectories took roughly 30 seconds using a bidirectional RRT (i.e. CBiRRT with only the collision constraint), which is reasonable considering the complexity of the models used and the clutter of the space. See Figure 11.2 for examples of the qR_{Ostart} and qR_{Ogoal} found for the wineglass and plate problems.

11.4 Summary and Discussion

We have presented an optimization-based approach to grasping and path planning for mobile manipulators performing pick-and-place tasks. The method consists of a co-evolutionary algorithm to find the optimal robot configurations and grasp for the object in its start and goal configurations. Once these two configurations are found, they can be connected using a planning algorithm like CBiRRT2. Our optimization algorithm significantly outperforms random sampling and has proven effective at finding high scoring grasp/base position combinations for pick-and-place tasks in very cluttered environments.

Some interesting prospects for future work lie in creating a method that finds an intermediate grasp when no common grasp for start and goal can be found. If our optimization algorithm is unable to find a $Grasp$ and positions that are valid for qO_{start} and qO_{goal} it is likely that no such combination exists and that regrasping is necessary to move the object from qO_{start} to qO_{goal}. In that case, the optimization results are still useful because we will have grasps and base positions that are valid for both qO_{start} and qO_{goal} in the populations (but no grasp valid for both). The remaining task is to find an intermediate regrasp that links two grasps in the grasp population preferred by different base positions. An approach like that of Simeon et al. [68] could then be used to find the intermediate grasp(s) necessary to move the object from qO_{start} to qO_{goal}.

Chapter 12

Constellation - Finding Configurations that Satisfy Multiple Constraints

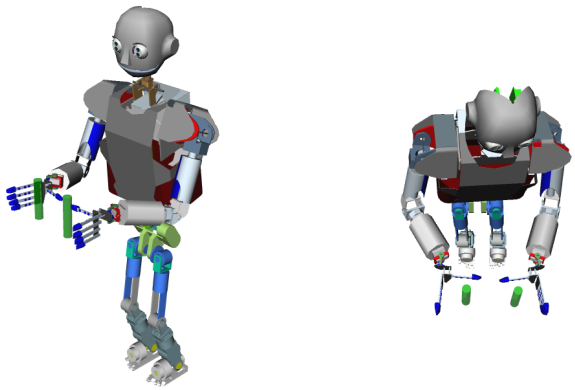
In previous sections we focused on finding configurations which meet constraints by using three methods: rejection, projection, and direct sampling. Rejection sampling works well when the constraint manifolds occupy significant volumes in the C-space, projection works well when we have an initial guess that is close to the constraint manifold, and direct sampling is only applicable when we have a parameterization of the constraint. While these strategies are useful in a great deal of situations (see, for instance, the example problems in Chapter 8), there are cases where none of these strategies is particularly effective. Generating goal configurations for a humanoid robot is one such case.

Consider a reaching task for a humanoid robot in a cluttered environment. In order to construct a path for this task, we must be able to generate one or more goal configurations, either before planning or during the planning process (as done by CBiRRT2). However, the structure of humanoid robots and the tasks they are expected to perform can severely restrict the set of feasible configurations. Humanoids must commonly obey simultaneous constraints on balance, collision-avoidance, and end-effector pose, among others. The feasible set of configurations is the intersection of the constraint manifolds corresponding to each of these constraints. It may be very difficult to parameterize this intersection set, since many of the individual constraints are not easily parameterized, which precludes using direct sampling. This intersection can also be lower-dimensional (if one of the constraints is lower-dimensional) so we cannot use rejection sampling to find a feasible configuration. Projection sampling does not require a parameterization or a specific dimensionality, but we usually do not have an initial guess close to the intersection set. Projection from a distant initial guess can fail due to opposing gradients for the various constraints, joint-limits, or singularities, even when null-space methods are used [19]. Thus we must employ a more sophisticated method to solve this kind of problem.

The work discussed in this chapter is currently in submission [106].

We approach the problem of generating configurations that satisfy multiple constraints as a set-intersection problem. The core operation used by our approach is *direct projection* – projecting a guess configuration to one of the constraint manifolds while projecting to the others in the null-space of the first constraint [19]. While this method suffers from the difficulties with projection described above, it is effective when starting close to the intersection set.

To generate a configuration near the intersection set our algorithm, *Constellation*, uses direct projection and downhill-simplex methods to generate configurations that meet at least one constraint. These configurations are then used as nodes in a graph. At each iteration of Constellation, we find the shortest cycle in this graph that contains a node on each constraint manifold and use that cycle to generate a new guess configuration, which is then projected to all the constraint manifolds. The nodes resulting from these projections are added to the graph and the process is repeated until a node satisfying all constraints is found. By focusing on short cycles of nodes that, together, meet all constraints, the algorithm is able to explore areas of C-space where the intersection of all constraint manifolds is likely to occur. The graph also allows the algorithm to proceed if a direct projection fails by reusing a node from a previous iteration. Also, since Constellation uses direct projection as its underlying projection operator, all problems that can be solved by a single direct projection will be solved by Constellation.



(a) **(b)**

Figure 12.1: The humanoid ARMAR-III [107], virtually enhanced with two 6 DOF legs (33 DOF total), performing a two-handed grasp while staying in balance. The legs are a scaled-up version of the iCub legs [108].

In the remainder of this chapter we first describe the Constellation algorithm and how we compute gradients for the various constraints. We then compare the performance of our approach to direct projection and a previously-proposed cyclic projection method [20], which attempts to find the intersection by iteratively projecting to a repeating sequence of constraints. We also compare to a variant of the cyclic projection method that uses the null-space when projecting. The four approaches are evaluated on reaching tasks for the 33 DOF ARMAR humanoid (see Fig. 12.1). Constellation is able to find

solutions where direct projection fails and also consistently outperforms cyclic projection, suggesting that it is a good choice for generating constrained configurations within reasonable time bounds.

12.1 The Constellation Algorithm

Our approach to finding a configuration that obeys a given set of constraints $\mathcal{C}$ is based on the idea of choosing a guess configuration q in each iteration and projecting it to each constraint $c \in \mathcal{C}$. Before discussing the algorithm in detail, we briefly explain how to project a configuration to a constraint.

Projecting a configuration to a constraint

A configuration q is projected to a constraint c by first computing the configuration's displacement Δx to c. This displacement can exist in an arbitrary space (commonly task space) as long as a Jacobian is available to map displacements in this space to C-space. For instance, for a task space constraint, the displacement is the difference vector between the current end-effector pose and the target pose.

Direct projection to meet a constraint c is then implemented as a gradient descent as shown in Alg. 8.

Algorithm 8: projectToConstraint($q_s, c_{prim}, \mathcal{C}_{sec}$)

while *NotStalledOrFailed()* **do**
  $\Delta x \leftarrow$ getDisplacement($q_s, c_{prim}, \delta_{prim}$)
  $\Delta x_{sec} \leftarrow$ getDisplacement($q_s, \mathcal{C}_{sec}, \delta_{sec}$)
  if $||\Delta x|| < \varepsilon$ *and* $||\Delta x_{sec}|| < \varepsilon$ **then**
    return q_s
  end
  $J \leftarrow$ getJacobian(q_s, c_{prim})
  $J_{sec} \leftarrow$ getJacobian($q_s, \mathcal{C}_{sec}$)
  $\Delta q_{error} \leftarrow J^{\#}\Delta x + (I - J^{\#}J)J_{sec}^{T}\Delta x_{sec}$
  $q_s \leftarrow q_s - \Delta q_{error}$
end
return *NULL*

Apart from the primary constraint c_{prim} that is the target for projection and the start configuration q_s, the gradient descent is given a set $\mathcal{C}_{sec}$ of secondary constraints, which should be satisfied if possible. $\mathcal{C}_{sec}$ consists of all constraints in $\mathcal{C}$ except the primary one.

We attempt to satisfy the secondary constraints in the null-space of the Jacobian-pseudoinverse $J^{\#}$ that is used for the primary target [19]. The Jacobians and displacements of the secondary constraints are stacked in J_{sec} and Δx_{sec}, respectively. Because J_{sec} can easily become highly over-constrained, we use the Jacobian-transpose instead of the pseudoinverse to obtain a movement in the null-space of the primary constraint.

When obtaining the displacements Δx and Δx_{sec}, the getDisplacement function is given the step sizes δ_{prim} and δ_{sec}, respectively. The displacements are clamped to the step size if they are longer. To keep the null-space projection from disturbing the primary projection, δ_{sec} should be smaller than δ_{prim}.

The projection fails if $||\Delta x||$ increases and stalls if the change in Δx is smaller than a certain threshold. Both conditions are accounted for by the NotStalledOrFailed Function in Alg. 8.

12.1.1 The Constellation algorithm

A common approach to finding a configuration in the intersection of several constraint manifolds is to perform direct projection as described above. However, if the initial configuration is far from the intersection of constraint manifolds, the projection can fail to find a solution that satisfies all constraints due to opposing gradients for the various constraints, joint-limits, or singularities.

Instead of trying to solve the problem using a single direct projection, the Constellation algorithm (Alg. 9) iteratively grows a graph G out of the projections of guess configurations. The graph is then used to find a promising next guess, whose projections again extend the graph.

Algorithm 9: Constellation($\mathcal{C}$)

```
1  G ← Empty Graph
2  addSample(G, C, randomSample())
3  while TimeRemaining() do
4  |   Z ← getShortestCycle(G)
5  |   q ← generateNextGuess(Z, C)
6  |   q_res ← addSample(G, C, q)
7  |   if q_res ≠ NULL then
8  |   |   return q_res
9  |   end
10 end
11 return NULL
```

The Constellation algorithm (Alg. 9) begins by generating a random configuration and applying direct projections to that configuration using a different c as the primary constraint for each projection. A direct projection is only considered successful if it results in a configuration that meets the primary

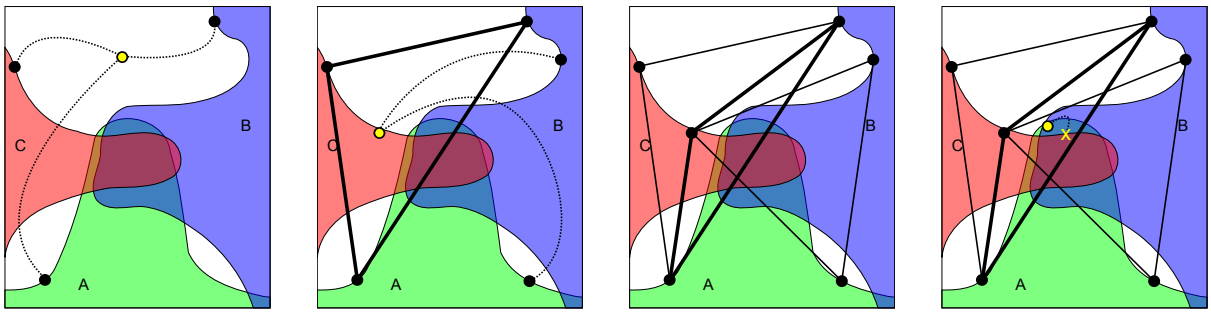


Figure 12.2: The Constellation algorithm attempting to find the intersection of the constraint manifolds A, B and C in C-space. Note that there is no guarantee that projections find the closest point on the manifold to the starting configuration. (a) The initial guess configuration is projected to the manifolds A, B and C. The guess configuration is shown in yellow, and graph nodes are shown in black. (b) Each node connects to the closest node on each constraint it does not satisfy. The graph has only one cycle, which is now used to determine the next guess. (c) The resulting nodes of projections from the guess configuration are inserted into the graph and connected as in (b). The new cycle of minimal length is highlighted. (d) A new guess is generated and projected to the manifold C, which produces a configuration in the intersection of all manifolds, thus solving the problem.

constraint (note that the resulting configuration may satisfy other constraints as well). The configurations generated by successful direct projections are added as nodes to a graph G (Alg. 10). After inserting these nodes, the algorithm creates edges between nodes using the method described in Alg. 11, the main idea being to connect nodes that satisfy different sets of constraints. The algorithm then finds the cycle of minimum length in the graph such that each constraint in $\mathcal{C}$ is met by at least one node in the cycle. This cycle marks a region where the constraint manifolds lie closer together than at any other place (according to the algorithm's current knowledge of the space), thus we would like to explore this area further. Constellation then computes a guess configuration using the nodes of the cycle (Alg. 12) and repeats the above process to generate more nodes. This process repeats until a direct projection produces a configuration that satisfies all constraints, the algorithm has considered all cycles in the graph, or the time limit is reached. An example run of the algorithm is depicted in Fig. 12.2.

To avoid choosing the same cycle repeatedly, the getShortestCycle function keeps a blacklist of cycles that have already been processed. If all cycles in G have been blacklisted, the algorithm returns failure, which is not shown in Alg. 9 for brevity.

12.1.2 Generating the next guess

The method that generates the next guess configuration is a key component of Constellation and heavily influences the performance of the algorithm. All calculations that are necessary to derive the next

Algorithm 10: addSample($G, \mathcal{C}, q$)

```
1  N ← ∅
2  for c ∈ C do
3      p ← projectToConstraint(q, c, C \ {c})
4      if p ≠ NULL then
5          if isSolution(p) then
6              return p
7          end
8          n ← addNode(G, p)
9          N ← N ∪ {n}
10     end
11 end
12 for n ∈ N do
13     connectNearestNeighbors(G, C, n)
14 end
15 return NULL
```

Algorithm 11: connectNearestNeighbors($G, \mathcal{C}, n$)

```
1 for c ∈ C do
2     if not satisfiesConstraint(n, c) then
3         m ← getClosestSatisfyingNode(G, c, n)
4         addEdge(G, n, m)
5     end
6 end
```

guess from a cycle are encapsulated in the generateNextGuess function and thus can be replaced easily. This allows us to test Constellation against variants that use other possible generateNextGuess methods.

The method presented here is inspired by the Downhill Simplex Algorithm (or Nelder-Mead-Algorithm) for the optimization of non-linear functions of several parameters [70]. We use this method because it significantly outperformed any other tested variant. This method (shown in Alg. 12) reflects the worst node n of the cycle Z through the weighted average of the other cycle nodes $Z \setminus \{n\}$. The quality of a node is determined by the sum of the displacements to each constraint. The higher the sum, the worse the quality of the node (see Alg. 13). These displacements are calculated in the space of each constraint, for instance in task space for end-effector pose constraints. An example of how a next guess configuration is generated is depicted in Fig. 12.3.

The last thing to discuss about the generateNextGuess method is the way the weighted average of a set of nodes is computed. If we simply average the node's positions without any weighting, each node would have the same influence on a joint-value in the average, regardless of this joint being used in projecting to the given constraint.

To see why this is a problem, let us consider an example problem where we have two simultaneous constraints: A pose constraint for the left hand and a pose constraint for the right hand. The left arm's joints are not important to project to the right hand constraint. Thus the corresponding columns in

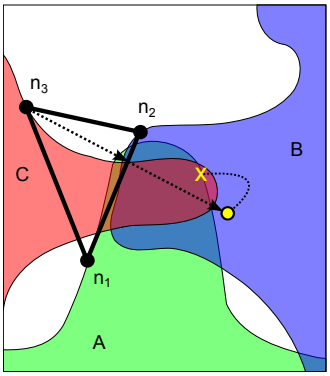


Figure 12.3: Determining the next guess from a cycle consisting of the nodes n_1, n_2 and n_3. Node n_3 is the worst node according to the displacement lengths to the constraints A, B and C. To determine the next guess configuration, the worst node n_3 is reflected through the weighted average of the cycle's other nodes n_1 and n_2. Projecting the guess configuration to constraint A then solves the problem.

Algorithm 12: generateNextGuess($Z, \mathcal{C}$)

```
1  n ← getWorstNode(Z, C)
2  c ← weightedAverage(Z \ {n})
3  return c + (c − n)
```

Algorithm 13: getWorstNode($Z, \mathcal{C}$)

```
1  w ← NULL
2  d_max ← 0
3  for n ∈ Z do
4  |   d ← 0
5  |   for c ∈ C do
6  |   |   d ← d+getDisplacement(n, c)
7  |   end
8  |   if d > d_max then
9  |   |   w ← n
10 |   |   d_max ← d
11 |   end
12 end
13 return w
```

the Jacobians used in the projection would be zero and hence the left arm's joint values would be unchanged during the projection to the right hand constraint (ignoring the null-space component). The next guess configuration would be biased toward these unchanged joint values if we simply averaged all configurations within the cycle. This would be a problem because the left arm's joints would be pulled back toward their initial values by the node that satisfies the right arm constraint and the right arm's joints would be pulled back similarly by the node that satisfies the left arm constraint. To prevent this problem, we try to decrease the influence of unchanged joint-values by using a weighted average as shown in Alg. 14.

The weightedAverage method considers the sum of all Jacobians that were necessary to project a configuration to a certain constraint. This Jacobian sum is then stored in the resulting node's meta-information. Note that the calculation of the Jacobian sum is not shown in Alg. 8 to improve readability.

To compute the weighted average, we consider each node of the given set $\mathcal{N}$ and retrieve the corre-

Algorithm 14: weightedAverage($\mathcal{N}$)

$c \leftarrow 0$
$s \leftarrow 0$
for $n \in \mathcal{N}$ **do**
  $\mathbf{J} \leftarrow$ getJacobianSum(n)
  $a \leftarrow \sum_{i=1}^{\text{size}(n)} ||\mathbf{J}_{*i}||$
  for $i \in \{1, \cdots, \mathit{size}(n)\}$ **do**
    $s_i \leftarrow s_i + \frac{1}{a} \cdot ||\mathbf{J}_{*i}||$
    $c_i \leftarrow c_i + \frac{1}{a} \cdot ||\mathbf{J}_{*i}|| \cdot n_i$
  end
end
for $i \in \{1, \cdots, \mathit{size}(n)\}$ **do**
  $c_i \leftarrow \frac{c_i}{s_i}$
end
return c

sponding Jacobian sum. A column of the Jacobian sum $\mathbf{J}_{*i}$ captures the contribution of one joint while the norm of this column $||\mathbf{J}_{*i}||$ is considered as a measure for the importance of joint i for the projection resulting in node n. We then weigh each joint value by its importance and the reciprocal of the sum of all importances of the current node before adding it to the accumulator c. weightedAverage($\mathcal{N}$) then returns the weighted average c after all nodes' contributions have been considered.

12.1.3 Constraints

To work with the Constellation algorithm, constraints must provide all necessary information for the gradient descent shown in Alg. 8. Thus a constraint has to provide two basic methods 1) the calculation of a displacement to the constraint and 2) the calculation of a Jacobian that can be used to determine a direction to move toward the constraint manifold. Both methods are specific to a constraint.

In this section, we will discuss three example constraints that we implemented and used together with the Constellation algorithm: a modified TSR constraint, a balance constraint, and a collision constraint.

The Modified TSR Constraint

The definition of a TSR is shown in Chapter 5. The task space displacement for a TSR is given by equation 5.6 and the Jacobian is simply the Jacobian of the end-effector it corresponds to. However, since we are now projecting to TSRs from distant configurations, we have found that relaxing the constraint in the following way yields better results.

If, for example, the end-effector is defined at the center of the palm, a constraint specifying a grasp of an object at position $(t_x, t_y, t_z)^T$ would have bounds such as

$$\mathbf{B}^w = \begin{bmatrix} t_x & t_x \\ t_y & t_y \\ t_z & t_z \\ 0 & 0 \\ 0 & 0 \\ -\pi & \pi \end{bmatrix}. \tag{12.1}$$

The desired hand pose is of course specific to the hand coordinate system and thus to the robot definition, but it demonstrates one issue: To allow full freedom in the yaw-component—meaning that we don't care from which direction the object is grasped—it is not always best to set the corresponding TSR entries to $-\pi$ and π. The task space displacement for this component would then always be zero, which forces a projection to keep the yaw value of the starting pose. It may be more desirable to allow free yaw when projecting from a distant configuration to give the projection more freedom. Thus when a dimension of task space is totally unconstrained we modify the TSR to take the corresponding components out of the task space displacement and the Jacobian.

The balance constraint

The second constraint to discuss is the balance constraint which is satisfied if the robot's center of gravity—projected to the ground—lies in the support polygon of the robot. For a humanoid, the support polygon would be the region under and in between the feet. A detailed discussion of the balance constraint can be found in [28].

For a given robot whose links are $l_1 = (x_{cog_1}, m_1), \cdots, l_N = (x_{cog_N}, m_N)$, each having a mass m_i and a center of gravity x_{cog_i} (in world coordinates), the robot's center of gravity is computed as

$$x_{cog} = \frac{1}{\sum_{i=1}^{N} m_i} \sum_{i=1}^{N} m_i \cdot x_{cog_i}. \tag{12.2}$$

For the following considerations let S represent the robot's support polygon and let $p(x) \in \mathbb{R}^2$ be the projection of a point $x \in \mathbb{R}^2$ onto S.

The displacement for the balance constraint is the negative vector from x_{cog} to its closest point in the support polygon. Note that this operation is done in the two dimensional ground plane, thus the z-component of x_{cog} has to be omitted.

$$\Delta x = \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot x_{cog} \right) - p\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot x_{cog} \right) \tag{12.3}$$

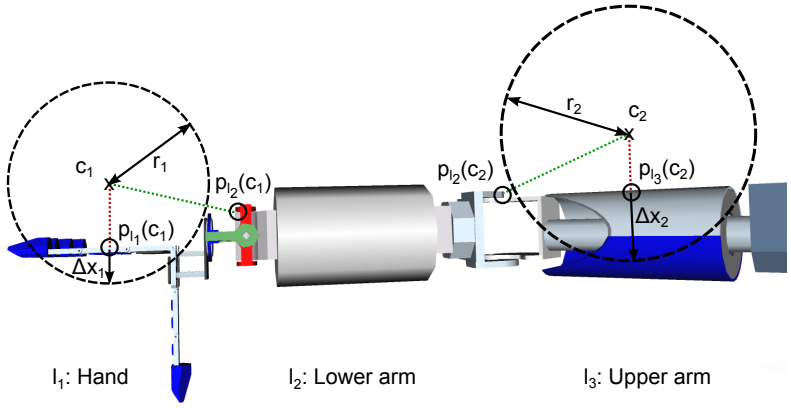


Figure 12.4: A humanoid arm consisting of three links l_1, l_2 and l_3 collides with two obstacle spheres. The collision constraint considers all combinations of spheres and links and calculates the point on each link that is closest to the sphere's center. If these points lie within a sphere we calculate a displacement d_i that is sufficient to move the link out of the sphere.

The Jacobian for the balance constraint is computed similarly to the center of gravity:

$$J = \frac{1}{\sum_{i=1}^{N} m_i} \sum_{i=1}^{N} m_i \cdot J_{l_i}(x_{cog_i}) \tag{12.4}$$

where $J_{l_i}(x_{cog_i})$ represents the Jacobian for the center of gravity of the i-th link.

The collision constraint

Constellation can also consider a constraint for collision-avoidance. We implement a simple version of this type of constraint, where we try to avoid spherical obstacles of arbitrary sizes and positions. The implementation of the collision constraint requires calculating the point on a link of the robot that has minimal distance to a given point in an obstacle. To compute those minimal distances we use the PQP collision-checker [109].

