Reachability-based Trajectory Design

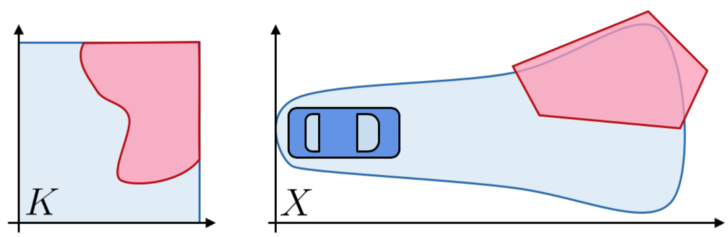
by

Shreyas Kousik

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Mechanical Engineering)
in the University of Michigan
2020

Doctoral Committee:

Assistant Professor Ramanarayan Vasudevan, Chair
Associate Professor Dmitry Berenson
Professor Jessy Grizzle
Associate Professor Necmiye Ozay



K
X

Shreyas Kousik
skousik@umich.edu
ORCID iD: 0000-0003-1348-7463

DEDICATION

For Thatha, who listened with delight to my defense.

ACKNOWLEDGMENTS

This work would not have been possible without the many friends I made along the way. At UM, I want to thank my Mechanical Engineering cohort, my Munger friends, the Stats crew, and Patrick. In Michigan, I want to thank my friends and mentors from GM, especially los Boricuas, and all my music lovers and townies in Ann Arbor. I also would not have been able to get this done without the immense and unwavering support from my family. Most importantly, my parents, who are always just a phone call away when life is challenging – I am lucky to have your unconditional love and support. Next, I have to thank my labmates and collaborators. Sean, it has been a pleasure to work with someone so bright and passionate, who is always down to kick around a strange idea. Wheels are better than legs! Patrick, I'm so glad we got to play with 'topes for the past few years. It's really rare to find someone who can be such a dear friend, excellent colleague, and optimal hiking companion. Fan, Hannah, Pengcheng, Utkarsh, Bohao, Daphna, Corina, Shankar, James, Stew, and MJR – I cherish our chats and struggles to get all those papers out the door. To everyone else in ROAHM Lab, thank you for being such a supportive, caring, and wonderful group of people. You made my time at Michigan a blast. To all my current collaborators, I'm excited to see what we'll come up with next! Finally, thank you, Ram. I knew you would be an awesome advisor from that very first phone call, when I was just some kid at Georgia Tech, and you already believed in me. From the start, you always told me and Pat to believe in ourselves instead of comparing ourselves to others – but of course, we have to compare ourselves to you! Over the years, I've grown so much because of your constant support and belief that I can indeed get cool stuff done. I hope to be as good a teacher, mentor, and friend to others as you have been to me.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

Autonomous mobile robots have the potential to increase the availability and accessibility of goods and services throughout society. However, to enable public trust in such systems, it is critical to certify that they are safe. This requires formally specifying safety, and designing motion planning methods that can guarantee safe operation (note, this work is only concerned with planning, not perception).

The typical paradigm to attempt to ensure safety is receding-horizon planning, wherein a robot creates a short plan, then executes it while creating its next short plan in an iterative fashion, allowing a robot to incorporate new sensor information over time. However, this requires a robot to plan in real time. Therefore, the key challenge in making safety guarantees lies in balancing performance (how quickly a robot can plan) and conservatism (how cautiously a robot behaves). Existing methods suffer from a tradeoff between performance and conservatism, which is rooted in the choice of model used describe a robot; accuracy typically comes at the price of computation speed.

To address this challenge, this dissertation proposes Reachability-based Trajectory Design (RTD), which performs real-time, receding-horizon planning with a simplified planning model, and ensures safety by describing the model error using a reachable set of the robot.

RTD begins with the offline design of a continuum of parameterized trajectories for the planning model; each trajectory ends with a fail-safe maneuver such as braking to a stop. RTD then computes the robot's Forward Reachable Set (FRS), which contains all points in workspace reachable by the robot for each parameterized trajectory. Importantly, the FRS also contains the error model, since a robot can typically never track planned trajectories perfectly. Online (at runtime), the robot intersects the FRS with sensed obstacles to provably determine which trajectory plans could cause collisions. Then, the robot performs trajectory optimization over the remaining safe trajectories. If no new safe plan can be found, the robot can execute its previously-found fail-safe maneuver, enabling perpetual safety.

This dissertation begins by presenting RTD as a theoretical framework, then presents three representations of a robot's FRS, using (1) sums-of-squares (SOS) polynomial programming, (2) zonotopes (a special type of convex polytope), and (3) rotatotopes (a generalization of zonotopes that enable representing a robot's swept volume). To enable real-time planning, this work also develops an obstacle representation that enables provable safety while treating obstacles as discrete,

finite sets of points. The practicality of RTD is demonstrated on four different wheeled robots (using the SOS FRS), two quadrotor aerial robots (using the zonotope FRS), and one manipulator robot (using the rotatotope FRS). Over thousands of simulations and dozens of hardware trials, RTD performs safe, real-time planning in arbitrary and challenging environments.

In summary, this dissertation proposes RTD as a general purpose, practical framework for provably safe, real-time robot motion planning.

CHAPTER 1

Introduction

While people are capable of performing a wide variety of tasks, they cannot always guarantee that any task will be completed safely and successfully. This is especially true for tasks that are difficult or dangerous to perform consistently and repeatedly, such as long-distance delivery, emergency response, and in-home care for the elderly.

Autonomous robots, if carefully developed and deployed, have the potential to perform many tasks in place of humans. But, what tasks should, and can, be automated? Much of farming and manufacturing is already automated. However, it is much harder to automate the distribution of goods, construction of infrastructure, and collaboration between robots and people. Many such tasks can be solved using mobile robots and manipulators, which can move through, and interact with, the world. One could certainly build such a robot without regard for the health and safety of people, but this robot is not likely to integrate well into society or be widely used. Therefore, while we should certainly build robots that are capable of completing tasks successfully, we should also be able to certify that such robots are **safe**.

This dissertation addresses how to perform **provably-safe robot motion planning** by proposing a method called Reachability-based Trajectory Design (RTD). This chapter introduces RTD by presenting an overview of the scope and goals of this work, then listing specific technical contributions. We also present the dissertation's organization and notation.

1.1 Overview

We now present the scope, goals, and context of this dissertation.

1.1.1 Scope and Goals

To move through or interact with the world, a robot must perceive its surroundings, predict the motion of other mobile actors, and plan its own motion. This work is only concerned with **planning**, not perception or prediction.

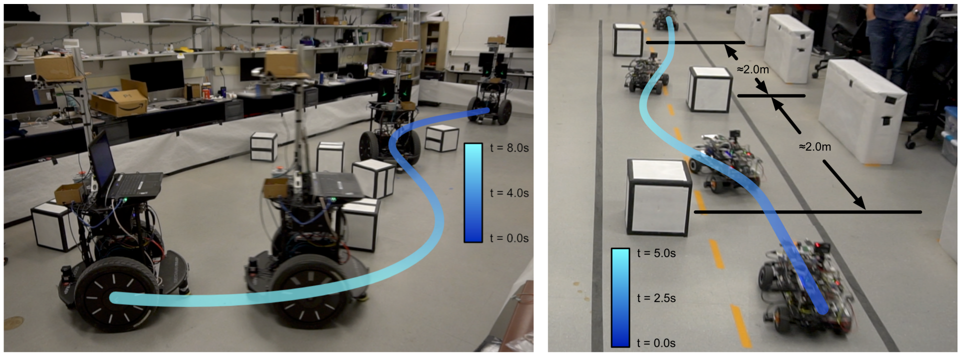


Figure 1.1: A Segway robot (left) and a Rover robot (right) use RTD to safely and successfully perform trajectory planning through a variety of random and structured static scenes [KVB$^+$20]. Each robot's trajectory is shown fading from dark to light with the passage of time.

The overall goal of this work is safe, real-time, receding-horizon robot motion planning. Achieving this goal requires addressing several sub-goals. First, we must specify mathematical descriptions of robots that are complex enough to describe robots accurately, but simple enough to enable real-time planning. Second, we must be able to compensate for our imperfect models. Third, we must be able to represent the salient information in a robot's environment necessary for safe planning.

Next, we discuss the context of this work. We introduce receding-horizon planning, which is performed using a planning hierarchy. Then, we briefly discuss reachability analysis, the underlying tool that enables RTD. Finally, we state the research gaps that RTD addresses.

1.1.2 Receding-Horizon Planning

Robots acquire new information from sensors with limited range, which we call a finite **sensor horizon**. To incorporate new sensor information into motion planning, robots typically use a **receding-horizon** strategy, wherein a robot executes a short plan while creating a new short plan in an iterative fashion. This strategy applies across different robot morphologies, such as wheeled robots [HGK10, KQCD15], aerial robots [GKM10], and manipulator arms [Hau12, MSS18].

Receding-horizon planning requires the robot to plan in **real time**. Real-time planning in **static** environments means the robot must create a new plan before it finishes executing its previously-planned trajectory. When a robot always has a plan available, we call its planning algorithm **persistently feasible** [KVB$^+$20]. If each plan is long in duration, this may not be difficult to achieve; similarly, if each plan ends with the robot stopped, then the robot can execute an entire plan, then stay stopped while planning its next motion. However, when environments are **dynamic** (that is, containing other moving actors), the robot must plan with respect to predictions of other actors' behavior. Since prediction accuracy decreases as prediction duration increases [JHJRV17], it is important that a robot can re-plan quickly, so that its plans (and predictions) can be of shorter duration.

1.1.3 The Planning Hierarchy

Receding-horizon planning is typically broken up into a three-tiered **planning hierarchy** (see, e.g., [KQCD15, GPMN15, KVB$^+$20, McN11, UAB$^+$08]). At the top of the hierarchy is a **path planner** that attempts to rapidly find a path through the robot's workspace from start to goal, typically by ignoring the robots dynamics. The output of the path planner is passed to a **trajectory planner**, which attempts to produce a dynamically-feasible trajectory that tracks the path as closely as possible. The output of the trajectory planner is passed to a **tracking controller**, which generates inputs for the robot's actuators to track a trajectory, typically using state feedback.

We discuss how one can attempt to enforce safety at each tier of this hierarchy in §2, wherein we review the relevant literature. In short, we find the following. The path planning tier may struggle to enforce safety because it sacrifices an accurate representation of the robot's dynamics for planning speed. The tracking controller tier may similarly struggle to enforce safety without incurring excessive conservatism, because a common approach to enforcing safety is to treat the path and trajectory planners as a disturbance. This leads us to develop RTD as a trajectory planner, which is able to safely bridge the gap between unsafe path planners and unsafe tracking controllers.

1.1.4 Reachability Analysis

To enable safe robot motion planning, we require a mathematical framework for describing how these robots move through the world. The particular framework used in this work is **reachability analysis**, hence the name Reachability-based Trajectory Design. Here, we briefly discuss what reachability analysis is, and why it is useful; we provide particular examples in §2.

Reachability analysis is concerned with how sets evolve when subject to vector fields. This framework can be used to assess the safety of a robot by expressing its body and states as elements of sets, and its motion as a vector field. In particular, we care that all points in space that are reachable by a robot, when executing a particular motion plan, lie outside of obstacles.

In this dissertation, we present a generic formulation of reachable sets for motion planning in §3. To implement this formulation, we perform reachability analysis using sums-of-squares (SOS) programming (§4), zonotope reachability (§6 and §8), and sampling (§7). Examples of each of these methods are discussed in §2.

1.1.5 Research Gap

The key challenge in robot motion planning is to enforce safety without sacrificing performance; this challenge arises from the high-dimensional models typically used to accurately describe robots, in contrast to the simplified models typically used for real-time planning. In other words, one must compensate for tracking error between accurate models and simplified planning models.

The key challenge in reachability analysis is to numerically represent and compute reachable sets for high-dimensional systems. Indeed, it is typically possible to compute reachable sets for simplified planning models, but, as with the challenge of real-time planning, one must incorporate tracking error.

RTD addresses this research gap by specifying how one should create a simplified planning model and represent tracking error, and by specifying a variety of methods to compute reachable sets. Note, we revisit these research gaps later in §2 in the context of the literature.

1.2 Contributions

The proposed method of this dissertation is **Reachability-based Trajectory Design** (RTD). This section introduces the reader to RTD, and lays the foundation for the rest of the dissertation. First, we summarize the contributions of RTD. Second, we list the specific contributions of each paper in which RTD has been developed.

1.2.1 Summary of Contributions

This work summarizes the development and implementation of RTD. As detailed across several papers, RTD has been applied to wheeled robots [KVJRV17, KVB+20, VSK+19, VKL+19, VLK+19], quadrotor drones [KHV19], and manipulator arms [HKZ+20].

RTD is a real-time, provably-safe, receding-horizon trajectory planner for robots in arbitrary environments. The method is successfully demonstrated on a wide variety of robot morphologies. RTD outperforms other methods in the current literature in terms of both safety and performance, meaning that a robot using RTD is more often able to reach desired locations without suffering collisions.

The particular contributions of this work are: (1) offline computation of parameterized reachable sets for a variety of robot morphologies; (2) online, receding-horizon computation of provably-safe trajectory plans; (3) simulations demonstrating RTD outperforming other recent methods in terms of both safety and task completion; (4) hardware demonstrations on five different platforms that demonstrate the versatility and efficacy of RTD.

1.2.2 Contributions per Paper

We now summarize the development of RTD in terms of the papers [KVJRV17, KVB+20, VSK+19, VKL+19, VLK+19, KHV19, HKZ+20]. This section presents a summary of each paper, and how they are linked together. The body of this dissertation breaks these papers apart into their theoretical and practical components, and reorganizes them into a cohesive framework, which we summarize in §1.3.

1.2.2.1 Safety for Wheeled Robots

The theory underlying RTD was first presented in [KVJRV17]. This work uses SOS programming (represented with a semidefinite program, or SDP) to compute a parameterized FRS of an autonomous car's trajectories, plus tracking error, offline; the FRS is represented as a semialgebraic set. At runtime, the FRS is intersected with obstacles, represented as semialgebraic sets, by solving another SDP, producing a semialgebraic set that overapproximates the unsafe trajectory parameters in a particular receding-horizon planning iteration. This work also establishes the minimum time horizon required for each plan, and the minimum sensor horizon required to certify safe planning. The autonomous car is described by a dynamic unicycle model, and the parameterized trajectories are Dubins paths [Dub57], with speed and yaw rate as the trajectory parameters. While this work provides sufficient bounds to ensure the safety of RTD's receding-horizon planning, the runtime SDP is too slow for real-time planning.

1.2.2.2 Real-time Performance

We solved the problem of real-time, safe planning with RTD in [KVB+20], as shown in Figure 1.1. This work provides a detailed method for representing obstacles as discrete, finite sets of points, as opposed to semialgebraic sets. Then, instead of using a SOS program to compute the set of unsafe trajectory parameters in each planning iteration, we need only perform a polynomial evaluation, which is three orders of magnitude faster. This work also extends a system decomposition approach, presented in [CHV+18] for HJB reachability, to SOS reachability analysis, enabling RTD to be applied for a bicycle model of a car with a lane change parameterization. In this work, RTD is applied to a Segway robot and a car-like Rover. This work bridges the gap between safety and real-time performance, thereby addressing the most important challenge discussed in the literature review. Furthermore, this work performs a comparison between RTD, RRT, and NMPC, and shows that RTD is able to outperform the other two planners in terms of safety and performance. However, the robots in this work only move at under 2 m/s, meaning that they are able to use a short trajectory duration, leaving it unclear how to extend RTD to larger robots in more realistic scenarios.

1.2.2.3 Increased Model Complexity

To show the practicality of RTD for larger robots at higher speeds in more realistic scenarios, we demonstrated the method on a high-fidelity model of a passenger car [VSK+19]. The car is simulated in CarSim [Mec18]. Using RTD, the car is able to autonomously navigate a 1 km test track (in the MCity proving ground [UMT15]) safely at up to 15 m/s (the speed limit of the test track), despite randomly-placed obstacles. We also showed that, when there is no feasible path forward, RTD causes the car to safely brake to a stop. The car is described by a high-dimensional bicycle model [LDM15, Eq. (1)], plus uncertainty to accommodate nonlinearities and modeling error resulting from gear shifts, tire forces, and the hundreds of states that are modeled in CarSim. Importantly, RTD is able to navigate the test track safely while planning in real time, which RRT and NMPC fail to do. However, this paper and all the previous work only consider static environments.

1.2.2.4 Planning in Dynamic Environments

We extended RTD to dynamic environments in [VKL+19], as shown in Figure 1.2. This work introduces the notion of fault to RTD, and provides a method for provably not-at-fault planning. Note, since RTD is only concerned with planning, not perception or prediction, we assume predictions are handed to the planner. The method is shown in simulation and hardware on the Segway, and on a carlike Electric Vehicle (EV). It outperforms a state lattice planner [McN11] in terms of safety and performance, by reaching desired goal locations more often, without causing any at-fault collisions. Unfortunately, this method uses a time discretization to represent moving obstacles, which adds conservatism to ensure not-at-fault planning, resulting in low average speeds and difficulty planning around many obstacles.

We enabled faster planning in more realistic and complex dynamic environments in [VLK+19]. This application of RTD uses a time partition instead of a time discretization, by computing an FRS over several time intervals, and treating predictions of obstacles as static within each interval. This approach drastically reduces the number of constraints required to represent unsafe plans at runtime, and reduces conservatism by removing the need to buffer obstacles to compensate for time discretization. Consequently, RTD is able to plan for the EV to drive at higher speeds (up to 7 m/s, whereas [VKL+19] could only plan up to 3 m/s). In addition, the EV is able to successfully traverse realistic scenarios, such as unsignaled, crowded four-way intersections.

1.2.2.5 Extensions Beyond Wheeled Robots

All of the previous work only considered wheeled mobile robots, represented in the plane. We have addressed this in two ways.

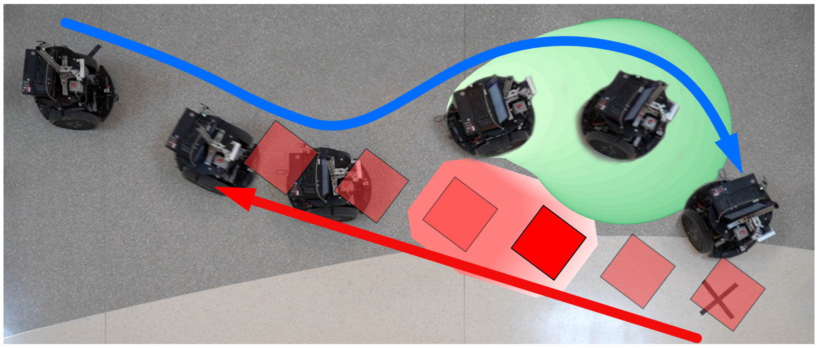


Figure 1.2: A bird's eye view shows RTD planning in a dynamic environment [VKL+19]. Here, the Segway robot moves from left to right, and dodges a red, box-shaped virtual obstacle moving from right to left. The blue arrow shows the Segway's trajectory, and the red arrow shows the obstacle's trajectory; both arrows are offset from Segway/obstacle for visual clarity. At one time instance, we see the Segway's time-varying reachable set as a green pear shape, and the prediction of the obstacle's motion as a light red set; both of these shapes fade from light to dark to indicate the flow of time.

We extended RTD to 3-D, for drones in static environments, in [KHV19]. This paper also provides a novel method for computing the parameterized FRS, by using zonotope reachability instead of SOS programming. Furthermore, this paper specifies a physics-based method for computing a quadrotor's tracking error with respect to parameterized trajectories; this was necessary due to the 22-dimensional space describing the quadrotor and parameterized trajectories, where the previous work never required sampling in more than 3 dimensions to compute tracking error.

We extended RTD to manipulator arms in [HKZ+20], as shown in Figure 1.3. This paper generalizes the zonotope reachability developed in [KHV19] for redundant manipulators, and enables planning with respect to arbitrary polytopic obstacles. RTD outperforms vanilla CHOMP [ZRD+13] on a variety of planning problems with varying difficulty in simulation, and is able to solve real-time planning problems on hardware. Notably, RTD is able to respond safely to the sudden appearance of an obstacle in front of the arm while it is in motion.

1.3 Dissertation Organization

The remainder of this document is organized as follows:

§2 We review the relevant literature.

§3 We develop a generic theoretical framework for RTD; in particular, we formally specify notions of safety and fault, and show how reachable sets can be used to formulate safe motion planning.

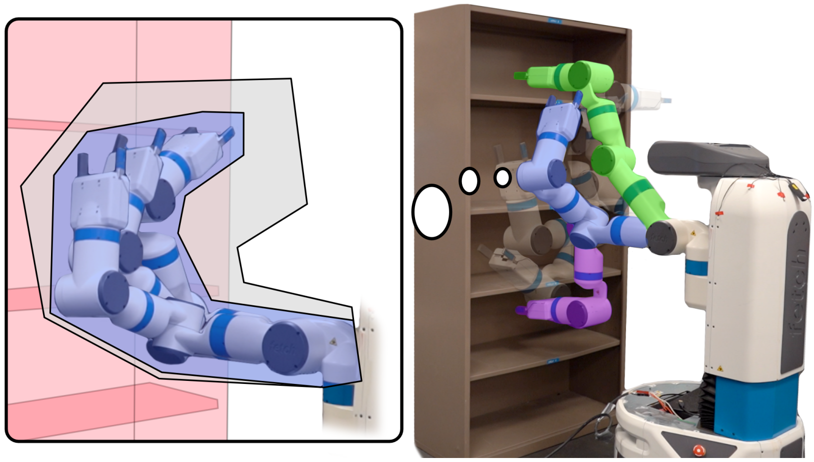


Figure 1.3: The Fetch robot's manipulator arm uses RTD to plan from a start pose (purple on a low shelf) to a goal pose (green on a high shelf) around a cabinet [HKZ+20]. The transparent arms show intermediate poses planned by RTD. One particular pose is shown in blue, with a callout on the left, to demonstrate how RTD sees its environment and plans. In the callout, the grey volume is the arm's reachable set of all possible trajectories in the given receding-horizon planning iteration. The blue volume, with several time steps shown, is the reachable set for the particular choice of trajectory parameters in the particular iteration; this blue volume is guaranteed to not intersect with the cabinet (light red), since RTD is able to provably generate collision-free trajectory plans.

§4 To implement the theory from §3, we develop an offline sums-of-squares polynomial approach to compute the robot's Forward Reachable Set (FRS) as a polynomial, and show how to use this polynomial at runtime to enable provably-safe motion planning.

§5 We specify a novel discretized obstacle representation for arbitrary planar (i.e., wheeled) robots that enables safe and real-time planning with the polynomial FRS.

§6 To extend RTD to robots outside the plane, we develop an FRS representation using zonotopes, a special type of convex polytope.

§7 We show how to incorporate tracking error into the polynomial and zonotope FRSes.

§8 We introduce rotatotopes, an extension of zonotopes that make RTD tractable for multi-link robots such as manipulators, whereas the polynomial and zonotope methods were restricted to single rigid-body robots.

§9 We demonstrate RTD on wheeled, aerial, and manipulator robots in simulation and hardware.

§10 We provide concluding remarks and future research directions.

1.4 Notation

We use the following notation throughout this work. Generic notation is summarized in Table 1.1. RTD-specific notation is summarized in Table 1.1.

Scalars and vectors, and functions that output them, are lowercase italic (e.g., a point x); the exception is Δ, used to denote a positive scalar. Sets and arrays/matrices, and functions that output them, are uppercase italic (e.g., a space X). Subscripts denote contextual information (e.g., a point $x_{\text{hi}} \in X_{\text{hi}}$ denotes the state in a robot's high-fidelity model). Superscripts in parentheses denote indices (e.g., a point $x^{(1)} \in \{x^{(i)}\}_{i=1}^n$. Exponents are in superscripts without parenthesis (e.g., x^2). The set containing the point x is $\{x\}$. We round a real number up (resp. down) to the nearest integer with $\lceil x \rceil$ (resp. $\lfloor x \rfloor$). If f is a function its preimage (or inverse, if the inverse exists) is f^{-1}. If A is a set, its power set is pow (A), its interior is interior(A), its complement is A^{C}, and its boundary is ∂A.

Category	**Symbol**	**Meaning**
spaces	$\mathbb{N}$	natural numbers
	$\mathbb{R}^n$	n-dimensional Euclidean space with $n \in \mathbb{N}$
	$\mathbb{S}^n$	n-dimensional unit sphere
	$\mathsf{SO}(n)/\mathsf{SE}(n)$	special orthogonal/Euclidean group associated with $\mathbb{R}^n$
scalars, vectors, and sets	$\{x^{(i)}\}_{i=1}^n$	superscripts denote indices
	x_*	subscripts denote contextual information
	f^{-1}	preimage (or inverse, if it exists) of a function f
	$\{x\}$	the set containing the point x
	pow (A)	the power set of a set A
	interior(A)	the interior A
	A^{C}	the complement of A
	∂A	the boundary of A

Table 1.1: Notation used throughout this work.

Category	**Symbol**	**Meaning**
timing	T	time, $T = [0, \infty)$
	$T^{(i)}$	time horizon of i^{th} planning iteration, $T^{(i)} \subset T$
	$t^{(i)}$	time at beginning of i^{th} planning iteration
	t_{f}	duration of each plan (i.e., length of each $T^{(i)}$)
	t_{plan}	planning timeout
state	Q	generalized coordinate space (i.e., configuration space)
	$\dot{Q}$	generalized velocity space ($\cong$ tangent space to Q)
	X_{hi}	high-fidelity model state
	f_{hi}	high-fidelity model, $f : T \times X_{\text{hi}} \times U \to \mathbb{R}^{n_{\text{hi}}}$
	x_{hi}	high-fidelity model state or trajectory
	X	planning model state
	f	planning model, $f : T \times X \times K \to \mathbb{R}^{n_X}$
	x	planning model state or trajectory
	p/v/a	position / velocity / acceleration
	θ/ω	heading / yaw rate
control inputs	U	control input space
	u	control input $u \in U$ or input signal $u : T \to U$
	u_k	feedback controller for plan $k \in K$
trajectory parameters	K	trajectory parameter space
	$K^{(i)}_{\text{unsf}}$	unsafe trajectory parameters in the i^{th} iteration
	k	generic trajectory parameter $k \in K$, also called a plan
	$k^{(i)}$	trajectory plan for the i^{th} planning iteration
workspace	W	workspace, $\mathbb{R}^2$ or $\mathbb{R}^3$
	O	obstacle, $O : T \to \text{pow}\,(W)$
	$\mathcal{P}^{(i)}$	prediction in i^{th} planning iteration, $\mathcal{P} : T^{(i)} \to \text{pow}\,(W)$
reachable sets	$\mathcal{R}_{\text{FRS}}$	Forward Reachable Set (FRS)
	$\mathcal{R}_{\text{plan}}$	Planning Reachable Set (PRS)
	$\mathcal{R}_{\text{err}}$	Error Reachable Set (ERS)
	$\mathcal{R}_{\text{obs}}$	Obstacle Reachable Set (ORS)
space sizes	n_Q / $n_{\dot{Q}}$	configuration / generalized velocity space size
	n_{hi}	high-fidelity model state space size
	n_X	planning model state space size
	n_U	control input space size
	n_K	trajectory parameter space size

Table 1.2: RTD-specific notation used throughout this work.

CHAPTER 2

Safe Motion Planning in the Literature

This chapter discusses the motion planning literature in four parts. We begin by discussing safety. Then, we explain how one can attempt to enforce safety at each tier of the receding-horizon planning hierarchy. Finally, we place RTD's contributions in the context of the literature.

2.1 Safety

2.1.1 Defining Safety

This work is concerned with **safe** receding-horizon planning. Safety means not colliding with obstacles, which can be static objects or other dynamic actors. Since safety is not always possible to enforce in dynamic environments, we must also consider **fault**, in the sense that the robot should be not-at-fault if a collision does occur with a dynamic obstacle [VKL+19, VLK+19, CSW+19, SSSS17].

To simplify the exposition, we only use the term "safety" for the remainder of this chapter, with the implication that safety encompasses not-at-fault behavior in dynamic environments.

2.1.2 Enforcing Safety in the Planning Hierarchy

Recall that receding-horizon planning is broken into a three-tiered planning hierarchy in §1.1.3. A path planner generates a coarse path that is handed to a trajectory planner, which obeys the dynamics of the robot and outputs a trajectory that is tracked by a tracking controller.

One could attempt to enforce safety at every tier of the planning hierarchy, but this may be unnecessarily conservative, which correlates with reduced performance. To avoid such conservatism, one can enforce safety at one tier, as long as one shows that this encompasses the behavior of the other tiers. This leads to three **safety paradigms**: (1) safety in the path planner, (2) safety in the trajectory planner, and (3) safety in the tracking controller. The proposed RTD method is in paradigm 2, meaning that it enforces safety in the trajectory planner.

Next, we discuss path planners, trajectory planners, and tracking controllers in terms of both safety and performance. We then summarize the challenges in the literature.

2.2 Path Planners

In path planning, one attempts to find a path (i.e., a connected curve) in the robot's **configuration space** from a start pose to a goal pose [LaV06]. In terms of safety, every point on the path should be collision free, but the path itself need not obey any dynamic model of the robot, since it is typically not parameterized by time (including time parameterization is the role of the trajectory planner). Ignoring time and dynamics allows one to reduce the computational effort required for generating the path, and instead spend that effort on **collision checking**; that is, checking if the points on the path are collision-free. Indeed, collision checking is the primary challenge in making these methods effective for real-time, safe planning.

A variety of methods exist for path planning, which can broadly be separated into **sample-and-check** methods and **gradient-based** methods. In this section, we discuss these two classes of methods in terms of advantages and disadvantages. We then briefly discuss collision checking.

2.2.1 Sample-and-Check Methods

Sample-and-check methods generate paths, as the name suggests, by iteratively sampling points in configuration space and collision checking the paths connecting the samples to nearby points. This approach allows one to build a family of paths as a discrete graph, for which choosing a particular path can be done quickly using, e.g., Dijkstra's algorithm. The speed and effectiveness of representing paths with discrete graphs has been well studied for decades, at least since the publication of the A* algorithm [HNR68].

We now present a few representative examples of sample-and-check methods. Perhaps the most well-known sample-and-check methods are Rapidly-exploring Random Trees (RRT) [LKJ01] and Probabilistic RoadMaps (PRM) [KSLO96]. Both of these admit "complete" versions, RRT* and PRM*, which are certified to eventually find an optimal collision-free path if one exists [KF11]. They can also be extended to dynamic environments [OF15] in a receding-horizon way [Hau12]. Since collision checking is computationally expensive, one can take a "lazy" approach to perform as few collision checks as possible [BK00, MSS18], which can result in rapid planning if one precomputes a PRM as though no collisions exist [KRSV10, MFJQ$^+$16]. To split the difference between sampling at runtime and sampling offline, one can use precomputed edges, based on a dynamic model of a robot, to build a graph of paths online; this approach is called a state lattice [PKK09, McN11].

The advantage of sample-and-check methods is that they transform the search for a continuous, connected path of configurations into a search on a discrete graph, which can be solved rapidly on a computer. However, achieving this requires discretizing a path, which makes it difficult to certify if the continuous path between discrete points is actually collision free; furthermore, depending on how one represents the continuous paths between points, it may not be possible to certify that a safe path can be transformed into a safe trajectory (i.e., the robot can typically not perfectly track a path).

2.2.2 Gradient-Based Methods

Gradient-based methods use the gradient of the distance from the robot to obstacles, for a sequence of configurations, to "push" the path out of collision. A classical example is the potential field method [BL91, War89], which uses gradient information to improve the quality of a graph that is built much in the same way as sample-and-check methods. Another classical example is the elastic band method [QK93], which attempts to bridge the gap between path and trajectory planning by smoothing a given path with gradient information. These classical methods are impressive first efforts to solve difficult planning problems; however, they do not make safety guarantees.

Recent gradient-based methods, such as CHOMP [ZRD$^+$13], TrajOpt [SDH$^+$14], and ITOMP [PPM12], expand on the idea of elastic bands with nonlinear optimization, inspired by optimal control. By initializing with an entire path from start to goal, these methods have the potential to find paths more quickly than sampling-based methods. Furthermore, by including path smoothness as an optimization cost, these methods can produce paths that are nearly dynamically-feasible, simplifying the subsequent step of trajectory planning. However, they can often converge to infeasible (i.e., unsafe) solutions, because a robot's environment is typically non-convex. These methods rely on finite differencing [QK93, ZRD$^+$13] or linearization [SDH$^+$14] to compute the gradient of the distance from a robot to obstacles, since it is difficult to represent the gradient analytically. Therefore, they make a tradeoff between safety and performance depending upon the fineness of path discretization. That is, they must choose between a faithful robot representation and computation speed. Methods exist to use the geometry and kinematics of a robot to conservatively produce swept-volumes and reduce the impact of coarse discretization [LaV06, SDH$^+$14]. However, the recent methods [ZRD$^+$13, SDH$^+$14, PPM12] also rely on penalizing collision avoidance in the nonlinear optimization cost (as opposed to using hard constraints), meaning that they cannot provably guarantee safety; that is, any solution to the nonlinear optimization must be validated by an external collision-checker [CSCC14, C$^+$13].

2.2.3 Collision Checking

A wide variety of methods exist to check a path for collisions, and can be applied to either sample-based or gradient-based methods, with the caveat that gradient-based methods typically need the distance from the robot to obstacles, as opposed to just a binary value.

The most common collision-checking approach is to consider discrete points along a path, and buffer obstacles to compensate for this discretization; for each discrete configuration, one checks if the corresponding robot volume in workspace intersects with the buffered obstacles [LaV06]. Alternatively, in the workspace of the robot, one can fit a convex hull around pairs of such discrete points, and collision check the hull to compensate for the robot's motion between the discrete points [SDH$^+$14]. Extensive work has also gone into continuous collision checking, wherein one checks an entire continuous (connected) path. For example, in the plane, one can fit polynomial splines to the motion of a robot's body through space, then collision check discrete points along these splines, then use the robot's body geometry and the curves' polynomial structure to detect collision points between the discretized points [YLJS18]; however this means one must spend tens of milliseconds checking a single path, as opposed to under a millisecond per discrete point. Similarly, in a 3-D workspace, one can represent a robot's swept volume along a path as a sphere swept along a surface, and conservatively represent this surface as a mesh [RLMK04]; one can make this method run in tens of milliseconds by applying a hierarchy of progressively more accurate (but conservative) representations of the robot to cull collision-free areas of the workspace, and thereby identify a time of first contact. This technique can be applied to high degree-of-freedom (DOF) robots and probabilistic environments while still providing collision checks in tens of milliseconds with parallelization [PPM20]. Finally, one can instead attempt to identify *velocity* obstacles, or choices of velocity through workspace that can cause a collision [VDBGLM11]; when robots are represented as collections of spheres, this can identify unsafe workspace velocities in microseconds per rigid body.

Unfortunately, each of these methods assumes that the robot can perfectly track a given path. When this is not the case, one must apply a heuristic to buffer, or dilate, obstacles in the workspace to account for tracking error or the robot's dynamics [LaV06]. On the other hand, if one considers the robot's dynamics directly when generating paths (i.e., by generating trajectories), one pays a computational penalty for increasing modeling accuracy.

2.2.4 Path Planner Summary

To summarize, path planners can often rapidly find paths to accomplish a task. However, since they use approximations such as discretization, finite differencing, and linearization, they typically do not certify safety, and instead only attempt to achieve high performance. Furthermore, it is often not possible to directly treat a collision-free path as a trajectory, meaning that any safety guarantees made by a path planner may not certify safety for the actual robot. In addition, since path planners do not produce time-parameterized plans, they can only attempt to certify collision avoidance in dynamic environments by conservatively treating the motion of any other dynamic actor as a single, static obstacle.

2.3 Trajectory Planners

In trajectory planning, one takes a given path and attempts to produce a trajectory to track the path. A trajectory should typically be **dynamically-feasible** with respect to a dynamic model of the robot. As noted above, collision-free paths may not be dynamically feasible if converted directly into trajectories. Therefore, trajectory planners must also perform collision checking. Furthermore, to guarantee safety, trajectory planners must account for **tracking error**, meaning a nonzero difference between a robot's actual state and the desired state in a trajectory plan. Tracking error arises from uncertainty such as model error and measurement noise, and results in a robot's inability to perfectly track a trajectory plan [MT16, KVB+20, BPA17]

As with path planners, we can broadly divide trajectory planners into **sample-and-check** and **gradient-based** methods. Note that RTD is most closely related to the gradient-based methods. We save all discussion of RTD for §2.5, after presenting the challenges of the existing methods in the literature.

2.3.1 Sample-and-Check Methods

Sample-and-check trajectory planners attempt to find a single trajectory by choosing samples and checking them for collision. These methods have been used for a variety of robots, such as autonomous cars [KFT+08] and quadrotor drones [MHD15, RBR16, MT16]. They typically rely on a path given by a path planner, so they are not concerned with building a graph throughout a robot's configuration space. In particular, a path planner provides waypoints for navigation, which can either be specified *a priori* (such as for autonomous driving on structured roads) or found by, e.g., an RRT before performing trajectory planning (for drones in arbitrary environments). This means the trajectory planner need only find a trajectory that satisfies a differential equation representing the robot in a **state space**.

Sample-and-check trajectory planners often do not directly address safety. Some approaches partially relegate safety to the path planner [MHD15, RBR16]; that is, the methods trust that the path planner actually has found a collision-free path, so a collision-free trajectory can be found by staying as close to the path as possible while incorporating the robot's dynamics. Others ensure that trajectories lie inside safe sets, which can be semi-algebraic [MT16] or polytopic [CSS15, CLS16, TLEH20]. For manipulator arms, trajectory planners typically relegate safety to the path planner, since collision checking is computationally expensive [PJ87, KS12]. However, since path plans cannot necessarily be tracked perfectly, some recent approaches generate and collision-check both a trajectory along a path and a separate fail-safe trajectory (braking to a stop) [AGLP19, PA15].

Sample-and-check trajectory planners can find solutions quickly when trajectories can be parameterized or specified *a priori*, and when attempting to track paths that are not near obstacles. However, since they do not make use of gradient information, it can be difficult for them to plan when many or all samples are in collision (such as may happen when near obstacles).

2.3.2 Gradient-Based Methods

Gradient-based trajectory planners, much like gradient-based path planners, represent a robot's trajectory as a sequence of discrete points, and use the gradient of each discrete point with respect to the decision variable of an optimization program to "push" the discrete points out of collision. However, these trajectory planners include additional constraints so that the discrete points along the trajectory are linked by forward-integrating the robot's dynamic (state space) model. The decision variables are therefore typically specified as the control inputs at each discrete point in time.

By the definition stated above, gradient-based trajectory planners are in fact an application of model predictive control (MPC) [KQCD15, BM99]. Therefore, they often combine the roles of trajectory planner and tracking controller. When a robot is described by linear dynamics, or the dynamics are linearized along the reference trajectory, one can formulate trajectory planning as a quadratic program, which can be solved quickly (assuming feasible solutions exist) [WB09]. Indeed, in the linear case (with convex constraints), robustness to disturbance (which can be used to formulate safety guarantees) is well-studied [BM99, VSG$^+$12]. Unfortunately, robots are typically described with nonlinear dynamics, but Nonlinear MPC (NMPC) cannot make the same rapid convergence guarantees due to solving a nonlinear program [PR14, KQCD15, KVB$^+$20]. For some robots (notably autonomous cars), a variety of methods leverage road structure to convexify the MPC problem, enabling safety guarantees in specific contexts [VSG$^+$12, PKA19].

2.3.3 Trajectory Planner Summary

To summarize, enforcing safety in the trajectory planner shares the challenge of collision checking with safety in the path planner (see §2.2.3). However, since the trajectory planner considers the dynamics of the robot, it is possible to consider tracking error, and to subsume the role of the tracking controller via MPC, making strict safety guarantees more tractable. The challenge, then, is to represent the nonlinear dynamics of the robot in a way that enables real-time, safe planning despite having to solve a nonlinear program. Some methods achieve this by leveraging the structure of the robot's environment, but, to the best of our knowledge, no general method exists that can guarantee safety without sacrificing performance.

2.4 Tracking Controllers

A tracking controller takes in a trajectory plan (often with associated nominal inputs) and the robot's current state, and generates a control input to drive the robot along the plan. Note, since the tracking controller directly generates actuator inputs, it typically operates at a much higher rate than the path planner or trajectory planner. For example, an MPC controller may operate on the order of hundreds of Hertz [WB09], as opposed to often under ten Hertz for path or trajectory planning [KVB+20].

Recall that robots experience **tracking error** due to uncertainty from sources such as imperfect sensors and imperfect models. To enforce safety, a tracking controller must bound tracking error in a robot's position. Based on how they ensure safety, we can broadly divide tracking controllers into two categories: **invariant set** methods and **reachable set** methods. Note that these two categories are similar in that invariant sets and reachable sets can often be computed using the same numerical methods [MBT05, SA19]. Also note that there are many other types of controllers, some of which attempt to enforce strict bounds given, e.g., bounded uncertainty. Here, we limit the scope of this discussion to what we feel are the most relevant controllers to this work.

2.4.1 Invariant Set Methods

An **invariant set** is a set of states within which a system will remain for all time [KMO+12]. We discuss two methods that use invariant sets: Control Barrier Functions (CBFs) and Hamilton-Jacobi (HJ) reachability analysis.

If one can specify a safe set *a priori*, then CBFs provide a method for certifying that the set is invariant, by synthesizing a controller that maintains that invariance [ACE+19]. CBFs have been applied to cases such as automotive lane keeping and cruise control [XGTA17], low-speed robots in crowds [CPG17], planar quadrotor flight [WS16], and manipulator trajectory tracking [SNGA19]; see [ACE+19] for many more examples. CBFs have the advantage of formulating control synthesis as a quadratic program, which can solve quickly online (if there exists a feasible solution). In addition, CBFs are agnostic to the method used to generate trajectories, and can therefore enforce safety for potentially-unsafe trajectory plans. However, these approaches compute a safety-enforcing control input only for the current time instant (i.e., they do not consider the future of a planned trajectory); this can cause a robot to behave conservatively, because it must maintain safety with respect to any possible future state that results from the current input. That is to say, this tracking controller method treats all higher-level planners as a disturbance.

HJ reachability analysis can be used to conservatively identify an invariant set of tracking error, by modeling the relative state between a robot's model and a (typically simpler) trajectory planning model [HCH+17]. The relative dynamic model is treated as a differential game, and solve the resulting partial differential equation (PDE) by gridding the relative state and control spaces [MBT05]. This approach can be used to synthesize a controller that maintains invariance, if one selects the robot model and reference trajectory model appropriately [HCH+17]. HJ reachability has been applied to control synthesis for quadrotors [HCH+17, CHV+18] and low-speed wheeled robots [BBB+19]. Much like CBFs, HJ approaches are agnostic to the trajectory planner, and can enforce safety given unsafe inputs. However, these approaches also treat all higher-level planners as a disturbance, leading to conservatism. Furthermore, since the synthesized controller is represented on a discrete grid, safety guarantees only hold in the limit as the grid spacing approaches zero [MBT05, Section III-A]. Luckily, due to the formulation as a differential game (in which the trajectory planning model attempts to escape from the robot), approximate solutions to the HJ PDE are usually sufficiently conservative to compensate for the gridding approach in practice. Next, we discuss reachable set tracking controllers.

2.4.2 Reachable Set Methods

A **reachable set** is the set of all states reached by a trajectory, or family of trajectories, under a particular control policy. Reachability analysis, as introduced in §1.1.4, is the framework used to compute reachable sets. We discuss two reachable set methods that are used for safe control: MPC (Model-Predictive Control), and Sums-of-Squares (SOS) programming.

Before discussing MPC as a tracking controller, recall that MPC often combines the roles of trajectory planning and tracking controller, as noted in §2.3. However, there is utility in maintaining separate roles, because nonlinear and robust MPC methods typically find a feasible solution more quickly (if one exists) when provided with a better initial guess at a solution [PR14, KQCD15, KVB+20]. In other words, if one spends computational effort on trajectory planning via, e.g., a sample-and-check approach, then it may be possible to compute inputs for trajectory tracking at a high rate using MPC.

MPC can be used to render a reachable set of tracking error invariant for some nonlinear systems [BAC06, YMCA13, BM99]; note, one must compensate for the requisite time discretization. A key advantage of MPC is that it produces a sequence of control inputs in a receding-horizon manner, as opposed to picking a control input only for the current time instant, or treating higher-level planners as a disturbance. This enables a less conservative choice of control inputs (with respect to ensuring safety), because the controller does not need to anticipate every possible future state that could result from the current control input. However, since MPC must discretize the trajectory, it has the same tradeoff between safety and performance that we saw for all path and trajectory planners that rely on discretization.

SOS programming can be used to synthesize polynomial feedback controllers for controlled polynomial systems via (backwards) reachability analysis. In particular, given a reference trajectory, one can conservatively approximate all initial conditions (and compute associated feedback controllers) that converge to within some distance of the trajectory, within some finite amount of time [TMTR10, SVBT14]. This approach has been applied to a variety of mobile robots, such as ground and aerial vehicles [MT16, SVBT14]. Note that [SA19] computes invariant sets for bipedal robots using SOS programming; we mention it here to show the connection between invariant set and reachable set methods.

The advantages and disadvantages of the SOS approach are as follows. Much like MPC, the SOS approach considers the entire duration of a reference trajectory, which reduces conservatism. Furthermore, since the dynamics and controller are polynomials, this method does not always need to discretize in time, resulting in safety guarantees without additional buffering of obstacles to compensate for discretization (note, time discrezation is used in [MT16]). Unfortunately, it is difficult to compute such feedback controllers for systems with more than three or four states due to the size of the semidefinite program (SDP) representation typically used to solve the SOS program [Las10]. Therefore, it can be difficult to certify robustness to some types of uncertainty, since adding parameteric uncertainty requires including parameters as additional dimensions [HKMV16]. Similarly, one may need to add dimensions to tolerate arbitrary trajectory plan inputs [SCH$^+$18, SYA19], so some existing SOS approaches instead require fixed, pre-specified trajectory plans to make the reachability analysis tractable [MT16].

2.4.3 Tracking Controller Summary

To summarize, enforcing safety in the tracking controller requires a tradeoff between safety and performance, just as with path planners and trajectory planners. For MPC approaches, this occurs due to approximations used to represent a trajectory plan and a robot's dynamics, which may prevent one from certifying safety for the original system. CBF, HJ, and SOS approaches, on the other hand, can provably certify safety, but may produce conservative behavior. In particular, CBF and HJ approaches treat higher-level planners as a disturbance, which produces additional conservatism. For SOS approaches, the memory required to compute the tracking controller increases as the conservatism decreases, limiting these approaches to low-dimensional system representations.

It is similarly difficult to compute CBFs or perform HJ reachability on high-dimensional systems.

2.5 RTD in Context

We now place RTD in the context of the literature. First, we revisit the research gap that RTD addresses at a high level. Second, we summarize the method to illustrate how it relates to the literature. Third, we comment on the generality of RTD. Fourth, we discuss how RTD performs (and provides novelty in) collision checking.

2.5.1 Research Gap Revisited

As mentioned before, the key challenge in robot motion planning is to enforce safety without sacrificing performance. This challenge appears because robots are typically described by high-dimensional, nonlinear models, which are difficult to use for real-time planning while making guarantees. Approximations such as time discretization of a plan, linearization of a robot's model, and not including the robot's dynamics, all can enable rapid planning. Unfortunately, each approximation made for the sake of performance introduces modeling error, which manifests as tracking error when the robot attempts to execute any given plan. It is challenging to provably account for tracking error at any tier of the planning hierarchy without incurring severe conservatism. In short, the **research gap** is to produce a safe, real-time, receding-horizon planning algorithm that can operate in arbitrary environments.

2.5.2 Method Summary

RTD directly addresses the aforementioned research gap: it is a trajectory planner that performs safe, real-time, receding-horizon planning in arbitrary environments. Enforcing safety at the trajectory planning tier allows RTD to incorporate a robot's dynamics (unlike many path planners), and the time evolution of tracking error (unlike many tracking controllers).

RTD uses reachable sets, which we noted throughout §2.4 can be conservative and difficult to apply to high-dimensional systems; in other words, we would expect a reachable set method to suffer in terms of performance to ensure safety. However, by using reachable sets for trajectory planning, instead of tracking, RTD is able to certify safety while achieving performance that rivals or exceeds other trajectory planning methods. This dissertation demonstrates such performance in later chapters.

RTD begins with offline modeling and reachability analysis. RTD uses a **high-fidelity model** to describe a robot's behavior, and a user-specified **planning model** to generate plans for the robot.

The planning model uses **trajectory parameters**, drawn from a compact set, to produce trajectories of finite duration. Importantly, every parameterized trajectory ends with a fail-safe maneuver; in this work, this maneuver is braking to a stop. By specifying that the planning model only produces bounded trajectories, and by requiring the high-fidelity model has bounded dynamics for a user-specified **tracking controller**, one can bound the **tracking error**. RTD then computes a **Forward Reachable Set** (FRS) conservatively for all parameterized trajectories of the planning model, plus tracking error, thereby containing all states reachable by the robot itself.

Online, RTD performs trajectory optimization in each receding-horizon planning iteration. First, the FRS is used to project obstacles from the robot's position states into the space of trajectory parameters, thereby identifying the set of unsafe parameters for the current iteration. Second, RTD performs trajectory optimization over the safe parameters, while enforcing a time limit on planning; if it can find a new trajectory plan within the time limit, then RTD passes that plan to the tracking controller. Otherwise, the robot continues executing its previous plan, which ends with a fail-safe maneuver.

2.5.3 Flexibility of RTD

RTD enforces safety at the trajectory planning tier of the planning hierarchy, which enables flexibility in the choice of path planner and tracking controller. RTD is agnostic to unsafe paths produced by a path planner, which frees the path planner from needing perfect collision checking. This approach enables RTD to achieve good performance with respect to navigating a robot through a cluttered environment, because the path planner can plan quickly without concern for safety. RTD is also agnostic to the type of tracking controller used, as long as one can upper bound the tracking error produced by that controller with respect to RTD's parameterized trajectories (note, one can trivially satisfy this bound by choosing a large tracking error amount, thereby incurring conservatism without sacrificing safety). This approach gives the user a choice in how conservatively RTD behaves, since one can use a smaller bound as soon as one designs a better tracking controller.

2.5.4 Collision Checking

Recall the variety of collision-checking methods discussed in §2.2.3. The key challenge is the tradeoff between accuracy and computation speed; indeed, to consider continuous time collision checking, or a robot's dynamics and uncertainty, one typically must allot more computation time. RTD addresses this tradeoff in two ways. First, by using the FRS, we generate continuous-time swept volumes of the robot that can be used for collision checking, thereby avoiding the challenge of choosing a discretization fineness; furthermore, the FRS can contain the motion of the robot subject to its dynamics, not just a kinematic model. Second, for planar robots, we prescribe an *ob-*

stacle discretization in §5 that enables provably-conservative continuous time and space collision checking.

Furthermore, recall that, for certain cases of robot representations and kinematics, one can identify *velocity* obstacles [VDBGLM11]. By identifying *trajectory parameter* obstacles using the FRS, RTD enables a generalization of this notion. For example, if one parameterizes a robot's velocity, then the FRS enables one to identify velocity obstacles. If one uses a more complex parameterization, such as time-varying velocity profiles, then RTD instead find velocity profile obstacles.

2.6 Chapter Review

In this chapter, we discussed the relevant literature in motion planning. In particular, we presented how one can attempt to enforce safety at each of the tiers of the planning hierarchy. We identified challenges, and discussed how RTD addresses these challenges.

The challenges, in short, are as follows. First, there is a tradeoff between the accuracy of describing the robot, and the ease of performing real-time motion planning. Furthermore, while a variety of methods exist to consider tracking error, they are often conservative, or require applying a heuristic that prevents formal safety guarantees to enable real-time performance. Finally, for path and trajectory planning, collision checking is the underlying cause of computational expense.

RTD addresses these challenges in four ways by leveraging reachable sets. First, by representing the robot continuously in time and space, RTD's reachable sets avoid the tradeoff between discretization fineness and performance. Second, by incorporating tracking error in the reachable sets, RTD enables compensating for the robot's dynamics, as opposed to requiring one to assume a kinematic model for collision checking. Third, by parameterizing trajectories in the reachable sets, RTD enables a generalization of velocity space obstacles to parameter space obstacles. Fourth, as is shown in the following chapters, RTD prescribes obstacle representations that allow for continuous time and space collision checking which can be used for real-time planning, and to generate collision-avoidance constraints with gradients.

This concludes the literature review. Next, we present a theoretical overview of RTD.

CHAPTER 3

A Unified Theoretical Framework for Safe Trajectory Planning

This chapter provides a theoretical overview of RTD. That is, we present and discuss mathematical objects and operations independent of how they are implemented or represented numerically. This allows us to broadly unify the various applications of RTD across different robot morphologies. We begin with a summary to provide a roadmap for the chapter. We then step through each part in more detail, to introduce logic and notation.

3.1 Chapter Summary

This chapter unifies all of the currently-published RTD papers into a single underlying theory. For the interested reader, the specific papers are [KVJRV17, KVB$^+$20, VKL$^+$19, VSK$^+$19, VLK$^+$19] for wheeled robots, [KHV19] for aerial robots, and [HKZ$^+$20] for manipulators.

Given a robot, the overall goal of RTD is to generate safe trajectory plans in real time. We do so by using reachable sets, computed offline, to describe a continuum of trajectory plans. At runtime, we choose one trajectory out of the continuum of plans in a receding-horizon fashion; that is, the robot picks one plan, then attempts to pick a new plan while tracking its previous plan. To pick plans, RTD uses an optimization formulation. To ensure these plans are collision-free, RTD uses reachable sets to identify the set of unsafe plans in any planning iteration, and then treats this unsafe set as a constraint for optimization. By repeatedly choosing safe plans, RTD enables the robot to be safe for all time. Note, as mentioned in the introduction, we consider a notion of fault in addition to safety when a robot is operating in dynamic environments, where it may be impossible to certify collision-free behavior.

The sections of this chapter are as follows. (§3.2) We begin by introducing the **high-fidelity model** that describes the robot's equations of motion; we also introduce the Segway robot as a running example for the chapter. (§3.3) Then, we explain how the high-fidelity model is used for

receding-horizon planning. (§3.4) Next, we introduce the robot's **workspace**, and explain how the robot and **obstacles** occupy volume, which allows us to define safety and fault in a collision; we also discuss how the robot **senses** and **predicts** obstacles. (§3.5) Since planning safe, not-at-fault trajectories with the high-fidelity model in real time is typically intractable, we introduce a simplified **planning model** that generates paramterized plans. (§3.6) We then introduce the **tracking controller** used to drive the high-fidelity model towards these parameterized plans, and discuss the resulting **tracking error**. (§3.7) To enable compensating for tracking error at runtime, we introduce the **Forward Reachable Set** (FRS), which contains the motion of the high-fidelity model tracking any parameterized plan; the FRS is computed offline. (§3.8) Finally, we discuss how the FRS is used to generate collision-avoidance constraints for online planning. (§3.9) We conclude the chapter with a brief summary.

3.2 The High-Fidelity Model

We now introduce the high-fidelity model used to describe the robot, and introduce the Segway robot as a running example to illustrate the various theoretical objects defined in this chapter. We then introduce a family of projection operators, which we use to relate the various subspaces that appear in motion planning. Finally, we discuss bounds on the robot's velocity and acceleration.

3.2.1 Time, States, Inputs, and the High-Fidelity Model

Let $T = [0, \infty) \subset \mathbb{R}$ represent time. Let $Q \subset \mathbb{R}^{n_Q}$ denote the **configuration space** of **generalized coordinates** for the robot. Let $\dot{Q} \subset \mathbb{R}^{n_{\dot{Q}}}$ denote **generalized velocities**. Let $X_{\text{hi}} = Q \times \dot{Q} \subset \mathbb{R}^{n_{\text{hi}}}$ denote the robot's **state space**, with the state denoted $x_{\text{hi}} = (q, \dot{q})$. Let $U \subset \mathbb{R}^{n_U}$ denote the space of **control inputs**.

The robot's equations of motion are given by a **high-fidelity model**, denoted $f_{\text{hi}} : T \times X_{\text{hi}} \times U \to F_{\text{hi}}$, for which

$$\dot{x}_{\text{hi}}(t) = f_{\text{hi}}(t, x_{\text{hi}}(t), u(\cdot)), \tag{3.1}$$

where $x_{\text{hi}} : T \to X_{\text{hi}}$ is a trajectory of the model with input $u(\cdot) \in U$. We require that F_{hi} and U are compact and f_{hi} is Lipschitz continuous on T, X_{hi}, and U. Note, we leave the domain of u ambiguous for now, but a typical example is a feedback controller $u : T \times X_{\text{hi}} \to U$. By this definition, for any compact subset of T and initial condition $x_{\text{hi},0} \in X_{\text{hi}}$, the trajectory x_{hi} (with input u) exists [KG02, Theorem 3.1]. We define such compact subsets of T below, in §3.3.

To facilitate understanding, we use the Segway as an example through the chapter:

Running Example 3.1. *The Segway robot can be described with generalized coordinates of its center-of-mass position and its heading* $(p_1, p_2, \theta) \in Q = \mathsf{SE}(2)$*, and generalized (longitudinal and angular) velocities,* $(v, \omega) \in \dot{Q} \subset \mathbb{R}^2$*. Note, we usually refer to* v *as just the "velocity." The high-fidelity model is a dynamic unicycle with control inputs for longitudinal and angular acceleration:*

$$f_{\text{hi}}(t, x_{\text{hi}}(t), u(\cdot)) = \begin{bmatrix} \dot{p}_1(t) \\ \dot{p}_2(t) \\ \dot{\theta}(t) \\ \dot{v}(t) \\ \dot{\omega}(t) \end{bmatrix} = \begin{bmatrix} v(t)\cos(\theta(t)) \\ v(t)\sin(\theta(t)) \\ \omega(t) \\ \text{sat}_v\left(\beta_v \cdot (u_v(\cdot) - v(t))\right) \\ \text{sat}_\omega\left(\beta_\omega \cdot (u_\omega(\cdot) - \omega(t))\right) \end{bmatrix}, \tag{3.2}$$

where $u = (u_v, u_\omega)$ *is typically a feedback controller that the robot uses to track planning trajectories (we provide an example of* u *later in this chapter). The functions* sat_v *and* sat_ω *saturate the accelerations. The constants* β_v *and* $\beta_\omega \in \mathbb{R}$ *are found using system identification.*

3.2.2 Projection Operators

We are often concerned with the position or velocity of the robot. To extract this information from an arbitrary state $x_{\text{hi}} \in X_{\text{hi}}$, we now introduce a generic family of operators to project to a subspace. Let S be any subspace of X_{hi}. We define the **projection operator** $\text{proj}_S : X_{\text{hi}} \to S$ that maps points from X_{hi} to S via the identity relation.

Running Example 3.2. *For the Segway, denote* $Q = P \times \Theta$*, with* $P = \mathbb{R}^2$ *for position and* $\Theta = \mathbb{S}^1$ *for heading. Similarly denote* $\dot{Q} = V \times \Omega$*, with* $V \subset \mathbb{R}$ *and* $\Omega \cong \mathbb{R}$ *(the tangent space of the unit circle). Then, if* $x_{\text{hi}} \in X_{\text{hi}}$ *is the Segway's state, its position is* $\text{proj}_P(x_{\text{hi}})$*, and its velocity is* $\text{proj}_V(x_{\text{hi}})$.

3.2.3 Maximum and Minimum Velocity and Acceleration

Notice that, since F_{hi} (the domain of f_{hi}) is compact, the robot's generalized velocity is bounded. Since the state space X_{hi} includes states for velocity and F_{hi} is compact, we further have that the robot's generalized accelerations are also bounded. Such acceleration bounds typically follow from the compactness of the control input space U (e.g., if the control inputs map to torques/accelerations).

We denote the robot's maximum (resp. minimum) generalized velocity as $\dot{q}_{\text{max}}$ (resp. $\dot{q}_{\text{min}}$) $\in \dot{Q}$. We denote the maximum (resp. minimum) generalized acceleration as $\ddot{q}_{\text{max}}$ (resp. $\ddot{q}_{\text{min}}$) $\in \mathbb{R}^{n_{\dot{Q}}}$ (that is, these bounds have the same dimension as $\dot{Q}$).

Note that these bounds are defined coordinate-wise, and do not incorporate state dependence. State-dependent limits appear in the model f_{hi}. For example, f_{hi} can represent the decrease in a wheeled robot's maximum possible yaw rate as a function of (increasing) speed in the plane.

Running Example 3.3. *The Segway's maximum velocity is* $v_{\text{max}} = \text{proj}_V(\dot{q}_{\text{max}})$.

3.3 Receding-Horizon Timing

Recall that RTD is a receding-horizon framework, meaning that the robot generates a plan of short duration, then generates a new plan while executing the current plan in an iterative manner. We now define plans and their timing (i.e., we explain what we mean by "short" duration).

For each i^{th} receding-horizon planning iteration, a **plan** is a trajectory $x_{\text{plan}}^{(i)} : T^{(i)} \to X_{\text{hi}}$, with

$$T^{(i)} = \left[t^{(i)}, t^{(i)} + t_{\text{f}}\right] \subset T, \text{ where} \tag{3.3}$$

$$t^{(i)} = (i-1) \cdot t_{\text{plan}}, \tag{3.4}$$

and $0 < t_{\text{plan}} < t_{\text{f}}$. We call t_{plan} the **timeout**, which is an amount of time within which the robot must find a new plan; so, the robot generates a new plan every t_{plan} seconds. We call t_{f} the **plan time horizon**, or duration of each plan. Note, $x_{\text{plan}}^{(i)}$ is *not necessarily* a trajectory of the high-fidelity model, but it is a trajectory in the high-fidelity model's state space. We clarify this notion after introducing a simplified planning model below.

In each i^{th} iteration, if the robot cannot find a new plan within t_{plan}, it must continue executing its previous plan, $x_{\text{plan}}^{(i-1)} : T^{(i-1)} \to X_{\text{hi}}$. Therefore, we assume there exists an initial plan $x_{\text{plan}}^{(0)} : [0, t_{\text{f}}] \to X_{\text{hi}}$. Typically, this initial plan is for the robot to stay stationary, so this assumption is not difficult to satisfy.

3.4 Workspace, Obstacles, and Sensing

Since RTD is concerned with collision avoidance, we now define the workspace, obstacles, and the robot's sensor behavior.

3.4.1 The Workspace and Forward Occupancy

The **workspace** $W \subseteq \mathbb{R}^2$ or $\mathbb{R}^3$ is the space in which the robot and other entities (such as obstacles or world boundaries) occupy volume.

Though W is not a subspace of X_{hi}, we define a special projection operator, $\text{proj}_W : X_{\text{hi}} \to W$, to return the robot's *center-of-mass* position in workspace for wheeled and aerial robots. For manipulators, this returns the position of the robot's baselink.

To define how the robot occupies volume, we use the following map. The **forward occupancy** map $\mathsf{FO} : X_{\text{hi}} \to \text{pow}\,(W)$ returns the subset of the workspace containing the volume of the robot at a state $x_{\text{hi}} \in X_{\text{hi}}$.

Running Example 3.4. *The Segway robot has a circular footprint; suppose it is of radius r. Then, at any state $x_{\text{hi}} \in X_{\text{hi}}$, its forward occupancy is given by*

$$\mathsf{FO}(x_{\text{hi}}) = \left\{ p \in W \mid \left\| \text{proj}_P(x_{\text{hi}}) - p \right\|_2 \leq r \right\}, \tag{3.5}$$

where P is the robot's position subspace and $W \subseteq \mathbb{R}^2$.