Let $\mathcal{S} = \{S_1 = (c_1, r_1), \cdots, S_N = (c_N, r_N)\}$ be the set of spherical obstacles. Each sphere S_i is defined by its center c_i and radius r_i. Let $\mathcal{R} = \{l_1, \cdots, l_K\}$ be the set of links of the robot. The point on l_i with the shortest distance to c_i will be denoted as $p_{l_i}(c_i)$ and the Jacobian for a specific point a on l_i as $J_{l_i}(a)$. Fig. 12.4 demonstrates the idea of the collision constraint.

Assuming $\{(c_1, r_1, l_1), \cdots, (c_M, r_M, l_M)\}$ is the set of sphere/link-pairs that collide and $d_i = c_i - p_{l_i}(c_i)$, the displacement for the collision constraint is

$$\Delta x_i = \left(\frac{r_i}{||d_i||} - 1 \right) \cdot d_i. \tag{12.5}$$

Δx_i is a vector of three components and Δx for the collision constraint is a concatenation all the Δx_i vectors.

The Jacobian to use with the collision constraint is then

$$J = \begin{bmatrix} J_{l_1}(p_{l_1}(c_1)) \\ \cdots \\ J_{l_M}(p_{l_M}(c_M)) \end{bmatrix}. \tag{12.6}$$

12.2 Results

In this section we present the results of several test runs in which we compare Constellation to direct projection, cyclic projection, cyclic projection with null-space, and a variant of Constellation. In the evaluations, we use a model of the humanoid ARMAR-III with 33 DOF as shown in Fig. 12.1.

We try the direct projection approach with each constraint as the primary target while attempting to satisfy the other constraints in the null-space. The approach is considered to be successful if any of these attempts results in a configuration that meets all constraints.

The cyclic projection method iteratively projects a starting configuration to a repeating sequence of constraints. This projection does not use the null-space. The constraint order is randomly determined at the beginning of a run and then kept unchanged until the run is finished. The null-space variant of cyclic projection differs from the standard method in that it uses the projection method of Alg. 8, which uses the null-space.

Table 12.1 lists and explains the constraints that are used in the test scenarios. The tolerance for meeting a constraint was set to 0.001 for all examples in this section. Before presenting the results of the test runs, we will discuss one distinct run to show some failure cases of the methods we compare to.

Name	**Definition**
Balance	The robot must be in balance, i.e. its center of gravity must be above the support polygon.
Collision	The scene contains several spheres at random positions. These spheres must not intersect the robot.
Grasp right	A cylinder in the scene has to be grasped with the right hand. The TSR for this constraint allows the grasp to rotate around the z-axis of the cylinder.
Stand	The robot has a fixed base in the right foot. The TSR for this constraint requires the left foot to be put near the right foot with the same orientation.

Table 12.1: Constraints used in the test scenarios

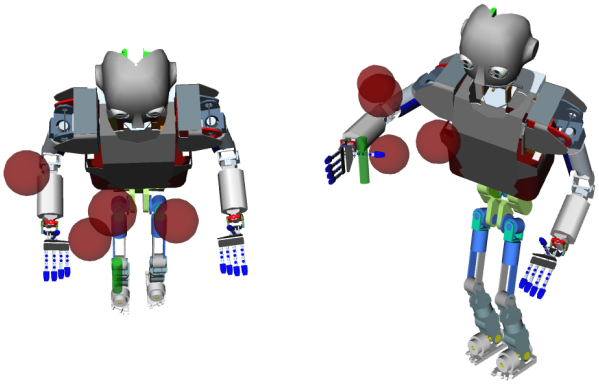


Figure 12.5: The example setup: The initial configuration on the left and the solution found by Constellation on the right.

12.2.1 An example test run

In this section we will present an example scene which cannot be solved using direct projection or cyclic projection, but can be solved by Constellation. The scene consists of 4 obstacle spheres and is depicted in Fig. 12.5.

When performing a direct projection to the grasp constraint, the robot leans to the side to reach the object. After the primary target (the grasp) is reached, the balance constraint and the stand constraint are not satisfied. To re-balance the robot, the balance constraint uses the left foot which is freely movable. This is a conflict between the balance constraint and the stand constraint which requires the left foot to be placed at a certain position next to the right foot. This conflict causes the projection to stall before finding a configuration that meets all constraints.

A direct projection to the stand constraint has similar results: After the primary target is reached, the grasp constraint tries to move the robot's hand towards the object within the null-space of the stand constraint. During this movement, the arm will intersect with the obstacles resulting in the grasp constraint opposing the collision constraint. Direct projections which use the other constraints as primary targets likewise are not able to satisfy all the constraints. These conflicts make the example scene unsolvable for a direct projection, no matter which constraint is used as the primary constraint.

Cyclic projection succeeds in projecting to all the constraint manifolds, however the method eventually stalls (i.e. it bounces repeatedly between nearly identical points on several constraint manifolds). As shown in [20], this method can easily stall if the constraint manifolds are non-convex.

12.2.2 Evaluation

Now that the difficulties of a distinct scene are clear, we present the results of a statistical test in which we gradually increase the complexity of the scene by adding more obstacle spheres at random positions. Each level of complexity, determined by the number of obstacle spheres in the scene, is tested with 10 different random seeds. One run, i.e. a specific level of complexity together with a specific random seed, is then performed using direct projection, cyclic projection, cyclic projection using the null-space, Constellation, and a variant of Constellation that only uses weighted averaging (as described in Alg. 14) to generate the next guess (i.e. no downhill-simplex is performed).

To initialize each algorithm we use the start configuration depicted in Fig. 12.5 and add a small random offset in C-space with a length of 0.1rad.

Table 12.2 lists the results of the different tests we ran. The obstacles were placed at random positions between the robot and the object to grasp. Obstacles were not allowed to intersect with a certain area around the robot and the object to avoid unsolvable setups as much as possible. Nevertheless unsolvable scenes are possible and likely. We consider a scene to be solvable if any of the tested methods succeeded. The percentages given in the Table 12.2 are relative to the number of solvable scenes (not to the total number of tested scenes).

# Sph.	Direct	Cyclic	Cyclic (ns)	Wgh. Avg.	Const.
0-2	95.83%	95.83%	100.00%	100.00%	100.00%
3-5	80.00%	75.00%	85.00%	80.00%	95.00%
6-8	56.25%	56.25%	56.25%	56.25%	100.00%
9-11	66.67%	66.67%	66.67%	66.67%	100.00%
Overall	78.26%	76.81%	81.16%	79.71%	98.55%

Table 12.2: Test results (7cm collision spheres)

The test used obstacle spheres with a radius of 7cm. Table 12.3 lists the results of a second test which used obstacle spheres with a radius of 2cm, giving the robot more space to maneuver. The sphere radius is the only difference between the two tests.

The results show that Constellation consistently outperforms the other approaches, i.e. a direct projection or cyclic projection with or without using the null-space. Making use of the null-space in cyclic projection considerably improves the results for this approach. However as the complexity of the tests increases, the percentage of scenes that are solved by Constellation, but not by cyclic projection reaches 23.8% (see Table 12.3). Cyclic projection without null-space and direct projection solve about half of the scenes Constellation solves at the highest level of complexity.

The results in Table 12.2 often show the same percentage mainly because the bigger obstacles make

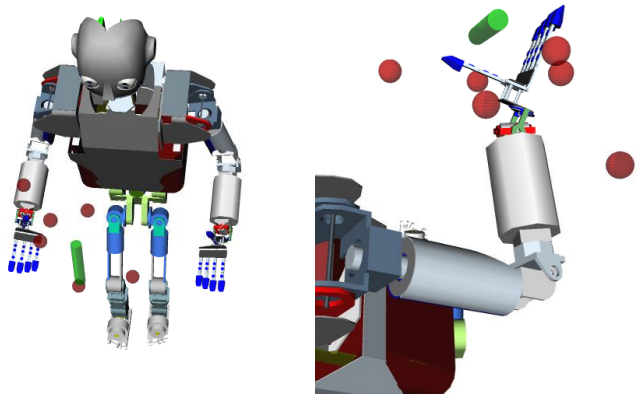


Figure 12.6: An example using 6 obstacle spheres of 2cm radius each. The initial setup on the left and the solution on the right.

# Sph.	Direct	Cyclic	Cyclic (ns)	Wgh. Avg.	Const.
0-2	93.10%	93.10%	96.55%	93.10%	96.55%
3-5	79.17%	75.00%	87.50%	79.17%	100.00%
6-8	62.50%	62.50%	75.00%	75.00%	95.83%
9-11	52.38%	47.62%	76.19%	71.43%	100.00%
Overall	73.47%	71.43%	84.69%	80.61%	97.69%

Table 12.3: Test results (2cm collision spheres)

the scenes unsolvable more quickly. In the higher levels of complexity (6-11 spheres) only a few of the scenes were still solvable.

Another interesting point about the results is that the method for generating the next guess (Alg. 12) is quite important. Instead of reflecting the worst node through the weighted average of the shortest cycle's other nodes, we tried performing a weighted average of the shortest cycle's nodes. This approach is more successful than projecting directly or cyclic projection without the null-space, but it is not as effective as cyclic projection using the null-space. Thus the way we generate the next guess is a key component of Constellation and it allows the algorithm to outperform the other approaches.

Table 12.4 compares Constellation to the cyclic method and the cyclic method using the null-space in terms of runtime. The runtimes are averaged over all runs considered solvable. Runs that timed out although the corresponding setup was solved by another method count with a runtime of 600s (the timeout value). The number of timeouts that occurred is given in parentheses.

Note, that we neither optimized Constellation nor the other methods regarding runtime. The measured runtimes refer to the test with 2cm collision spheres shown in Table 12.3.

# Spheres	Cyclic	Cyclic (ns)	Constellation
0-2	100.9 (2)	27.4 (1)	29.2 (1)
3-5	222.4 (6)	97.5 (3)	50.3 (0)
6-8	254.2 (9)	175.2 (6)	87.6 (1)
9-11	394.0 (11)	190.9 (5)	43.5 (0)

Table 12.4: Runtime comparisons (in seconds)

12.3 Summary and Discussion

The structure of humanoid robots and the tasks they are expected to perform can severely restrict the feasible robot configurations to a small or even lower-dimensional set. We approach the problem of generating configurations in this set by searching for a point in the intersection of all constraint manifolds in C-space. Starting with an initial guess, our algorithm, *Constellation*, iteratively moves closer to the intersection of the constraint manifolds using a combination of projection and downhill-simplex methods. Since Constellation uses direct projection as its underlying projection operator, all problems that can be solved by a single direct projection will be solved by Constellation. We compared the performance of our approach to direct projection and a previously-proposed cyclic projection method on reaching tasks for a 33 DOF humanoid robot. Constellation is able to find solutions where direct projection fails and also consistently outperforms cyclic projection.

Significant work remains to be done to analyze how Constellation performs and why it is so successful. One interesting area to explore is the effects of using null-space projection. Intuitively, it seems that attempting to satisfy the other constraints in the primary constraint's null-space will move configurations closer to the intersection set than simply projecting to each constraint without using the null-space. Indeed we observed that this was a much more effective strategy for Constellation and we have quantified its benefits for cyclic projection as well. However, since constraint manifolds can be non-convex and disjoint, it is not clear that this strategy will always be beneficial, as it may lead to configurations that are near a disconnected component of a secondary constraint. In this case, the cycle containing this node may be short, but the guess configuration generated from it may not be useful.

Another area to explore is a more principled construction of the graph used by Constellation. The current algorithm simply connects each node to the nearest-neighbor on each other constraint. However, as in the case described above, the nearest neighbor is not always the best choice for forming a cycle. Ideally we would like to connect a node to all other nodes on all other constraint manifolds. However, when we do this the cycle computation becomes quite time-consuming and many of the cycles are too large to be useful. A method that connects nodes based on a combination of the distance between them and the likelihood that the connection would contribute to the diversity of cycles in the graph would be beneficial.

Chapter 13

Planning with Soft Constraints: Exploring Cost-Space Chasms

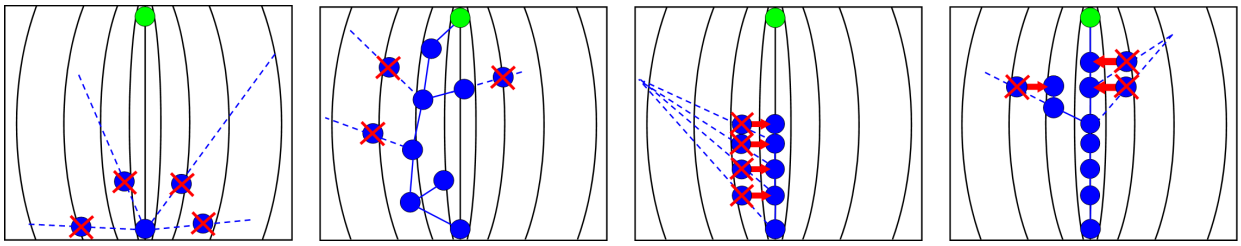
Our recent work has explored extending the methods developed in Sections 4–7 to planning with soft constraints—i.e. constraints that we would prefer to satisfy but that are not required to be met. Soft constraints can be encoded into a cost function $G(q) \in \mathcal{R}_{\geq 0}$, which represents the cost of a configuration (note that we are again addressing scleronomic holonomic constraints). The problem definition (using the notation of Chapter 4) combining both hard and soft constraints would then be

$$\underset{\tau}{\operatorname{argmin}} \int_{q \in \tau} G(q) : q \in \mathcal{M}_{\mathcal{C}_i} \quad \forall q \in \tau(\mathcal{S}_i) \quad \forall i \in \{1...n\}. \tag{13.1}$$

Planning paths for manipulators becomes more difficult when we wish to optimize the cost of the path in addition to satisfying hard constraints. The high dimensionality of the problem precludes the exhaustive computation necessary to find the globally optimal path. Thus our efforts in this area do not focus on providing a guarantee for the optimality of a path, though this would of course be desirable. Instead, we seek to develop a practical algorithm for producing low-cost paths for problems commonly encountered in manipulation planning and evaluate it empirically relative to the state-of-the-art.

Recently the Transition-based RRT (T-RRT) [73] was presented as a method to manage the growth of a search tree in a high-dimensional cost-space. This algorithm uses a process similar to stochastic optimization methods, where a temperature parameter determines whether to accept a certain extension of the tree. When the T-RRT is stuck in a local minimum (i.e. many extensions are being rejected), the temperature is increased to allow more exploration. When the algorithm accepts a higher-cost

The work discussed in this chapter was published in [110].



(a) T-RRT Start **(b)** T-RRT Final **(c)** GradienT-RRT Start **(d)** GradienT-RRT Final

Figure 13.1: Illustration of the two algorithms' performance in a cost-space chasm. The left image of each pair shows the state of the tree before a temperature increase and the right image shows the tree after the increase. The T-RRT is unable to sample the bottom of the chasm and increases temperature while GradienT-RRT uses gradient-descent (red arrows) to make progress before increasing temperature. GradienT-RRT generates a higher-quality path because it is able to generate nodes at the bottom of the chasm.

extension, the temperature is decreased so the tree does not over-explore high-cost regions. In this way, T-RRT is biased to explore lower-cost regions before allowing higher-cost extensions which may be necessary to find a feasible path.

The T-RRT was shown to be very successful on many problems and consistently outperforms the standard RRT [99] and the heuristically-guided RRT [74]. However, we have found that the T-RRT does not perform well for cost functions that induce *cost-space chasms*–narrow or even lower-dimensional low-cost passages around which cost increases. Such cost functions are especially relevant in manipulation planning where we may seek to penalize deviation from a lower-dimensional constraint on end-effector pose or navigate a narrow low-cost region in a manipulator's workspace. The T-RRT struggles in the cost-space chasms created by such cost functions because it relies on the sampling strategy of the RRT, which makes sampling in such lower-dimensional or narrow regions impossible or extremely unlikely. Thus almost every extension becomes a cost increase if the starting configuration is at the bottom of the chasm. As a result, the T-RRT explores the chasm slowly and the path generated lies on the higher-cost sides of the chasm, not on the lower-cost bottom (see Figure 13.1). Cost-space chasms thus significantly hinder the performance of the T-RRT for many types of cost functions relevant to manipulation planning.

To address this problem we present the GradienT-RRT algorithm, which combines the strengths of the T-RRT with local gradient-descent. The algorithm works by the same principles as the T-RRT, except that a gradient step is included during the extension. The gradient of the cost function allows this algorithm to explore spaces that are too narrow or lower-dimensional to explore by sampling alone. A key issue with incorporating the gradient is to avoid trapping the tree in local minima and to retain the beneficial exploration properties of the T-RRT.

In the follow subsections we describe the T-RRT, formally define cost-space chasms, and show why they hinder the performance of the T-RRT. We then introduce the GradienT-RRT and discuss the

trade-offs inherent in using the gradient. To show the versatility and practicality of our approach we also present general cost functions in workspace, task space, and C-space, and show how to compute their gradients efficiently. We then compare the performance of GradienT-RRT and T-RRT on three example problems where these cost functions induce cost-space chasms. Finally, we demonstrate a real-world implementation of GradienT-RRT, where it is used to plan a path amongst uncertain workspace occupancy for the HERB robot.

13.1 T-RRT

The purpose of the T-RRT is to find feasible low-cost paths through high-dimensional cost-spaces. The algorithm manages the trade-off between optimality and exploration by using a transition test similar to stochastic optimization methods. T-RRT maintains a temperature value $temp$ which determines the probability of allowing a higher-cost node to be added to the tree. $temp$ is automatically tuned by the algorithm and its behavior is controlled through several parameters, the most important being $nFailMax$. $nFailMax$ determines how many higher-cost nodes are rejected before increasing temperature.

We show a bi-directional implementation of the T-RRT in Algorithms 15 and 16. The algorithm is identical to the Bidirectional RRT except for the call to the ConsiderCost function (Algorithm 17 with GradienTRRT = False) in the extension. This function checks if a new configuration q_s has cost less than or equal to the cost of its parent. If it does, q_s is added to the search tree and the extension continues. If not, the cost difference is put through a temperature check. If it passes, the configuration is added to the tree, $temp$ is decreased by a factor of α, and $nFail$ is set to 0. If not, $nFail$ is incremented and the extension terminates. If $nFail$ reaches $nFailMax$, $temp$ is increased by a factor of α and $nFail$ is set to 0.

Thus the T-RRT tries growing lower-cost nodes $nFailMax$ number of times before allowing a higher probability of accepting a higher-cost node. Such an approach is quite reasonable because we want the algorithm to explore low-cost regions while avoiding being trapped in local minima. The $nFailMax$ parameter effectively controls the path quality vs. search-time of the algorithm. Higher $nFailMax$ values bias the algorithm toward reducing cost at the expense of exploration. Thus the search takes more time for higher $nFailMax$ if a cost increase is necessary to find a feasible path.

Our description of the T-RRT above does not explicitly discuss enforcing hard constraints (except collision) because the focus of this chapter is on soft constraints. However, our implementation does plan with hard constraints as well. The integration of the two constraint types is fairly straightforward: after generating a q_s in either the Extend function or in the ConsiderCost function and before evaluating its cost, we apply the ConstrainConfig function (Algorithm 6) to produce a new q_s that meets the hard constraints, if possible. If ConstrainConfig cannot generate such a q_s, the extension terminates.

Algorithm 15: Bidirectional T-RRT(q_s, q_g)

T_a.Init(q_s); T_b.Init(q_g);
T_a.temp = T_b.temp = initTemp;
while TimeRemaining() **do**
  $q_{rand} \leftarrow$ RandomConfig();
  $q^a_{near} \leftarrow$ NearestNeighbor(T_a, q_{rand});
  $q^a_{reach} \leftarrow$ **Extend**(T_a, q^a_{near}, q_{rand});
  $q^b_{near} \leftarrow$ NearestNeighbor(T_b, q^a_{reach});
  $q^b_{reach} \leftarrow$ **Extend**(T_b, q^b_{near}, q^a_{reach});
  if q^a_{reach} = q^b_{reach} **then**
    $P \leftarrow$ ExtractPath(T_a, q^a_{reach}, T_b, q^b_{reach});
    return SmoothPath(P);
  else
    Swap(T_a, T_b);
  end
end
return NULL;

Algorithm 16: Extend(T, q_{near}, q_{target})

$q_s \leftarrow q_{near}$; $q^{old}_s \leftarrow q_{near}$;
while *true* **do**
  if $q_s = q_{target}$ **then**
    return q_s;
  else if $\|q_{target} - q_s\| > \|q^{old}_s - q_{target}\|$ **then**
    return q^{old}_s;
  end
  $q^{old}_s \leftarrow q_s$;
  $q_s \leftarrow q_s + \min(\Delta q_{step}, \|q_{target} - q_s\|) \frac{(q_{target} - q_s)}{\|q_{target} - q_s\|}$;
  $q_s \leftarrow$ ConsiderCost(T, q^{old}_s, q_s);
  if $q_s \neq$ NULL **then**
    T.AddVertex(q_s, c);
    T.AddEdge(q^{old}_s, q_s);
  else
    return q^{old}_s;
  end
end

Algorithm 17: ConsiderCost(T, q^{old}_s, q_s)

if *not* CollisionFree(q^{old}_s, q_s) **then**
  return NULL;
if $c_j < c_i$ **then return** q_s;
$d_{ij} \leftarrow \|q^{old}_s - q_s\|$;
$p \leftarrow exp\left(\frac{-(G(q_s) - G(q^{old}_s))/d_{ij}}{T.temp}\right)$;
if Rand$(0,1) < p$ **then**
  $T.temp \leftarrow T.temp/\alpha$;
  $T.nFail \leftarrow 0$;
  return q_s;
else
  if GradienTRRT **then**
    $\nabla q \leftarrow$ GetGradient(q_s);
    $q_g \leftarrow q_s - \nabla q$;
    $d_{ij} \leftarrow \|q^{old}_s - q_g\|$;
    $p \leftarrow exp\left(\frac{-(G(q_g) - G(q^{old}_s))/d_{ij}}{T.temp}\right)$;
  end
  if $T.nFail > T.nFailMax$ **then**
    $T.temp \leftarrow \alpha(T.temp)$;
    $T.nFail \leftarrow 0$;
  else
    $T.nFail \leftarrow T.nFail + 1$;
  end
  if GradienTRRT ***and*** CollisionFree(q^{old}_s, q_g) ***and*** $(G(q_g) < G(q^{old}_s)$ ***or*** Rand$(0,1) < p)$ **then**
    return q_g;
  end
  return NULL;
end

13.2 Cost-Space Chasms

Although the T-RRT works quite well in many scenarios, we found that it did not perform well when it needed to traverse narrow regions of low cost surrounded by regions of increasing cost. These structures are especially common in manipulation planning problems. For example, they arise from cost functions that bias the planner to imitate a demonstrated path, prefer an end-effector orientation, or avoid areas of uncertain occupancy in the workspace (see Section 13.5).

The cost functions in these problems create narrow or lower-dimensional low-cost regions surrounded by increasing cost. We call this type of structure a *cost-space chasm*, which we will define formally by introducing a property called c-*goodness* (inspired by ϵ-*goodness* [111]).

Let $\mathcal{Q}$ denote the C-space and $\mathcal{F} \subseteq \mathcal{Q}$ denote the free C-space. For a point $p \in \mathcal{F}$, let $S(p)$ consist of those points of $\mathcal{F}$ that can be connected to p by a straight line. For a $\mathcal{X} \subseteq \mathcal{Q}$, let $\mu(\mathcal{X})$ denote its volume.