3.4.2 Obstacles, Safety, and Fault

We define an **obstacle** as a map $O : T \to \text{pow}\,(W)$. Suppose that, at time $t \in T$, the robot is at a state $x_{\text{hi}}(t) \in X_{\text{hi}}$. We say the robot is in **collision** if it intersects the obstacle, meaning $\mathsf{FO}(x_{\text{hi}}(t)) \cap O(t) \neq \emptyset$; so, **safe** means not in collision. Note, static obstacles are those for which $O(t_1) = O(t_2)$ for any $t_1, t_2 \in T$. We assume that no obstacle travels faster than some known quantity, $v_{\text{max,obs}} \geq 0$.

When obstacles are able to move, there are situations where it is impossible to avoid collision (e.g., an obstacle can move into our robot even if our robot is stationary). Therefore, we consider **fault** in a collision with a dynamic obstacle. In this work, our robot is **not-at-fault** if it is stationary, which is typically acceptable for, e.g., low-speed wheeled robots and collaborative manipulators. Note, we assume the robot can stay stopped indefinitely.

This definition of fault allows us to establish a general framework of safe and not-at-fault motion planning, without requiring us to model interactions between our robot and other agents (meaning, we can focus on planning without closing the planning/perception loop for now). Note that RTD is not limited to this definition of fault; but, we leave the extension to interaction modeling, and more general definitions of fault (e.g., [CSW$^+$19]), for future work.

3.4.3 Predictions and Sensing

The robot does not typically have direct access to information about obstacles for infinite time, so we instead consider predictions of obstacles in each planning iteration. Consider the i^{th} planning iteration. Suppose there are $n \in \mathbb{N}$ obstacles that the robot must consider for collision avoidance during $T^{(i)}$, denoted $O^{(j)}$, $j = 1, \cdots, n$. Then a **prediction** is a map $\mathcal{P}^{(i)} : T^{(i)} \to \mathrm{pow}\,(W)$ for which

$$\mathcal{P}^{(i)}(t) \supseteq \bigcup_{j=1}^{n} O^{(j)}(t). \tag{3.6}$$

Note, this definition requires predictions to be *correct* (they do contain the motion of all obstacles in the workspace within the time horizon $T^{(i)}$ and *conservative* (they may contain points that are not reached by any obstacle during $T^{(i)}$). This type of conservatism is also called a **buffer**, or dilation of the size of each obstacle. In the later chapters, for each robot morphology, we specify how to produce this buffer.

We now present a simplified notion of sensing obstacles; recall that this work is concerned with planning, not with perception. In particular, we define the **sensor horizon**, $d_{\text{sense}} > 0$, as a distance within which the robot can sense and predict obstacles. Suppose the robot is at a state $x_{\text{hi}}(t)$, and consider $D_{\text{sense}} : X_{\text{hi}} \to \mathrm{pow}\,(W)$ defined as

$$D_{\text{sense}}(x_{\text{hi}}(t)) = \left\{p \in W \mid \left\|\text{proj}_W(x_{\text{hi}}(t) - p)\right\|_2 \leq d_{\text{sense}}\right\}. \tag{3.7}$$

That is, D_{sense} returns a closed ball of radius d_{sense} about the robot's position in workspace. Suppose O is an obstacle; if $O(t) \cap D_{\text{sense}}(x_{\text{hi}}(t))$, we say that O is **sensed**. We assume that, at any $t \in T$, there is a finite number of sensed obstacles. We further assume that the robot can generate predictions of all sensed obstacles.

3.5 The Planning Model

To generate safe plans, one must consider obstacles and then generate a trajectory for the high-fidelity model within the timeout t_{plan}. Doing so can be intractable for complex high-fidelity models, so we instead use a simplified planning model, which generates parameterized plans.

In this section, we introduce the planning model, discuss the coordinate frame used for planning, and explain how to lift simplified planning model trajectories to the high-fidelity state space to enable full-state feedback control. Finally, we preview how our parameterized plans are used at runtime; the online planning procedure is detailed in §3.8.

3.5.1 The Planning Model

To define the planning model, we introduce the following spaces. Let $T_{\text{plan}} = [0, t_{\text{f}}] \subset \mathbb{R}$ be the **plan time horizon**, which is of duration t_{f}, just like each i^{th} planning iteration $T^{(i)}$. Let $X \subset \mathbb{R}^{n_X}$

($n_X \in \mathbb{N}$) be the **planning space**, which is a subspace of X_{hi}; typically, $X = Q$, but it can also include some or all of the states in $\dot{Q}$. Let $K \subset \mathbb{R}^{n_K}$ ($n_K \in \mathbb{N}$) be a space of **trajectory parameters**.

The **planning model** is $f : T_{\text{plan}} \times X \times K \to \mathbb{R}^{n_X}$ for which

$$\dot{x}(t;k) = f(t, x(t;k), k) \tag{3.8}$$

$$\dot{k} = 0, \tag{3.9}$$

so $x : T_{\text{plan}} \to X$ is a trajectory of the planning model, and the notation $x(t;k)$ denotes that the trajectory is parameterized by k. From here on, we refer to any such trajectory x of the planning model as a **plan**, which overlaps with the definition of a plan as a trajectory $x_{\text{plan}}^{(i)} : T^{(i)} \to X_{\text{hi}}$ of the high-fidelity model; below, in §3.5.3, we lift x to $x_{\text{plan}}^{(i)}$ to resolve this conflict.

We require the planning model to have three additional properties. First, f is continuous and differentiable almost everywhere in T_{plan}, X, and K. Second, there exists a point in the planning space X from which every plan begins; we denote this point $x_0 \in X$ such that $x(0;k) = x_0$ for all $k \in K$. Third, every plan ends with a stop, meaning $f(t_{\text{f}}, \cdot, k) = 0$ for all $k \in K$. Note, we elaborate on this second property below, in §3.5.2.

Running Example 3.5. *For the Segway, $X = P$, the position subspace of $Q = P \times \Theta$, with state $x = (p_1, p_2) \in X$. The planning model is*

$$f(t, x(t;k), k) = s(t) \begin{bmatrix} k_1 - k_2 \cdot (p_2(t;k) - p_{2,0}) \\ k_2 \cdot (p_1(t;k) - p_{1,0}) \end{bmatrix} \text{ with} \tag{3.10}$$

$$s(t) = \begin{cases} 1 & t \in [0, t_{\text{plan}}) \\ 1 - \frac{t - t_{\text{plan}}}{t_{\text{f}} - t_{\text{plan}}} & t \in [t_{\text{plan}}, t_{\text{f}}] \end{cases}, \tag{3.11}$$

with the point $x_0 = (p_{1,0}, p_{2,0}) \in P$. Trajectories of this model end in a stop because of the scaling function $s : T_{\text{plan}} \to [0,1]$.

This model creates circular arc trajectories (that is, Dubins' paths parameterized by time), with longitudinal velocity k_1 and angular velocity k_2, initial position x_0, and an initial heading of $\theta(0) = 0$. To see why, rewrite the model as

$$\dot{x}(t;k) = s(t) \underbrace{\begin{bmatrix} 0 & -k_2 \\ k_2 & 0 \end{bmatrix}}_{A(k)} x(t;k) + s(t) \underbrace{\begin{bmatrix} k_1 + k_2 p_{2,0} \\ -k_2 p_{1,0} \end{bmatrix}}_{b(k)}. \tag{3.12}$$

Therefore, for any fixed $k \in K$, $A(k)$ defines a linear time-varying system with complex eigenval-

ues at any $t \in T_{\text{plan}}$; this produces a circular vector field about the point $b(k)$.

3.5.2 The Planning Frame and the World Frame

By starting every parameterized plan at x_0, each evolves in a coordinate frame relative to the robot at the beginning of any plan. Another way to think of this is that every plan begins at the same pose *relative to the robot* at each time $t^{(i)}$. Therefore, we call the coordinate frame centered at $x_0 \in X$ the **planning frame**. Fixing x_0 in this way makes it tractable to compute reachable sets, introduced in later in this chapter and used to formulate collision-avoidance constraints.

However, obstacles do not appear in the world frame, not the planning frame. To resolve this, we introduce a pair of operators, $\mathsf{world2plan} : W \times X_{\text{hi}} \to W$ and $\mathsf{plan2world} : W \times X_{\text{hi}} \to W$. These operators transform points back and forth between the world origin, $0 \in W$, and the coordinate frame centered at $\text{proj}_W(x_0)$ and rotated, if necessary, for the robot's pose. Note, for a manipulator robot with a fixed baselink, these functions are not needed.

Running Example 3.6. *Suppose the Segway is at a state $x_{\text{hi}} = (p, \theta, v, \omega) \in X_{\text{hi}}$, where $p \in X \subset \mathbb{R}^2$ is the robot's center-of-mass position. Then, a point $w \in W$ can be shifted to the robot's local frame by*

$$\mathsf{world2plan}(w, x_{\text{hi}}) = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} (p - w) + x_0. \tag{3.13}$$

Notice that $X \cong W$, so we abuse notation to directly add a point in X to a point in W. To shift from the planning frame to the world frame, we similarly use

$$\mathsf{plan2world}(w, x_{\text{hi}}) \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} (p - x_0) + w. \tag{3.14}$$

3.5.3 Lifting the Planning Model to the High-Fidelity Model

The previous discussion explains the spatial relationship between the planning model and the robot's workspace. We now discuss the relationship between the planning model and the high-fidelity model. In particular, we resolve the definition of a plan as both a trajectory $x_{\text{plan}}^{(i)}$ in the high-fidelity model's state space, and a trajectory x of the planning model.

To do so, we define a map $\mathsf{liftplan} : \mathbb{N} \times X \to X_{\text{hi}}$, which is specific to a given robot and planning model. Suppose the robot is in the i^{th} planning iteration, and recall that $t^{(i)} \in T$ is the time at the beginning of $T^{(i)} \subset T$. So, let the robot be at initial state $x_{\text{hi},0}^{(i)}$ at time $t^{(i)}$, with plan

$k^{(i)} \in K$. Then, the **lifted plan** is $x^{(i)}_{\text{plan}} : T^{(i)} \to X_{\text{hi}}$ for which

$$x^{(i)}_{\text{plan}}(t; k^{(i)}) = \mathsf{liftplan}(i, x(t - t^{(i)}; k^{(i)})) \tag{3.15}$$

$$= x^{(i)}_{\text{hi},0} + \int_{t^{(i)}}^{t} f_{\text{lift}}(\tau, x^{(i)}_{\text{plan}}(\tau; k^{(i)}), k^{(i)}) d\tau, \tag{3.16}$$

where the argument $i \in \mathbb{N}$ to liftplan is used in place of $t^{(i)}$, $k^{(i)}$, and $x^{(i)}_{\text{hi},0}$, which would otherwise be needed as arguments. Here, $f_{\text{lift}} : T^{(i)} \times X_{\text{hi}} \times K \to \mathbb{R}^{n_{\text{hi}}}$ is the **lifted planning model**, which is specific to any given robot and trajectory parameterization. Typically, f_{lift} has the property that, for any $k \in K$, $f_{\text{lift}}(t, x_{\text{hi}}(t; k), k)$ is equivalent to $f(t, x(t; k), k)$ in each coordinate of the planning space X, even though the initial conditions of the lifted plan and planning model trajectory are not the same (that is, $\text{proj}_X(x^{(i)}_{\text{plan}}(0; k)) \neq x(0; k)$). Another way to think of this is that the lifted plan evolves as though plan2world has been applied to the entire planning frame, so $\text{proj}_X(x^{(i)}_{\text{hi},0}) = \mathsf{plan2world}(x_0)$.

Notice that liftplan shifts the domain of x from T_{plan} to $T^{(i)}$. Furthermore, liftplan does not necessarily just extend the codomain of f from X to X_{hi}; instead, the lifted planning model f_{lift} enables $x^{(i)}_{\text{plan}}$ to evolve in X_{hi} as well as in X (this is made more clear by the running example below). Since the lifted plan evolves in X_{hi}, the argument $x^{(i)}_{\text{hi},0}$ ensures that it starts at the robot's initial state at the i^{th} planning iteration. This lets us generate a full-state trajectory that the high-fidelity model can track using, e.g., closed-loop feedback.

Running Example 3.7. *To lift Segway plans generated by* (3.10) *from* X *to* X_{hi}*, we apply the planning model to the states of the high-fidelity model as expected. Note, from the above discussion, we need only define* f_{lift} *as per* (3.16). *With* $(*) = (t, x^{(i)}_{\text{plan}}(t; k), k)$ *to replace the input arguments for space, we have*

$$f_{\text{lift}}(*) = \begin{bmatrix} \dot{p}_1(*) \\ \dot{p}_2(*) \\ \dot{\theta}(*) \\ \dot{v}(*) \\ \dot{\omega}(*) \end{bmatrix} = \begin{bmatrix} s(t - t^{(i)})k_1 \cos(\theta(t; k)) \\ s(t - t^{(i)})k_1 \sin(\theta(t; k)) \\ s(t - t^{(i)})k_2 \\ \frac{d}{dt}s(t - t^{(i)})k_1 \\ \frac{d}{dt}s(t - t^{(i)})k_2 \end{bmatrix}. \tag{3.17}$$

Notice that we shift time from $T^{(i)}$ *to* T_{plan} *for the scaling function* s *from the planning model. Also, this example of* f_{lift} *creates discontinuous longitudinal and angular acceleration profiles due to the* $\frac{d}{dt}s(\cdot)$ *terms; but, this is produces a full-state trajectory that can be tracked by the Segway.*

3.5.4 Using Trajectory Parameters Online

Per the above definitions, each trajectory parameter $k \in K$ maps to a plan. We take advantage of this to use K as the domain for trajectory optimization at runtime. Consequently, we further conflate the word **plan** with $k \in K$; that is, in each i^{th} iteration, we call any choice of k a plan.

While using parameterized plans is a limitation in contrast to traditional MPC approaches (which optimize over, e.g., inputs $u : T \to U$), this approach has several benefits. First, it enables computing reachable sets that are less conservative than would be for any possible input. Second, it simplifies the design of a tracking controller for the high-fidelity model, since we need not track any arbitrary trajectory. Third, by optimizing over the parameters at runtime, we are able to certify continuous-time safety, whereas approaches that optimize over inputs drawn from U typically have to discretize time (e.g., [PR14, YMCA13, WB09]). Note, this third benefit relies on our ability to compute reachable sets that capture the motion of the robot over continuous time.

The online planning procedure using the trajectory parameters is summarized later on, in Algorithm 1, in §3.8.

3.6 Tracking Controller and Error

Now, we discuss how the robot tracks plans using a controller, and we discuss the resulting tracking error. We then discuss the relationship between the robot's initial condition and its possible choices of plans, in the sense that some plans will cause more tracking error than others. Finally, we address the modeling error between the high-fidelity model and the actual robot.

3.6.1 The Tracking Controller

Suppose the robot is in its i^{th} planning iteration, and has generated a plan $k \in K$ to track. Further on in this work, we use $k^{(i)}$ (as opposed to just $k \in K$) to denote the i^{th} plan, but we omit the index i to ease notation here. Going forward, we say k to mean the plan parameterized by $k \in K$; for example, we say the robot tracks k.

To drive the high-fidelity model towards k, the robot uses a feedback controller

$$u_k : T^{(i)} \times X_{\text{hi}} \to U, \tag{3.18}$$

where the subscript indicates that this controller attempts to track k. This controller results in a closed-loop high-fidelity model given by

$$\dot{x}_{\text{hi}}(t;k) = f_{\text{hi}}\left(t, x_{\text{hi}}(t;k), u_k\big(t, x_{\text{hi}}(t;k)\big)\right). \tag{3.19}$$

To perform feedback control, we typically use a lifted plan, as shown in the following example.

Running Example 3.8. *The Segway uses a proportional-derivative controller. Let $G_P \in \mathbb{R}^{2\times 2}$, $G_\Theta \in \mathbb{R}^{1\times 1}$, and $G_{\dot{Q}} \in \mathbb{R}^{2\times 2}$ be matrices of control gains. Suppose the Segway is in the i^{th} planning iteration, starting from initial condition $x_{\text{hi},0}^{(i)}$, and let $k \in K$. Then, the Segway's tracking controller is given by*

$$u_k(t, x_{\text{hi}}(t;k)) = G_P e_P(t;k) + G_\Theta \cdot (\theta(t;k) - s(t - t^{(i)})k_2 t) + \tag{3.20}$$

$$+ G_{\dot{Q}} \begin{bmatrix} s(t - t^{(i)})k_1 - v(t;k) \\ s(t - t^{(i)})k_2 - \omega(t;k) \end{bmatrix}, \tag{3.21}$$

with the position error e_P given in the robot's body-fixed coordinate frame:

$$e_P(t;k) = \begin{bmatrix} \cos(\theta(t;k)) & \sin(\theta(t;k)) \\ -\sin(\theta(t;k)) & \cos(\theta(t;k)) \end{bmatrix} \text{proj}_P(x_{\text{hi}}(t;k) - x_{\text{plan}}^{(i)}(t;k)), \tag{3.22}$$

where $x_{\text{plan}}^{(i)}(t;k) = \mathsf{liftplan}(i, x(t - t^{(i)}; k))$. Notice that $k_2 t = \text{proj}_\Theta(x_{\text{plan}}^{(i)}(t;k))$ and similarly the error terms for v and ω are functions of the lifted plan.

3.6.2 Tracking Error

Notice from the example above that the purpose of the tracking controller is to reduce the error between the robot's state and the (lifted) plan. We now consider this notion of error independent of any particular planning iteration; that is, we consider error as a function of just the robot's initial condition and trajectory parameter for the duration of any plan. To that end, we define the **tracking error** as a trajectory $x_{\text{err}} : T_{\text{plan}} \to \mathbb{R}^{n_{\text{hi}}}$ for which

$$x_{\text{err}}(t; x_{\text{hi},0}^{(i)}, k) = x_{\text{hi}}(t;k) - \mathsf{liftplan}(i, x(t - t^{(i)}; k)), \tag{3.23}$$

where $x_{\text{hi}} : T^{(i)} \to X_{\text{hi}}$ is the trajectory of the closed-loop high-fidelity model (3.19), and $x : T_{\text{plan}} \to X_{\text{hi}}$ is the trajectory of the planning model.

Note, (3.18) gives the error in the high-fidelity model's state, which is why the codomain of x_{err} is the same dimension as X_{hi}. However, for the purpose of collision avoidance, we are concerned with the tracking error *in workspace*; to this end, we revisit this notion of tracking error later in the chapter, when we introduce reachable sets.

3.6.3 Bounds on Choice of Plans

Notice that the tracking error in (3.23) is a function of the robot's initial condition at the beginning of any plan. If the robot is allowed to pick any arbitrary $k \in K$, given an initial condition $x_{\text{hi},0}$, then it is possible that the tracking error is very large. For example, if the robot is traveling at high speed, and then chooses a plan with zero parameterized speed.

To mitigate tracking error as a function of the initial condition, we define a generic **parameter bounds function** $\mathcal{K}_{\text{lim}} : X_{\text{hi}} \to \text{pow}\,(K)$ that returns a subset of K containing allowable choices of plans for a given initial condition. Note, $\mathcal{K}_{\text{lim}}$ is defined for each robot and planning model.

Running Example 3.9. *For the Segway, recall that $(k_1, k_2) \in K \subset \mathbb{R}^2$ parameterizes the robot's longitudinal and angular velocities. Let $\Delta_v, \Delta_\omega > 0$ be allowable bounds on the commanded change in either velocity, for a given initial velocity. Then, for any $x_{\text{hi},0} \in X_{\text{hi}}$,*

$$\mathcal{K}_{\text{lim}}(x_{\text{hi},0}) = \left\{(k_1, k_2) \in K \;\mid\; |k_1 - \text{proj}_V(x_{\text{hi},0})| \leq \Delta_v \text{ and } |k_2 - \text{proj}_\Omega(x_{\text{hi},0})| \leq \Delta_\omega\right\}. \quad (3.24)$$

3.6.4 Modeling Error

The purpose of RTD is to plan safe trajectories that compensate for tracking error, which presumes that the high-fidelity model is correct. However, this is not always true in practice. To compensate for such **modeling error**, we introduce the following assumption concerning the robot hardware.

Assumption 3.1. *Suppose the robot is tracking k in its i^{th} planning iteration, so $t \in T^{(i)}$. Let $x_j(t; k)$ be the value of the j^{th} coordinate of the state $x_{\text{hi}}(t; k)$ given by the closed-loop high-fidelity model as in* (3.19). *We assume that there exists a value $\varepsilon_{x_j} > 0$, and that the robot is able to perform state estimation, such that $x_j(t; k)$ is within ε_{x_j} of the value of the robot's actual j^{th} coordinate.*

We introduce modeling error here, *after* introducing trajectory parameters and the corresponding feedback controllers, to emphasize that our parameterized plans and receding-horizon formulation make such an assumption reasonable.

Running Example 3.10. *For the Segway hardware, we find $\varepsilon_{p_1}, \varepsilon_{p_2} \approx 0.1$ m when running indoors on a tile floor. Due to the robot's wheel encoders, accelerometer, and high-torque motors, we find that ε_θ, ε_v, and ε_ω are negligible.*

3.7 Reachable Sets

At this point, we have established the robot's high-fidelity and planning models, and begun to relate them through tracking error. We have also established obstacles as portions of the workspace to avoid. To enable the identification of collision-avoiding plans, we use objects called reachable sets (RSs), hence the name Reachability-based Trajectory Design.

In particular, we define a Forward Reachable Set (FRS), which contains the forward occupancy of the robot tracking any parameterized plan (meaning, it includes tracking error). This set is defined over the robot's time, initial condition, workspace, and trajectory parameters. This lets us consider the reachable subsets corresponding to any particular plan. The overall goal of RTD is to find a plan (in each receding-horizon iteration) for which the corresponding subset of the FRS does not intersect predictions of obstacles. In practice, the FRS is computed offline, then used online for planning.

In this section, first, we define the FRS. Second, we redefine the FRS in terms of the planning model and tracking error, to better understand its structure. Third, we redefine predictions as reachable sets, to allow us to compare the FRS with predictions directly during online planning, as illustrated in Figure 3.1.

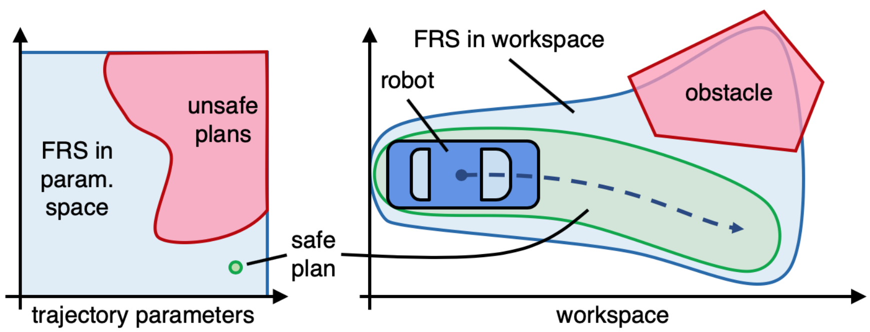


Figure 3.1: An overview of the FRS for a wheeled robot in dark blue; the FRS is shown in light blue, projected into the trajectory parameter space on the left and the workspace on the right. An obstacle in the workspace corresponds to a set of unsafe trajectory parameters. At runtime, we use this unsafe set as a collision avoidance constraint for trajectory optimization; any feasible solution is provably collision-free. An example feasible (safe) plan is shown as a green point in the parameter space and as a dashed blue line in the workspace, along with the green collision-free subset of the FRS corresponding to that plan. In this figure, the obstacle and workspace are shown in the robot's planning frame, with the robot at the initial condition x_0.

3.7.1 The Forward Reachable Set

To define the FRS, we first define a special set of initial conditions:

$$X_{\text{hi},0} = \left\{ x_{\text{hi},0} \in X_{\text{hi}} \mid \text{proj}_X(x_{\text{hi},0}) = x_0 \right\}, \tag{3.25}$$

where x_0 is the initial condition of each plan in the planning frame.

The FRS, denoted $\mathcal{R}_{\text{FRS}}$, contains all times and points in workspace reachable by the robot, including tracking error, for each trajectory parameter, and for each initial condition drawn from $X_{\text{hi},0}$:

$$\begin{aligned}\mathcal{R}_{\text{FRS}} = \Big\{&(t, x_{\text{hi},0}, p, k) \in T_{\text{plan}} \times X_{\text{hi},0} \times W \times K \ \mid \ x_{\text{hi}}(0; k) = x_{\text{hi},0}, \\ &\dot{x}_{\text{hi}}(t; k) = f\big(t, x_{\text{hi}}(t; k), u_k(t, x_{\text{hi}}(t; k))\big),\ k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}), \\ &\text{and } p \in \mathsf{FO}(x_{\text{hi}}(t; k))\Big\}.\end{aligned} \tag{3.26}$$

This means that the FRS is defined in the *planning frame*, not the *world frame*, per §3.5.2. Note, in the model f_{hi}, the use of the tracking controller u_k for each k implies that $\mathcal{R}_{\text{FRS}}$ includes tracking error.

Instead of presenting a running example here, we treat the Segway's FRS implementation in detail in §4. See the Segway implementation details and numerical results in §9.1.

3.7.2 The Planning and Error Reachable Sets

To better understand the structure of the FRS, we first consider an RS of the planning model, called the Planning Reachable Set (PRS). Then, we consider an RS for the tracking error, called an Error Reachable Set (ERS). Finally, we generate the FRS by combining the PRS and ERS; informally,

$$\text{FRS} = \text{ERS} + \text{PRS}.$$

The reason for this decomposition is that, when implementing the FRS numerically, we typically have to compute the PRS and ERS separately, then combine them together either offline or at runtime.

The PRS, denoted $\mathcal{R}_{\text{plan}}$, contains all times and planning states reachable by the planning model for each trajectory parameter:

$$\mathcal{R}_{\text{plan}} = \left\{(t, x, k) \in T_{\text{plan}} \times X \times K \ \mid \ x = x_0 + \int_0^t f(\tau, \tilde{x}(\tau; k), k)d\tau\right\}, \tag{3.27}$$

where again $x_0 \in X$ is the initial condition for every planning model trajectory per the definition in §3.5; as expected, $\mathcal{R}_{\text{plan}}$ is in the planning frame.

The ERS, denoted $\mathcal{R}_{\text{err}}$, contains all times and tracking errors achieved by the robot for each possible initial condition:

$$\begin{aligned}\mathcal{R}_{\text{err}} = \Big\{&(t, x_{\text{hi},0}, e) \in T_{\text{plan}} \times X_{\text{hi},0} \times \mathbb{R}^{n_{\text{hi}}} \mid \exists\, k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}) \text{ s.t.} \\ & e = x_{\text{hi}}(t;k) - x_{\text{plan}}(t;k), \text{ where} \\ & \dot{x}_{\text{hi}}(t;k) = f\big(t, x_{\text{hi}}(t;k), u_k(t, x_{\text{hi}}(t;k))\big),\ x_{\text{hi}}(0;k) = x_{\text{hi},0}, \\ & \dot{x}_{\text{plan}}(t;k) = f_{\text{lift}}\big(t, x_{\text{plan}}(t;k), k\big),\ \text{and } x_{\text{plan}}(0;k) = x_{\text{hi},0}\Big\}.\end{aligned} \tag{3.28}$$

Note, this definition makes use of the lifted planning model, but, since the ERS is independent of any particular planning iteration, we do not use $\mathsf{liftplan}$.

The condition $\text{proj}_X(x_{\text{hi},0}) = x_0$ ensures that the ERS is also defined in the planning frame, so the robot starts from the same state (in the planning space X) as each plan $k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0})$. Without this condition, the tracking error could be made arbitrarily large at $t = 0 \in T_{\text{plan}}$. Typically, this means that, at $t = 0$, there is zero generalized coordinate error (i.e., in the Q coordinates of X_{hi}), but nonzero generalized velocity error (the $\dot{Q}$ coordinates). In other words, we assume that the high-fidelity model of the robot is accurate, so we're able to accurately estimate the robot's state at the beginning of each receding-horizon planning iteration by using the high-fidelity model. This assumption is reasonable because t_{plan} is usually small (i.e., we replan often). See §3.6.4 for how we treat modeling error (which is not a focus of this work).

We can now rewrite the FRS in terms of the PRS and ERS.

Proposition 3.2.

$$\begin{aligned}\mathcal{R}_{\text{FRS}} = \Big\{&(t, x_{\text{hi},0}, p, k) \in T_{\text{plan}} \times X_{\text{hi},0} \times W \times K \mid \exists\, x \in X \text{ s.t.} \\ & (t, x, k) \in \mathcal{R}_{\text{plan}},\ (t, x_{\text{hi},0}, e) \in \mathcal{R}_{\text{err}},\ \text{and } p \in \mathsf{FO}(x + \text{proj}_X(e))\Big\},\end{aligned} \tag{3.29}$$

where $\text{proj}_X(e)$ *means the projection of the tracking error into the coordinates of* $\mathbb{R}^{n_{\text{hi}}}$ *corresponding to* X*, since* $X_{\text{hi}} \subset \mathbb{R}^{n_{\text{hi}}}$.

Proof. First, note that $\text{proj}_X(x_{\text{hi},0}) = x_0$, so the initial condition requirement of $\mathcal{R}_{\text{FRS}}$ is obeyed. Furthermore, $k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0})$ is implied since $(t, x_{\text{hi},0}, e) \in \mathcal{R}_{\text{err}}$; that is, the bounds on the choice of trajectory parameter are respected by the robot's initial condition. Then notice that $\text{proj}_X : X_{\text{hi}} \to X$ is linear, because both X and X_{hi} are embedded in vector spaces. To complete the proof, we must show that, if we add the (projected) tracking error to the planning model state, we recover the high-fidelity model's state in the subspace X. Notice that

$$\begin{aligned}\mathrm{proj}_X(e) &= \mathrm{proj}_X(x_{\mathrm{hi}}(t;k) - x_{\mathrm{plan}}(t;k)) && (3.30)\\ &= \mathrm{proj}_X(x_{\mathrm{hi}}(t;k)) - \mathrm{proj}_X(x_{\mathrm{plan}}(t;k)) && (3.31)\\ &= \mathrm{proj}_X(x_{\mathrm{hi}}(t;k)) - x, && (3.32)\end{aligned}$$

where x_{hi} and x_{plan} are as in (3.28). Here, (3.31) follows from the linearity of the projection operator and (3.32) follows from (3.28). Then we have the desired result: $x + \mathrm{proj}_X(e) = \mathrm{proj}_X(x_{\mathrm{hi}}(t;k))$. □

3.7.3 Predictions as Reachable Sets

Recall that a prediction is a map $\mathcal{P}^{(i)} : T^{(i)} \to \mathrm{pow}\,(W)$. We can consider the graph of a prediction as a forward reachable set for all sensed obstacles, and then *intersect* this reachable set with the FRS to identify plans that could cause collisions. To that end, we define an **obstacle reachable set** (ORS), given the prediction $\mathcal{P}^{(i)}$ in the i^{th} planning iteration:

$$\begin{aligned}\mathcal{R}_{\mathrm{obs}}^{(i)} = \Big\{(t,p,k) \in T_{\mathrm{plan}} \times W \times K \ \mid p = \mathsf{world2plan}(w, x_{\mathrm{hi},0}^{(i)}),\\ w \in \mathcal{P}^{(i)}(t + t^{(i)}),\ \text{and } k \in K\Big\},\end{aligned} \tag{3.33}$$

where $x_{\mathrm{hi},0}^{(i)}$ is the robot's initial condition at the beginning of the i^{th} planning iteration. Notice that $\mathcal{R}_{\mathrm{obs}}^{(i)}$ is in the *planning frame*, which enables direct comparison to the FRS in the following section.

3.8 Online Planning

In each i^{th} receding-horizon planning iteration, RTD attempts to find a new plan $k^{(i)} \in K$. To do so, we solve an optimal control problem over the parameter space K, with constraints representing collision avoidance, and bounds on the parameters due to the robot's initial condition $x_{\mathrm{hi},0}^{(i)}$.

The cost function for this optimal control problem comes from a **high level planner**, which typically ignores the dynamics of the robot and returns an intermediate pose, or **waypoint**, for the robot to attempt to reach in the i^{th} iteration.

The online planning procedure is summarized in Algorithm 1 at the end of this section. A single planning iteration is illustrated in Figure 3.2.

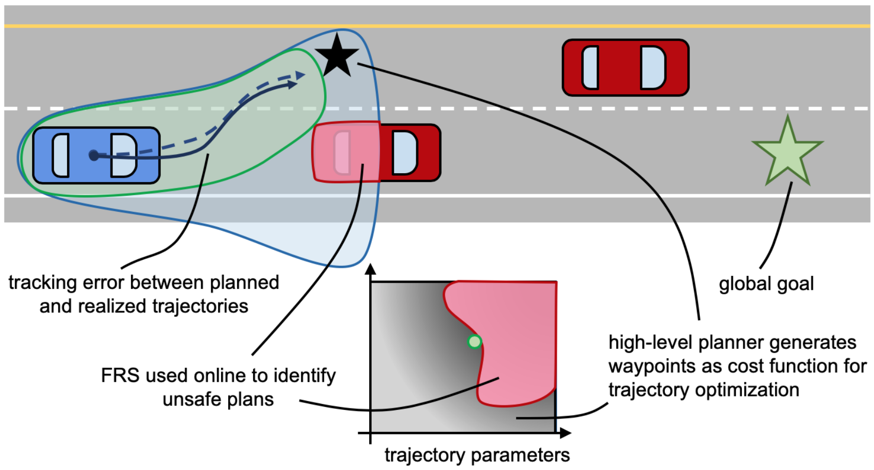


Figure 3.2: A single online planning iteration. Note, predictions of the obstacles are not shown. The high-level planner generates an intermediate waypoint (black star), which defines a cost function in the trajectory parameter space (shown as a gradient). The FRS is used to identify unsafe trajectory parameters, shown as the intersection of the FRS and an obstacle in the workspace, and as a pink region of the parameter space. Trajectory optimization finds a feasible plan, shown as a green point in the parameter space, and a dashed line in the workspace, with the corresponding subset of the FRS in green. The solid line shows the high-fidelity model trajectory with tracking error, which is contained in the green subset of the FRS corresponding to the safe plan.

3.8.1 The Initial Condition

Recall that the robot attempts to create a new plan while *simultaneously* tracking a previous plan. Therefore, the initial condition for the new plan must be *predicted*, using the robot's closed loop high-fidelity model. In other words, if the robot is currently tracking plan $k^{(i-1)}$ while attempting to find $k^{(i)}$, it uses

$$x^{(i)}_{\text{hi},0} = x_{\text{hi}}(t^{(i)} + t_{\text{plan}}; k^{(i-1)}) \tag{3.34}$$

as the initial condition for the i^{th} planning iteration. The initial condition is at time $t^{(i)} + t_{\text{plan}}$ because every planning iteration is of duration t_{plan}. Here,

$$\dot{x}_{\text{hi}}(t; k^{(i-1)}) = f_{\text{hi}}(t, x_{\text{hi}}(t; k^{(i-1)}), u_{k^{(i-1)}}(\cdot))\ \forall\, t \in [t^{(i)}, t^{(i)} + t_{\text{plan}}] \tag{3.35}$$

is given by the closed-loop high-fidelity model. Per §3.1, for the purpose of this work, we assume that $x^{(i)}_{\text{hi},0}$ is a correct estimate of the robot's state (see Assumption 3.1 for when this assumption does not hold).

We must then consider the appropriate subset of the FRS for the given initial condition. For the initial condition $x^{(i)}_{\text{hi},0}$, we denote this set

$$\mathcal{R}^{(i)}_{\text{FRS}} = \left\{(t,p,k) \in \mathcal{R}_{\text{FRS}} \;\mid\; (t, x^{(i)}_{\text{hi},0}, p, k) \in \mathcal{R}_{\text{FRS}}\right\}, \tag{3.36}$$

where i is used to refer to the initial condition in a similar way to $\mathsf{liftplan}$, which uses i as an argument to refer to $t^{(i)}$, $k^{(i)}$, and $x^{(i)}_{\text{hi},0}$.

3.8.2 Identifying Unsafe Plans

We conservatively identify all unsafe plans for the i^{th} planning iteration by projection the *intersection* of the FRS and ORS into the trajectory parameter space; we confirm that doing so is correct with the following proposition.

Proposition 3.3. *Suppose the robot is in the i^{th} planning iteration, with initial condition $x^{(i)}_{\text{hi},0} \in X_{\text{hi}}$. The set of unsafe plans in the i^{th} receding-horizon planning iteration is overapproximated by the parameter space projection of the FRS intersected with the ORS:*

$$K^{(i)}_{\text{unsf}} \subseteq \text{proj}_K\left(\mathcal{R}^{(i)}_{\text{FRS}} \cap \mathcal{R}^{(i)}_{\text{obs}}\right). \tag{3.37}$$

Proof. This proposition follows from the construction of the FRS and ORS; we proceed by unraveling definitions. Suppose $k \in K$ could cause a collision during $T^{(i)}$; that is, there exists $t \in T_{\text{plan}}$ and some obstacle $O : T \to \text{pow}\,(W)$ such that $O(t + t^{(i)}) \cap \mathsf{FO}(x_{\text{hi}}(t;k)) \neq \emptyset$, where $x_{\text{hi}}(t;k)$ is the trajectory of the closed-loop high-fidelity model starting from initial condition $x^{(i)}_{\text{hi},0}$ (and using the controller u_k). So, we must show that $k \in K^{(i)}_{\text{unsf}}$. From the FRS definition (3.26) and from (3.36), for any $p \in \mathsf{FO}(x_{\text{hi}}(t;k))$, we have $(t,p,k) \in \mathcal{R}^{(i)}_{\text{FRS}}$. From the prediction definition (3.6), we know that $p \in O(t + t^{(i)}) \implies p \in \mathcal{P}^{(i)}(t + t^{(i)})$. Consequently, from the ORS definition (3.33), we know that $p \in \mathcal{P}^{(i)}(t + t^{(i)}) \implies (t,p,k) \in \mathcal{R}^{(i)}_{\text{obs}}$. Therefore, since (t,p,k) is in both $\mathcal{R}^{(i)}_{\text{FRS}}$ and $\mathcal{R}^{(i)}_{\text{obs}}$, $k \in \text{proj}_K(\mathcal{R}^{(i)}_{\text{FRS}} \cap \mathcal{R}^{(i)}_{\text{obs}})$, completing the proof. $\square$

3.8.3 Trajectory Optimization

Let cost : $K \to \mathbb{R}$ be an arbitrary cost function for trajectory optimization (we specify how we typically construct cost below in §3.8.4). Then, in each planning iteration, we attempt to solve the following program within the timeout t_{plan}:

$$k^{(i)} = \underset{k^{(i)} \in K}{\text{argmin}} \quad \mathsf{cost}(k^{(i)}) \tag{3.38}$$

$$\text{s.t.} \quad k^{(i)} \notin \text{proj}_K(\mathcal{R}^{(i)}_{\text{FRS}} \cap \mathcal{R}^{(i)}_{\text{obs}}) \tag{3.39}$$

$$k^{(i)} \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}). \tag{3.40}$$

Recall that the parameter bounds function $\mathcal{K}_{\text{lim}}$ that produces a subset of the space K that is feasible for the robot's initial condition $x^{(i)}_{\text{hi},0}$.

If (3.38) cannot be solved within t_{plan} seconds, then the robot continues tracking its previous plan $k^{(i-1)}$, which ends in a stop per the definition of every parameterized plan. While tracking $k^{(i-1)}$, the robot can still attempt to replan (i.e., to find $k^{(i+1)}$ to begin at time $t^{(i)} + t_{\text{plan}}$).

3.8.4 The High-Level Planner

Recall that we typically break planning into a three-tiered hierarchy (see §1.1.3 and §2), with a *high-level planner* at the top, a *trajectory planner* in the middle, and a *tracking controller* at the bottom. In this chapter, we have developed RTD as a trajectory planner, and discussed how we treat the tracking controller. To complete the hierarchy, we use a high-level planner to generate the cost function for trajectory optimization each receding-horizon planning iteration.

RTD is agnostic to the high-level planner. For wheelend and aerial robots, we typically use RRT or A* [LaV06] as the high-level planner; these ignore the robot's high-fidelity and planning models, and are only tasked with generating a **waypoint** in W between the robot's initial condition $\text{proj}_W(x^{(i)}_{\text{hi},0})$ and a global goal location. Note, the waypoint need not be collision-free; but, using intermediate waypoints is convenient for encoding a coarse way to guide a robot around an obstacle.

Running Example 3.11. *Suppose that $x^{(i)}_{\text{des}} \in W$ is a waypoint generated by the high-level planner in the i^{th} iteration. An example cost function for the Segway is*

$$\mathsf{cost}(k^{(i)}) = \left\| x(t_{\text{plan}}; k^{(i)}) - \mathsf{world2plan}(x^{(i)}_{\text{des}}) \right\|_2^2, \tag{3.41}$$

where $x : T_{\text{plan}} \to X$ is the trajectory of the planning model.

3.8.5 The Online Planning Algorithm

We now put all of the previous portions of this chapter together for safe, real-time, online planning. RTD uses Algorithm 1 in each i^{th} planning iteration.

For the purpose of enforcing the timeout t_{plan}, we assume that Lines 2–5 execute instantaneously. In practice, we usually allot ≈ 0.1 seconds for these lines. Note, the cost function generation and obstacle can prediction can be run in parallel to the rest of the algorithm, in an anytime fashion.

Algorithm 1 The i^{th} receding-horizon planning iteration (executes while robot is tracking $k^{(i-1)}$)

1: **require** previous trajectory $x_{\text{hi}}^{(i-1)} : T^{(i-1)} \to X_{\text{hi}}$, previous plan $k^{(i-1)} \in K$, FRS $\mathcal{R}_{\text{FRS}}$, and parameter bounds function $\mathcal{K}_{\text{lim}}$

2: $\mathsf{cost}^{(i)} \leftarrow \texttt{GenerateCostFunc}(x_{\text{hi}}^{(i-1)}(t^{(i)}))$

3: $x_{\text{hi},0}^{(i)} \leftarrow \texttt{PredictInitialState}(x_{\text{hi}}^{(i-1)})$

4: $\mathcal{R}_{\text{FRS}}^{(i)} \leftarrow \texttt{GetFRS}(\mathcal{R}_{\text{FRS}}, x_{\text{hi},0}^{(i)})$

5: $\mathcal{R}_{\text{obs}}^{(i)} \leftarrow \texttt{PredictObstacles}(x_{\text{hi},0}^{(i)})$

6: $k^{(i)} \leftarrow \texttt{FindTrajectory}(t_{\text{plan}}, \mathsf{cost}^{(i)}, \mathcal{R}_{\text{FRS}}^{(i)}, \mathcal{R}_{\text{obs}}^{(i)}, \mathcal{K}_{\text{lim}}(x_{\text{hi},0}^{(i)}))$

7: **return** new plan $k^{(i)}$ or else continue $k^{(i-1)}$

3.8.6 Provably Safe, Not-at-Fault Planning

To conclude this chapter, we confirm that using Algorithm 1 ensures safe, not-at-fault planning for all time (that is, it ensures persistent feasibility). To do so for wheeled and aerial robots, we specify a minimum sensor horizon d_{sense} as in §3.4, to ensure the robot can sense and predict any obstacle that it would need to avoid in any planning iteration. In the case of (stationary) manipulators, we assume that d_{sense} is large enough to sense and predict any obstacle that would enter the workspace during any planning iteration.

To specify the minimum sensor horizon, consider the following program

$$
\begin{aligned}
d_{\text{min}} = \max_{k^{(0)},k^{(1)},x_{\text{hi},0}} \quad & d_{\text{plan}} + d_{\text{stop}} + v_{\text{max,obs}} \cdot (t_{\text{plan}} + t_{\text{f}}) && (3.42)\\
\text{s.t} \quad & k^{(0)}, k^{(1)} \in K,\ x_{\text{hi},0} \in X_{\text{hi}},\ \text{proj}_X(x_{\text{hi},0}) = x_0, && (3.43)\\
& d_{\text{plan}} = \int_0^{t_{\text{plan}}} \left\|\text{proj}_W(x_{\text{hi}}(t; k^{(0)}))\right\|_2 dt && (3.44)\\
& d_{\text{stop}} = \int_{t_{\text{plan}}}^{2t_{\text{plan}}+t_{\text{f}}} \left\|\text{proj}_W(x_{\text{hi}}(t; k^{(1)}))\right\|_2 dt, && (3.45)\\
& x_{\text{hi}}(0; k^{(0)}) = x_{\text{hi},0},\ x_{\text{hi}}(t_{\text{plan}}; k^{(1)}) = x_{\text{hi}}(t_{\text{plan}}; k^{(0)}), && (3.46)\\
& \dot{x}_{\text{hi}}(t; k^{(0)}) = f_{\text{hi}}(t, x_{\text{hi}}(t; k^{(0)}), u_{k^{(0)}}(\cdot))\ \forall\, t \in [0, t_{\text{plan}}), && (3.47)\\
& \dot{x}_{\text{hi}}(t; k^{(1)}) = f_{\text{hi}}(t, x_{\text{hi}}(t; k^{(1)}), u_{k^{(1)}}(\cdot))\ \forall\, t \in T^{(1)}, && (3.48)
\end{aligned}
$$

where $T^{(1)} = [t_{\text{plan}}, 2t_{\text{plan}} + t_{\text{f}}] \subset T$. Recall that $v_{\text{max,obs}}$ is the maximum speed of any obstacle. The quantity d_{min} is the maximum rectilinear distance that the robot can travel during the entirety of any plan (the distance d_{stop}), after traveling the maximum possible distance along a previous plan (the distance d_{plan}), plus the maximum distance any obstacle can travel over the same duration. Before proceeding, we check that this quantity exists:

Lemma 3.4. *The quantity d_{min} exists.*

Proof. Notice that: (1) $v_{\text{max,obs}} \cdot (t_{\text{plan}} + t_{\text{f}})$ is a constant, (2) K is compact, (3) f_{hi} produces continuous trajectories, and (4) X_{hi} is bounded in all of its coordinates that are not shared with X by assumption. Since the rectilinear distance of a continuous trajectory in $\mathbb{R}^n$ is continuous [R+64], (3.42) is maximizing a continuous function on a compact set, completing the proof. □

Now, we use d_{min} to ensure safety.

Theorem 3.5. *Suppose that, at $t = 0 \in T$, the robot has a safe plan $k^{(0)}$ for its first planning iteration, and suppose it is at a state $x^{(0)}_{\text{hi},0}$. Suppose that the robot's sensor horizon is $d_{\text{sense}} > d_{\text{min}}$ as in* (3.42)*. Then, using Algorithm 1 in each planning iteration, the robot is safe and not-at-fault for all time $t \in T$.*

Proof. This proof follows from the definitions and assumptions throughout this chapter, and by induction on the planning iteration.

First, we check that d_{sense} is sufficiently large to ensure that the robot can sense and predict all obstacles that could cause a collision during any i^{th} iteration. At any time $t^{(i)}$ at the beginning of any planning iteration, notice that d_{sense} is greater than or equal to the maximum distance that the robot can travel over the time interval $[t^{(i)}, t^{(i)} + t_{\text{plan}} + t_{\text{f}}]$, plus the maximum distance that any obstacle could travel over that same time interval. In other words, the robot is able to sense and predict all obstacles that must be considered for safety in iteration $(i + 1)$. This follows from our definition of sensing in (3.7) (see §3.4.3).

Now, we can complete the proof by induction. First, notice that the robot is safe and not-at-fault for its initial plan, $k^{(0)}$. Suppose that it is safe and not-at-fault for the i^{th} plan. If it cannot find a new plan $k^{(i+1)}$ using (3.38), then plan $k^{(i)}$ ensures it is safe and not-at-fault for all time, because it comes to a collision-free stop, and can remain stopped indefinitely by assumption (see §3.4.2). If it can find a new plan $k^{(i+1)}$ that is feasible to (3.38), then that new plan is safe and not-at-fault by Proposition 3.3 and the definition of $\mathcal{K}_{\text{lim}}$ (see §3.6.3). $\square$

Corollary 3.6. *Recall by Assumption 3.1 that the high-fidelity model is accurate to within ε_{x_j} in each coordinate x_j of x_{hi}. Let $\varepsilon_{\text{hi}} \in \mathbb{R}^{n_{\text{hi}}}$ be a vector concatenating all the ε_{x_j} accuracies, and let $\varepsilon_W = \|\text{proj}_W(\varepsilon_{\text{hi}})\|_2$. When considering the robot hardware, we must ensure $d_{\text{sense}} \geq d_{\text{min}} + 2\varepsilon_W$.*

Proof. At the beginning of any i^{th} planning iteration, the robot is at most ε_W away from its workspace position as given by the high-fidelity model. Then, while tracking the i^{th} plan (assuming one is found), it is at most ε_W away from the trajectory of the high-fidelity model. Therefore, by adding $2\varepsilon_W$, we compensate for the cumulative inaccuracy. $\square$

Note, to create ε_W, we use $\|\cdot\|_2$ instead of $\|\cdot\|_\infty$ since our definition of sensing uses a 2-norm ball.

This concludes the online planning section. The takeaway is that we have shown how, by following all definitions and assumptions established throughout this theoretical overview of RTD, one can ensure that the robot is safe and not-at-fault for all time.

3.9 Chapter Review

The takeaway of this chapter is that RTD is a general framework for safe receding-horizon planning, independent of numerical representations and robot morphology.

3.9.1 Chapter Summary

In this chapter, we have provided a theoretical, generic overview of Reachability-based Trajectory Design. We introduced the high-fidelity model used to describe the robot, and explained the context of receding-horizon planning. We then covered how obstacles and the robot occupy volume in the workspace, and specifies how the robot must sense and predict obstacles. Then, we introduced a simplified planning model to make real-time, safe planning tractable. Since the high-fidelity model cannot perfectly track the parameterized plans of the planning model, we then covered the notion of tracking error. To compensate for tracking error during online planning, we constructed the robot's Forward Reachable Set to contain the motion of the robot when tracking any parameterized plan, from any initial condition. Finally, we used the Forward Reachable Set to identify all unsafe plans in each receding-horizon planning iteration, and proved that doing so renders the robot safe and not-at-fault when using our online planning algorithm.

3.9.2 What Is Missing?

As per the takeaway above, this theoretical presentation has not discussed how to implement RTD numerically, or for any particular robot. Implementation is nontrivial, especially for objects such as the FRS, which contain infinitely many points that are related by the potentially high-dimensional and nonlinear high-fidelity model of the robot. Often, we find that implementing the objects in this chapter directly as written is intractable. In the following chapters, we therefore detail specific methods to implement RTD for wheeled, aerial, and manipulator robots that preserve the critical properties of real-time, safe, not-at-fault trajectory planning.

CHAPTER 4

Forward Reachable Sets via Sums-of-Squares Programming

In this chapter, we use sums-of-squares (SOS) programming to perform RTD's offline reachability computation of the Forward Reachable Set (FRS). For now, we assume the existence of the Error Reachable Set (ERS) in 4.1, and reserve our computation of the ERS to §7.

To place this approach in the context of the literature, note, that we have only applied it to rigid-body wheeled robots (such as the Segway). Other approaches have applied similar SOS techniques to reachable sets for aerial robots, but these either rely on a finite library of trajectories [MT16] or are used to synthesize feedback controllers that treat higher-level planners as a disturbance, incurring large conservatism [SCH$^+$18]. For future work, it may be possible to extend the present approach to aerial robots by leveraging system decomposition techniques [KHV19, SCH$^+$18]. To that end, we discuss a generic decomposition technique in this chapter.

The sections of this chapter are as follows. (§4.1) We begin by representing the ERS using a differential equation, which we call a **tracking error model**. (§4.2) Next, we express a simplified version of the FRS that omits initial conditions and conservatively approximates the robot's forward occupancy. (§4.3) Then, to compute this simplified FRS, we formulate an infinite-dimensional linear program (LP) over continuous functions, and show that sub/super-level sets of these functions contain the FRS. (§4.4) To implement the infinite-dimensional LP, we conservatively approximate it using SOS polynomials of finite degree by applying Lasserre's hierarchy of moment relaxations [Las10]; we also observe that the memory usage of this approach scales poorly with the planning model dimension. (§4.5) To combat the memory usage challenges, we present a system decomposition approach to compute lower-dimensional reachable sets, then combine them into a reachable set for a higher-dimensional system. (§4.6) Next, we present a method for computing the FRS over a sequence of short time intervals, which we find enables less conservative trajectory planning in dynamic environments when compared to the FRSes computed in §4.4 and §4.5. (§4.7) To understand how we recover the original FRS from the simplified one (as in §4.2) used for SOS programming, we present a procedure called FRS swapping, wherein we reintroduce the robot's initial conditions to the FRS. (§4.8) Finally, to conclude the section, we explain how to use the FRS representation produced by SOS programming online, to generate constraints for trajectory optimization.

4.1 The Tracking Error Model

To enable SOS reachability analysis, we represent the tracking error as a differential equation, which we call the tracking error model. Doing so lets us compute the FRS using the planning model, plus tracking error as a disturbance, leveraging the disturbance/control synthesis approach in [MVTT14]. Recall that the planning model is $f : T_{\text{plan}} \times X \times K \to \mathbb{R}^{n_X}$. We similarly define the **tracking error model** $f_{\text{err}} : T \times X \times K \to \mathbb{R}^{n_X}$, and assume it exists as follows:

Assumption 4.1. *There exists $f_{\text{err}} : T_{\text{plan}} \times K \to \mathbb{R}^{n_X}$ for which*

$$\max_{x_{\text{hi},0} \in X_{\text{hi},0}} |\text{proj}_X(x_{\text{hi}}(t;k)) - x(t;k)| \leq \int_0^t f_{\text{err}}(\tau, k) d\tau \tag{4.1}$$

for all $t \in T_{\text{plan}}$ and $k \in K$, where and the absolute value is taken elementwise. Here, x_{hi} is the trajectory of the closed-loop high-fidelity model, and x is the trajectory of the planning model. We further assume f_{err} is Lipschitz continuous in t, x, and k.

Recall that, if $x_{\text{hi},0} \in X_{\text{hi},0}$, then $\text{proj}_X(x_{\text{hi},0}) = x_0$. So, f_{err} overapproximates the tracking error in the X dimensions for all trajectories of the closed-loop high-fidelity model that evolve in the planning frame.

We now check that this type of tracking model lets us recover any individual trajectory of the high-fidelity model.

Lemma 4.2. *Let $L_\delta = L^1(T_{\text{plan}}, [-1,1]^{n_X}])$ denote the space of absolutely integrable functions from T_{plan} to $[-1,1]$. Suppose $x_{\text{hi},0} \in X_{\text{hi}}$ and $\text{proj}_X(x_{\text{hi},0}) = x_0$. Then there exists a "disturbance" $\delta \in L_\delta$ such that, in the planning space X, the trajectory high-fidelity model is equivalent to the trajectory of the planning model plus the tracking error model times the disturbance:*

$$\text{proj}_X(x_{\text{hi}}(t;k)) = x(t;k) \tag{4.2}$$

where $\dot{x}_{\text{hi}}(t;k) = f_{\text{hi}}(t, x_{\text{hi}}(t;k), u_k(\cdot))$, $x_{\text{hi}}(0;k) = x_{\text{hi},0}$,

$$\dot{x}(t;k) = f(t, x(t;k), k) + f_{\text{err}}(t,k) \cdot \delta(t) \tag{4.3}$$

almost everywhere $t \in T_{\text{plan}}$, $x(0;k) = x_0$, and $f_{\text{err}}(\cdot) \cdot \delta(t)$ is taken elementwise.

Proof. Using Assumption 4.1, almost everywhere $t \in T_{\text{plan}}$, we can pick $\delta(t) \in [-1,1]^{n_X}$ such that

$$\text{proj}_X(x_{\text{hi}}(t;k)) - \int_0^t f(\tau, x(\tau;k), k)d\tau + x_0 = \int_0^t f_{\text{err}}(\tau,k)\delta(\tau)d\tau. \tag{4.4}$$

We can rearrange (4.4) and apply (4.3) to fulfill (4.2). □

Notice that Lemma 4.2 echoes the informal notion of FRS = PRS + ERS.

4.2 A Simplified FRS for SOS Reachability

To make computation tractable with SOS reachability, we redefine the FRS as follows. We subsume the robot's forward occupancy given by FO into a single **initial condition set** $X_0 \subset X$, then flow this entire set forward according to the planning model with tracking error introduced above. In particular, we define X_0 to satisfy the following:

$$X_0 + \{x(t;k)\} \supseteq \mathsf{FO}(x(t;k)) \;\forall\, k \in K, \tag{4.5}$$

where $\dot{x}(t;k) = f(t, x(t;k), k) + f_{\text{err}}(t,k) \cdot \delta(t)$ almost everywhere $t \in T_{\text{plan}}$. Note, X_0 need not have nonzero volume in every coordinate of X.

As an example, consider the case of a robot for which $X = \mathsf{SE}(2) = P \times \Theta$, recalling that we restrict the current SOS approach to wheeled robots. Then $X_0 \subset P$ must be large enough to contain all rotations of that rigid body for any trajectory parameter (so X_0 has zero volume in the Θ subspace of $\mathsf{SE}(2)$).

So, by applying the dynamics f to the entire volume X_0 during reachability analysis, we can conservatively approximate the motion of the robot's rigid body. At the end of this section, we note how one can circumvent this source of conservatism by choosing the planning model f as in the Segway running example 3.5.

Now consider the following simplified FRS:

$$\mathcal{R}_{\text{SOS}} = \Big\{(t,x,k) \in T_{\text{plan}} \times X \times K \mid \exists\, x_{\text{hi},0} \in X_{\text{hi},0},\ \tilde{x}_0 \in X_0, \text{ and} \tag{4.6}$$

$$\delta \in L_\delta \text{ s.t. } \tilde{x}(0;k) = x_0,\ x = \tilde{x}(t;k) + \tilde{x}_0,\ k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}), \tag{4.7}$$

$$\text{and } \dot{\tilde{x}}(\tau;k) = f(\tau, \tilde{x}(\tau;k), k) + f_{\text{err}}(\tau,k) \cdot \delta(\tau) \;\forall\, \tau \in T_{\text{plan}}\Big\}, \tag{4.8}$$

where, as a reminder, $x_0 \in X$ is the initial condition of every plan in the planning frame. Notice that, in comparison to the original FRS formulation, this FRS effectively takes the union over all initial conditions $x_{\text{hi},0}$, and all possible rotations of the robot's body per (4.5). This can also be seen by the fact that $\mathcal{R}_{\text{SOS}}$ does not include FO in its defintion.

4.3 An Infinite-Dimensional Linear Program

Now, we use f and f_{err} to conservatively estimate $\mathcal{R}_{\text{SOS}}$ by formulating an infinite-dimensional LP on continuous functions. To do so, we use a pair of linear operators,

$$\mathcal{L}_f : AC(T_{\text{plan}} \times X \times K) \to C(T_{\text{plan}} \times X \times K) \text{ and} \tag{4.9}$$

$$\mathcal{L}_{f_{\text{err}}} : AC(T_{\text{plan}} \times X \times K) \to C(T_{\text{plan}} \times X \times K), \tag{4.10}$$

where $AC(D)$ (resp. $C(D)$) denotes the set of absolutely continuous (resp. continuous) functions $D \to \mathbb{R}$. Given a test function $g : T_{\text{plan}} \times X \times K \to \mathbb{R}$, these operators perform the following:

$$\mathcal{L}_f g(t,x,k) = \frac{\partial g}{\partial t}(t,x,k) + (\nabla_x g \cdot f)(t,x,k) \tag{4.11}$$

$$\mathcal{L}_{f_{\text{err}}} g(t,x,k) = \frac{\partial g}{\partial t}(t,x,k) + (\nabla_x g \cdot f_{\text{err}})(t,x,k), \tag{4.12}$$

where ∇_x takes the gradient (of g) with respect to the coordinates of X. In other words, these operators take the total derivative of g with respect to the vector fields f and f_{err}, hence their linearity.

Now we set up the following LP, adapted from [MVTT14, Program (D)]:

$$\inf_{g_{\text{dyn}}, g_{\text{stat}}, d} \int_{X \times K} g_{\text{stat}}(x,k) d\lambda_{X \times K} \tag{4.13}$$

$$\text{s.t} \quad -\mathcal{L}_f g_{\text{dyn}}(t,x,k) - d(t,x,k) \geq 0, \tag{4.14}$$

$$\mathcal{L}_{f_{\text{err}}} g_{\text{dyn}}(t,x,k) + d(t,x,k) \geq 0, \tag{4.15}$$

$$-\mathcal{L}_{f_{\text{err}}} g_{\text{dyn}}(t,x,k) + d(t,x,k) \geq 0, \tag{4.16}$$

$$d(t,x,k) \geq 0 \tag{4.17}$$

$$-g_{\text{dyn}}(0,x,k) \geq 0 \tag{4.18}$$

$$g_{\text{stat}}(x,k) \geq 0 \tag{4.19}$$

$$g_{\text{stat}}(x,k) + g_{\text{dyn}}(t,x,k) - 1 \geq 0, \tag{4.20}$$

where (4.14)–(4.17) and (4.20) are on $T_{\text{plan}} \times X \times K$, (4.18) is on $X_0 \times K$, and (4.19) is on $X \times K$. The given data for this problem are the models f and f_{err} and the sets T_{plan}, X, and K. The infimum is taken over $g_{\text{dyn}}, g_{\text{stat}}, d \in C^1(T_{\text{plan}} \times X \times K)$.

We now provide some insight into this program. Note that this program is the dual to an infinite-dimensional program on measures [MVTT14, Program (P)]; the supports of these measures represent the sets X_0 and $\mathcal{R}_{\text{SOS}}$, plus the reachable sets of the disturbances $\delta \in L_\delta$. The decision variable g_{dyn} is analogous to a Lyapunov function along trajectories produced by $f + f_{\text{err}} \cdot d$, starting from initial conditions in X_0, as evidenced by the constraints (4.14)–(4.16). We use the subscript "dyn" because the 0-sublevel set of g_{dyn} contains states in X and associated *times* that are reached by the planning + tracking error models (we prove this statement below). That is, g_{dyn} allows us to express the robot's time-varying motion, and is later used to formulate online trajectory optimization constraints for *dynamic* environments. Similarly, g_{stat} allows us to express the robot's motion in $X \times K$ over all $t \in T_{\text{plan}}$, thereby enabling us to formulate trajectory optimization constraints at runtime for *static* environments by inspecting the 1-superlevel set of g_{stat} (this follows from (4.20), as we prove shortly). Finally, d allows us to represent all $\delta \in L_\delta$; that is, it stands in for the "disturbance" used to add the tracking error to the planning model. To see this, combine (4.15)–(4.17) to get $|\mathcal{L}_{f_{\text{err}}} g_{\text{dyn}}(t,x,k)| \leq d(t,x,k)$.

We now check that this program does indeed conservatively approximate the FRS:

Theorem 4.3. *If* $(g_{\text{dyn}}, g_{\text{stat}}, d)$ *is a feasible solution to* (4.13)*, then* g_{dyn} *is non-positive and decreasing along trajectories given by* $f + f_{\text{err}} \cdot \delta$ *for any* $\delta \in L_\delta$*. That is, if* $(t,x,k) \in \mathcal{R}_{\text{SOS}}$*, then* $g_{\text{dyn}}(t,x,k) \leq 0$.