If the cost of configurations were uniform (i.e. we consider only obstacles) a useful measure of the narrowness of the space at some point $p \in \mathcal{F}$ would be the ratio of the volume reachable from that point by a straight line to the volume defined by the sampling bounds $(\mu(S(p))/\mu(Q))$. However, if the cost of configurations in $S(p)$ is not uniform, it will be more difficult to reach some configurations in $S(p)$ than others using a planner like the T-RRT. The work along the line segment from p to $q \in S(p)$ determines the level of difficulty and it is defined as

$$W(p,q) = \int_{\mathcal{L}_{pq}^{+}} \frac{\partial G(\mathcal{L}_{pq}(t))}{\partial t} dt \tag{13.2}$$

where $G(\cdot)$ is the cost of a configuration, $\mathcal{L}_{pq}$ is the line segment from p to q, and $\mathcal{L}_{pq}^{+}$ is the portion of the line segment where the cost function has positive slope. This function captures the amount of cost-increase we must overcome to move from p to q. Note that the definition of work in [73] contains another term that accounts for the length of $\mathcal{L}_{pq}$, but it is not relevant for computing c-goodness.

We would now like to compute the *work-weighted* volume of $S(p)$ and compare that to the volume of the C-space. We weigh all $q \in S(p)$ using a function $f(p,q)$ and the work-weighted volume of $S(p)$ is the integral of $f(p,q)$ over all $q \in S(p)$. There is no single correct way to define $f(p,q)$, as it represents a mixture between the units of work and volume. However, $f(p,q)$ should have the following properties:

1. $f(p,q) \in [0,1]$
2. $f(p,q_1) < f(p,q_2)$ if and only if $W(p,q_1) > W(p,q_2)$

3. $f(p, q_1) = f(p, q_2)$ if and only if $W(p, q_1) = W(p, q_2)$

A weight of $f(p, q) = 1$ should mean that moving from p to q requires no work and $f(p, q) = 0$ should mean that it requires infinite work. There are many ways to define an $f(p, q)$ with these properties; for example $f(p, q) = e^{-W(p,q)/k}$ for some positive constant k.

If a p satisfies the equation

$$\int_{S(p)} f(p, q)dq \geq c\mu(\mathcal{Q}) \tag{13.3}$$

for a non-negative c, then we say that p is c-good. Defining $f(p, q)$ according to the above requirements produces what we would expect for the uniform-cost case: $f(p, q) = 1$ for all q and the left side of Equation 13.3 becomes $\mu(S(p))$. Thus c-goodness for the uniform-cost case is simply a measure of the narrowness of the space ($\mu(S(p)) \geq c\mu(Q)$).

A cost-space chasm is then a set of connected configurations with low c-goodness. We term such sets "cost-space chasms" because they resemble the deep steep-sided chasms between mountains when visualized.

Despite the T-RRT's success in many scenarios, we found that it did not perform well for high-dimensional problems involving such chasms. To understand why, consider a node of the tree that is near the bottom of a chasm. This node will have low c-goodness, which means that we have a low probability of generating a collision-free extension that is not a significant cost increase. The T-RRT's exploration in this type of scenario is thus quite slow and the resulting paths lie on the higher-cost sides of the chasm, not on the lower-cost bottom.

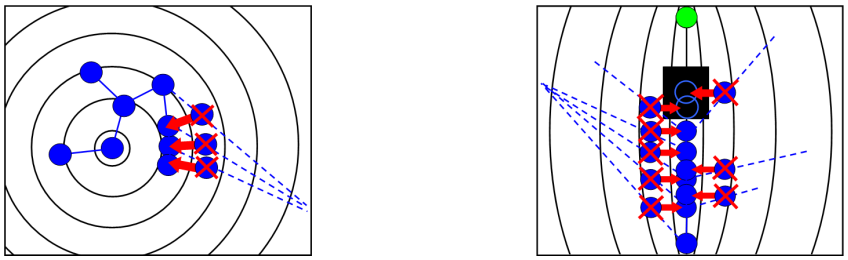
13.3 GradienT-RRT

To address these drawbacks, we propose combining the T-RRT with local gradient-descent of the cost function. Because gradient-descent does not rely on sampling to find lower-cost configurations, we can use it to navigate cost-space chasms. However, we must be very careful in how we include the gradient. For example, a naive strategy would be to apply the gradient at every node generated by the T-RRT. Doing so would inhibit the T-RRT's exploration and cause the tree to be trapped in local minima.

The GradienT-RRT algorithm differs from T-RRT in the ConsiderCost function (Algorithm 17 with GradienTRRT = True). If the configuration q_s is rejected (because it is higher-cost and it failed the

temperature check), then we compute the gradient at q_s and take a gradient step away from q_s. This produces the configuration q_g. If q_g is lower cost than the parent of q_s or passes the temperature check, we accept it. If q_g has a higher cost and fails the temperature check, we reject it and terminate the extension.

It is important to note that the $nFail$ counter and the temperature are not affected by the acceptance/rejection of q_g in GradienT-RRT; $nFail$ is only modified with respect to the cost of q_s. Not modifying $nFail$ and the temperature for gradient nodes is important because sometimes nodes generated using the gradient are only slightly higher in cost than the parent of q_s (for instance consider a tree trying to grow out of a cost-space bowl; the gradient will push the extension back into the bowl, as in Figure 13.2a). Such nodes will pass the temperature check and decrease the temperature, which can trap the tree in a local minimum (effectively causing it to spiral around the bowl) and prevent the temperature from increasing significantly.



(a) **(b)**

Figure 13.2: Illustration of two difficult situations for GradienT-RRT. (a) The tree must grow out of a cost-space bowl. (b) The tree's progress in a chasm is blocked by an obstacle. In both of these situations, GradienT-RRT will generate more nodes in the low-cost region than T-RRT without making significant progress (the number of extra nodes depends on the $nFailMax$ parameter). This will slow down the algorithm, however the tree will eventual grow out of the bowl and around the obstacle as the temperature increases.

Incorporating the gradient in this way allows us to preserve the exploration properties of the T-RRT while addressing its weakness in navigating cost-space chasms. However, in situations where the gradient does not help the tree grow, GradienT-RRT may perform slower than T-RRT (see Figure 13.2), although it will likely find a lower-cost path.

GradienT-RRT also inherits the probabilistic completeness of T-RRT because it generates the same nodes as T-RRT plus extra nodes resulting from the gradient steps. Since generating extra nodes can only increase coverage of the feasible manifold as time goes to infinity, the probabilistic completeness of T-RRT is preserved.

13.4 Costs and Gradients

The GradienT-RRT method relies on a fast computation of the gradient of the cost at a given configuration. For robot manipulators, many useful cost functions reside in one of three spaces: workspace, task space, and C-space. We present a general cost function for each space that is useful for manipulation planning and describe how to compute the cost of a configuration $G(q)$ as well as the gradient ∇q efficiently.

13.4.1 Workspace Costs

Workspace constraints are the most common constraints in motion planning, where they are used to represent obstacles. However these constraints are usually binary; either the robot is in collision or it is not. Imagine, however, that the robot was planning in a workspace with uncertain occupancy, so that the cost of a configuration depended on the belief of the robot being in collision. Our approach to computing the cost and gradient for such a scenario is similar to that used in CHOMP [112], which considers distance to obstacles as the cost of a configuration.

(a) **(b)**

Figure 13.3: (a) Example of a scene with uncertain occupancy generated using a laser-scanner. Pink: known obstacles, blue: uncertain voxels. (b) Five hundred points sampled in the volume of the Barrett arm with a bias toward the links of the hand.

Let the workspace cost be $C_w(x)$ for $x \in \mathbb{R}^3$. $G(q)$ is then the integral of $C_w(x)$ over $x \in R(q)$, where $R(q)$ is the space occupied by the robot at the configuration q. Since computing the exact integral is prohibitively expensive, we perform the following process to quickly approximate its value.

First, we can pre-compute a voxelization of $C_w(x)$ to save on computation time when evaluating $G(q)$ in the planner. To do this, we discretize the workspace into a voxel grid V. Each voxel $v \in V$ is assigned a cost $C_w(v)$ from the workspace cost at its center, which reflects the desirability of occupying

v with any part of the robot. In the uncertainty example above, $C_w(v)$ would depend on the probability of v being occupied by an obstacle (see Figure 13.3a).

We also pre-compute a set of points within the volume of each of the robot's links. When the robot is placed in a given configuration, we transform the points inside each link by the link's pose and check which voxels the points occupy. The sum of the voxel cost for each point is the approximated cost of the configuration. Note that this process can be made arbitrarily accurate by adding more points and by increasing the resolution of V, though the time needed to evaluate the cost will increase.

To obtain the points in the robot's volume, we first divide each robot link into simplices using a combination of convex decomposition [113] and Delaunay Triangulation. We then choose a simplex with probability proportional to its volume and generate a sample point within that simplex by taking a random convex combination of its vertices [102]. We repeat this process for some number of iterations to generate a uniformly sampled set of points X in the robot's volume. The sampling can also be biased toward lower-volume but more important parts of the robot (such as the fingers) by biasing the selection of simplices (see Figure 13.3b).

Besides evaluating the cost of a configuration, GradienT-RRT requires the gradient of the cost function. To compute the gradient $\boldsymbol{\nabla} x$ of a point $x \in R(q)$, we can take the partial derivative of the cost function along each spatial dimension and evaluate it at x. However, $\boldsymbol{\nabla} x$ exists in the workspace and we must transform this vector into the C-space in order to obtain $\boldsymbol{\nabla} q$. We can do this through the Jacobian $\mathbf{J}(q, x)$, which is defined by

$$\dot{x} = \mathbf{J}(q, x)\dot{q}. \tag{13.4}$$

We can then generalize the Jacobian to account for multiple points $x_1, x_2, ... \in R(q)$

$$\mathbf{J}(q, x_1, x_2, ...) = \begin{bmatrix} \mathbf{J}(q, x_1) \\ \mathbf{J}(q, x_2) \\ \vdots \end{bmatrix}. \tag{13.5}$$

The C-space gradient is then

$$\boldsymbol{\nabla} q = \mathbf{J}(q, x_1, x_2, ...)^T \left[\boldsymbol{\nabla} x_1^T, \boldsymbol{\nabla} x_2^T, ...\right]^T. \tag{13.6}$$

In order to implement the above process efficiently we must have a fast way to compute $\boldsymbol{\nabla} x$. We can use V to approximate this gradient by first finding the voxel v that contains x and then computing the

partial derivative as the difference in cost between voxels neighboring v along each spatial dimension. We can thus compute the gradient for each $x \in X$ and arrive at the C-space gradient via Equation 13.6.

13.4.2 Task Space Costs

Another important class of constraints lie in the Task Space of robot, which is the space of the pose of the robot's end-effector. This space includes both the translation and rotation components and it is homeomorphic to $SE(3)$.

In Chapter 5, we described TSRs, which specify sets of valid configurations in $SE(3)$. We showed that such sets are particularly useful for specifying manipulation tasks such as reaching to grasp an object or manipulating objects with pose constraints. For more complex pose constraints, we have also described the TSR Chain representation in Chapter 6, which allows TSRs to be defined relative to each other.

TSRs and TSR Chains previously defined binary constraints on end-effector pose: either the end-effector was inside the volume allowed by the TSR or it was not. We can extend these representations for cost-space planning by using the task space distance between the end-effector pose at q and the nearest TSR or TSR Chain as $G(q)$. Once we have this distance, the gradient can be computed using the Jacobian pseudo-inverse method (Algorithm 1).

13.4.3 C-Space Costs

Finally, we consider constraints defined in the C-space. Many useful constraints can be defined in the C-space; including constraints on torque and joint limits. As an example, we present a cost function that is designed to bias the planner toward a set of known configurations. This cost function is useful for biasing the search toward a desired configuration, such as elbow-up or elbow-down, or toward a desired path.

Suppose we have a finite set of configurations U with associated costs $G(u)$ for all $u \in U$. We would like to evaluate the cost of a configuration $q \notin U$. Since we would like to bias the planner toward U, the cost should increase as we move away from U. To achieve this behavior, we propose a cost function that has the following desirable properties:

1. The function is smooth.

2. All $u \in U$ contribute to $G(q)$.

3. $G(q)$ is affected more by closer $u \in U$.

The parameters needed to define the function are the following:

- The finite set of configurations U along with $G(u)$ for all $u \in U$
- A covariance Σ for each $u \in U$

where Σ is an $n \times n$ symmetric matrix used to weight the distance metric. Thus the weighted squared distance between q and the ith configuration $u_i \in U$ is $d_i = (q - u_i)^T \Sigma_i^{-1} (q - u_i)$. The cost function is

$$G(q) = s \sum_{i=1}^{|U|} \left(\frac{G(u_i)}{d_i} + 1 \right) \qquad s = \left(\sum_{j=1}^{|U|} \frac{1}{d_j} \right)^{-1}. \tag{13.7}$$

The derivative of the weighted squared distance to u_i is $d_i' = 2(q - u_i)^T \Sigma_i^{-1}$. The gradient of the cost is then

$$\boldsymbol{\nabla} q = s \sum_{i=1}^{|U|} \left[\left(\frac{G(u_i)}{d_i} + 1 \right) \left(\sum_{j=1}^{|U|} \frac{d_j'}{d_j^2} \right) s - \frac{G(u_i) d_i'}{d_i^2} \right]. \tag{13.8}$$

Note that a small offset is added to d_i when the distance approaches zero to prevent numerical instability.

13.5 Example Problems

We now present three manipulation planning problems which consider different types of cost. Each problem is designed to highlight the need for an efficient method of navigating cost-space chasms. We compare the performance of the T-RRT and GradienT-RRT on the three problems for varying values of $nFailMax$ (see Figure 13.7) and discuss the results. We set the parameters $\Delta q_{step} = 0.05$, $initTemp = 0.01$, and $\alpha = 2$ for all experiments. The magnitude of the gradient step was bounded to be less than 0.05rad for the first and third examples and 0.1rad for the second.

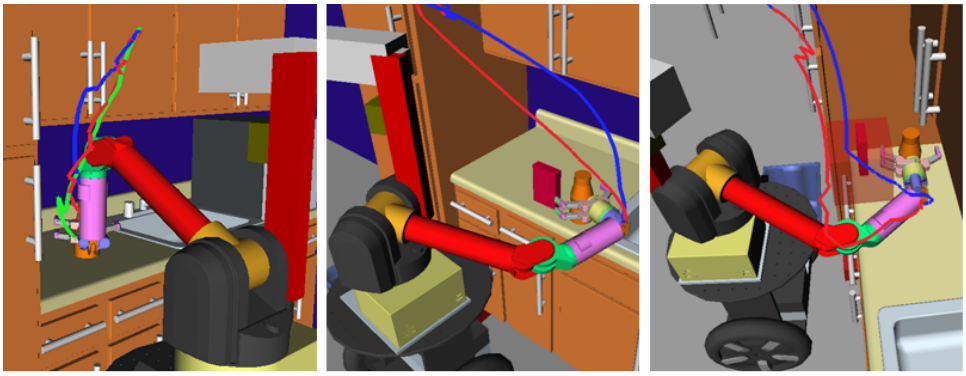


Figure 13.4: Examples of paths planned for the three problems for $nFailMax$ = 30 (only the translation of the end-effector is shown) after 300 iterations of shortcut smoothing. (a) C-space path imitation, (b) Task Space constraint, (c) Uncertain workspace occupancy. Blue: T-RRT path. Red: GradienT-RRT path. The green path in (a) is the path learned from demonstration.

13.5.1 Path Imitation using a C-Space Constraint

For many problems involving human-robot interaction, it is desirable for the paths generated by the robot to conform to some legibility or aesthetic criteria [114] that can be demonstrated via example paths. To incorporate these considerations into our planner we build on a learning-from-demonstration strategy: We first train a learning algorithm on a set of example paths and then construct a C-space cost function around the optimal path produced by the learning algorithm. Configurations closer to the learned path receive a lower cost than configurations farther from it. This construction biases the algorithm to search around the learned path first while also allowing it to explore regions of the C-space farther from the learned path as time goes on. The advantage of this approach is that we can bias the search toward the learned path while taking into account feasibility constraints which are not accounted for by the learning algorithm, such as collision.

In this example problem, the robot's task is to reach for a bottle on a kitchen counter (Figure 13.4a). Five example paths have been provided to the robot for reaching for the same bottle in different locations on the counter (Figure 13.5a). All of the example paths have the robot's elbow pointing up throughout the path (as opposed to pointing down or to the side). We use the GMM-GMR toolbox [115] to compute a set of Gaussians that describe the paths and to obtain the learned path from the Gaussians. The learned path also has associated variances in each dimension of C-space, expressing the confidence of each predicted point in each dimension (Figure 13.5b).

We use the cost function described in Section 13.4.3 to transform the learned path into a cost function. The set U consists of the points in the path (sampled at a fixed resolution), and the Σ values are the variances of the prediction. The costs of all points in U are set to 0. The resulting cost function is shaped like a chasm in C-space with the learned path at the bottom (see Figure 13.5c).

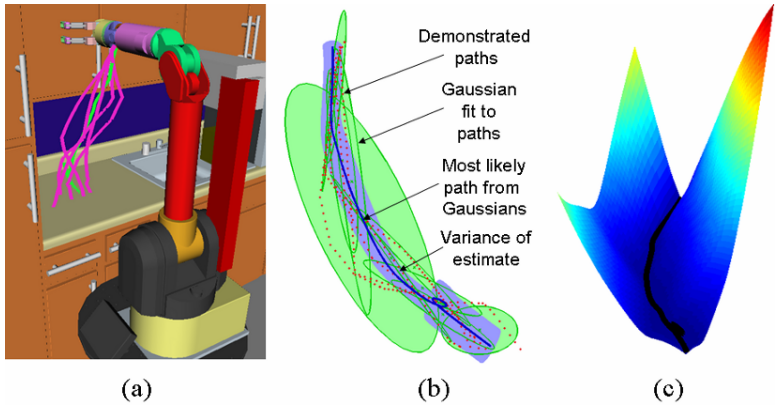


Figure 13.5: (a) Example C-space paths (purple) and the learned path (green), only the translation of the end-effector is shown. (b) Process of learning the path using the GMM-GMR toolbox; the path is shown for joints five and six of the robot. (c) The learned path transformed into a cost function using Equation 13.7.

13.5.2 Task Space Constraints

Many common tasks in manipulation planning also involve constraints on end-effector pose. While we have treated these as hard constraints in previous sections, there are also situations where they can be seen as soft constraints. Consider a reaching task where the pose of the end-effector is not constrained. We have observed that paths generated by RRTs for such problems involve significant (and often needless) rotation of the wrist. The task is accomplished but users find that the rotation of the wrist is unpredictable so they feel less comfortable around the robot.

To address such a constraint on end-effector pose, we employ TSRs or TSR Chains as soft constraints by using the task space distance to the TSR as $G(q)$. The search is thus biased to generate nodes on or near the constraint manifold defined by the TSR while still allowing the algorithm to explore beyond this manifold as time goes on.

In this example problem the task is again to reach for a bottle on the counter (Figure 13.4b), however we assign a TSR to bias the robot not to tilt its end-effector throughout the path. Since we are restricting two DOF of rotation, this constraint induces a lower-dimensional zero-cost manifold in the C-space; an infinitely thin chasm with cost increasing in all directions around it.

13.5.3 Uncertain Workspace Occupancy

Another important use for soft constraints is in planning paths in uncertain environments, where we can not be sure if some areas of the environment are empty or filled because of incomplete sensor data. In such environments we want to encroach as little as possible on uncertain areas. While it may be

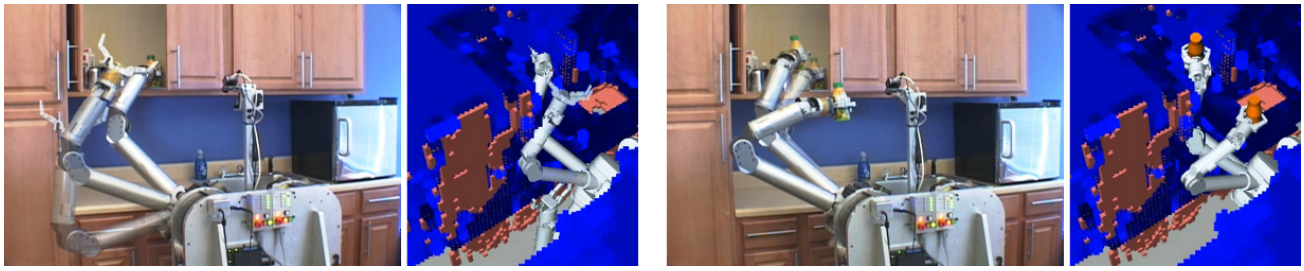


Figure 13.6: Time-lapse images from the execution of two paths planned by GradienT-RRT with uncertain workspace occupancy. The blue areas in the robot's world model (right) represent the probability of occupancy of a cell, with light blue being less occupied and dark blue being more occupied.

tempting to treat uncertain areas as obstacles, this can lead to infeasible planning queries because, for instance, the goal configuration may intersect an uncertain area.

Our approach to this problem is to create a voxel grid over the workspace and label each voxel with its probability of occupancy. We compute the cost of a configuration of the robot from a set of points sampled within its geometry using the methods of Section 13.4.1. The planner is thus biased to stay away from higher-occupancy regions in the workspace during the path and to escape from these regions without increasing cost if the start or goal lies within one of them. However, given more time the algorithm will begin to explore the higher-occupancy regions as well. Note that we still impose hard collision constraints for known obstacles in addition to this constraint.

In this example problem, the task is to reach for a bottle in a cluttered space (Figure 13.4c). An uncertain region has been placed in front of the robot. It is possible for the robot to avoid the uncertain region completely by reaching around it, but this induces a narrow passage in workspace between the uncertain region and the boundary of the robot's reachability. Thus the constraint is a one-sided chasm, with the workspace closer to the base of the robot being higher cost and the workspace farther from the robot being unreachable.

We also implemented a similar scenario on the HERB robot, where the uncertainty in the environment is derived from laser-scan data (Figure 13.6). Here the bottle is in a cabinet which is occluded from the laser scanner by the cabinet door. Note that the bottle is engulfed by the uncertainty created by this occlusion so the robot cannot simply avoid the uncertain region when reaching to grasp. Several snapshots from the execution of paths planned by GradienT-RRT for this example are shown in Figure 13.6.

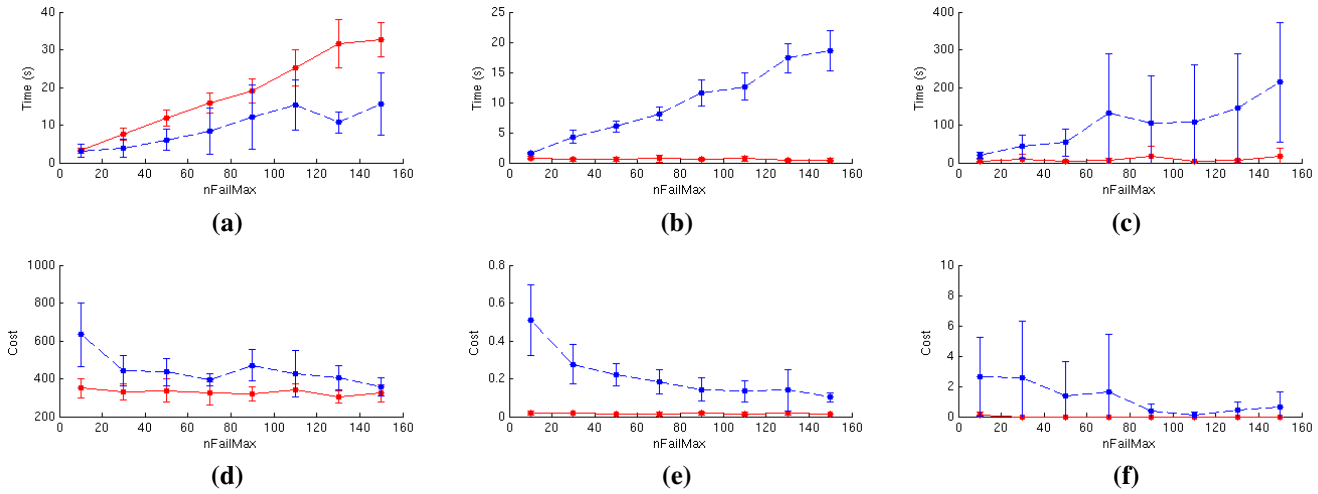


Figure 13.7: (a-c) Average planning times for the C-space, task space, and workspace examples, respectively. (d-f) Corresponding average cost integral of paths after 300 iterations of shortcut smoothing. Blue Dashed: T-RRT, Red: GradienT-RRT. The error bars are standard deviations computed over 10 runs for each value of $nFailMax$.

13.5.4 Discussion of Results

Comparisons of run time and path integral cost are shown in Figure 13.7. In all three examples, the costs of the paths produced by GradienT-RRT were lower on average than those produced by the T-RRT for all values of $nFailMax$. The computation times of GradienT-RRT were much lower than those for T-RRT for the task space and uncertainty examples. The path imitation example's computation time is higher (Figure 13.7a) because the learned path collides with the upper cabinet and the bottle; i.e. the chasm intersects an obstacle (as in Figure 13.2b). As a result the GradienT-RRT attempts to generate more nodes in the chasm, which slows down the algorithm at higher values of $nFailMax$.

What is most encouraging about these results is that the cost of the path produced by GradienT-RRT only decreases slightly when increasing $nFailMax$. The GradienT-RRT is far less sensitive to the $nFailMax$ parameter because it does not rely on sampling to traverse a chasm; it uses the gradient to generate nodes in low cost regions. Thus we can set $nFailMax$ to be relatively low (between 30 and 50) and still achieve roughly the same performance as setting it higher. We believe that this is a significant improvement over T-RRT, where increasing $nFailMax$ is the only way to compute better paths but doing so produces an increase in computation time.

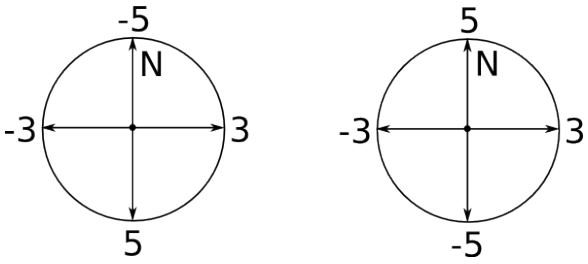


Figure 13.8: An example of two soft constraints having opposing gradients.

13.6 Summary and Discussion

We have presented the GradienT-RRT algorithm as a method to navigate chasms in cost space. The algorithm is a combination of the T-RRT and local gradient descent. GradienT-RRT can traverse cost-space chasms more quickly and produce lower-cost paths than T-RRT because it takes advantage of both gradient-descent and sampling methods to find lower-cost configurations. We have also described general cost functions in workspace, task space, and C-space that are useful for manipulation planning and that naturally induce such cost-space chasms. When we compared GradienT-RRT and T-RRT on our example problems, we found that the GradienT-RRT outperformed T-RRT in terms of the cost of the final path for all examples and in terms of computation time for two of the three examples. We also found that GradienT-RRT was not very sensitive to the $nFailMax$ parameter, making it easier to tune than T-RRT. Finally we showed GradienT-RRT running on the physical HERB robot, where it planned to retrieve an object while considering uncertain workspace occupancy.

We are excited by the potential of GradienT-RRT to solve planning problems in high-dimensional cost-spaces because it takes advantage of both gradient and sampling-based methods. There are potential applications for this algorithm in bioinfromatics where low-energy configurations of molecules are difficult to find through random-sampling alone [77].

In future work we would like to explore planning with multiple soft constraints that produce conflicting gradients at many configurations. Specifically, we are interested in ways to optimize multiple cost functions without mixing them into a single cost using a weighted sum (a popular method to combine criteria). The drawback of mixing criteria in this way is that gradient vectors for different criteria can oppose one another. In such cases, adding gradient vectors together into a single direction vector produces undesirable results. Figure 13.8 shows a simple example. The two compasses in this figure represent two optimality criteria. The change in cost when moving in each of the cardinal directions is shown for each direction. The gradient for the criterion on the left points south, while the gradient for the criterion on the right points north. Adding these gradient directions with an equal weight will produce a zero-length resultant vector. If the weights are not equal but similar, either north or south will be chosen but there will be a significant penalty from the constraint that is weighted lower. Clearly we can attain a better result by choosing west when the weights are roughly equal. Computing the gradients of the individual criteria and adding them does not provide this result.

One alternative to computing the gradient of each constraint and then combining them in a weighted sum is to test the value of the combined cost function for some discretization of all possible direction vectors. While this may be feasible in low-dimensional spaces, it is highly impractical for the high-dimensional problems we consider. Finding the lowest-cost direction to move from a certain configuration in cost-space quickly is an interesting prospect for future work.

Another issue to address in GradienT-RRT is the generation of goal configurations during the planning process. This is important because, as we have shown in previous sections, there is often an infinite set of goal configurations in many types of manipulation problems. Empirically, we have found that the cost of a path often greatly depends on the cost of the goal, especially in the uncertain workspace occupancy example above.

However, goal sampling for soft constraints is not as straightforward as for hard constraints. The main difficulty is that goals that are added earlier in the search tend to grow larger trees, even though the goals themselves are not low-cost. This is because GradienT-RRT and T-RRT seek to minimize the *difference* in cost along the path, not the absolute cost of a configuration. Thus high-cost goals can grow large trees if the algorithm is able to find lower-cost configurations around them. Even if a low-cost goal is eventually added, it often does not have sufficient time to grow a tree before the tree rooted at the older higher-cost goal connects to the start tree. An anytime approach [75] may be beneficial here. Such an approach would allow the higher-cost goal tree to connect and produce a path but would then prune that tree to allow the newer lower-cost tree(s) to connect. This process would iterate until the planning time expires.

Chapter 14

Generating and Selecting from Grasp Sets

Chapter 5 described how to plan paths with TSRs for grasping tasks. In that context, we assumed that the goal TSR for the robot's hand had been constructed such that any hand pose sampled from it would lead to a valid grasp after executing the grasping controller (i.e. closing the fingers). In practice, we constructed those TSRs through manual tuning of TSR parameters and experiments on the physical objects. For simple objects like cylindrical bottles and small boxes the manual approach was quite straightforward. However, creating TSRs for grasping more complex objects, like the pitcher in Section 8.6, is quite tedious.

This chapter and the following chapter describe automatic methods for generating a list of feasible grasps given an object and scene geometry. This chapter presents a method to precompute a set of grasps offline such that an appropriate grasp from this set can be selected quickly online. The following chapter describes an online process for grasp synthesis that takes slightly more time but is able to find grasps in more cluttered environments.

Though these methods are quite effective at generating grasps, treating the set of valid grasps as a list of individual pose/joint-value pairs has several disadvantages compared to the TSR representation, which can specify a volume of grasp poses. The main disadvantage being that we cannot apply the methods of Chapter 10 to compensate for pose uncertainty because there is no volume of intersection between lists of grasps. Ultimately, we would like a fully automatic method for generating TSRs for grasping tasks that takes lists of grasps produced by the methods described in this chapter and the next and generalizes them into a TSR. We describe our work toward such a method in Chapter 16.

This chapter presents an algorithm that generates a set of stable grasps specific to a robot's hand and the object to be manipulated. This generation process is quite time consuming, so it is intended to be done offline. When the robot is tasked with grasping the object online, we show how to quickly

The work discussed in this chapter was published in [116].

choose a grasp from this set to use as a goal for a planning algorithm. This work's contribution to the grasping literature is that it is able to deal with extremely cluttered environments, an area that is often overlooked by other work in this field.

The remainder of the chapter is outlined as follows: In subsection 1 we give an outline of the grasp planning framework. In subsection 2 we describe the steps needed to compute the grasp-scoring function. In subsection 3 we show that our method greatly outperforms the naive approach to grasp selection and show results in simulation and on the HRP2 humanoid.

14.1 Grasp Planning Framework

This framework is similar to many previous approaches in that it uses force-closure [104] to evaluate good grasps when the object is alone in the environment. It relies on a sampling of the grasp-parameter space to find a set of successful grasps for each object [117]. However, once this set of valid grasps is computed for the given object, the following problem occurs: Out of the potentially thousands of precomputed grasps, *which one should be chosen for a given environment?* Many grasps are infeasible due to collisions with the environment, still more can be unreachable because of the kinematics of the robot arm. The naive approach is to keep trying grasps in an arbitrary order and take the first one that is collision-free and reachable. Because of the potentially large number of grasps in a grasp set, this approach is extremely time-consuming. An alternative possibility is to prune the grasp set; however, there is always the risk that pruning can eliminate the only feasible grasp in the current environment. It is also possible to apply manipulation planning techniques from [50, 118, 67] to move obstacles out of the way. But the problem of grasp selection still remains since picking up and moving obstacles implies finding a way to grasp them.

Instead we propose a new framework for grasp selection that combines the force-closure scores of our grasp set with features of the object's local environment and features of the robot kinematics to produce an overall grasp-scoring function. We use this function to evaluate each grasp and sort the grasps according to their scores.

Our method of grasp planning consists of two main phases: a precomputation phase and an online computation phase (see Figure 14.1). The *precomputation* phase uses a geometric model of our hand and the target object along with our grasping controller to build a set of feasible grasps in terms of a grasp stability metric like force-closure. The *online computation* phase computes the score of each grasp for the given environment using the grasp-scoring function. The grasps are ranked in order of the scores assigned by the grasp-scoring function and validated in order of rank using collision checking and IK algorithms.

Since the grasp-scoring function cannot guarantee that a grasp will be successful, grasps must be

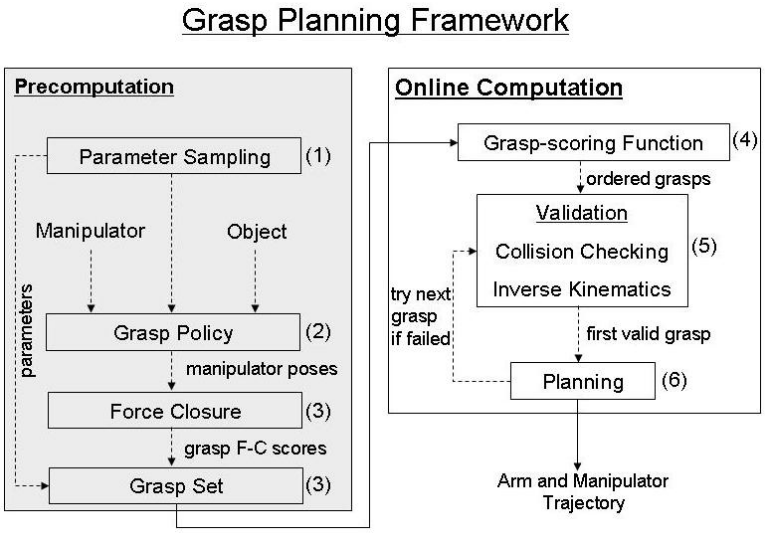


Figure 14.1: The proposed grasp planning framework. The precomputation part need only be computed once for each object-hand pair. Once the grasp set is built by precomputation, the online part of the framework sorts grasps using the grasp-scoring function and plans a trajectory to get to the first valid grasp.

validated individually in the current environment by running the grasping controller. If there is no collision with environment obstacles during the execution of the grasping controller and the grasp is reachable, it is considered valid.

14.1.1 Defining a Grasp

Choosing a parameterization for grasping policies is key to grasp planning. Since our goal is to generalize beyond a given hand, the definition of parameters must be as broad as possible, yet the definition must also include hand-specific information to take advantage of the unique capabilities of a given hand. Thus we define the following general parameters:

- $\mathbf{P}_d$, the direction of approach of the hand (a 3D unit vector)
- $\mathbf{P}_t$, the 3D point on the surface of the object that the hand is approaching
- $\mathbf{P}_r$, the roll of the hand about the approach direction
- $\mathbf{P}_p$, parameter(s) defining the preshape[1] of the hand

[1]The preshape is the initial joint values of the hand.

In addition to the above, other hand-specific parameters can be included in $\mathbf{P}$. Also define O as the object to be grasped and E as the current environment. Note that the above parameter definition is only used inside this algorithm, the grasp that is output by the algorithm is passed to a planner as a 6D hand pose with an associated preshape.

The grasping controller is a function that maps a set of parameters $\mathbf{P}$ to a final grasping pose of the hand. Our grasping controller for obtaining the final pose, joint values, and contact wrenches from $\mathbf{P}$ is shown in Figure 14.2.

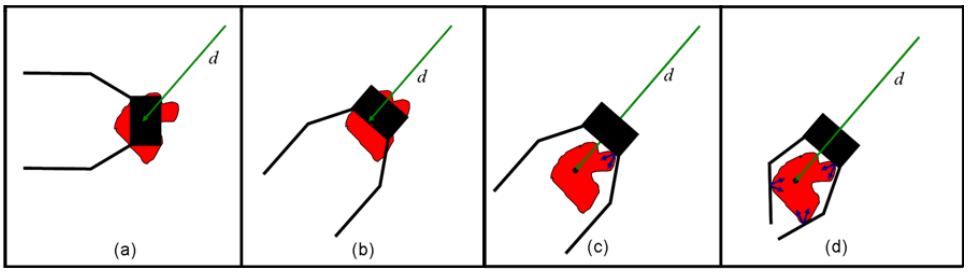


Figure 14.2: The grasping controller that is used to evaluate grasp quality. (a) The hand is placed inside the object at its target grasp point (specified by $\mathbf{P}_t$). (b) The hand is rotated so that its palm normal faces along $\mathbf{P}_d$. (c) The hand is moved back along $-\mathbf{P}_d$ until it is just out of collision. (d) The fingers are wrapped around the object until every finger is in collision or has hit its joint limits. The contact wrenches (blue arrows) are used to compute force closure.

We define a grasp to be *valid* if

1. The hand does not collide with obstacles in the scene during execution of the grasping controller.
2. The ensuing grasp is in force-closure.

14.2 Precomputing a Valid Grasp Set

Given the above grasping controller, we can generate $\mathbf{P}$ values and determine their force-closure quality for a given O. $\mathbf{P}$ values that produce successful grasps are stored along with their grasp quality for online use. Storing grasps saves computation time because we never need to re-evaluate force-closure online, which is a costly process.

However, we must be careful to generate $\mathbf{P}$ values that are likely to yield force-closure for the object, otherwise we will waste time computing grasps that have no chance of achieving force-closure. Thus we sample our grasp-parameter space by first sampling the surface of the object to get $\mathbf{P}_d$ and $\mathbf{P}_t$ and then discretizing the remaining parameters in $\mathbf{P}$. This sampling scheme allows us to concentrate on the promising subsets of pose space when generating grasps.

14.3 Grasp-scoring Function

The goal of the grasp-scoring function is to take into account all relevant information about the environment to score grasps in terms of likelihood of success. It is essential that the grasp-scoring function be computed quickly because it must be evaluated online for the current environment. Since fully simulating a grasp is time consuming, the grasp-scoring function must rely on approximations. Therefore, grasps must be validated in simulation even if they receive a high score.

Let $G(O, E, \mathbf{P})$ be the score of the grasp defined by the parameters $\mathbf{P}$ performed on object O in environment E. Computing $G(O, E, \mathbf{P})$ requires computing the following three criteria:

- $G_q(\mathbf{P})$, the force-closure score of $\mathbf{P}$
- $G_b(E,\mathbf{P})$, the robot-relative position score
- $G_e(O, E,\mathbf{P})$, the environment clearance score

Once the three scores are computed, they are combined into a final score as shown in the equation below.

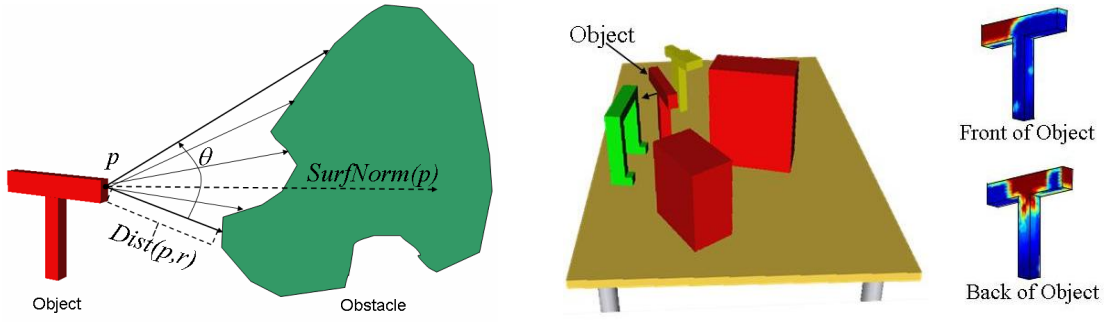
$$G(O, E, \mathbf{P}) = e^{(c_1 G_q)} e^{(c_2 G_b)} e^{(c_3 G_e)} \tag{14.1}$$

where c_1, c_2, and c_3 are coefficients that determine the relative importance of the criteria in computing the overall score. We compute the three criteria as follows:

We set G_q by looking it up in our precomputed grasp set (Section 14.2).

$G_b(E,\mathbf{P})$ takes into consideration the position of the robot in the given environment. Since the robot's base is assumed to be fixed, we use standard IK algorithms to check if a given grasp is feasible. However, testing IK feasibility for all grasps in the grasp set is time-consuming. As a simple approximation, we set G_b to be the cosine of the angle between the hand approach direction and the vector from the robot to the target object. This ensures that grasps where the palm faces away from the robot are preferred over grasps where the palm faces the robot (which are less likely to be reachable).

$G_e(O, E,\mathbf{P})$ allows us to consider the environment around the object when computing the grasp score. In essence, we would like to calculate the clearance along the approach direction $\mathbf{P}_d$ at the point $\mathbf{P}_t$. If that clearance is small, the hand is likely to collide when executing the grasp specified by $\mathbf{P}$. When the clearance is high, there is a greater chance that the grasp will be collision-free. G_e is computed by launching a cone of rays from $\mathbf{P}_t$ oriented along $\mathbf{P}_d$. The shortest collision-free segment of any ray is



(a) **(b)**

Figure 14.3: (a) Computing the minimum distance value for one point in the distance map. The red object is the object we wish to grasp, the green one is an obstacle in the environment. θ is the width of the cone whose tip is at p and whose alignment is the surface normal at p. $Dist(p, r)$ is the minimum distance as evaluate over all rays, r, in the cone. (b) Distance map for the 'T'-shaped object in the scene shown. Distances range from dark red (high clearance) to blue (low clearance). The arrow denotes the front of the object.

the value of G_e (see Figure 14.3a). To save time, we compute the G_e for all points on the surface of the object at once and compile them into a *distance map* (see Figure 14.3b).

14.4 Results

To gauge the usefulness of the grasp-scoring function, we performed a benchmark experiment on two robots in simulation and implemented the algorithm on an HRP2 robot at the Digital Human Research Center.

In order to test our algorithm's effectiveness, we compare it to a more naive approach to grasp selection. In the naive approach, a set of valid grasps is created in exactly the same manner as in our approach. However, the grasps are tested on the object in the given scene in a random order instead of using the grasp-scoring function. The key question is: how many grasps must we test before finding a successful one when using the grasp-scoring function vs. the naive method?

We compare the statistics for 50 randomly generated scenes. When using the naive approach, we re-run the validation 10 times for each scene, each time re-ordering the set randomly. We record the indices of the first successful grasp of each run and compute their mean. This mean is compared to the index of the first successful grasp when using the grasp-scoring function to sort the grasp set. The robots used were an HRP2 robot, with the hand shown in Figure 14.4 and a Puma 560 robot with a Barrett hand. The two robots have very different kinematics and reachability ranges. We ran experiments on these two robots to show that our framework, and especially the grasp-scoring function, is general enough to be applied to any reasonable robot-hand combination.

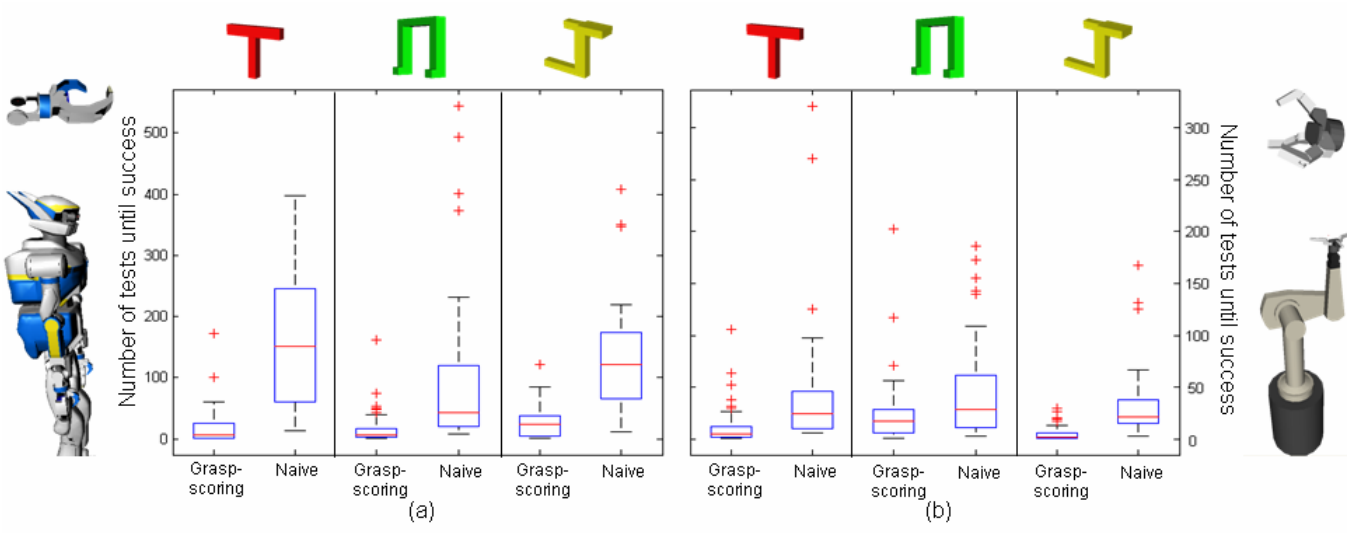


Figure 14.4: Box plot of the statistics for the benchmark experiment on the three objects for the HRP2 and Puma robots. Objects 1,2, and 3 shown at the top (left to right) were placed in 50 random scenes. The vertical axis is the number of grasps tried before a successful one was found. The red line in each box plot represents the number of tests necessary to find a successful grasp averaged over the 50 scenes. Lower is better. (a) Statistics when using the HRP2. (b) Statistics when using the Puma+Barrett hand robot.