Proof. We use a Lyapunov-style argument. Notice from (4.18) that $g_{\text{dyn}}(0,\tilde{x}_0,k) \leq 0$ for all $\tilde{x}_0 \in X_0$ and $k \in K$. So, for any $t \in T_{\text{plan}}$, $k \in K$, and $\delta \in L_\delta$, we have

$$\begin{aligned} g_{\text{dyn}}(t,x(t;k),k) &= g_{\text{dyn}}(0,x(0;k),k) + \int_0^t \Big(\mathcal{L}_f g_{\text{dyn}}(\tau,x(\tau;k),k)\Big)d\tau + \\ &\quad + \int_0^t \Big(\mathcal{L}_{f_{\text{err}}} g_{\text{dyn}}(\tau,x(\tau;k),k)\cdot\delta(\tau)\Big)d\tau \qquad (4.21) \\ &\leq g_{\text{dyn}}(0,x(0;k),k) + \int_0^t \mathcal{L}_f g_{\text{dyn}}(\tau,x(\tau;k),k) + \\ &\quad + \int_0^t d(\tau,x(\tau;k),k)d\tau \qquad (4.22) \\ &\leq g_{\text{dyn}}(0,x(0;k),k). \qquad (4.23) \end{aligned}$$

Here, (4.21) follows from the Fundamental Theorem of Calculus, (4.22) follows from (4.15) and (4.16), and (4.23) follows from (4.14). □

Corollary 4.4. *If* $\exists\, t \in T_{\text{plan}}$ *for which* $(t,x,k) \in \mathcal{R}_{\text{SOS}}$*, then* $g_{\text{stat}}(x,k) \geq 1$.

Proof. From (4.19), $g_{\text{stat}}(x,k) \geq 0$ on all of $X \times K$. From (4.20) it follows that $g_{\text{stat}}(x,k) \geq 1 - g_{\text{dyn}}(t,x,k)$ for all $t \in T_{\text{plan}}$. The desired result then follows from Theorem 4.3. □

4.4 Implementing the LP with SOS Programming

We now approximate (4.13) with finite-degree SOS polynomials.

4.4.1 SOS Polynomials

To proceed, we require the following notation and assumptions. Let $\mathbb{R}[y]$ denote the ring of polynomials in the variable y, and let $\mathbb{R}_l[y]$ denote the polynomials in y up to degree l. We require polynomial representations of the dynamics and domain of the LP above:

Assumption 4.5. *The models f and f_{err} are polynomials of finite degree. The sets X, X_0, and K have the following semi-algebraic representations:*

$$X = \{x \in \mathbb{R}^{n_X} \mid h_X^{(i)}(x) \geq 0 \; \forall \, i = 1, \cdots, n_X\}, \tag{4.24}$$

$$X_0 = \{x \in \mathbb{R}^{n_X} \mid h_{X_0}^{(i)}(x) \geq 0 \; \forall \, i = 1, \cdots, n_X\}, \text{ and} \tag{4.25}$$

$$K = \{k \in \mathbb{R}^{n_K} \mid h_K^{(i)}(k) \geq 0 \; \forall \, i = 1, \cdots, n_K\}, \tag{4.26}$$

where all $h_K^{(i)} \in \mathbb{R}[k]$, all $h_X^{(i)}, h_{X_0}^{(i)} \in \mathbb{R}[x]$. Finally, there exists $n \in \mathbb{N}$ such that, for any $q = (t, x, \tilde{x}_0, k) \in T_{\text{plan}} \times X \times X_0 \times K$, $n - \|q\|_2^2 \geq 0$.

This last assumption is required by [Las10, Theorem 2.15]. Also, notice that T_{plan} admits a semi-algebraic representation:

$$h_{T_{\text{plan}}}(t) = t \cdot (t_{\text{f}} - t). \tag{4.27}$$

Before proceeding, we note that it is critical to scale the robot models and spaces correctly:

Remark 4.6. *The planning model f and tracking error model f_{err} typically represent trajectories that attain values of magnitude greater than 1 in each state. However, when representing the FRS with SOS polynomials, we must scale f and f_{err}, along with the spaces T_{plan}, X, X_0, and K, to be contained within an interval of $[-1, 1]$ in each state/coordinate. This is because the polynomial representation of the robot's FRS can become numerically unstable when we are evaluating polynomials of high degree (e.g., degree 12) on values larger than 1.*

Now, to ease notation, we collect the polynomials representing these sets in the following subsets of $\mathbb{R}[t]$, $\mathbb{R}[x]$, and $\mathbb{R}[k]$:

$$H_{T_{\text{plan}}} = \{h_{T_{\text{plan}}}\}, \tag{4.28}$$
$$H_X = \{h_X^{(1)}, \cdots, h_X^{(n_X)}\}, \tag{4.29}$$
$$H_{X_0} = \{h_{X_0}^{(1)}, \cdots, h_{X_0}^{(n_X)}\}, \text{ and} \tag{4.30}$$
$$H_K = \{h_K^{(1)}, \cdots, h_K^{(n_K)}\}. \tag{4.31}$$

Now, we define sets of SOS polynomials. Let $Q_{2l}(H_{T_{\text{plan}}}, H_X, H_K)$ be the set of polynomials $p \in \mathbb{R}_{2l}[t, x, k]$ that can be expressed as

$$p = s^{(0)} + s^{(1)} h_{T_{\text{plan}}} + \sum_{i=1}^{n_X} s^{(i+1)} h_X^{(i)} + \sum_{i=1}^{n_K} s^{(i+n_X+2)} h_K^{(i)}, \tag{4.32}$$

for some polynomials $\{s^{(i)}\}_{i=1}^{n_X+n_K+2} \subset \mathbb{R}_{2l}[t, x, k]$ that are sums-of-squares of other polynomials. Similarly, define $Q_{2l}(H_X, H_K)$ and $Q_{2l}(H_{X_0}, H_K) \subset \mathbb{R}_{2l}[x, k]$. Note, this use of Q_{2l} is unrelated to the configuration space Q; we use this notation to be consistent with our prior use in the literature [KVB$^+$20].

Note, by Schmüdgen's Positivstellensatz, all such p are non-negative on the (compact) semi-algebraic domains [Las10, Theorem 2.14]. This enables our implementation, since the constraints in the LP require continuous functions that are positive on particular compact sets. By searching over positive polynomials, we ensure that these constraints are satisfied.

4.4.2 SOS Relaxation of the Infinite-Dimensional LP

We now define the l^{th} order relaxed SOS program representation of (4.13):

$$\inf_{g_{\text{dyn},l}, g_{\text{stat},l}, d_l} \quad y_{X\times K}^T \text{vec}(g_{\text{stat},l}) \tag{4.33}$$
$$\text{s.t.} \quad -\mathcal{L}_f g_{\text{dyn},l} - d_l \in Q_{2l_f}(H_{T_{\text{plan}}}, H_X, H_K) \tag{4.34}$$
$$\mathcal{L}_{f_{\text{err}}} g_{\text{dyn},l} + d_l \in Q_{2l_{\text{err}}}(H_{T_{\text{plan}}}, H_X, H_K) \tag{4.35}$$
$$-\mathcal{L}_{f_{\text{err}}} g_{\text{dyn},l} + d_l \in Q_{2l_{\text{err}}}(H_{T_{\text{plan}}}, H_X, H_K) \tag{4.36}$$
$$d_l \in Q_{2l}(H_{T_{\text{plan}}}, H_X, H_K) \tag{4.37}$$
$$-g_{\text{dyn},l}(0, \cdot) \in Q_{2l}(H_{X_0}, H_K) \tag{4.38}$$
$$g_{\text{stat},l} \in Q_{2l}(H_X, H_K) \tag{4.39}$$
$$g_{\text{stat},l} + g_{\text{dyn},l} - 1 \in Q_{2l}(H_{T_{\text{plan}}}, H_X, H_K), \tag{4.40}$$

where the infimum is taken over the polynomials $g_{\text{dyn}}, g_{\text{stat},l}, d_l \in \mathbb{R}_{2l}[t, x, k]$. The vector $y_{X\times K}$ contains moments associated with the Lebesgue measure $\lambda_{X\times K}$, so

$$\int_{X\times K} g_{\text{stat}}(x,k)d\lambda_{X\times K} = y_{X\times K}^{\top}\text{vec}(g_{\text{stat}}) \tag{4.41}$$

for any $g_{\text{stat},l} \in \mathbb{R}_{2l}[x, k]$ [MVTT14]. The numbers l_f (resp. l_{err}) are the smallest integers so that $2l_f$ (resp. $2l_{\text{err}}$) are greater than the total degree of $\mathcal{L}_f g_{\text{dyn},l}$ (resp. $\mathcal{L}_{f_{\text{err}}} g_{\text{dyn},l}$). Note, this means that the total degree of (4.33) scales with the degree of the planning model and tracking error model.

To implement (4.33), we consider the dual program, which is a semi-definite program (SDP) [Las10, MVTT14].

Importantly, Theorem 4.3 and Corollary 4.4 hold for the relaxed program (4.33) [MVTT14, Theorem 6]. In other words, the 0-sublevel set of $g_{\text{dyn},l}$ and the 1-superlevel set of $g_{\text{stat},l}$ both contain trajectories of the planning model plus tracking error (where $g_{\text{dyn},l}$ also included the time component of any such trajectory). This means that $g_{\text{dyn},l}$ and $g_{\text{stat},l}$ overapproximate the FRS, which is important for proving safety: if the subset of the overapproximated FRS corresponding to a trajectory parameter lies outside of an obstacle, then the robot also lies outside of the obstacle. Note, as the program degree l increases, the overapproximation of $\mathcal{R}_{\text{SOS}}$ with $g_{\text{dyn},l}$ and $g_{\text{stat},l}$ becomes provably less conservative [MVTT14, Theorem 7].

4.4.3 Sums-of-Squares Memory Usage

To motivate the next section, and to make the reader aware of the potential limitations of this SOS approach, we discuss the memory usage required by our implementation of (4.33), which uses Spotless [TPM13] to transform the SOS program into an SDP that is then solved with MOSEK [Mos10].

Solving (4.33) is memory-intensive. To see why, first note that the monomials of each polynomial are free variables (a polynomial of degree $2l$ and dimension n has $\binom{2l+n}{n}$ monomials); each free variable is stored as a 64 bit double. The memory required by (4.33) grows as $\mathrm{O}((n+1)^l)$ for fixed l, and $\mathrm{O}(l^{n+1})$ for fixed n [MVTT14, Section 4.2]. As a second-order solver, MOSEK computes the Hessian of each constraint in (4.33) [Mos10, Section 11.4], which requires memory proportional to the number of free variables squared. To estimate the number of free variables generated by (4.33), one can sum the monomials in each decision variable polynomial ($g_{\text{dyn},l}, g_{\text{stat},l}, d_l$, which are degree $2l$, and the s polynomials as in (4.32) used to produce the SOS constraints for each semi-algebraic set, for which the degree is specified by the degree of the program).

Consider the Segway planning model in Running Example 3.5, and suppose we use a tracking error model of degree 3; note, the model has 5 dimensions $\dim(T_{\text{plan}} \times X \times K)$. Solving the $l = 5$

case of (4.33) requires approximately 1.4×10^5 free variables, and used approximately 100 GB of memory. We were unable to solve $l = 6$. When testing (4.33) on a 7-D system, we found that $l = 3$ produced 1.1×10^5 free variables, and used ≈ 500 GB of memory; we were unable to solve the $l = 4$ case on a computer with 3.5 TB of memory.

4.5 System Decomposition

To address the memory challenges presented above, we now present a system decomposition approach for computing the FRS with SOS programming. Here, we first compute an FRS for separate subsystems of the planning model plus tracking error, then reconstruct the FRS of the full system using the lower-dimensional FRSes. This is an adaptation of [CHV$^+$18] from Hamilton-Jacobi reachability to SOS reachability. Note that, while recovering the exact FRS of the full system is not always possible, the recovered FRS is a guaranteed overapproximation, which is useful for collision avoidance purposes. To proceed, we first define self-contained subsystems, then present an infinite-dimensional LP to reconstruct an FRS given FRSes of these subsystems, and finally present a SOS implementation.

4.5.1 Self-contained Subsystems

We define **self-contained subsystems** as follows; note, we present the case for two subsystems, but this method can generalize to any number. Consider a planning model f with state $x \in X$, which we refer to as the **full system**. Suppose we can write $x = (x_1, x_2, x_\mathrm{s})$, with dynamics

$$\begin{aligned}
\dot{x}_1(t) &= f_1(t, x_1(t;k), x_\mathrm{s}(t;k), k) \\
\dot{x}_2(t) &= f_2(t, x_2(t;k), x_\mathrm{s}(t;k), k) \\
\dot{x}_\mathrm{s}(t) &= f_\mathrm{s}(t, x_\mathrm{s}(t;k), k),
\end{aligned} \tag{4.42}$$

and notice that f_1 does not depend on x_2 , f_2 does not depend on x_1, and, f_s does not depend on either x_1 or x_2. Then we define $z_1 = (x_1, x_\mathrm{s})$ and $z_2 = (x_2, x_\mathrm{s})$ as the coordinates of the self-contained subsystems. The subscript "s" denotes that the coordinates x_s are shared between both subsystems. Let Z_1 and Z_2 denote the subspaces of X that contain the z_1 and z_2 states, respectively; we assume that these sets admit semi-algebraic representations, just as with X, X_0, and K.

4.5.2 Subsystem FRSes

Now, to compute the FRS for each subsystem, we specify the planning and tracking error models as

$$\dot{z}_i(t) = \begin{bmatrix} \dot{x}_i(t) \\ \dot{x}_{\text{s}}(t) \end{bmatrix} = \begin{bmatrix} f_i(t, x_i(t;k), x_{\text{s}}(t;k), k) \\ f_s(t, x_{\text{s}}(t;k), k) \end{bmatrix} + \begin{bmatrix} f_{\text{err},i}(t,k) \cdot \delta_i(t) \\ f_{\text{err,s}}(t,k) \cdot \delta_{\text{s}}(t) \end{bmatrix}, \tag{4.43}$$

where $(\delta_i(t), \delta_{\text{s}}(t)) \in L^1(T_{\text{plan}}, [-1,1]^{\dim(Z_i)})$ is the disturbance for the self-contained subsystem. We then solve (4.13), replacing X with Z_i and f and f_{err} with the models in (4.43).

4.5.3 FRS Reconstruction

Now we *reconstruct* the FRS of the full system. Let $(g_{\text{dyn},i}, g_{\text{stat},i}, d_i)$ be a feasible solution to (4.13) for self-contained subsystem i, with $i = 1, 2$. Then define

$$G_{\text{rec}} = \Big\{(x,k) \in \times X \times K \mid \exists\, t \in T_{\text{plan}} \text{ s.t.} \tag{4.44}$$

$$g_{\text{dyn},1}(t, \text{proj}_{Z_1}(x), k) \leq 0 \text{ and } g_{\text{dyn},2}(t, \text{proj}_{Z_2}(x), k) \leq 0\Big\}, \tag{4.45}$$

where the subscript "rec" denotes reconstruction. In other words, we reconstruct the FRS using the functions $g_{\text{dyn},i}$ that are negative and decreasing along trajectories of each subsystem; the reconstructed FRS contains points in X that are reached by *both* subsystems (which can extend to *all* subsystems if there are more than two). To find G_{rec}, we pose the following infinite-dimensional LP:

$$\inf_{g_{\text{rec}}} \quad \int_{X\times K} g_{\text{rec}}(x,k)\, d\lambda_{X\times K} \tag{4.46}$$

$$\text{s.t.} \quad g_{\text{rec}}(x,k) \geq 1 \;\forall\, (x,k) \in G_{\text{rec}} \tag{4.47}$$

$$g_{\text{rec}}(x,k) \geq 0 \;\forall\, (x,k) \in X \times K. \tag{4.48}$$

We implement (4.46) as a SOS program as follows. Suppose we solve (4.33) for each self-contained subsystem with degree l, to get $g_{\text{dyn},l,1}, g_{\text{dyn},l,2} \in \mathbb{R}_{2l}[t,x,k]$; note, per (4.44), we do not need to hold on to the other decision variables of (4.33), so we omit them here to ease notation. Let

$$H_{\text{dyn}} = \{-g_{\text{dyn},l,1}, -g_{\text{dyn},l,2}\}, \tag{4.49}$$

and let $m \in \mathbb{N}, m \geq l$. Then we reconstruct the FRS with the following SOS program:

$$\inf_{g_{\text{rec},m}} \quad \int_{X \times K} y_{X \times K}^{\top} \text{vec}(g_{\text{rec},m}) \tag{4.50}$$

$$\text{s.t.} \quad g_{\text{rec},m} - 1 \in Q_{2m}(H_{\text{dyn}}, H_{T_{\text{plan}}}, H_X, H_K) \tag{4.51}$$

$$g_{\text{rec},m} \in Q_{2m}(H_X, H_K), \tag{4.52}$$

where $Q_{2m}(\cdot)$ denotes the degree $2m$ polynomials that can be written as in (4.32).

We confirm that this program reconstructs the FRS of the full system:

Theorem 4.7. *Suppose $g_{\text{dyn},l,1}, g_{\text{dyn},l,2} \in \mathbb{R}_{2l}[t, x, k]$ are parts of the feasible solutions to* (4.33) *for each self-contained subsystem with degree l. Suppose $g_{\text{rec},m}$ is a feasible solution to* (4.50) *(using $g_{\text{dyn},l,1}$ and $g_{\text{dyn},l,2}$). Then $\mathcal{R}_{\text{SOS}}$ is a subset of the 1-superlevel set of $g_{\text{rec},m}$.*

Proof. Suppose that $x : T_{\text{plan}} \to X$ is a trajectory of the full system with $\dot{x}(t; k) = f(t, x(\cdot), k) + f_{\text{err}}(t, k) \cdot \delta(t)$, and $\delta \in L_\delta$ is a disturbance profile for the full system. By definition, $\mathcal{R}_{\text{SOS}}$ contains every such trajectory x; so, we must show that $g(x(t; k), k) \geq 1$ for all $t \in T_{\text{plan}}$. By Theorem 4.3 and [MVTT14, Theorem 6], $g_{\text{dyn},l,1}(t, \text{proj}_{Z_1}(x(t; k)), k) \leq 0$ for subsystem 1, and similarly with $g_{\text{dyn},l,2}$ for subsystem 2. This means that $(x(t; k), k) \in G_{\text{rec}}$ for any $t \in T_{\text{plan}}$. Since $g_{\text{rec}} \geq 1$ on G_{rec}, we are done. $\square$

Another way to think of G_{rec} is as the *intersection of the back-projections* of the FRS of each subsystem into the full planning space X. Let $z_1(t; k) = \text{proj}_{Z_1}(x(t; k))$ for all $t \in T_{\text{plan}}$, and similarly z_2, where x is the trajectory from the proof above. From [CHV+18, Lemma 1], we have

$$x(t; k) \in \left\{ x \in \text{proj}_{Z_1}^{-1}(z_1(t; k)) \cap \text{proj}_{Z_2}^{-1}(z_2(t; k)) \right\}, \tag{4.53}$$

where $\text{proj}_S^{-1}(x) = \{x \in X \mid \text{proj}_S(x)\}$ is the *back-projection operator* from a subspace S to the full system space X.

4.6 The FRS Over Small Time Intervals

We now present a method for breaking T_{plan} into several time intervals and computing an FRS with SOS programming on each one [VLK+19]. This approach enables significantly less conservative trajectory planning in dynamic environments, because the resulting FRS representation produces fewer constraints for online trajectory optimization than the representations presented earlier in this chapter.

4.6.1 Time Interval Motivation

To motivate this section, we discuss how the FRS is used for online trajectory planning. Let g_{dyn} and g_{stat} be part of a feasible solution to 4.13. Consider an obstacle $O : T_{\text{plan}} \to \text{pow}\,(W)$ that has been mapped to the planning frame.

Suppose the obstacle is static. Then, to choose trajectories that avoid this obstacle, by Theorem 4.3, we must ensure that $g_{\text{stat}}(x, k) < 1$ for all $x \in O(t)$ where t is any time in T_{plan}. This requires an infinite number of constraints if $O(t)$ contains an infinite number of points (e.g., if $O(t)$ is a polygon in W). In §5, we present a method for conservatively representing any such $O(t)$ with a finite number of discrete points *at online*, resolving the issue in the case of static obstacles. In the case of static obstacles, this results in safe but fast trajectory planning.

However, if the obstacle is dynamic, then we must ensure that $g_{\text{stat}}(t, x, k) > 0$ for every $t \in T_{\text{plan}}$ and all $x \in O(t)$. Therefore, this also requires an infinite number of constraints, but for both t and x. While dynamic obstacles also admit a conservative, discretized, finite representation in §5.7.3, unfortunately, we find that discretizing time results in an unideal tradeoff between conservatism and computational expense. In practice, such a time discretization means that a wheeled robot cannot plan with respect to more than one or two dynamic obstacles at online, which is impractical for, e.g., an autonomous car surrounded by pedestrians.

To combat the challenges with this online dynamic obstacle discretization, here, we partition T_{plan} into small intervals, and compute the FRS over each such interval offline. Then, at online, by treating dynamic obstacles as static in each small time interval, we can leverage the static obstacle discretization mentioned above (see §5.7.4). Remarkably, doing so reduces *both* conservatism and computation time for online trajectory planning.

4.6.2 A Secondary Infinite-Dimensional LP

Our approach is to use the solution to the original infinite-dimensional LP 4.13, to construct a similar LP for of a finite number of time intervals. In other words, we solve for the functions representing the FRS over the entirety of T_{plan}, then use them to find the FRS broken into time intervals, hence the notion of a *secondary* LP.

Let $n_{\text{RS}} \in \mathbb{N}$ be a number of time intervals, with the subscript as a reminder that this integer is used for the reachable set. Let $\Delta_t = t_{\text{f}}/n_{\text{RS}}$, so that

$$T_{\text{plan}} = [0, \Delta_t] \cup [\Delta_t, 2\Delta_2] \cup \cdots \cup [t_{\text{f}} - \Delta_t, t_{\text{f}}] \tag{4.54}$$

$$= I^{(1)} \cup I^{(2)} \cup \cdots \cup I^{(n_{\text{RS}})} \tag{4.55}$$

That is, each $I^{(i)} = [(i-1)\Delta_t, i \cdot \Delta_t]$ for $i = 1, \cdots, n_{\text{RS}}$. We refer to this as a **partition** of T_{plan} in a minor abuse of vocabulary ($I^{(i)} \cap I^{(i+1)} \neq \emptyset$, but the intersection is only a single point). Recall that, for SOS programming, we must define the domain of the cost and constraints as a compact sets.

Suppose that, for a planning model f and tracking error model f_{err}, we have computed g_{dyn} and g_{stat} as feasible solutions to (4.13). Then, we pose the following LP on continuous functions:

$$\inf_{g_{\text{stat}}^{(i)}} \quad \int_{X\times K} g_{\text{stat}}^{(i)}(x,k)d\lambda_{X\times K} \tag{4.56}$$

$$\text{s.t.} \quad g_{\text{stat}}^{(i)}(x,k) + g_{\text{dyn}}(t,x,k) - 1 \geq 0 \text{ on } I^{(i)} \times X \times K \tag{4.57}$$

$$g_{\text{stat}}^{(i)}(x,k) \geq 0 \text{ on } X \times K. \tag{4.58}$$

Notice the similarity between (4.57) and (4.20) (wherein $g_{\text{stat}} \geq 1 = g_{\text{dyn}}$).

Lemma 4.8. *Suppose $g^{(i)}$ is feasible to* (4.56)*. If there exists $t \in I^{(i)}$ such that $(t,x,k) \in \mathcal{R}_{\text{SOS}}$, then $g_{\text{stat}}^{(i)}(x,k) \geq 1$.*

Proof. This follows from (4.57) and Corollary 4.4. That is, since $g_{\text{dyn}}(t,x,k) \leq 0$ on trajectories in $I^{(i)} \times X \times K$, it follows that $g_{\text{stat}}^{(i)}(x,k) \geq 1$ on those same trajectories. □

4.6.3 SOS Relaxation

As before, we apply Lasserre's hierarchy to relax (4.56) to a finite-degree SOS program of degree l:

$$\inf_{g_{\text{stat},l}^{(i)}} \quad y\top_{X\times K}\text{vec}(g_{\text{stat},l}^{(i)}) \tag{4.59}$$

$$\text{s.t.} \quad g_{\text{stat},l}^{(i)} + g_{\text{dyn},l} - 1 \in Q_{2l}(H_{I^{(i)}}, H_X, H_K) \tag{4.60}$$

$$g_{\text{stat},l}^{(i)} \in Q_{2l}(H_X, H_K), \tag{4.61}$$

where $g_{\text{dyn},l}$ is part of a feasible solution to (4.33) and $H_{I^{(i)}} = \{h_{I^{(i)}}\}$ for which

$$h_{I^{(i)}}(t) = (t - t^{(i)})(t^{(i+1)} - t). \tag{4.62}$$

Note that applying this time interval method requires first solving (4.33), *then* solving (4.59) for every $I^{(i)}$. Therefore, the offline computation time is greatly increased by this method. However, this penalty is worth paying to enable faster and less conservative online trajectory planning, as mentioned earlier.

4.7 Recovering the Original FRS

Recall that, in §4.2, we simplified the FRS definition by taking the union over all initial conditions. In practice, this would be very conservative, because the tracking error model f_{err} would have to hold as in Assumption 4.1 for *every* possible initial condition of the robot. To combat this, we use **FRS swapping**. The idea, in essence, is to partition the space $X_{\text{hi},0}$ of initial conditions into a finite number of subsets, then compute the simplified FRS (i.e. $\mathcal{R}_{\text{SOS}}$) on each subset. Online, at the beginning of each receding-horizon planning iteration, we select the particular FRS corresponding to the robot's initial condition; in other words, we *swap* to the correct FRS. Note, we have already seen this logic in Algorithm 1 (see `GetFRS` on Line 4). Importantly, the union of all such simplified FRSes lets us conservatively recover the original FRS, $\mathcal{R}_{\text{FRS}}$.

Note that a naïve implementation of FRS swapping, where we partition the entire space $X_{\text{hi},0} \subset \mathbb{R}^{n_{\text{hi}}}$ of initial conditions, would be intractable due to the dimension n_{hi} of the high-fidelity model. However, notice that $X_{\text{hi},0}$ occupies zero volume in the subspace X. This means we need only partition the robot's initial generalized velocity space, not its generalized coordinate space. We find that this makes FRS swapping tractable across a variety of robot morphologies.

FRS swapping lets us conservatively *recover the original FRS*, $\mathcal{R}_{\text{FRS}} \subseteq T_{\text{plan}} \times X_{\text{hi},0} \times X \times K$. Consider the i^{th} receding-horizon planning iteration, with initial condition $x_{\text{hi},0}^{(i)}$. Where one would choose $\mathcal{R}_{\text{FRS}}^{(i)}$ specific to $x_{\text{hi},0}^{(i)} \in X_{\text{hi},0}$ as in §3, we instead choose $\mathcal{R}_{\text{SOS}}^{(i)}$ for which $\text{proj}_{X_{\text{hi},0}}(x_{\text{hi},0}^{(i)}) \in X_{\text{hi},0}^{(j)}$, where $X_{\text{hi},0}^{(j)}$ is the j^{th} subset in the partition of $X_{\text{hi},0}$. By *conservative*, we mean that, if $(t, x_{\text{hi},0}, x, k) \in \mathcal{R}_{\text{FRS}}$, then there exists a subset $X_{\text{hi},0}^{(j)}$ of $X_{\text{hi},0}$ (by construction) such that $(t, x, k) \in \mathcal{R}_{\text{SOS}}^{(i)}$. The inclusion does not necessarily hold in the opposite direction.

This idea of FRS swapping lets us forecast our approach of computing the ERS in §7. As hinted earlier, the tracking error function f_{err} is typically smaller for a smaller range of initial conditions. Therefore, by partitioning $X_{\text{hi},0}$, we can compute a separate f_{err} for each subset of $X_{\text{hi},0}$; since the magnitude of f_{err} is smaller for some subsets of $X_{\text{hi},0}$, the corresponding FRS is smaller for those same subsets. A smaller FRS is less likely to intersect with obstacles, and thereby eliminates less choices of plans online; so, FRS swapping reduces conservatism. The takeaway here is, while f_{err} is a tracking error model representation of the ERS specific to the SOS approach, this notion of partitioning $X_{\text{hi},0}$ enables us to compute the ERS less conservatively than if we were to subsume all tracking error over all initial conditions.

4.8 Online Planning

We now discuss how the polynomial representation of the FRS is used online. Suppose $g_{\text{dyn},l}, g_{\text{stat},l}$ are part of a feasible solution to (4.33), computed offline. Suppose that the robot is in its i^{th}

planning iteration, with an obstacle reachable set $\mathcal{R}_{\text{obs}}^{(i)} \subset T_{\text{plan}} \times W \times K$ as in (3.33). Recall that $\mathcal{R}_{\text{obs}}^{(i)}$ is a prediction of obstacles that has been mapped to the robot's planning frame. The purpose of computing the FRS is to enable us to identify the unsafe plans as in (3.37), which we restate here:

$$K_{\text{unsf}}^{(i)} \subseteq \text{proj}_K(\mathcal{R}_{\text{FRS}}^{(i)} \cap \mathcal{R}_{\text{obs}}). \tag{4.63}$$

4.8.1 Generic Constraint Formulation

To see how we represent the intersection of the FRS and ORS in (4.63), notice that the 0-sublevel set of $g_{\text{dyn},l}$ conservatively represents the FRS:

$$\text{proj}_{T_{\text{plan}} \times W}(\mathcal{R}_{\text{FRS}}^{(i)}) \subseteq \{(t,x) \in T_{\text{plan}} \times W \mid \exists\, k \in K \text{ s.t. } g_{\text{dyn},l}(t,x,k) \leq 0\}\,, \tag{4.64}$$

which follows from Theorem 4.3 and [MVTT14, Theorem 6]. It follows that, for any $(t,x) \in \text{proj}_{T_{\text{plan}} \times W}(\mathcal{R}_{\text{obs}})$ and $k \in K$,

$$k \in K_{\text{unsf}}^{(i)} \implies g_{\text{dyn},l}(t,x,k) \leq 0. \tag{4.65}$$

Therefore, we can rewrite the trajectory optimization program (3.38) to find the i^{th} plan, $k^{(i)}$, as follows:

$$k^{(i)} = \underset{k \,\in\, K}{\text{argmin}} \quad \mathsf{cost}(k) \tag{4.66}$$

$$\text{s.t.} \quad g_{\text{dyn},l}(t,x,k) > 0 \;\forall\, (t,x) \in \text{proj}_{T_{\text{plan}} \times W}(\mathcal{R}_{\text{obs}}^{(i)}) \tag{4.67}$$

$$k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}). \tag{4.68}$$

4.8.2 Static Obstacles Formulation

Notice that (4.67) presumes dynamic obstacles in $\mathcal{R}_{\text{obs}}^{(i)}$. If $\mathcal{R}_{\text{obs}}^{(i)}$ only contains static obstacles, then one can instead use the constraint

$$g_{\text{stat},l}(x,k) < 1 \;\forall\, x \in \text{proj}_W(\mathcal{R}_{\text{obs}}^{(i)}). \tag{4.69}$$

Note by Theorem 4.7 that this constraint formulation still holds if one instead uses the outputs of the system decomposition SOS program.

4.8.3 Time Interval FRS Formulation

Suppose that we have broken T_{plan} into n_{RS} intervals $I^{(n)}$, $n = 1, \cdots, n_{\text{RS}}$ as in §4.6, and suppose we compute $g^{(n)}_{\text{stat},l}$ for each interval using (4.59). Then, for each interval $I^{(n)}$, we formulate the collision-avoidance constraint as

$$g_{\text{stat},l}(x, k) < 1 \; \forall \; x \in \text{proj}_{I^{(n)} \times W}(\mathcal{R}^{(i)}_{\text{obs}}). \tag{4.70}$$

Notice the similarity between (4.70) and (4.69) In other words, we treat the obstacles in $\mathcal{R}^{(i)}_{\text{obs}}$ as static in each interval $I^{(n)}$.

4.8.4 An Infinite-Dimensional Problem

Unfortunately, the constraints (4.67), (4.69), and (4.70) all would typically need to be satisfied on an infinite number of points, because $\mathcal{R}^{(i)}_{\text{obs}}$ is usually a continuum (for example, if obstacle predictions are represented as polytopes in the workspace). One way to remedy this challenge is to solve an SDP at runtime to find all $k \in K$ that satisfy the constraint [KVJRV17]. However, doing so is not practical for real-time planning [KVB+20]. Therefore, in §5, we propose a finite, discretized obstacle representation that enables real-time planning while conservatively approximating (4.67) and (4.69).

4.9 Chapter Review

The takeaway of this chapter is that one can use SOS programming to compute an FRS representation that contains the motion of a robot in the plane for a continuum of trajectory plans, and that includes tracking error. Importantly, the resulting polynomial representation enables one to generate constraints for online trajectory optimization such that any feasible solution to these constraints is a provably-safe trajectory plan.

4.9.1 Chapter Summary

This chapter presented a sums-of-squares (SOS programming approach to compute the Forward Reachable Set (FRS) offline. We began with a generic formulation [KVJRV17], and noted that it can suffer from memory limitations. To alleviate these issues, we then presented a system decomposition approach to enable computing the SOS FRS for higher-dimensional systems [KVB+20]. Then, noting that the outputs of these SOS programs can be difficult to use with dynamic obstacles, we presented a method for computing the FRS over a prespecified set of time intervals, which enables real-time planning in dynamic environments [VLK$^+$19]. We concluded the chapter by explaining how to use these FRS representations for online planning.

4.9.2 What Is Missing?

However, our online planning formulation, as presented, may result in a numerically-intractable, infinite-dimensional trajectory-optimization problem. To address this challenge, in §5, we present an obstacle representation that enables safe, real-time planning with the SOS FRS. See §5.8.1 for an example usage of this obstacle representation.

CHAPTER 5

A Discretized Obstacle Representation for Safe, Real-Time Planning

In §4, we used a sums-of-squares (SOS) programming approach to compute a robot's Forward Reachable Set (FRS) offline. This approach represents the FRS as polynomial level sets. To use this FRS representation at runtime, we evaluate the polynomial on points representing obstacles in the robot's workspace to determine if a given plan is safe. Unforunately, for common obstacle representations such as occupancy grids or polygons, this may require evaluating the FRS polynomial on a potentially-infinite number of points.

To address this challenge, the present chapter develops a finite, discretized obstacle representation for wheeled robots operating in the plane (i.e., the configuration space is $Q \subseteq \mathsf{SE}(2)$). We prove that, if a trajectory plan avoids each of the discrete points, then the trajectory plan also avoids all obstacles. Note, this chapter summarizes results developed in three papers: [KVB$^+$20, VKL$^+$19, VLK$^+$19].

Importantly, the results developed in this chapter are generalizable outside of RTD. That is, we provide a generic discretized obstacle representation that can be used *by any motion planning algorithm* for fast, *correct* collision checking. However, to ensure *safe* motion planning, the underlying motion planner must be able to certify safety *independent* of any obstacle representation. To that end, we pair this representation with RTD, for which we developed safety guarantees in a generic way in §3.

Note that other discretized obstacle representations exist. For example, one can cover the robot and obstacles with a (finite) set of closed 2-norm balls [VG18], or compute a Euclidean distance transform of obstacles as a (discrete) voxel representation [ZRD$^+$13]. One can also buffer (i.e. dilate, or increase the size of) obstacles to account for continuous-time motion of a robot [LaV06, Chapter 5.3.4]. The novelty of our proposed representation in this chapter is that, instead of requiring the robot to be a certain distance from, e.g., the centers of a finite number of balls, we require the robot to avoid the discrete points themselves. This means that one need not use a set-to-point distance computation to ensure collision avoidance. Such a representation is important, for example, when one represents a robot's motion using polynomial level sets (as we do in §4), where collision avoidance may require solving a non-convex optimization program (e.g. with set-to-point distance as in [Fer00]) or a large semi-definite program [KVJRV17].

The sections of this chapter are as follows. (§5.1) First, we review what it means for a plan to be safe in terms of the FRS, as per §3. (§5.2.1) Then, we formally define several common geometric objects used throughout the chapter, and present a generic geometric definition of the robot's motion through space. We also discuss assumptions on the robot and obstacles. (§5.3) Next, we introduce five geometric quantities used to construct the discretized obstacle representation in the case of static obstacles. (§5.4) We find these quantities by constructing several optimization programs that leverage the robot's geometry. (§5.5) We then propose an algorithm to construct the discretized obstacle using the found geometric quantities. (§5.6) Next, we certify that this discretized obstacle representation ensures safety in static environments. (§5.7) Finally, we extend our representation to dynamic obstacles. (§5.8) To conclude the chapter, we show how to use our discretized obstacle representation with the polynomial FRS representation from §4, review the chapter contributions, and briefly discuss future research directions.

5.1 Discretized Obstacle Motivation

To motivate this chapter, we begin by reviewing our definitions of obstacles and safety. Throughout this chapter, we assume that the robot is in a single planning iteration (e.g., the i^{th} iteration for time horizon $T^{(i)} \subset T$); we avoid the index i to ease notation.

5.1.1 Obstacles and Safety via the FRS

First, we briefly reintroduce obstacles and the obstacle reachable set (ORS). Suppose $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ is a set of obstacles $O^{(n)} : T \to \text{pow}\,(W)$. Recall that $\mathcal{R}_{\text{obs}} \subset T_{\text{plan}} \times W \times K$ is the ORS for the current planning iteration, as in §3.7. Per the ORS definition, if the robot has sensed n_{obs} obstacles, $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$, then, for any $t \in T_{\text{plan}}$,

$$\text{proj}_{\{t\}\times W}(\mathcal{R}_{\text{obs}}) \supseteq \bigcup_{n=1}^{n_{\text{obs}}} O^{(n)}(t). \tag{5.1}$$

We begin this chapter by assuming all obstacles are static, meaning

$$O^{(n)}(t_1) = O^{(n)}(t_2)\ \forall\, n \in \{1, \cdots, n_{\text{obs}}\} \text{ and } t_1, t_2 \in T. \tag{5.2}$$

Furthermore, we have

$$\text{proj}_W(\mathcal{R}_{\text{obs}}) \supseteq \bigcup_{n=1}^{n_{\text{obs}}} O^{(n)}, \tag{5.3}$$

where we have dropped the time notation for each $O^{(n)} \subset W$ since the obstacles are assumed to be static. Note, we extend our approach to dynamic obstacles in §5.7.

Now we review our definition of safety. Recall that, in §3.4.2, the robot is unsafe along a trajectory $x_{\text{hi}} : T \to X_{\text{hi}}$ if there exists some n and t for which

$$\mathsf{FO}(x_{\text{hi}}(t)) \cap O^{(n)} \neq \emptyset. \tag{5.4}$$

Once we introduced the FRS, we were able to redefine safety in a new way, on a plan-by-plan basis as in (3.37). In particular, in each i^{th} receding-horizon iteration, we identify a set of plans K_{unsf} produced by projecting the intersection of the FRS and an obstacle reachable set into the space K:

$$K_{\text{unsf}} \subseteq \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \mathcal{R}_{\text{obs}}), \tag{5.5}$$

where, again, we have dropped the index i denoting the current receding-horizon planning iteration. Note this is a minor abuse of notation; we are conflating the "full" FRS, $\mathcal{R}_{\text{FRS}} \subset T_{\text{plan}} \times X_{\text{hi},0} \times W \times K$, with the FRS for the current planning iteration and initial condition $x_{\text{hi},0}$. We can think of this as the set

$$\mathcal{R}_{\text{FRS}} \;\leftarrow\; \text{proj}_{T_{\text{plan}} \times W \times K}(\mathcal{R}_{\text{FRS}} \cap (T_{\text{plan}} \times \{x_{\text{hi},0}\} \times W \times K)), \tag{5.6}$$

which would usually be denoted $\mathcal{R}_{\text{FRS}}^{(i)}$ (here, $\leftarrow$ denotes assignment of a variable at runtime).

5.1.2 The Discretized Obstacle

Unfortunately, as suggested by (5.1), $\mathcal{R}_{\text{obs}}$ typically contains an infinite number of points (i.e., the set has cardinality $|\mathcal{R}_{\text{obs}}| = \infty$); so, it can be numerically intractable to ensure collision avoidance for every one of these points during online trajectory planning. To resolve this challenge, in this chapter, we seek a finite, discretized representation of $\mathcal{R}_{\text{obs}}$. We call this representation the **discretized obstacle**, which we denote

$$O_{\text{disc}} \subset W, \tag{5.7}$$

for which $|O_{\text{disc}}| < \infty$. The goal of this chapter is to construct O_{disc} such that

$$\text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \mathcal{R}_{\text{obs}}) \subseteq \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap (T_{\text{plan}} \times O_{\text{disc}} \times K)), \tag{5.8}$$

which implies that

$$K_{\text{unsf}} \subseteq \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap (T_{\text{plan}} \times O_{\text{disc}} \times K)) \tag{5.9}$$

by (5.5).

We show examples of point obstacles and discretized obstacles in Figure 5.1.

5.1.3 Incorporating Dynamic Obstacles

Note, when we return to dynamic obstacles in in §5.7, we seek a discretization

$$\left\{\{t^{(n)}\} \times O_{\text{disc}}^{(n)}\right\}_{n=1}^{n_t} \subset T_{\text{plan}} \times W, \tag{5.10}$$

for some $n_t \in \mathbb{N}$ (which we prescribe how to choose). In this case, each $t^{(n)} \in T_{\text{plan}}$ corresponds to a discretized obstacle constructed in a similar manner as O_{disc} for static obstacles.

5.1.4 Unsafe Parameters for a Point Obstacle

To foreshadow the utility of the discretized obstacle, we now identify the set of unsafe trajectory parameters with respect to a single point obstacle.

Lemma 5.1. *Suppose the robot is in a planning iteration at initial condition $x_{\text{hi},0} \in X_{\text{hi}}$. Let the robot's be as in* (3.26)*, but only corresponding to the current initial condition; denote it as $\mathcal{R}_{\text{FRS}} \subset T_{\text{plan}} \times W \times K$. Suppose that, in the current planning iteration, the robot has detected an obstacle $\{o\} \subset W$ with $|\{o\}| = 1$, so that*

$$\mathcal{R}_{\text{obs}} = T_{\text{plan}} \times \mathsf{world2plan}\left(\{o\}\right) \times K \tag{5.11}$$

is the ORS for this planning iteration. Suppose that the robot is not currently intersecting $\{o\}$, and it is tracking a previously-found plan that avoids collision with $\{o\}$. Consider the set

$$K_o = \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \mathcal{R}_{\text{obs}}). \tag{5.12}$$

If the robot tracks any $k \in K_o^{\mathsf{C}}$, then it does not collide with the point obstacle o at any time while tracking that plan k.

Proof. Note that this use of $\mathcal{R}_{\text{FRS}}$ is a minor abuse of notation as in (5.6). Allowing this notation, the claim follows from the definition of the parameterized plans and the definition of the FRS. $\square$

Note that, though the robot avoids collision with the point obstacle in Lemma 5.1, it may come *infinitesimally* close to the point obstacle when tracking some $k \in K_o^{\mathrm{C}}$.

5.2 Definitions and Assumptions

We now define several geometric objects used to represent obstacles and construct the discretized obstacle. We then specify the robot's motion through the world geometrically. Finally, we place assumptions on obstacles.

5.2.1 Geometric Objects

We use several objects from planar geometry. Let $\mathbb{R}^2$ denote the plane. We refer to the canonical coordinate axes of $\mathbb{R}^2$ as the (horizontal) x-axis and the (vertical) y-axis.

Let $I \subset \mathbb{R}^2$ denote a **line segment**, also called an **interval** when it lies along one of the canonical axes of $\mathbb{R}^2$. Let $E_I = \{e_1, e_2\}$ denote the **endpoints** of I, such that I can be written:

$$I = \{e_1 + s \cdot (e_2 - e_1) \mid s \in [0, 1]\}. \tag{5.13}$$

The **length** of I is the quantity $\mathsf{length}(I) = \|e_2 - e_1\|_2$.

Suppose that I is a line segment with distinct endpoints. Then we call the line that passes through both endpoints, denoted

$$L_I = \{e_1 + s \cdot (e_2 - e_1) \mid s \in \mathbb{R}\}, \tag{5.14}$$

the **line defined by** I.

Let $U \subset \mathbb{R}^2$ be an arbitrary set with a boundary, and let $u_1, u_2 \in \partial U$. Then we call the line segment

$$C = \{a_1 + s \cdot (u_2 - u_1) \mid s \in [0, 1]\} \tag{5.15}$$

a **chord** of U. Note, C need not be contained entirely inside U; that is, it may be that $C \not\subset U$, such as when U is non-convex.

A **circle** $\Omega \subset \mathbb{R}^2$ of radius $r > 0$ with center $p \in \mathbb{R}^2$ is the set

$$\Omega = \left\{q \in \mathbb{R}^2 \mid \|p - q\|_2 = r\right\}. \tag{5.16}$$

An **arc** $A \subset \mathbb{R}^2$ is any connected, closed, strict subset of a circle.

5.2.2 Robot Assumptions and Motion

To understand how to relate the motion of the robot's body to the discretized obstacle representation, we now provide a generic, geometric expression for the robot's body, and forward occupancy, and trajectories.

First, we assume the following about the shape of the robot.

Assumption 5.2. *The robot's **body** is a convex, compact set $B \subset W$, with nonzero volume, in the robot's planning frame.*

Note that, if the robot's body is not convex, but is compact, it can be bounded within a convex hull or rectangular bounding box [FS75]. We emphasize that this method applies to *arbitrary* convex, compact robot bodies.

The reader may recall the initial condition set X_0 in §4. Indeed, we treat B as X_0 in the case where $X = \mathbb{R}^2$; and, if $X = \mathsf{SE}(2)$, we assume that B is equivalent to X_0 at a rotation of 0 rad. To this end, we assume that there exists a point $c_0 = \mathrm{proj}_{\mathbb{R}^2}(x_0) \in B$ that is the **center of rotation** for the robot in its local coordinate frame (the meaning of this will become clear in the following paragraph).

Second, we express forward occupancy as follows. Notice that, the robot's motion, as expressed by the high-fidelity model, evolves in $\mathsf{SE}(2)$; that is, for the purpose of collision avoidance, we are concerned with the rotations and translations of the robot's body along any high-fidelity model trajectory. To this end, we define the following object:

Definition 5.3. *We define a **transformation** $H^{(t)} : \mathrm{pow}\left(\mathbb{R}^2\right) \to \mathrm{pow}\left((\mathbb{R}^2\right)$ indexed by a time $t \in T_{\text{plan}}$ and parameterized by a **translation** $p^{(t)} \in \mathbb{R}^2$ and a **rotation** $R^{(t)} \in \mathsf{SO}(2)$, such that, for a singleton set $\{q\} \subset \mathbb{R}^2$, we have*

$$H^{(t)}(\{q\}) = \{R^{(t)} \cdot (q - c_0) + p^{(t)}\}, \tag{5.17}$$

where c_0 is the center of rotation of the robot's body in its local coordinate frame; typically, c_0 is the center of geometry or center of mass.

Note, $t \in T_{\text{plan}}$ because the goal of RTD is to certify collision avoidance for each plan. We apply these transformations to the robot's body to understand the robot's motion through the workspace

$$H^{(t)}B = \{H^{(t)}(q) \mid q \in B\}, \tag{5.18}$$

where we omit parentheses around B to increase readability. To see how this relates to the robot's forward occupancy, suppose the robot is at a state $x_{\text{hi}}(t') \in X_{\text{hi}}$ at a time $t' \in T^{(i)}$ (for example, when tracking a trajectory in the i^{th} receding-horizon planning iteration). We assume that

$$\mathsf{FO}(x_{\text{hi}}(t')) \subseteq H^{(t'-t^{(i)})}B \subset W. \tag{5.19}$$

Recall that $T^{(i)} = [t^{(i)}, t^{(i)} + t_{\text{f}}]$, so $t' - t^{(i)}$ shifts time to $T_{\text{plan}} = [0, t_{\text{f}}]$ to match the index of $H^{(t)}(\cdot)$.

Third, we use transformations to express trajectories of the robot geometrically as follows.

Definition 5.4. *We define a **transformation family** as a set*

$$\begin{aligned} \big\{H^{(t)} \mid t \in T_{\text{plan}},\ (p^{(t)}, R^{(t)}) \in \mathsf{SE}(2) \text{ continuous w.r.t. } t \\ \text{and } H^{(0)} = 0\big\}, \end{aligned} \tag{5.20}$$

where $(p^{(t)}, R^{(t)})$ are the parameters of $H^{(t)}$ per Definition 5.3. We use the shorthand $\{H^{(t)}\}$ to refer to such sets of transformation.

Recall that the high-fidelity model is assumed to produce continuous trajectories per §3.2. So, Definition 5.4 allows us to generically express any continuous trajectory of the robot's body in the plane.

The reason for these representations is that, to understand how to discretize obstacles, we must be able to express arbitrary (but continuous) robot motion with respect to obstacles. Note this approach also means that the results in this chapter are not RTD-specific. That is, while we use the receding-horizon time intervals and context of RTD to produce the discretized obstacle, this method can be applied to collision-avoidance for any planning or controls approach that considers a robot in $\mathsf{SE}(2)$. The utility of RTD is that, we can use this obstacle representation to *certify safe planning*, because RTD certifies safe motion planning independent of this particular obstacle representation.

5.2.3 Obstacle Assumptions

Recall that, to develop the discretized obstacle representation, we begin by assuming that all obstacles are static in §5.1.1.

We require the following obstacle geometry.

Assumption 5.5. *Let $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ be the set of static obstacles in the ORS, $\mathcal{R}_{\text{obs}}$. We assume that each $O^{(n)} \subset W$ is a closed, compact **polygon** with a finite number of vertices and edges.*

That is, ∂O can be written as a finite collection of line segments (as defined above). Importantly, we do *not* assume that each polygon obstacle is convex.

Note, Assumption 5.5 holds for common obstacle representations such as occupancy grids. Also, the assumption that each $O^{(n)}$ is closed and compact is fulfilled by the assumption that the robot has a finite sensor horizon (see §3.4.3). That is, even for an infinitely large obstacle, we need only consider the portion of it that intersects the robot's sensor horizon in each planning iteration (and recall that §3 provides a minimum size of this sensor horizon to ensure safety).

5.3 Five Geometric Quantities

We now introduce five geometric quantities, b, b_{max}, r, a, and r_{max}, which enable construction of the discretized obstacle. At the end of this section, we provide examples of each of these quantities for robots with rectangular and circular bodies. In §5.4, we show that these quantities exist and can be found for arbitrary convex, compact robot bodies.

5.3.1 Buffer and Point Spacing Motivation

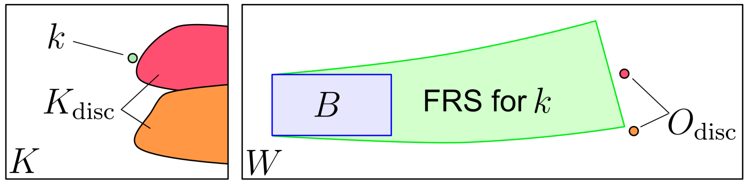
Before introducing these quantities, consider the following candidate method for constructing a discretized obstacle. Since our obstacles are closed, compact polygons by assumption, suppose that construct O_{disc} by we sampling a finite number of points from the boundary of each obstacle polygon. The rationale here is that, since the high-fidelity model of the robot produces continuous trajectories, if the robot starts outside every obstacle, then it cannot enter any obstacle without passing through an obstacle boundary. However, this strategy may be insufficient to prevent collisions. Suppose our robot has a rectangular body. Then, for any pair of points sampled from the boundary of an obstacle, a corner of the robot's body could still pass between these two points and cause a collision. This is shown in Figure 5.1b.

To resolve this issue, we must *buffer* the obstacle by some amount (the quantity b prescribed in this chapter), to prevent such collisions, as shown in Figure 5.1c. Furthermore, the *point spacing*, or distance between adjacent points in the discretized obstacle, must be sufficiently small such that, even if our robot passes between a pair of points, it cannot collide with the obstacle unless it collides with one or both points. If such a property holds, then ensuring collision avoidance with each point is equivalent to ensuring collision avoidance with the entire obstacle.

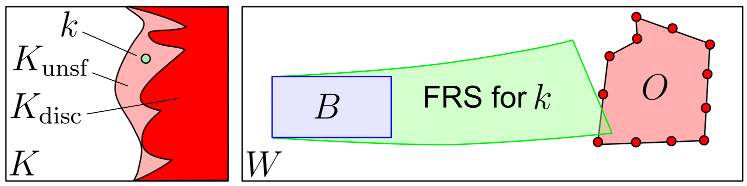
The purpose of this chapter is to rigorously define the buffer and point spacing to enable constructing O_{disc}.

5.3.2 The Buffer and Its Bound

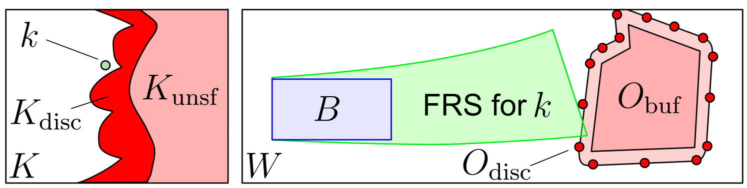
First, we define the buffer:



(a)



(b)



(c)

Figure 5.1: Motivation and method for buffering and discretizing obstacles. In each subfigure, the trajectory parameter space K is on the left, and the robot's workspace is on the right. The robot has a rectangular body B in blue. In the first subfigure, the obstacle consists of two points, labeled O_{disc}; the corresponding unsafe trajectory parameters K_{disc} are shown in K on the left. A safe k is chosen, and the corresponding subset of the FRS is shown on the right. In the second subfigure, the obstacle is a closed, compact polygon O, with corresponding pink unsafe plans K_{unsf} shown on the left. A discretized obstacle is constructed by sampling ∂O, and the corresponding unsafe parameters are shown as K_{disc} on the left; we see that there exist parameters that are safe with respect to this discretized obstacle, but unsafe for the actual obstacle O. In the third subfigure, we remedy this issue by buffering the obstacle to produce O_{buf}, then constructing the discretized obstacle from the buffered obstacle boundary. The unsafe plans for the discretized (buffered) obstacle are a provably superset of the unsafe plans for the (unbuffered) obstacle.

Definition 5.6. *Let $b > 0$ be a distance, called a* ***buffer****. Let $O_{\text{buf}} \subset W$ be a* ***buffered obstacle****, $O_{\text{buf}} \supset \bigcup_{n=1}^{n_{\text{obs}}} O^{(n)}$. In particular, this is a set such that, in any connected component of O_{buf}, the maximum Euclidean distance between O_{buf} and any $O^{(n)}$ is b:*

$$O_{\text{buf}} = \left\{ q \in W \mid \exists\, n \in \{1, \cdots, n_{\text{obs}}\} \text{ and } p \in O^{(n)} \text{ s.t. } \|p - q\|_2 \leq b \right\}. \tag{5.21}$$

For notation's sake, we define a function, buffer, *for which*

$$O_{\text{buf}} = \mathsf{buffer}(\{O^{(n)}\}_{n=1}^{n_{\text{obs}}},\ b). \tag{5.22}$$

Since buffering obstacles reduces the free space available for motion planning, we wish to upper bound the buffer to ensure that our obstacle representation is not unnecessarily conservative. To that end, we introduce the **buffer bound**, $b_{\text{max}} > 0$. To construct the discretized obstacle, we choose $b \in (0, b_{\text{max}})$.

5.3.3 The Point Spacing, Arc Point Spacing, and Their Bound

With the buffer and its bound established, we turn to the point spacing. To do so, we first inspect the geometry of the buffered obstacle:

Lemma 5.7. *Let the buffered obstacle O_{buf} be as in Definition 5.6. Then the boundary ∂O_{buf} consists of a finite set of line segments, $\mathcal{I}$, and a finite set of arcs, $\mathcal{A}$. That is, suppose $n_{\mathcal{I}} \in \mathbb{N}$ (resp. $n_{\mathcal{A}}$ is the number of line segments (resp. arcs). Then we can write*

$$\partial O_{\text{buf}} = \left(\bigcup_{I^{(i)} \in \mathcal{I}} I^{(i)} \right) \cup \left(\bigcup_{A^{(i)} \in \mathcal{A}} A^{(i)} \right). \tag{5.23}$$

Proof. This claim follows from [FHW12, Section 9.2], which we paraphrase here. In essence, since each $O^{(n)}$ is a polygon, the set O_{buf} is the Minkowski sum of a polygon with a closed disk of radius b. Recall that each obstacle $O^{(n)}$ is closed and bounded by Assumption 5.5. The procedure of constructing O_{buf} is also called "offsetting" a polygon by the distance b. Since each $O^{(n)}$ is closed and bounded, O_{buf} is a closed and bounded shape with a boundary consisting of line segments (corresponding to the edges of each $O^{(n)}$) and arcs (corresponding to the vertices of each $O^{(n)}$). $\square$

Now we can define the point spacing and arc point spacing:

Definition 5.8. *Consider a discretized obstacle O_{disc} that is generated by selecting a finite set of points from ∂O_{buf} such that the points are spaced by a distance $r > 0$ along the line segments, and a distance $a > 0$ along the arcs. We call r the* ***point spacing*** *and a the* ***arc point spacing****.*

We prove in §5.4 that, by choosing r and a as functions of the buffer b, the robot cannot pass far enough between any pair of points in O_{disc} to cause a collision.

Similar to the upper bound b_{max} on the buffer, we find a **point spacing bound** r_{max} for r and a (note, r is also an upper bound of a per Lemma 5.22 later in this chapter). Recall that b_{max} limits the buffer size to prevent obstacles from reducing the free space available for planning. On the other hand, $r_{\max}$ ensures that the discretized obstacle points are close enough to each other so that the robot cannot pass between them.

Now we have defined the geometric quantities b (buffer), $b_{\max}$ (buffer bound), r (point spacing), a (arc point spacing), and $r_{\max}$ (point spacing bound). In §5.4, we explain how to find each one.

5.3.4 Examples

Before proving that each of these quantities exist, and can be computed, we provide a pair of examples for two common robot body shapes: a rectangle, and a circle. The quantities are found analytically for these shapes, and visual proof is provided in Figure 5.2.

Example 5.9. *Suppose the robot body B is a rectangle with width $w > 0$ and length $l > w$. Then the bounds are $r_{\max} = w$ and $b_{\max} = \frac{w}{2}$. Pick $b \in (0, b_{\max})$. Then we have*

$$r = 2b \quad \text{and} \quad a = 2b\sin\left(\frac{\pi}{4}\right). \tag{5.24}$$

A visual proof, with $b_{\max}$ omitted for clarity, is shown in Figure 5.2a.

Example 5.10. *Suppose the robot body B is a circle with radius $\rho > 0$. Then the bounds are $r_{\max} = 2\rho$ and $b_{\max} = \rho$. Pick $b \in (0, b_{\max})$, and construct the (positive) angles*

$$\theta_1 = \cos^{-1}\left(\frac{\rho - b}{\rho}\right) \quad \text{and} \quad \theta_2 = \cos^{-1}\left(\frac{b}{2\rho}\right). \tag{5.25}$$

Then we have

$$r = 2\rho\sin\theta_1 \quad \text{and} \quad a = 2b\sin\theta_2. \tag{5.26}$$

A visual proof, with $b_{\max}$ omitted for clarity, is shown in Figure 5.2b.

5.4 Finding the Geometric Quantities

We now describe how to find the geometric quantities described in §5.3. The arguments in this section describe a procedure to compute the quantities for an *arbitrary* convex, compact robot body. Note that the examples in §5.3.4 are sufficient to enable most readers to use this proposed method, so the more casual reader can skip this section.

This section proceeds in four steps. First, we find the upper bound $r_{\max}$ on the point spacing. Second, we use $r_{\max}$ to find the buffer bound $b_{\max}$. Third, we find the point spacing r for a choice

(a) (b)

Figure 5.2: Examples (and visual proof) of the geometric quantities $r_{\max}$, r, b, and a, used to construct the discretized obstacle, for rectangular and circular robot bodies.

of buffer $b \in (0, b_{\max})$. Fourth and finally, we find the arc point spacing a, again for a choice of buffer $b \in (0, b_{\max})$.

5.4.1 The Point Spacing Bound

We now seek to understand how close together points must be in the discretized obstacle. We do this by upper bounding the point spacing with the quantity $r_{\max}$. We find $r_{\max}$ first, because finding the remaining quantities depends on it.

This discussion builds on Theorem 1 of [Str82]. To build intuition, imagine a wall in the workspace W, with a gap that is large enough for the robot to pass through. If we keep shrinking this gap, eventually the robot is unable to pass through without collision. In this subsection, informally, we find the largest gap that the robot cannot pass all the way through. We use the size of the gap as the upper bound $r_{\max}$ on the spacings r and a when constructing O_{disc}. Imagine that the buffered obstacle's boundary is treated as the wall. If the wall is sampled so that points are closer than $r_{\max}$ apart, this is akin to a gap of width at most $r_{\max}$ between each pair of points.

To proceed, we first formally define the notion of passing the robot's body through a line segment. Then, we find the size of the "largest gap" discussed above.

To define "passing through" a gap, represented by a line segment I, we first establish a halfplane P_I that is "defined" by I; we use P_I as a region that the robot begins in, so that, to pass through I, the robot must leave the halfplane P_I. To create this halfplane, consider the function $\delta_\pm : \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ for which

$$\begin{aligned}\delta_\pm(e_1, e_2, p) = \frac{1}{\|e_2 - e_1\|_2}\Big((e_{2y} - e_{1y})p_x - \\ - (e_{2x} - e_{1x})p_y - e_{2y}e_{1y} - e_{2y}e_{1x}\Big),\end{aligned} \tag{5.27}$$

where the subscript x or y denotes the corresponding coordinate of a point in $\mathbb{R}^2$. If I has distinct endpoints $\{e_1, e_2\}$, then $\delta_\pm(e_1, e_2, p)$ is the perpendicular distance from the point p to the line defined by I. The sign of $\delta_\pm(e_1, e_2, p)$ is positive if p lies to the "left" of the line defined by I, relative to the "forward" direction from e_1 to e_2. The function $\delta_\pm$ is illustrated in Figure 5.4a. We use $\delta_\pm$ to define a halfplane in $\mathbb{R}^2$ as follows:

Definition 5.11 (Halfplane Defined by I)**.** *Let $c_0 \in B$ denote the center of the robot's body at time 0. Let I be a line segment with two distinct endpoints $E_I = \{e_1, e_2\}$. Then $P_I \subset \mathbb{R}^2$ denotes the closed* ***halfplane defined by*** *I; this halfplane is determined by the line defined by I and by c_0 as:*

$$P_I = \{p \in X \mid \text{sign}(\delta_\pm(e_1, e_2, p)) = \text{sign}(\delta_\pm(e_1, e_2, c_0))\}\,, \tag{5.28}$$

where $\text{sign}(a) = 1$ *for $a \geq 0$ and -1 otherwise. Now suppose that I is a line segment of length 0, i.e. $e_1 = e_2$, so we cannot directly define P_I as in* (5.28)*. Suppose that $e_1 \neq c_0$. We can pick a point e' for which $(e' - e_1) \cdot (c_0 - e_1) = 0$ where $\cdot$ denotes the standard inner product on $\mathbb{R}^2$. This means that the line segment from e_1 to c_0 is perpendicular to the line segment from e_1 to e'. Then, P_I is given by* (5.28)*, but using e' in place of e_2.*

In the case where $e_1 = e_2 = c_0$, P_I is undefined. See Figures 5.3 and 5.4a for illustrations of the different cases of P_I. Notice that, except when $e_1 = e_2 = c_0$, P_1 is always a closed halfplane. The utility of P_I is that, if the line defined by I does not intersect B, then $B \subset P_I$, i.e. P_I contains B. So, we can use P_I as a region that the robot starts in at time 0.

Definition 5.12 (Passing Through)**.** *Let $I \subset (\mathbb{R}^2 \setminus B)$ be a line segment with endpoints E_I, and P_I be the halfplane defined by I. Suppose that the robot lies fully within P_I at time 0, i.e. $B \subset$* $\text{interior}(P_I)$*. Let $\{H^{(t)}\}$ be a transformation family. Let t_0, t_1 be indices in $(0, t_\text{f}]$ such that $H^{(t)}B$ intersects the "middle" of I, i.e. $H^{(t)}B \cap (I \setminus E_I) \neq \emptyset$, for all $t \in [t_0, t_1]$. Furthermore, suppose that $H^{(t)}B \subset P_I$ for all $t \in [0, t_0)$, and that no $H^{(t)}B$ can intersect the endpoints E_I (i.e. $H^{(t)}B \cap E_I = \emptyset$) except at $t = t_\text{f}$. We say that such a transformation family attempts to* ***pass*** *B* ***through*** *I. If B is able to leave P_I while passing through I, i.e. $H^{(t)}B \subset P_I^\text{C}$, then B is said to* ***pass fully through*** *I.*

See Figure 5.3 for an illustration of passing through and passing fully through. The motion of the robot at each t is represented by each set $H^{(t)}B$.

Notice that, if B must pass through I, it is not allowed to go "around" I when passing through. Furthermore, over the time horizon $[t_0, t_1]$ in Definition 5.12, the set made by the intersection $H^{(t)}B \cap I$ is a chord of $H^{(t)}B$ [Str82, Theorem 1]. We now state a property of B used to bound the size of the aforementioned "gap in a wall" in Lemma 5.14 below.

Definition 5.13 (Thickness, Width, and Diameter)**.** *Given a unit vector $\hat{u}$ in $\mathbb{R}^2$ at an angle θ relative to the x-axis, the* ***thickness*** *of B along this unit vector is the distance between the two*

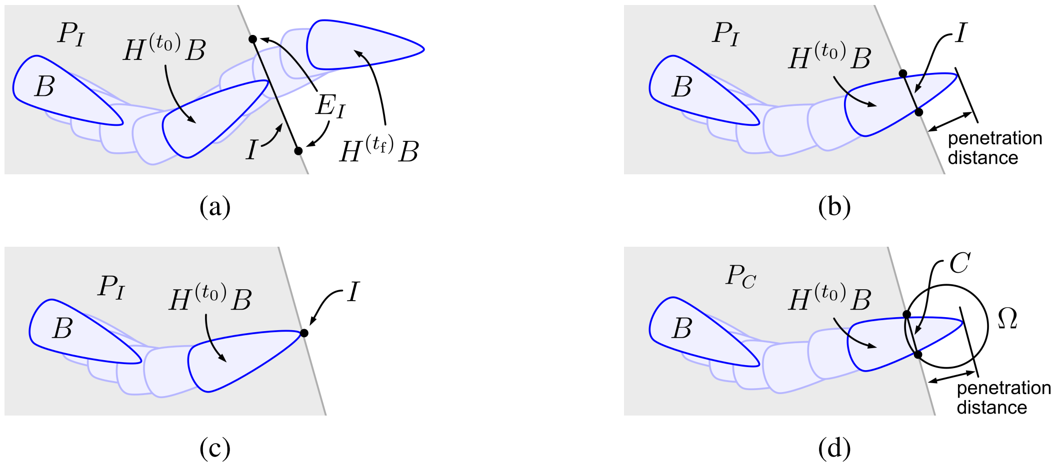


Figure 5.3: Passing through (as in Definition 5.12), penetrating (as in Definition 5.16), and penetrating into a circle (as in Definition 5.21). In each subfigure, a family $\{H^{(t)}\}$ of continuous rotations and translations attempts to pass the convex, compact set B through the line segment I with endpoints E_I. At $t = 0$, B lies in the halfplane P_I, defined by I. Each figure contains B at its initial position $H^{(0)}B$ and final position $H^{(t_f)}B$ indicated by a dark outline. The lighter outlines between these positions show examples of B being translated and rotated as each $H^{(t)}$ is applied. In Figure 5.3a, B is able to pass fully through I; the index $t_0 \in T_{\text{plan}}$ where B first touches I is also shown with a dark outline. In Figure 5.3b, B is unable to pass fully through I, but penetrates through I by some distance into P_I^{C}. In Figure 5.3c, the line segment I has length 0, so B cannot pass through it, but instead stops as soon as it touches I, and achieves 0 penetration distance through I. Note that, in this case, P_I is defined by a line perpendicular to the line segment from I to the center of mass of B, as per Definition 5.11. In Figure 5.3d, the circle Ω has a chord C, and B penetrates into Ω through C by the penetration distance shown. The halfplane defined by C is denoted P_C.

*unique lines that are tangent to B and perpendicular to the vector. The **width** of B is defined as the minimum thickness of B when searching over all $\theta \in [0, 2\pi)$, and the **diameter** of B is, similarly, the maximum thickness [Str82, Section 1].*

See Figure 5.4b for an illustration of thickness. Note that the width is nonzero and finite because B is compact and has nonzero volume..

Lemma 5.14 (Point Spacing Bound)**.** *[Str82, Theorem 1] Let $I \subset (\mathbb{R}^2 \setminus B)$ be a line segment with endpoints E_I and length $l > 0$. Let B be the robot's body at time 0, with width $w > 0$. Then B can pass through I if and only if $w < l$.*

Proof. While the proof is available in [Str82], we prove this lemma again here to build intuition. Recall that B is convex and compact with nonzero volume.

Suppose a transformation family $\{H^{(t)}\}$ passes B through I. Then there exists an interval of time $[t_0, t_1] \subset (0, t_f]$ for which $H^{(t)}B \cap (I \setminus E_I)$ is nonempty for all $t \in [t_0, t_1]$; note that $t_1 > t_0$

Figure 5.4: An arbitrary, compact, convex set B lies in the plane. In Figure 5.4a, the line segment I defines the closed halfplane P_I (the filled grey area) using the function $\delta_\pm$ from (5.27). If the endpoints of I are labeled e_1 and e_2, then the set P_I contains all points $p \in \mathbb{R}^2$ for which the sign of $\delta_\pm(e_1, e_2, p)$ is the same as the sign of $\delta_\pm(e_1, e_2, c_0)$, where c_0 is the center of B. In Figure 5.4b, a unit vector $\hat{u}$ is fixed to the origin with angle θ. The thickness of B is given by the distance between the two unique lines that are tangent to B and perpendicular to $\hat{u}$.

because B has nonzero volume. The set $H^{(t)}B \cap (I \setminus E_I)$ is a chord of $H^{(t)}B$ with length greater than or equal to the width w. Since B can pass fully through I, the endpoints E_I never intersect any $H^{(t)}B$. Therefore the length of the chord $H^{(t)}B \cap I$ is always less than l, so $l > w$.

Now suppose $w < l$. If B has diameter d, then B can fit completely inside a rectangle with short side length w and long side length d [FS75, Theorem 3] This rectangle can be rotated so that its short side is parallel to I, then pass fully through I by pure translation, i.e. with no further rotations. Since B fits inside the rectangle, B can pass fully through I. □

From this lemma, the robot's width defines the smallest gap that the robot can pass through. Therefore, we define r_{max} as the robot's width:

Definition 5.15. *The quantity r_{max} denotes the **point spacing bound**, which is equal to the width of the robot body B.*

The maximum point spacing relates to the points in the discretized obstacle O_{disc} as follows. As illustrated in Figure 5.1c, the discretized obstacle O_{disc} is constructed by first buffering an obstacle O by the distance b, then sampling the boundary of the buffered obstacle O_{buf} such that the distance between consecutive sampled points is strictly less than r_{max}. Note, we refer to such consecutive sampled points as **adjacent points** of the discretized obstacle O_{disc}.

Finding r_{max} correctly is critical. Suppose that we attempt to pass B through the gap between two adjacent points of O_{disc}, and do not allow B to overlap with either of the points while passing through. Since each pair of adjacent points of O_{disc} are strictly closer than r_{max} to each other, we know by Lemma 5.14 that the robot can never pass fully through the gap. In other words, the quantity r_{max} must either be found exactly or underapproximated to ensure safety. Methods exist to exactly compute the width of arbitrary compact convex sets. For example, the algorithm by [FS75] finds the smallest bounding rectangle of the set; then the length of the rectangle's shorter leg is the set's width. A geometric procedure to find the width is presented in [Str82, Section 1]

Next, we use $r_{\max}$ to bound the buffer with the quantity $b_{\max}$.

5.4.2 The Buffer Bound

As in §5.4.1, imagine a wall with gap of width $r_{\max}$. Lemma 5.14 proves that the robot cannot pass fully through this gap. However, the robot can still penetrate through the gap by some distance before it gets stopped by the wall. In this section, we find the farthest distance that the robot can penetrate through the gap. We use this maximum penetration distance as an upper bound $b_{\max}$ on the obstacle buffer, so $b \in (0, b_{\max})$.

Recall that our intention is to sample the boundary of the buffered obstacle to produce a set O_{disc}. So, the spacing between adjacent points of O_{disc} must be smaller than $r_{\max}$. If the robot is not allowed to touch any points in O_{disc}, it cannot penetrate farther than the distance $b_{\max}$ between any pair of adjacent points. So, obstacles do not need to be buffered by a distance larger than $b_{\max}$. We prove the existence of $b_{\max}$ below in Lemma 5.17. To proceed, we first define the word "penetrate" precisely.

Definition 5.16 (Penetration Distance)**.** *Let $I \subset (X \setminus B)$ be a line segment. Let P_I be the half-plane defined by I, and suppose $B \subset P_I$ strictly. Let $\{H^{(t)}\}$ be a transformation family that attempts to pass B through I. Suppose B cannot pass fully through I, and that $H^{(t_f)}B \cap P_I^{\mathsf{C}}$ is nonempty, so there is some portion of B that does pass through I. Consider all line segments perpendicular to I with one endpoint on I and the other at a point in $H^{(t_f)}B$ in P_I^{C}. We call the maximum length of any of these line segments the* ***penetration distance of*** B ***through*** I*. The set $H^{(t_f)}B$* ***penetrates*** *I by this distance, as in Figure 5.3b. If I is of length 0, then the penetration distance of B through I is always 0, as in Figure 5.3c.*

Lemma 5.17 (Buffer Bound)**.** *Let B be the robot's body at time 0, with width $r_{\max}$. Let $I_{r_{\max}} \subset (W \setminus B)$ be a line segment of length $r_{\max}$. Then there exists a maximum penetration distance $b_{\max}$ that can be achieved by passing B through $I_{r_{\max}}$.*

Proof. This proof is illustrated in Figure 5.5. We sketch the intuition first. To find $b_{\max}$, we use transformation families $\{H^{(t)}\}$ to pass B through $I_{r_{\max}}$. Recall that B cannot pass fully through $I_{r_{\max}}$ by Lemma 5.17. Then, we measure the penetration distance corresponding to each transformation family to find a supremum.

Now we restate this concept more rigorously. Note that B is compact and convex with nonzero volume. To ease the exposition, assume without loss of generality that B lies entirely in the left

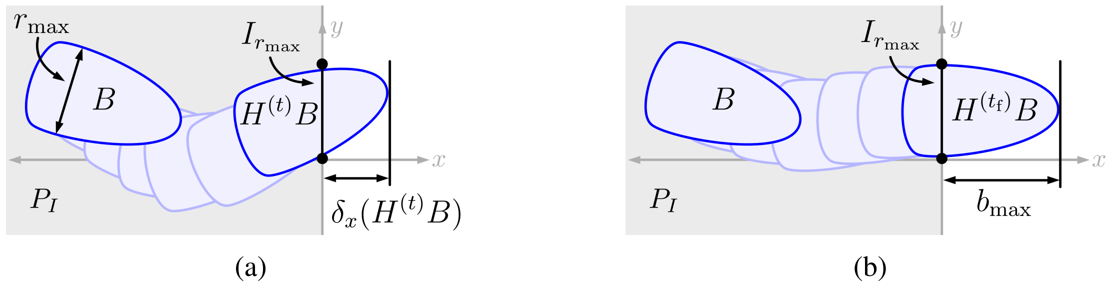


Figure 5.5: An arbitrary compact, convex set B of width $r_{\max}$ penetrates a line segment $I_{r_{\max}}$ by the distance $b_{\max}$ when a transformation family $\{H^{(t)}\}$ is applied to pass B through $I_{r_{\max}}$. Since $I_{r_{\max}}$ is of length $r_{\max}$, B cannot pass fully through by Lemma 5.14. At the initial index $t = 0$ and the final index $t = t_{\mathrm{f}}$, the sets $H^{(0)}B$ and $H^{(t_{\mathrm{f}})}B$ are shown with dark outlines. A sampling of intermediate indices $t \in (0, t_{\mathrm{f}})$ are shown with light outlines. The first subfigure shows a suboptimal solution; the second subfigure shows the optimal solution to identify the buffer bound $b_{\max}$.

half-plane of $\mathbb{R}^2$, and that $I_{r_{\max}}$ is fixed to the origin and oriented vertically in the upper half-plane, so $I_{r_{\max}} = \{0\} \times [0, r_{\max}]$; this is a reasonable assumption because all of the operations in any transformation family are invariant to the initial rotation/translation of B. In this case, the half-plane $P_{r_{\max}}$ defined by $I_{r_{\max}}$ is the closed left half-plane. This too can be done without loss of generality because, when passing B through $I_{r_{\max}}$ with a transformation family $\{H^{(t)}\}$, we only care about the position of B *relative* to $I_{r_{\max}}$ at each $t \in [0, t_{\mathrm{f}}]$.

Let $\mathcal{H}_{r_{\max}}$ denote the set of all transformation families $\{H^{(t)}\}$ that attempt to pass B through $I_{r_{\max}}$ as per Definition 5.12. By Lemma 5.14, B cannot pass fully through $I_{r_{\max}}$ because $I_{r_{\max}}$ is of length $r_{\max}$; but B may penetrate $I_{r_{\max}}$ by some nonzero distance, which depends upon the transformation family $\{H^{(t)}\}$. We must show that, across all $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$, there is a maximum penetration distance.

Consider an arbitrary $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$. Since $I_{r_{\max}}$ is collinear with the y-axis, we can find the penetration distance of B through $I_{r_{\max}}$ corresponding to $\{H^{(t)}\}$ using a function $\delta_x : \mathrm{pow}\,(\mathbb{R}^2) \to \mathbb{R}$, which returns the right-most point of a set $A \subset \mathbb{R}^2$:

$$\delta_x(A) \;=\; \sup_a \{\, a_x \;\mid\; a \in A \,\}\,, \tag{5.29}$$

where a_x is the x-component of the point a. So, given a particular $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$, $\delta_x(H^{(t_{\mathrm{f}})}B)$ is the penetration distance of B through $I_{r_{\max}}$ by Definition 5.16. Recall that B is compact (i.e. closed and bounded in X) and that B cannot pass fully through $I_{r_{\max}}$ by Lemma 5.14 (i.e. the horizontal displacement achieved by $H^{(t_{\mathrm{f}})}B$ is bounded). Therefore, $\delta_x(H^{(t_{\mathrm{f}})}B)$ is upper bounded.

We have shown that the penetration distance is bounded for each $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$. To prove the claim that there is a maximum penetration distance, we must show that the value of δ_x is upper bounded across all $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$. In other words, we want to know that the following supremum is finite:

$$b_{\max} = \sup_{\{H^{(t)}\}} \delta_x(H^{(t_{\mathrm{f}})}B) \tag{5.30}$$

$$\text{s.t.} \quad \{H^{(t)}\} \in \mathcal{H}_{r_{\max}}. \tag{5.31}$$

Recall from Definition 5.13 that B has a finite diameter (suppose we denote it d), which is the largest possible distance between two parallel lines that are tangent to B. So, for any $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$, if $\delta_x(H^{(t_{\mathrm{f}})}B) > d$, then B has passed fully through I. But this is impossible by Lemma 5.14. Since $\{H^{(t)}\}$ was arbitrary, (5.30) is upper bounded. □

To relate Lemma 5.17 to the robot, consider the following. If we buffered an obstacle O by the amount $b_{\max}$, and spaced points along the boundary of O by a distance less than $r_{\max}$, then the *farthest* that the robot could pass between any pair of adjacent points *without touching either point* is strictly less than $b_{\max}$. Therefore, the robot could not collide with the obstacle without touching one of the points. In other words, if the robot avoids every such point, then the robot avoids the obstacle. Since we choose the buffer $b \in (0, b_{\max})$, it is critical to underapproximate $b_{\max}$.

Next, we find the point spacing r using $b \in (0, b_{\max})$.

5.4.3 The Point Spacing

Let $r_{\max}$ be as in Definition 5.15 and $b_{\max}$ as in Lemma 5.17. We choose $b \in (0, b_{\max})$, then use b to find the point spacing r. We prove that r exists below, in Lemma 5.20. First, we need two intermediate results about chords of compact sets.

Lemma 5.18. *Given any three distinct, parallel chords of a convex, compact set in $\mathbb{R}^2$, the middle chord is not the shortest of the three.*

We now restate this more formally. Let $A \subset \mathbb{R}^2$ be a convex, compact set with nonzero volume. Let C_1, C_2, and C_3 be three chords of A such that $C_1 \parallel C_2 \parallel C_3$ and $C_i \cap C_j = \emptyset$ for any $i \neq j$. Suppose the chords have lengths l_1, l_2, and l_3, respectively. Furthermore, assume that there exists at least one line segment within A that intersects C_2, and that has one endpoint on C_1 and the other endpoint on C_3; in other words, C_2 lies between C_1 and C_3. Then$l_1 \geq l_3$ implies that $l_2 \geq l_3$, and $l_1 > l_3$ implies that $l_2 > l_3$.

Proof. Let $e_{i,1}$ and $e_{i,2}$ denote the endpoints of each chord C_i where $i = 1, 2, 3$. By definition, these endpoints lie in ∂A. Without loss of generality, assume that all three chords are oriented vertically (rotating the chords and the shape A does not change the relative position of the chords to each other or to A). Also suppose without loss of generality that each $e_{i,1}$ is the "upper" endpoint;

we can do this without loss of generality because each chord is a line segment by definition, and because we can swap the labels of the endpoints of a line segment without changing the set of points in the line segment. Define the line segments I_1 from $e_{1,1}$ to $e_{3,1}$ and I_2 from $e_{1,2}$ to $e_{3,2}$. Since A is convex, $I_1, I_2 \subset A$.

Suppose C_1 and C_3 have the same length, so $l_1 = l_3$. Then the quadrilateral with edges given by the line segments C_1, I_1, C_3, and I_3 is a parallelogram Q_{para} (two of its sides are parallel and of equal length). So, every line segment inside Q_{para} that is parallel to C_1 has length $l_1 = l_3$. Furthermore, Q_{para} lies completely inside A because A is convex; this means that $C_2 \cap Q_{\text{para}}$ is a chord of Q_{para} that is parallel to C_1, and $C_2 \cap Q_{\text{para}} \subseteq C_2$. Then, since the length of $C_2 \cap Q_{\text{para}} = l_1$, the length of C_2 is $l_2 \geq l_1 \geq l_3$.

Now suppose $l_1 > l_3$. Then the quadrilateral with edges C_1, I_1, C_3, and I_3 is a trapezoid Q_{trap} (two of its sides are parallel and of different lengths) that lies within A. Since $l_1 > l_3$, every line segment inside Q_{trap} that is parallel to C_1 is strictly shorter than C_1. So, similar to the logic for Q_{para} above, the length of $\kappa_2 \cap Q_{\text{trap}}$ is greater than l_3, meaning that $l_2 > l_3$. □

Next, we use Lemma 5.18 to understand the shape of the body as it passes through a line segment in Lemma 5.19. In particular, Lemma 5.19 shows that, as the robot penetrates farther through a line segment, the size of the intersection between the robot and the line segment increases. We use this result in Lemma 5.20 to bound r above and below.

Lemma 5.19. *Let B be the robot's body at time 0, with width $r_{\max}$. Let $I_{r_{\max}} \subset (\mathbb{R}^2 \setminus B)$ be a line segment of length $r_{\max}$. Let $P_{r_{\max}}$ be the closed half-plane defined by $I_{r_{\max}}$ and containing B, and suppose that $B \subset P_{r_{\max}}$. Suppose the transformation family $\{H^{(t)}\}$ attempts to pass B through $I_{r_{\max}}$. Suppose $t_0 > 0$ such that, for each $t \in [t_0, t_{\mathrm{f}}]$, the set $C_t := H^{(t)}B \cap I_{r_{\max}}$ is nonempty and is a chord of $H^{(t)}B$. Then, for any $t > t_0$, every chord of $H^{(t)}B$ that is parallel to $I_{r_{\max}}$ and lies in $P^C_{r_{\max}}$ is shorter than C_t.*

Proof. This claim follows directly from Definition 5.12 of passing through and from Lemma 5.18.

To see why, first recall that B is convex and compact with nonzero volume. As in Lemma 5.17, without loss of generality assume $I_{r_{\max}}$ lies along the y-axis with its lower endpoint fixed to the origin, i.e. $I_{r_{\max}} = \{0\} \times [0, r_{\max}]$, and that B lies in the closed left half-plane, which is $P_{r_{\max}}$. Let $t \in (t_0, t_{\mathrm{f}}]$ be arbitrary and let C_t denote the chord $H^{(t)}B \cap I_{r_{\max}}$. Note that t_0 exists by Definition 5.12. In addition, for any $t \in (t_0, t_{\mathrm{f}}]$, the set $H^{(t)}B \cap I_{r_{\max}}$ is a chord of $H^{(t)}B$ [Str82, Theorem 1]. Notice that the length of C_t is less than or equal to $r_{\max}$ by the definition of passing through. By Lemma 5.14, B cannot pass fully through $I_{r_{\max}}$. Therefore, there exists a chord C_1 of $H^{(t)}B$ that lies in $P_{r_{\max}}$, is parallel to $I_{r_{\max}}$, and has length greater than or equal to $r_{\max}$. Otherwise, $H^{(t)}B$ could pass fully through $I_{r_{\max}}$ by translation. Since $t > t_0$, $H^{(t)}B \cap P^{\mathrm{C}}_{r_{\max}}$ is nonempty by Definition 5.12 of passing through. Therefore, there exist chords of $H^{(t)}B$ that lie in $P^{\mathrm{C}}_{r_{\max}}$ and are parallel to

$I_{r_{\max}}$. Let C_2 be any such chord. The chords C_1, C_t, and C_2 are three parallel, distinct chords of the convex, compact set $H^{(t)}B$, and the length of C_1 is greater than the length of C_t. Therefore, by Lemma 5.18, C_2 is shorter than C_t. Since C_2 was arbitrary, we are done. □

Now we are ready to prove the existence of r. The proof also provides a method to construct r, which is illustrated in Figure 5.6.

Lemma 5.20. *Let $B \subset \mathbb{R}^2$ be the robot's body at time 0, with width $r_{\max}$. Let $b_{\max}$ be the buffer bound corresponding to B (as in Lemma 5.17). Pick $b \in (0, b_{\max})$. Then there exists $r \in (0, r_{\max}]$ such that, if I_r is a line segment of length r, and if $\{H^{(t)}\}$ is any transformation family that attempts to pass B through I_r, then the penetration distance of B through I_r is less than or equal to b.*

Proof. We first sketch the intuition for the proof. As in Lemma 5.17, we attempt to pass B through a line segment $I_{r_{\max}}$ of length $r_{\max}$, but B cannot pass fully through $I_{r_{\max}}$ by Lemma 5.14. Each time we pass B through $I_{r_{\max}}$, we stop passing it through when the penetration distance of B through $I_{r_{\max}}$ is equal to b. Then, we measure the length of the line segment $H^{(t_{\text{stop}})}B \cap I_{r_{\max}}$, where $H^{(t_{\text{stop}})}$ is the transformation at the time we stopped passing B through $I_{r_{\max}}$. The length of the smallest such line segment is the desired point spacing r.

We now proceed rigorously. Let $I_{r_{\max}} \subset (X \setminus B)$ be a line segment of length $r_{\max}$. Without loss of generality, suppose that $I_{r_{\max}}$ is vertical with its lower endpoint at the origin, so $I_{r_{\max}} = \{0\} \times [0, r_{\max}]$; and suppose that $B \subset X \subset \mathbb{R}^2$ lies entirely in the closed left half-plane. See the proof of Lemma 5.17 for why $I_{r_{\max}}$ and B can be placed this way without loss of generality; in brief, the rotations and translations required can be undone.

Next, we discuss how we measure horizontal distance (to constrain the penetration distance to b) and vertical span (to find the distance r). Unlike in Lemma 5.17, instead of letting B penetrate through $I_{r_{\max}}$ by the distance $b_{\max}$, we limit the penetration distance to $b < b_{\max}$. Since $I_{r_{\max}}$ is oriented vertically at the origin, we can measure the penetration distance through $I_{r_{\max}}$ using the horizontal distance given by δ_x from (5.29), which returns the maximum x-coordinate over all points in a set in $\mathbb{R}^2$. To measure vertical span, we define the map $\delta_y : \operatorname{pow}(\mathbb{R}^2) \to \mathbb{R}_{\geq 0}$ as follows:

$$\delta_y(A) \;=\; \sup\{a_y \mid a \in A\} \;-\; \inf\{a_y \mid a \in A\}, \tag{5.32}$$

where a_y denotes the y-component of a.

Now, we find r by constructing the line segment I_r. Let $\mathcal{H}_{r_{\max}}$ be the set of all transformation families $\{H^{(t)}\}$ that attempt to pass B through $I_{r_{\max}}$. Suppose that $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$ is a transformation family for which, at $t = T$, the penetration distance of B through $I_{r_{\max}}$ is b. In other words, $\delta_x(H^{(t_{\text{f}})}B) = b$. Consider the line segment $I_r = H^{(t_{\text{f}})}B \cap I_{r_{\max}}$ (this is a line segment by Theorem 1 of [Str82]). Then, under the transformation family $\{H^{(t)}\}$, B penetrates through I_r by the distance b, and the length of I_r is given by $\delta_y(H^{(t_\mathrm{f})}B \cap I_{r_\mathrm{max}})$. So, our goal is to find the shortest I_r over all such $\{H^{(t)}\}$; the length of the shortest I_r is the distance r claimed by the premises. Consider the following program to achieve this goal:

$$
\begin{aligned}
r \quad = \quad \inf_{\{H^{(t)}\}} \quad & \delta_y(H^{(t_\mathrm{f})}B \cap I_{r_\mathrm{max}}) && (5.33)\\
\text{s.t.} \quad & \{H^{(t)}\} \in \mathcal{H}_{r_\mathrm{max}}, && (5.34)\\
& \delta_x(H^{(t_\mathrm{f})}B) = b. && (5.35)
\end{aligned}
$$

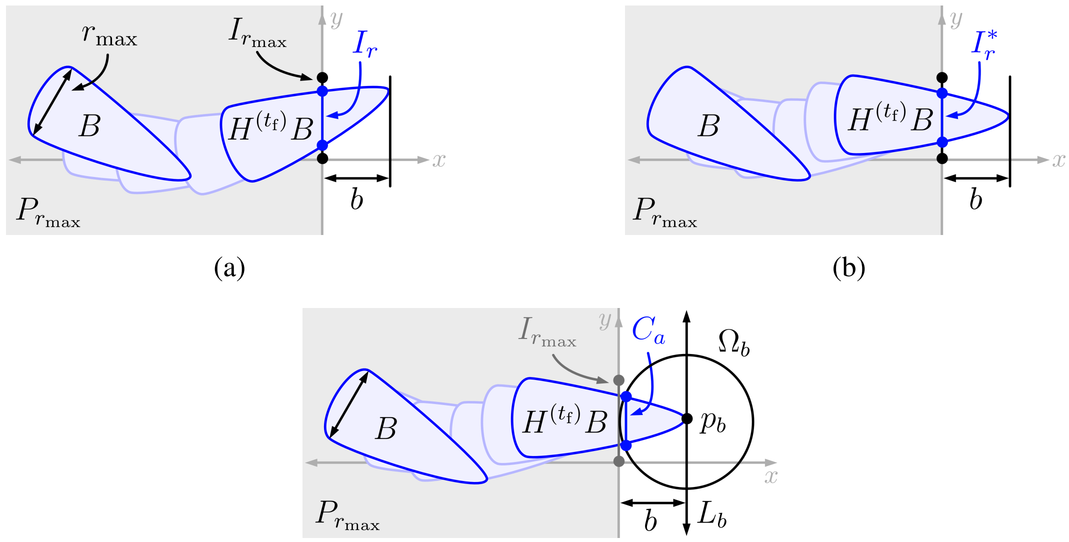
We first check that feasible solutions exist for (5.33). By Lemma 5.17, there exist $\{H^{(t)}\} \in \mathcal{H}_{r_\mathrm{max}}$ for which $\delta_x(H^{(t_\mathrm{f})}B) = b_\mathrm{max} > b$. For any such $\{H^{(t)}\}$, since $H^{(0)}B = B$ (which lies in the left half-plane), we have that $\delta_x(H^{(0)}B) \leq 0$. Then, since $\{H^{(t)}\}$ is continuous in t by Definition 5.4, there must exist some $t_b \in (0, t_\mathrm{f})$ for which $\delta_x(R_{t_b}B) = b$. So, again using that $\{H^{(t)}\}$ is continuous, we can "cut off" the time index t at t_b and then rescale time so that t_b becomes t_f as follows. For $t \in [0, t_b]$, let $t' = \frac{t_\mathrm{f}}{t_b}t$. Then the family $\{H^{(t')} \mid t' \in [0, t_\mathrm{f}]\}$ for which $H^{(t')} = H^{(t)}$ is a family in $\mathcal{H}_{r_\mathrm{max}}$ for which B penetrates through I_{r_max} by the distance b.

Now we check that $r \in (0, r_\mathrm{max}]$. Suppose that $\{H^{(t)}\}$ is a feasible solution to (5.33). Notice that $\{H^{(t)}\}$ cannot pass B fully through I_{r_max} by Lemma 5.14, so $\delta_y(H^{(t_\mathrm{f})}B \cap I_{r_\mathrm{max}}) \leq r_\mathrm{max}$ is immediate. By Definition 5.12 of passing through, $H^{(t_\mathrm{f})}B \cap I_{r_\mathrm{max}}$ must be nonempty, so $r = \delta_y(H^{(t_\mathrm{f})}B \cap I_{r_\mathrm{max}}) \geq 0$.

Finally, we show that (5.33) achieves a minimum $r > 0$. Let $\{H^{(t)}\}$ be a feasible solution. Suppose for the sake of contradiction that there is no $\varepsilon > 0$ for which $r \geq \varepsilon$. Let $C_r = H^{(t_\mathrm{f})}B \cap I_{r_\mathrm{max}}$, which is a chord of $H^{(t_\mathrm{f})}B$ [Str82, Theorem 1]. By Lemma 5.19, no chord parallel and to the right of C_r can be longer than C_r, because I_{r_max} is of length $r_\mathrm{max} \geq r$ and parallel to C_r. But then, if $\varepsilon = 0$, since B has nonzero volume, there can be no nonempty chords to the right of C_r, which contradicts the fact that $\{H^{(t)}\}$ attempts to pass B through I and as a result violates (5.35). □

A suboptimal, feasible solution to (5.33) is shown in Figure 5.6a; an optimal solution for the same B is shown in Figure 5.6b. With Lemma 5.20, and specifically (5.33), we find the **point spacing** r.

We use r as follows. Suppose our robot has a body B, with width r_max as in Definition 5.15, and associated maximum penetration distance b_max as in Lemma 5.17. Pick $b \in (0, b_\mathrm{max})$. Suppose $O \subset X$ is a (polygonal) obstacle. Construct O_buf, the buffered obstacle, with (5.21). Recall by Lemma 5.7 that the boundary of the buffered obstacle consists of line segments and arcs. Then, r lets us construct the portion of the discretized obstacle O_disc that corresponds to the line segments in ∂O_buf. In particular, suppose we sample each line segment of ∂O_buf such that adjacent points are



(a) (b) (c)

Figure 5.6: An illustration of Program (5.33) in Figures 5.6a and 5.6b, and Program (5.36) in Figure 5.6c. The set B is an arbitrary convex, compact shape, and starts at $t = 0$ in the left half-plane P_I. The transformation family $\{H^{(t)}\}$ attempts to pass B through $I_{r_{\max}}$. At time T, $H^{(t_f)}B$ is stopped such that its penetration distance through $I_{r_{\max}}$ is the distance b. Program (5.33) attempts to find the smallest line segment I_r that can be created when passing B through $I_{r_{\max}}$ up to the penetration distance b; a suboptimal, feasible solution is shown in Figure 5.6a, and an optimal solution is shown in Figure 5.6b. Program (5.36) attempts to find the smallest chord C_a of a circle Ω_b for which B cannot penetrate farther than b into Ω_b through C_a. This is shown in Figure 5.6c, which starts from a feasible solution to (5.33), then centers the circle Ω_b on a point of $H^{(t_f)}B$ that has penetrated to the distance b past $I_{r_{\max}}$. The chord C_a is defined by points in the intersection of $\partial H^{(t_f)}B$ with Ω_b, and is therefore also a chord of $H^{(t_f)}B$. In this case, the optimal C_a is shown.

no farther than r apart. Then, by Lemma 5.20, if I_r is a line segment between two of these adjacent points, the robot can penetrate no further than b through I_r. In other words, the robot cannot reach O by going "between" the adjacent points of the line segments.

However, we have not yet explained how to sample the arcs of ∂O_{buf}. We do so next, by finding the arc point spacing a.

5.4.4 The Arc Point Spacing

Note that we cannot necessarily use r as the point spacing distance when sampling the arcs of ∂O_{buf}. To understand why, informally, imagine B penetrating into a circle Ω of radius $b \in (0, b_{\max})$ instead of a line segment of length $r_{\max}$ as in Lemma 5.20. Suppose that B stops when it touches the center of the circle. For the sake of argument, suppose that the boundary ∂B (which exists because B is compact by Assumption 5.2) intersects Ω in exactly two points; then, in the intersection of B

with Ω, there is an arc of radius b between these two points. If the length of this arc were equal to r, for an arbitrary convex B, then we could sample "along" each arc by the distance r. But this is not true in general; one can check that it is false if B is circular, as in Example 5.10. Therefore, we need a different point spacing for the arcs, which is the **arc point spacing** a.

Before finding the arc point spacing a, we extend the concepts of passing through and penetrating from line segments to circles and arcs:

Definition 5.21. *Let $\Omega \subset \mathbb{R}^2$ be a circle of radius R with center p. Let B be the robot's body at time 0. Let C be a chord of Ω. Then **passing** B **into** Ω **through** C is defined as passing B through the chord C as in Definition 5.12. If the length of C is less than the width of B, then, by Lemma 5.14, B cannot pass fully through C, but does penetrate the chord up to some distance as in Definition 5.16. Let P_C be the closed half-plane defined by C as in Definition 5.11. The **penetration of** B **into** Ω **through** C is the maximum Euclidean distance from any point in $B \cap C$ to a point in $B \cap P_C^{\mathrm{C}}$.*

This definition is illustrated in Figure 5.3d. We prove that a exists with the following lemma.

Lemma 5.22. *Let B be the robot's body at time 0 with width $r_{\max}$. Let $b_{\max}$ be the maximum penetration distance corresponding to B (as in Lemma 5.17). Pick $b \in (0, b_{\max})$, and let $\Omega \subset (\mathbb{R}^2 \setminus B)$ be a circle of radius b centered at a point $p \in X$. Then there exists a number $a \in (0, r_{\max})$ such that, if C_a is any chord of Ω of length a, then the penetration of B into Ω through C is no larger than b.*

Proof. We begin with a sketch of the proof to build intuition. This proof proceeds much as for Lemma 5.20 to find the point spacing r. To prove that a exists, we pass B through a line segment $I_{r_{\max}}$ of length $r_{\max}$, up to a penetration distance of b. Then, we translate the circle Ω of radius b such that B is penetrating into this circle. From the intersection of the circle with B, we find a chord C_a. The length of C_a depends on the transformation family $\{H^{(t)}\}$ used to pass B through $I_{r_{\max}}$. We search across all such transformation families to find the smallest C_a, the length of which is the desired arc point spacing a.

Now we proceed rigorously. Recall by Assumption 5.2 that B is compact, convex, and has nonzero volume. Let $I_{r_{\max}} \subset (\mathbb{R}^2 \setminus B)$ be a line segment of length $r_{\max}$. As in Lemma 5.17 (used to find $b_{\max}$), suppose without loss of generality that $I_{r_{\max}}$ is oriented vertically, with its lower endpoint fixed at the origin, so $I_{r_{\max}} = \{0\} \times [0, r_{\max}]$. Suppose without loss of generality that B lies fully in the left half-plane, which is $P_{r_{\max}}$, the half-plane defined by $I_{r_{\max}}$. This can be done without loss of generality because it only requires rotation and translation of B and $I_{r_{\max}}$, which can be undone.

Let $\mathcal{H}_{r_{\max}}$ be the set of all transformation families that attempt to pass B through $I_{r_{\max}}$. By Lemma 5.20, there exist $\{H^{(t)}\} \in \mathcal{H}_{r_{\max}}$ for which the penetration distance of B through $I_{r_{\max}}$ is equal to b. Such $\{H^{(t)}\}$ are feasible solutions to (5.33). Let $L_b = \{b\} \times \mathbb{R}$ be the vertical line at $x = b$. Let $\{H^{(t)}\}$ be a feasible solution to (5.33). Then, there exists at least one point in $H^{(t_f)}B$ that lies on L_b. Let C_b denote the set $H^{(t_f)}B \cap L_b$, which is a chord of $H^{(t_f)}B$ [Str82, Theorem 1]. Note that C_b may have length 0, i.e. it is a point, and that C_b is compact, because it is the intersection of two compact sets [Mun00, Theorem 17.1 and Theorem 26.2]. Place the circle Ω (with radius b) tangent to the y-axis, and centered at any point $p_b \in C_b$. Let Ω_b denote this translation of Ω. Recall the function δ_x from (5.29), which returns the right-most point of a set in $\mathbb{R}^2$. With these objects, we pose following program to find the shortest chord C_a for which B penetrates into Ω_b through C_a by the buffer distance b:

$$
\begin{aligned}
a \quad = \quad & \inf_{\{H^{(t)}\},p_b,p_1,p_2} && \|p_1 - p_2\|_2 && (5.36)\\
& \text{s.t.} && \{H^{(t)}\} \in \mathcal{H}_{r_{\text{max}}} && (5.37)\\
& && \delta_x(H^{(t_f)}B) = b, && (5.38)\\
& && p_b \in L_b \cap H^{(t_f)}B, && (5.39)\\
& && p_1, p_2 \in \Omega_b \cap \partial H^{(t_f)}B, && (5.40)
\end{aligned}
$$

where p_1 and p_2 are the endpoints of C_a.

We now construct a feasible solution to (5.36). Let $\{H^{(t)}\}$ be a feasible solution to (5.33), so $\delta_x(H^{(t_f)}B) = b$, which satisfies (5.37) and (5.38). Since $L_b \cap H^{(t_f)}B$ is nonempty as discussed above, we can pick p_b to satisfy (5.39), and create Ω_b centered at p_b. Then $A_b = \Omega_b \cap H^{(t_f)}B$ is an arc of radius b; we justify that A_b is indeed an arc in the next paragraph. Let p_1 and p_2 be the endpoints of A_b, satisfying (5.40). Let C_a be the chord that lies between the endpoints of A_b. Then, $H^{(t_f)}B$ penetrates into Ω_b through C_a by the distance b. This is illustrated in Figure 5.6c.

Now we justify that A_b is indeed an arc of radius b with two endpoints. First, notice that the intersection $\Omega_b \cap H^{(t_f)}B$ is nonempty for two reasons. One, because Ω_b is centered on a point in $\partial H^{(t_f)}B$; and two, because $\delta_x(H^{(t_f)}B) = b$, which implies that there exists at least one line segment inside $H^{(t_f)}B$ that is in the open right half-plane and of length b. Furthermore, because $H^{(t_f)}B$ has nonzero volume (Assumption 5.2), A_b has exactly two endpoints, which lie on the boundary of $H^{(t_f)}B$. Otherwise, there would exist a pair of points in $H^{(t_f)}B$ that are connected by a line segment that does not lie fully in $H^{(t_f)}B$, which would violate the convexity of $H^{(t_f)}B$.

Now, we check that $a \in (0, r_{\text{max}})$. Let $(\{H^{(t)}\}, p_b, p_1, p_2)$ be a feasible solution to (5.36). By construction, B penetrates into Ω_b through C_a by $b < b_{\text{max}}$. Then the length a of C_a is less than r_{max}, otherwise, by Lemma 5.17, B could penetrate into Ω_b through C_a by farther than b. Now suppose that $a = 0$. Then, by Lemma 5.19, there can be no nonempty chords of $H^{(t_f)}B$ between C_a and the center of the circle p_b, but then B does not penetrate into Ω_b through C_a. □

Lemma 5.22 provides the arc point spacing $a \in (0, r_{\text{max}})$, with a constructive method for finding a

for arbitrary compact, convex robot bodies. As with r, we find a analytically for rectangular and circular bodies in Examples 5.9 and 5.10. Note that, by finding $r \in (0, r_{\text{max}})$ with (5.33), then replacing $I_{r_{\text{max}}}$ with I_r (a line segment of length r) in the proof of Lemma 5.22, one can show that $a < r$.

Now we have proven the existence of, and developed methods to find, the geometric quantities r_{max}, b_{max}, r, and a. Next, we use these quantities to construct the discretized obstacle.

5.5 Constructing the Discretized Obstacle for Static Environments

We now present an algorithm to construct the discretized obstacle for static environments. That is, the algorithm takes in a set of *static* obstacles, $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$, create a buffered obstacle O_{buf}, then discretize the buffered obstacle boundary to produce the discretized obstacle O_{disc}. Later, in Theorem 5.23, we prove that, if the robot cannot collide with any point in O_{disc}, then it also cannot collide with the obstacle.

To proceed, first, we review the buffered obstacle. Second, we establish three useful functions that make use of the boundary of the buffered obstacle. Third and finally, we present Algorithm 2 to construct the discretized obstacle.

5.5.1 The Buffered Obstacle

Let $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ consist of polygons, as in Assumption 5.5, *in the planning frame* (recall that we are currently considering the case of static obstacles in the planning frame obstacle reachable set $\mathcal{R}_{\text{obs}}$). Suppose B is the robot's footprint at time 0, which is compact and convex with nonzero volume by Assumption 5.2. Suppose that r_{max} is found for B as in Definition 5.15 and b_{max} as in Lemma 5.17. Select $b \in (0, b_{\text{max}})$, then find r with (5.33) and a with (5.36). Buffer the obstacle to produce O_{buf} as in (5.21), which we restate here:

$$O_{\text{buf}} = \left\{ q \in W \mid \exists\, O \in \{O^{(n)}\}_{n=1}^{n_{\text{obs}}} \text{ and } p \in O \text{ s.t. } \|p - q\|_2 \leq b \right\} \tag{5.41}$$
$$= \mathsf{buffer}(\{O^{(n)}\}_{n=1}^{n_{\text{obs}}},\ b). \tag{5.42}$$

Now, we can discretize ∂O_{buf}.

5.5.2 Sampling the Boundary of the Buffered Obstacle

We now introduce three functions, $\mathsf{getLineSegments}$, $\mathsf{getArcs}$, and sample to discretize the boundary of the buffered obstacle.

The first two functions extract the lines and arcs from the boundary of the buffered obstacle. Recall that, by Lemma 5.7, we can break ∂O_{buf} into a finite set of line segments, $\mathcal{I}$, and a finite set of arcs $\mathcal{A}$. The function $\mathsf{getLineSegments}$ takes in the buffered obstacle O_{buf} and returns the set $\mathcal{I}$ of all line segments on ∂O_{buf}. Similarly, the function $\mathsf{getArcs}$ takes in O_{buf} and returns the set $\mathcal{A}$ of all arcs on ∂O_{buf}.

We now define a third function, $\mathsf{sample} : \text{pow}(\mathbb{R}^2) \times \mathbb{R} \to \text{pow}(\mathbb{R}^2)$, to discretize the line segments and arcs. Suppose $S \subset \mathbb{R}^2$ is a connected curve with exactly two endpoints and no self-intersections; note we are conflating a curve with its image. Let $s > 0$ be a distance. Then $P = \mathsf{sample}(S, s)$ is a set containing the endpoints of S. Furthermore, if the total arclength along S is greater than s, then P also contains a finite number of points spaced along S such that, for any point in P, there exists at least one other point that is no farther away than the arclength s along S. Note that the line segments in $\mathcal{I}$ and the arcs in $\mathcal{A}$ can be parameterized, and sample can be implemented using interpolation of a parameterized curve.

5.5.3 Constructing the Discretized Obstacle

Using the functions above, and the geometric quantities developed through this chapter, we produce the discretized obstacle with Algorithm 2.

We now briefly explain the output of Algorithm 2. Suppose that O_{disc} is constructed from a buffered obstacle O_{buf} using Algorithm 2. Then O_{disc} contains the endpoints of each line segment or arc of ∂O_{buf}, since it is constructed using sample. Furthermore, for each line segment of ∂O_{buf}, O_{disc} contains additional points spaced along the line segment such that each point is within the distance r (in the 2-norm) from at least one other point. Similarly, for each arc of ∂O_{buf}, O_{disc} contains points spaced along the arc such that each point is within the arclength a of at least one other point; this implies that distance between any pair of adjacent points along each arc is no more than a. Finally, note that $|O_{\text{disc}}|$ is finite, because (1) there are a finite number of polygons in $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ (see Assumption 5.5), (2) each polygon has a finite number of edges, and (3) r and $a > 0$.

5.6 Proving Safety

Now, we formalize the notion that O_{disc} conservatively represents the obstacles $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$. That is, if the robot avoids collision with every point in O_{disc}, then it avoids every obstacle in $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$.

Algorithm 2 $O_{\text{disc}} = \mathsf{discretizeObstacle}(\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}, b, r, a)$

$O_{\text{buf}} \leftarrow \mathsf{buffer}(\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}, b)$ // buffer static obstacles by b, and take their union
$\mathcal{I} \leftarrow \mathsf{getLineSegments}\,(O_{\text{buf}})$
$\mathcal{A} \leftarrow \mathsf{getArcs}\,(O_{\text{buf}})$
$O_{\text{disc}} \leftarrow \emptyset$ // initialize output
for $I \in \mathcal{I}$
    $O_{\text{disc}} \leftarrow O_{\text{disc}} \cup \mathsf{sample}(I, r)$
end for
for $A \in \mathcal{A}$
    $O_{\text{disc}} \leftarrow O_{\text{disc}} \cup \mathsf{sample}(A, a)$
end for
return O_{disc}

In other words, we seek to prove (5.9) (restated here in Theorem 5.23).

Theorem 5.23. *Let B be the robot's body with width $r_{\max}$. Let $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}} \subset (W \setminus B)$ be a set of static obstacles in the robot's planning frame, with corresponding unsafe parameters $K_{\text{unsf}} = \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \mathcal{R}_{\text{obs}})$ as in 5.5, where $\text{proj}_W(\mathcal{R}_{\text{obs}})$ is the union of all of the obstacles. Suppose that the maximum penetration depth $b_{\max}$ is found for B as in Lemma 5.17. Pick $b \in (0, b_{\max})$, and find the point spacing r with* (5.33) *and the arc point spacing a with* (5.36)*. Construct the buffered obstacle O_{buf} as in* (5.21)*, then construct the discretized obstacle O_{disc} using Algorithm 2. Then, the set of all unsafe trajectory parameters corresponding to $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ is a subset of the trajectory parameters corresponding to O_{disc}. That is, if we define*

$$K_{\text{disc}} = \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap (T_{\text{plan}} \times O_{\text{disc}} \times K)), \tag{5.43}$$

then

$$K_{\text{unsf}} \subseteq K_{\text{disc}}. \tag{5.44}$$

Proof. In short, we show that any trajectory parameter outside of K_{disc} cannot cause any point on the robot to enter any obstacle in $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ at any time $t \in [0, t_{\text{f}}]$.

First, recall that the robot's high-fidelity model produces continuous trajectories of the robot's body in $\mathbb{R}^2$. So, we can represent the motion of the robot over the time horizon T_{plan} using a transformation family $\{H^{(t)}\}$.

Second, we review the geometry of the boundary of the buffered obstacle. Suppose $k \in K_{\text{disc}}^{\text{C}}$ is arbitrary, and the robot begins at an arbitrary initial condition $x_{\text{hi},0} \in X_{\text{hi}}$. Let $\{H^{(t)}\}$ be a transformation family that describes the robot's motion when tracking the trajectory parameterized by k. Consider a pair (p_1, p_2) of adjacent points of O_{disc}. Recall that the function sample returns the endpoints of any line segment or arc on ∂O_{buf}, in addition to points spaced along the line segment or arc if necessary. Therefore, by Algorithm 2, (p_1, p_2) is either from a line segment or from an arc of ∂O_{buf} (recall that, by Lemma 5.7, ∂O_{buf} consists exclusively of line segments and arcs). By construction, if p_1 is on a line segment (resp. arc), then p_2 is within the distance r (resp. a) along the line segment; this also holds if either point is an endpoint of a line segment or arc. So, to prove the claim, we will consider two cases: (1) where (p_1, p_2) is from a line segment, and (2) where (p_1, p_2) is from an arc.

Consider the case when (p_1, p_2) is from a line segment I of ∂O_{buf}. By (5.21), the distance from $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ to any point on I is b. By the definition of K_{unsf} and by Lemma 5.1, when tracking the trajectory parameterized by k, the robot can approach infinitesimally close to p_1 and/or p_2, but cannot contain them, for any $t \in [0, t_{\text{f}}]$. Then, by continuity of the robot's trajectory and the construction of r via Lemma 5.20, no point in the robot can penetrate farther than b through I.

Now consider when (p_1, p_2) is from an arbitrary arc A of ∂O_{buf}. By Equation (5.21), there exists some obstacle $O^{(n)}$ for which the distance from $O^{(n)}$ to any point on A is b. Each such arc is a section of a circle of radius b. By Lemma 5.1, the robot cannot collide with p_1 or p_2 for any $t \in [0, t_{\text{f}}]$. So, by continuity of the robot's trajectory and by Lemma 5.22, the robot cannot pass farther than the distance b into A through the chord (of A) with endpoints p_1 and p_2.

Since I and A were arbitrary, there does not exist any $t \in [0, t_{\text{f}}]$ for which $H^{(t)}B \cap O^{(n)}$ is nonempty for any $O^{(n)} \in \{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$. In other words, the robot does not collide with any obstacle by passing through any line segment or arc of ∂O_{buf}. Since k was arbitrary, we conclude that there does not exist any $k \in K_{\text{disc}}^{\text{C}}$ for which the robot collides with any obstacle, completing the proof. $\square$

Theorem 5.23 provides the main result of this chapter: we can use O_{disc} to conservatively approximate K_{unsf} for static obstacles. That is, we can represent obstacles using only a discrete, finite subset of the robot's workspace, and *not lose safety guarantees*.

5.7 Extension to Dynamic Obstacles

Up to this point, we have developed a discretized obstacle representation for collections of static obstacles. We now extend the discretized obstacle to incorporate temporal information, enabling collision-free guarantees with respect to dynamic obstacles. Recall that, in this work, the robot is not at fault if it is stopped during a collision, per §3.4.2. Since every parameterized plan ends with the robot stopped, if we can ensure collision avoidance for the entirety of a single plan, then the robot is perpetually not-at-fault (see §3.8 for more details). To this end, we now propose two methods of representing dynamic obstacles with a collection of discrete points.

To proceed, we first briefly review of dynamic obstacles and the corresponding unsafe trajectory parameters. Second, we briefly restate the geometric quantities necessary for the static discretized obstacle, which we use to construct the discretized dynamic obstacle as well. Third, we present a discretized dynamic obstacle for the case when the FRS is represented in continuous time (such as the first sums-of-squares FRS method in §4). Fourth, we present a discretized dynamic obstacle for the case when the FRS is defined over time intervals (as in §4.6, and later on in §6 and §8).

5.7.1 A Reminder of Dynamic Environments and Unsafe Plans

Let $\{O^{(n)}\}_{n=1}^{n_{\text{obs}}}$ be the obstacles that we must consider in the current planning iteration. Suppose that we have mapped the current receding-horizon planning time interval $T^{(i)}$ to the generic planning time horizon T_{plan}; that is, we continue to drop the index i denoting the i^{th} receding-horizon planning iteration. Then, per (3.6), a prediction is a map $\mathcal{P} : T_{\text{plan}} \to \text{pow}\,(W)$ such that $\mathcal{P}(t) \supseteq \bigcup_{n=1}^{n_{\text{obs}}} O^{(n)}(t)$ for any $t \in T_{\text{plan}}$. Further recall that, per §3.4.2, no obstacle travels faster than some known speed $v_{\text{max,obs}} \geq 0$.

Now recall that, we used predictions to define the ORS, $\mathcal{R}_{\text{obs}} \subset T_{\text{plan}} \times W \times K$, in (3.33). The ORS contains all times and points reached by the prediction, and associates each of these times and points with every trajectory parameter; again, we have dropped the index i so we write $\mathcal{R}_{\text{obs}}$ instead of $\mathcal{R}_{\text{obs}}^{(i)}$. Just as we assumed that all obstacles are polygons, we assume the following:

Assumption 5.24. *We assume that, for any $t \in T_{\text{plan}}$, $\text{proj}_{\{t\}}(\mathcal{R}_{\text{obs}})$ is a union of a finite number of closed, compact polygons, each with a finite number of edges and vertices.*

Finally, suppose, as we did in §5.1, that $\mathcal{R}_{\text{FRS}} \subset T_{\text{plan}} \times W \times K$ is the robot's FRS for the current planning iteration (meaning, the FRS corresponding to the robot's initial condition $x_{\text{hi},0}$). The set of unsafe plans for the current iteration is then

$$K_{\text{unsf}} \subseteq \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \mathcal{R}_{\text{obs}}), \tag{5.45}$$

as we saw before in §3.

5.7.2 A Reminder of Geometric Quantities for Obstacle Discretization

For the remainder of this section, suppose we have a robot with body B, width r_{max}, and buffer bound b_{max} as found in §5.4. In each claim in this section, we pick a buffer distance $b \in (0, b_{\text{max}})$, then compute the point spacing r with Lemma 5.20 and the arc point spacing a with Lemma 5.22.

5.7.3 Continuous Time Discretized Dynamic Obstacle

Our first approach is to discretize T_{plan}, then discretize the workspace obstacle at each discrete time. That is, we pick some $n_t \in \mathbb{N}$ such that the discretized obstacle is represented as a collection

$$\begin{aligned} O_{\text{disc}} = \Big\{ & O^{(n)}_{\text{disc}} \in T_{\text{plan}} \times \text{pow}\,(W) \mid O^{(n)}_{\text{disc}} = (n\Delta_t, O^{(n)}), \text{ with} \\ & \Delta_t = \tfrac{t_{\text{f}}}{n_t},\ O^{(n)} \subset W, \text{ and } |O^{(n)}| < \infty\ \forall\, n = 0, \cdots, n_t \Big\}. \end{aligned} \tag{5.46}$$

However, we specify how to construct this discretization in the reverse order. That is, first, we specify how to discretize in space at a single time $n\Delta_t \in T_{\text{plan}}$, and explain the rationale behind doing so. Then, we specify how to upper-bound the time discretization Δ_t. Finally, we prove that using the proposed discretized obstacle representation ensures collision avoidance during the entire time horizon T_{plan}.

First, we specify how to discretize in space at a given time. In short, we sample the boundary of the buffered obstacle, plus enough points in the interior of the buffered obstacle that no point is farther than $r/2$ from another point. That is, we augment the sampling function, sample, which we used to sample the line segments and arcs of the boundary of the buffered (static) obstacle in the previous sections.

Definition 5.25. *We redefine* sample : pow $(W) \times \mathbb{R} \times \mathbb{R} \to$ pow (W) *to take in a buffered polygon and return a (finite) set of discrete points $O^{(n)}$ as follows. Let $O^{(t)}_{\text{buf}} = \mathsf{buffer}(\mathsf{proj}_{\{t\}}(\mathcal{R}_{\text{obs}}),\ b)$, which consists of buffered polygons per Assumption 5.24 and Lemma 5.7. Suppose*

$$O = \mathsf{sample}(O^{(t)}_{\text{buf}}, r, a). \tag{5.47}$$

We require that O has the following properties

1. *If $\mathcal{I}$ and $\mathcal{A}$ are sets containing the line segments and arcs defining $\partial O^{(t)}_{\text{buf}}$, then O contains the endpoints of every such line segment and arc, plus additional points spaced no farther than r (resp. a) apart on every line segment (resp. arc); that is,* sample *returns the output of Algorithm 2.*

2. *The discretized obstacle O also contains points in* interior$(O^{(t)}_{\text{buf}})$ *such that, for any point $o \in O^{(t)}_{\text{buf}}$, there exists a point $o' \in O$ such that its distance to o is bounded by r: $\|o - o'\|_2 \le r$.*

3. *For any point o sampled from $\mathcal{I}$ or $\mathcal{A}$, there exists $o' \in O\ \cap$ interior$(O^{(t)}_{\text{buf}})$ such that its distance to o is bounded by r: $\|o - o'\|_2 \le r$.*

To see how it is possible to fulfill the second and third conditions, recall that $O_{\text{buf}}^{(t)}$ is compact by assumption. Therefore, one can cover $O_{\text{buf}}^{(t)}$ with a finite number of 2-norm balls of radius $r/2$, each centered either on a line segment or arc of $\partial O_{\text{buf}}^{(t)}$, or centered on a point in interior$(O_{\text{buf}}^{(t)})$. Then O can be constructed from the centers of all of these balls.

The reason for sampling the interior of the buffered obstacle is as follows. At the beginning of a planning iteration, our robot may lie *inside* the ORS for a dynamic obstacle at some time $t \in T_{\text{plan}}$; that is, an obstacle may be predicted to occupy the same space that our robot occupies at time $0 \in T_{\text{plan}}$, so we must choose a plan to leave that area. Now suppose that an obstacle is large. Then its prediction may entirely cover the body of our robot. In this case, if we consider only the boundary of the prediction (i.e. $O_{\text{buf}}^{(t)}$ above) for discretizing the obstacle, then it may be possible for us to move *within* the prediction without colliding with any such boundary points; but, doing so would still cause us to collide with the interior of the predicted obstacle. Therefore, we must sample the interior of the predictions to correctly identify unsafe plans.

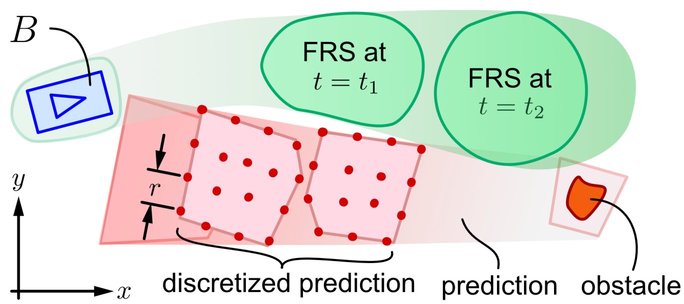


Figure 5.7: Discretized obstacles for dynamic environments. Time is shown fading from light to dark for both the robot and the obstacle prediction. The robot is moving from left to right for a given plan, with the corresponding FRS shown in green for the entire trajectory, and with dark outlines for two times. An obstacle prediction, discretized as in §5.7.3, is shown at the corresponding times. By ensuring collision avoidance at t_1 and $t_2 \in T_{\text{plan}}$, and choosing the buffer size and discretization fineness correctly, we can ensure collision avoidance for all of T_{plan}.

We now show that sample lets us ensure that the robot is collision free at an arbitrary time $t \in T_{\text{plan}}$ (but only at that time).

Lemma 5.26 (Not-at-fault at a time $t \in T_{\text{plan}}$)**.** *Pick $b \in (0, b_{\text{max}})$, and construct r with* (5.33) *and a with* (5.36)*. Let $t \in T_{\text{plan}}$. Suppose we use* sample *to discretize $O_{\text{buf}}^{(t)} = \mathsf{proj}_{\{t\}}(\mathcal{R}_{\text{obs}})$ as in Definition 5.25:*

$$O_{\text{disc}}^{(t)} = \mathsf{sample}(O_{\text{buf}}^{(t)}, r, a) \tag{5.48}$$

Consider the set

$$K_{\text{unsf}}^{(t)} = \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \{t\} \times O_{\text{disc}}^{(t)} \times K). \tag{5.49}$$

Suppose the robot tracks any $k \in K \setminus K_{\text{unsf}}^{(t)}$, and suppose it is not in collision at any time in $[0, t)$. Then, the robot is not in collision at time t.

Proof. From the proof of Theorem 5.23 and the definition of $\mathcal{R}_{\text{obs}}$, we have that the robot is not at fault with respect to the boundary of the obstacles at time t. Let

$$B^{(t,k)} = \text{proj}_W(\mathcal{R}_{\text{FRS}} \cap \{t\} \times W \times \{k\}), \tag{5.50}$$

where we use "B" to remind the reader that this subset of $\mathcal{R}_{\text{FRS}}$ contains all possible locations of the body of the robot for plan k at time t. Consider the case where the prediction is completely overlapping our robot, meaning

$$B^{(t,k)} \subset O_{\text{buf}}^{(t)}. \tag{5.51}$$

To complete the proof, we must show that

$$B^{(t,k)} \cap O_{\text{disc}}^{(t)} \neq \emptyset, \tag{5.52}$$

which would imply that $k \in K_{\text{unsf}}^{(t)}$ (a contradiction). Recall that $O_{\text{disc}}^{(t)}$ contains points in the interior of $O_{\text{buf}}^{(t)}$ such that no two points are farther than r apart, by construction. But, since $r < r_{\text{max}}$ (the width of the robot's body B), there exists no configuration of the robot such that its body can lie inside $O_{\text{buf}}^{(t)}$ without intersecting at least one point in $O_{\text{buf}}^{(t)}$, which follows from Lemma 5.14. Since $B^{(t,k)}$ is not smaller than B by the FRS definition, it follows that $B^{(t,k)} \cap O_{\text{disc}}^{(t)} \neq \emptyset$. By "not smaller" we mean that there exists at least one rotation and translation of B such that $B \subset B^{(t,k)}$ strictly. □

Recall that the purpose of this entire chapter is to find discrete sets of points such that, if the robot avoids collision with all such points, then it avoids collision with the obstacle itself. To this end, Lemma 5.26 tells us that, by picking enough points in the interior of the prediction at time t, we ensure that safe plans force the robot to be outside of the prediction, otherwise its body will overlap the discretized obstacle points. As a reminder, though we have posed Lemma 5.26 in the language of RTD (that is, the robot is tracking a plan $k \in K$), this discretized obstacle representation can be used to formulate collision-avoidance constraints for *any* motion planning method.

Now we extend Lemma 5.26 to a short time interval. First, we define a new type of buffer, and thereby derive bounds on the time discretization.

Definition 5.27. *Recall that our robot has a maximum generalized velocity $\dot{q}_{\text{max}}$ per §3.2.3. Here, let $v_{\text{max}} = \dot{q}_{\text{max}}$ denote the robot's maximum speed in the plane (as a reminder that we are considering robots for which $Q = \mathsf{SE}(2)$). Let $v_{\text{rel}} = v_{\text{max}} + v_{\text{max,obs}}$ (i.e., the maximum relative speed between the robot and any obstacle). We define a **temporal buffer***

$$b_t \in \left(0, \frac{1}{2}t_{\text{f}} \cdot v_{\text{rel}}\right). \tag{5.53}$$

*and a corresponding **maximum time discretization***

$$\Delta_{t,\text{max}} = \frac{2b_t}{v_{\text{rel}}}. \tag{5.54}$$

Now we use the temporal buffer to guarantee the robot is collision-free over a short time interval:

Lemma 5.28 (Collision avoidance for a short time interval)**.** *Pick $b \in (0, b_{\text{max}})$, and construct r with* (5.33) *and a with* (5.36)*. Pick $b_t \in (0, \frac{1}{2}t_{\text{f}} \cdot v_{\text{rel}})$, and $\Delta_t \in (0, \Delta_{t,\text{max}})$. Let $t_1 \in [0, t_{\text{f}} - \Delta_{t,\text{max}}] \subset T_{\text{plan}}$, and $t_2 = t_1 + \Delta_t$. Let*

$$O_{\text{buf}}^{(t_1)} = \mathsf{buffer}(\mathsf{proj}_{\{t_1\}}(\mathcal{R}_{\text{obs}}),\ b + b_t), \tag{5.55}$$

and similarly $O_{\text{buf}}^{(t_2)}$. Then create

$$O_{\text{disc}}^{(t_1)} = \mathsf{sample}(O_{\text{buf}}^{(t_1)}, r, a), \tag{5.56}$$

and similarly $O_{\text{disc}}^{(t_2)}$. Consider the set

$$K_{\text{unsf}}^{(t_1)} = \mathsf{proj}_K(\mathcal{R}_{\text{FRS}} \cap \{t_1\} \times O_{\text{disc}}^{(t_1)} \times K), \tag{5.57}$$

and similarly $K_{\text{unsf}}^{(t_2)}$. Consider an arbitrary trajectory parameter

$$k \in K \setminus (K_{\text{unsf}}^{(t_1)} \cup K_{\text{unsf}}^{(t_2)}). \tag{5.58}$$

If the robot tracks k, and is not in collision at any time in $[0, t_1)$, then it is not in collision at any time in the interval $[t_1, t_2]$.