For both robots and all objects, the grasp-scoring function clearly outperforms the naive approach to grasp selection (see Figure 14.4). For the HRP2, grasp-scoring outperformed the naive method in 98%, 100%, and 98% of scenes for objects 1, 2, and 3, respectively. For each scene, grasp scoring performed 57.06, 11.51, and 10.59 times better on average than the naive approach for objects 1, 2 and 3, respectively.

For the Puma robot, grasp-scoring outperformed the naive method in 96%, 72%, and 98% of scenes. Grasp scoring performed 6.861, 3.101, and 12.54 times better on average than the naive approach. The results for the Puma robot are less impressive because the Puma has a far larger reachability, thus many more grasps have an IK solution and there is a greater chance that a random grasp is reachable.

To evaluate our method in a real environment with a real robot, we ran several experiments on the HRP2. We used a motion capture system to get the position of the robot and the objects in our scene.

The first task was for HRP2 to reach and grasp objects 1 and 2 and place them into a trash can. The second task was for HRP2 to pick up object 2 and place it upside-down on the other side of the table. Because of the reachability constraints of the robot, this task could not be accomplished using only one arm, so re-grasping the object was necessary. We required all trajectories to be collision-free and required each object to stay within the robot's grasp during the trajectory. Since we compute the grasp-scoring function online, the robot is able to compensate for changing scenes and the addition/removal of obstacles and objects to grasp. See Figure 14.5 for snapshots from the execution of these experiments.

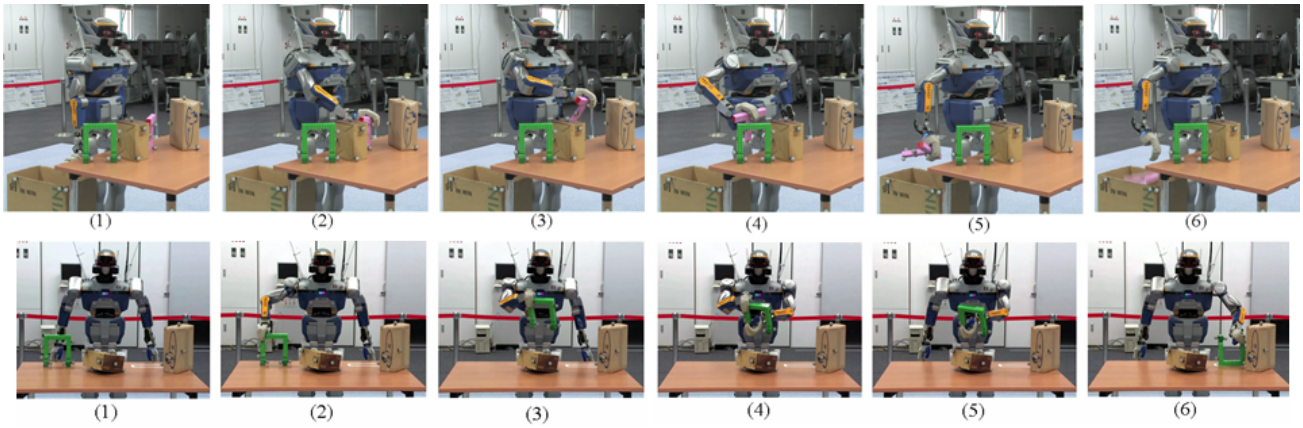


Figure 14.5: Top: The HRP2 reaching, grasping, and dropping object 1 into the trash can. Bottom: The HRP2 picking up, regrasping, and putting object 2 into its goal pose.

14.5 Summary and Discussion

This chapter presented a framework for grasping which takes into account the kinematics of the robot, the environment around the object being grasped, and the grasp's force-closure quality. By precomputing a set of grasps offline and efficiently computing a grasp-scoring function online to rank the grasps, we are able to quickly find stable, collision-free, reachable grasps in cluttered environments. We showed that our framework significantly outperforms a more naive approach and that it can be applied to two different robots with very different hands. We also described an implementation of the framework on the HRP2 humanoid, which allows it to perform complex manipulation tasks in a dynamic environment.

While the framework presented here is effective for many kinds of objects and scenes, it is not applicable when we do not have a model of the target object prior to encountering it. This may be the case when we encounter a new object in a scene and sense its geometry with a laser scanner. Also, this method is limited to the grasps it has pre-computed, it has no way to adapt those grasps to the current environment. As a result, we have encountered scenes where none of the grasps in our pre-computed list are valid (though a valid grasp clearly exists). The following chapter describes an online method for grasp synthesis which addresses these drawbacks.

Chapter 15

Online Grasp Synthesis

In the previous chapter, we approached the problem of grasp selection by generating a large number of stable grasps for an object offline and then ranking those grasps using information about the object's current environment online. The grasps were then tested in order of rank. While successful for many objects and environments, this method proved problematic for certain environment-object combinations because the algorithm is "locked in" to pre-computed grasps; it cannot generate new grasps even if all that is required is to move the wrist slightly to avoid an obstacle.

The limitations of our previous method prompted us to investigate a more robust approach to grasp generation. The framework presented in this chapter uses a preshape along with information about the environment to search for a 6D end-effector pose. It employs an optimizer with a novel cost function to search the space of poses for grasps that are likely to be valid. The search returns a set of grasps that is validated and then passed on to further stages of planning. The goal of this algorithm is to generate a set of grasps that is not only in force-closure for a given object, but also feasible in a given environment and to do so quickly.

15.1 Definitions

We define the configuration of the hand by its *preshape* and its pose. For our algorithm, a preshape is the ideal set of joint values of the hand for grasping a particular object. Pose is described as a 6D vector P, comprising of position $P_p \in \mathbb{R}^3$, and orientation $P_o \in \mathbb{H}$ represented as a quaternion. A *Grasp* consists of two parts: A preshape and a pose. Note that this representation uses fewer parameters

The work discussed in this chapter was published in [119] and [120].

than that of Section 14. We also define a *Directed Point*, which consists of a 3D position(in meters) and a 3D unit vector representing orientation.

We implemented a grasping controller, in simulation and on the physical Barrett hand, which allows the fingers to wrap around objects. The hand starts at a certain set of joint values and each finger is curled in until it collides with any obstacle or reaches a joint limit. If a finger is controlled by more than one joint, the distal joints follow the motion of the proximal joint of the finger. If the proximal link collides with an obstacle, the distal joints continue to curl in. Note that this is similar to the grasping controller of Section 14 except that no movement of the hand is performed.

The act of grasping is completely described by the preshape, the pose of the hand, and the action of the grasping controller.

15.2 Preshapes

Methods for determining a preshape for a given object have been thoroughly studied in robotics and neuroscience literature for many years and are outside the scope of this thesis. Preshapes can be selected from a preshape set based on nearest-neighbor algorithms [24] or through analysis of the affordances of an object [121]. Also, rule-based [122] and heuristic methods [85] can be used to determine a class of grasp to use, which can then be translated into a preshape. In practice, we use the technique of Li et al. [24] combined with a manual selection of preshapes for more difficult objects.

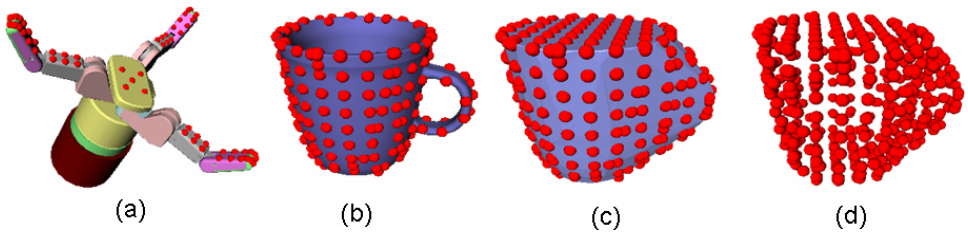


Figure 15.1: (a) Sampling on the contacting surface of the Barrett Hand in a certain preshape. (b) The sampled surface of a mug. (c) The sampled surface of that mug's convex hull. (d) The combined mug samples.

For a given preshape, the contacting surface of the hand is sampled using a set of directed points, see Figure 15.1 for an example of such a sampling. This is necessary because subsequent steps of the framework will rely on these points to quickly match shapes and evaluate potential grasps. We also impose the constraint that the directed points on the fingertips must be in force-closure for the preshape to be admissible. The preshape is meant to be a rough guess of the joint values of the desired grasp based only on the properties of the object and the hand. However, to determine the pose of the grasp, we must take into account the object's local environment.

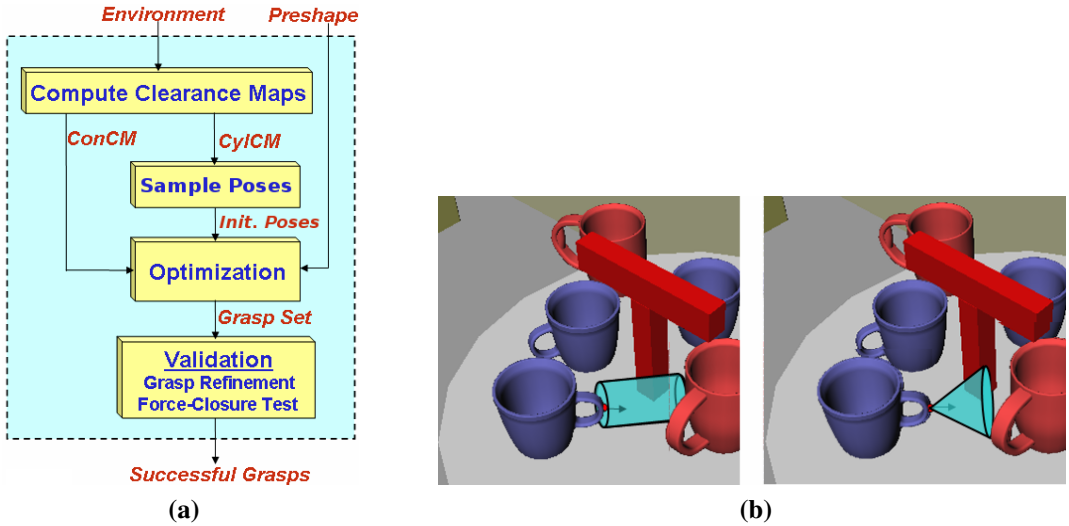


Figure 15.2: (a) Outline of the algorithm for online grasp synthesis. (b) Depiction of computing clearance for one point on an object using cylinders (left) and cones (right). The cone/cylinder is oriented along the outward-facing surface normal of a point on the surface of the object. The length of the longest cone/cylinder that is collision-free is the clearance score assigned to that point.

15.3 Finding a set of Poses

This section describes the computation of a set of valid poses given a preshape and the environment geometry. We define a pose to be valid if it produces a valid grasp. The computation necessary to validate a pose consists of running the grasp controller, collision-checking with the scene at every time step, and evaluating force-closure.

In our experiments with the Barrett Hand, this process took roughly 0.45 seconds for a single grasp using PQP [123], a state of the art collision checker, and MATLAB's `linprog`, a state of the art linear program solver used to evaluate force-closure. Given these run times, only about 2 poses can be validated in one second, necessitating techniques to focus search in the 6 dimensional pose space.

Our optimization algorithm in tended to quickly find poses that are likely to be valid. These poses are then passed on to the expensive validation step. The algorithm has two main parts (see Figure 15.2a). First, we sample promising poses using information about the object's local environment and seed an optimizer with this initial sampling. Second, the optimizer uses a novel cost function which attempts to *predict* the validity of a given pose, thereby bypassing the expensive validation step. The poses produced by the optimizer are then validated.

Generating an Initial Seed

We use the Cylindrical Clearance Map (CylCM) to generate good initial seeds for the optimizer. The CylCM scores the likelihood of the hand being in collision with the scene at a given pose using inexpensive ray-collision checking.

To compute the CylCM, we use pre-computed samples of the surface of the object and its convex hull (Figure 15.1). We denote the set of sample points by O_{dp}. The CylCM at each point in O_{dp} is defined as the length of the longest cylinder that can be placed at the point and oriented along the outward surface normal, without colliding with the scene(Figure 15.2b).

The radius of the cylinder is the radius of the bounding cylinder of the fixed part of the hand. This fixed part, termed so because it is not controlled by any joints in the hand, usually consists of the palm and portions of the wrist. The fixed part is guaranteed to be collision free if the CylCM score is larger than the length of the bounding cylinder. Note that this is a conservative estimate.

Using the above check, we extract those sample points $\boldsymbol{p}$ that are guaranteed to be collision-free for the fixed part. Given a certain number of desired seeds, N, we sample points from $\boldsymbol{p}$ proportional to their CylCM score. Equation 15.1 is used to generate P_p from these points.

$$P_p = \boldsymbol{p} + hl_{\max}\hat{\boldsymbol{n}} \tag{15.1}$$

where h is chosen uniformly from $[0, 1]$, $l_{\max}$ is the length of the fingers when they are fully extended, and $\hat{\boldsymbol{n}}$ is the outward-facing surface normal at $\boldsymbol{p}$. At each $\boldsymbol{p}$, Equation 15.1 produces poses that range uniformly from the hand's palm being in contact with the object ($h = 0$) to being barely able to touch it with the fingertips ($h = 1$).

To generate P_o, we point the hand along $-\hat{\boldsymbol{n}}$ and add a random rotation about $\hat{\boldsymbol{n}}$ to randomize the roll of the hand. The hand's origin is assumed to be the center of the palm and the hand's orientation is the hand's typical direction of approach when grasping.

Cost Function

An optimizer takes as input the initial seed and outputs a set of poses that is likely to be valid. The cost function used by the optimizer must accurately predict hand-scene collisions during the execution of the grasp controller as well as the force-closure of the ensuing grasp. We combine three individual cost functions to produce the cost of a pose for object O in environment E:

1. *Approximate Collision* - $X(P, E)$ - measures the likelihood of the fixed part being in collision.

2. *Fit Cost* - $F(P, O)$ - measures the error of the fit between the preshape and the object. The larger the error, the larger the likelihood that the grasp is not in force-closure.

3. *Contact Safety Cost* - $S(P, O, E)$ - measures the likelihood of collision when the grasp controller curls in the fingers.

The combined cost function $C(P, O, E)$ is given by

$$C(P, O, E) = \frac{F(P, O) + \zeta S(P, O)}{X(P, E)} \tag{15.2}$$

where ζ is the trade-off between fit and contact safety costs.

To compute $X(P, E)$, we place the bounding cylinder for the fixed part of the hand at the candidate pose and check collision with environment obstacles. If the approximating cylinder is not in collision $X(P, E) = 1$, otherwise $X(P, E) = 0$ making $C(P, O, E) = \infty$. We do not check collision with O, allowing the optimizer to utilize the fixed part of the hand for the grasp.

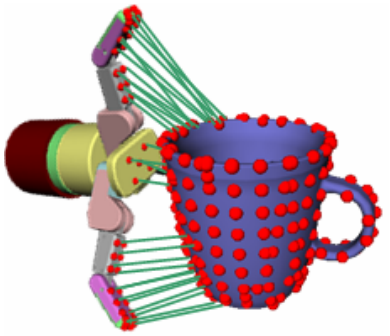


Figure 15.3: Matching preshape points to their nearest neighbors on the object. The green lines represent nearest neighbor pairings and the average length of these lines is $F(P, O)$. Samples on the convex hull of the mug are not shown.

To compute $F(P, O)$, we transform the directed points on the surface of the hand to the pose. We denote the transformed points as P_{dp}. We then perform a nearest-neighbors query to find directed points in O_{dp} that are closest to the directed points in P_{dp}. The distances to the closest points are averaged as $F(P, O)$, which is a measure of how well the preshape fits the object at that pose (see Figure 15.3). If the preshape and pose are not compatible, the grasp controller is unlikely to end up in a configuration similar to the preshape (as it curls in the fingers until all have collided), thus making it hard to predict if the ensuing grasp will be in force-closure.

In our experiments, we found that requiring all points on the hand surface to match to points on the surface of the object produced undesirable results. The hand was unable to grasp objects in many scenes because the entire hand was required to be close to the object, which requires a great deal of clearance around the object. Furthermore, we found that fingertip contact was often sufficient for

force-closure. Thus, we compute $F(P, O)$ for the set of surface points on the distal links of the fingers as well as for the entire set of surface points of the hand and choose the minimum.

To motivate our approximation of $S(P, O, E)$, recall that no part of the hand is allowed to collide with obstacles in the environment during the grasping process. To prevent the fingers from colliding with the object prematurely while approaching it, they must first be spread out from their target preshape. Once the pose is reached, the grasp controller closes the fingers, making contact. If the fingers are to be opened to some degree and the position of the hand in the preshape is to be reached by a planner, it is clear that there must by free space around where the object is to be contacted. Thus it is more likely that the hand will be able to safely contact an object at a point surrounded by free space than it is to contact the object at a point close to other obstacles.

To compute the cost of contacting the object at each of the sample points, we use a procedure similar to that used to compute the CylCM. At each of the sample points of the object, we compute the Conical Clearance Map (ConCM), which uses the same procedure as the CylCM except with cones instead of cylinders. The height of the longest collision-free cone directed along the outward-facing surface normal at a point on the surface of the object becomes that point's ConCM score (see Figure 15.2b). Again, ray-collision checking is used to compute this score efficiently. The angle of the cone is chosen experimentally.

The choice of a cone is motivated by the grasp controller. Imagine fingers curling in toward a particular contact point from many poses. As they curl in, they will arc toward their final destinations and a set of arcs starting above the plane of a point (as defined by the surface normal) and terminating at that point can be enveloped by a cone. The larger the cone, the more arcs are feasible for that contact point and thus the higher the probability that a grasp will be able to contact the object at the given point without colliding with obstacles.

Once the ConCM score is computed for every sample on the surface of the object, the scores are thresholded, giving points lower than the threshold a cost of 1 and points higher than the threshold a cost of 0. We term these costs as *point safety costs*. $S(P, O, E)$ is then the sum of the point safety costs for each point on the object that is closest to a point on the hand.

Optimization

Once we have generated an initial seed of poses, we need to decide how to use it to generate grasps for validation. One approach is to generate a large number of initial samples, determine their cost using the cost function described above, and pass some number of the top poses on to the validation step. However, most of the poses in the initial sampling will not be useful and an optimizer is needed to focus on and explore good regions of pose space.

We use a Genetic Algorithm (GA), which starts with a small initial sample as the seed population and runs until convergence. The top poses of the final population are then passed on to the validation step. The GA searches in pose space using standard crossover and mutation operators to generate new poses around already-discovered promising poses while also exploring the space. The GA terminates when the cost of the best individual does not change significantly for four generations.

To refine the poses further after the GA is finished, a pose is "snapped" into place by aligning the preshape points with their nearest-neighbors on the object. To do this, we use the first technique discussed in [124]. This technique uses Singular Value Decomposition (SVD) to find the least-squares-best transform matrix to align two sets of points.

15.4 Validation

There are two parts to the validation of grasps returned by the optimizer: grasp refinement and force-closure testing.

Because the hand, when placed at the pose of the grasp with the corresponding preshape, may be interpenetrating with the object, we must determine where the fingers would actually collide with the object when running the grasping controller before evaluating force closure and checking collision with environment obstacles. Interpenetration is dangerous, especially at the palm, because it can cause the collision checker to give spurious contacts which will disrupt the force-closure test. The refinement process is described in Figure 15.4.

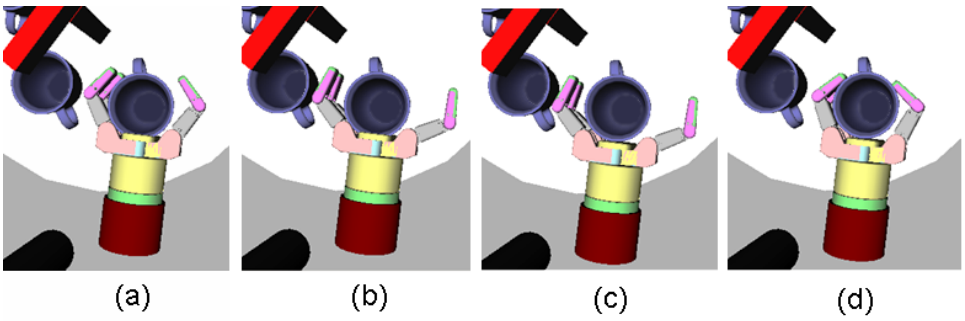


Figure 15.4: Grasp refinement process for an example grasp. (a) An example grasp, as passed to the validation step. Note the interpenetration of the palm. (b) First, the fingers are uncurled until they reach collision or a joint limit and then curled until they are halfway between their starting position and the obstacle with which they collided. (c) If the hand is not in collision with the object at this step, this step is skipped. Otherwise, the hand is moved backward along the line defined by P_o until the hand is no longer in collision with the object and then moved forward slightly so that the hand is barely colliding with the object. This is done mainly to preserve palm contact with minimal interpenetration. (d) The fingers are curled in until each finger collides or joint limit.

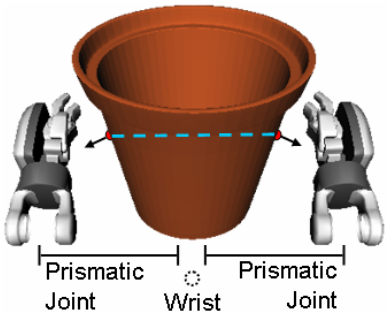


Figure 15.5: Depiction of two-handed grasping extension. A pair of sampled directed points (red) is shown along with the distance between them (blue). Virtual prismatic joints and the virtual wrist are shown for the HRP3 hands.

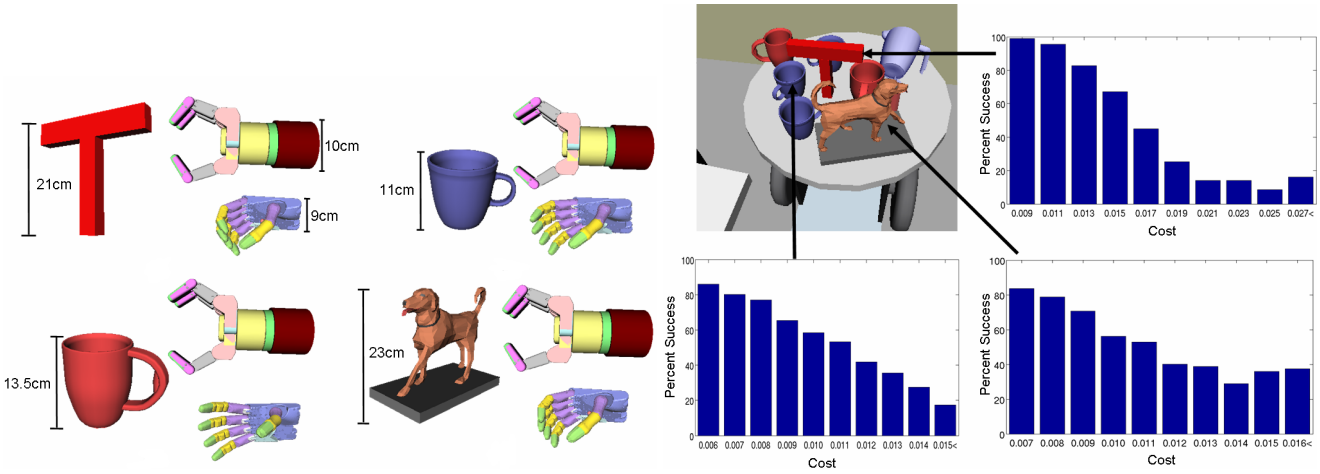
After refinement, the contact points between the hand and the object are extracted and passed to the force-closure test. To test force-closure we use the linear-programming method of Zheng and Qian [125].