Proof. By Lemma 5.26, the robot is not in collision at t_1. Furthermore, the closest the robot can be to any obstacle at time t_1 is the distance b_t (since the obstacle is buffered by $b + b_t$. Therefore, for the robot to collide with any obstacle in $[t_1, t_2]$, it must travel a relative distance strictly greater than $2b_t$. However, the farthest the robot can travel between t_1 and t_2 relative to any obstacle is

$$v_{\text{rel}} \cdot \Delta_t \leq v_{\text{rel}} \cdot \frac{2b_t}{v_{\text{rel}}} = 2b_t, \tag{5.59}$$

since $\Delta_t \in (0, \Delta_{t,\text{max}})$. $\square$

The time discretization in Lemma 5.28 is illustrated in Figure 5.7.

Finally, we extend Lemma 5.28 to the whole interval T_{plan}.

Theorem 5.29 (Collision avoidance for all $t \in T_{\text{plan}}$)**.** *Let $\mathcal{R}_{\text{obs}} \subset T_{\text{plan}} \times W \times K$ be the obstacle reachable set for the current planning iteration. Pick $b \in (0, b_{\text{max}})$, and construct r with* (5.33) *and a with* (5.36)*. Pick $b_t \in (0, \frac{1}{2}t_{\text{f}} \cdot v_{\text{rel}})$, and construct $\Delta_{t,\text{max}} = 2b_t/v_{\text{rel}}$. Choose $n_t \in \mathbb{N}$ such that $\Delta_t = t_{\text{f}}/n_t \leq \Delta_{t,\text{max}}$. Let*

$$T_{\text{disc}} = \{0, \Delta_t, 2\Delta_t, \cdots, n_t\Delta_t\}. \tag{5.60}$$

For each $t^{(n)} \in T_{\text{disc}}$, construct

$$O_{\text{buf}}^{(n)} = \mathsf{buffer}(\mathsf{proj}_{\{t^{(n)}\}}(\mathcal{R}_{\text{obs}}),\ b + b_t), \tag{5.61}$$

$$O_{\text{disc}}^{(n)} = \mathsf{sample}(O_{\text{buf}}^{(n)}, r, a),\ \text{and} \tag{5.62}$$

$$K_{\text{unsf}}^{(n)} = \mathsf{proj}_K(\mathcal{R}_{\text{FRS}} \cap \{t^{(n)}\} \times O_{\text{disc}}^{(n)} \times K). \tag{5.63}$$

Consider an arbitrary trajectory parameter

$$k \in K \setminus \left(\bigcup_{n=1}^{n_t} K_{\text{unsf}}^{(n)} \right). \tag{5.64}$$

Suppose the robot tracks k, is not in collision at $t = 0$, and does not begin tracking any new plan during T_{plan}. Then it is not in collision at any $t \in T_{\text{plan}}$, and it is not-at-fault for any $t \geq t_{\text{f}}$.

Proof. The collision free claim follow follows from applying Lemma 5.28 successively on each time interval $[(n-1) \cdot \Delta_t, n \cdot \Delta_t]$ for $n = 1, \cdots, n_1 - 1$. Since every $k \in K$ ends with the robot stopped, the robot is not-at-fault for all time after t_{f}. $\square$

Theorem 5.29 lets us guarantee collision-free behavior by discretizing the dynamic obstacle reachable set in time and space. That is, we have constructed the representation desired in (5.46).

Unfortunately, as one might notice from Definition 5.25, there is a tradeoff between conservatism and discretization fineness. That is, one must use a larger temporal buffer b_t to enable a larger time discretization via $\Delta_{t,\text{max}}$. A smaller $\Delta_{t,\text{max}}$ is therefore preferable. But choosing a smaller $\Delta_{t,\text{max}}$ results in more discrete obstacle points. Recall that we treat each point as a collision-avoidance constraint at runtime; in general, more constraints results in slower online trajectory optimization (see, e.g., [KVJRV17] or [KZZV20, Section V]). Since we limit RTD to a duration t_{plan} in each receding-horizon planning iteration, having fewer constraints is preferable. We address this challenge next.

5.7.4 Time Interval Discretized Dynamic Obstacle

Recall that, in §4.6, we broke the planning time horizon T_{plan} into a collection of n_{RS} intervals $\{I^{(n)}\}_{n=1}^{n_{\text{RS}}}$, with the intention being to treat predictions of obstacles as static in each of these intervals (note, we are now using $I^{(n)}$ to refer to time intervals as in §4.6, not line segments as we did in §5.2.1). From the dynamic discretized obstacle construction above, one may notice that, if we can choose $n_{\text{RS}} < n_t$, then we should be able to produce less discretization points, and therefore less constraints for online trajectory optimization. However, there is an additional benefit: by treating obstacles as static for each $I^{(n)}$, we eliminate the temporal buffer b_t, and therefore reduce the conservatism of our approach as well.

We make use of the same sampling strategy as above, and essentially restate Lemma 5.28 for the case when T_{plan} (and the ORS) is broken into n_{RS} short time intervals. First, we have to assume that we still have a polygonal representation of predictions:

Assumption 5.30. *Let $\mathcal{R}_{\text{obs}} \subset T_{\text{plan}} \times W \times K$ be the obstacle reachable set for the current planning iteration. Suppose that we have broken the planning time horizon in to $n_{\text{RS}} \in \mathbb{N}$ intervals, so that $T_{\text{plan}} = \bigcup_{n=1}^{n_{\text{RS}}} I^{(n)}$. We assume that, for any $I^{(n)}$, the set $\text{proj}_{\{I^{(n)}\}}(\mathcal{R}_{\text{obs}})$ is a polygon, or can be overapproximated by a polygon.*

Now we can construct the discretized dynamic obstacle on the time intervals $I^{(n)}$:

Theorem 5.31. *Let $\mathcal{R}_{\text{obs}}$ and $\{I^{(n)}\}_{n=1}^{n_{\text{RS}}}$ be as in Assumption 5.30. Pick $b \in (0, b_{\text{max}})$, and construct r with* (5.33) *and a with* (5.36)*. For each $I^{(n)}$, construct*

$$O_{\text{buf}}^{(n)} = \mathsf{buffer}(\text{proj}_{\{I^{(n)}\}}(\mathcal{R}_{\text{obs}}),\ b), \tag{5.65}$$

$$O_{\text{disc}}^{(n)} = \mathsf{sample}(O_{\text{buf}}^{(n)}, r, a),\ \text{and} \tag{5.66}$$

$$K_{\text{unsf}}^{(n)} = \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \{t^{(n)}\} \times O_{\text{disc}}^{(n)} \times K). \tag{5.67}$$

Consider an arbitrary trajectory parameter

$$k \in K \setminus \left(\bigcup_{n=1}^{n_t} K_{\text{unsf}}^{(n)} \right). \tag{5.68}$$

Suppose the robot tracks k, is not in collision at $t = 0$, and does not begin tracking any new plan during T_{plan}. Then it is not in collision at any $t \in T_{\text{plan}}$, and it is not-at-fault for any $t \geq t_{\text{f}}$.

Proof. This result follows by applying Lemma 5.26 (which guarantees collision-free behavior at any $t \in T_{\text{plan}}$) iteratively to the $\mathcal{R}_{\text{obs}}$ projected into each interval $I^{(n)}$ for $n = 1, \cdots, n_{\text{RS}}$; that is, we treat the predictions of obstacles as static in each $I^{(n)}$, and are effectively taking the union of all predicted obstacle positions during each $I^{(n)}$. As a reminder, since each $k \in K$ ends with the robot stopped, the robot is not-at-fault for all time after $t_{\text{f}} \in T_{\text{plan}}$. $\square$

This concludes the presentation of discretized obstacles for dynamic environments, and therefore concludes the theoretical development of the chapter.

5.8 Chapter Review

The takeaway of this chapter is a general discretized obstacle formulation based on the geometry of the robot. The representation enables real-time online planning for RTD, because it enables converting polygonal representations of obstacles (which contain a continuum of points) into a finite, discrete number of points. Each point becomes a constraint for online trajectory optimization, so this finite list is numerically tractable whereas an infinite list of constraints may not be.

5.8.1 Example Discretized Obstacle Usage for Polynomial FRS

We conclude this presentation of the discretized obstacle with an example of the discretized obstacle in practice, given a polynomial FRS representation generated as in §4. Suppose that $g_{\text{dyn},l} \in \mathbb{R}[t, x, k]$ and $g_{\text{stat},l} \in \mathbb{R}[x, k]$ are degree l polynomials representing the FRS, as computed in §4. Recall that the 0-sublevel set (resp. 1-superlevel set) of $g_{\text{dyn},l}$ (resp. $g_{\text{stat},l}$) provably contains the FRS (see Theorem 4.3 and Corollary 4.4). Let $O_{\text{dyn}} \subset T_{\text{plan}} \times W$ be a discretized dynamic obstacle produced as in Theorem 5.29. Then, for the decision variable k at runtime, we need only enforce the finite list of constraints

$$g_{\text{dyn},l}(t, o, k) > 0 \; \forall \, (t, o) \in O_{\text{dyn}}. \tag{5.69}$$

Similarly, if $O_{\text{stat}} \subset W$ is a discretized (static) obstacle as in Theorem 5.23, we need only enforce the finite list of constraints

$$g_{\text{stat},l}(o, k) < 1 \; \forall \, o \in O_{\text{stat}}. \tag{5.70}$$

Any k that is feasible to (5.69) (resp. (5.70)) is provably collision-free, which follows from Theorem 4.3 and Theorem 5.29 (resp. Theorem 5.23).

In the case of a time interval discretized dynamic obstacle, and time interval polynomial FRS as in §4.6, we can apply Theorem 5.31 with a similar formulation to (5.70).

5.8.2 Chapter Summary

This chapter began by reviewing predictions and formulating a theoretical discretized obstacle representation (§5.1). We then defined several geometric objects, and placed assumptions on the geometry of the robot and obstacles (§5.2.1). The majority of this chapter was then spent defining and computing five geometric quantities that are necessary for producing the discretized obstacle representation for static obstacles (§5.3 and §5.4). We used this quantities to construct the discretized obstacle (§5.5), and proved that this representation ensures safety (§5.6). Finally, we extended our method to the case of dynamic obstacles, and proved again that it ensures collision-free planning (§5.7).

5.8.3 What is Missing?

This chapter has presented a discretized obstacle that one can use for RTD or any other motion planning method for rigid body robots in the plane. However, this representation can generate a large number of discrete points, each of which is typically mapped to a collision-avoidance constraint at runtime. More constraints typically results in slower online trajectory optimization. Furthermore, such constraints are not convex, because we seek to have the robot avoid reaching points in its workspace; so the feasible region for each constraint is the workspace minus a point, which is not convex. To resolve these challenges with nonlinear, nonconvex optimization, we have explored branch-and-bound strategies for RTD [KZZV20]. But, there is still work to be done in finding obstacle representations that produce *fewer* constraints, or a *minimal* number of constraints, to ensure faster online optimization; this work is a first step in this direction, because we explicitly use the robot's geometry to bound the spacing between discrete obstacle points.

CHAPTER 6

Forward Reachable Sets via Zonotopes

In this chapter, we represent the Forward Reachable Set (FRS) using zonotopes, a special class of convex polytopes in Euclidean space. Recall that the SOS FRS representation in Chapter 4 was used only for wheeled robots operating in the plane, due to dimensionality constraints. This limitation motivates our development of zonotope reachable sets, enabling RTD for aerial robots that have high-dimensional, nonlinear models which are currently out of reach of our SOS methods. Later, in §8, we adapt zonotopes into a more general class of objects called rotatotopes for RTD on manipulators.

The sections of this chapter are as follows. (§6.1) We begin by introducing zonotopes. (§6.2) Then, we introduce the zonotope FRS representation. (§6.3) Next, we introduce a concept called *slicing*, which lets us identify subsets of the zonotope FRS that correspond to particular trajectory parameters. (§6.4) Finally, we use slicing to identify unsafe plans and thereby formulate online trajectory optimization with the zonotope FRS.

6.1 Zonotopes

We begin this chapter by defining zonotopes and noting several useful properties that make them amenable to numerical representation of reachable sets.

6.1.1 Definition and Notation

A **zonotope** is a set in $\mathbb{R}^n$ that can be written as the convex combination of a **center** $c \in \mathbb{R}^n$ and **generators** $g^{(1)}, \cdots, g^{(m)} \in \mathbb{R}^n$, $m \in \mathbb{N}$:

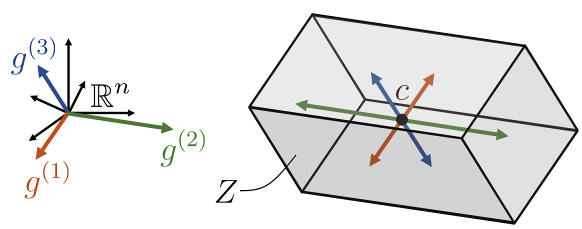
$$Z = \left\{ y \in \mathbb{R}^n \;\middle|\; y = c + \sum_{i=1}^{m} \beta^{(i)} g^{(i)},\ -1 \leq \beta^{(i)} \leq 1 \right\}. \tag{6.1}$$

We refer to the values $\beta^{(i)}$ as the **coefficients** of the zonotope. We represent zonotopes numerically by storing their centers and generators as arrays. Notice that the center and generators uniquely define any zonotope, which means the coefficients can be left implicit for most applications [KHV19]. An example zonotope is shown in Figure 6.1.

However, we make heavy use of the coefficients themselves in this work. This is because our goal is to represent the high-dimensional set $\mathcal{R}_{\text{FRS}} \subset T_{\text{plan}} \times X_{\text{hi}} \times X \times K$; as we will see, constructing generators that are nonzero in some dimensions of $\mathcal{R}_{\text{FRS}}$ allows us to find the subsets of $\mathcal{R}_{\text{FRS}}$ corresponding to particular trajectories or obstacles. Numerically representing such subsets requires choosing particular coefficient values for those generators, while leaving the remaining generators alone. To that end, we introduce the following zonotope notation:

$$Z = c + \sum_{i=1}^{m} \langle \beta^{(i)} \rangle g^{(i)}. \tag{6.2}$$

which means the zonotope Z centered at c, with generators $\left\{g^{(i)}\right\}_{i=1}^{m}$, and **indeterminates** $\left\{\langle \beta^{(i)} \rangle\right\}_{i=1}^{m}$. In other words, we treat $\langle \beta^{(i)} \rangle$ as symbolic coefficients that take (all of the) values in $[-1, 1]$. That is, these indeterminates equivalently represent the interval $[-1, 1]$, and one can use interval arithmetic to understand what it means to multiply them with, e.g., generators [Alt10]. Instead of denoting the indeterminates as intervals, our notation emphasizes their role as coefficients of the generators; later in this section, we *evaluate* indeterminates by assigning them a particular value from $[-1, 1]$. Note, we always use Greek lowercase letters in angle brackets to denote indeterminates.



$$Z = \left\{ c + \beta^{(1)} g^{(1)} + \beta^{(2)} g^{(2)} + \beta^{(3)} g^{(3)} \mid \beta^{(i)} \in [-1, 1] \right\}$$

Figure 6.1: An example zonotope Z (the grey volume) in $\mathbb{R}^n$ with three generators (in orange, green, and blue), and a center c (in black).

6.1.2 Zonotope Properties

We now note two useful properties of zonotopes, which follow from the definition in (6.1). Define two example zonotopes

$$X = x + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle g_X^{(i)} \quad \text{and} \quad Y = y + \sum_{j=1}^{s} \langle \gamma^{(j)} \rangle g_Y^{(j)}. \tag{6.3}$$

First, the *Minkowski sum* of zonotopes is given as follows:

$$X \oplus Y = x + y + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle g_X^{(i)} + \sum_{j=1}^{s} \langle \gamma^{(j)} \rangle g_Y^{(j)}. \tag{6.4}$$

Notice that $X \oplus Y$ is again a zonotope, with $r + s$ generators and indeterminates.

Second, consider a linear map $A : \mathbb{R}^n \to \mathbb{R}^n$ and suppose $X \subset \mathbb{R}^n$. Then

$$AX = Ax + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle A g_X^{(i)}. \tag{6.5}$$

Notice that the indeterminates are not altered by this transformation.

We present a third property, a method for checking if two zonotopes intersect, later in §6.4.

6.2 Zonotope FRS

We now discuss how we represent the FRS with zonotopes. Recall that, informally, FRS = PRS + ERS. We make use of an open-source toolbox [Alt15] to compute the PRS, and reserve our ERS computation for §7. Here, as with the SOS programming approach, we place specific assumptions on the form of the ERS to make the FRS computationally tractable.

In this section, we first discuss how one can conservatively approximate a PRS as in (3.27) with zonotopes. Then, we explain the format required for the ERS. Finally, we produce the FRS.

6.2.1 The Planning Reachable Set

First, recall the PRS definition (3.27):

$$\mathcal{R}_{\text{plan}} = \left\{ (t, x, k) \in T_{\text{plan}} \times X \times K \;\middle|\; x = x_0 + \int_0^t f(\tau, \tilde{x}(\tau; k), k) d\tau \right\}, \tag{6.6}$$

where f denotes the planning model, x_0 is the initial condition of each plan, and $\tilde{x}$ is the trajectory of each plan. An example zonotope PRS is shown in Figure 6.2.

To compute the PRS using [Alt15], we require the following three items. First, we augment the planning model with the parameters k as artificial states such that

$$\begin{bmatrix} \dot{x}(t;k) \\ \dot{k}(t) \end{bmatrix} = \begin{bmatrix} f(t, x(t;k), k) \\ 0 \end{bmatrix}. \tag{6.7}$$

Notice that the trajectory parameters do not evolve over T_{plan}. However, as a reminder, they still parameterize time-varying planned trajectories. A common example, which we make use of in later chapters, is to parameterize the coefficients of a time-varying polynomial in each planning state.

Second, we split the time horizon T_{plan} into $n_{\text{RS}} \in \mathbb{N}$ intervals just as we did for SOS reachability in §4.6. Let $\Delta_t = t_{\text{f}}/n_{\text{RS}}$ and

$$\begin{aligned} T_{\text{plan}} &= [0, \Delta_t] \cup [\Delta_t, 2\Delta_2] \cup \cdots \cup [t_{\text{f}} - \Delta_t, t_{\text{f}}] && (6.8) \\ &= I^{(1)} \cup I^{(2)} \cup \cdots \cup I^{(n_{\text{RS}})} && (6.9) \end{aligned}$$

As before, each $I^{(i)} = [(i-1)\Delta_t, i \cdot \Delta_t]$ for $i = 1, \cdots, n_{\text{RS}}$.

Third, we create an initial condition set as a zonotope:

$$Z^{(0)}_{\text{plan}} = z_0 + \sum_{i=1}^{n_K} \langle \kappa^{(i)}_{k_i} \rangle g^{(i)}_{k_i} \subset X \times K, \tag{6.10}$$

with

$$z_0 = \begin{bmatrix} x_0 \\ k_0 \end{bmatrix} \quad \text{and} \quad g^{(i)}_{k_i} = \begin{bmatrix} 0_{n_X \times 1} \\ \Delta_{k_i} e_{k_i} \end{bmatrix}. \tag{6.11}$$

We specify $k_0 \in \mathbb{R}^{n_K}$ and $\Delta_{k_i} > 0$; we use $e_{k_i} \in \mathbb{R}^{n_K}$ to denote a vector of zeros with 1 in the i^{th} coordinate. The generators $g^{(i)}_{k_i}$ have the subscript k_i to denote that they correspond to each parameter k_i; they have the superscript index i to index each of the n_K such generators.

Another way to think of the generators $g^{(i)}_{k_i}$ is as the columns of a matrix

$$G_k = \begin{bmatrix} 0_{n_X \times n_K} \\ \text{diag}\left(\Delta_{k_1}, \Delta_{k_2}, \cdots, \Delta_{k_{n_K}}\right) \end{bmatrix} \in \mathbb{R}^{(n_X + n_K) \times n_K}, \tag{6.12}$$

where diag places its arguments on the diagonal of a matrix of appropriate size, with all other entries as zero. In fact, a common parameterization of zonotopes is as a *center vector* and a *generator matrix* in this form, which is also typically how zonotopes are represented numerically [Alt15].

Notice that each generator $g_{k_i}^{(i)}$ causes $Z_{\text{plan}}^{(1)}$ to span the distance $2\Delta_{k_i}$ in the i^{th} coordinate of K. Therefore, this representation assumes that K is a *box*-shaped set, meaning that each i^{th} coordinate of k is drawn from a closed interval centered at the k_0, and K is the Cartesian product of all n_K of these intervals. For example,

$$K = [k_{0,1} - \Delta_{k_1}, k_{0,1} + \Delta_{k_1}] \times [k_{0,2} - \Delta_{k_2}, k_{0,2} + \Delta_{k_2}] \times \cdots \tag{6.13}$$

$$\cdots \times [k_{0,n_K} - \Delta_{k_{n_K}}, k_{0,n_K} + \Delta_{k_{n_K}}] \tag{6.14}$$

in the case where $n_K > 2$.

Using these dynamics, time intervals, and initial condition, [Alt15] then produces a set of zonotopes for which

$$Z_{\text{plan}}^{(i)} = (F^{(i-1)} Z_{\text{plan}}^{(i-1)}) \oplus L^{(i)} \subset X \times K, \tag{6.15}$$

where $F^{(i-1)}$ is the matrix exponential of the linearized augmented dynamics (6.7), and $L^{(i)}$ is a zonotope that compensates for linearization error and continuous time [Alt15]. By continuous time, we mean that each $Z_{\text{plan}}^{(i)}$ contains all states reached by the planning model at any time in the interval $I^{(i)}$. Applying the operation (6.15) n_{RS} times produces a set of zonotopes denoted

$$\{Z_{\text{plan}}^{(i)}\}_{i=1}^{n_{\text{RS}}}. \tag{6.16}$$

Notice that the index runs from 1 to n_{RS}, to match the time intervals $I^{(i)}$. That is, $Z_{\text{plan}}^{(0)}$ is only used as an initial condition, and is subsumed into $Z_{\text{plan}}^{(1)}$ during the first application of (6.15).

Importantly, using [Alt10, Theorem 3.3 and Proposition 3.7], one can prove that the set $\{Z_{\text{plan}}^{(i)}\}_{i=1}^{n_{\text{RS}}}$ *conservatively* approximates the PRS:

Lemma 6.1. *If* $(t, x, k) \in \mathcal{R}_{\text{plan}}$ *and* $t \in I^{(i)} \subset T_{\text{plan}}$, *then* $(x, k) \in Z_{\text{plan}}^{(i)}$.

6.2.2 The Error Reachable Set

Now we specify an ERS zonotope representation. First, recall the ERS definition from (3.28):

$$\begin{aligned}\mathcal{R}_{\text{err}} = \Big\{ &(t, x_{\text{hi},0}, e) \in T_{\text{plan}} \times X_{\text{hi},0} \times \mathbb{R}^{n_{\text{hi}}} \mid \exists\, k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}) \text{ s.t.} \\ &e = x_{\text{hi}}(t;k) - x_{\text{plan}}(t;k), \text{ where} \\ &\dot{x}_{\text{hi}}(t;k) = f\big(t, x_{\text{hi}}(t;k), u_k(t, x_{\text{hi}}(t;k))\big), \ x_{\text{hi}}(0;k) = x_{\text{hi},0}, \\ &\dot{x}_{\text{plan}}(t;k) = f_{\text{lift}}\big(t, x_{\text{plan}}(t;k), k\big), \text{ and } x_{\text{plan}}(0;k) = x_{\text{hi},0} \Big\}.\end{aligned} \tag{6.17}$$

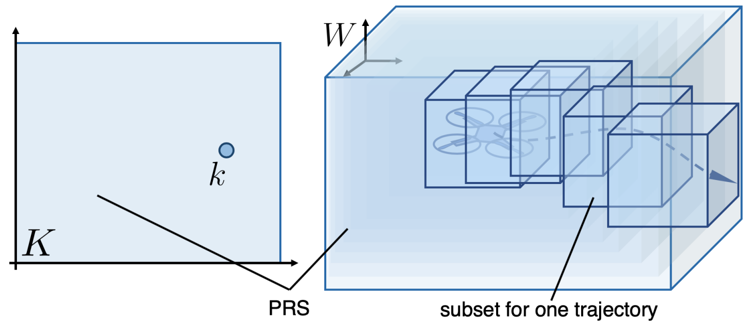


Figure 6.2: An illustration of the PRS for an aerial robot. The PRS is shown as a sequence of high-dimensional zonotopes, projected into K and W as boxes. The particular subset of the PRS corresponding to one plan k is also shown, with the resulting sliced PRS shown as a sequence of zonotopes surrounding the trajectory parameterized by k. This subset is found by slicing the zonotope PRS as in (6.28).

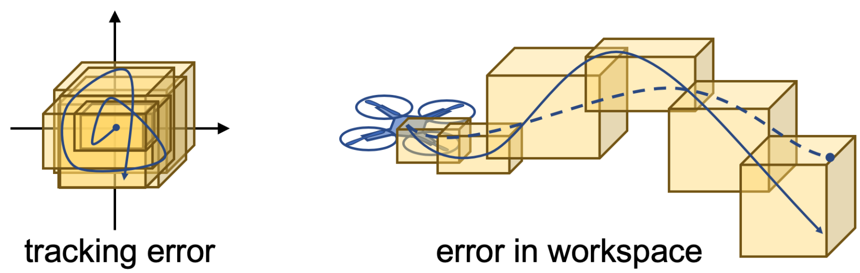


Figure 6.3: An illustration of the ERS as a collection of zonotopes for a single trajectory plan and the resulting tracking error. The tracking error zonotopes are shown in the space $\mathbb{R}^{n_{\text{hi}}}$ on the left, along with the tracking error as a solid blue curve. The planned trajectory is a dashed curve on the right, with the executed trajectory as a solid curve. The tracking error zonotopes are overlaid on both trajectories to show how they can be constructed to contain the error when they are shifted to contain the planned trajectory.

We require the following ERS representation for the zonotope FRS.

Assumption 6.2. *Suppose we partition the space $X_{\text{hi},0}$ into a finite number of subsets. For each subset $X_{\text{hi},0}^{(j)}$ and each $I^{(i)} \subset T_{\text{plan}}$, we assume that there exists a zonotope $Z_{\text{err}}^{(i,j)} \subset \mathbb{R}^{\dim W}$ with the following property. If $(t, x_{\text{hi},0}, e) \in \mathcal{R}_{\text{err}}$, $t \in I^{(i)}$, and $x_{\text{hi},0} \in X_{\text{hi},0}^{(j)}$, then $\text{proj}_X(e) \in Z_{\text{err}}^{(i,j)}$.*

Here, $\text{proj}_X(e)$ denotes that we are only considering the X dimensions of $e \in \mathbb{R}^{n_{\text{hi}}}$. Note, this partition of $X_{\text{hi},0}$ is essentially the same as the one used for FRS swapping in §4.7. However, in practice, computing each $Z_{\text{err}}^{(i,j)}$ is much less computationally intensive than computing an FRS using the SOS programming approach; so, we typically use a much finer partition of $X_{\text{hi},0}$ for the zonotope approach. A subset of the ERS, represented with zonotopes, is shown in Figure 6.3.

6.2.3 The Forward Reachable Set

Now, we conservatively approximate the FRS with zonotopes, which enable a literal application of the informal notion, FRS = PRS + ERS. To do so, we first have to specify how to add zonotopes of different dimension.

Definition 6.3. *Consider two zonotopes,*

$$X = x + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle g_X^{(i)} \quad \text{and} \quad Y = y + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle g_Y^{(j)} \tag{6.18}$$

with $X \subset \mathbb{R}^n$ and $Y \subset \mathbb{R}^m$ (note $r, s \in \mathbb{N}$ per the zonotope notation). Suppose that $n < m$. We specify the Minkowski sum of these zonotopes as

$$X \oplus Y = \begin{bmatrix} x \\ 0_{(m-n)\times 1} \end{bmatrix} + y + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle \begin{bmatrix} g_X^{(i)} \\ 0_{(m-n)\times 1} \end{bmatrix} + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle g_Y^{(j)}. \tag{6.19}$$

In other words, we pad x and each $g_X^{(i)}$ with an appropriate number of zeros. Note, the order of the dimensions of a zonotope are arbitrary. Here we assume the first n dimensions of $Y \subset \mathbb{R}^m$ correspond to all n dimensions of $X \subset \mathbb{R}^n$.

Now, recall the set X_0 from 4.2 that is located at $x_0 \in X$, and is large enough to contain all rotations of the robot's body when tracking any parameterized trajectory. We assume X_0 exists for wheeled or aerial robots when using the zonotope FRS method; we use a different formulation of the FRS for manipulators that does not require X_0.

Theorem 6.4. *Suppose $X_0 \subset X$, $x_0 \in X_0$, and X_0 is large enough to contain all rotations of a robot's rigid body in the case of a wheeled or aerial robot; let $X_0 = \emptyset$ for a manipulator. Further suppose that X_0 is a zonotope. If $(t, x_{\text{hi},0}, x, k) \in \mathcal{R}_{\text{FRS}}$, $t \in I^{(i)}$, and $x_{\text{hi},0} \in X_{\text{hi},0}^{(j)}$, then*

$$(x, k) \in X_0 \oplus Z_{\text{plan}}^{(i)} \oplus Z_{\text{err}}^{(i,j)} =: Z_{\text{FRS}}^{(i,j)}, \tag{6.20}$$

where the addition of zonotopes with mismatched dimension is as in Definition 6.3.

Proof. This follows from the FRS definition (3.26), Lemma 6.1 (which ensures the zonotope PRS is conservative), and Assumption 6.2 (which ensures the zonotope ERS is conservative). $\square$

In other words, we construct the PRS and ERS zonotopes so that they overapproximate the FRS when added together (for each time in T_{plan}, and for a given initial condition). We call each $Z_{\text{FRS}}^{(i,j)}$ an **FRS zonotope**.

6.3 Slicing the Zonotope FRS

As mentioned earlier, we make heavy use of the indeterminate representation of a zonotope's coefficients. In particular, we use them for *slicing*, wherein we evaluate a zonotope's indeterminates to produce a new zonotope that is a subset, or *slice*, of the original. This operation allows us to identify the subset of a zonotope FRS that corresponds to a particular trajectory or obstacle, and therefore enables online trajectory optimization with a zonotope FRS. In this section, we first define slicing, then show how it applies to the FRS zonotope.

6.3.1 Slicing Definition

To define slicing, we first denote the **evaluation** of an indeterminate $\langle\beta\rangle$ by removing the angle brackets, so $\beta \in [-1, 1]$. Now, we use evaluation to define **slicing**. Consider an arbitrary zonotope $Z = c + \sum_{i=1}^{m} \langle\beta^{(i)}\rangle g^{(i)}$. Then

$$\mathsf{slice}\big(Z, \langle\chi^{(j)}\rangle, \chi^{(j)}\big) = c + \chi^{(j)} g^{(j)} + \sum_{i\neq j,\ i\leq m} \langle\chi^{(i)}\rangle g^{(i)}. \tag{6.21}$$

By picking a value $\chi^{(j)} \in [-1, 1]$ for the j^{th} indeterminate $\langle\chi^{(j)}\rangle$, we produce a zonotope with fewer generators/indeterminates. Notice that, since the center is linearly combined with the generators per (6.1), when we evaluate an indeterminate, we shift the center of the original zonotope.

We can extend slicing to take in multiple indeterminates at a time. Collect the indices i in $I = \{1, \cdots, m\}$ and let $J \subset I$. Then we denote slicing as

$$\mathsf{slice}\big(Z, \{\langle\chi^{(j)}\rangle\}_{j\in J}, \{\chi^{(j)}\}_{j\in J}\big) = c + \sum_{j\in J} \chi^{(j)} g^{(j)} + \sum_{i\in I\setminus J} \langle\chi^{(i)}\rangle g^{(i)}. \tag{6.22}$$

6.3.2 Sliceability

Now we define the notion of *sliceable* generators, which is most easily understood with an example. Let $Z = c + \langle\beta\rangle g$, with just one indeterminate/generator. Suppose indeterminate $\langle\alpha\rangle$ is passed into $\mathsf{slice}(Z, \cdot, \cdot)$, and notice that $\langle\alpha\rangle$ is not paired with any generators of Z. We would expect that none of the indeterminates of the zonotope are evaluated. That is,

$$\mathsf{slice}(Z, \langle\beta\rangle, \beta) = c + \beta g, \text{ but} \tag{6.23}$$

$$\mathsf{slice}(Z, \langle\alpha\rangle, \alpha) = c + \langle\beta\rangle g. \tag{6.24}$$

In this case, we say that the generator g is **sliceable** by $\langle\beta\rangle$, or $\langle\beta\rangle$**-sliceable**.. Similarly, g is *not* $\langle\alpha\rangle$-sliceable. Sliceability is important because, as one might have noticed from the construction of the zonotope FRS, not all generators are sliceable by the indeterminates that represent the trajectory parameters.

In §8, we revisit slicing for a larger class of zonotope-like objects called *rotatotopes*, which we use to represent rotations of a manipulator's links.

6.3.3 Slicing the Zonotope FRS

Now we apply slicing to the zonotopes representing the FRS. First, we briefly review why slicing is useful. Notice that each FRS zonotope $Z_{\text{FRS}}^{(i,j)}$ as in (6.20) is defined over $X \times K$. Recall, per §3.8, that the parameters are our decision variables for online trajectory optimization. Therefore, for any $k \in K$, we want to ensure that the *subset* of each $Z_{\text{FRS}}^{(i,j)}$ corresponding to k lies outside of obstacles. Slicing allow us to identify such subsets of the FRS zonotopes.

To make slicing tractable, we must identify which generators of an FRS zonotope are, in fact, sliceable per §6.3.2. We formalize this notion with k**-sliceable generators**, the existence of which is proven in the following lemma:

Lemma 6.5. *Suppose $Z_{\text{FRS}}^{(i,j)} = X_0 \oplus Z_{\text{plan}}^{(i)} \oplus Z_{\text{err}}^{(j)} \subset X \times K$. Then $Z_{\text{FRS}}^{(i,j)}$ has at least n_K generators $\{g_{k_n}^{(n)}\}_{n=1}^{n_K}$, and associated indeterminates $\{\langle \kappa_{k_n}^{(n)} \rangle\}_{n=1}^{n_K}$, with the following two properties. First, each $g_{k_n}^{(n)} \in \mathbb{R}^{n_X + n_K}$ is zero in all of its entries corresponding to K, except for a single nonzero element Δ_{k_n} in the n^{th} entries. Second, each $g_{k_n}^{(n)}$ may have nonzero elements in the entries corresponding to the X dimensions of $Z_{\text{FRS}}^{(i,j)}$.*

Proof. We prove this claim by induction. Notice that $Z_{\text{plan}}^{(0)}$ satisfies these conditions on its generators (and indeterminates) by construction. Recall that each $Z_{\text{plan}}^{(i)}$ is constructed as in (6.15).

First, we check that the claim holds for $Z_{\text{plan}}^{(1)}$. Since the linearized dynamics represented by $F^{(0)}$ are zero in the k dimensions, it follows from (6.5) that $Z_{\text{plan}}^{(1)}$ has the same values as $Z_{\text{plan}}^{(0)}$ for each $g_{k_n}^{(n)}$ in the k dimensions. Also, recall from (6.5) that the operation $F^{(0)} Z_{\text{plan}}^{(0)}$ does not alter any of the indeterminates of $Z_{\text{plan}}^{(0)}$. Next, notice that the zonotope $L^{(i)}$ (which compensates for linearization error and continuous time) does not add any volume in the k dimensions [Alt15], because the augmented model (6.7) is 0 in those dimensions, and because the Minkowski sum of zonotopes increases the number of generators, as opposed to altering the generators themselves. By the same logic, this operation increases the number of indeterminates, but does not change any of the existing indeterminates. Finally, notice that the addition of $X_0 \subset \mathbb{R}^{n_X}$ and $Z_{\text{err}}^{(1,j)} \subset \mathbb{R}^{n_X}$ to produce $Z_{\text{FRS}}^{(1)}$ does not add any generators with nonzero volume in the k dimensions, by Definition 6.3. To see why, recall that the first n_X entires of the center and generators correspond to X, and the remaining n_K correspond to K, so the zero padding in Definition 6.3 holds as written when adding X_0 and $Z_{\text{err}}^{(1,j)}$ to the $Z_{\text{plan}}^{(i)} \subset X \times K$.

To complete the proof, suppose that $Z_{\text{plan}}^{(i-1)}$ fulfills the claim. Notice that $Z_{\text{plan}}^{(i)}$ is created by applying (6.15) to $Z_{\text{plan}}^{(i-1)}$, where the same logic holds for the linearized dynamics, linearization error, and continuous time that proved the claim for $Z_{\text{plan}}^{(1)}$. As with $Z_{\text{plan}}^{(1)}$, the addition of X_0 and $Z_{\text{err}}^{(i,j)}$ does not introduce any generators with nonzero elements in the k dimensions. $\square$

In other words, each FRS zonotope has exactly one $\langle \kappa_{k_n}^{(n)} \rangle$-sliceable generator for each $n = 1, \cdots, n_K$. To ease notation, and to emphasize their utility, we call these generators k-sliceable.

Now we confirm that the zonotope FRS is conservative after it is sliced, which follows nearly directly from the construction of the zonotope FRS:

Theorem 6.6. *Suppose that* $(t, x_{\text{hi},0}, x, k) \in \mathcal{R}_{\text{FRS}}$, $t \in I^{(i)}$, *and* $x_{\text{hi},0} \in X_{\text{hi},0}^{(j)}$. *Let* $Z_{\text{FRS}}^{(i,j)}$ *be as in Theorem 6.4. Denote* $k = (k_1, \cdots, k_{n_K}) \in K$, *and define the values*

$$\kappa^{(n)} = \frac{k_n - k_{0,n}}{\Delta_{k_n}} \in [-1, 1] \tag{6.25}$$

for each $n = 1, \cdots, n_K$, *where* $k_0 = (k_{0,1}, \cdots, k_{0,n_K})$ *denotes the center of* K *per* (6.13). *Then,*

$$(x, k) \in \mathsf{slice}\left(Z_{\text{FRS}}^{(i,j)}, \left\{ \langle \kappa_{k_n}^{(n)} \rangle \right\}_{n=1}^{n_K}, \{\kappa^{(n)}\}_{n=1}^{n_K} \right) =: Z_{\text{slice}}^{(i,j)}. \tag{6.26}$$

Proof. First notice that $\text{proj}_K(Z_{\text{slice}}^{(i,j)}) = \{k\}$ by the definition of the slice operator. Now we must show that $x \in \text{proj}_X(Z_{\text{slice}}^{(i,j)})$. Notice, again by definition of the slice operator and Definition 6.3 (addition of zonotopes of mismatched dimension), that we have

$$Z_{\text{slice}}^{(i,j)} = Z_{\text{plan,slice}}^{(i)} \oplus X_0 \oplus Z_{\text{err}}^{(i,j)}, \text{ where} \tag{6.27}$$

$$Z_{\text{plan,slice}}^{(i)} = \mathsf{slice}\left(Z_{\text{plan}}^{(i)}, \{\langle \kappa_{k_n}^{(n)} \rangle\}_{n=1}^{n_K}, \{\kappa^{(n)}\}_{n=1}^{n_K} \right). \tag{6.28}$$

Then, there exists $p \in Z_{\text{plan,slice}}^{(i)}$ (by [Alt10, Theorem 3.3. and Proposition 3.7]) and $e \in Z_{\text{err}}^{(j)}$ (by Assumption 6.2) such that $x = p + e$, completing the proof. $\square$

The takeaway from this section is that we can and slice the FRS to find the subset corresponding to a given trajectory, assuming the existence of the ERS as a set of zonotopes. Next, we use this type of slicing with zonotope intersection (Lemma 6.7) to identify unsafe plans online.

6.4 Online Planning

We now discuss how we use the zonotope FRS to generate obstacle avoidance constraints for runtime planning. For this section, suppose the robot is in a planning iteration with initial condition

$x_{\text{hi},0} \in X^{(j)}_{\text{hi},0}$, and that we have constructed the set of FRS zonotopes, denoted $Z^{(i,j)}_{\text{FRS}}$ for each time interval $I^{(i)}$, plus the corresponding error zonotopes $Z^{(i,j)}_{\text{err}}$. The goal of this section is to (conservatively) identify a set $K_{\text{unsf}} \subset K$ containing plans that could cause a collision. Recall the definition (3.37), which we restate here as

$$K_{\text{unsf}} \subseteq \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \mathcal{R}_{\text{obs}}). \tag{6.29}$$

Note, we have dropped the index i that we used earlier to denote the i^{th} planning iteration (see §3.8), to avoid confusion with our usage of i for the FRS time intervals $I^{(i)} \subset T_{\text{plan}}$.

6.4.1 Obstacle Representation

Before we identify the unsafe parameters, we require that the $\mathcal{R}_{\text{obs}}$ is represented as zonotopes. Suppose there are n_{obs} obstacles that are predicted in $\mathcal{R}_{\text{obs}}$. In particular, we assume that, for each $i = 1, \cdots, n_{\text{RS}}$, there exists a collection of zonotopes $\{Z^{(i,m)}_{\text{obs}}\}_{m=1}^{n_{\text{obs}}}$ such that

$$\text{proj}_{I^{(i)} \times X}(\mathcal{R}_{\text{obs}}) \subseteq \bigcup_{m=1}^{n_{\text{obs}}} Z^{(i,m)}_{\text{obs}}. \tag{6.30}$$

That is, $Z^{(i,m)}_{\text{obs}}$ contains all points reached by obstacle m for all $t \in I^{(i)}$. In other words, we assume the existence of a set of zonotopes that overapproximate the obstacle predictions for each time interval of the FRS.

6.4.2 Zonotope Intersection

Our goal for this section is to identify the subset of the FRS that intersects with obstacles (if such a subset exists), so that we can exclude it during trajectory optimization. To this end, we introduce the following lemma to check if two zonotopes intersect.

Lemma 6.7. *[GNZ03, Lemma 5.1] Let X and Y be as in* (6.3)*. Then X and Y intersect if the center of y is in the zonotope centered at x, with the generators/indeterminates of both X and Y:*

$$X \cap Y \neq \emptyset \iff y \in \left(x + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle g_X^{(i)} + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle g_Y^{(j)} \right). \tag{6.31}$$

Notice that this is equivalent to checking if

$$y \in X \oplus \left(0 + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle g_Y^{(j)} \right). \tag{6.32}$$

In other words, the Minkowski sum enables us to check if two zonotopes intersect, which is convenient because the zonotope Minkowski sum is straightforward to implement numerically per (6.4). Lemma 6.7 is illustrated in Figure 6.4.

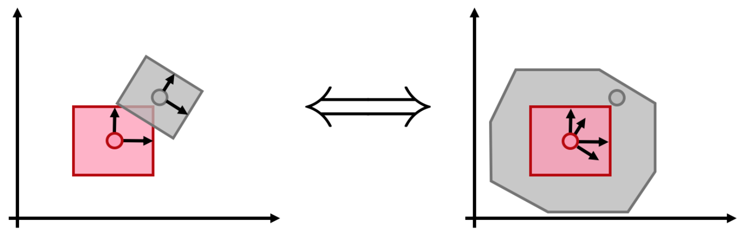


Figure 6.4: A visual proof of the intersection of zonotopes using the Minkowski sum. The grey and pink zonotopes intersect on the left (generators shown in black, and centers shown as points), meaning the center of the grey zonotope is inside the Minkowski sum of the pink zonotope with the generators of the grey zonotope.

6.4.3 Identifying Unsafe Plans

Now, we can we identify the unsafe trajectory parameters corresponding to just one time interval $I^{(i)}$. To simplify notation, we assume that $n_{\text{obs}} = 1$, so we need only consider a single obstacle zonotope $Z_{\text{obs}}^{(i)}$ for the i^{th} time interval. Note, we extend this single obstacle formulation to any finite $n_{\text{obs}} \in \mathbb{N}$ below in §6.4.5.

First, we use the notion of k-sliceable generators introduced above.

Corollary 6.8 (to Lemma 6.5)**.** *We can separate the generators of the FRS zonotope $Z_{\text{FRS}}^{(i,j)}$ into k-sliceable and non-k-sliceable generators:*

$$Z_{\text{FRS}}^{(i,j)} = c + \sum_{n=1}^{n_K} \langle \kappa_{k_n}^{(n)} \rangle g_{k_n}^{(n)} + \sum_{n=1}^{n_{\text{extra}}} \langle \beta^{(n)} \rangle g^{(n)}, \tag{6.33}$$

where $n_{\text{extra}} \in \mathbb{N}$.

Proof. This follows directly from Lemma 6.5. □

Importantly, per Theorem 6.6, no matter our choice of $k \in K$, the non-k-sliceable generators are left unchanged when $Z_{\text{FRS}}^{(i,j)}$ is sliced.

Second, we reorganize the centers and generators of the FRS and obstacle zonotopes. This lets us leverage the relationship between zonotope intersection and Minkowski sums in Lemma 6.7 to identify unsafe parameters.

Lemma 6.9. *Let $Z_{\text{FRS}}^{(i,j)}$ be as in* (6.33)*. Consider the zonotope*

$$Z_k^{(i,j)} = c + \sum_{n=1}^{n_K} \langle \kappa_{k_n}^{(n)} \rangle g_{k_n}^{(n)} \tag{6.34}$$

built from the center and k-sliceable generators of $Z_{\text{FRS}}^{(i,j)}$. Suppose that we buffer the obstacle by the non-k-sliceable generators of $Z_{\text{FRS}}^{(i,j)}$:

$$Z_{\text{buf}}^{(i,j)} = Z_{\text{obs}}^{(i)} \oplus \left(0 + \sum_{n=1}^{n_{\text{extra}}} \langle \beta^{(n)} \rangle g^{(n)} \right). \tag{6.35}$$

Then we have the following equivalence

$$Z_{\text{FRS}}^{(i,j)} \cap Z_{\text{obs}}^{(i)} = \emptyset \iff Z_k^{(i,j)} \cap Z_{\text{buf}}^{(i,j)} = \emptyset. \tag{6.36}$$

Proof. Note, we use the index (i,j) for $Z_{\text{buf}}^{(i,j)}$ because the extra generators are related to the time interval $I^{(i)}$ *and* to the error zonotopes corresponding to the j^{th} subset of $X_{\text{hi},0}^{(j)}$. Also note, the zonotope $Z_k^{(i,j)}$ can be written as in (6.34) by applying Corollary 6.8. The result then follows by applying the definition of zonotope intersection as in Lemma 6.7. □

Next, we check if a plan is unsafe by checking if a point lies inside a zonotope:

Theorem 6.10. *Let $Z_k^{(i,j)}$ be as in* (6.34)*. Consider $z_{\text{slice}}^{(i,j)} : K \to \mathbb{R}^{\dim(W)}$ given by*

$$z_{\text{slice}}^{(i,j)}(k) = \mathsf{slice}\left(Z_k^{(i,j)}, \{\langle \kappa_{k_n}^{(n)} \rangle\}_{n=1}^{n_K}, \{\kappa_{k_n}^{(n)}\}_{n=1}^{n_K} \right), \tag{6.37}$$

where we denote $k = (k_1, \cdots, k_{n_K})$, and $\kappa_{k_n}^{(n)} = \frac{k_n - k_{0,n}}{\Delta_{k_n}}$ for each $n = 1, \cdots, n_K$. Let $Z_{\text{buf}}^{(i,j)}$ be as in Lemma 6.9. We claim that $z_{\text{slice}}^{(i,j)}$ is affine in k, and that

$$k \in K_{\text{unsf}}^{(i)} \implies z_{\text{slice}}(k) \in Z_{\text{buf}}^{(i,j)}, \tag{6.38}$$

where $K_{\text{unsf}}^{(i)} \subset$ is the set of unsafe plans for time interval $I^{(i)} \subset T_{\text{plan}}$.

Proof. To see that $z_{\text{slice}}^{(i,j)}$ is affine in k, notice that, from the definition of slicing,

$$z_{\text{slice}}^{(i,j)}(k) = c + \sum_{n=1}^{n_K} \kappa_{k_n}^{(n)} g_{k_n}^{(n)} \in \mathbb{R}^{\dim(W)}. \tag{6.39}$$

where c is the center of $Z_k^{(i,j)}$ as in (6.34); recall that $g_{k_n}^{(n)}$ are constants with respect to k. Also note, the codomain of z_{slice} is $\mathbb{R}^{\dim(W)}$ because, if we slice $Z_k^{(i,j)}$ by any $k \in K$, we produce a point; this follows from Lemma 6.5, which defines k-sliceable generators of $Z_{\text{FRS}}^{(i,j)}$, and from (6.34). The desired result then follows from Lemma 6.7 and Lemma 6.9. □

The utility of Theorem 6.10 is that it lets us construct constraints on $k \in K$ that are practical for numerical trajectory optimization below.

6.4.4 Numerical Constraint Formulation

We now present a numerically tractable formulation for the unsafe parameters as identified by Theorem 6.10. To do so, we first require the following intermediate result, which provides a numerical method to check if a point lies inside a zonotope using a pair of arrays:

Lemma 6.11. *[Alt10, Theorem 2.1] Let $Z \subset \mathbb{R}^n$ be a zonotope with m linearly independent generators. Let $p = 2\binom{m}{n-1}$. Then this zonotope admits a halfspace representation defined by a matrix $A \in \mathbb{R}^{p\times n}$ and a vector $b \in \mathbb{R}^n$:*

$$\max(Ax - b) \leq 0 \iff x \in Z. \tag{6.40}$$

Now, are ready to construct constraints on K that represent safe trajectory parameters numerically:

Corollary 6.12 (to Theorem 6.10)**.** *Let $K_{\text{unsf}}^{(i)} \subset$ denote the set of unsafe plans for time interval $I^{(i)} \subset T_{\text{plan}}$. Suppose $Z_{\text{obs}}^{(i)}$ is the zonotope obstacle representation as above. Let $Z_{\text{buf}}^{(i,j)}$ be the buffered obstacle zonotope as in* (6.35)*, with halfspace representation $(A_{\text{buf}}^{(i,j)}, b_{\text{buf}}^{(i,j)})$. Finally, let $z_{\text{slice}}^{(i,j)} : K \to \mathbb{R}^{\dim(W)}$ be as in* (6.37)*. Then we identify the safe trajectory parameters as*

$$k \in K \setminus K_{\text{unsf}}^{(i)} \impliedby -\max\left(A_{\text{buf}}^{(i,j)} z_{\text{slice}}^{(i,j)}(k) - b_{\text{buf}}^{(i,j)}\right) < 0. \tag{6.41}$$

Proof. This follows from Theorem 6.10 and Lemma 6.11. □

Note that, since A_{buf} is a linear operator and $z_{\text{slice}}^{(i,j)}$ is affine in k, (6.41) lets us check if a plan k is unsafe by taking the maximum of an affine operation. Another way to think of this is that we are overapproximating the unsafe set $K_{\text{unsf}}^{(i)}$ with a polytope. So, at runtime, this constraint representation means that we must ensure that k lies outside a polytope. Critically, this type of constraint admits an analytic subgradient [Pol12, Theorem 5.4.5], which makes it practical for fast (but nonlinear) optimization.

6.4.5 Trajectory Optimization Formulation

To conclude this section, we rewrite the trajectory optimization program (3.38) (see §3.8) using the safety constraints produced from the zonotope FRS. For completeness' sake, we extend the above discussion to the multiple obstacle case, and bring back the receding-horizon planning iteration index.

Suppose that RTD is in the n^{th} receding-horizon planning iteration. Suppose we have a zonotope obstacle representation $\{Z_{\text{obs}}^{(i,m)}\}_{m=1}^{n_{\text{obs}}}$ as in (6.30), where $i = 1, \cdots, n_{\text{RS}}$ indexes the FRS time intervals, and $m = 1, \cdots, n_{\text{obs}}$ indexes the obstacle zonotopes. Suppose that $x_{\text{hi},0} \in X_{\text{hi},0}^{(j)}$ is used to construct the FRS zonotopes $Z_{\text{FRS}}^{(i,j)}$ for each $i = 1, \cdots, n_{\text{RS}}$. Let $A_{\text{buf}}^{(i,j,m)}$ and $b_{\text{buf}}^{(i,j,m)}$ be the halfspace representation of $Z_{\text{buf}}^{(i,j,m)}$ for time interval i and obstacle zonotope m. Then, RTD attempts to solve the following optimization program to find the plan $k^{(n)}$:

$$k^{(n)} = \underset{k \,\in\, K}{\text{argmin}} \quad \mathsf{cost}(k) \tag{6.42}$$

$$\text{s.t.} \qquad -\max\left(A_{\text{buf}}^{(i,j,m)} z_{\text{slice}}^{(i,j)}(k) - b_{\text{buf}}^{(i,j,m)}\right) < 0 \tag{6.43}$$

$$k^{(i)} \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}), \tag{6.44}$$

where (6.43) holds for all $i = 1, \cdots, n_{\text{RS}}$ and all $m = 1, \cdots, n_{\text{obs}}$.

6.5 Chapter Summary

The takeaway of this chapter is a method to compute a zonotope FRS for RTD, and a method to use it online at runtime.

6.5.1 Chapter Summary

In this chapter, we showed how to construct an augmented planning model to enable computing a zonotope FRS. We then introduced the concept of slicing, which lets us formulate collision avoidance constraints by identifying unsafe subsets of the zonotope FRS. Finally, we showed how to use this FRS representation for online planning, and noted that it produces continuous-time collision-avoidance constraints with analytic subgradients, which are suitable for fast, real-time nonlinear trajectory optimization.

6.5.2 What is Missing?

This presentation has only applied to rigid body robots so far, which is also the case with the sums-of-squares approach from §4. We address this by extending RTD to manipulators in §8.

This chapter has also required assumptions about tracking error represented as zonotopes. We address these assumptions next, in §7.

CHAPTER 7

Error Reachable Sets via Sampling

While RTD uses a simplified planning model to generate plans, it also seeks to compensate for the tracking error that arises due to the mismatch between the high-fidelity model of the robot and the planning model. In §4 and §6, we placed assumptions on the representation of tracking error. In this present chapter, we present a generic approach to computing these tracking error representations to fulfill these assumptions.

Recall that we represent tracking error theoretically (see §3.7.2) as an Error Reachable Set (ERS). In this chapter, we rely on a sampling-based approach to compute the ERS. This is due to the high-dimensional, nonlinear high-fidelity models typically used to describe a robot's equations of motion, which typically render SOS and zonotope reachability intractable. However, by leveraging our trajectory parameterization to identify where tracking error is maximized, we identify a discrete, finite subset of a robot's initial conditions and trajectory parameters that achieve maximum tracking error.

The sections of this chapter are as follows. (§7.1) First, we inspect the robot's dynamics to understand how to find worst-case tracking error. (§7.2) We then leverage this worst-case error to develop a generic sampling-based algorithm to approximate the ERS. (§7.3) Next, we show how this ERS sampling algorithm can be applied to the SOS polynomial FRS in §4 and to the zonotope FRS in §6. (§7.4) We conclude with a summary and brief discussion of what work is left to do for the ERS representation.

7.1 Maximizing Tracking Error

To begin this chapter, we identify the conditions under which tracking error is maximized. We use this to guide our sampling strategy to compute the ERS.

7.1.1 FRS Reminder

First, we remind the reader of the structure of the FRS, $\mathcal{R}_{\text{FRS}} \subset T_{\text{plan}} \times X_{\text{hi},0} \times W \times K$. Recall that, per (3.26), the FRS contains all trajectories of the closed-loop high-fidelity model when tracking any parameterized trajectory plan from any initial condition (in the planning frame). Therefore, for any plan $k \in K$, the corresponding tracking error is a function of the robot's initial condition at the beginning of any plan, and of the plan itself.

7.1.2 A Partition of the Initial Condition Set

It is typically intractable to represent the FRS directly as in (3.26), due to the high-dimension of the space $T_{\text{plan}} \times X_{\text{hi},0} \times W \times K$. In practice, we instead represent the FRS over *subsets* of $X_{\text{hi},0}^{(j)} \subset X_{\text{hi},0}$, where $\bigcup_j X_{\text{hi},0}^{(j)} = X_{\text{hi},0}$. Then, the FRS for each $X_{\text{hi},0}^{(j)}$ assumes the *worst case* tracking error holds for every trajectory starting from every initial condition in $X_{\text{hi},0}^{(j)}$. The reader may recall that this simplification of the FRS was introduced in §4.2, and was the rationale behind FRS swapping in §4.7 for the SOS polynomial approach to computing the FRS. Similarly, we use an ERS indexed by the initial condition sets $X_{\text{hi},0}^{(j)}$ for the zonotope FRS in §6.2.2.

Therefore, in this chapter, we seek to identify the worst case tracking error for a given subset of the entire initial condition set.

7.1.3 Forecasting A Sampling Strategy

Suppose that we have an initial condition set $X_{\text{hi},0}^{(j)} \subset X_{\text{hi},0}$. Recall that $X_{\text{hi},0} = \{x_{\text{hi}} \in X_{\text{hi}} \mid \text{proj}_X(x_{\text{hi}}) = x_0\}$; that is, $X_{\text{hi},0}$ is all initial conditions of the robot in its *planning frame*. So, we typically only need to partition the initial conditions in the robot's space of generalized velocities $\dot{Q}$, as opposed to partitioning the generalized coordinates Q, because $\text{proj}_Q(X_{\text{hi},0}) = x_0$ when $X = Q$ (which is the case for all robots considered in this work).

In other words, given a set of initial conditions $X_{\text{hi},0}^{(j)}$, we seek to sample the robot's initial (generalized) velocities and trajectory parameters to maximize tracking error. Then, we treat all possible tracking error in $X_{\text{hi},0}^{(j)}$ as though it is this maximized tracking error. This is a conservative approach, as one expects is necessary to make strong statements about safety; but, the conservatism has the potential to be mitigated by choosing a finer partition of $X_{\text{hi},0}$.

7.1.4 Where is Tracking Error Maximized?

So, we now seek to answer the question of where (in $X_{\text{hi},0}^{(j)}$) tracking error is maximized, meaning which samples should we choose? Again, the goal is to choose a finite number of sampled initial conditions and trajectory parameters that display worst-case tracking error.

To answer this question, we note that most robot actuators can be approximated with linear dynamics. To see why, note that most robots use torque (acceleration) control to drive actuators towards desired positions or speeds. Indeed, most actuators use PD or PID controllers to transform higher-level commands (such as the output of a tracking controller) into motion. For example, the Segway hardware [KVB$^+$20] and the Fetch robot [WFK$^+$16] both use PID control for their motors. This is useful because, even if a robot's equations of motion are nonlinear, the relationship between actuator velocity and acceleration is linear when the acceleration is a control input. Indeed, across a wide variety of robots, we find that this paradigm of low-level (i.e., actuator) linearity holds [KVB$^+$20, KHV19, VKL$^+$19, HKZ$^+$20].

Of course, we must address the fact that robot and actuator dynamics are not actually linear. Consider the Segway's high-fidelity model in Example 3.1, which treats the velocity and yaw rate as having linear dynamics with respect to the control inputs, but then saturates the yaw and longitudinal accelerations. But, in terms of tracking error, we expect behavior such as saturation to only *increase* the tracking error; we find that the same increase in tracking error holds for systems such as quadrotors with drag §9.6, or wheeled robots with tire forces §9.3.

Our strategy is then to consider the linear approximation of a robot to identify where tracking error *should* be maximized, but then compute the tracking error using the nonlinear, closed-loop high-fidelity model. Since we identify a finite subset of $X_{\text{hi},0}^{(j)} \times K$ as samples at which tracking error is maximized, we can then use standard numerical solvers to find trajectories of the high-fidelity model for each of these samples, and evaluate the tracking error directly at each sample.

In other words, we maximize tracking error by assuming linear actuators, but computing tracking error for the full nonlinear system. To that end, we state the following proposition for a 1-D linear system, which one can think of as an actuator:

Proposition 7.1. *Consider a 1-D single integrator linear system with input:*

$$\frac{d}{dt}\begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ u(t, x(t)) \end{bmatrix}, \tag{7.1}$$

with $x(t) \in \mathbb{R}$ representing position. Suppose that $x_{\text{des}} : T_{\text{plan}} \to \mathbb{R}$ is a once-differentiable desired trajectory, and

$$u(t, x(t)) = \kappa_{\text{p}} \cdot (x(t) - x_{\text{des}}(t)) + \kappa_{\text{d}} \cdot (\dot{x}(t) - \dot{x}_{\text{des}}(t)), \tag{7.2}$$

where κ_{p} and κ_{d} are control gains that can be chosen freely; i.e., this is a PD controller. Suppose that $x(0) = 0$ and $x_{\text{des}}(0) = 0$; i.e., the system has no tracking error initially. Further suppose that *the initial velocity is drawn from an interval:*

$$\dot{x}(0) \in [\dot{x}_{\text{min}}, \dot{x}_{\text{max}}] \subset \mathbb{R}. \tag{7.3}$$

Let $t \in T_{\text{plan}}$. Then the tracking error magnitude, $|x(t) - x_{\text{des}}(t)|$, is maximized when $\dot{x}(0) = \dot{x}_{\text{min}}$ or $\dot{x}(0) = \dot{x}_{\text{max}}$.

Proof. Consider the tracking error system

$$z(t) = \begin{bmatrix} z_1(t) \\ z_2(t) \end{bmatrix} = \begin{bmatrix} x(t) - x_{\text{des}}(t) \\ \dot{x}(t) - \dot{x}_{\text{des}}(t) \end{bmatrix}. \tag{7.4}$$

Plugging in u, we can rewrite this as

$$\dot{z}(t) = \begin{bmatrix} 0 & 1 \\ \kappa_{\text{p}} & \kappa_{\text{d}} \end{bmatrix} z(t) = Az, \tag{7.5}$$

which is an autonomous linear system with the solution

$$z(t) = e^{At} z(0) = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} z(0). \tag{7.6}$$

If we pick κ_{p} and κ_{d} such that $a_{12} \neq 0$, then

$$|z_1(t)| = |x(t) - x_{\text{des}}(t)| = |a_{12}(\dot{x}(0) - \dot{x}_{\text{des}}(0))|, \tag{7.7}$$

which is maximized when $\dot{x}(0) = \dot{x}_{\text{min}}$ or $\dot{x}(0) = \dot{x}_{\text{max}}$. □

In other words, Proposition 7.1 states exactly what we would expect. For a given desired trajectory and range of possible initial velocities, the worst-case tracking error is produced when the initial velocity is as far as possible from the desired trajectory's velocity. This tells us how to choose velocity samples to maximize tracking error. It also tells us how to choose desired trajectory samples to maximize tracking error:

Corollary 7.2. *(to Proposition 7.1). Assume the premises of Proposition 7.1, but that $\dot{x}(0) = \dot{x}_0 \in \mathbb{R}$ (i.e, we pick an initial condition). Suppose the $x_{\text{des}} : T_{\text{plan}} \to \mathbb{R}$ is drawn from a compact set of possible trajectories, $\mathcal{X}_{\text{des}}$, and that any $x_{\text{des}} \in \mathcal{X}_{\text{des}}$ is bounded, continuous, and at least once-differentiable. Then, if $t \in T_{\text{plan}}$, the tracking error $|x(t) - x_{\text{des}}(t)|$ is maximized when $x_{\text{des}}(t)$ is either maximized or minimized.*

Proof. First note that, since $\mathcal{X}_{\text{des}}$ is compact and each $x_{\text{des}} \in \mathcal{X}_{\text{des}}$ is bounded and once-differentiable,

$\max_{x_{\text{des}} \in \mathcal{X}_{\text{des}}} \dot{x}_{\text{des}}(t)$ exists. Similarly, $\min_{x_{\text{des}} \in \mathcal{X}_{\text{des}}} \dot{x}_{\text{des}}(t)$ exists. Then the desired result follows from setting up the error system as in the proof of Proposition 7.1 and inspecting (7.7). $\square$

Corollary 7.2 tells us that, for a given initial condition, the tracking error is maximized by commanding the largest allowable change in velocity, as we would expect.