15.5 Extension to Two-Handed Grasping

Our algorithm can also be applied to two-handed grasps by treating the two hands as fingers connected by virtual joints to a virtual wrist (see Figure 15.5). By arranging the hands in this way, we turn the problem of two-handed grasping into the problem of grasping with a large gripper that has no wrist, which can be handled by our algorithm. The fingers of both hands are locked into an initial position and preshapes are generated by placing the hands opposite to each other and spreading the hands apart. The grasping controller moves the hands toward each other along the virtual prismatic joints shown in Figure 15.5. Once both hands make contact their fingers are curled in as usual.

15.6 Results

We tested our algorithm on two types of hands: a three-fingered 4DOF Barrett hand and an anthropomorphic 22DOF Shadow hand. Figure 15.6a shows the four test objects and their respective preshapes. The objects chosen are meant to represent various levels and types of difficulties for grasping. Object A (the red T) is larger than both hands and contains large concavities, however its surface geometry is very regular and simple. Object B (the blue mug) has smaller concavities but its geometry is significantly more complicated. Object C (the red mug) is similar to Object B, but it is significantly larger, making it more difficult to grasp in tight spaces. Object D (the dog statue) is the most difficult because it contains sizable concavities and its surface geometry is very erratic.



(a) **(b)**

Figure 15.6: (a) Four objects used in the experiments and the preshapes used for these objects for both the Barrett and Shadow hands. Top Left: Object A. Top Right: Object B. Bottom Left: Object C. Bottom Right: Object D (b) Comparison of scores to percent success for the objects in the scene shown when using the Barrett Hand.

15.6.1 Cost Function Evaluation

To gauge the effectiveness of the contact safety cost portion of the cost function, we compare the contact points of successful grasps to points classified as safe or unsafe by our cost function. The results are displayed in Figure 15.7. This figure shows that successful grasps rarely make contact near points deemed to be unsafe. Thus it is correct to assign higher costs to those points because contacts near them are rarely a part of successful grasps.

We also compare the overall cost assigned by the cost function to the probability of success, see Figure 15.6b. To do this, we generated 20,000 grasps for each of the three objects in the scene shown in Figure 15.6b. Each grasp is validated, and the comparison between scores given by the cost function and percent of successful grasps with roughly that score is shown. The trend in each graph is clear, the lower the cost of a grasp, the more probable it is to succeed, thus the cost corresponds well to probability of success for the objects tested in this scene.

15.6.2 Optimization Comparison

To gauge the effectiveness of our optimization algorithm, we compare our GA to a Random Sampling method. In the Random Sampling method, we sample a much larger number of initial seed poses and rank them by the cost function instead of running the GA. To compare these two methods, we again use

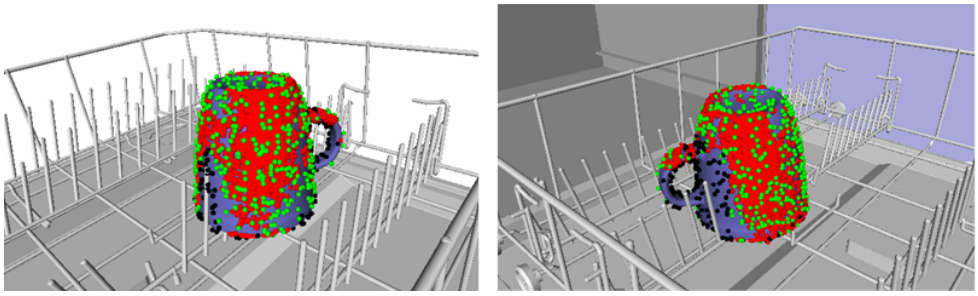


Figure 15.7: Comparison of contact safety costs with contact points from 1000 successful grasps (generated using random sampling). The red points are the contact points of the grasps, the green points are surface samples of the object deemed to be safe, and the black points are surface samples deemed unsafe. Two views of the same example are shown.

the objects and scene shown in Figure 15.6b. The GA method was run 63 times with an initial sampling size of 160 poses. The top 10% of poses in each of the final populations are validated, generating a total of 1008 poses along with their success or failure for each object. The Random Sampling method is run once, with a sampling size of 10,080 poses for each object. The top 10% of poses are validated, generating 1008 poses along with their success or failure for each object. The percent success (number of successful poses/1008) for each object is shown in Table 15.1.

	Object A	Object B	Object C	Object D
Barrett Hand				
Genetic Algorithm	98.8	78.6	72.2	60.1
Random Sampling	73.2	31.9	60.0	39.3
Shadow Hand				
Genetic Algorithm	91.1	94.8	96.9	68.3
Random Sampling	81.3	66.4	81.4	70.8

Table 15.1: Percent Success for GA and Random Sampling

The GA method clearly outperforms Random Sampling because it is far more likely to generate successful grasps, even though it starts with a far smaller initial sampling. This occurs because the GA focuses its search on good solutions, generating new poses near low-cost poses, which are also likely to have a low cost. The GA also explores the space at the same time, so new poses farther from the initial sampling can be found. By contrast, the Random Sampling method is locked in to the initial seed and has no way to focus on good poses.

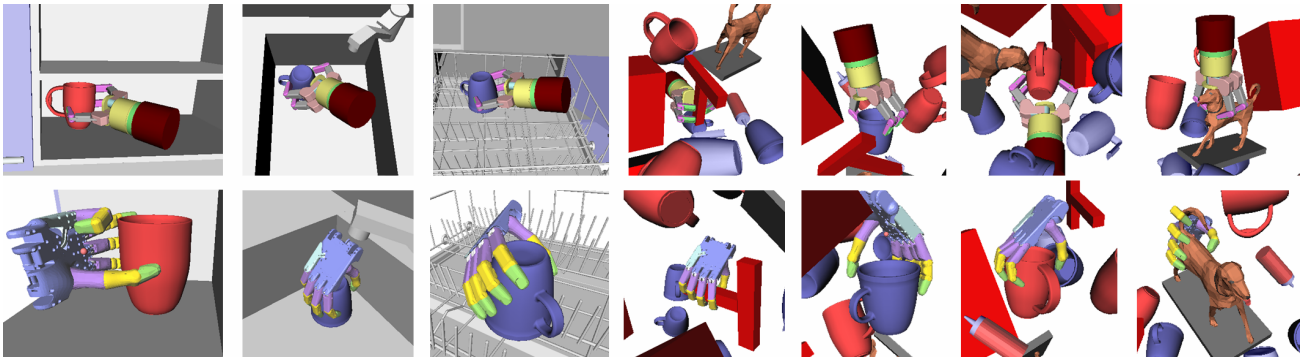


Figure 15.8: The Barrett (top row) and Shadow (bottom row) hands successfully grasping objects in scenes 1, 2, 3, 4A, 4B, 4C, and 4D, respectively from left to right.

15.6.3 Trials in Simulated Scenes

We tested our algorithm on the test objects in several representative scenes. The algorithm was run 30 times in each scene with an initial sampling of 160 poses and the top 10% of poses in each final population were validated. The percent success of the generated grasps is shown in Table 15.2. Examples of successful grasps in the test scenes are shown in Figure 15.8.

	1	2	3	4A	4B	4C	4D
Barrett Hand	96.3	97.5	83.3	84.4	88.7	80.3	54.7
Shadow Hand	82.9	95.0	96.3	91.9	94.0	90.0	43.5

Table 15.2: Percent success in test scenes.

Scenes 4A, 4B, 4C, and 4D were randomly generated by placing the object to be grasped at the origin and dispersing obstacles (2 blue mugs, 2 red mugs, 2 big boxes, a pitcher, a ketchup bottle, and a dog statue) around it. Obstacles were placed around the object at random poses within a cube of $50cm$. No collisions were allowed. For each test object we generated 100 random scenes and ran the algorithm 30 times in each scene. The percent success in Table 15.2 is averaged over the scenes. As expected, object D is the most difficult to grasp with this algorithm, receiving the lowest success rate. Objects A and B turned out to have very similar success rates, illustrating that algorithm compensates for different geometries very well.

15.6.4 Comparison to Previous Work

To benchmark our algorithm, we show the success rates of two previously-proposed algorithms for the Barrett Hand in Table 15.3. The *Primitives* algorithm [85] works by first defining preshapes and

approach directions to grasp primitive objects such as cylinders, boxes, and cones. An approximation of the geometry of the object to be grasped is then generated using these primitive shapes. The grasps corresponding to the approximating primitives are tested in the given scene using a grasp controller and force-closure test that is similar to ours. To generate success rates, we tested all grasps in the grasp set (roughly 300 for each object) in the given scene and recorded the percent that were valid.

The *Grasp List* algorithm is the one proposed in Section 14. This algorithm generates a large list (roughly 600) of force-closure grasps offline for an object to be grasped by sampling the preshape and pose parameters of the hand and running the grasp controller. When the object is placed a new scene, the CylCM is constructed as in our proposed algorithm. The grasps in the table are then sorted by their CylCM scores and the top sixteen[1] are tested for collision. No force-closure test is necessary since all grasps in the table are known to be in force-closure.

Because both of the algorithms we compare to are deterministic they were run once in each test scene including each of the four-hundred randomly-generated scenes. We found that most of the cases where these methods failed were due to wrist collisions or finger collisions when executing the grasping controller.

	1	2	3	4A	4B	4C	4D
Proposed Algorithm	96.3	97.5	83.3	84.4	88.7	80.3	54.7
Primitives	0	0	0.95	12.8	12.2	12.4	10.2
Grasp List	0.68	0	2.33	44.9	51.7	48.3	53.2

Table 15.3: Comparison of Percent Success for Barrett Hand

The algorithm presented in this chapter clearly outperforms the other algorithms in every scene, except for scene 4D where Grasp Lists and the proposed algorithm perform very similarly. This similarity is due to the difficulty of grasping object D, causing both algorithms to struggle.

15.6.5 Run Times

Table 15.4 shows the run times of each component of our algorithm averaged over 30 runs in each of the 100 randomly-generated scenes. These results were obtained on an Intel Dual-Core 2.4GHz PC with 4 GB of RAM.

CylCM and ConCM values are average times needed to compute clearance maps. Cost Fn values are the average sum of all cost function evaluations per run. Total values are the sums of all previous

[1] This is the same number of grasps validated during the validation step of the algorithm proposed in this chapter.

	CylCM	ConCM	Cost Fn	Total	Validation (per grasp)
Barrett Hand					
Object A	0.164	1.15	0.994	2.31	0.42
Object B	0.475	1.90	1.01	3.38	0.48
Object C	0.552	2.79	1.13	4.47	0.50
Object D	0.413	3.84	1.31	5.57	0.42
Shadow Hand					
Object A	0.169	1.21	1.28	2.67	0.78
Object B	0.475	1.97	1.47	3.91	1.20
Object C	0.576	2.88	1.52	4.98	1.22
Object D	0.428	4.03	1.72	6.18	0.88

Table 15.4: Average run times for each part of the algorithm

values. Validation times for a single grasp averaged over all runs for each object in each scene are also shown.

Note that the time needed to construct the CylCM and the ConCM varies with the surface area of the object because points are sampled on the surface at a fixed resolution, thus larger objects will have more surface points. The cost evaluation times also increase with the number of surface points because the Nearest-Neighbor query needs to evaluated more points, but this increase is moderate.

15.6.6 Two-Handed Grasping

We also ran the two-handed extension of our algorithm for the left and right hands of the HRP3. Each hand has 6 degrees of freedom in the fingers. The algorithm was run 30 times in the scenes shown in Figure 15.9. The success rates were 94.8% and 95.0% on average for scenes 5 and 6, respectively. Since the objects were quite large, computing the CylCM and ConCM took an average of 22.5 seconds. This run time can be improved by using a lower-resolution sampling of the object, however doing so may cause greater inaccuracy in fitting preshapes.

15.6.7 Experiments on Robot

We implemented our algorithm on a robot consisting of a 7DOF WAM arm and a Barrett Hand. The task for the robot was to pick up two different kinds of mugs arranged arbitrarily on a table. The

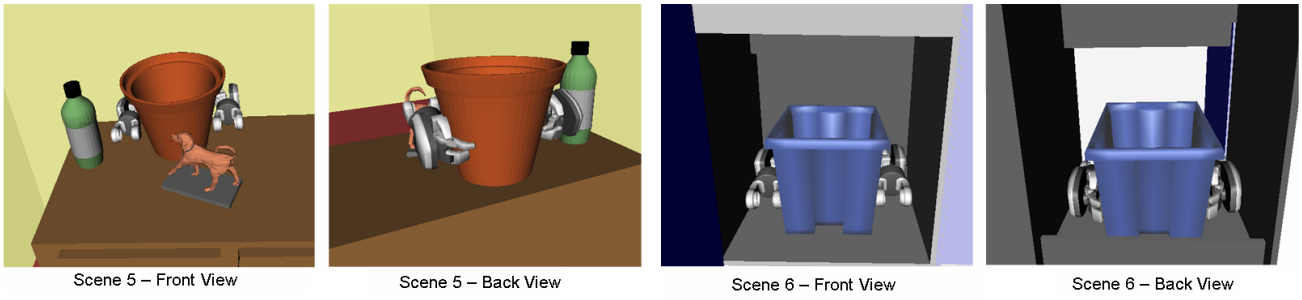


Figure 15.9: Two test scenes used to evaluate the two-handed grasping extension. Two grasps generated by our algorithm are shown.

Figure 15.10: The robot grasping blue and red mugs in various scenes. In the three right-most pictures an artificial obstacle was added above the mugs to make the problem more difficult.

system uses an overhead camera to identify the mugs and obtain their poses. The grasp set generated by our algorithm is passed to a planner that uses inverse kinematics and a BiDirectional RRT to plan an arm trajectory to the pose.

Once the arm is in position, the fingers are curled in, squeezing the mug. The mug is then lifted up by 3cm. Snapshots of objects B and C being lifted in several scenes are shown in Figure 15.10.

To demonstrate that our algorithm can work in more confined spaces than our vision system can handle, we placed an artificial obstacle above the mug to prevent it from being grasped from the top in the three right-most scenes in Figure 15.10. No artificial obstacles were used in the two left-most scenes.

15.7 Summary and Discussion

This chapter presented an efficient and general algorithm for online grasp synthesis in cluttered environments. We have demonstrated the ability of this algorithm to consistently generate force-closure grasps for a wide range of objects and scenes in a few seconds and shown that it outperforms other approaches. We have also demonstrated the generality of the algorithm by evaluating it across several manipulators of varying complexity and structure and we have described an implementation using the

WAM and Barrett hand, as well as extending the algorithm to two-handed grasping tasks.

Ideally, the preshape used in a given scene should not be decided only by the geometry of the object but also by the properties of that scene. Currently our algorithm is limited to considering only one preshape at a time, so the optimizer has no choice in preshapes. However, multiple instances of the algorithm can be run in parallel, with each instance using a different preshape. Alternatively, preshapes could be ranked for an object and the algorithm could run for each preshape in order of rank, terminating when enough valid grasps are found.

Chapter 16

Toward the Automatic Generation of TSRs for Grasping

Sections 14 and 15 described automatic methods for generating a list of feasible grasps given an object and scene geometry. These methods are quite effective at generating grasps in cluttered scenes but treating the set of valid grasps as a list of individual pose/joint-value pairs has three main disadvantages. First, given a pose from the list, there is little chance that the hand can be placed at exactly this pose relative to the object in the real world because of execution error and uncertainty about the object's pose. When the set of grasps is a group of separate points we cannot use the methods of Chapter 10 to compensate for uncertainty because there is no volume of intersection. Second, there are some situations where certain grasp types are more desirable or more likely to be feasible than others, yet a list affords no method of correlating grasps so each grasp is its own type. Third, a list of grasps is incomplete since it is only a sampling of possible grasps, it would be more desirable to have a continuous representation so that we have more freedom in picking grasps to satisfy other constraints, such as reachability and collision avoidance.

Ultimately, we would like a fully automatic method for generating TSRs for grasping tasks. However automatically constructing TSRs in full pose space has proven quite difficult because the space of 6D TSRs is too large to explore within reasonable time bounds. We are currently able to construct two types of TSRs: *cylindrical* and *translational*. Cylindrical TSRs describe 2-dimensional pose regions: the pose is allowed to vary in rotation about a fixed axis and in translation along the same axis. Such a TSR can capture many grasp types, for instance grasping a mug from the side or from the top. Translational TSRs describe 3-dimensional pose regions: the pose is allowed to vary in all three translation dimensions but is not allowed to rotate. Such a TSR can capture grasps for boxes and other objects with sharp angles. Though these two types of TSRs only describe 2 or 3 dimensional volumes in the pose space, they are still quite useful for capturing common grasp types.

Given the models of the hand and object and the joint values of a preshape, we aim to construct a set of TSRs that captures all the ways this preshape can be used to grasp the object. To do this we generate a set of grasps that are valid for the object and use those grasps to guide the construction of the two types of TSRs. The construction process examines the relationship between pairs of grasps by finding the axis of rotation or the translation between them. These relations are then clustered and the clusters are converted to TSRs. Once the TSRs are formed, they are iteratively refined by re-assigning grasps to various clusters and recomputing the corresponding TSRs. Once the refinement is complete, the bounds of the TSRs are computed by discretizing between the extremal grasps that were assigned to that TSR's cluster.

Though this process is able to generate a valid set of TSRs, this work is still preliminary. The main issue holding back the automatic generation of TSRs is the lack of an adequate quality metric. We discuss these issues and prospects for enhancing our method in Section 16.4.

16.1 Generating Example Grasps

In order to generate TSRs, we must first start with example grasps. These example grasps will be used to guide the construction of TSRs and determine parameters in the TSRs' definition. It is important to generate an adequate set of example grasps that includes all grasp types the user is interested in. This can be done by hand; i.e. manually positioning the hand relative to the object and recording the pose. However, such an approach is tedious and runs the risk of excluding grasp types which are not obvious to the user.

Alternatively, we can employ the method of generating grasp lists presented in Chapter 14. This method samples the surface of the object to obtain a set of surface normals. The hand then approaches the object with the palm facing along each surface normal and executes a grasp at different distances from the surface. The rotation of the hand about the surface normal is also sampled. All successful grasps are recorded and stored for later use.

Although this technique by no means exhausts the space of possible grasps, we have found that it does produce many grasps of differing types, thus it is appropriate for our application.

16.2 Automatic Construction of TSRs

Though we focus on two distinct types of TSRs, the construction of them follows a common outline:

1. Compute relations between all pairs of example grasps

2. Compute clusters of similar relations for each grasp
3. Convert clusters to TSRs
4. Refine the TSRs through grasp re-assignment
5. Compute the bounds of the TSRs

The implementation of each of these steps depends on the TSR type being computed. The remainder of this section describes how to perform these steps for translational and cylindrical TSRs.

Translational TSRs

Translational TSRs describe 3-dimensional pose regions where the pose is allowed to vary in all three translation dimensions but is not allowed to rotate. They are constructed as follows.

1. Compute Relations

To compute translational TSRs from example grasps, we start by computing the relative transform between every pair of grasps. If the relative transform involves a significant rotation (greater than 0.1rad), we discard it. This leaves us with relative transforms between grasps that are close to pure translations.

2. Compute Clusters of Relations

We then form a cluster for each grasp that contains all grasps which are close to pure translations relative to that grasp.

3. Convert clusters to TSRs

To convert a cluster of grasp poses to a translational TSR, we first compute the mean of the translation components of the grasp poses in that cluster and change their coordinates to be relative to the mean. We then compute the SVD of the translation components, which gives us the eigenvectors of the set of translations. These eigenvectors are converted to the axes of a coordinate system at the mean (the axes describe the principle deviation in the points). This coordinate system is set as $\mathbf{T}_w^0$. The translation of $\mathbf{T}_e^w$ is set to the translation of $\mathbf{T}_w^0$. The rotation of $\mathbf{T}_e^w$ is set to the rotation of the pose that minimizes the average distance to all other poses in the cluster (to account for the small deviation in rotation we allow). We take a guess at the bounds of the TSR by setting them to the minimum and maximum deviation from the center in every translation dimension (the rotation bounds are set to 0). These bounds are computed more accurately in step 5.

4. Refine TSRs through Grasp Re-assignment

We now have a set of TSRs and a cluster of grasps associated with each TSR in the set. However, at this stage, there are usually several TSRs with very low support, i.e. the TSRs correspond to a happy accident in sampling or geometry which causes a few grasps to align but does not correspond to a desirable grasp type. One approach to removing such TSRs is to throw away TSRs whose associated grasps are fewer than some threshold. However, this approach has the downside of eliminating rare grasps simply because they are rare. The grasps could represent a unique grasp type that was not captured well by the examples.

Alternatively, we can use a threshold on distance to filter out unlikely TSRs. The motivation is that grasps within a distance threshold of a given TSR can be assigned to this TSR, but they can also be assigned to any other TSR if they are within that TSR's distance threshold. Thus we compute the distance from each grasp to each TSR (using the distance metric of Section 5.2). We then threshold the distance to only consider the viable TSRs for each grasp. Every grasp then picks among its viable TSRs on the basis of popularity, i.e. a grasp picks the viable TSR to which the most other grasps are closest. If a TSR is assigned less than three grasps, it is deleted. After grasp re-assignment, the TSRs are recomputed to take into account their current associated grasps. The process of re-assigning and recomputing TSRs iterates until the number of TSRs converges.

5. Compute Bounds of TSRs

We now have a good hypothesis of the centers and end-effector offsets of our TSRs but we still need to determine the bounds accurately so that every grasp within the bounds is likely to be valid. To do this, we discretize each of the translation dimensions within the approximate bounds and test the feasibility of grasps at every point in the discretization. This testing yields a three-dimensional array of binary values, where ones represent successful grasps and zeros represent failed grasps. We search this array for the biggest box that contains only successful grasps (i.e. the box that contains the most ones and no zeros) and take this box as the bounds of the TSR. Sometimes the array may contain several large boxes, so after selecting the biggest box, we invert those values and search for the biggest box again. This process is repeated until the biggest box no longer contains more than some threshold of ones. Each box is assigned its own TSR. This process is meant to capture as much volume as possible but sometimes results in too many boxes because a grasp in the middle turns out to be infeasible. This issue is discussed further in Sections 16.3 and 16.4.

Cylindrical TSRs

Cylindrical TSRs describe 2-dimensional pose regions: the pose is allowed to vary in rotation about a fixed axis and in translation along the same axis. We follow a procedure similar to the one above to compute cylindrical TSRs.

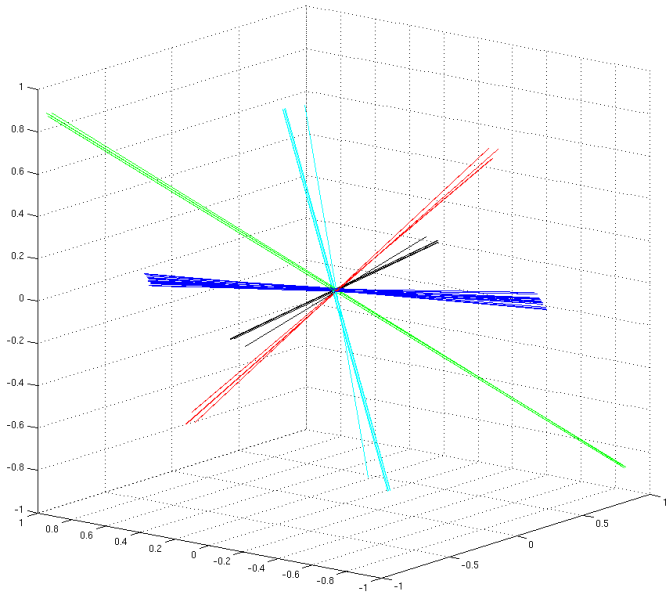


Figure 16.1: The lines represent rotation axes between a single reference grasp pose and many other grasp poses. The rotation axes are clustered using a mean-shift algorithm and each cluster is denoted by a different color.

1. Compute Relations

We start by computing the relative transform between all pairs of TSRs and converting these transforms into twists. Twists give us a line of rotation that relates the two grasps. If the magnitude of this rotation is small (i.e. the transform is close to a pure translation) we discard it.

2. Compute Clusters of Relations

We then form clusters of grasps using these twists. For each grasp, we extract all twists involving that grasp and input them into a mean-shift clustering algorithm. This algorithm finds the biggest cluster of similar twists (see Figure 16.1). The clustering algorithm requires two components to perform its task: 1) A distance metric between twists, and 2) A function to compute the mean twist from a set of twists. Since we are not concerned with the *magnitude* of the rotation between grasps in a cluster, we cannot simply use a weighted Euclidean metric on quaternions or similar methods as the distance metric. Instead, we treat every twist as a line segment (determined by intersecting the line of the twist with a large bounding box). We then define a straightforward distance metric between line segments: a weighted sum of the length of the line of closest approach between two segments and the angle between the segments.

We compute the mean line segment in two steps. First we compute the center of the closest line of approach between all pairs of segments. We take the mean of these center points as one point on our mean line segment. To obtain the other point, we extract the directions of each segment as a pair of

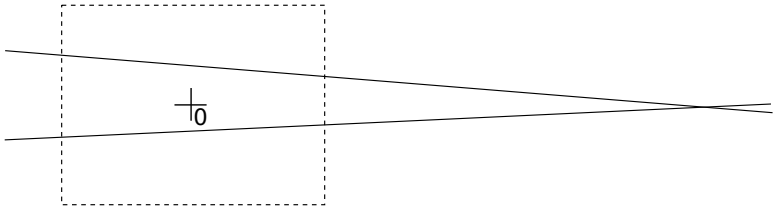


Figure 16.2: The two lines are clearly quite different and would not be grouped into the same cluster when bounding the lines with the dashed bounding box. If we consider infinite lines, however, the lines would be grouped together because they eventually intersect and the angle between them is not large.

opposing unit vectors and find the least-squares line that fits the collection of the vector's endpoints best (using SVD). A point on this least-squares line becomes the second point on our mean line segment. We compute the line that contains these two points and intersect it with the bounding box to obtain the mean line segment.

We emphasize that it is important to consider line segments, not infinite lines (although infinite lines correspond more closely to twists). This is because it is difficult to define a distance metric between infinite lines that yields desirable results. For instance, consider a pair of lines that are nearly parallel but separated by a large distance near the origin, as in Figure 16.2. Clearly these lines should belong to separate clusters, yet we can construct these lines such that the line of closest approach between them is quite far from the origin, and the length of this line is small. Thus these two lines would actually be grouped into a single cluster, which is undesirable. Restricting twists to be segments of finite length produces much better results in terms of clustering grasps into TSRs.

3. Convert clusters to TSRs

After running the mean-shift algorithm, we extract the biggest cluster for each grasp along with the mean line segment for each cluster. We then convert these clusters to cylindrical TSRs as follows. Unlike with translational TSRs, we determine $\mathbf{T}_w^0$ and $\mathbf{T}_e^w$ simultaneously. To do this, we iterate through the grasp transforms in the cluster and compute candidate TSRs using each transform as $\mathbf{T}_e^w$. For a given transform a TSR is computed by setting the translation of $\mathbf{T}_w^0$ to be at the center of the mean line segment with the z axis pointed along the mean line segment. The x axis points toward the the palm of the hand. $\mathbf{T}_e^w$ is then set as the given transform (but converted to the coordinates of $\mathbf{T}_w^0$). The bounds of the TSR are set to allow infinite translation and rotation about the z axis and all other bounds are set to 0. We then compute the mean distance from each candidate TSR to all the transforms in this cluster and take the TSR with the minimum distance. Note that we have yet to determine the correct bounds for this TSR; this will be done later.

4. Refine TSRs through Grasp Re-assignment

We now have a set of TSRs, each with a set of associated grasps. Because we created a cluster for each example grasp, many of the clusters will be redundant, so we now merge similar TSRs. We merge

two TSRs if 1) The distance between the z axes of the TSRs is small (using the line segment distance metric) and 2) The distance between the $\mathbf{T}_e^w$ of one TSR and the other TSR is also small (using the distance metric of Section 5.2). The second check is necessary to ensure that TSRs with similar axes but different end-effector offsets are not merged. After merging, the TSRs are recomputed taking into account their new members. We take a guess at the bounds of the TSR by setting them to the minimum and maximum deviation from the center in the two degrees of freedom.

As with translational TSRs, there are too many small TSRs at this point in the process. We perform the same grasp re-assignment process as used for translational TSRs to refine the TSRs and iterate until the number of TSRs converges.

5. Compute Bounds of TSRs

Finally, we compute the bounds of the TSRs by discretizing their two freedoms and testing each discrete grasp. We apply the same bounding-box-of-ones method as used for the translational TSRs to extract bounds which contain only successful grasps (up to the testing discretization). As with the translational case, a single TSR may be divided into multiple TSRs depending on which tested grasps fail.

16.3 Results

We conducted two experiments to evaluate our approach. First, we checked the fidelity of our method by recovering a TSR from poses sampled from that TSR. Second, we evaluate our approach for generating cylindrical and translational TSRs for two example objects.

TSR Recovery

An important aspect of our approach is that the method can recover an input TSR from an adequate sampling of poses from that TSR. However, since we do not have an adequate quality metric, we have no way to verify that the poses from the input TSR are stable and subsequently no way to determine if the grasps from our recovered TSR are stable. We circumvent this problem by examining how well the method reconstructs the original TSR. Thus we do not perform step 5 of the algorithm (because this requires a quality metric), we simply use the bounds of the TSR that were guessed in earlier steps. We then examine the maximum error between the original TSR and a set of samples from the recovered TSR as well as comparing the original and recovered bounds.

We generated 50 sample poses from a cylindrical TSR for a bottle (Figure 16.3). We then input these poses into our algorithm for finding cylindrical TSRs. We did not give the algorithm any information

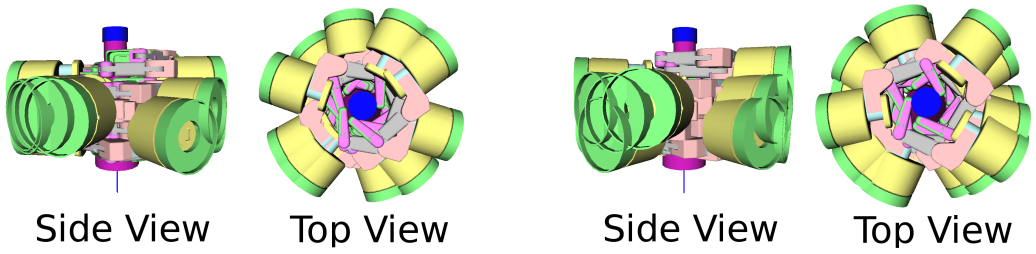


Figure 16.3: Left: Two views of the original TSR. Right: Two views of the recovered TSR. The blue line represents the axis of rotation of the cylindrical TSR. Twenty grasps sampled from each TSR are shown.

about the number of TSRs to generate or any hypothesis TSRs. Running steps 1-4 of the algorithm produced a single output TSR. We then sampled 100 poses from this TSR and evaluated their distance to the original TSR. The maximum distance to the original TSR from any sample pose was 3.92×10^{-17}, showing that the recovered TSR did not include any area that was significantly outside the original TSR. The size of the bounds was also comparable: We found that the recovered TSR captured 99% and 97% of the original TSR's extents in translation and rotation, respectively. Thus the original and recovered TSRs are nearly identical.

Cylindrical and Translational TSR Construction

To evaluate our method we conducted two experiments with the Barrett hand in simulation: 1) Generate cylindrical TSRs for a mug. 2) Generate translational TSRs for a box. The cylindrical TSR experiment produced 29 TSRs for the mug. Several TSRs were quite large (see Figure 16.4), however the method also produced TSRs that were quite small. There were two reasons for the small TSRs. First, the force-closure metric would invalidate a grasp which was well within a hypothesized TSR, thus the TSR was split into smaller pieces (this occurred for the left-most TSR in Figure 16.5). The other reason for a small TSR was that it corresponded to a chance alignment in the grasp sampling process, this was the case for the other two TSRs in Figure 16.5.

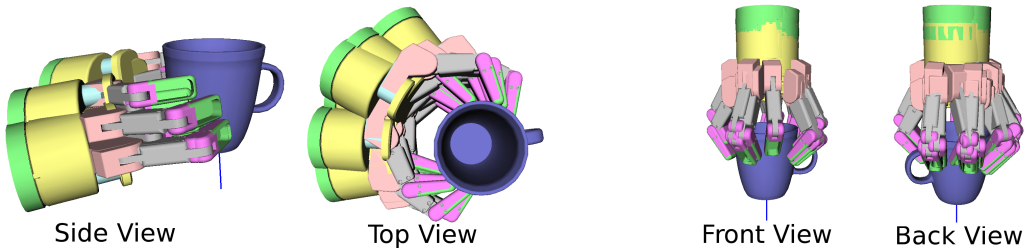


Figure 16.4: Two large cylindrical TSRs produced by our method. The blue line represents the axis of rotation. Ten grasps sampled from each TSR are shown.

The translational TSR experiment produced 68 TSRs for the box. Our method did produce fairly large TSRs for some faces of the box, as shown in Figure 16.6. However, the method produced poor results for other faces, where a large TSR was split by an invalid grasp. Figure 16.7 shows an example of one large TSR being split into four smaller TSRs because of an invalid grasp.

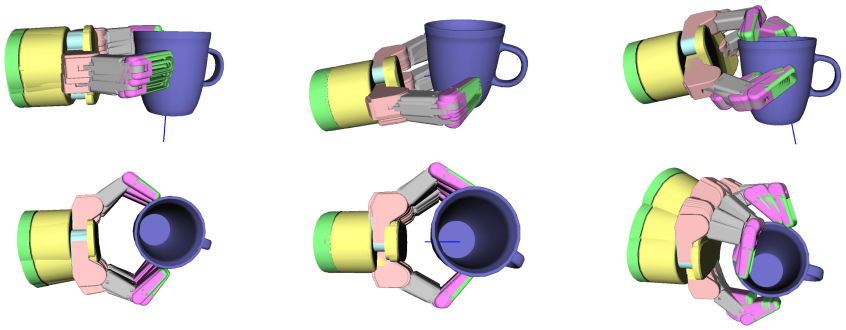


Figure 16.5: Three small cylindrical TSRs produced by our method. The top row is the view from the side, the bottom row is the view from above. Ten grasps sampled from each TSR are shown.

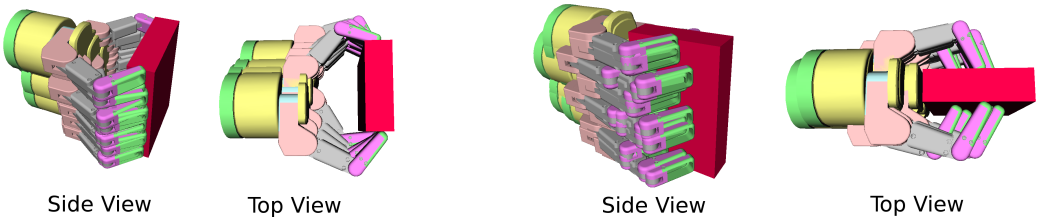


Figure 16.6: Two large translational TSRs for grasping the box. Ten grasps sampled from each TSR are shown.

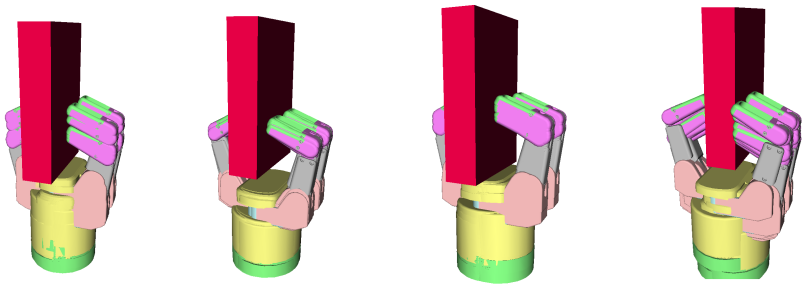


Figure 16.7: One translational TSR was split because of an invalid grasp into four smaller translational TSRs. Ten grasps sampled from each TSR are shown.

16.4 Summary and Discussion

Given the models of the hand and object and the joint values of a preshape, our method aims to construct a set of TSRs that captures all the ways this preshape can be used to grasp the object. To

do this we generate a set of grasps that are valid for the object and use those grasps to guide the construction of translational and cylindrical TSRs. The 5-step construction process computes relations between grasp poses, clusters the relations, and converts the clusters to TSRs. It then refines the TSRs and computes the bounds of a TSR using discretization and checking of grasp stability.

Though our method is capable of producing effective TSRs, some of the TSRs are very small and are not useful for compensating for uncertainty. The number of small TSRs could be fixed by setting the threshold for TSR size higher when computing bounds in step 5 but we would run the risk of loosing unique small TSRs. As shown in Figures 16.5 and 16.7, the grasp evaluation (which uses the force closure metric) splits bigger TSRs into smaller pieces because of a perceived failed grasp and instituting a higher bound on TSR size would eliminate such TSRs completely.

The main issue holding back the automatic generation of TSRs is the lack of an adequate quality metric. The bounds of a TSR are computed by testing a discrete set of grasps to determine if they are stable. A stability metric like force-close is too restrictive because it does not take into account the movement of the object after initial contact is made. Grasps that are robust to pose uncertainty usually "pull-in" the object after making initial contact (as in [126]), which is not captured by the force-closure metric. A full dynamic simulation may be be able to determine the final position of the object after the grasp is executed, however fast simulation methods like those in Open Dynamics Engine (ODE) are notoriously inaccurate for grasping and the more-accurate methods are far too time-consuming. Recent work in dynamic simulation [127, 128] appears to provide reasonable accuracy without sacrificing speed. These simulation methods can be used to determine grasp quality and, in conjunction with the methods in this chapter, to create larger TSRs, which are more robust to uncertainty.

There are also more fundamental issues to address with this method. The key question is: What is a good set of TSRs? One possibility is to maximize the volume of the union of all TSRs while minimizing the number of TSRs. However, it may be difficult to compute this volume because of TSR overlap. It is also unclear what the trade-off between the volume and the number of TSRs should be. Also, this implies that we should discard unique grasp types simply because they don't occupy much volume. Perhaps another criterion to consider is the "diversity" of the set, which would be some measure of variance over the parameters of the TSRs.

Ideally we would also like to compute TSRs in full 6D pose space, however, this is currently impossible because of the discretization required to validate TSRs. Also, more sophisticated clustering algorithms would be required to group grasps together without the cylindrical or translational types.

Chapter 17

Summary

The goal of this thesis was to expand the frontiers of manipulation planning by creating algorithms that plan with a variety of constraints. We devoted a large part of our work to planning with end-effector pose constraints, since these are some of the most useful and common constraints for manipulation tasks. Our pose-constrained manipulation planning framework has three main components: constraint representation, constraint-satisfaction strategies, and a sampling-based approach to planning. These three components come together to create an efficient and probabilistically complete manipulation planning algorithm called the Constrained BiDirectional RRT (CBiRRT2). The underpinning of our framework for pose-related constraints is our Task Space Regions (TSRs) representation. TSRs are intuitive to specify, can be efficiently sampled, and the distance to a TSR can be evaluated very quickly, making them ideal for sampling-based planning. Most importantly, TSRs are a general representation of pose constraints that can fully describe many practical tasks. For more complex tasks, TSRs can be chained together to create more complex end-effector pose constraints, such as those needed to manipulate articulated objects. TSRs can also be used to construct plans that are guaranteed to succeed despite pose uncertainty.

Our constrained manipulation planning framework also allows planning with multiple simultaneous constraints. For instance, collision, torque, and balance constraints can be included along with multiple constraints on end-effector pose. Closed-chain kinematics constraints can also be included as a relation between end-effector pose constraints. In addition, we can plan optimal placements for a robot's base while selecting end-effector poses.

We have applied our framework to a wide range of problems for several robots, both in simulation and in the real world. These problems include grasping in cluttered environments, lifting heavy objects, and two-armed manipulation, to name a few. These example problems demonstrate our framework's practicality, and our proof of probabilistic completeness gives our approach a theoretical foundation.

To broaden the applicability of the framework to more difficult cases, we have also developed the

Constellation algorithm for finding configurations that satisfy multiple constraints without a good initial guess. Constellation can be seen as yet another constraint-satisfaction strategy to be used in our framework, especially when generating goals for a planner.

Yet despite their applicability to a wide range of practical problems, the above algorithms are only capable of planning with hard constraints. We also explored planning with soft constraints using the GradienT-RRT algorithm, which outperformed the state-of-the-art approach to high-dimensional path planning with costs.

Finally, in the second part of this thesis, we focused on generating end-effector goal constraints for grasping tasks. We developed two grasping algorithms that are capable of generating end-effector goal poses for planners very efficiently. The key contribution of these algorithms is that they can find grasps in extremely cluttered environments—an important issue that has been largely overlooked in the grasp planning literature. We then described our work on computing TSRs from the grasp lists produced by these algorithms so that TSRs can be automatically generated for grasping tasks.

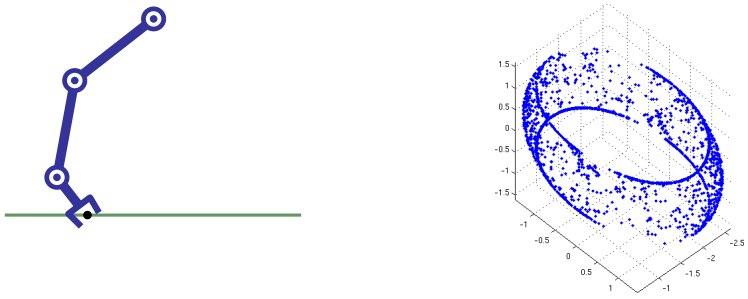
Chapter 18

Discussion and Future Work

This thesis has presented a framework for representing and exploring feasible configurations in the context of manipulation planning and shown how to use this framework to solve real-world manipulation problems. The techniques presented in this thesis can be applied to manipulation planning tasks where the constraints are evaluated as functions of a robot's configuration. Since our research aims to develop a robot that can assist people in the home, we focus on two important tasks for this application that can be formulated using only constraints on configuration: pick-and-place manipulation and manipulating articulated objects, such as doors. The work presented here is not meant to address tasks that require complex forceful interaction with the environment, such as the peg-in-hole problem, or tasks that require planning with dynamics, such as throwing a ball. Though we only consider scleronomic holonomic constraints and plan for only quasi-static motion, these restrictions still allow a robot to perform many useful tasks (as shown in Chapter 8) and permit a rich variety of constraints, including constraints on collision-avoidance, torque, balance, closed-chain kinematics, and end-effector pose.

Some of the most common constraints in manipulation planning involve the pose of a robot's end-effector. These constraints arise in tasks such as reaching to grasp an object, carrying a cup of coffee, or opening a door. The main advantage of our approach to planning with pose constraints is the generality in the constraint representation and planning method. We are able to tackle a wide range of problems without resorting to highly specialized techniques. This is evidenced in part by the low number of parameters in CBiRRT2 and that, despite a wide range of problems and robots, the values of these parameters were constant over many example problems.

Since our framework is built around a sampling-based planner we inherit many of the difficulties of this approach to planning. For instance, it would be difficult to incorporate non-holonomic constraints and dynamics into our framework because these constraints would disrupt the distance metric used by the RRT. On the other hand, we can employ a wide range of techniques that speed up sampling-based planning and repair paths in the presence of moving obstacles. These techniques can be incorporated



(a) (b)

Figure 18.1: Depiction of a TSR and samples on the corresponding constraint manifold (generated using CBiRRT2). (a) The end-effector must be on the line with an orientation within ± 0.7rad of downward. (b) The sampling is biased toward the boundaries of the manifold.

into our planner to make it faster and more robust.

One criticism of TSRs is that the constraint representation may not be sufficiently rich. For instance, some modifications to TSR Chains are necessary to accommodate constraints where degrees of freedom are coupled (as with screw constraints). Indeed, TSRs and TSR Chains cannot capture every conceivable constraint, nor are they intended to. Instead, these representations attempt to straddle the trade-off between practicality and expressiveness. TSRs have proven sufficient for solving a wide range of real-world manipulation problems while still remaining relatively simple and efficient to use in a sampling-based planner. While a more expressive representation is surely possible, we have yet to find one that is as straightforward to specify and as convenient for sampling-based planning.

Even if we were to adopt a different constraint representation to solve a specific problem, many parts of our framework would still be applicable. For instance, we would still be able to use CBiRRT2 as long as the representation provided for fast sampling and distance-checking methods. The proof of probabilistic completeness (Chapter 9) would also apply, because it addresses pose constraints in general, not only TSRs.

One drawback of our framework is that we have not yet devised a method to prioritize constraints. As a result, we can only specify simultaneous constraints using an OR structure if we use the projection strategy. This means that the planner has the option of satisfying constraint 1 OR constraint 2 OR constraint 3, and so on. One way to address this issue would be to attempt to satisfy constraints in order of their priority, though this is not equivalent to a truly prioritized framework. Another approach would be to use projection operators that allow prioritization, like those discussed in Section 9.5.1, however it is unclear how to retain probabilistic completeness while using those operators. The Constellation algorithm could be used to generate configurations that satisfy all the constraints, but in cases where this is not possible, we would have to modify the algorithm to account for prioritization.

A practical issue with CBiRRT2's method of planning on constraint manifolds is that the distribution

of samples may sometimes be undesirable, i.e. the projection strategy biases samples toward the boundaries of the manifold (Figure 18.1). This bias leads to an over-exploration of the boundaries of the manifold to the detriment of exploring the manifold's interior. It can cause the algorithm to perform slowly if an interior point of the manifold is needed to complete a path. On the other hand, the algorithm is much faster at finding configurations on the boundary, which can be useful for problems such as the weight-lifting example (Section 8.5), where the robot must slide the dumbbell to the edge of the table in order to lift it.

In Chapter 16, we discussed our work toward automatically generating TSRs for grasping tasks. This work raises some important questions in terms of automatically generating constraints for planning. For instance, could a robot look at a scene and determine all the areas where a given object can be placed? Such a task would require understanding where the object *can* be placed (through grounding the idea of placing geometrically) and also taking into account user preferences for where objects *should* be placed. Another important issue to explore would be extracting constraints from visual data and/or interaction with the environment [129, 130]. This would be useful for inferring the locations of door hinges or other articulated mechanisms, for example. Automatically creating TSRs for these constraints would be an interesting direction for future work.