So, we can now answer the question of where tracking error is maximized. Given a set of initial velocities and trajectory parameters, tracking error is maximized when the initial velocity is maximized or minimized, and when the trajectory parameter commands the largest possible change in velocity. We use this to guide a sampling strategy to find the worst-case tracking error next.

7.2 Sampling to Compute the ERS

We now use the reationale developed in the previous section to estimate the ERS via sampling. We begin by reviewing relevant notation. Then, we present our sampling procedure in four steps. Finally, we summarize this procedure in Algorithm 3 as a general method to estimate the ERS. We discuss particular representations of the ERS in the next section, §7.3.

7.2.1 Notation Review

Before proceeding, we review the notation of robot's coordinates. Recall that Q is the generalized coordinate space (i.e. the configuration space), and $\dot{Q}$ is the generalized velocity space; and $X_{\text{hi}} = Q \times \dot{Q}$ is the state space of the high-fidelity model. Recall also that $X_{\text{hi},0} = \{x_{\text{hi},0}\} \times \dot{Q} \subset X_{\text{hi}}$ be the space of possible initial conditions, with $\text{proj}_X(x_{\text{hi},0}) = x_0$; in other words, $X_{\text{hi},0}$ contains all generalized velocities in the planning frame.

We assume that the robot's high-fidelity model is configuration-invariant (i.e., f_{hi} does not depend on $q \in Q$, so that we need not sample in the generalized coordinates. Note, this is not necessarily always true, such as for a drone experiencing ground effect, which we address in the implementation in §9.6 by adding an additional initial condition dimension for sampling the tracking error.

7.2.2 Partition of the Generalized Velocity Space

To begin, we partition the generalized velocity space. Note that, as we did earlier with our partition of time in §4.6 and §6, we slightly abuse the word "partition" to mean breaking a set into subsets for which the intersection of any two subsets is not necessarily empty, but is of measure 0 in the Lebesgue sense.

Recall that the robot has maximum and minimum generalized velocities, $\dot{q}_{\text{min}}$ and $\dot{q}_{\text{max}} \in \dot{Q}$; suppose that these are defined coordinatewise. Then, in each i^{th} coordinate of $\dot{Q}$, the robot's initial velocity can be drawn from an interval

$$\dot{Q}_i = [\dot{q}_{\text{min},i}, \dot{q}_{\text{max},i}]. \tag{7.8}$$

In other words, we are assuming that $\dot{Q}$ can be treated as an $n_{\dot{Q}}$-dimensional interval

$$\dot{Q} = \dot{Q}_1 \times \dot{Q}_2 \times \cdots \times \dot{Q}_{n_{\dot{Q}}}. \tag{7.9}$$

Now, suppose we partition $\dot{Q}$ into $n_{\text{part}} \in \mathbb{N}$ subsets, so

$$\dot{Q} = \bigcup_{j=1}^{n_{\text{part}}} \dot{Q}^{(j)}, \tag{7.10}$$

where each of these subsets is again an $n_{\dot{Q}}$-dimensional interval

$$\dot{Q}^{(j)} = \dot{Q}_1^{(j)} \times \dot{Q}_2^{(j)} \times \cdots \times \dot{Q}_{n_{\dot{Q}}}^{(j)}. \tag{7.11}$$

That is, $\dot{Q}_i^{(j)} \subseteq \dot{Q}_i$ for each $i = 1 \cdots, n_{\dot{Q}}$. We collect this partition of $\dot{Q}$ in the set

$$\dot{\mathcal{Q}}_{\text{part}} = \{\dot{Q}^{(j)} \mid j = 1, \cdots, n_{\text{part}}\}. \tag{7.12}$$

7.2.3 Sampling Generalized Velocities

Let $\dot{Q}^{(j)} \in \dot{\mathcal{Q}}_{\text{part}}$. Notice that $\dot{Q}^{(j)}$ is an $n_{\dot{Q}}$-dimensional box, meaning that it has $2^{n_{\dot{Q}}}$ extreme points, or "corners." That is, for any such extreme point $\dot{q} \in \dot{Q}^{(j)}$,

$$\text{proj}_{\dot{Q}_i}(\dot{q}) \in \{\dot{q}_{\text{min},i}^{(j)}, \dot{q}_{\text{max},i}^{(j)}\}, \tag{7.13}$$

where $\dot{Q}_i^{(j)} = [\dot{q}_{\text{min},i}^{(j)}, \dot{q}_{\text{max},i}^{(j)}]$ is the interval comprising $\dot{Q}^{(j)}$ in its i^{th} coordinate. We define $\mathsf{getVelocitySamples} : \dot{\mathcal{Q}}_{\text{part}} \to \text{pow}\left(\dot{Q}\right)$ to extract these extreme points:

$$\{\dot{q}_{\text{smpl}}^{(j,n)}\}_{n=1}^{2^{n_{\dot{Q}}}} = \mathsf{getVelocitySamples}(\dot{Q}^{(j)}). \tag{7.14}$$

These are **generalized velocity samples**.

7.2.4 Sampling Trajectory Parameters

Now we sample worst-case feasible trajectory parameters for each generalized velocity sample.

First, we assume the following about the structure of the parameter space. Recall that, given an initial condition $x_{\text{hi},0} \in X_{\text{hi},0}$, we have a feasible set of possible plans that we can choose, given by $\mathcal{K}_{\text{lim}}(x_{\text{hi},0})$ as in §3.6.3. To approximate the ERS, we assume that we can write

$$\mathcal{K}_{\text{lim}}(x_{\text{hi},0}) = [k_{\text{min},1}, k_{\text{max},1}] \times [k_{\text{min},2}, k_{\text{max},2}] \times \cdots \times [k_{\text{min},n_K}, k_{\text{max},n_K}] \subseteq K. \tag{7.15}$$

That is, we assume $\mathcal{K}_{\text{lim}} : X_{\text{hi}} \to \text{pow}\,(K)$ returns an n_K-dimensional interval. If this is not the case, then we assume that $\mathcal{K}_{\text{lim}}(x_{\text{hi},0})$ can be *overapproximated* with such an interval, which is reasonable since K is compact. We overapproximate the set because we care about identifying worst-case tracking error. For example, in the case of a quadrotor drone with a bounded maximum acceleration in any direction, $\mathcal{K}_{\text{lim}}(x_{\text{hi},0})$ may return a closed 2-norm ball in K of possible commanded accelerations, which we can then overapproximate with a multidimensional interval (i.e., a closed ∞-norm ball).

We also assume without loss of generality that the trajectory parameters that cause worst-case tracking error are drawn from the endpoints of these intervals. This is reasonable because the trajectory parameters usually specify commanded velocities or accelerations, and we know from Corollary 7.2 that we can maximize tracking error by maximizing our commanded change in velocity. The parameters in all of our implementations in §9 fulfill this assumption.

So, our strategy is, for each $\dot{q}_{\text{smpl}}^{(j,n)}$, we again choose the extreme points, or "corners," of the interval

$$\{k_{\text{smpl}}^{(j,n,m)}\}_{m=1}^{2^{n_K}} = \mathcal{K}_{\text{lim}}((x_0, \dot{q}_{\text{smpl}}^{(j,n)})) \tag{7.16}$$
$$= K_{\text{smpl}}^{(j,n)}, \tag{7.17}$$

where $(x_0, \dot{q}_{\text{smpl}}^{(j,n)}) \in X_{\text{hi},0}$ (under the assumption that $X = Q$ and $X_{\text{hi}} = Q \times \dot{Q}$); recall that $x_0 \in X$ is the initial condition for every plan in the planning frame. That is, we choose samples $k_{\text{smpl}}^{(j,n,m)}$ for which either

$$\text{proj}_{K_i}(k_{\text{smpl}}^{(j,n,m)}) = \max(\text{proj}_{K_i}(K_{\text{smpl}}^{(j,n)})), \quad \text{or} \tag{7.18}$$
$$\text{proj}_{K_i}(k_{\text{smpl}}^{(j,n,m)}) = \min(\text{proj}_{K_i}(K_{\text{smpl}}^{(j,n)})), \tag{7.19}$$

and notice that $m = 1, \cdots, 2^{n_K}$ because we sample both the upper and lower extrema of each i^{th} interval $[k_{\text{min},i}^{(j,n)}, k_{\text{max},i}^{(j,n)}]$ that comprises $K_{\text{smpl}}^{(j,n)}$ (where $i = 1, \cdots, n_K$).

We define $\mathsf{getTrajParamSamples} : \text{pow}\,(K) \to \text{pow}\,(K)$ to extract these extreme points of the multidimensional interval $K^{(m)}_{\text{smpl}}$, meaning that

$$\{k^{(j,n,m)}_{\text{smpl}}\}_{m=1}^{2^{n_K}} = \mathsf{getTrajParamSamples}(K^{(j,n)}_{\text{smpl}}). \tag{7.20}$$

With this strategy, the total number of samples is

$$n_{\text{smpl}} = n_{\text{part}} \times (2^{n_{\dot{Q}}}) \times (2^{n_K}), \tag{7.21}$$

which may be large. Typically, $n_{\dot{Q}}$ and $n_K = 2$ or 3, and $n_{\text{part}} \approx 10$, resulting in hundreds of thousands of samples. However, recall that we estimate the ERS offline; so, we can sample offline, and in parallel. In practice, we find that sampling to compute the ERS typically takes on the order of minutes for wheeled robots in the plane, and on the order of an hour for a quadrotor drone.

7.2.5 Computing the Tracking Error for Each Sample

Next, we compute the tracking error for each sample. Recall that, in (3.23), we defined the tracking error as a trajectory $x_{\text{err}} : T^{(i)} \to \mathbb{R}^{n_{\text{hi}}}$ in the i^{th} receding-horizon planning iteration, for which

$$x_{\text{err}}(t; x^{(i)}_{\text{hi},0}, k) = x_{\text{hi}}(t; k) - \mathsf{liftplan}(i, x(t - t^{(i)}; k)), \tag{7.22}$$

where $x_{\text{hi}} : T^{(i)} \to X_{\text{hi}}$ is the trajectory of the closed-loop high-fidelity model (3.19), $x : T_{\text{plan}} \to X$ is the trajectory of the planning model, and $\mathsf{liftplan}$ extends the codomain of x to X_{hi} (see (3.15)).

Now notice that each sample is of the form $(\dot{q}^{(n,j)}_{\text{smpl}}, k^{(m)}_{\text{smpl}}) \in \dot{Q} \times K$. Therefore, we can generate a tracking error trajectory for each sample, which we denote

$$x^{(j,n,m)}_{\text{err}}(\,\cdot\,; x^{(j,n)}_{\text{hi},0}, k^{(j,n,m)}_{\text{smpl}}) : T_{\text{plan}} \to \mathbb{R}^{n_{\text{hi}}}, \tag{7.23}$$

with $x^{(j,n)}_{\text{hi},0} = (x_0, \dot{q}^{(j,n)}_{\text{smpl}}) \in X_{\text{hi},0}$. In practice, we estimate x_{err} numerically using, e.g., the MATLAB `ode45` solver. Note that, for such solvers, since T_{plan} is compact and x and x_{hi} are continuous and twice differentiable by construction, one can provably bound the numerical integration error at each $t \in T_{\text{plan}}$ [Zha20, Chapter 5].

7.2.6 Storing the Worst-Case Tracking Error

Finally, for each subset of our partition of $\dot{Q}$, we store the maximum and minimum (i.e. worst-case) tracking error achieved in planning space. That is, we store the trajectories as data points

$e^{(j,t)}_{\text{max}}$ and $e^{(j,t)}_{\text{min}} \in \mathbb{R}^{n_X}$ for which

$$e^{(j,t)}_{\text{max}} = \text{elmax}_{n,m}\{\text{proj}_X(x^{(j,n,m)}_{\text{err}}(t))\} \text{ and} \tag{7.24}$$

$$e^{(j,t)}_{\text{min}} = \text{elmin}_{n,m}\{\text{proj}_X(x^{(j,n,m)}_{\text{err}}(t))\}, \tag{7.25}$$

where elmax and elmin take the max/min elementwise, $n = 1, \cdots, 2^{n_{\dot{Q}}}$, and $m = 1, \cdots, 2^{n_K}$. Again, we estimate $e^{(j,t)}_{\text{max}}$ and $e^{(j,t)}_{\text{min}}$ numerically, and typically represent the tracking error data at a finite, discrete set of times in T_{plan} as would be output by a numerical ODE solver (recall that we represent the robot with the ODE high-fidelity model f_{hi} and planning model f).

7.2.7 The ERS Estimation Algorithm

We summarize the above sampling steps here in Algorithm 3. We note that this procedure is performed offline, and the outermost for loop is parallelizable. The output of this algorithm is a collection of worst-case tracking error trajectories, which we post-process in a manner specific to a given FRS representation in the following section.

7.3 ERS Representations

We now discuss how we post-process the tracking error data numerically for use with the sums-of-squares polynomial FRS representation in §4, and with the zonotope FRS representation in §6.

The data are as follows. Algorithm 3 produces a collection of worst-case tracking error trajectories that we can think of as the tuples

$$(e^{(j,t)}_{\text{max}}, e^{(j,t)}_{\text{min}}) \in \mathbb{R}^{n_X} \times \mathbb{R}^{n_X} \tag{7.26}$$

for each $j = 1, \cdots, n_{\text{part}}$ and $t \in T_{\text{plan}}$. Note, since we cannot store every $t \in T_{\text{plan}}$ in practice, we usually discretize T_{plan} and store the corresponding data.

7.3.1 ERS Representation for the Polynomial FRS

In §4.1, Assumption 4.1, we assume that the tracking error is represented with a model f_{err} : $T_{\text{plan}} \times K \to \mathbb{R}^{n_X}$ for which

$$\max_{x_{\text{hi},0} \in X_{\text{hi},0}} |\text{proj}_X(x_{\text{hi}}(t;k)) - x(t;k)| \leq \int_0^t f_{\text{err}}(\tau, k) d\tau \tag{7.27}$$

Algorithm 3 Error Reachable Set via Sampling

require generalized velocity partition $\dot{\mathcal{Q}}_{\text{part}}$

par for $j = 1, \cdots, n_{\text{part}}$

 $\{\dot{q}_{\text{smpl}}^{(j,n)}\}_{n=1}^{2^{n_{\dot{Q}}}} \leftarrow \mathsf{getVelocitySamples}(\dot{Q}^{(j)})$

 for $n = 1, \cdots, 2^{n_{\dot{Q}}}$

  $x_{\text{hi},0}^{(j,n)} \leftarrow (x_0,\ \dot{q}_{\text{smpl}}^{(j,n)})$ // create initial condition sample

  $K_{\text{smpl}}^{(j,n)} \leftarrow \mathcal{K}_{\text{lim}}(x_{\text{hi},0}^{(j,n)})$ // get trajectory parameter set for init. cond.

  $\{k_{\text{smpl}}^{(j,n,m)}\}_{m=1}^{2^{n_K}} \leftarrow \mathsf{getTrajParamSamples}(K_{\text{smpl}}^{(j,n)})$ // get trajectory parameter samples

  for $m = 1, \cdots, 2^{n_K}$ // for each trajectory parameter sample

   compute $x_{\text{err}}^{(j,n,m)} : T_{\text{plan}} \to \mathbb{R}^{n_{\text{hi}}}$ as in (7.23)

  end for

  // compute and store tracking error data at each $t \in T_{\text{plan}}$:

  $e_{\text{max}}^{(j,t)} \leftarrow \text{elmax}_{n,m}\{\text{proj}_X(x_{\text{err}}^{(j,n,m)}(t))\}$

  $e_{\text{min}}^{(j,t)} \leftarrow \text{elmin}_{n,m}\{\text{proj}_X(x_{\text{err}}^{(j,n,m)}(t))\}$

  store $e_{\text{max}}^{(j,t)}$ and $e_{\text{min}}^{(j,t)} \in \mathbb{R}^{n_X}$ associated with $\dot{Q}^{(j)} \subset Q$ and each $t \in T_{\text{plan}}$.

 end for

end par for

for all $t \in T_{\text{plan}}$ and $k \in K$, where and the absolute value is taken elementwise. Here, x_{hi} is the trajectory of the closed-loop high-fidelity model, and x is the trajectory of the planning model. We further assume f_{err} is Lipschitz continuous in t, x, and k.

To produce such a model, we first take the worst-case data absolute values:

$$e^{(j,t)} = \max\{|e_{\text{max}}^{(j,t)}|, |e_{\text{min}}^{(j,t)}\}, \tag{7.28}$$

again indexed by j and t. Then, for each j (i.e., each subset of the partition of $\dot{Q}$), we fit $f_{\text{err}}^{(j)} \in \mathbb{R}[t]$ as a polynomial, $f_{\text{err}} : T_{\text{plan}} \to \mathbb{R}$ for which

$$f_{\text{err}}^{(j)}(t) \geq e^{(j,t)} \tag{7.29}$$

using standard numerical polynomial fitting tools (e.g., `polyfit` in MATLAB). In other words, we estimate f_{err} for each subset of the initial condition space.

Notice that, in each $f_{\text{err}}^{(j)}$, we drop the dependence on k that is allowed in f_{err}. This is a conservative approach, in that we are assuming the same worst-case tracking error holds for every $k \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0})$ for any $x_{\text{hi},0} \in \dot{Q}^{(j)}$. However, we can mitigate this conservatism by choosing n_{part} (the fineness of the partition of $\dot{Q}$) to be larger, and by noticing that, in each $\dot{Q}^{(j)} \in \dot{\mathcal{Q}}_{\text{part}}$, we are still restricting k using $\mathcal{K}_{\text{lim}}$. In other words, we are not assuming all possible k can be chosen from any initial condition.

By partitioning $\dot{Q}$ and computing $f_{\text{err}}^{(j)}$ for each $\dot{Q}^{(j)} \in \dot{\mathcal{Q}}_{\text{part}}$, we require one to compute a polynomial FRS for *each* $j = 1, \cdots, n_{\text{part}}$. Then, at runtime, one should select the correct FRS for the robot's current initial condition. In other words, by estimating the ERS as we have done here, we enable FRS swapping as in §4.7.

7.3.2 ERS Representation for the Zonotope FRS

In §6.2.2, Assumption 6.2, we assume that the tracking error is represented with zonotopes

$$Z_{\text{err}}^{(i,j)} \subset \mathbb{R}^{\dim W}, \tag{7.30}$$

where $i = 1, \cdots, n_{\text{RS}}$ indexes the time intervals $I^{(i)} \subset T_{\text{plan}}$ over which the zonotope FRS is computed.

We can produce this representation directly from the data as in (7.26). For this discussion, consider time interval $I^{(i)} \subset T_{\text{plan}}$.

First, we get the worst-case tracking error in that interval, and project it into the workspace dimensions of X:

$$e_{\text{max},W}^{(i,j)} = \text{elmax}_{t \in I^{(i)}}\{\text{proj}_W(e_{\text{max}}^{(j,t)})\} e_{\text{min},W}^{(i,j)} \qquad = \text{elmin}_{t \in I^{(i)}}\{\text{proj}_W(e_{\text{min}}^{(j,t)})\}, \tag{7.31}$$

where we abuse the notation to let proj_W project the corresponding dimensions of the *tracking error*, which does not evolve in the planning space X (the usual domain of proj(W)), but does evolve in $\mathbb{R}^{n_X}$.

Notice that the index order is (i, j) corresponding to (time, initial condition) for the error zonotopes, whereas the order is reversed as (j, t) for the stored tracking error from Algorithm 3. We have deliberately left the indices in these formats to match the formats of their respective sections of this text.

Now, we can create a center as the mean worst-case error, and generators to span the worst-case error. That is, define a center

$$c_{\text{err}}^{(i,j)} = \frac{1}{2}\left(e_{\text{max},W}^{(i,j)} + e_{\text{min},W}^{(i,j)}\right) \in \mathbb{R}^{\dim(W)}, \tag{7.32}$$

and a matrix

$$G_{\text{err}}^{(i,j)} = \operatorname{diag}\left(\frac{1}{2}\left(e_{\max,W}^{(i,j)} + e_{\min,W}^{(i,j)}\right)\right) \in \mathbb{R}^{\dim(W)\times\dim(W)}. \tag{7.33}$$

Then the error zonotope is

$$Z_{\text{err}}^{(i,j)} = c_{\text{err}}^{(i,j)} + \sum_{n=1}^{\dim(W)} \langle\beta^{(n)}\rangle g_{\text{err}}^{(i,j,n)} \tag{7.34}$$

where $g_{\text{err}}^{(i,j,n)}$ is the n^{th} column of $G_{\text{err}}^{(i,j)}$. We produce these error zonotopes offline for each j^{th} subset of $\dot{Q}$, and for each time interval $I^{(i)} \subset T_{\text{plan}}$, and use them online as in §6.4.

7.4 Chapter Review

The takeaway of this chapter is a generic method of representing a robot's tracking error via sampling. Importantly, we leveraged the dynamic model of the robot to identify worst-case tracking error, enabling a conservative tracking error computation using only a discrete, finite subset of the robot's initial conditions and trajectory parameters.

7.4.1 Chapter Summary

We began this chapter by identifying initial conditions and trajectory parameters that lead to worst-case tracking error. We then developed a sampling-based algorithm, using this rationale, to store the worst-case tracking error in a manner amenable to FRS computation with either the sums-of-squares approach in §4 or the zonotope approach in §6.

7.4.2 What is Missing?

While we use a physics-guided sampling approach to estimate worst-case scenario tracking error, we still have not proven that this is indeed the actual worst-case tracking error. However, we have found in practice that this approach is sufficiently conservative to enable safe planning. The reason for this is, when we sample to generate the worst-case tracking error, we are forcing the robot to make large changes in its commanded velocity or acceleration. However, when we run the robot online, we almost never command such large changes in practice. Therefore, as we report in §9, we have seen no crashes in any of thousands of simulations or dozens of hardware demos for wheeled robots, drones, or manipulators.

Furthermore, the approach detailed in this chapter is relegated to offline computation, assuming that the robot's high-fidelity model is correct. Recall that we do not consider modeling error in this work, per §3.1. However, going forward, it is critical to incorporate modeling error into this tracking error computation. Furthermore, it is important to be able to update the robot's ERS online if the robot learns its own model as it operates, such as by using adaptive or learning-based control [AGST13, HWMZ20].

CHAPTER 8

Forward Reachable Set via Rotatotopes

In §6, we introduced the zonotope FRS. This reachable set formulation assumed that robot is a single rigid body, so it can be used for wheeled and aerial robots. Unfortunately, such an approach is not tractable for multi-link redundant manipulators. To address this, we now introduce a more general class of zonotope-like objects call rotatotopes, which are parameterized swept volumes that enable representing an FRS for a manipulator. These objects are constructed by first computing a zonotope reachable set in the space of a manipulator's joint angles (i.e., its configuration space), then multiplying these zonotopes by each other and by the link volume.

The sections of this chapter are as follows. (§8.1) We begin by introducing notation and assumptions used to apply RTD to manipulators. (§8.2) We then present our strategy for manipulator RTD at a high level. (§8.3) Next, we define rotatotopes. (§8.4) We then create an FRS for the manipulator using rotatotopes. (§8.5) To enable using the FRS at runtime, we revisit the concept of slicing from §6.3. (§8.6) Finally, we use the rotatotope FRS is for online trajectory optimization.

8.1 Manipulator Notation and Assumptions

To apply RTD to manipulators, we use the following notation and assumptions. As we have done with the other robot morphologies, we specify $X = Q$, and $X_{\text{hi}} = Q \times \dot{Q}$. Therefore, we consider a manipulator with $n_X \in \mathbb{N}$ DOFs and $n_X + 1$ links, including a 0^{th} link, or baselink.

8.1.1 Kinematics

The manipulator kinematics are as follows. We assume that the baselink is stationary, and leave mobile manipulators to future work. We assume that the manipulator has only (single-axis) revolute joints, so its configuration space $Q = \mathbb{S}^{n_Q}$ where n_Q is the number of degrees-of-freedom (DOFs). We further assume the manipulator is a single kinematic chain, wherein joint j connects the (predecessor) link $j - 1$ to its (successor) link j. Note, one can create multi-DOF joints by using virtual links with zero length.

We denote the elements in the kinematic chain as follows. Each link j has a local coordinate frame with its origin located at joint j. The rotation matrix $R_j(x_j) \in \mathsf{SO}(3)$ describes the rotation of link j relative to link $j-1$ (by joint j). The displacement $l_j \in \mathbb{R}^3$ denotes the position of joint $j+1$ on link j, in the frame of link j. The volume $L_j \subset \mathbb{R}^3$ denotes the volume occupied by link j in its own coordinate frame. So, if $x \in X$, then we can write the **forward occupancy for link** j as $\mathsf{FO}_j : X \to \mathrm{pow}\,(W)$ for which:

$$\mathsf{FO}_j(x) = \left\{ \sum_{n<j} \left(\prod_{m \leq n} R_m(x_m) \right) l_n \right\} \oplus \left\{ \left(\prod_{n \leq j} R_n(x_n) \right) L_j \right\}, \tag{8.1}$$

where x_n (resp. x_m) denotes the n^{th} (resp. m^{th}) coordinate of x, and l_n is the displacement for joint n. The curly brackets are used to emphasize that both sides of $\oplus$ are sets, for the purpose of set (Minkowski) addition. For the rotation matrices here, $\prod$ denotes *right multiplication* with increasing index:

$$\prod_{i=1}^{n} R_i = R_1 R_2 \cdots R_n. \tag{8.2}$$

It follows that the forward occupancy of the robot (as introduced in §3.4) can be written

$$\mathsf{FO}(x) = \bigcup_{j=1}^{n_X} \mathsf{FO}_j(x) \subset W. \tag{8.3}$$

8.1.2 Dynamics

We treat manipulator dynamics as follows. We assume that the manipulator has a motor at each joint, with sufficient torque to track kinematic trajectories that are prescribed separately for each joint; we find that this is true in practice for the Fetch robot [WFK$^+$16] to which we have applied RTD [HKZ$^+$20]. We further assume that there is no tracking error, and only consider the development of the PRS for the remainder of this section. Again, in practice, the Fetch hardware had tracking error of consistently less than 0.01 rad per joint on the trajectories we tested.

We leave alternative joint types (e.g., unactuated and prismatic), dynamic forces on each link (e.g., Coriolis forces), and tracking error (using an approach such as [GA17]) for future work.

8.2 Manipulator RTD Overview

We now provide a high-level overview of RTD for manipulators.

8.2.1 Offline Reachability Analysis

We begin by computing a Joint Reachable Set (JRS) separately for each joint. The JRS is a zonotope FRS as has been presented earlier in this chapter, but in the space of *sines and cosines* of each joint angle; that is, we observe how trajectories of the planning model evolve along the unit circle $\mathbb{S}^1$. Informally, this choice makes the planning model *less* nonlinear by treating the sines and cosines as states themselves. This leads to a less conservative reachable set, *and* avoids the challenge of taking the sine or cosine of a zonotope, which would be necessary to generate an FRS in the workspace if we computed the JRS in the configuration space directly.

8.2.2 Online Planning

We construct the rotatotope FRS at runtime to account for the robot's initial conditions. Given the JRS, we reshape the zonotopes into matrix zonotopes (defined below), then *multiply* these matrix zonotopes with regular zonotopes in W that represent the link volume. This operator produces the objects we call rotatotopes. Since the matrix zonotopes represent parameterized trajectories in $\mathsf{SO}(3)$, when we multiply them by zonotopes in W, we produce parameterized swept volumes corresponding to the arm's motion in workspace when tracking any (parameterized) trajectory. In other words, the rotatotopes represent the FRS. As with SOS and zonotope RTD, we use the FRS to identify unsafe trajectory parameters, which we represent as constraints for trajectory optimization.

8.3 Rotatotopes

We now formally introduce rotatotopes. First, we introduce matrix zonotopes. Second, revisit the indeterminate coefficients used to define zonotopes. Third and finally, we define rotatotopes.

8.3.1 Matrix Zonotopes

We now introducte matrix zonotopes. Recall that zonotopes are sets in $\mathbb{R}^n$; since one can think of matrices as points in Euclidean space, the definition of a **matrix zonotope** is the same as the definition of a zonotope:

$$M = \left\{ Y \in \mathbb{R}^{n\times n} \ \middle|\ Y = C + \sum_{i=1}^{m} \beta^{(i)} G^{(i)},\ -1 \le \beta^{(i)} \le 1 \right\}, \tag{8.4}$$

with the **center** C and each **generator** $G^{(i)} \in \mathbb{R}^{n\times n}$, and coefficients $\beta^{(i)} \in [-1, 1]$. Note, the elements in a matrix zonotope need not be square, but we only make use of square matrix zonotopes in this work (since we use matrix zonotopes to represent rotation matrices). As with zonotopes, we use shorthand notation for matrix zonotopes to emphasize the center, generator, and indeterminate coefficients:

$$M = C + \sum_{i=1}^{m} \langle \beta^{(i)} \rangle G^{(i)}. \tag{8.5}$$

8.3.2 Indeterminate Products

Recall that a zonotope can be written $Z = c + \sum \langle \beta^{(i)} \rangle g^{(i)}$, with indeterminates $\langle \beta^{(i)} \rangle$ per (6.2). We now define indeterminate **products**, which we need for constructing rotatotopes, because multiplying matrix zonotopes requires us to multiply their indeterminates as well. Consider two indeterminates $\langle \alpha \rangle$ and $\langle \beta \rangle$. We denote their product as

$$\langle \alpha \rangle \langle \beta \rangle = \langle \alpha\beta \rangle. \tag{8.6}$$

Bringing both α and β inside $\langle \cdot \rangle$ emphasizes that the resulting object behaves as a single indeterminate coefficient with two variables.

To show this more clearly, we define the **evaluation** of a product of indeterminates, analogous to how we defined indeterminate evaluation earlier in (6.21). If $\alpha \in [-1, 1]$, then the evaluation of $\langle \alpha\beta \rangle$ is denoted

$$\alpha \langle \beta \rangle. \tag{8.7}$$

Notice that $\langle \alpha\beta \rangle$ always produces a value in $[-1, 1]$ when both indeterminates are evaluated. Suppose that $\langle \alpha\beta \rangle$ corresponds to a generator g (such indeterminate products paired with generators appear shortly, when we define rotatotopes). Then evaluating α results in a generator αg with corresponding indeterminate $\langle \beta \rangle$; that is, since the value α is a scalar, it commutes with $\langle \beta \rangle$ and g.

Notice that an indeterminate product is a monomial of indeterminates. We define the **degree** of an indeterminate product as number of unique indeterminate variables it contains. For example, $\deg\langle \alpha\beta \rangle = 2$, and $\deg\langle \gamma_1 \gamma_2 \cdots \gamma_n \rangle = n$. The degree allows us to differentiate zonotopes from rotatotopes.

8.3.3 Rotatotopes

To define rotatotopes, first, we define products of matrix zonotopes. Then we produce rotatotopes by multiplying matrix zonotopes with regular zonotopes.

We now define the **matrix zonotope product** (assuming square matrix zonotopes). Define two example matrix zonotopes in $\mathbb{R}^{n\times n}$:

$$X = C_X + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle G_X^{(i)} \quad \text{and} \quad Y = C_Y + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle G_Y^{(j)}. \tag{8.8}$$

That is, C_X, $G_X^{(i)}$, C_Y, and $G_Y^{(j)} \in \mathbb{R}^{n\times n}$ Then one can multiply X and Y:

$$XY = \left(C_X + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle G_X^{(i)} \right) \left(C_Y + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle G_Y^{(j)} \right) \tag{8.9}$$

$$\begin{aligned} &= C_X C_Y + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle C_X G_Y^{(j)} + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle G_X^{(i)} C_Y + \\ &\quad + \sum_{i=1}^{r} \sum_{j=1}^{s} \langle \chi^{(i)} \upsilon^{(j)} \rangle G_X^{(i)} G_Y^{(j)}. \end{aligned} \tag{8.10}$$

Now let $C_{XY} = C_X C_Y$ and $G^{(i,j)}$ denote all $C_X G_Y^{(j)}$, $G_X^{(i)} C_Y$, and $G_X^{(i)} G_Y^{(j)}$ terms. Notice that XY is no longer a matrix zonotope. However, it can be overapproximated by a matrix zonotope:

Lemma 8.1. *Let X and Y be as in* (8.8)*. Let $XY \subset \mathbb{R}^{n\times n}$ be as in* (8.9)*. Define the matrix zonotope*

$$M = C_X CY + \sum_{j=1}^{s} \langle \upsilon^{(j)} \rangle C_X G_Y^{(j)} + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle G_X^{(i)} C_Y + \sum_{i=1}^{r} \sum_{j=1}^{s} \langle \lambda^{(i,j)} \rangle G_X^{(i)} G_Y^{(j)}, \tag{8.11}$$

where $\deg \langle \lambda^{(i,j)} \rangle = 1$ *for all* $i = 1, \cdots, r$ *and* $j = 1, \cdots, s$*. Then*

$$A \in XY \implies A \in M. \tag{8.12}$$

Proof. For any values $\chi^{(i)}$ and $\upsilon^{(j)}$ of the indeterminates of X and Y, one can choose $\lambda^{(i,j)} = \chi^{(i)} \upsilon^{(j)} \in [-1, 1]$. □

Now, we define rotatotopes by multiplying one or more matrix zonotopes with a regular zonotope.

Definition 8.2. *Consider a collection of $n \in \mathbb{N}$ matrix zonotopes, $\{M^{(i)} \in \mathbb{R}^{n\times n}\}_{i=1}^{n}$. Let $Z \subset \mathbb{R}^n$ be a zonotope. Then*

$$V = \left(\prod_{i-1}^{n} M^{(i)} \right) Z \subset \mathbb{R}^n \tag{8.13}$$

*is a **rotatotope**. Note, $\prod$ denotes right multiplication with increasing index.*

Rotatotopes are in fact a specific type of polynomial zonotopes [Alt13].

We now provide an example of Definition 8.2. Consider the example zonotope $Z = c + \sum_{j=1}^{m} \langle \beta^{(j)} \rangle g^{(j)}$, with c and $g^{(j)} \in \mathbb{R}^n$. Taking X as in (8.8), we have

$$XZ = \left(C_X + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle G_X^{(i)} \right) \left(c + \sum_{j=1}^{m} \langle \beta^{(j)} \rangle g^{(j)} \right) \tag{8.14}$$

$$= C_X c + \sum_{i=1}^{r} \langle \chi^{(i)} \rangle G_X^{(i)} c + \sum_{j=1}^{m} \langle \beta^{(j)} \rangle C_X g^{(j)} + \sum_{i=1}^{r} \sum_{j=1}^{m} \langle \chi^{(i)} \beta^{(j)} \rangle G_X^{(i)} g^{(j)}. \tag{8.15}$$

So, $XZ \subset \mathbb{R}^n$ is a rotatotope. Similar to how we often use Z for zonotopes and M for matrix zonotopes, we typically use V to denote rotatotopes, since we use them to represent swept volume.

In general, we denote rotatotopes as

$$V = c_V + \sum_{i=1}^{p} \langle \gamma_1 \gamma_2 \cdots \gamma_n^{(i)} \rangle g_V^{(i)}. \tag{8.16}$$

As with matrix zonotopes and zonotopes, rotatotopes have a center, generators, and indeterminates. The notation $\langle \gamma_1 \gamma_2 \cdots \gamma_n^{(i)} \rangle$ indicates a product of n indeterminates, the entirety of which is indexed by i. Notice that this implies V is constructed by multiplying $n-1$ matrix zonotopes with each other and with one zonotope. While not all indeterminates γ_j necessarily appear in every $\langle \gamma_1 \gamma_2 \cdots \gamma_n^{(i)} \rangle$, we list them all in this manner as shorthand.

We can overapproximate rotatotopes with zonotopes:

Lemma 8.3. *Let $V = c_V + \sum_{i=1}^{p} \langle \gamma_1 \gamma_2 \cdots \gamma_n^{(i)} \rangle g_V^{(i)}$. Define the zonotope $Z = c_V + \sum_{i=1}^{p} \langle \beta^{(i)} \rangle g_V^{(i)}$. Then $V \subset Z$.*

Proof. Suppose $x \in V$. Then, for each $i = 1, \cdots, p$, there exists $\beta^{(i)} \in [-1, 1]$ such that $\beta^{(i)} = \gamma_1 \gamma_2 \cdots \gamma_n^{(i)}$, meaning $x \in Z$. □

The Minkowski sum of two rotatotopes is given as follows.

Lemma 8.4. *Consider two rotatotopes, V and $U \subset \mathbb{R}^n$, given by*

$$V = c_V + \sum_{i=1}^{r} \langle \gamma_1 \cdots \gamma_n^{(i)} \rangle g_V^{(i)} \quad \text{and} \quad U = c_U + \sum_{j=1}^{s} \langle \lambda_1 \cdots \lambda_m^{(j)} \rangle g_U^{(j)}. \tag{8.17}$$

Then their Minkowski sum is

$$V \oplus U = c_V + c_U + \sum_{i=1}^{r} \langle \gamma_1 \cdots \gamma_n^{(i)} \rangle g_V + \sum_{j=1}^{s} \langle \lambda_1 \cdots \lambda_m^{(j)} \rangle g_U^{(j)}. \tag{8.18}$$

Proof. This follows directly from the definition of a rotatotope. To see this, notice that any point in $V \oplus \{c_U\}$ can be produced on the right-hand side of (8.18) by choosing all $\lambda_i = 0$, and similarly any point in $U \oplus \{c_V\}$ can be produced by choosing all $\gamma_i = 0$. $\square$

8.4 Rotatotope FRS

Now we construct an FRS of the swept volume of a manipulator. First, we define the Joint Reachable Set (JRS) in terms of the sines and cosines of each joint, and represent it JRS with zonotopes. Second, we define the swept volume FRS of each link, then represent it with rotatotopes.

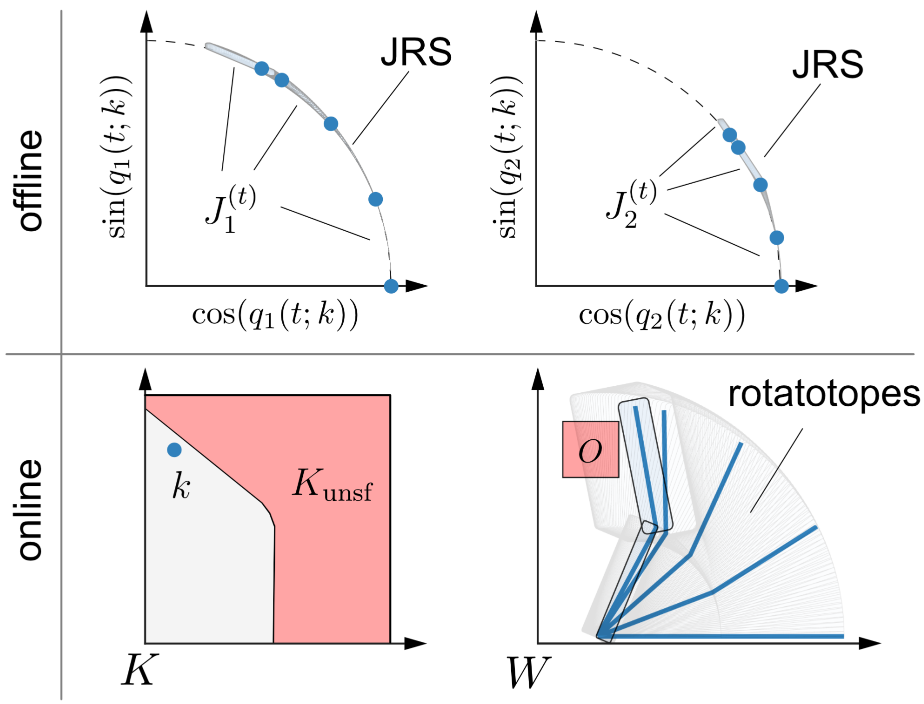


Figure 8.1: An overview of the proposed method for a 2-D, 2-link arm. Offline, RTD computes the JRSs, shown as the collection of small grey zonotoeps overlaid on the unit circle (dashed) in the sine and cosine spaces of two joint angles. Note that each JRS is conservatively approximated, and parameterized by trajectory parameters K. Online, the JRSs are composed to form the arm's reachable set, comprised of rotatotopes (large light grey sets in the workspace W), maintaining a parameterization by K. An obstacle O (light red) is mapped to the unsafe set of trajectory parameters $K_{\text{unsf}} \subset K$ on the left, by intersection with each rotatotope. The parameter k represents a trajectory, shown at five time steps (blue arms in W, and blue dots in joint angle space). The subset of the arm's reachable set corresponding to k is shown for the last time step (light blue boxes with black border), critically not intersecting the obstacle, which is guaranteed because $k \notin K_{\text{unsf}}$.

8.4.1 Offline JRS Computation

We now define the planning model for a single joint, augmented with the parameters for use with [Alt15]; then, we define the JRS as the planning reachable set of this model. Note that our planning model for manipulators defines joint angle trajectories, but we specify the model in terms of the sines and cosines of the joint angle for each j^{th} joint to enable the construction of (rotation) matrix zonotopes later on. Let $f_j : T_{\text{plan}} \times K \to R$ denote the planning model for the j^{th} joint. Then we write the augmented planning model as

$$\frac{d}{dt}\begin{bmatrix} \cos(x_j(t;k)) \\ \sin(x_j(t;k)) \\ k \end{bmatrix} = \begin{bmatrix} -\sin(x_j(t;k))f_j(t,k) \\ \cos(x_j(t;k))f_j(t,k) \\ 0 \end{bmatrix}. \tag{8.19}$$

Notice that $\cos(x_j)$ and $\sin(x_j)$ are treated as states. Therefore, we rewrite (8.19) as

$$\frac{d}{dt}\begin{bmatrix} c_j(t;k) \\ s_j(t;k) \\ k \end{bmatrix} = \begin{bmatrix} -s_j(t;k)f_j(t,k) \\ c_j(t;k)f_j(t,k) \\ 0 \end{bmatrix}. \tag{8.20}$$

With this planning model, we define the JRS of joint i as

$$\mathcal{R}_{\text{JRS},j} = \Big\{(t,c,s,k) \in \mathbb{R}^2 \times K \mid c = c_j(t;k),\ s = s_j(t;k), \tag{8.21}$$

$$\text{and } \tfrac{d}{dt}(c_j(t;k), s_j(t;k), k) \text{ is as in (8.20)}\Big\}. \tag{8.22}$$

In other words, $\mathcal{R}_{\text{JRS},j}$ contains the times, sines, and cosines of the j^{th} joint angle reached for each $k \in K$.

We represent the JRS with zonotopes as follows. Recall that the zonotope reachability tool [Alt15] requires three inputs: the parameter-augmented planning model above, a partition of time, and an initial condition set. We specify the initial condition set for this sine/cosine planning model as

$$J_j^{(0)} = z_0 + \sum_{n=1}^{n_K} \langle \kappa_{k_n}^{(n)} \rangle g_{k_n}, \tag{8.23}$$

where

$$z_0 = \begin{bmatrix} 1 \\ 0 \\ k_0 \end{bmatrix} \quad \text{and} \quad g_{k_n} = \begin{bmatrix} 0_{2\times 1} \\ \Delta_{k_n} e_{k_n} \end{bmatrix}, \tag{8.24}$$

just as we did for the regular zonotope PRS in (6.10). Notice that we assume all trajectories begin at $(c_j(0;k), s_j(0;k)) = (1,0)$; in other words, the point x_0 in the planning frame is $0 \in Q$. This is because we can rotate the JRS around the unit circle to compensate for different initial angles *at runtime*.

We use the same partition of time (6.8), with $n_{\text{RS}} \in N$ intervals, so

$$T_{\text{plan}} = \bigcup_{n=1}^{n_{\text{RS}}} I^{(n)} \tag{8.25}$$

where $I^{(n)} = [(n-1)\cdot\Delta_t, n\cdot\Delta_t]$ with $\Delta_t = t_{\text{f}}/n_{\text{RS}}$.

Finally, we produce a zonotope JRS for each j^{th} joint, denoted

$$\{J_j^{(n)}\}_{n=1}^{n_{\text{RS}}} \subset \mathbb{R}^2 \times K. \tag{8.26}$$

8.4.2 From Zonotopes to Matrix Zonotopes

Before we construct the FRS, we explain how to produce matrix zonotopes from the JRS zonotopes in (8.26). We illustrate this procedure with a specific example of a 3-axis rotation matrix. But, we note that this approach is generalizable by reshaping the JRS zonotopes into, e.g., canonical Euler rotation matrices [LaV06, Chapter 3].

We proceed in seven steps. First, we set up the example rotation matrix. Second, we confirm that the JRS zonotopes are conservative. Third, we inspect the form of the JRS zonotopes. Fourth, we reshapce the JRS zonotope into a matrix zonotope. Fifth, we confirm that the matrix zonotope is conservative, meaning any rotation matrix produced by joint j is contained inside that matrix zonotope. Sixth, we comment on the structure of the matrix zonotope. Sevent, we discuss how to account for the initial rotation of each joint.

First, we provide the example family of rotation matrices that we will overapproximate with a matrix zonotope. Recall that the rotation of link j produced by joint j is given by $R_j(x_j)$, where $x_j \in X$ is the angle of joint j. Suppose, for example, that joint j rotates link j about its local

3-axis; then, by [LaV06, (3.39)], we have the rotation matrix

$$R_j(x_j) = \begin{bmatrix} \cos(x_j) & -\sin(x_j) & 0 \\ \sin(x_j) & \cos(x_j) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{8.27}$$

Second, we note that the relevant values in the rotation matrix above are also in our zonotope JRS. That is, that any sine/cosine attained by the joint j for a plan k is contained in the zonotope JRS, by Lemma 6.1. In other wordsif $t \in I^{(n)} = [(n-1)\cdot\Delta_t, n\cdot\Delta_t]$ and $k \in K$, then,

$$(t, c, s, k) \in \mathcal{R}_{\text{JRS},j} \implies (c, s) \in J_j^{(n)}. \tag{8.28}$$

Third, we inspect the form of the JRS zonotopes representing $\mathcal{R}_{\text{JRS},j}$. Notice that, by Lemma 6.5, each zonotope is of the form

$$J_j^{(n)} = z_j^{(n)} + \sum_{i=1}^{n_K} \langle \kappa_{k_i}^{(i)} \rangle g_{k_i}^{(n,i)} + \sum_{i=1}^{n_{\text{extra}}^{(n)}} \langle \beta^{(i)} \rangle g_{\text{extra}}^{(n,i)} \tag{8.29}$$

where $n_{\text{extra}}^{(n)} \in \mathbb{N}$. To be clear, the index n is for the interval $I^{(n)} \subset T_{\text{plan}}$, j is for the joint, and i is for the sums of k-sliceable and extra (non-k-sliceable) generators. The extra generators are due to linearization error and continuous time, as per (6.15). The center $z_j^{(n)}$ and generators of $J_j^{(n)}$ can be written

$$z_j^{(n)} = \begin{bmatrix} \bar{c}_j^{(n)} \\ \bar{s}_j^{(n)} \\ k_0 \end{bmatrix}, \quad g_{k_j}^{(n,i)} = \begin{bmatrix} \Delta_{c_j}^{(n,i)} \\ \Delta_{s_j}^{(n,i)} \\ \Delta_{k_j} e_{k_j} \end{bmatrix}, \quad \text{and} \quad g_{\text{extra}}^{(i)} = \begin{bmatrix} c_{\text{extra}}^{(n,i)} \\ s_{\text{extra}}^{(n,i)} \\ 0_{n_K \times 1} \end{bmatrix}, \tag{8.30}$$

where $\bar{c}_j^{(n)}$, $\bar{s}_j^{(n)}$, $\Delta_{c_j}^{(n,i)}$, $\Delta_{s_j}^{(n,i)}$, $c_{\text{extra}}^{(n,i)}$, and $s_{\text{extra}}^{(n,i)}$ are all real numbers (found using [Alt15]) such that (8.28) holds.

Fourth, we reshape the JRS zonotope into a matrix zonotope. Now consider the matrix zonotope

$$M_j^{(n)} = \begin{bmatrix} \bar{c}_j^{(n)} & -\bar{s}_j^{(n)} & 0 \\ \bar{s}_j^{(n)} & \bar{c}_j^{(n)} & 0 \\ 0 & 0 & 1 \end{bmatrix} + \sum_{i=1}^{n_K} \langle \kappa_{k_i}^{(i)} \rangle \begin{bmatrix} \Delta_{c_j}^{(n,i)} & -\Delta_{s_j}^{(n,i)} & 0 \\ \Delta_{s_j}^{(n,i)} & \Delta_{c_j}^{(n,i)} & 0 \\ 0 & 0 & 0 \end{bmatrix} + \\ + \sum_{i=1}^{n_{\text{extra}}^{(n)}} \langle \beta^{(i)} \rangle \begin{bmatrix} c_{\text{extra}}^{(n,i)} & -s_{\text{extra}}^{(n,i)} & 0 \\ s_{\text{extra}}^{(n,i)} & c_{\text{extra}}^{(n,i)} & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{8.31}$$

Fifth, we confirm that this matrix zonotope is conservative. Indeed, it follows from (8.28) that, if $t \in I^{(n)}$ and $k \in K$, then

$$R_j(x_j(t;k)) \in M_j^{(n)}. \tag{8.32}$$

So, the matrix zonotope $M_j^{(n)}$ contains all rotation matrices produced by joint j. Therefore, if we multiply $M_j^{(n)}$ by a link volume (represented as a zonotope), we overapproximate the rotated swept volume of that link in its local coordinate frame during $I^{(n)}$. By adding the translation due to other links, we overapproximate the swept volume of the link for any $k \in K$.

Sixth, notice that constructing $M_j^{(n)}$ means we omitted k_0 and Δ_{k_j}. That is to say, we have $M_j^{(n)} \subset \mathbb{R}^{3\times 3}$, whereas $J_j^{(n)} \subset \mathbb{R}^2 \times K$. However, recall that k_0 and Δ_{k_j} are chosen by construction in (8.24); that is, these values define the space K, which we construct. Furthermore, these values are the same for all $n \in \{1, \cdots, n_{\text{RS}}\}$ by Lemma 6.5. Therefore, in practice, we store them separately from all $M_j^{(n)}$, and reference them as needed for numerical implementation. To see this more clearly, recall that, if $k \in K$, then we can pick

$$\kappa_{k_i} = \frac{k_i - k_{0,i}}{\Delta_{k_i}} \tag{8.33}$$

to evaluate the k-sliceable generator indeterminates.

Seventh and finally, we discuss how to account for the initial rotation of each joint. Recall that the JRS zonotopes all begin from $(c_i, s_i) = (1, 0)$, corresponding to an angle of 0 for joint j; but the actual joint angle is almost never 0 when operating the robot. Suppose that the joint is at an initial angle $x_j \neq 0$, and $M_j^{(n)}$ is the matrix zonotope for that joint. We rotate the matrix zonotope to account for the initial condition:

$$M_j^{(n)} \leftarrow R_j(x_j) M_j^{(n)}, \tag{8.34}$$

where $\leftarrow$ indicates that we are reassigning $M_j^{(n)}$ in the sense that it is being used as a variable at runtime. Note that this is equivalent to rotating the JRS zonotopes about the unit circle before reshaping them into matrix zonotopes.

Note that, while we presented this example for the case of a 3-axis rotation, using [LaV06, Chapter 3], one can construct matrix zonotopes with the strategy presented here, but for arbitrary joint rotation axis directions.

8.4.3 Online Rotatotope FRS Construction

We now use the JRS to produce rotatotopes representing the FRS of the entire manipulator for all of the parameterized trajectories. Importantly, because we have to account for the initial conditions of the robot as in (8.34), *the rotatotope FRS is constructed online*. As mentioned above, we multiply the matrix zonotopes by the link volume and joint displacements to product a collection of rotatotopes. The goal of this procedure is to overapproximate the forward occupancy map FO_j for each j^{th} link and any $k \in K$. For this discussion, suppose that the robot is at an initial condition $x_{\text{hi},0} \in X_{\text{hi}}$.

Before proceeding, we require that the link displacements and volumes can be represented with zonotopes as follows. First, recall that $l_j \in \mathbb{R}^3$ denotes the displacement of joint $j+1$ relative to joint j in the frame of link j. Notice that l_j is a zonotope (centered at l_j, with no generators). Second, recall that $L_j \subset \mathbb{R}^3$ is the volume occupied by link j in its own coordinate frame. We assume that L_j is a zonotope (or is overapproximated by a zonotope), which is always possible for compact sets [GNZ03].

Suppose the matrix zonotopes for each joint and each time interval $I^{(n)} \subset T_{\text{plan}}$ are constructed as above, in §(8.4.2) Then, analogous to the forward occupancy map FO_j in (8.1), we produce the following rotatotope for each joint j:

$$V_j^{(n)} = \left\{ \sum_{m<j} \left(\prod_{r \leq m} M_m^{(n)} \right) l_m \right\} \oplus \left\{ \left(\prod_{m \leq j} M_m^{(n)} \right) L_j \right\}, \tag{8.35}$$

where $\prod$ again denotes right multiplication as in (8.2). Recall that the Minkowski sum of rotatotopes is define in Lemma 8.4.

We confirm that this rotatotope representation is conservative later, in Theorem 8.5, after revisiting slicing next.

8.5 Slicing Rotatotopes

We now revisit slicing from (6.21). First, we define two operations for indeterminates, which we need for slicing as we did in §6. Then, we define the general slicing algorithm. Finally, we note that the rotatotope FRS is sliceable.

8.5.1 Indeterminate Removal and Inclusion

We now define indeterminate removal and inclusion, which are both operations we need to enable slicing.

First, to invert the indeterminate product, we define the indeterminate **removal** operation, denoted by $\backslash$. Let $\langle\alpha\rangle$ and $\langle\beta\rangle$ be indeterminates of degree 1. Then,

$$\langle\alpha\beta\rangle \setminus \langle\alpha\rangle = \langle\beta\rangle, \tag{8.36}$$

$$\langle\alpha\rangle \setminus \langle\beta\rangle = \langle\alpha\rangle, \quad \text{and} \tag{8.37}$$

$$\langle\alpha\rangle \setminus \langle\alpha\rangle = 1 \in \mathbb{R}. \tag{8.38}$$

Second, we introduce indeterminate **inclusion** using $\in$, which returns "true" if a unique indeterminate is a factor of an indeterminate product, or "false" otherwise. For example, $\langle\alpha\rangle \in \langle\alpha\beta\rangle$ is true;. Similarly, we can write $\langle\alpha\rangle \notin \langle\beta\rangle$.

Notice that inclusion is related to removal as follows. Let $\langle\gamma\rangle$ be a product of indeterminates (i.e., $\deg(\gamma) > 1$). Then,

$$\langle\alpha\rangle \in \langle\gamma\rangle \iff \langle\gamma\rangle \setminus \langle\alpha\rangle \neq \langle\gamma\rangle. \tag{8.39}$$

This is equivalent to checking if $\deg(\langle\gamma\rangle \setminus \langle\alpha\rangle) \leq \deg(\langle\gamma\rangle)$ (we define $\deg(1) = 0$).

In addition, inclusion provides an alternative definition of **sliceability** as in §6.3.2. Consider an indeterminate/generator pair $\langle\gamma\rangle g$, where $\deg(\gamma) > 1$. If $\langle\alpha\rangle \in \langle\gamma\rangle$, then g is $\langle\alpha\rangle$**-sliceable**.

8.5.2 The Slicing Algorithm

We define slicing generically with Algorithm 4. The algorithm takes in a zonotope or rotatotope, V, a list of indeterminates $\{\langle\sigma^{(i)}\rangle\}_{i=1}^{n}$, and a list of values $\{\sigma^{(i)}\}_{i=1}^{n}$ for each of those indeterminates.

To slice V, we initialize the algorithm output as the center of V, which is not sliced. Then, we iterate through the input indeterminates/values and the generators/indeterminates of V. If an input indeterminate is included in an indeterminate of V, then we evaluate that indeterminate (meaning the corresponding generator is multiplied by the input value), and remove the input indeterminate from the indeterminate of V; the result of this evaluation/removal operation is added to the output. If the input indeterminate is not included in an indeterminate of V, then we add that indeterminate/generator of V to the output; that is, it is left unevaluated, and therefore unsliced.

Algorithm 4 $V_{\text{slice}} = \mathsf{slice}\big(V, \{\langle\sigma^{(i)}\rangle\}_{i=1}^{n}, \{\sigma^{(i)}\}_{i=1}^{n}\big)$.

// Let $V = c + \sum_{j=1}^{m} \langle\beta^{(j)}\rangle g^{(j)}$ denote an input zonotope or rotatotope, with indeterminates that may or may not include the inputs to $\mathsf{slice}(\cdot)$ above. That is, each $\langle\beta^{(j)}\rangle$ may be a product of some $\langle\sigma^{(i)}\rangle$, and/or of other indeterminates.
$V_{\text{slice}} \leftarrow c$ // initialize the output
for $i = 1, \cdots, n$ // iterate over input indeterminates/values
  for $j = 1, \cdots, m$ // iterator over generators/indeterminates of V
    if $\langle\sigma^{(i)}\rangle \in \langle\beta^{(j)}\rangle$ // if this generator is $\sigma^{(i)}$-sliceable...
      $V_{\text{slice}} \leftarrow V_{\text{slice}} + \big(\langle\beta^{(j)}\rangle \setminus \langle\sigma^{(i)}\rangle\big)\, \sigma^{(i)} g^{(j)}$ // evaluate the indeterminate and remove it
    else
      $V_{\text{slice}} \leftarrow V_{\text{slice}} + \langle\beta^{(j)}\rangle g^{(j)}$ // do not slice this particular generator of V
    end if
  end for
end for
return V_{slice}

8.5.3 Slicing the Rotatotope FRS

Now, we check that, if we can slice the rotatotope FRS for a given plan $k \in K$, we recover the points in workspace reachable by the arm when tracking that k.

First, we point out that the FRS rotatotopes have k-sliceable generators. Notice that the K part of $V_j^{(n)}$ is left implicit. That is, just as $M_j^{(n)} \subset \mathbb{R}^{3\times 3}$, we have $V_j^{(n)} \subset W$. However, recall that $J_j^{(n)}$ has k-sliceable generators, with indeterminates $\langle\kappa_{k_i}^{(i)}\rangle$ for $i = 1, \cdots, n_K$. From the construction of $V_j^{(n)}$, we know that $V_j^{(n)}$ also has these indeterminates, and therefore has k-sliceable generators.

Second, we remind the reader of the relationship between the FRS and forward occupancy. Recall the general definition of $\mathcal{R}_{\text{FRS}}$ in (3.26), and FO in (8.3). In particular, if the robot starts from an initial condition $x_{\text{hi},0}$, and is tracking a plan k at time $t \in T_{\text{plan}}$, then

$$p \in \mathsf{FO}(x(t;k)) \iff (t, x_{\text{hi},0}, p, k) \in \mathcal{R}_{\text{FRS}} \tag{8.40}$$

by construction of $\mathcal{R}_{\text{FRS}}$.

Third and finally, we confirm that the FRS rotatotopes behave “as expected” when sliced. That is, we check that the rotatope FRS is conservative, meaning that any point in the FRS is also in the rotatotope FRS:

Theorem 8.5. *Suppose $t \in I^{(n)}$, $k \in K$, and $(t, x_{\text{hi},0}, p, k) \in \mathcal{R}_{\text{FRS}}$, where p is reached by the robot's j^{th} link; that is, $p \in \mathsf{FO}_j(x(t;k))$. Suppose l_m and L_j are represented as zonotopes. Suppose that each $V_j^{(n)}$ is constructed as in (8.35), where the matrix zonotopes $M_j^{(n)}$ account for the initial joint angles due to $x_{\text{hi},0}$ as in (8.34). Denote $k = (k_1, \cdots, k_{n_K})$, and define the values*

$$\kappa_{k_i}^{(i)} = \frac{k_{0,i} - k_i}{\Delta_{k_i}}. \tag{8.41}$$

Then, p is in the j^{th} sliced rotatotope FRS:

$$p \in \mathsf{slice}\Big(V_j^{(n)}, \{\langle \kappa_{k_i}^{(i)} \rangle\}_{i=1}^{n_K}, \{\kappa_{k_i}^{(i)}\}_{i=1}^{n_K}\Big). \tag{8.42}$$

Proof. By Lemma 6.1, we know that the zonotope JRS is conservative, meaning that $(t, c, s, k) \in J_j^{(n)}$, where c and s are the cosine and sine of joint angle x_j at time t for plan k. By construction of $M_j^{(n)}$, we know that $R_j(x_j) \in M_j^{(n)}$. Since the forward occupancy is given directly by the zonotope displacements l_m (with $m = 1, \cdots, n_X$), and by the zonotope volume L_j, the proof follows from noticing that (8.35) is analogous to the definition of FO_j (8.1). □

8.6 Online Planning

We now discuss how to use the rotatotope FRS to generate collision avoidance constraints for online trajectory optimization. The procedure is nearly identical to that for the zonotope FRS in §6.4.

For this section, suppose the robot is in a planning iteration with initial condition $x_{\text{hi},0} \in X_{\text{hi},0}$. Suppose also that we have an obstacle reachable set $\mathcal{R}_{\text{obs}}$ of predictions of obstacle motion. Recall that the goal in online planning is to identify the unsafe plans $K_{\text{unsf}} \subset K$ as in (3.37):

$$K_{\text{unsf}} \subseteq \text{proj}_K(\mathcal{R}_{\text{FRS}} \cap \mathcal{R}_{\text{obs}}), \tag{8.43}$$

where we have dropped the notation indicating a particular planning iteration (e.g., $K_{\text{unsf}}^{(i)}$, $\mathcal{R}_{\text{FRS}}^{(i)}$, and $\mathcal{R}_{\text{obs}}^{(i)}$ as used for the theory in §3).

8.6.1 Obstacle Representation

Before identifying K_{unsf}, we require that $\mathcal{R}_{\text{obs}}$ is represented with zonotopes, just as in §6.4.1. Suppose there are n_{obs} obstacles that are predicted in $\mathcal{R}_{\text{obs}}$. We assume that, for each $n = 1, \cdots, n_{\text{RS}}$, there exists a collection of zonotopes $\{Z_{\text{obs}}^{(n,m)}\}_{m=1}^{n_{\text{obs}}}$ such that

$$\text{proj}_{I^{(n)} \times X}(\mathcal{R}_{\text{obs}}) \subseteq \bigcup_{m=1}^{n_{\text{obs}}} Z_{\text{obs}}^{(n,m)}. \tag{8.44}$$

In other words, each zonotope $Z_{\text{obs}}^{(n,m)}$ contains all points in workspace reached by obstacle m for all $t \in I^{(n)}$. Note that common obstacle representations, such as voxel grids, can be easily expressed using zonotopes.

8.6.2 Fully-Sliceable Generators

Suppose that we have constructed the FRS rotatotopes $V_j^{(n)}$ for each joint j and each time interval $I^{(n)} \subset T_{\text{plan}}$, given the initial condition $x_{\text{hi},0}$. Just as with zonotopes, before identifying unsafe plans, we first separate the rotatotope generators into k-sliceable and non-k-sliceable generators:

Lemma 8.6. *Each $V_j^{(n)}$ can be written as*

$$V_j^{(n)} = c_j^{(n)} + \sum_{i=1}^{n_{\text{slice}}} \langle \kappa^{(i)} \rangle g_{\text{slice}}^{(i)} + \sum_{i=1}^{n_{\text{extra}}} \langle \beta^{(i)} \rangle g_{\text{extra}}^{(i)}, \tag{8.45}$$

where the indeterminates $\kappa^{(i)}$ have the the following properties. First, for each $i = 1, \cdots, n_{\text{slice}}$, there exists at least one $j \in \{1, \cdots, n_K\}$ such that $\kappa_{k_j}^{(j)} \in \langle \kappa^{(i)} \rangle$; that is, every $\langle \kappa^{(i)} \rangle$ is a product of at least one k-sliceable indeterminate. Second, if $\langle \alpha \rangle$ is any non-k-sliceable indeterminate, then $\langle \alpha \rangle \notin \langle \kappa^{(i)} \rangle$; that is, every $\langle \kappa^{(i)} \rangle$ is only a product of k-sliceable indeterminates.

Before we prove this lemma, note that n_{slice}, n_{extra}, and all the generators and indeterminates in (8.45) are unique to each joint j and time interval n; we omit the indices n and j to ease notation. Also note that, by Lemma 6.8, each JRS zonotope $J_j^{(n)}$ can be divided into k-sliceable and non-k-sliceable generators as in (8.29), meaning each $M_j^{(n)}$ can also be divided in this way by construction. However, for $V_j^{(n)}$, we make a slightly different statement: there is one set of generators that are products of *only* k-sliceable generators/indeterminates, and another set of generators that may or may not be k-sliceable.

Proof. (of Lemma 8.6) We prove this lemma by constructing the claimed generator/indeterminate pairs.

Consider the product of the matrix zonotopes $M_a^{(n)}$ and $M_b^{(n)}$ for joints $a, b \in \{1, \cdots, n_X - 1\}$, with $a < b$. To produce $M_a^{(n)} M_b^{(n)}$, we multiply all the k_a-sliceable generators of $M_a^{(n)}$ with all the k_b-sliceable generators of $M_b^{(n)}$, thereby producing generators sliceable by both k_a and k_b, plus other generators that are sliceable by k_a, k_b, or neither.

Now suppose we produce a rotatotope $V_c^{(n)}$ for joint c (with $b < c \leq n_X$) using the product $M_a^{(n)} M_b^{(n)}$ as in (8.35). Then $V_c^{(n)}$ contains generators produced by $M_a^{(n)} M_b^{(n)} L_c$, where L_c is the link volume represented as a zonotope. This means that all of the generators that are both k_a- and k_b-sliceable are multiplied by the *center* of L_c to produce generators of $V_c^{(n)}$ that are again both k_a- and k_b-sliceable. Similarly, for any joint $d \in \{1, \cdots, c-1\}$, the link displacement terms $M_a^{(n)} M_b^{(n)} l_d$ produce generators of $V_c^{(n)}$ that are k_a- and k_b-sliceable. Since a, b, and d were arbitrary, the proof is complete. In particular, in (8.45), for any $j \in \{1, \cdots, n_X\}$, the generators that are k_a-sliceable for any $a \leq j$, and not sliceable by any other indeterminates, are those that we denote $g_{\text{slice}}^{(i)}$; we denote the indeterminates corresponding to each of these generators as $\langle \kappa^{(i)} \rangle$. $\square$

We call such generators **fully k-sliceable**, meaning the generators with *only* k-sliceable indeterminates.

8.6.3 Identifying Unsafe Plans

We now identify the unsafe plans for link j, assuming a single obstacle (i.e., $n_{\text{obs}} = 1$) to ease exposition. We extend this to all of the robot's links, and to any finite number of obstacles, at the end of this section.

First, we overapproximate the subset of $V_j^{(n)}$ containing the non-fully-k-sliceable generators with a zonotope:

Lemma 8.7. *Suppose we separate the generators of $V_j^{(n)}$ as in Lemma 8.6. Define the zonotope*

$$Z_{\text{extra},j}^{(n)} = 0 + \sum_{i=1}^{n_{\text{extra}}} \langle \gamma^{(i)} \rangle g_{\text{extra}}^{(i)}, \tag{8.46}$$

where $\deg(\gamma^{(i)}) = 1$ for all $i = 1, \cdots, n_{\text{extra}}$. Define the rotatotope

$$V_{\text{slice},j}^{(n)} = c_j^{(n)} + \sum_{i=1}^{n_{\text{slice}}} \langle \kappa^{(i)} \rangle g_{\text{slice}}^{(i)}. \tag{8.47}$$

Then

$$V_j^{(n)} \subseteq Z_{\text{extra},j}^{(n)} \oplus V_{\text{slice},j}^{(n)}. \tag{8.48}$$

Proof. This claim follows from Lemma 8.4 (which defines the rotatotope Minkowski sum) and Lemma 8.7 (we can overapproximate a rotatotope with a zonotope). $\square$

Second, we notice that slicing $V_{\text{slice},j}^{(n)}$ produces a point. This is similar to our approach of slicing the FRS zonotopes to a point in (6.37), and enables a nearly identical constraint construction.

Lemma 8.8. *Let $V^{(n)}_{\text{slice},j}$ be as in* (8.47)*. Let $k = (k_1, \cdots, k_{n_K}) \in K$, and define the values $\kappa^{(i)}_{k_i} = \frac{k_{0,i}-k_i}{\Delta_{k_i}}$. Then slicing $V^{(n)}_{\text{slice}}$ by all $\langle \kappa^{(i)}_{k_i} \rangle$ produces a point in $\mathbb{R}^3$:*

$$\mathsf{slice}\Big(V^{(n)}_{\text{slice},j}, \{\langle \kappa^{(i)}_{k_i} \rangle\}_{i=1}^{n_K}, \{\kappa^{(i)}\}_{i=1}^{n_K}\Big) \in \mathbb{R}^3. \tag{8.49}$$

Proof. The claim follows from Lemma 8.6. Notice that every indeterminate $\langle \kappa^{(i)} \rangle$ (for indices $i \in \{1, \cdots, n_{\text{slice}}\}$) is a product of one or more of the indeterminates $\langle \kappa^{(m)}_{k_m} \rangle$ (for $m \in \{1, \cdots, n_K\}$, meaning these indeterminates are of degree 1 and each corresponds to k_m as indicated by the subscript). Since $V^{(n)}_{\text{slice},j}$ only contains fully-k-sliceable generators, the slicing operation in (8.49) results in the evaluation of *every* indeterminate of $V^{(n)}_{\text{slice},j}$. $\square$

Finally, we are ready to identify unsafe trajectory parameters for the obstacle zonotope $Z^{(n)}_{\text{obs}}$ (recall that $n_{\text{obs}} = 1$ for now).

Theorem 8.9. *Let $Z^{(n)}_{\text{extra},j}$ be as in* (8.46)*. For each link j, define the function $v^{(n)}_{\text{slice},j} : K \to \mathbb{R}^3$ for which*

$$v^{(n)}_{\text{slice},j} = \mathsf{slice}\Big(V^{(n)}_{\text{slice},j}, \{\langle \kappa^{(i)}_{k_i} \rangle\}_{i=1}^{n_K}, \{\kappa^{(i)}\}_{i=1}^{n_K}\Big), \tag{8.50}$$

with the indeterminates and values are given as in the premises of Lemma (8.8)*. We claim that*

$$k \in K_{\text{unsf}} \implies \exists\, j \in \{1, \cdots, n_X\} \text{ s.t. } v^{(n)}_{\text{slice},j}(k) \in Z^{(n)}_{\text{obs}} \oplus Z^{(n)}_{\text{extra},j}. \tag{8.51}$$

That is, if k is unsafe, then at least one link rotatotope (partially overapproximated by a zonotope $Z^{(n)}_{\text{extra},j}$) is in collision when sliced by k.

Proof. To prove (8.51), first notice that, by separating $V^{(n)}_j$ into its fully-sliceable generators and a zonotope of all other generators, we produce a conservative overapproximation of the forward occupancy FO_j for link j using Lemma 8.3. Recall, by *conservative*, we mean that " $\implies$ " holds in (8.51), but " $\impliedby$ " does not necessarily hold. This conservatism follows from two facts: (1) $V^{(n)}_j$ is already conservative by Theorem 8.5, and (2) we overapproximate the non-fully-k-sliceable generators with a zonotope. The claim then follows from Lemma 8.6 and zonotope intersection as in Lemma 6.7. $\square$

8.6.4 Numerical Constraint Formulation

We now provide a method for numerical implementation of the unsafe parameters identified by Theorem 8.9. By design, this procedure is nearly identical to the one we presented in §6.4.4.

First, we note a useful property of $v^{(n)}_{\text{slice},j}$:

Lemma 8.10. *The function* $v_{\text{slice},j}^{(n)} : K \to \mathbb{R}^3$ *is a polynomial in* k.

Proof. First, recall the construction of $V_j^{(n)}$, where we multiply the indeterminates of matrix zonotopes together and with the link zonotope. Therefore, one can think of $V_j^{(n)}$ as a polynomial of indeterminates, with the rotatotope generators as coefficients. This polynomial structure is not lost when we separate $V_j^{(n)}$ into its fully-k-sliceable generators to construct $v_{\text{slice},j}^{(n)}$. The claim then follows from the definition of slicing as evaluation of the indeterminates in $v_{\text{slice},j}^{(n)}$, and the fact that $v_{\text{slice},j}^{(n)}$ only uses the k-sliceable generators of $V_j^{(n)}$. □

Now we create constraints on K to represent K_{unsf}:

Corollary 8.11 (to Theorem 8.9). *Let* $Z_{\text{obs}}^{(n)}$, K_{unsf}, $v_{\text{slice},j}^{(n)}$, *and* $Z_{\text{extra},j}^{(n)}$ *be as in the premises of Theorem 8.9. Let* A_{obs} *and* b_{obs} *be a halfspace representation such that*

$$x \in Z_{\text{obs}}^{(n)} \oplus Z_{\text{extra}}^{(n)} \iff \max\left(A_{\text{buf}} x - b_{\text{buf}}\right) \leq 0. \tag{8.52}$$

Then we can identify safe trajectory parameters as:

$$k \in K \setminus K_{\text{unsf}} \impliedby \exists\, j \in \{1, \cdots, n_X\} \text{ s.t. } -\max\left(A_{\text{buf}} v_{\text{slice},j}^{(n)}(k) - b_{\text{buf}}\right) < 0. \tag{8.53}$$

Proof. The existence of A_{buf} and b_{buf} follow from Lemma 6.11 and the construction of $Z_{\text{obs}}^{(n)} \oplus Z_{\text{extra}}^{(n)}$ as a zonotope. The desired result then follows from Theorem 8.9, □

The utility of Lemma 8.10 and Corollary 8.11 is that we can check if a given plan k is safe by taking the max of a polynomial, which is efficient for online trajectory planning. That is, we can collision check the entire trajectory parameterized by k *in continuous time and space* using a fast polynomial evaluation. And, for the purpose of gradient-based optimization, this formulation also admits an analytic subgradient [Pol12, Theorem 5.3.5] just as in §6.4.

8.6.5 Trajectory Optimization Formulation

To conclude this section, as did for the SOS and zonotope FRSes, we rewrite the trajectory optimization program (3.38) (see §3.8) using the safety constraints above. As we did for the zonotope case, we extend the above discussion to the multiple obstacle case, and bring back the receding-horizon planning iteration index.

Suppose that RTD is in the n^{th} receding-horizon planning iteration. Suppose we have a zonotope obstacle representation $\{Z_{\text{obs}}^{(n,m)}\}_{m=1}^{n_{\text{obs}}}$ as in (8.44), where $n = 1, \cdots, n_{\text{RS}}$ indexes the FRS time intervals, and $m = 1, \cdots, n_{\text{obs}}$ indexes the obstacle zonotopes. Suppose that $x_{\text{hi},0} \in X_{\text{hi},0}$ is used to construct the FRS rotatotopes $V_j^{(n)}$ for each joint (indexed by $j = 1, \cdots, n_X$) and each time interval (indexed by $n = 1, \cdots, n_{\text{RS}}$). Let $A_{\text{buf}}^{(j,n,m)}$ and $b_{\text{buf}}^{(j,n,m)}$ be the halfspace representation of $Z_{\text{obs}}^{(n,m)} \oplus Z_{\text{extra}}^{(j,n,m)}$ for joint j, time interval n, and obstacle zonotope m.

Then, RTD attempts to solve the following optimization program to find the plan $k^{(n)}$ in the n^{th} receding-horizon iteration:

$$k^{(n)} = \underset{k \,\in\, K}{\text{argmin}} \quad \mathsf{cost}(k) \tag{8.54}$$

$$\text{s.t.} \qquad -\max\left(A_{\text{buf}}^{(j,n,m)} v_{\text{slice},j}^{(n)}(k) - b_{\text{buf}}^{(j,n,m)}\right) < 0 \tag{8.55}$$

$$k^{(i)} \in \mathcal{K}_{\text{lim}}(x_{\text{hi},0}), \tag{8.56}$$

where (8.55) holds for all joints $j = 1, \cdots, n_X$, FRS time intervals $n = 1, \cdots, n_{\text{RS}}$, and obstacles $m = 1, \cdots, n_{\text{obs}}$. As a reminder, $\mathcal{K}_{\text{lim}}$ provides limits on the choices of plans given the robot's initial condition (e.g., it can prescribe a maximum commanded acceleration given the current joint velocities in $x_{\text{hi},0}$).

8.7 Chapter Review

The takeaway of this chapter is a method for representing parameterized swept volumes of arbitrary manipulators, which enables guaranteed collision-free motion planning.

8.7.1 Chapter Summary

We began this chapter by establishing notation and assumptions for manipulators, the most important being that, in this work, we only consider the manipulator kinematics. We then explained manipulator RTD at a high level. Next, we introduced rotatotopes, and showed how to use them to build a manipulator FRS. We also confirmed that rotatotopes are sliceable like FRS zonotopes, and developed a general slicing algorithm. Finally, we showed how to use the rotatotope FRS for online planning.

8.7.2 What is Missing?

The manipulator dynamics are the key ingredient that is necessary for future development of manipulator RTD. Note that manipulator tracking error can be represented with zonotopes [GA17]. We suspect that forces due to gravity and Coriolis effects can be included in our formulation as additional trajectory parameters, perhaps with uncertainty over the duration of a single plan.

This concludes our presentation of the rotatotope FRS for manipulator RTD, and concludes all theoretical contributions of RTD in this dissertation. In the following chapter, we provide implement the SOS, zonotope, and rotatotope approaches RTD on specific robotic platforms in simulation and on hardware.

CHAPTER 9

Implementations and Comparisons

This chapter demonstrates RTD, as developed in the previous chapters, on seven different robot platforms with three different morphologies. In total, we demonstrate safe, real-time trajectory planning over thousands of simulations and dozens of hardware demonstrations. We also provide extensive comparisons to a variety of state-of-the-art methods for each robot, and show that RTD outperforms these methods in terms of safety and task completion.

The sections of this chapter are as follows. (§9.1) We demonstrate RTD on a Segway wheeled robot in unstructured static and dynamic scenarios. (§9.2) We then use a Rover wheeled robot to show RTD in structured autonomous lane change scenarios. (§9.3) We extend the wheeled robot approaches to much higher speeds with a Ford Fusion passenger sedan. (§9.4) We also demonstrate RTD interacting with pedestrians and making guaranteed-safe unprotected left turns with an Electric Vehicle car-like platform. (§9.5) Then, we transition to RTD for aerial robots, with a Hummingbird quadrotor in random static environments. (§9.6) We extend the aerial robot work to a Parrot Mambo microdrone, demonstrating RTD with aerodynamic drag, ground effect, and dynamic obstacles. (§9.7) We conclude by presenting manipulator RTD on a Fetch manipulator.

9.1 The Segway Wheeled Robot

The Segway is a differential-drive wheeled robot. We use this platform to demonstrate how RTD enables real-time, collision-free trajectory planning in unstructured, random environments. Note, the Segway is used as a running example in §3. Also see Figures 1.1 and 1.2.

The robot is simulated using our open-source MATLAB simulator [KVL19], with code available [KVV19]. We use $t_{\text{plan}} = 0.5$ s for the receding-horizon planning timeout.

The hardware is as follows. Note, the Segway has a circular body of radius 0.38 m. A Hokuyo UTM-30LX planar lidar, which is accurate up to 4.0 m away, is mounted to the front of the robot. The robot is controlled by a 4.0 GHz laptop with 64 GB of memory, running MATLAB and the Robot Operating System [QCG^{+}09]. We use Google Cartographer for localization and mapping

[HKRA16]. All computation is run onboard. We specify $t_{\text{plan}} = 0.5$ s, but reserve 0.2 s in each planning iteration for mapping and communication delays, so RTD is given 0.3 to run trajectory optimization, which uses MATLAB's `fmincon` generic nonlinear solver. A video is available: `https://youtu.be/FJns7YpdMXQ`.

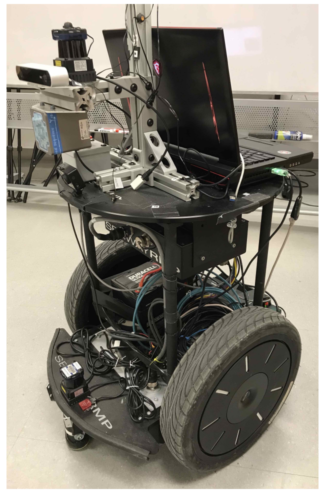


Figure 9.1: The Segway wheeled robot.

9.1.1 High-Fidelity Model

The Segway robot has generalized coordinates of its center-of-mass position and its heading $(p_1, p_2, \theta) \in Q = \mathsf{SE}(2)$, and generalized longitudinal and angular velocities, $(v, \omega) \in \dot{Q} \subset \mathbb{R}^2$. Note, we refer to v as just the "velocity." The high-fidelity model is a dynamic unicycle with control inputs for longitudinal and angular acceleration, restated from Running Example 3.1:

$$f_{\text{hi}}(t, x_{\text{hi}}(t), u(\cdot)) = \begin{bmatrix} \dot{p}_1(t) \\ \dot{p}_2(t) \\ \dot{\theta}(t) \\ \dot{v}(t) \\ \dot{\omega}(t) \end{bmatrix} = \begin{bmatrix} v(t)\cos(\theta(t)) \\ v(t)\sin(\theta(t)) \\ \omega(t) \\ \text{sat}_v\left(\beta_v \cdot (u_v(\cdot) - v(t))\right) \\ \text{sat}_\omega\left(\beta_\omega \cdot (u_\omega(\cdot) - \omega(t))\right) \end{bmatrix}, \tag{9.1}$$

where $u = (u_v, u_\omega)$ commands the velocity and yaw rate. We limit the velocity to $[0, 1.5]$ m/s and the yaw rate to $[-1, 1]$ rad/s. The function sat_v saturates the longitudinal acceleration to be in $[-5.9, 5.9]$ m/s^2. Similarly, sat_ω saturates the angular acceleration to be in $[-3.75, 3.75]$ rad/s^2. The constants $\beta_v = 3.00$ and $\beta_\omega - 2.95$ are found using system identification.

For the hardware, we use modeling error of $\varepsilon_{p_1} = \varepsilon_{p_2} = 0.1$ m in the plane per Assumption 3.1. We find the modeling error in the other states to be negligible.

9.1.2 Planning Model

The Segway's planning model is as in Running Example 3.5. We specify that $X = P$, the position subspace of $Q = \mathsf{SE}(2) = P \times \Theta$, with state $x = (p_1, p_2) \in X$. The planning model is

$$f(t, x(t;k), k) = s(t) \begin{bmatrix} k_1 - k_2 \cdot (p_2(t;k) - p_{2,0}) \\ k_2 \cdot (p_1(t;k) - p_{1,0}) \end{bmatrix} \text{ with} \tag{9.2}$$

$$s(t) = \begin{cases} 1 & t \in [0, t_{\text{plan}}) \\ 1 - \frac{t - t_{\text{plan}}}{t_{\text{f}} - t_{\text{plan}}} & t \in [t_{\text{plan}}, t_{\text{f}}] \end{cases}, \tag{9.3}$$

with the point $x_0 = (p_{1,0}, p_{2,0}) \in P$. Trajectories of this model end in a stop because of the scaling function $s : T_{\text{plan}} \to [0, 1]$. This model creates circular arc trajectories (that is, Dubins' paths parameterized by time), with longitudinal velocity k_1 and angular velocity k_2, initial position x_0, and an initial heading of $\theta(0) = 0$. We choose $t_{\text{plan}} = 0.5$ s, and t_{f} large enough for the robot to brake to a stop while obeying the maximum acceleration reported above.

We specify the trajectory parameter space as

$$k_1 \in [0, 1.5] \text{ m/s and} \tag{9.4}$$

$$k_2 \in [-1, 1] \text{ m/s.} \tag{9.5}$$

Given an initial condition $x_{\text{hi},0} = (p_{1,0}, p_{2,0}, \theta_0, v_0, \omega_0) \in X_{\text{hi},0}$, we specify that

$$\mathcal{K}_{\text{lim}}(x_{\text{hi},0}) = [v_0 - \Delta_v, v_0 + \Delta_v] \times [\omega_0 - \Delta_\omega, \omega_0 + \Delta_\omega] \cap K, \tag{9.6}$$

where the intersection with K is to ensure that we obey the bounds on k_1 and k_2. We use $\Delta_v = 0.5$ m/s and $\Delta_\omega = 1$ rad/s.

9.1.3 Tracking Controller

The Segway uses a proportional-derivative controller as in Running Example 3.8. Let $G_P \in \mathbb{R}^{2\times 2}$, $G_\Theta \in \mathbb{R}^{1\times 1}$, and $G_{\dot{Q}} \in \mathbb{R}^{2\times 2}$ be matrices of control gains. Suppose the Segway is in the i^{th} planning iteration, starting from initial condition $x_{\text{hi},0}^{(i)}$, and let $k \in K$. Then, the Segway's tracking controller is given by

$$u_k(t, x_{\text{hi}}(t;k)) = G_P e_P(t;k) + G_\Theta \cdot (\theta(t;k) - s(t - t^{(i)}) k_2 t) + \tag{9.7}$$

$$+ G_{\dot{Q}} \begin{bmatrix} s(t - t^{(i)}) k_1 - v(t;k) \\ s(t - t^{(i)}) k_2 - \omega(t;k) \end{bmatrix}, \tag{9.8}$$

with the position error e_P given in the robot's body-fixed coordinate frame:

$$e_P(t;k) = \begin{bmatrix} \cos(\theta(t;k)) & \sin(\theta(t;k)) \\ -\sin(\theta(t;k)) & \cos(\theta(t;k)) \end{bmatrix} \mathrm{proj}_P(x_{\mathrm{hi}}(t;k) - x^{(i)}_{\mathrm{plan}}(t;k)), \tag{9.9}$$

where $x^{(i)}_{\mathrm{plan}}(t;k) = \mathsf{liftplan}(i, x(t - t^{(i)}; k))$. Notice that $k_2 t = \mathrm{proj}_\Theta(x^{(i)}_{\mathrm{plan}}(t;k))$ and similarly the error terms for v and ω are functions of the lifted plan. The control gain values are available in the repository [KVV19].

9.1.4 Forward Reachable Set

To compute the Segway's FRS, we first compute the ERS, then use sums-of-squares programming approach in §4. To compute the ERS, we first partition its generalized velocity space, as per the ERS computation in §7, into three subsets:

$$\dot{Q}^{(1)} = [0.0\text{ m/s}, 0.5\text{ m/s}] \times [-1\text{ rad/s}, +1\text{ rad/s}] \tag{9.10}$$

$$\dot{Q}^{(2)} = [0.5\text{ m/s}, 1.0\text{ m/s}] \times [-1\text{ rad/s}, +1\text{ rad/s}] \tag{9.11}$$

$$\dot{Q}^{(3)} = [1.0\text{ m/s}, 1.5\text{ m/s}] \times [-1\text{ rad/s}, +1\text{ rad/s}]. \tag{9.12}$$

We then apply Algorithm 3. We store the tracking error function $f^{(j)}_{\mathrm{err}} \in \mathbb{R}_3[t]$ for each $\dot{Q}^{(j)}$ as in §7.3.1 (that is, each $f^{(j)}_{\mathrm{err}}$ is a degree 3 time-varying polynomial).

Given the ERS representation as above, we compute the FRS by solving for the functions $g_{\mathrm{dyn},l}$ and $g_{\mathrm{stat},l}$ in (4.33) with $l = 5$. We use the planning model f as in (9.2), and the error models as above. For running the Segway in static environments, we compute the FRS without the time scaling s; instead, we choose t_{f} large enough that the planning model trajectories travel the same distance without braking as they would with braking; we found that this approach led to a faster FRS computation, with lower memory usage. See the online RTD tutorial [KV19] and Segway-specific open-source implementation [KVV19], as well as the paper [KVB+20].

9.1.5 Simulation in Static Environments

Environment The simulated static environment for the Segway is a 9×5 m^2 room, with the longer dimension oriented east-west. The room is filled with 6 to 15 randomly-distributed box-shaped obstacles with a side length of 0.3 m. A random start location is chosen on the west side of the room and a random goal is chosen on the east side. We created 1000 such environments.

A trial is considered successful if the Segway reaches the goal without collision (i.e., touching any obstacles). Since obstacles are distributed randomly, it may be impossible to reach the goal in some trials; we address this by counting the number of collisions and number of goals reached separately.

High-Level Planner In these environments, for the Segway's high-level planner, we use Dijkstra's algorithm on a graph representing a grid in the robot's planning space X; this provides a coarse path and intermediate waypoints between the Segway and the global goal. At each i^{th} planning iteration (which begins at $t^{(i)} \in T$), the trajectory optimization cost function is the Euclidean distance between the waypoint and the robot's position at time $t^{(i)} + t_{\text{plan}}$ when tracking $k^{(i)}$ (the decision variable for online trajectory optimization).

RTD Online Planning To implement RTD at runtime, we use the polynomial $g_{\text{stat},l}$ representing the FRS to construct obstacle constraints, then solved online trajectory optimization using MATLAB's `fmincon` solver in each receding-horizon planning iteration [Mat19b]. To generate constraints, we discretize obstacles as in §5, with $b = 0.001$. The Segway has a circular body with $r_{\text{max}} = 0.76$ m, resulting in $r = 0.055$ m and $a = 0.002$ m per Example 5.10.

Comparison Methods We compare RTD to two other methods. First, an RRT trajectory planner based on [KFT$^+$08, PKA16, PLM06]; these papers describe a variety of heuristics for growing a tree of a robot's trajectories with nodes in the high-fidelity state space. Second, a pseudospectral Nonlinear Model-Predictive Control (NMPC) method called GPOPS-II [PR14]. We test both of these methods with and without the receding-horizon timeout t_{plan} enforced.

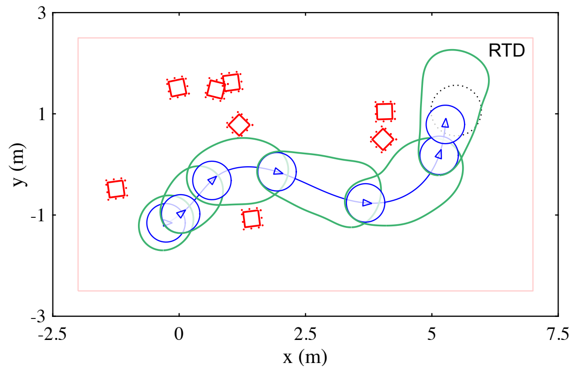
Since neither of these methods prescribe how to correctly buffer obstacles to ensure safety, we tested several buffer sizes; we found that a buffer of 0.45 m provides the best balance of performance and safety for both methods [KVB$^+$20, Experiment 1].

Note that NMPC methods are effective for *tracking* planned trajectories; however, we use NMPC here to *generate* trajectory plans. As per the literature in §2, trajectory planning is typically performed by planning a collision-free path, then smoothing it reparameterizing it by time (i.e., an RRT-style approach); or by planning a coarse path, then relying on a tracking controller to smoothly and safely track it (i.e., an MPC-style approach). RTD lies between these two paradigms of path planners such as RRT and tracking controllers such as NMPC, so we compare against both to show that this middle tier of trajectory planning is the "right" place to enforce safety.

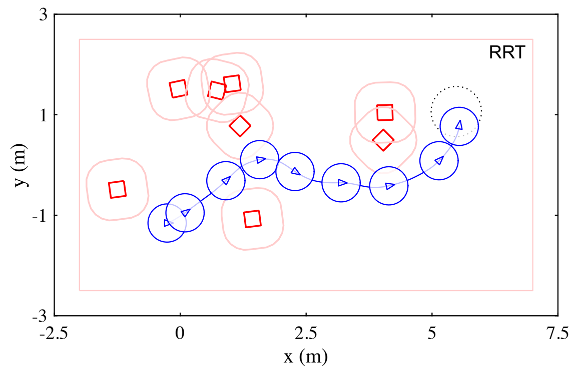
Results RTD has no collisions, as expected, and reaches goal locations the most often out all methods with the planning timeout enforced. Both RRT and NMPC experience collisions. Importantly, when the planning timeout is enforced, both of these methods suffer significant reductions in performance, with NMPC unable to find any solutions in the vicinity of obstacles. See Figures 9.2, 9.4, and 9.3 for example environments and results for all three planners. The results are summarized in Table 9.1.

Method	Goals [%]	Collisions (%)
RTD[1]	**96.3**	**0.0**
RRT[1]	78.2	2.4
NMPC[1]	0.0	**0.0**
RRT[2]	86.2	0.6
NMPC[2]	**97.0**	0.6

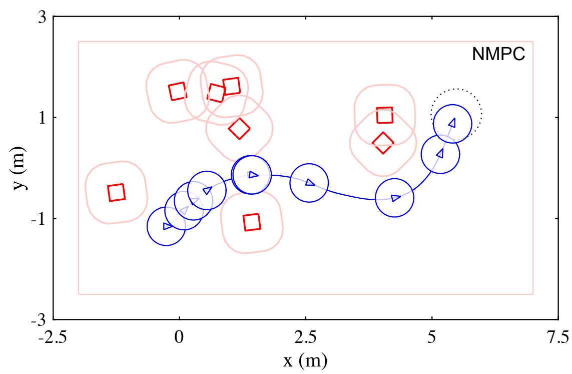
Table 9.1: Segway simulation/comparison results in 1000 random static environments. We compare to an RRT based on [KFT+08, PKA16, PLM06], and NMPC [PR14]. Note, [1] indicates that real-time planning (the timeout t_{plan}) was enforced, and [2] indicates that real-time planning was *not* enforced. This distinction is also shown with a dashed line.



(a)

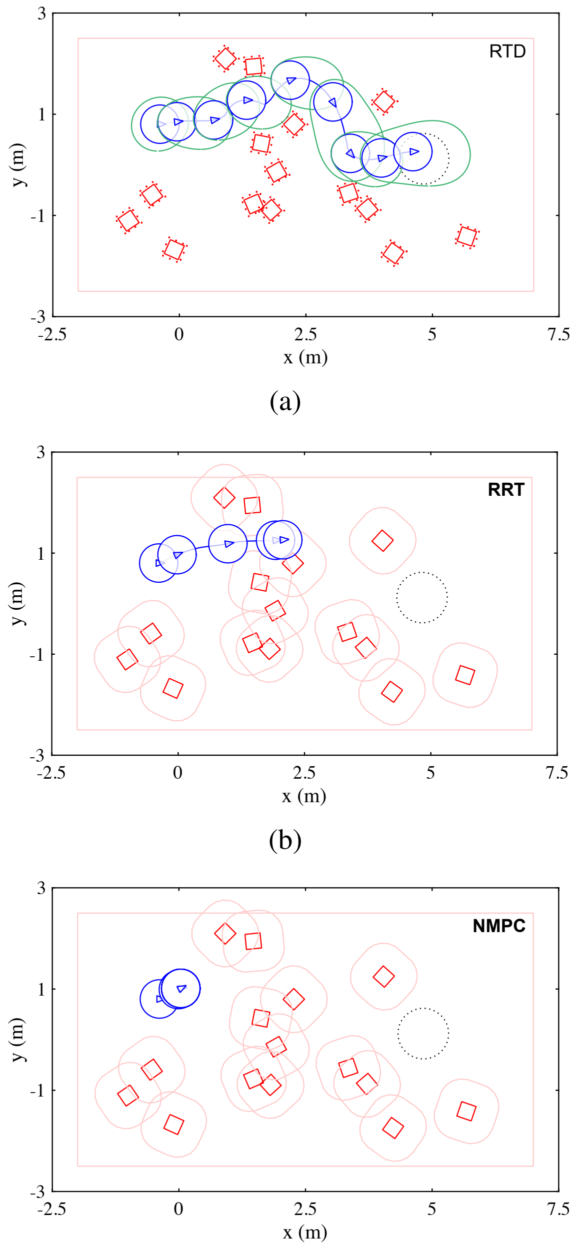


(b)



(c)

Figure 9.2: Sample simulation environments for the Segway, which starts on the west (left) side of the environment, with the goal plotted as a dotted circle on the east (right) side of the environment. The Segway's pose is plotted as a solid circle every 1.5 s, or less frequently when the Segway is stopped or spinning in place. For RTD, contours of the FRS are plotted to show the reachable set corresponding to the plans in each planning iteration. The actual (non-buffered) obstacles for all three planners are plotted as solid boxes. For RTD, the discretized obstacle is plotted as points around each box. For RRT and NMPC, the buffered obstacles are plotted as light lines around each box. This figure shows an environment where all three planners are successful. Row 2 shows an environment where RTD is successful, but RRT and NMPC are not.

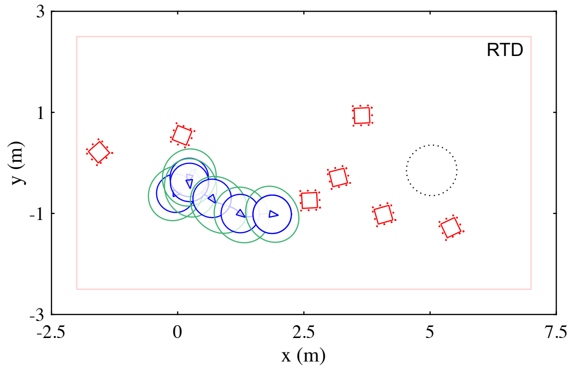


(a)

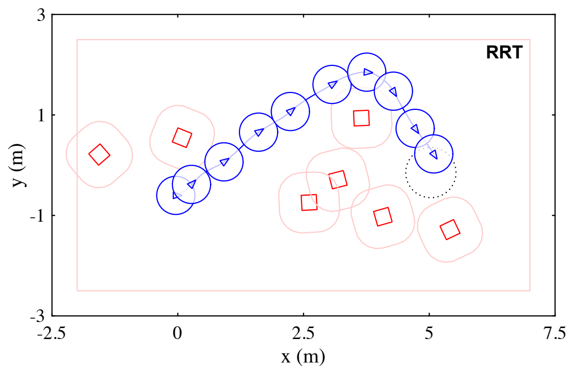
(b)

(c)

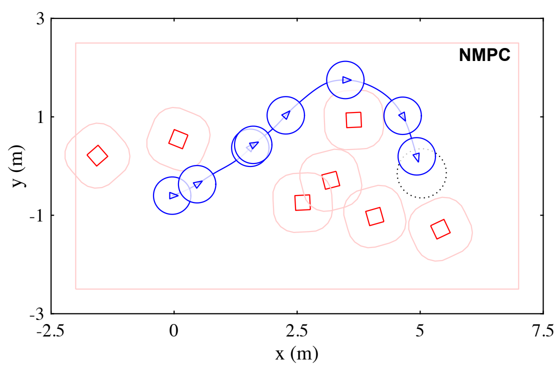
Figure 9.3: Sample simulation environments for the Segway, with the same plotting convention as Figure 9.2. RTD is successful, whereas RRT and NMPC are not. RRT attempts to navigate a gap between several obstacles, where it is unable to find a new plan; it collisides when it tries to brake along its previously-planned trajectory. NMPC brakes because it cannot compute a safe plan to navigate the same gap where RRT collided; here, NMPC happens to brake safely and gets stuck because it cannot find a new plan fast enough.



(a)



(b)



(c)

Figure 9.4: Sample simulation environments for the Segway, with the same plotting convention as Figure 9.2. RTD stops safely, but fails to reach the goal, whereas RRT and NMPC do reach the goal. RTD initially turns north more sharply than RRT or NMPC, which forces it to brake safely; it then finds a safe path south, which causes the high-level planner to reroute it even farther south to where there is no feasible solution, causing RTD to get stuck because the southern route is considered feasible by the high-level planner. RRT and NMPC reach the goal because they do not turn north as sharply initially, so the high-level planner is able to route them north and around the obstacles.

9.1.6 Simulation in Dynamic Environments

Environment The simulated dynamic environment for the Segway is a 20×10 m^2 world with 1–10 0.3×0.3 m^2 box-shaped obstacles. We ran 100 trials for each number of obstacles (1000 trials total). In each trial, a random start and goal are chosen approximately 18 m apart.

Each obstacle moves at a random constant speed, up to 1 m/s, along a random piecewise-linear path. We ensured that obstacles do not stay on the (random) goal, so that a feasible path to the goal always exists at some time during each simulation.

We do not model interactions; that is, the obstacles may randomly act aggressively and cause collisions. Therefore, as per §3.4.2, we count *at-fault* collisions, meaning that the Segway is not at-fault if it is stationary during a collision.

High-Level Planner We use a "straight line" HLP. That is, at each receding-horizon planning iteration, we generate an intermediate waypoint 1 m ahead of the robot, along a straight line between the robot and the global goal.

RTD Online Planning We use the same FRS as computed for the static case, but using $g_{\text{dyn},l}$ instead of $g_{\text{stat},l}$, to create collision-avoidance constraints for dynamic obstacles as in §5.7. We also computed an FRS for a $[1.5, 2.0]$ m/s speed range (the static environments only have the Segway traveling up to 1.5 m/s).

To represent obstacles at runtime, we use the temporal discretization based approach in §5.7, with $b = 0.15$ m, $b_t = 0.35$, and r and a computed as in Example 5.10. Note, $v_{\text{rel}} = 3.0$ m/s (the relative speed between our robot and obstacles, used to compute the temporal discretization).

Comparison Methods We compare against a State Lattice (SL) approach [McN11] to produce a graph of possible paths at each planning iteration, which attempt to reach the intermediate waypoint generated by the straight line HLP. We search this graph using Lazy SP to minimize the number of collision checks needed [DS16]. We parameterize the output path of SL by time to produce a trajectory plan according to [McN11], and ensure that every such trajectory ends with the robot stopped, to include a braking maneuver. We empirically found that it was necessary to buffer obstacles by 0.43 m to balance performance (reaching goals often) with collisions. We also modified the Segway to use a linear MPC controller, since the tracking controller used for RTD resulted in a large number of collisions. In other words, we improved the Segway's tracking ability, thereby giving SL an advantage over RTD.

Results RTD was able to reach every goal (the environments only have dynamic obstacles, so a feasible path to the goal always exists), with no at-fault collisions. SL, on the other hand, only reached the goal 92.4% of the time, and otherwise caused collisions. We found, however, that RTD caused the robot to travel approximately 0.1 m/s slower than SL on average; in other words, we traded a small amonut of conservatism for the benefit of not colliding with dynamic obstacles. Results are summarized in Table 9.2.

Method	**Goals [%]**	**At-Fault Collisions [%]**
RTD	**100**	**0.0**
SL	92.4	7.6

Table 9.2: Segway simulation/comparison results in 1000 random dynamic environments. RTD outperforms a State Lattice (SL) approach [McN11], and causes no at-fault collisions. RTD outperforms both RRT and NMPC when real-time planning is enforced.

9.1.7 Hardware Demonstration

For static environments, the Segway is run on a 4×8 m^2 tile floor with 30 cm cubical obstacles randomly distributed just before run time. The Segway has no prior knowledge of the obstacles. Two points are picked on opposite ends of the room and used as the start and goal points in an alternating fashion. A video for static environments is available: https://youtu.be/FJns7YpdMXQ.

For dynamic environments, the Segway runs indoors at up to 1.5 m/s in similar scenarios as in simulation. Virtual dynamic obstacles ($v_{\text{max,obs}} = 1$ m/s) are created in MATLAB. The testing area is smaller than the simulation world, so we only test with up to 3 obstacles. The room boundaries are treated as static obstacles. A video for dynamic environments is available: https://youtu.be/9mMZyyLUiPg.

9.2 The Rover Wheeled Robot

The Rover is a front wheel steering, all-wheel drive platform, and demonstrates the utility of RTD in car-like applications. The robot is shown in Figure 1.1. A video is available: https://youtu.be/bgDEAi_Ewfw.

The robot is simulated using our open-source MATLAB simulator [KVL19], with code available [KVV19]. We use $t_{\text{plan}} = 0.5$ s for the receding-horizon planning timeout in simulation.

The hardware is as follows. The Rover has a rectangular body of length 0.5 m and width 0.29 m centered at the center of mass. The distance from the rear axle to the center of mass, l_r, is 0.0765 m. The Rover is equipped with a front-mounted Hokuyo UST-10LX planar lidar for sensing and localization; as the Rover runs indoors, we found this sensor to be accurate up to at least 3.5 m away given occlusions and obstacle density. An NVIDIA TX-1 computer on-board is used to run the sensor drivers, state estimator, feedback controller, and low-level motor controller. The Rover uses ROS [QCG+09] to communicate with a 2.9 GHz CPU with 64 GB of memory over wifi. The laptop is used for localization and mapping and to run RTD's online trajectory optimization. We use $t_{\text{plan}} = 0.375$ s on the hardware, because the robot must navigate a more cluttered environment in hardware than in simulation, so it must plan more often.

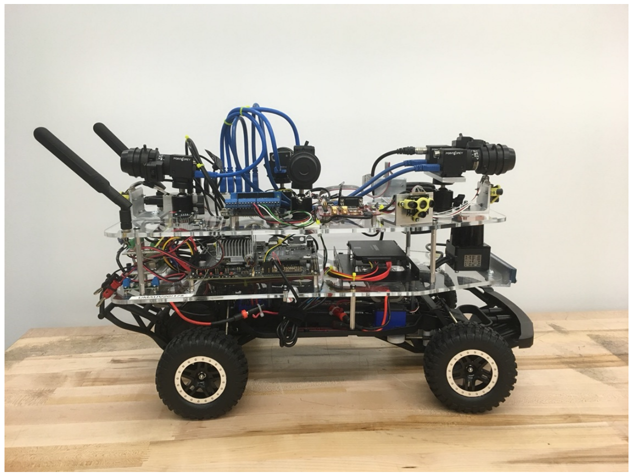


Figure 9.5: The Rover wheeled robot.

9.2.1 High-Fidelity Model

The Rover's high-fidelity model has a state vector $x_{\text{hi}} = (p_1, p_2, \theta, v, \delta)$, where v is longitudinal speed and δ is the angle of the front (steering) wheels. The high-fidelity model f_{hi} is

$$\frac{d}{dt}\begin{bmatrix} p_1 \\ p_2 \\ \theta \\ v \\ \delta \end{bmatrix} = \begin{bmatrix} v\cos(\theta) - \dot{\theta}\cdot(c_1 + c_2 v^2)\sin(\theta) \\ v\sin(\theta) + \dot{\theta}\cdot(c_1 + c_2 v^2)\cos(\theta) \\ \frac{v}{c_3 + c_4 v^2}\tan(\delta) \\ c_5 + c_6\cdot(v - u_1) + c_7\cdot(v - u_1)^2 \\ c_9\cdot(u_2 - \delta) \end{bmatrix}, \tag{9.13}$$

with $u = (u_1, u_2)$ commanding the speed and steering wheel angle. Both v and u_1 are in $[0, 2]$ m/s; both δ and u_2 are in $[-0.5, 0.5]$ rad. This model utilizes steady-state assumptions for the lateral dynamics, but the constants c_2 and c_4 account for wheel slip [Raj11]. Motion capture data was used to fit the constants, $c_1, \cdots, c_9 \in \mathbb{R}$. Note that acceleration bounds are enforced implicitly by the bounds on v, δ, and u.

For the hardware, we use $\varepsilon_{p_1} = \varepsilon_{p_2} = 0.1$ m for the modeling error in the robot's position in the plane, as in Assumption 3.1 (see the robot's states below in the high-fidelity model). We found the error in the other states to be negligible.

9.2.2 Planning Model

The Rover's planning space X is $\mathsf{SE}(2)$, with the planning model f given as

$$\frac{d}{dt}\begin{bmatrix} p_1(t) \\ p_2(t) \\ \theta(t) \end{bmatrix} = \begin{bmatrix} k_3 \cdot (1 - \frac{1}{2}\theta(t)^2) - l_r\omega(t,k) \cdot (\theta(t) - \frac{1}{6}\theta(t)^3) \\ k_3 \cdot (\theta(t) - \frac{1}{6}\theta(t)^3) - l_r\omega(t,k) \cdot (1 - \frac{1}{2}\theta(t)^2) \\ \omega(t,k) \end{bmatrix} \tag{9.14}$$

$$\omega(t,k) = \frac{-1}{2}k_2 t + k_1 \cdot (1 - t). \tag{9.15}$$

We report t_f below, because we compute separate FRSes for a variety

Notice that this trajectory planning model does not explicitly include braking to a stop. However, we incorporate such braking behavior implicitly, by choosing t_f large enough that the robot can stop from its max speed (2 m/s) within the distance traveled along any parameterized trajectory.

9.2.3 Tracking Controller

The Rover uses a PD controller similar to the Segway's. See the exact controller in [KVV19].

9.2.4 Forward Reachable Set

To compute the Rover's FRS, we first compute the ERS as follows. We partition the robot's initial condition space of speed, yaw rate, and heading, as per the ERS computation in §7, into 42 subsets $\dot{Q}^{(j)}$. Each subset spans one of seven evenly-spaced ranges in the steering wheel angle (drawn from $[-0.5, 0.5]$ rad), one of three velocity ranges (0.0–0.75, 0.75–1.5, and 1.5–2.0 m/s), and either positive or negative initial heading. Note, we include heading because the Rover's planning model is specific to a road-like scenario, where the Rover's relative heading to the road's direction of travel determines its possible range of lateral velocities. We then apply Algorithm 3. We store the tracking error function $f_\text{err}^{(j)} \in \mathbb{R}_3[t]$ for each subset of the initial condition space as in §7.3.1 (that is, each $f_\text{err}^{(j)}$ is a degree 3 time-varying polynomial).

For each range of initial speeds, we set t_f separately for the planning model, since the robot can brake to a stop in less time at lower speeds. In particular, we use $t_\mathrm{f} = 1.25$ s for initial speeds of $0.0 - -0.75$ m/s, and $t_\mathrm{f} = 1.5$ s otherwise.

Given these 42 tracking error models, we then use sums-of-squares programming approach in §4.5 to compute 42 FRSes. For each FRS, we use the system decomposition approach to leverage the fact that the Rover's planning model is separable in p_1 and p_2; we solve (4.33) with $l = 4$

for each subsystem. For the decomposed systems, we overapproximate the robot's body with the set X_0 as a 0.58×0.26 m^2 rectangle; this rectangular initial set is decomposed along with the subsystems of the planning model. Given the polynomial $g_{\text{dyn},l}$ for each subsystem, we then reconstruct each full system FRS using (4.50) to find the polynomial $g_{\text{rec},m}$, with $m = 5$.

9.2.5 Simulation in Static Environments

Environment The simulated environment for the Rover is a larger version of the mock road depicted in Figure 1.1. The simulated road is oriented along the high-fidelity model p_1 direction, and is centered at $p_2 = 0$. It is 2.0 m wide (including the shoulder), with two 0.6 m wide lanes centered at $p_2 = 0.3$ m and $p_2 = -0.3$ m. In each trial, three randomly sized box-shaped obstacles of lengths 0.4–0.6 m and widths 0.2–0.3 m are placed in alternating lanes. This obstacle arrangement is used to force the Rover to attempt two lane changes per trial; note that our RTD implementation is not specialized to this particular obstacle arrangement. The obstacles have a random heading of $\pm$ 2 degrees relative to the road, and their centers are allowed to vary by $\pm$ 0.1 m from lane center in the y-dimension. The spacing between the obstacles in the x-direction is given by a normal distribution with a mean of 4 m and standard deviation of 0.6 m. The Rover begins each trial centered in a random lane, with a velocity of 0 m/s.

A trial is considered successful if the Rover crosses a line positioned 30 m after the third obstacle without colliding with any obstacle or road boundary.

High-Level Planner Recall that the Rover operates on a mock road, with two lanes. To this end, the Rover's HLP attempts to place a waypoint a set "lookahead" distance ahead of the robot in the same lane; if the straight line between the robot and this waypoint is in collision with any obstacle, the waypoint is switched to the other lane. For the simulation, we use a lookahead distance of 4 m; on the hardware, we use 1.5 m.

RTD Online Planning To implement RTD at runtime, we use the polynomial $g_{\text{rec},m}$ and the discretized obstacle representation in §5.8.1 to create collision-avoidance constraints in each receding-horizon planning iteration. We use a buffer $b = 0.01$ m for the Rover, resulting in the point spacing $r = 0.02$ m and arc point spacing $a = 0.014$ m as per Example 5.9.

We solve the online trajectory optimization program with MATLAB's `fmincon` nonlinear solver in each planning iteration.

Comparison Methods We compare RTD to the same two other methods as the Segway in §9.1.5. First, an RRT (with heuristics from [KFT$^+$08, PKA16, PLM06]), and NMPC (via GPOPS-

II [PR14]). We test both of these methods with and without the receding-horizon timeout t_{plan} enforced.

Since neither of these methods prescribe how to correctly buffer obstacles to ensure safety, we tested several buffer sizes. We found that buffering obstacles by taking a Minkowski sum of each obstacle with an axis-aligned rectangle of size 0.29×0.26 m^2 provides the best balance of performance and safety for both methods [KVB$^+$20, Experiment 1].

Results RTD reaches nearly as many goals as RRT, but with no collisions; and NMPC cannot reach any goals because it is unable to plan fast enough. We find that RRT and NMPC are able to leverage the structured on-road scenario to display impressive performance in comparison to the random environments of the Segway. However, these methods are not able to certify collision avoidance. Example simulations are shown in Figure 9.6, and the results are summarized in Table 9.3.

Method	Goals [%]	Collisions (%)
RTD[1]	95.4	**0.0**
RRT[1]	97.6	0.1
NMPC[1]	0.0	**0.0**
RRT[2]	**99.8**	0.0
NMPC[2]	99.6	0.0

Table 9.3: Rover simulation/comparison results in 1000 mock-road static environments. Note, [1] indicates that real-time planning (the timeout t_{plan}) was enforced, and [2] indicates that real-time planning was *not* enforced. This distinction is also shown with a dashed line. When real-time planning is enforced, RTD reaches nearly as many goals as RRT, but with no collisions; and NMPC cannot reach any goals because it is unable to plan fast enough.

9.2.6 Hardware Demonstration

The Rover is tested on a 7 m long mock road, which is a tiled surface, as shown in Figure 1.1. This setup resembles the simulation environment, but with a shorter road and smaller obstacles. The robot never crashed. A video is available: https://youtu.be/bgDEAi_Ewfw.

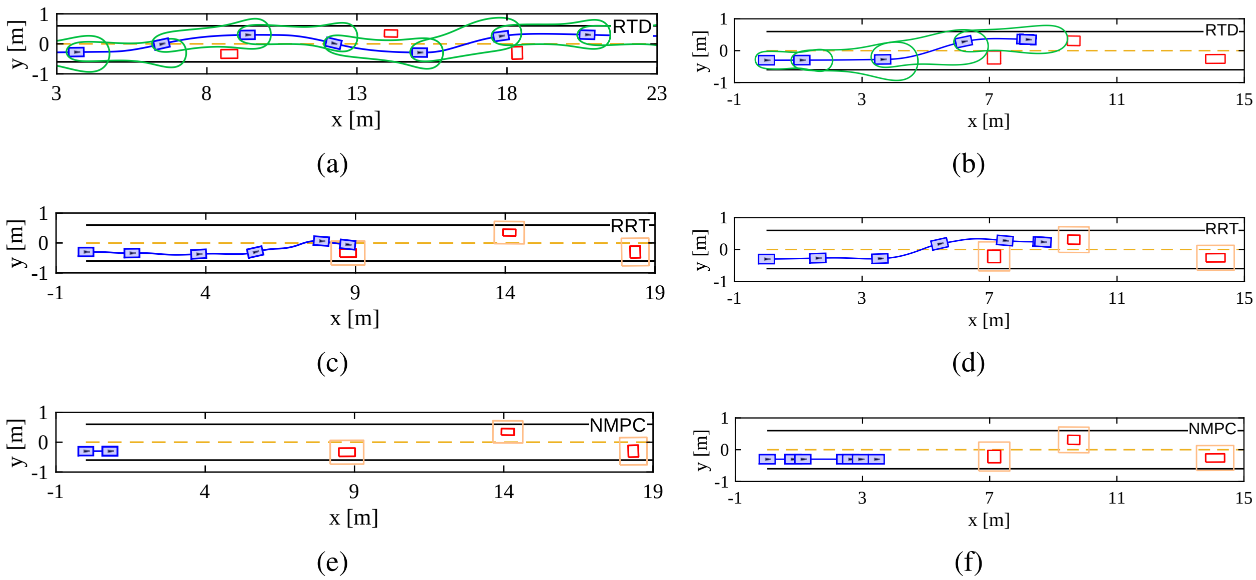


Figure 9.6: Two sample environments from the Rover simulations. The Rover's trajectory, starting from the far left, is a solid line, and its pose at several sample time instances is plotted with solid rectangles. Obstacles are plotted as red boxes. Buffered obstacles for RRT and NMPC are plotted with light solid lines. Subfigures (a) and (b) show RTD avoiding the obstacles. The subset of the FRS associated with the optimal parameter every 1.5 s is plotted as a contour. Subfigures (c) and (d) show the RRT method. In Subfigure (c), RRT is unable to safely track its planned trajectory around the first obstacle. In Subfigure (d), RRT is able to come to a stop before the second obstacle. Subfigures (e) and (f) show NMPC, which stops due to enforcement of real-time planning limits.

9.3 The Fusion Passenger Sedan

The Fusion is a passenger sedan equipped for autonomous driving. This robot was made available to us in the CarSim high-fidelity simulator [Mec18] by the Ford Motor Company. We use this robot to demonstrate RTD planning at much higher speeds (up to 15 m/s) and over a longer total distance per simulation than we did with the Segway or Rover. This shows that RTD enables real-time, safe planning despite complicated tire and powertrain dynamics (the CarSim model has a hybrid powertrain with an automatic transmission). See the robot in Figure 9.7. A video is available: https://youtu.be/lmtki6elFlw.

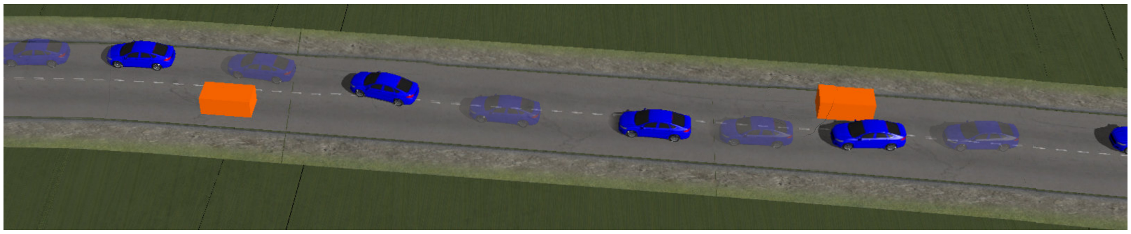


Figure 9.7: The Fusion passenger sedan using RTD to safely and autonomously plan and perform a double lane-change around static obstacles at 15 m/s (which is the speed limit of the road shown). The robot is simulated in the high-fidelity CarSim environment [Mec18], which models the robot's hybrid powertrain and tire dynamics. Using RTD, the robot successfully navigated a 1 km test track, populated with random obstacles, with no collisions.

9.3.1 High-Fidelity Model

We use a bicycle model similar to [LDM15, (1)] as the high-fidelity model:

$$x_{\text{hi}} = \frac{d}{dt}\begin{bmatrix} p_1 \\ p_2 \\ \theta \\ v_1 \\ v_2 \\ \omega \end{bmatrix} = \begin{bmatrix} v_1 \cos\theta - v_2 \sin\theta \\ v_1 \sin\theta + v_2 \cos\theta \\ \omega \\ \frac{1}{m}\tau_1 - \frac{1}{m}\tau_{\text{f},2}\sin\delta + v_2\omega \\ \frac{1}{m}\tau_{\text{f},2}\cos\delta + \frac{1}{m}\tau_{\text{r},2} - v_1\omega \\ \frac{l_{\text{f}}}{I_z}\tau_{\text{f},2}\cos\delta - \frac{l_{\text{r}}}{I_z}\tau_{\text{r},2} \end{bmatrix}, \tag{9.16}$$

where p_1 and p_2 are the position of the robot's center of mass, θ is the robot's heading in the global coordinate frame, v_1 and v_2 are longitudinal and lateral speed of the center of mass, and ω is yaw rate. The constants m, I_z, l_{f}, and l_{r} are the robot's mass, yaw moment of inertia, distance from the front wheel to center of mass, and distance of the rear wheel to center of mass.

The control inputs for this model are the steering wheel angle δ and the force τ_1. We fit polynomials relating the CarSim throttle/brake inputs to the driving force, τ_1, and find a linear relationship between wheel angle, δ, and steering wheel angle. We fit a simplified Pajecka tire model [LDM15, (2a, 2b)] to the tire forces $\tau_{\text{f},2}$ and $\tau_{\text{r},2}$. Since τ_1, $\tau_{\text{f},2}$, and $\tau_{\text{r},2}$ are continuous, this high-fidelity model is continuous.

We found the modeling error, as in Assumption, empirically as $\varepsilon_{x_1} = \varepsilon_{x_2} = 0.1$ m, $\varepsilon_\theta = 0.02$ rad, $\varepsilon_{v_1} = 0.4$ m/s, $\varepsilon_{v_2} = 0.08$ m/s, and $\varepsilon_\omega = 0.05$ rad/s.

9.3.2 Planning Model

The planning model for the Fusion is similar to the Segway's in (9.2), but with a slightly different lateral velocity profile:

$$f(t, x(t;k), k) = \begin{bmatrix} k_1 - k_2 \cdot (p_2(t;k) - p_{2,0}) \\ v_2^*(k) + k_2 \cdot (p_1(t;k) - p_{1,0}) \end{bmatrix} \tag{9.17}$$

$$v_2^*(k) = k_1 \left(l_\text{r} - \frac{m l_\text{f}}{c_\text{r}(l_\text{r} + l_\text{f})} k_2^2 \right) \tag{9.18}$$

Here, k_1 specifies longitudinal speed and k_2 specifies a constant desired yaw rate. The value c_r is the rear cornering stiffness from the tire force model in (9.16). The lateral speed v_2^* is derived from steady-state, linear tire force assumptions [SHB14, Section 10.1.2].

As with the Rover, instead of explicitly including a braking maneuver, we choose t_f empirically to be large enough such that the distance traveled by any parameterized trajectory is longer than the distance required for the robot to brake along that same trajectory.

We implement $\mathcal{K}_\text{lim}$ (i.e., bounds on the choices of trajectory parameters as a function of the robot's initial condition in each planning iteration) as follows. We limit the commanded change in speed to 1 m/s, and the commanded change in yaw rate to 0.25 rad/s. Note, a new speed and yaw rate are commanded every 0.5 s. A more aggressive commanded change could be used, but this was sufficient for RTD to navigate the challenging test track environment reported below.

9.3.3 Tracking Controller

We implement the tracking controller $u_k : T_\text{plan} \times X_\text{hi} \to U$ (for each $k \in K$) using MATLAB's linear MPC toolbox; in other words, we apply a standard linear MPC formulation [GPM89].

9.3.4 Forward Reachable Set

First, we compute the ERS by partitioning the robot's space of generalized initial velocities into intervals of 2 m/s in length, and its yaw rate into intervals of 0.25 rad/s in length, as in §7. We fit a degree 3 time-varying polynomial, $f_\text{err}^{(j)}$, to the worst-case tracking error in each j^th subset of the generalized initial velocity space as in §7.3, and as we did with the Segway and Rover above.

We then compute an FRS for each $f_\text{err}^{(j)}$, using the planning model f above, and the SOS approach in (4.33). In particular, we find the polynomials $g_{\text{dyn},l}$ and $g_{\text{stat},l}$ with $l = 5$.

Note, while the robot travels up to 15 m/s, we scale the planning model and tracking error models as in Remark 4.6 so that every state is within an interval $[-1, 1]$, to ensure numerical stability. A generic method to perform such scaling is available in our open-source tutorial [KV19].

9.3.5 Simulation in Static Environments

Environment The robot runs on a 1.036 km, counter-clockwise, closed loop test track with 7 turns (with approximate curvatures of 0.005–0.04 m^{-1}) and two 4 m wide lanes. Twenty stationary obstacles (with random length of 3.3–5.1 m length and width of 1.7–2.5 m) are distributed around the track in random lanes and randomly spaced 40–55 m apart. We generated ten such random tracks; though the mean obstacle spacing is the same, the tracks vary in difficulty. For example, some tracks require performing overtaking maneuvers in a corner. The robot begins each simulation at the northwest corner of the track in the left lane, with the first obstacle at least 50 m away. A trial is considered successful if the robot completes one lap of the track with no collisions and without leaving the road + shoulder.

High-Level Planner Much like for the Rover, the HLP places waypoints ahead of the vehicle at a *lookahead distance* proportional to the vehicle's current speed. If the lane centerline, from the vehicle's current position and lane to the waypoint, intersects an obstacle, the waypoint is switched to the other lane to encourage a lane change. Lane keeping is not explicitly enforced but is encouraged via the trajectory optimization cost function, which is to minimize the robot's Euclidean distance to the waypoint in each planning iteration.

RTD Online Planning We use the obstacle discretization in §5.8.1, with $g_{\text{stat},l}$ representing the FRS as above, to construct collision-avoidance constraints at runtime. The robot has a rectangular body of size 4.8×1.8 m^2. We choose a buffer size $b = 0.05$ m, so the point spacing is $r = 0.1$ m and the arc point spacing is $a = 0.07$ m, per Example 5.9.

We solve the online trajectory optimization program with MATLAB's `fmincon` nonlinear solver in each planning iteration.

Comparison Methods We compare RTD to the same two methods as the Segway and the Rover (see §9.1.5). First, an RRT (with heuristics from [KFT$^+$08, PKA16, PLM06]), and NMPC (via GPOPS-II [PR14]). We test both of these methods with and without the receding-horizon timeout t_{plan} enforced.

Both methods plan using the robot's high-fidelity model, but this model only represents the robot's center of mass position, so we have to buffer obstacles to compensate for the size of the robot's body. Since neither of these methods prescribe how to buffer obstacles to ensure safety, we found a buffer amount empirically. In particular, for our simulations, obstacles are buffered by 4 m in the robot's direction of travel and 1.25 m in the lane width direction.