Inferring constraints from demonstration and/or experience can also be useful for generating soft constraints, as shown in the path-imitation example of Chapter 13. These constraints can help guide the planner toward more desirable motion without over-restricting it. A promising future direction for this work would be to imitate the reasoning behind the demonstrations rather than the demonstrations themselves. It would be interesting to explore how planners could learn heuristics that would guide the planning process from demonstated paths or even previous runs of the planner.

Finally, there remains a great deal of work to be done in planning with uncertainty. The work presented in Chapter 10 and the uncertain workspace occupancy example in Chapter 13 make some headway in addressing this difficult problem, but much remains to be done. Can we plan grasps given noisy sensor measurement that produce voxel grids with uncertain occupancy? Can we include information-gathering actions in our approach to high-dimensional path planning? Can the robot update its uncertainty models and re-plan quickly as it executes a trajectory? These are the kinds of questions we wish to address in future work.

Appendix A

Appendix

A.1 Faster Short-cut Smoothing

The short-cut smoothing method is one of the fastest and most popular methods for shortening a path. It is frequently used as a post-processing step for sampling-based planners like the RRT or PRM. The method operates by selecting two nodes on the path at random and then attempting to "short-cut" those nodes with a straight line. If the straight-line path between the two nodes is feasible, the path segment between them is replaced by a straight line. This process is repeated for a fixed number of iterations or until a time limit is reached. When planning with constraints, we modified this method by using the extension method of CBiRRT2 instead of a straight line (Chapter 7).

In either case, we found that many short-cut iterations were being wasted attempting to short-cut parts of the path that were already straight (or nearly straight). To prevent wasting computation time, we implemented a simple check: At each iteration, we compute the distance d between the two random nodes selected by the short-cut and compare it to the length l of the segment between those to nodes. If $l - d \leq \alpha d$, we skip the current iteration. We use $\alpha = 0.1$ in our implementation.

This check eliminates unnecessary short-cuts according to the value of α. Higher α values will tend to reject more short-cuts, so smoothing will be faster, however the resulting paths will likely be longer. An α of 0 will produce identical behavior to standard short-cut smoothing but may save some iterations by rejecting attempts to short-cut segments that are exactly straight.

Implementing this simple check allowed us to save a great deal of time when shortening paths, and resulted in noticeably more responsive robot behavior.

A.2 Multi-root RRTs

The CBiRRT2 algorithm, like other planners, can consider multiple start and goal configurations while planning and it can generate starts and goals during the planning process. When we present this algorithm, we are often asked how this feature is implemented. While it may seems quite difficult to grow multiple trees from multiple roots and decide which tree should extend toward which other tree, the implementation is actually quite simple.

In our implementation, an RRT is stored as a linked list of nodes. We maintain two lists: one for the start tree(s) and one for the goal tree(s). Each node in a list contains the index of its parent. When a root node is added to the start or goal list (either before the planner runs or while the planner is running), we set the parent index of this node to -1. Then, when we trace back the path from the connection point between a start branch and a goal branch (i.e. when two trees have connected and we have found a path), we terminate the trace when we reach a node with a parent index of -1.

Using this straightforward implementation, there is no need to manage which tree grows at which time. All nodes are treated equally when performing a nearest-neighbor check: we simply find the closest node in the start or goal list to a random sample.

Bibliography

[1] T. Lozano-Perez, J. Jones, E. Mazer, P. O'Donnell, W. Grimson, P. Tournassoud, and A. Lanusse, "Handey: A robot system that recognizes, plans, and manipulates," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 1987, pp. 843–849.

[2] R. P. Paul, "Modelling, trajectory calculation and servoing of a computer controlled arm." Ph.D. dissertation, Department of Computer Science, Stanford University, 1972.

[3] P. Winston, *The MIT Robot*. Edinburgh University Press, 1972, pp. 431–463.

[4] A. P. Ambler, H. G. Barrow, C. M. Brown, R. H. Burstall, and R. J. Popplestone, "A versatile computer-controlled assembly system," in *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, Aug. 1973, pp. 298–307.

[5] K. Ikeuchi, H. K. Nishihara, B. K. P. Horn, P. Sobalvarro, and S. Nagata, "Determining Grasp Configurations using Photometric Stereo and the PRISM Binocular Stereo System," *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 46–65, Mar. 1986.

[6] T. Lozano-Perez and R. Brooks, "An Approach to Automatic Robot Programming," Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep. A.I. Memo No. 842, 1985.

[7] P. Cheng, J. Fink, S. Kim, and V. Kumar, "Cooperative towing with multiple robots," 2008.

[8] E. W. Aboaf, S. M. Drucker, and C. G. Atkeson, "Task-level robot learning: Juggling a tennis ball more accurately," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 1989.

[9] T. Vose, P. Umbanhowar, and K. M. Lynch, "Friction-induced velocity fields for parts sliding on a rigid oscillated plate," in *Proc. Robotics: Science and Systems*, 2008.

[10] M. Stilman, "Task constrained motion planning in robot joint space," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007.

[11] Y. Koga, K. Kondo, J. Kuffner, and J. claude Latombe, "Planning motions with intentions," in *SIGGRAPH*, 1994.

[12] K. Yamane, J. Kuffner, and J. Hodgins, "Synthesizing animations of human manipulation tasks," in *SIGGRAPH*, 2004.

[13] Z. Yao and K. Gupta, "Path planning with general end-effector constraints: using task space to guide configuration space search," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.

[14] E. Drumwright and V. Ng-Thow-Hing, "Toward interactive reaching in static environments for humanoid robots," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006.

[15] D. Bertram, J. Kuffner, R. Dillmann, and T. Asfour, "An integrated approach to inverse kinematics and path planning for redundant manipulators," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2006.

[16] J. H. Yakey, S. M. LaValle, and L. E. Kavraki, "Randomized path planning for linkages with closed kinematic chains," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 6, pp. 951–958, 2001.

[17] J. Cortes and T. Simeon, "Sampling-based motion planning under kinematic loop-closure constraints," in *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2004.

[18] S. Dalibard, A. Nakhaei, F. Lamiraux, and J.-p. Laumond, "Whole-Body Task Planning for a Humanoid Robot: A Way to Integrate Collision Avoidance," in *Humanoids*, 2009.

[19] S. Sentis and O. Khatib, "Synthesis of whole-body behaviors through hierarchical control of behavioral primitives," *International Journal of Humanoid Robotics*, vol. 2, pp. 505–518, December 2005.

[20] P. Combettes, "The foundations of set theoretic estimation," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 182–208, Feb. 1993.

[21] A. Saxena, J. Driemeyer, J. Kearns, and A. Ng, "Robotic grasping of novel objects," in *Proc. Neural Information Processing Systems (NIPS)*, 2007.

[22] K. Hsiao and T. Lozano-Perez, "Imitation learning of whole-body grasps," in *Robotics: Science and Systems Workshop on Manipulation for Human Environments*, 2006.

[23] R. Pelossof, A. Miller, P. Allen, and T. Jebara, "An SVM learning approach to robotic grasping," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2004.

[24] Y. Li, J. Fu, and N. Pollard, "Data driven grasp synthesis using shape matching and task-based pruning," in *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[25] N. Ratliff, J. A. Bagnell, and S. S. Srinivasa, "Imitation learning for locomotion and manipulation," in *Proc. IEEE-RAS International Conference on Humanoid Robots*, 2007.

[26] Z. Li and S. Sastry, "Task oriented optimal grasping by multifingered robot hands," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 1987, pp. 389–394.

[27] X. Zhu and J. Wang, "Synthesis of force-closure grasps on 3-D objects based on the Q distance," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 4, pp. 669–679, 2003.

[28] T. Sugihara and Y. Nakamura, "Whole-body cooperative balancing of humanoid robot using COG jacobian," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.

[29] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *IEEE Transactions on Robotics and Automation*, vol. 3, no. 1, pp. 43–53, 1987.

[30] R. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[31] C. G. Atkeson and S. Schaal, "Learning tasks from a single demonstration," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 1997, pp. 1706–1712.

[32] D. Bentivegna, C. G. Atkeson, and G. Cheng, "Learning tasks from observation and practice," *Robotics and Autonomous Systems*, vol. 47, pp. 163–169, 2004.

[33] M. Howard, S. Klanke, M. Gienger, C. Goerick, and S. Vijayakumar, "Learning potential-based policies from constrained motion," in *Proc. IEEE-RAS International Conference on Humanoid Robots*, 2008.

[34] S. LaValle and J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2000.

[35] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[36] J. Bobrow, S. Dubowsky, and J. Gibson, "Time-optimal control of robotic manipulators along specified paths," *International Journal of Robotics Research*, vol. 4, pp. 3–17, 1985.

[37] R. Fikes and N. Nilsson, "STRIPS: A new approach to the application of theorem proving," *Artificial Intelligence Journal*, vol. 2, pp. 189–208, 1971.

[38] N. Nilsson, *Principles of Artificial Intelligence*. Wellsboro, PA: Tioga Publishing Company, 1980.

[39] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. San Francisco, CA: Morgan Kaufman, 2004.

[40] A. Ambler and R. Popplestone, "Inferring the positions of bodies from specified spatial relationships," *Artificial Intelligence*, vol. 6, no. 2, pp. 157 – 174, 1975.

[41] R. Finkel, R. Taylor, R. Bolles, R. Paul, and J. Feldman, "AL, A programming System for Automation," Computer Science Department, Stanford University, Tech. Rep. CS-456, 1974.

[42] R. Taylor, "The synthesis of manipulator control programs from task-level specifications." Ph.D. dissertation, Computer Science Department, Stanford University, 1976.

[43] L. I. Lieberman and M. A. Wesley, "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly," *IBM Journal of Research and Development*, vol. 21, no. 4, pp. 321–333, July 1977.

[44] M. Vande Weghe, D. Ferguson, and S. S. Srinivasa, "Randomized path planning for redundant manipulators without inverse kinematics," in *Proc. IEEE-RAS International Conference on Humanoid Robots*, 2007.

[45] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decre, R. Smits, E. Aertbelien, K. Claes, and H. Bruyninckx, "Constraint-Based Task Specification and Estimation for Sensor-Based Robot Systems in the Presence of Geometric Uncertainty," *International Journal of Robotics Research (IJRR)*, vol. 26, no. 5, pp. 433–455, 2007.

[46] S. Seereeram and J. Wen, "A global approach to path planning for redundant manipulators," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 1, pp. 152–160, Feb 1995.

[47] G. Oriolo, M. Ottavi, and M. Vendittelli, "Probabilistic motion planning for redundant robots along given end-effector paths," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and System (IROS)*, 2002.

[48] G. Oriolo and C. Mongillo, "Motion planning for mobile manipulators along given end-effector paths," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2005.

[49] L. Sciavicco and B. Siciliano, *Modeling and Control of Robot Manipulators*, 2nd ed. Springer, 2000, pp. 96–100.

[50] M. Stilman, J. U. Schamburek, J. Kuffner, and T. Asfour, "Manipulation planning among movable obstacles," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2007.

[51] Y. Hirano, K. Kitahama, and S. Yoshizawa, "Image-based object recognition and dexterous hand/arm motion planning using RRTs for grasping in cluttered scene," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.

[52] T. Lozano-Perez, M. Mason, and R. H. Taylor, "Automatic synthesis of fine-motion strategies for robots," *International Journal of Robotics Research*, vol. 3, no. 1, 1984.

[53] M. Erdmann, "Using Backprojections for Fine Motion Planning with Uncertainty," *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 19–45, Mar. 1986.

[54] J. Canny, "On computability of fine motion plans," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 1989.

[55] D. Montana, "Contact stability for two-fingered grasps," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 4, pp. 421–430, 1992.

[56] H. Hanafusa and H. Asada, *Robot Motion*. The MIT Press, 1982, ch. Stable prehension by a robot hand with elastic fingers.

[57] M. R. Cutkosky, *Robotic Grasping and Fine Manipulation*. Kluwer Academic, Aug. 1985.

[58] V.-D. Nguyen, "Constructing Stable Grasps," *The International Journal of Robotics Research*, vol. 8, no. 1, pp. 26–37, Feb. 1989.

[59] J. Ponce, S. Sullivan, A. Sudsang, J.-D. Boissonnat, and J.-P. Merlet, "On Computing Four-Finger Equilibrium and Force-Closure Grasps of Polyhedral Objects," *The International Journal of Robotics Research*, vol. 16, no. 1, pp. 11–35, Feb. 1997.

[60] N. S. Pollard, "Closure and quality equivalence for efficient synthesis of grasps from examples," *The Internation Journal of Robotics Research*, vol. 23, pp. 595–613, June 2004.

[61] R. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainty," *The Internation Journal of Robotics Research*, vol. 5, no. 4, pp. 56–68, May 1986.

[62] M. Sallinen, "Modelling and estimation of spatial relationships in sensor-based robot work-cells," Ph.D. dissertation, VTT Electronics/University of Oulu, Oulu, Finland, 2003.

[63] D. Hsu, J.-C. Latombe, and S. Sorkin, "Placing a robot manipulator amid obstacles for optimized execution," in *Proc. IEEE International Symposium on Assembly and Task Planning (ISATP'99)*, 1999, pp. 280–285.

[64] M. Zhao, N. Ansari, and E. Hou, "Mobile Manipulator Path Planning By A Genetic Algorithm," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1992, pp. 681–688.

[65] M. Chen and A. M. S. Zalzala, "A genetic approach to motion planning of redundant mobile manipulator systems considering safety and configuration," *Journal of Robotic Systems*, vol. 14, no. 7, pp. 529–544, Jul. 1997.

[66] J. Vannoy and J. Xiao, "Real-Time Adaptive Motion Planning (RAMP) of Mobile Manipulators in Dynamic Environments With Unforeseen Changes," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1199–1212, Oct. 2008.

[67] S. Cambon, F. Gravot, and R. Alami, "A robot task planner that merges symbolic and geometric reasoning," in *Proc. European Conference on Artificial Intelligence*, 2004.

[68] T. Simeon, J. Laumond, J. Cortes, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, p. 729, 2004.

[69] R. Alami, J. Laumond, and T. Simon, "Two manipulation planning algorithms," in *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 1994.

[70] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal 7*, pp. 308–313, 1965.

[71] A. Escande and A. Kheddar, "Contact planning for acyclic motion with tasks constraints," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2009.

[72] A. Neumaier, "Complete Search in Continuous Global Optimization and Constraint Satisfaction," *Acta Numerica*, pp. 1–94, 2004.

[73] L. Jaillet, J. Cortés, and T. Siméon, "Sampling-Based Path Planning on Configuration-Space Costmaps," *IEEE Transactions on Robotics*, vol. 26, no. 4, pp. 635–646, Aug. 2010.

[74] C. Urmson and R. Simmons, "Approaches for heuristically biasing RRT growth," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2003.

[75] D. Ferguson and A. Stentz, "Anytime RRTs," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006.

[76] A. Ettlin and H. Bleuler, "Randomised Rough-Terrain Robot Motion Planning," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006.

[77] M. Apaydin, A. Singh, D. Brutlag, and J.-C. Latombe, "Capturing molecular energy landscapes with probabilistic conformational roadmaps," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2001.

[78] P. Chen and Y. Hwang, "SANDROS: a dynamic graph search algorithm for motion planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 3, pp. 390–403, Jun 1998.

[79] R. Geraerts and M. H. Overmars, "Creating High-quality Paths for Motion Planning," *The International Journal of Robotics Research*, vol. 26, no. 8, pp. 845–863, 2007.

[80] S. Rodriguez and N. Amato, "An obstacle-based rapidly-exploring random tree," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2006.

[81] J. Pan, L. Zhang, and D. Manocha, "Retraction-based RRT planner for articulated models," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, May 2010.

[82] D. Hsu and J. Reif, "The bridge test for sampling narrow passages with probabilistic roadmap planners," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2003.

[83] S. Dalibard and J. Laumond, "Control of probabilistic diffusion in motion planning," in *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2008.

[84] D. Hsu, G. Sanchez-Ante, H.-l. Cheng, and J.-C. Latombe, "Multi-level free-space dilation for sampling narrow passages in PRM planning," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2006.

[85] A. T. Miller, S. Knoop, H. I. Christensen, and P. K. Allen, "Automatic grasp planning using shape primitives," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2003.

[86] C. Goldfeder, M. Ciocarlie, and P. Allen, "The Columbia grasp database," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, May 2009.

[87] D. Berenson, S. S. Srinivasa, D. Ferguson, A. Collet, and J. Kuffner, "Manipulation planning with workspace goal regions," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

[88] D. Berenson, S. S., and K. J., "Task space regions: A framework for pose-constrained manipulation planning," *The International Journal of Robotics Research*, March 2011.

[89] D. Berenson, J. Chestnutt, S. S. Srinivasa, J. J. Kuffner, and S. Kagami, "Pose-Constrained Whole-Body Planning using Task Space Region Chains," in *Proc. IEEE-RAS International Conference on Humanoid Robots*, 2009.

[90] D. Berenson, S. S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation planning on constraint manifolds," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

[91] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010.

[92] M. Walker and D. Orin, "Efficient dynamic computer simulation of robotic mechanisms," *ASME Journal of Dynamic Systems Measurement and Control*, vol. 104, pp. 205–211, 1982.

[93] R. Diankov, S. S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation planning with caging grasps," in *Proc. IEEE-RAS International Conference on Humanoid Robots*, 2008.

[94] D. Berenson and S. Srinivasa, "Probabilistically complete planning with end-effector pose constraints," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, May 2010.

[95] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

[96] J. Burdick, "On the inverse kinematics of redundant manipulators: characterization of the self-motion manifolds," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, May 1989, pp. 264–270.

[97] H. K. Khalil, *Nonlinear Systems*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

[98] P. Svestka, "On probabilistic completeness and expected complexity of probabilistic path planning," Department of Computer Science, Utrecht University, Utrecht, Netherlands, Tech. Rep. UU-CS-96-20, 1996.

[99] J. Kuffner, J. J. and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[100] K. Hauser and J. Latombe, "Multi-Modal Motion Planning in Non-Expansive Spaces," in *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2008.

[101] D. Berenson, S. Srinivasa, and J. Kuffner, "Addressing Pose Uncertainty in Manipulation Planning Using Task Space Regions," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2009.

[102] L. Devroye, *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986, pp. 567–571.

[103] D. Berenson, J. Kuffner, and H. Choset, "An optimization approach to planning for mobile manipulation," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2008.

[104] M. T. Mason, *Mechanics of Robotic Manipulation*. MIT Press, 2001.

[105] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1998.

[106] P. Kaiser, D. Berenson, N. Vahrenkamp, T. Asfour, R. Dillmann, and S. Srinivasa, "Constellation - An Algorithm for Finding Robot Configurations that Satisfy Multiple Constraints," in *submitted to IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2011.

[107] T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schröder, and R. Dillmann, "Toward Humanoid Manipulation in Human-Centred Environments," *Robotics and Autonomous Systems*, vol. 56, pp. 54–65, January 2008.

[108] N. Tsakarakis, G. Metta, G. Sandini, D. Vernon, R. Beira, F. Becchi, L. Righetti, J. Santos-Victor, A. Ijspeert, M. Carrozza, and D. Caldwell, "iCub - The Design and Realization of an Open Humanoid Platform for Cognitive and Neuroscience Research," *Journal of Advanced Robotics, Special Issue on Robotic platforms for Research in Neuroscience*, vol. 21, no. 10, pp. 1151–1175, 2007.

[109] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha, "Fast proximity queries with swept sphere volumes," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[110] D. Berenson, T. Simeon, and S. Srinivasa, "Addressing Cost-Space Chasms in Manipulation Planning," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[111] L. Kavraki, J. Latombe, R. Motwani, and P. Raghavan, "Randomized Query Processing in Robot Path Planning," *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 50–60, Aug. 1998.

[112] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, May 2009.

[113] J. Ratcliff. (2010) Convex decomposition library. [Online]. Available: http://code.google.com/p/convexdecomposition/

[114] E. A. Sisbot, L. F. Marin-Urias, R. Alami, and T. Simeon, "Human aware mobile robot motion planner," *IEEE Transactions on Robotics*, vol. 23, pp. 874–883, 2007.

[115] S. Calinon, F. Guenter, and A. Billard, "On Learning, Representing, and Generalizing a Task in a Humanoid Robot," *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, vol. 37, no. 2, pp. 286–298, Apr. 2007.

[116] D. Berenson, R. Diankov, K. Nishiwaki, S. Kagami, and J. Kuffner, "Grasp planning in complex scenes," in *Proc. IEEE-RAS International Conference on Humanoid Robots*, 2007.

[117] A. Morales, T. Asfour, P. Azad, S. Knoop, and R. Dillmann, "Integrated grasp planning and visual object localization for a humanoid robot with five-fingered hands," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006.

[118] K. Okada, A. Haneda, H. Nakai, M. Inaba, and H. Inoue, "Environment manipulation planner for humanoid robots using task graph that generates action sequence," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[119] D. Berenson and S. S. Srinivasa, "Grasp synthesis in cluttered environments for dexterous hands," in *Robotics Science and Systems (RSS) Workshop on Robot Manipulation: Intelligence in Human Environments*, May 2008.

[120] ——, "Grasp synthesis in cluttered environments for dexterous hands," in *Proc. IEEE-RAS International Conference on Humanoid Robots*, December 2008.

[121] E. Oztop and M. Arbib, "Schema design and implementation of the grasp-related mirror neuron system," *Biological Cybernetics*, vol. 87, pp. 116–140, 2002.

[122] S. Stansfield, "Robotic grasping of unknown objects: A knowledge-based approach," *International Journal of Robotics Research*, vol. 10, pp. 314–326, 1991.

[123] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha, "Fast proximity queries with swept sphere volumes," Department of Computer Science, UNC Chapel Hill, Tech. Rep. TR99-018, 1999.

[124] D. Eggert, A. Lorusso, and R. Fisher, "Estimating 3-D rigid body transformations: a comparison of four major algorithms," *Machine Vision and Applications*, vol. 9, pp. 272–290, 1997.

[125] Q. Zheng and W.-H. Qian, "An enhanced ray-shooting approach to force-closure problems," *Journal of Manufacturing Science and Engineering*, vol. 128, no. 4, pp. 960–968, 2006.

[126] M. R. Dogar and S. S. Srinivasa, "Push-Grasping with Dexterous Hands : Mechanics and a Method," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[127] B. Nguyen and J. C. Trinkle, "dvc3D : a three dimensional physical simulation tool for rigid bodies with contacts and Coulomb friction," in *Proc. Joint International Conference on Multibody System Dynamics*, 2010.

[128] E. Drumwright and D. A. Shell, "An Evaluation of Methods for Modeling Contact in Multibody Simulation," in *Proc. IEEE International Conference on Robotics and Automation ICRA 2011*, 2011.

[129] D. Katz, Y. Pyuro, and O. Brock, "Learning to manipulate articulated objects in unstructured environments using a grounded relational representation," in *Robotics Science and Systems (RSS)*, 2008.

[130] J. Sturm, V. Pradeep, and C. Stachniss, "Learning kinematic models for articulated objects," in *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.