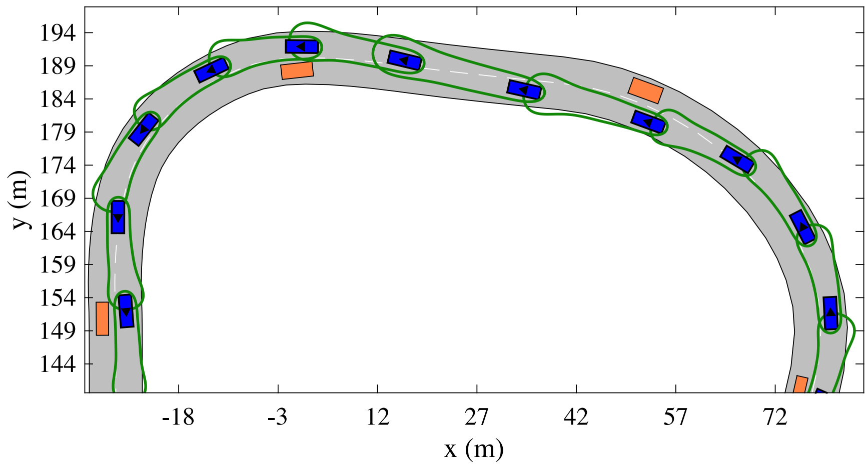


Figure 9.8: The Fusion passenger sedan navigating a section of a 1 km test track using RTD at up to 15 m/s. The robot is plotted every 1.5 s (that is, every third receding-horizon planning iteration, since $t_{\text{plan}} = 0.5$ for this robot); its FRS subset corresponding to each planned trajectory is shown in green, and static obstacles are shown in orange. Since the FRS lies outside of all obstacles, the robot provably avoids collision.

Results RTD has no collisions, as expected, and is able to complete the entire test track in all 10 trials; note, we constructed the trials such that they always have a feasible route around the track, as evidenced by NMPC's success when real-time planning is not enforce. However, RRT and NMPC struggle to plan for the Fusion's high-fidelity model in real time, and therefore instead plan safe stopping maneuvers, resulting in failure to complete a significant portion of the test track. While one could potentially tune the hyperparameters of RRT and NMPC to increase their performance, it is unclear how to do so while guaranteeing safety. The results are summarized in Table 9.4. RTD is shown navigating two corners of the test track in Figure 9.8.

Method	**% of Track Completed**		**Collisions**	**Safe Stops**
	Avg	Max		
RTD[1]	**100**	**100**	**0**	**0**
RRT[1]	13	38	1	9
NMPC[1]	0	0	0	10
RRT[2]	31	86	0	10
NMPC[2]	100	100	0	0

Table 9.4: Fusion simulation/comparison results in 10 trials of a 1 km test track with random static obstacles. Note, [1] indicates that real-time planning (the timeout t_{plan}) was enforced, and [2] indicates that real-time planning was *not* enforced. This distinction is also shown with a dashed line. RTD outperforms both RRT and NMPC because those methods struggle to plan with the robot's high-fidelity model in real time, and instead have to frequently plan safe stopping maneuvers.

9.4 The EV Wheeled Robot

The EV (Electric Vehicle) is a four-wheel-drive, electric, two-passenger, car-like robot. We use this robot much like the Fusion, to demonstrate RTD on a larger-scale passenger vehicle operating in traffic-like scenarios, as well as crowded dynamic environments. In particular, we demonstrate safe unprotected left turns with oncoming traffic. Two videos are available: https://youtu.be/PGBxoPMRvg8 and https://youtu.be/5CD-9qVT3js.

The robot is simulated using our open-source MATLAB simulator [KVL19]. We use $t_{\text{plan}} = 0.5$ s for the receding-horizon planning timeout in simulation.

The hardware is as follows. The EV has a rectangular 2.4×1.3 m^2 body. ROS [QCG$^+$09] runs on-board on a 2.6 GHz computer, enabling access to sensor data and actuator commands. RTD is run in MATLAB on a 3.1 GHz laptop which communicates with the on-board ROS network via ethernet. The EV performs localization with a Robosense RS-Lidar-32 and saved maps [BWWN19].

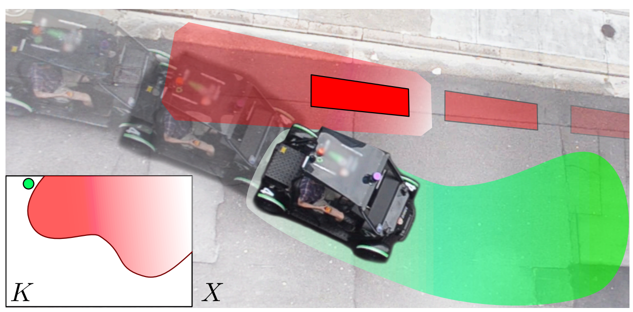


Figure 9.9: An illustration of the EV performing an obstacle avoidance maneuver around a rectangular dynamic obstacle. Past positions of the EV and the obstacle are shown with opacity increasing with time. For the current planning iteration, a prediction of the obstacle is shown fading from light to dark, and the corresponding unsafe trajectory parameters are shown in the inset space K. The EV's particular choice of trajectory plan is shown as a green point in K, and the corresponding subset of the FRS is shown in green fading from light to dark as time passes.

9.4.1 High-Fidelity Model

We use the following high-fidelity model, a single-track bicycle model similar to the Rover and Fusion:

$$\begin{bmatrix} \dot{p}_1 \\ \dot{p}_2 \\ \dot{\theta} \\ \dot{\delta} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos\theta - \dot{\theta}(c_1 + c_2 v^2)\sin\theta \\ v\sin\theta + \dot{\theta}(c_1 + c_2 v^2)\cos\theta \\ \tan(\delta)v(c_3 + c_4 v^2)^{-1} \\ c_5(\delta - u_1) \\ c_6 + c_7(v - u_2) + c_8(v - u_2)^2 \end{bmatrix}, \tag{9.19}$$

where θ is heading, δ is steering angle, and v is speed. We bound the speed to $[0, 3]$ m/s on the hardware and $[0, 5]$ m/s in simulation. Saturation limits are $|\delta(t)| \leq 0.50$ rad, $|\dot{\delta}(t)| \leq 0.50$ rad/s, and $\dot{v}(t) \in [-6.86, 3.50]$ m/s^2. The coefficients $c_1, \cdots, c_8$ are fit to localization data. We find $\varepsilon_{p_1} = \varepsilon_{p_2} = 0.1$ m (as in Assumption 3.1) for the position error, and error in other states is negligible.

9.4.2 Planning Model

We use the following planning model, which is similar to the Segway's in (9.2), but with the time at which the robot comes to a stop allowed to vary. This leads to a time-switched model:

$$f(t, x(t;k), k) = s(t,k)\begin{bmatrix} 1 - \frac{k_1}{l}x_2(t;k) \\ \frac{k_1}{l}x_1(t;k) \end{bmatrix}, \tag{9.20}$$

$$s(t,k) = \begin{cases} v_0 + a_{\text{acc}}t, & t \in T_1(k) \\ k_2, & t \in T_2(k) \\ k_2 - a_{\text{brk}}(t - \tau_1(k) - \tau_2(k)), & t \in T_3(k) \\ 0, & t \in T_4(k). \end{cases} \tag{9.21}$$

with $|k_1| \leq 0.5$ rad, $k_2 \in [0, 5]$ m/s, and $a_{\text{acc}} = a_{\text{brk}} = 2.0$ m/s^2. The final time t_{f} is chosen large enough for the vehicle to come to a stop when tracking any trajectory parameterized by $k \in K$. The time intervals $T_1, \cdots, T_4 : K \to \text{pow}(T_{\text{plan}})$ are

$$T_1(k) = [0,\ \tau_1(k)) \tag{9.22}$$

$$T_2(k) = [\tau_1(k),\ \tau_1(k) + \tau_2(k)), \tag{9.23}$$

$$T_3(k) = [\tau_1(k) + \tau_2(k),\ \tau_1(k) + \tau_2(k) + \tau_3(k)) \tag{9.24}$$

$$T_4(k) = [\tau_1(k) + \tau_2(k) + \tau_3(k),\ \tau_4(k)]. \tag{9.25}$$

The times $\tau_1, \cdots, \tau_4 : K \to [0, t_\mathrm{f}]$ are:

$$\tau_1(k) = \tfrac{k_2 - v_0}{a_\mathrm{acc}}, \tag{9.26}$$
$$\tau_2(k) = k_3, \tag{9.27}$$
$$\tau_3(k) = \tfrac{k_2}{a_\mathrm{brk}}, \tag{9.28}$$
$$\tau_4(k) = t_\mathrm{f}, \tag{9.29}$$

so that $T_\mathrm{plan} = \bigcup_{i=1}^{4} T_i(k)$ for any $k \in K$ by construction. Note that this model explicitly includes a braking maneuver.

To implement $\mathcal{K}_\mathrm{lim}$, in each planning iteration, we limit commanded changes in k_1 (resp. k_2) to 0.1 rad (resp. 0.5 m/s) relative to the previous planning iteration.

9.4.3 Tracking Controller

We use a linear MPC controller to implement u_k in simulation, similar to the Fusion [GPM89]. The hardware has a custom, black-box vector pursuit controller; we estimate its performance by fitting the coefficients in the high-fidelity model.

9.4.4 Forward Reachable Set

We compute two types of FRSes for the EV. The first is for arbitrary scenarios, and has the range of wheel angles and velocity parameters discussed above. The second is for left turns only, which allows validated such maneuvers across the whole intersection; for this, the FRS initial speed and wheel angle are limited to 0–2 m/s and -0.1 to 0.1 rad, respectively, and the time t_f is large enough to cross an entire intersection and then brake safely to a stop (e.g., 7 s).

We follow the ERS computation procedure in §7.3.1. We break the space of initial velocity and initial steering angle into 35 (that is, 5 velocity ranges and 7 wheel angle ranges) evenly-spaced subsets, and compute the ERS and FRS on each subset. We follow the FRS swapping procedure at runtime as in §4.7.

We compute the FRS over time intervals as in §4.6. Notice that the time intervals in (9.22) are k-dependent; while the formulation in §4.6 does not explicitly state that such k-dependence is possible, in fact the SOS program (4.59) is able to accomodate this type of k-dependence without alteration.

9.4.5 Simulation in Dynamic Environments

Environment We test two scenarios.

The first scenario is a 60×10 m^2 open area with 1–10 0.3×0.3 m^2 dynamic box-shaped obstacles moving along random paths up to 2 m/s. We created 1000 such scenarios.

The second scenario requires an unprotected left turn at a 4-way intersection, followed by driving straight for 30 m (see Figure 9.10). We created 100 random scenarios with lane widths and corner radii of 3.5–4.0 m. At any time, up to 4 obstacle cars (length 2.5–4.0 m and width 1.25–2 m) travel along randomly chosen lanes, and randomly choose to turn, at constant speeds of up to 7 m/s. Up to 2 pedestrians randomly cross one of the four cross walks up to 2 m/s. The ego vehicle starts in the right lane either at the intersection or 30 m away, with initial speed and wheel angle of 0. We created 100 such scenarios.

High-Level Planner In the first set of scenarios, the robot uses the same straight-line HLP as the Segway. That is, it attempts to minimize its distance to a waypoint along a straight line between itself and the goal in each planning iteration.

In the second set of scenarios (left turns), the robot attempts to reach a waypoint placed manually in the lane center where it should arrive after completing a left turn; from it uses the lane centerline to generate waypoints. In other words, the HLP returns waypoints based on the geometry of the road. As usual, the cost function in each planning iteration is the Euclidean distance to the waypoint, either at t_{f} for the left turns, or at t_{plan} for driving in a lane.

RTD Online Planning We test both dynamic obstacle discretizations presented in §5.7, with the point spacings computed as in Example 5.9. For the time discretization approach, we use $b_t = 0.35$ m and $b = 0.1$ m. For the time interval approach, we use $b = 0.1$ m.

As with the Segway, Rover, and Fusion, we create collision-avoidance constraints as in §4.8. We solve the online trajectory optimization program, subject to these constraints, with MATLAB's `fmincon` nonlinear solver in each planning iteration.

Comparison Methods For the first set of simulated scenarios, we compare RTD against itself, with the two different dynamic obstacle discretization methods from §5.7. We also compare against the State Lattice (SL) approach in [McN11], which was used for the Segway as well (see §9.1.5).

For the second set of simulated scenarios (left turns), we again compare RTD against itself, with the two different dynamic obstacle discretization methods. We also compare against a standard linear MPC controller [GPM89] performing feedback around a hand-crafted left turn trajectory (in other words, we give the comparison method a distinct advantage over RTD by handing it a trajectory *a priori*). While the robot is stopped waiting to begin the left turn, we collision check the entire left turn trajectory at discrete times (every 0.01 s), and do not begin tracking it until the entire trajectory is collision free with respect to all obstacles. This ensures a fair comparison, since

RTD seeks to validate the entire left turn before it begins the maneuver.

Results As expected, RTD produces no at-fault collisions in any scenario. In the first set of scenarios (random scenarios), the time interval version of RTD outperforms both the time discretization RTD and the SL approach. In the second set of scenarios (left turns), the time interval RTD formulation outperforms the time discretization formulation and linear MPC; however, it usually takes 6 more seconds to complete the left turn than linear MPC. We find that this is due to the linear MPC approach determining that the entire left turn is feasible much earlier than RTD, and beginning the maneuver only to result in a collision; in other words, RTD takes on some conservatism to ensure that the entire maneuver is actually feasible before execution.

In general, we see that the time discretization RTD formulation is much more conservative than the time interval formulation. This is expected, because the time discretization formulation requires the additional temporal buffer b_t (i.e., obstacles are treated as larger, which reduces the free space available for planning), and because the robot must consider many more constraints per planning iteration; this means it is more likely to be unable to find a new trajectory in each planning iteration, and instead must safely stop by continuing a previously-found trajectory.

The results are summarized in Table 9.5. See Figure 9.10 for an example left turn.

World	Method	Goals [%]	AFC [%]	ATTG [s]	AS [m/s]
random	RTD (disc)	90.7	**0.0**	41.1	1.99
	RTD (int)	**96.8**	**0.0**	**24.1**	**3.06**
	SL	77.3	17.2	28.1	2.88
left turn	RTD (disc)	91.0	**0.0**	49.9	1.50
	RTD (int)	**99.0**	**0.0**	20.9	2.71
	Linear MPC	80.0	19.0	**14.9**	**3.35**

Table 9.5: EV simulation/comparison results in 1000 random scenarios, and 100 left turn scenarios. RTD is treated with two different methods of representing obstacles. First the time discretization method (disc), and second, the time interval method (int). We also compare against a State Lattice (SL) method [McN11] in the random scenarios, and a generic linear MPC method [GPM89] in the left turn scenarios. We compare the percentage of goals reached, the percentage of trials that had at-fault collisions (AFC), the average time taken to reach the goal (ATTG), and the average speed (AS). Note, the average speed for the left turns appears low because the robot begins stopped, and must wait until it finds an entire feasible left turn trajectory, then must accelerate to 5 m/s to navigate through the intersection. RTD never causes an at-fault collision, as expected. In the random scenarios, the time interval RTD formulation reaches the most goals, in the shortest time, with the highest average speed. In the left turn scenarios, the time interval formulation reaches the most goals by taking on slightly more conservatism than the linear MPC approach, which is aggressive (hence its lowest time to goal and highest average speed) at the expense of causing collisions.

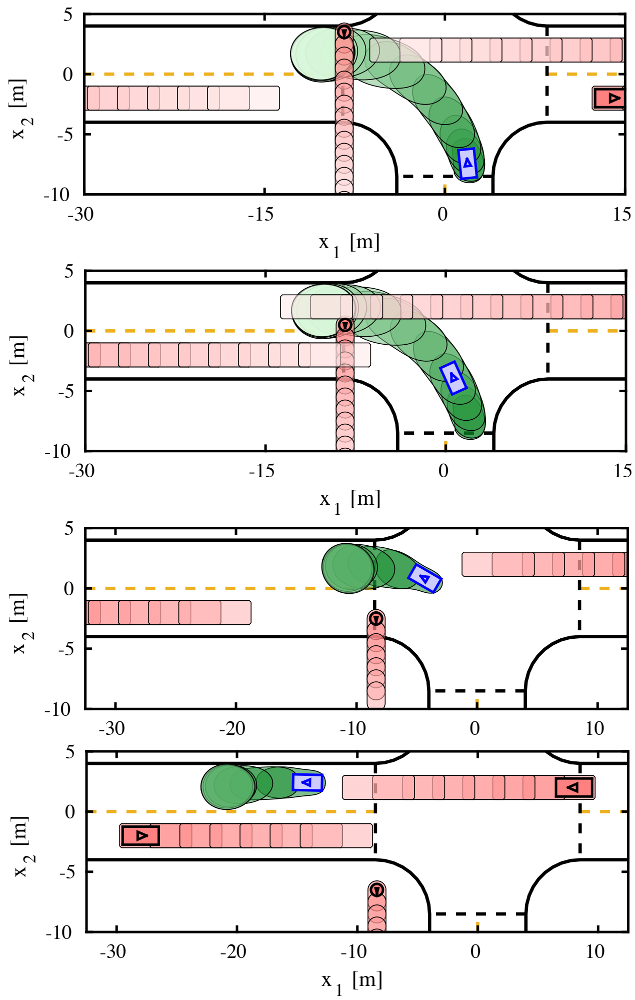


Figure 9.10: Timelapse of EV (blue) completing a left turn. Figures show time at 0.0, 2.0, 3.0, and 5.0 s from top to bottom. Obstacles and their prediction are plotted in red. The vehicle obstacles are traveling at 5 m/s. The pedestrian is traveling at 2 m/s. The EV begins the scenario stopped at the intersection. The FRS intervals are shown in green. Obstacle predictions and the FRS intervals fade from dark to light with increasing time. The left turn maneuver is longer in duration, and therefore requires longer predictions, than the driving-straight maneuvers (which begin after the ego vehicle completes the turn at $t = 3.0$ s).

9.4.6 Hardware Demonstration

The EV runs outdoors in a large open area at up to 3 m/s, with a safety driver.

First, we tested an 80 m stretch of open pedestrian walkway, populated with static concrete cube-shaped obstacles. The EV successfully navigates this scenario at 3 m/s with no collisions. See the video: `https://youtu.be/5CD-9qVT3js`.

Second, we tested more structured, car-like scenarios, to show a variety of overtake maneuvers. We used an open 60 m area, which is large enough that we do not consider static obstacles. Virtual obstacles ($v_{\text{max,obs}} = 1.5$ m/s) resembling people or cyclists are created in MATLAB. The robot had no collisions. See the video: https://youtu.be/PGBxoPMRvg8.

9.5 The Hummingbird Quadrotor

We use a simulated Hummingbird quadrotor [Asc19] to demonstrate RTD on aerial robots using the zonotope FRS. This implementation also demonstrates the utility of the ERS. We compare RTD with and without trajectory-dependent tracking error, and see that, by incorporating trajectory dependence, the robot performs with much lower conservatism. A video is available: https://youtu.be/toFpIC7Zh18.

The robot is simulated using our open-source MATLAB simulator [KVL19]. We use $t_{\text{plan}} = 0.75$ s for the receding-horizon planning timeout.

9.5.1 High-Fidelity Model

The state space is $X_{\text{hi}} = P \times V \times \Omega \times \mathsf{SO}(3)$ with state $x_{\text{hi}} = (p, v, \omega, R)$, where $p \in P \subset \mathbb{R}^3$ is position in the inertial frame; $v \in V \subset \mathbb{R}^3$ is velocity; $\omega \in \Omega \subset \mathbb{R}^3$ is angular velocity; and $R \in \mathsf{SO}(3)$ is attitude. The inertial frame P is spanned by unit vectors denoted e_1, e_2, and e_3 with e_3 pointing "up" relative to the ground, so Re_3 is the net thrust direction of the quadrotor's body-fixed frame. We write the model as per [LLM10]:

$$\begin{aligned}
\dot{p} &= v \\
\dot{v} &= \tau R e_3 - m g e_3 \\
\dot{\omega} &= J^{-1}\left(\mu - \omega \times J\omega\right) \\
\dot{R} &= R\hat{\omega},
\end{aligned} \tag{9.30}$$

where $\hat{\cdot} : \mathbb{R}^3 \to \mathfrak{so}(3)$ is the hat map that maps a 3D vector to a skew-symmetric matrix [LLM10] (i.e., $\mathfrak{so}(n)$ denotes the tangent space to $\mathsf{SO}(3)$). The constant $\mathrm{g} = 9.81$ m/s^2 is acceleration due to gravity. The quadrotor's mass is $m \in \mathbb{R}$, and its moment of inertia matrix is $J \in \mathbb{R}^{3\times3}$. We assume J is diagonal and constant, and can be written $J = \text{diag}(j_1, j_2, j_3)$. The control input is $u = (\tau, \mu) \in U \subset \mathbb{R}^4$, where $\tau \in \mathbb{R}$ is net thrust and $\mu \in \mathbb{R}^3$ is the body moment; these inputs are related to rotor speeds as:

$$\begin{bmatrix} \tau \\ \mu \end{bmatrix} = \begin{bmatrix} k_\tau & k_\tau & k_\tau & k_\tau \\ 0 & k_\tau \ell & 0 & -k_\tau \ell \\ -k_\tau \ell & 0 & k_\tau \ell & 0 \\ k_\mu & -k_\mu & k_\mu & -k_\mu \end{bmatrix} \begin{bmatrix} \omega_{\text{rot},1}^2 \\ \omega_{\text{rot},2}^2 \\ \omega_{\text{rot},3}^2 \\ \omega_{\text{rot},4}^2 \end{bmatrix}, \tag{9.31}$$

where k_τ and k_μ are rotor parameters, ℓ is the length from quadrotor center of mass to each rotor center, and $\omega_{\text{rot},i}$ is the speed of the i^{th} rotor [PMK$^+$13, LLM10]. We assume commanded inputs can be achieved instantaneously (i.e., the rotor dynamics are fast compared to (9.30)), but that rotor speed is bounded (i.e., the inputs can saturate) [LLM10, MK11, MHD15]. We set the quadrotor's max speed as $v_{\text{max}} > 5$ in any direction, since aerodynamic drag can be compensated by rotor thrust up to 6 m/s [TK20, HHWT11].

Since the Hummingbird is only used in simulation, we do not consider any modeling error as in Assumption 3.1.

We implement (9.30) with the specifications of an AscTec Hummingbird [Asc19, DGZD15] (see Table 9.6).

The quadrotor's high-fidelity model (9.30) is simulated by Euler integration with a 5 ms time step; the rotation matrix dynamics are implemented as Lie-Euler integration on $\mathsf{SO}(3)$ as in [CMO14, (7)] with $F_{y_n} = \hat{\omega}_n$. This was done to avoid Euler angle singularities. Euler integration was found empirically to match a Runge-Kutta/Munthe-Kaas 4$^{\text{th}}$ order method within millimeters in the quadrotor's position dimensions over the time horizon T_{plan}, while taking approximately 25% of the computation time. We include the numerical integration error as tracking error in the computation of the ERS below.

Robot [Asc19, DGZD15]		**Control** [MK11]		**Desired Traj.** [MHD15]	
Param.	**Value**	**Param.**	**Value**	**Param.**	**Value**
m	0.547 kg	G_x	$2.00 I_{3\times3}$	t_{plan}	0.75 s
j_1, j_2	0.0033 kgm^2	G_v	$0.50 I_{3\times3}$	t_{peak}	1 s
j_3	0.0058 kgm^2	G_R	$1.00 I_{3\times3}$	t_{f}	3 s
k_τ	1.5E-7 $\frac{\text{N}}{\text{rpm}^2}$	G_ω	$0.03 I_{3\times3}$	$\kappa_v^\pm$	$\pm$5 m/s
k_μ	3.75E-9 $\frac{\text{Nm}}{\text{rpm}^2}$	v_{max}	5 m/s	$\kappa_a^\pm$	$\pm$10 m/s^2
ℓ	0.27 m	a_{max}	3 m/s^2	$\kappa_{\text{peak}}^\pm$	$\pm$5 m/s
ω_{rot}	1100–8600 rpm	d_{sense}	12 m		

Table 9.6: Hummingbird implementation parameters

9.5.2 Planning Model

The planning space for this robot is $X = P$ (i.e., the position subspace of the high-fidelity model). We use a planning model that generates desired position trajectories with polynomials in time, generated separately in each coordinate of X, based on [MHD15], but modified so each trajectory has two piecewise polynomial segments, to include a fail-safe maneuver. We first present a 1D model, then extend it to 3D. Model parameters are in Table 9.6.

Consider a 1D, twice-differentiable, desired position trajectory $p_{\text{des}} : T_{\text{plan}} \to \mathbb{R}$, given by a planning model $f_{\text{1D}} : T_{\text{plan}} \times K_{\text{1D}} \to \mathbb{R}$:

$$\dot{p}_{\text{des}}(t;\kappa) = f_{\text{1D}}(t,\kappa) = \frac{c_1(t,\kappa)}{6}t^3 + \frac{c_2(t,\kappa)}{2}t^2 + \kappa_a t + \kappa_v, \tag{9.32}$$

where $\kappa_a = \ddot{p}_{\text{des}}(0)$ is the initial desired acceleration, $\kappa_v = \dot{p}_{\text{des}}(0)$ is the initial desired speed, and κ_{peak} is a desired peak speed to be achieved at a time $t_{\text{peak}} \in [t_{\text{plan}}, t_{\text{f}}]$. The values of c_1, c_2 are given by [MHD15, (64)] as

$$\begin{bmatrix} c_1(t,\kappa) \\ c_2(t,\kappa) \end{bmatrix} = \frac{1}{(c_3(t))^3} \begin{bmatrix} -12 & 6c_3(t) \\ 6c_3(t) & -2(c_3(t))^2 \end{bmatrix} \begin{bmatrix} \Delta_v(t,\kappa) \\ \Delta_a(t,\kappa) \end{bmatrix}, \tag{9.33}$$

$$c_3(t) = \begin{cases} t_{\text{peak}} & t \in [0, t_{\text{peak}}) \\ t_{\text{f}} - t_{\text{peak}} & t \in [t_{\text{peak}}, t_{\text{f}}], \end{cases} \tag{9.34}$$

$$\Delta_v(t,\kappa) = \begin{cases} \kappa_{\text{peak}} - \kappa_v - \kappa_a t_{\text{peak}} & t \in [0, t_{\text{peak}}) \\ -\kappa_{\text{peak}} & t \in [t_{\text{peak}}, t_{\text{f}}], \end{cases} \tag{9.35}$$

$$\Delta_a(t,\kappa) = \begin{cases} -\kappa_a & t \in [0, t_{\text{peak}}) \\ 0 & t \in [t_{\text{peak}}, t_{\text{f}}]. \end{cases} \tag{9.36}$$

This model produces a desired position trajectory that begins at the speed κ_v with acceleration κ_a at $t = 0$. The trajectory accelerates to a speed of κ_{peak} at $t = t_{\text{peak}}$, at which point the desired acceleration is 0; the trajectory then slows down to desired speed and acceleration of 0 at $t = t_{\text{f}}$ (this is the fail-safe maneuver). Notice that c_3, Δ_v, and Δ_a are piecewise constant in t, with a jump discontinuity at t_{peak}. Therefore, c_1 and c_2 are piecewise constant in t, which makes (9.32) a piecewise polynomial in time. By construction, (9.32) and its derivative (acceleration) are continuous functions of time. Note, a desired position trajectory can be translated arbitrarily, so we assume that the planning frame is centered at $p_{\text{des}}(0) = 0 \in X_{\text{1D}}$. Then, any desired position trajectory given by (9.32) is uniquely determined by κ for all $t \in T_{\text{plan}}$.

Note, we specify that κ_v, κ_a, and κ_{peak} lie in compact intervals $[\kappa_v^-, \kappa_v^+]$, $[\kappa_a^-, \kappa_a^+]$, and $[\kappa_{\text{peak}}^-, \kappa_{\text{peak}}^+]$, so $K_{1\text{D}}$ is the Cartesian product of these three intervals. The lower and upper bounds are reported in Table 9.6.

We now make a 3D planning model by using the model (9.32) for each dimension, and creating a larger parameter space $K = K_{1\text{D}} \times K_{1\text{D}} \times K_{1\text{D}} \subset \mathbb{R}^9$. For a trajectory $x_{\text{des}} : T_{\text{plan}} \to X$, we denote the model as $f : T \times K \to \mathbb{R}^3$ for which

$$f(t,k) = \begin{bmatrix} f_{1\text{D}}(t,\kappa_1) \\ f_{1\text{D}}(t,\kappa_2) \\ f_{1\text{D}}(t,\kappa_3) \end{bmatrix}, \tag{9.37}$$

with trajectory parameter $k = (\kappa_1, \kappa_2, \kappa_3) \in K$, where each $\kappa_i = (\kappa_{v,i}, \kappa_{a,i}, \kappa_{\text{peak},i})$ is the peak speed, initial speed, and initial acceleration in dimension $i = 1, 2, 3$. As in the 1-D case, WLOG we let $x_{\text{des}}(0) = 0$. For notational purposes, let $k_{\text{peak}} = (\kappa_{\text{pk},1}, \kappa_{\text{pk},2}, \kappa_{\text{pk},3})$ and similarly for k_v and k_a. Then $k = (k_v, k_a, k_{\text{peak}})$ by reordering, and we denote $K = K_v \times K_a \times K_{\text{peak}}$. By construction, (9.37) includes a fail-safe (braking) maneuver.

We create $\mathcal{K}_{\text{lim}}$ (i.e., bounds on which k can be chosen at each planning iteration) as follows. First recall that the drone's speed is bounded: $\|k_{\text{peak}}\|_2 \leq v_{\text{max}}$. Second, since k_{peak} is a desired velocity and k_v is the initial velocity, the quantity $\frac{1}{t_{\text{peak}}} \|k_{\text{peak}} - k_v\|_2$ determines an approximate desired acceleration $a_{\text{max}} > 0$. Therefore, for an initial condition $x_{\text{hi},0}$, $\mathcal{K}_{\text{lim}}(x_{\text{hi},0})$ returns all k_{peak} for which $\frac{1}{t_{\text{peak}}} \|k_{\text{peak}} - k_v\|_2 \leq a_{\text{max}}$.

Note that acceleration due to gravity is not included in the planning model. However, gravity is accounted for by the low-level controller specified next.

9.5.3 Tracking Controller

We use a tracking controller based on [MK11]. The control input $u_k(t, x_{\text{hi}}(t;k)) = (\tau(t), \mu(t))$ is given by

$$\begin{aligned} \tau(t) &= \|-G_x e_x(t) - G_v e_v(t) + mge_3 + m\ddot{x}_{\text{des}}(t)\|_2 \\ \mu(t) &= -G_\omega e_\omega(t) - G_R e_r(t) \end{aligned} \tag{9.38}$$

where R_{des} is found as in [MK11, Section IV] and ω_{des} is found as in [MK11, Section III]. In simulation, τ and μ are converted to rotor speeds and saturated using (9.31). At any time t, the state error used for feedback is

$$\begin{aligned} e_x(t) &= x(t) - x_{\text{des}}(t) \\ e_v(t) &= v(t) - \dot{x}_{\text{des}}(t) \\ e_R(t) &= \frac{1}{2}\left(R_{\text{des}}(t)^\top R(t) - R(t)^\top R_{\text{des}}(t)\right)^\vee \\ e_\omega(t) &= \omega(t) - \omega_{\text{des}}(t), \end{aligned} \tag{9.39}$$

where $(\cdot)^\vee : \mathfrak{so}(3) \to \mathbb{R}^3$ is the **vee map** that maps a skew-symmetric matrix to a 3D vector [LLM10]. The feedback gains and rotor speed saturation parameters are reported in Table 9.6.

Note that, by including feedforward terms for angular acceleration and fulfilling other mild assumptions, one can modify (9.38) to provably asymptotically drive tracking error to zero as time tends to infinity for any particular reference trajectory [LLM10]; however, since we are planning in a receding-horizon way, we find that (9.38) tracks trajectories with low error over the time horizon T_{plan} when commanding speeds up to $v_{\max} = 5$ m/s and accelerations of $a_{\max} = 3$ m/s^2.

9.5.4 Forward Reachable Set

We compute a zonotope FRS using the method in §6; that is the FRS is the PRS, plus the ERS, plus the body.

We use the open-source toolbox [Alt15] to compute a zonotope PRS for the planning model (9.37). We use a time discretization of $\Delta_t = 0.02$ s to create the partition of time needed for the zonotope PRS computation; that is, since $t_{\text{f}} = 3$ s, we have $n_{\text{RS}} = 150$.

We represent the ERS with the method in §7.3.2. To do so, we partition the drone's initial velocity space into equally-sized boxes (in each of the three velocity directions) of side length 0.7 m/s, which results in approximately 103,000 subsets of $\dot{Q}$ (the total number of error zonotopes is 103,000 times n_{RS}). Note, we do not partition the drone's angular velocity subspace. This is because the drone's rotation dynamics are, by assumption, much faster than its translation dynamics [HHWT11]; in other words, the drone can rapidly rotate itself to point its thrust vector in any desired direction. Furthermore, by treating the initial attitude matrix R as the identity for every iteration of Algorithm 3, we are in fact making a conservative assumption that the drone's initial rotation is not necessarily aligned with its trajectory plan. It takes approximately 1 hr to run Algorithm 3 on a 3.1 GHz laptop. The error zonotopes are stored in a lookup table of approximately 8.6 MB.

We represent the body as a zonotope of size $0.55 \times 0.55 \times 0.55$, which is large enough to include the robot (of dimensions $0.54 \times 0.54 \times 0.0855$ m^3 [Asc19, DGZD15]) across all body attitudes that we see in simulation.

9.5.5 Simulation in Static Environments

Environment We simulate 500 cluttered worlds with 120 random static obstacles each, plus obstacles representing the world boundaries. An example simulation is shown in Figures 9.11 and 9.12. Each world is $80 \times 20 \times 10$ m in volume, with a random start location at one end and a random goal location at the other.

High-Level Planner We use a straight-line HLP similar to the wheeled robots in dynamic environments reported above. That is, in each receding-horizon planning iteration, the drone attempts to reach a waypoint along a straight line between itself and the global goal. This waypoint is 1.5 m ahead of the robot, plus a distance proportional to half its current speed. The trajectory optimization cost function is to minimize the Euclidean distance to the waypoint at the time t_{peak}.

RTD Online Planning All obstacles are represented as cuboid zonotopes. We use these zonotopes to generate collision-avoidance constraints as in §6.4.

Instead of using `fmincon` to solve the online trajectory optimization, we use a sampling-based approach. That is, in each receding-horizon iteration, we sample approximately randomly-generated 20,000 choices of $k \in K$, evaluate the collision-avoidance constraints on each one, and then evaluate the cost function on each feasible remaining k. We find that this is faster than using `fmincon`, at the expense of slightly suboptimal performance with respect to the cost function.

Note that the trajectory parameters k_v and k_a are fixed by the drone's initial condition at the beginning of each planning iteration. Therefore, we only need to sample in the space of possible peak speeds, k_{peak}.

Comparison Methods We compare RTD against itself, with and without the ERS. The zonotope ERS representation allows us to incorporate trajectory-dependent tracking error. As a comparison, we treat tracking error as 0.1 m uniformly in every direction; that is, the ERS zonotopes $Z_{\text{err}}^{(i,j)}$ are cubes of 0.2 m to a side for all i and j (that is, for all times and initial conditions). This size of constant tracking error was found by taking the maximum of all tracking error in any direction when computing the ERS with Algorithm 3.

Results RTD had no collisions with either tracking error approach, as expected. With the trajectory-dependent tracking error, the robot reached the goal in 91.2% of trials (and otherwise stopped safely). With the constant tracking error, the robot reached the goal in only 84.8% of trials. Note, we did not expect 100% of goals reached, since the trials used randomly-generated obstacles, so some simulated worlds have no feasible path between start and goal. This result confirms that including trajectory-dependent tracking error reduces conservatism.

Figure 9.11 shows RTD in a single receding-horizon planning iteration. Figure 9.12 shows an entire example simulation world, and a trajectory planned (iteratively) by RTD from start to goal; the same planning iteration as in Figure 9.11 is shown here, but zoomed out.

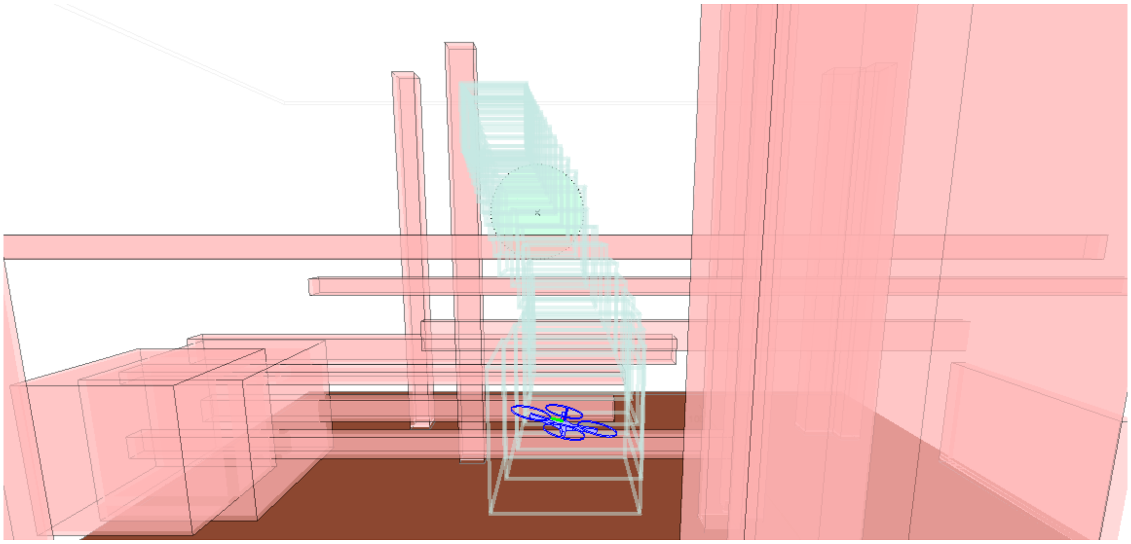


Figure 9.11: An example trajectory planned online in a cluttered environment with obstacles in light red and the ground in brown. The tube of light blue boxes, which does not intersect any obstacles, is the subset of the zonotope FRS for the current plan plus tracking error, so the quadrotor (in dark blue) is guaranteed to fly within the tube. The world and trajectory are shown in Figure 9.12.

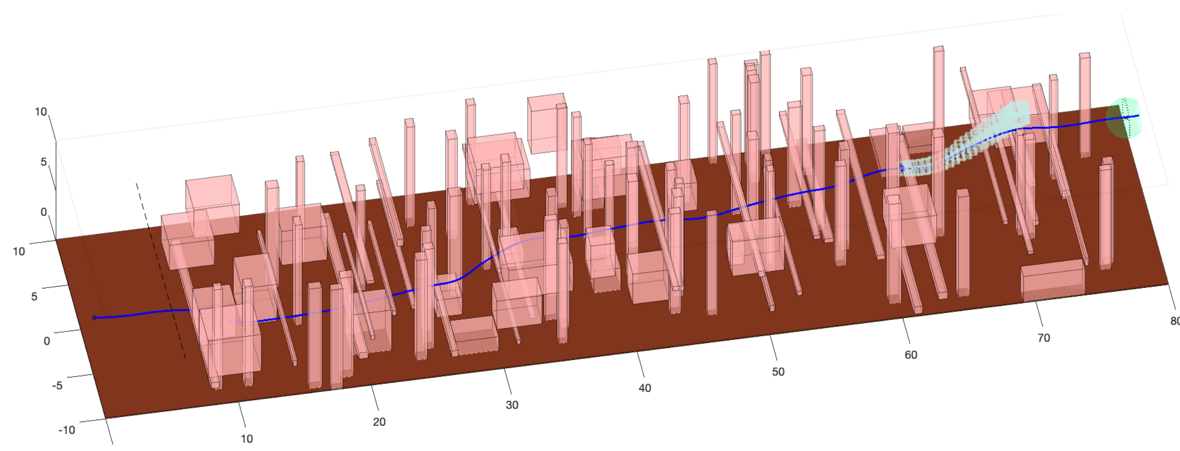


Figure 9.12: The example simulated world from Figure 9.11, with obstacles in light red, the ground in brown, world boundaries as axes, and the global goal as a light green sphere. A trajectory of the quadrotor is shown in dark blue, and goes from left to right. The quadrotor's reachable set (light blue) is shown for the same planning iteration as in Figure 9.11.

9.6 The Mambo Quadrotor

We use a Parrot Mambo microdrone [Par19] to demonstrate RTD on aerial robot hardware. This extends the work from the Hummingbird in §9.5 to include (1) aerodynamic drag, (2) ground effect, and (3) dynamic obstacles. A video is available: https://youtu.be/1cldHVQK3Yw.

The robot is simulated using our open-source MATLAB simulator [KVL19]. We use $t_{\text{plan}} = 0.5$ s for the receding-horizon planning timeout.

The hardware is as follows. The drone has a mass of 63 g with motion capture markers included. Its body fits within a cube of size $0.2 \times 0.2 \times 0.2$ m^3. State estimation is provided by a PhaseSpace Impulse X2E motion capture system. We send commands to the drone (see the high-fidelity model control inputs below) over Bluetooth at approximately 10 Hz, using PyParrot [McG19]; the drone tracks these commands at approximately 100 Hz with an on-board proprietary controller.

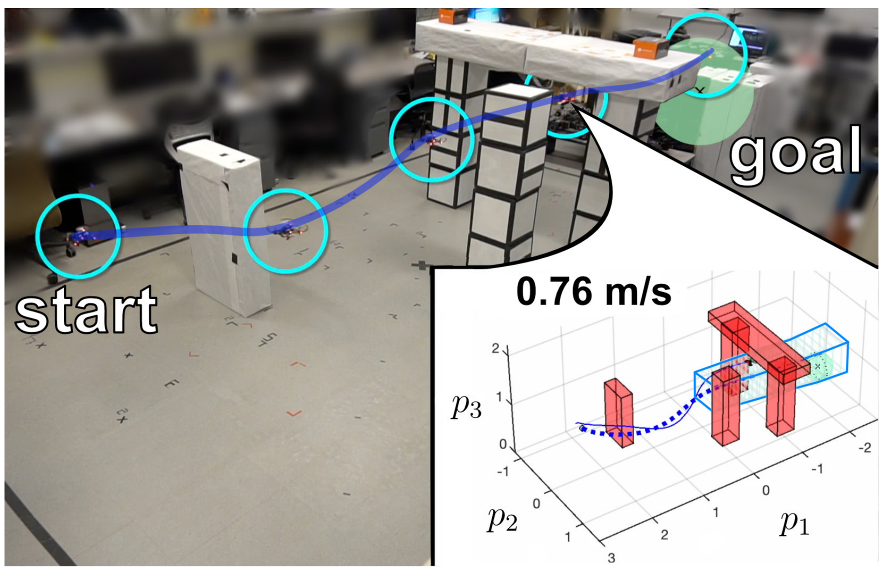


Figure 9.13: The Parrot Mambo navigates around static obstacles to reach a global goal (green sphere on the right) without collision despite tracking error. The callout in the bottom right shows the drone's planned trajectory (dashed blue), realized trajectory (solid blue, also overlaid in the photo), and current speed. The blue box is the FRS corresponding to the plan at the time shown, composed of a sequence of zonotopes, all of which lie outside of the obstacles thereby ensuring collision avoidance.

9.6.1 High-Fidelity Model

We model the drone as a 3-DOF point mass (it translates, but does not rotate), with state space $X_{\text{hi}} = P \times V_1 \times V_2$ (we treat vertical velocity V_3 as an input, because the hardware is built this way

[Par19]). The Mambo has state $x = (p_1, p_2, p_3, v_1, v_2)$ for which

$$\dot{p}_1 = v_1 \tag{9.40a}$$

$$\dot{p}_2 = v_2 \tag{9.40b}$$

$$\dot{p}_3 = c_1 u_{v_3} + c_2 (u_{\mathrm{p}}^2 + u_{\mathrm{r}}^2)^{\frac{1}{2}} + c_3 \exp(c_4 p_3) \tag{9.40c}$$

$$\dot{v}_1 = c_5 \sin(u_{\mathrm{p}}) + c_6 \sin(u_{\mathrm{r}}) + c_7 |v_1| v_1 \tag{9.40d}$$

$$\dot{v}_2 = c_8 \sin(u_{\mathrm{p}}) + c_9 \sin(u_{\mathrm{r}}) + c_{10} |v_2| v_2. \tag{9.40e}$$

The scalars $c_1, \cdots, c_{10} \in \mathbb{R}$ are model coefficients obtained from system identification. The inputs u_{p}, u_{r}, and u_{v_3} are discussed below. The terms in (9.40c), from left to right, represent the commanded vertical speed, the reduction in vertical speed due to pitch and roll, and ground effect. For this model, the ground plane is at $p_3 = 0$. In (9.40d) and (9.40e), from left to right, the terms represent acceleration due to pitch, acceleration due to roll, and aerodynamic drag.

The drone's control inputs are $u = (u_{\mathrm{p}}, u_{\mathrm{r}}, u_{\mathrm{yr}}, u_{v_3}) \in U = [-1, 1]^4$, where u_{p} is pitch, u_{r} is roll, u_{yr} is yaw rate, and u_{v_3} is vertical speed. Notice that each control input lies within $[-1, 1]$, and is scaled to the appropriate units by the drone's on-board flight controller. The yaw rate, roll, and pitch commands are with respect to the 3-2-1 (yaw-pitch-roll) convention for converting the drone's attitude to Euler angles. Note that, based on the 3-2-1 Euler angle order, *positive* pitch causes acceleration in the $+e_1$ direction, but *negative* roll causes acceleration in the $+e_2$ direction (hence $c_6 > 0$ but $c_8 < 0$).

Our model is unusual in that we model acceleration in the plane $P_1 \times P_2$, but speed in P_3. This is because the Mambo's flight controller accepts vertical speed commands instead of thrust, resulting in (9.40c) fitting the flight data well. For this model, we numerically implement the velocity projection operator as $\mathrm{proj}_V : X_{\mathrm{hi}} \times U \to V$ for which $\mathrm{proj}_V(x_{\mathrm{hi}}, u) = (v_1, v_2, \dot{p}_3)$ with $x_{\mathrm{hi}} \in X_{\mathrm{hi}}$ and $\dot{p}_3$ as in (9.40c).

Note, we do not model the Mambo's rotation dynamics. For our simulations and hardware demonstration, the Mambo's yaw is always close to 0 rad; we collected flight data accordingly for system identification, which is why u_{yr} does not appear in the model. Further, since the Mambo is a microdrone, it has much faster rotational dynamics than translational dynamics; based on flight data, we noticed that its on-board (black box) flight controller is able to achieve desired velocities in the plane by quickly pitching or rolling to accelerate, then returning to level flight. To incorporate a drone's rotation dynamics, one can use the methods presented for the Hummingbird above.

A Simulink high-fidelity model of the Mambo is available [Mat19a]. However, we do not use this model because we have changed the drone's mass and inertia by attaching motion capture markers, and because the Simulink model does not use the same control inputs as those we specify above. Instead of explicitly measuring the drone's changed mass and inertia, we represent them implicitly in the model coefficients. To fit the model coefficients, we first use a PhaseSpace Impulse X2E motion capture system to record approximately 400 s of the drone's position and attitude at 100 Hz. The data was collected with a desired yaw of 0 rad. We fit polynomials to the position data to smooth it, and manually discard the first and last 5–10% of the data where the polynomial fit is poor. We then differentiate the polynomials to approximate velocity and acceleration. Finally, we use nonlinear least squares to fit the coefficients.

9.6.2 Planning Model

We use the same planning model, based on [MHD15], as we used for the Hummingbird in §9.5.2. However, we limit v_{max} to 1.5 m/s.

9.6.3 Tracking Controller

We use the same PD tracking controller, based on [MK11], as the Hummingbird (see §9.5.3). For the hardware, we send commands generated by this controller to the drone over bluetooth using PyParrot [McG19].

9.6.4 Forward Reachable Set

We compute the FRS in the same way as the Hummingbird in §9.5.4, using [Alt15] to compute the PRS, and Algorithm 3 to compute the ERS. The body is overestimated by a cuboid zonotope of size $0.2 \times 0.2 \times 0.2$ m^3.

Importantly, to compute the ERS, we partition the initial condition space in both velocity and *altitude* (the p_3 state, which determines ground effect). We partition the velocity range of $[-1.5, 1.5]$ m/s into 7 subsets in each of the 3 velocity dimensions. We partition the altitude range of $[0, 2]$ m into 9 subsets. This results in approximately 450,000 error zonotopes; it takes approximately 1.5 hours to run Algorithm 3 on a 3.4 GHz laptop. Note that this is slower than the Hummingbird ERS computation, because we simulate the Mambo using MATLAB'S `ode45` solver as opposed to Euler integration. The ERS is stored in a lookup table of approximately 4 MB.

9.6.5 Simulation in Static Environments

Environment The drone flies indoors, in a $5.4 \times 2.2 \times 2$ m^3 rectangular area (the maximum altitude is 2 m). In every trial, the drone begins at one end of the flight area, and must traverse the entire flight area to reach a global goal area (a sphere of radius 0.25 m) located at a height of 1.5 m.

We simulate 1000 static environments, which contain between 0 and 18 random cuboid obstacles. We ensure the obstacles are not within 0.5 m of the start or goal; however, there is no guarantee that any trial has a feasible path from start to goal. A trial is successful if the robot reaches the global goal without any collisions.

High-Level Planner We use an RRT* HLP [KF11] to generate a waypoint up to 1 m away from the drone, along a collision-free path, at each receding-horizon planning iteration. The RRT* code is available online [KVL19]. The trajectory optimization cost function is to minimize the Euclidean distance to the waypoint at the time t_{peak}.

RTD Online Planning All obstacles are represented as cuboid zonotopes. We use these zonotopes to generate collision-avoidance constraints as in §6.4.

We solve RTD's online trajectory optimization method in two ways. First, we use the `fmincon` generic nonlinear solver. Second, we use the sampling approach presented for the Hummingbird in §9.5.5. Both methods have the timeout of $t_{\text{plan}} = 0.5$ s enforced.

Comparison Methods As mentioned above, we compare RTD to itself using both `fmincon` and a sampling method to solve the online trajectory optimization program.

We also compare against two different state-of-the-art spline-based approaches: [RBR16], which solves a quadratic program (QP) to generate a spline, and [MHD15], which provides an analytic spline. For these methods, we buffer obstacles by 0.05 m to compensate for the drone's body, plus an additional 0.1 m to compensate for tracking error. Both of these methods are implemented in MATLAB, in our open-source simulator [KVL19], as is RTD.

We run all simulations on both a 3.4 GHz processor and a 2.8 GHz processor. This is to show how RTD performs under computation constraints such as one might find on a small drone with limited processing power.

Results RTD has no collisions, as expected, regardless of the method used to implement its online trajectory optimization. The spline-based methods cannot make collision avoidance guarantees, because they do not provably account for tracking error. Furthermore, when using a sampling-based approach to perform trajectory optimization, RTD reaches the most goals regardless of processor speed.

We also find that RTD with sampling and the analytic splines [MHD15] reach goals more often than the RTD and spline approaches that rely on gradient-based optimization to find trajectories. This is especially noticeable when using the slower 2.8 GHz processor, where `fmincon` notably struggles to find solutions within the t_{plan} timeout. We were surprised to see that the `quadprog` approach struggled, but we attribute this to the potential for numerical instability in constructing splines in small, cluttered environments [RBR16].

Method	Goals Reached [%]	Collisions [%]
RTD + sampling	**93.7 / 86.4**	**0.0 / 0.0**
RTD + `fmincon`	72.0 / 19.0	**0.0 / 0.0**
spline + `quadprog` [RBR16]	50.1 / 41.4	5.6 / 7.7
spline + sampling [MHD15]	81.6 / 74.1	5.6 / 4.0

Table 9.7: Static obstacles results from 1000 trials for the Mambo microdrone. The slash separates trials run on two different processors (3.4 / 2.8 GHz). Our proposed RTD reaches the most goals, and never causes collisions, regardless of processor speed. We also see that sampling methods outperform derivative-based methods (`quadprog` and `fmincon`) for trajectory optimization.

9.6.6 Simulation in Dynamic Environments

Environment We also simulate 1000 dynamic environments. The start, goal, and environment size are the same as for the static environments. Each environment contains between 0 and 3 static obstacles, and either 1 or 2 human-shaped dynamic obstacles of size $0.75 \times 0.75 \times 2$ m^3. These obstacles travel along a randomly-generated path at a randomly-selected speed between 0.25 and 0.75 m/s. To avoid introducing additional complexity in terms of assessing fault in collisions, we do not simulate interactions; that is, the dynamic obstacles do not alter their course to avoid the drone. This makes it more challenging to successfully navigate the dynamic environments, since the obstacles may be aggressive. We mitigate this by providing all methods (RTD and the comparison method mentioned below) with perfect predictions of each obstacle's motion up to 3 s into the future. A trial is successful if the robot reaches the global goal without causing any at-fault collisions.

High-Level Planner We use a straight-line HLP, where the robot attempts to reach a waypoint 1 m along a straight line between itself and the global goal in each receding-horizon planning iteration. The trajectory optimization cost function is to minimize the Euclidean distance to the waypoint at the time t_{peak}.

RTD Online Planning We use the same approach as above for the Mambo in static environments. That is, all obstacles are represented as cuboid zonotopes, and we generate collision-avoidance constraints as in §6.4. We solve RTD's online trajectory optimization method using the `fmincon` generic nonlinear solver and the sampling approach presented for the Hummingbird in §9.5.5. Both methods have the timeout of $t_{\text{plan}} = 0.5$ s enforced.

Comparison Methods In addition to comparing RTD against itself with `fmincon` and sampling, we compare to a potential field tracking controller [FKS20]. We tuned this controller empirically to avoid collision with a single dynamic obstacle as presented in [FKS20]. Note, this method is *not* a trajectory planner; instead, we use the output of the straight line HLP in each planning iteration as the desired location for the controller to drive the robot towards. The controller creates artificial forces to repel itself from static and dynamic obstacles. We implement the controller at a rate of 100 Hz.

Results RTD causes no at-fault collisions, as expected. It significantly outperforms the potential field tracking controller; this shows the important of trajectory planning to enforce dynamic obstacle avoidance in arbitrary scenarios. Recall that we noticed the same trend with the linear MPC for the EV in §9.4.

The potential field tracking controller [FKS20] struggles to navigate arbitrary environments because the dynamic obstacles often push the drone towards static obstacles, trapping the drone and causing a collision. This is expected, because the tracking controller does not plan an entire trajectory to avoid becoming trapped.

Method	**Goals Reached [%]**	**Collisions [%]**
RTD + sampling	**99.4 / 92.9**	**0.0 / 0.0**
RTD + `fmincon`	96.9 / 54.8	**0.0 / 0.0**
potential field [FKS20]	58.5 / 58.6	40.5 / 40.4

Table 9.8: Dynamic obstacles results from 1000 trials for the Mambo microdrone. The slash separates trials run on two different processors (3.4 / 2.8 GHz). The trends are the same as for static obstacles (see Table 9.7). Notice the potential field low-level controller [FKS20] has nearly identical numbers regardless of processor speed, which is expected since it is not performing trajectory optimization.

9.6.7 Hardware Demonstration

We applied RTD to the hardware drone in four static environments, and three dyanmic environments (using a car-like Rover robot [KVB+20], or two humans, as obstacles). The drone never crashed. A video is available: https://youtu.be/1cldHVQK3Yw.

The hardware demonstration revealed three unmodeled aerodynamic effects. First, open vents in the ceiling of our testing area produced wind. Second, dynamic obstacles created wake. Third, downwash from the drone augmented the ground effect near obstacles. Due to the Mambo's small size (63 g including motion capture markers), the first two effects were able to push the drone up to several centimeters out of its FRS (i.e., prevent strict collision-avoidance guarantees) on two occasions (all other instances were mediated by the drone's feedback controller). The third effect caused the drone to struggle to maintain a low altitude, but was mediated by the goal's 1.5 m altitude. We plan to include these effects in RTD for future work.

9.7 The Fetch Manipulator

We use the Fetch mobile manipulator platform [WFK+16] to demonstrate RTD's rotatotope FRS presented in §8. To the best of our knowledge, this the is the first provably safe and real-time manipulator trajectory planner.

The Fetch arm has 7 revolute DOFs (the robot has other DOFs for its wheeled mobile base; we have not yet implemented RTD on the mobile base). We use the arm's first 6 DOFs for RTD, and treat the body as an obstacle. The 7th DOF controls end effector orientation, which does not affect the volume used for collision checking. We command the hardware via ROS [QCG+09] over WiFi. RTD is implemented in MATLAB, CUDA, and C++, on a 3.6 GHz computer with an Nvidia Quadro RTX 8000 GPU.

The robot is simulated using our open-source MATLAB simulator [KVL19]. The code used to simulate the robot is available online [HZKR20]. We use $t_{\text{plan}} = 0.5$ s for the receding-horizon planning timeout in both simulation and hardware.

9.7.1 Robot Model

First, we note that the Fetch's motors and built-in motor controllers produce negligible tracking error (within 0.01 rad) with respect to our choice of parameterized trajectories below. We expect that this assumption will not hold when we move to more aggressive dynamic motion, and grasping heavy objects; but, our current (kinematic) approach is an important first step towards these goals. Also see §8.1.2.

Recall that, for the rotatotope FRS formulation, we parameterize the (kinematic) joint trajectories directly. We use $X = Q = \mathbb{S}^6$ (that is, we are considering 6 revolute DOFs). We define a velocity parameter $k_v \in \mathbb{R}^{n_Q}$ that defines the joint's initial velocity, and an acceleration parameter $k_a \in \mathbb{R}^{n_Q}$ that specifies a constant acceleration over $[0, t_{\text{plan}})$. We write $k_v = (k_{v_1}, \cdots, k_{v_{n_Q}})$ and similarly for k_a. We denote $k = (k_v, k_a) \in K \subset \mathbb{R}^{n_K}$, where $n_K = 2n_Q$. The trajectories are given by

$$\dot{x}_i(t; k) = \begin{cases} k_v + k_a t, & t \in [0, t_{\text{plan}}) \\ \frac{k_v + k_a t_{\text{plan}}}{t_{\text{f}} - t_{\text{plan}}}(t_{\text{f}} - t), & t \in [t_{\text{plan}}, t_{\text{f}}], \end{cases} \tag{9.41}$$

with $x_i(0;k) = 0$ for all k (that is, $x_0 = 0$ is the center of the planning frame for the planning model). These trajectories brake to a stop over $[t_{\text{plan}}, t_{\text{f}}]$ with constant acceleration.

We create K as the Cartesian product of a parameter interval for each joint:

$$K = K_1 \times K_2 \times \cdots K_{n_Q} m \tag{9.42}$$

where K_i denotes the parameters for joint i. For each joint i, we specify $K_i = K_{v_i} \times K_{a_i}$, where

$$K_{v_i} = \left[k_{v_i,0} - \Delta_{k_{v_i}}, k_{v0,i} + \Delta_{k_{v_i}}\right] \text{ and} \tag{9.43}$$

$$K_{a_i} = \left[k_{a_i,0} - \Delta_{k_{a_i}}, k_{a_i,0} + \Delta_{k_{a_i}}\right], \tag{9.44}$$

with $k_{v_i,0}$, $k_{a_i,0}$, $\Delta_{k_{v_i}}$, and $\Delta_{k_{a_i}} \in \mathbb{R}$. We also ensure that K_{a_i} is small enough to enforce the acceleration limits of the manipulator.

We use $t_{\text{plan}} = 0.5$ s and $t_{\text{f}} = 1.0$ s. For each i^{th} joint, we treat the joint limits as $\dot{x}_{\text{max},i} = \pi$ rad/s and $\ddot{x}_{\text{max},i} = \pi/3$ rad/s^2.

9.7.2 Forward Reachable Set

We apply the procedure detailed in §8.4 as written. We use the toolbox [Alt15] to compute the JRS as a set of zonotopes as in (8.26). We partition time with $\Delta_t = 0.01$ s, resulting in a JRS represented by 100 zonotopes.

While we do not compute an ERS for the Fetch, we did find that the JRS becomes less conservative when we choose a smaller range for k_v. Therefore, we partition the space K_{v_i} into 400 small intervals, and set $k_{v_i,0}$ and $\Delta_{k_{v_i}}$ appropriately given $\dot{x}_{\text{max},i}$ as above. For each subset in this partition of K_{v_i}, we set $\Delta_{k_{a_i}}$ as no more than $\pi/24$ rad/s^2 while ensuring that the parameter range does not allow the robot to exceed its joint speed limits. At runtime, we choose the appropriate JRS for each joint's current speed (i.e., we use FRS swapping as in §4.7).

9.7.3 Simulation in Static Environments

Environment We created two sets of trials. The first set, Random Obstacles, shows that RTD can handle arbitrary tasks (see Fig 9.14). This set contains 100 tasks with random (but collision-free) start and goal configurations, and random box-shaped obstacles. Obstacle side lengths vary from 1 to 50 cm, with 10 trials for each $n_O = 4, 8, ..., 40$. The second set, Hard Scenarios, shows that RTD guarantees safety where a comparison method (see below) converges to an unsafe trajectory. There are seven tasks in the Hard Scenarios set: (1) from below to above a table, (2) from one side of a wall to another, (3) between two vertical posts, (4) from one set of shelves to another, (5) from inside to outside of a box on the ground, (6) from a sink to a cupboard, (7) through a small window. These are shown in Figure 9.15.

High-Level Planner To illustrate that RTD can enforce safety, we use two different HLPs, neither of which is guaranteed to generate collision-free waypoints. First, a straight-line HLP that generates waypoints along a straight line between the arm and a global goal in configuration space. Second, an RRT* [KF11] that only ensures the arm's *end effector* is collision-free. Thus, *RTD can act as a safety layer on top of an unsafe RRT**.

Note, we allot 0.1 s of t_{plan} to the HLP in each iteration, and give RTD the rest of t_{plan}. We cannot use CHOMP as a receding-horizon planner with these HLP waypoints, because it requires a collision-free goal configuration.

The cost function for trajectory optimization is to minimize the Euclidean distance *in configuration space* to the waypoint at each planning iteration.

RTD Online Planning We represent all obstacles as zonotopes, and apply the procedure in §8.6 as written.

We use a GPU with CUDA to compose the rotatotopes in parallel, taking approximately 10–20 ms to compose a full RS. Constraint generation is also parallelized across obstacles and time steps (this takes approximately 10–20 ms for 20 obstacles).

We solve the online trajectory optimization program with IPOPT [WLMK20]. Note, we pass this solver analytic subgradients for the collision-avoidance constraints.

Comparison Method To assess the difficulty of our simulated environments, we ran CHOMP [ZRD+13] via MoveIt [CSCC14] (default settings, straight-line initialization). We emphasize that CHOMP is not a receding-horizon planner [CSCC14]; it attempts to find a plan from start to goal with a single optimization program. However, CHOMP provides a useful baseline to measure the performance of RTD. To the best of our knowledge, no open-source, real-time receding-horizon planner is available for a direct comparison. The cost function for CHOMP is the default, created to have the arm reach the input goal configuration (which we ensure is collision free for each trial).

We did not attempt to tune CHOMP to only find feasible plans (e.g., by buffering the arm), since this incurs a tradeoff between safety and performance. Note, in MoveIt, infeasible CHOMP plans are not executed (if detected by an external collision-checker).

Results RTD produced no collisions in any trial, as expected. It reaches a comparable number of goals to CHOMP in the Random Obstacles trials, but in a receding-horizon way. It is also able to solve 5/7 of the Hard Scenarios, with which CHOMP struggled.

For the Random Obstacles trials, the results are as follows, and are summarized in Table 9.14 oth RTD and CHOMP reach nearly the same number of goals; however, all of CHOMP's failures were trials in which it converged to a collision. We report the mean solve time (MST) of RTD over all planning iterations, while the MST for CHOMP is the mean over all 100 tasks. Directly comparing timing is not possible since RTD and CHOMP use different planning paradigms; we report MST to confirm RTD is capable of real-time planning (note that that RTD's MST is less than $t_{\text{plan}} = 0.5$). We also report the mean normalized path distance (MNPD) of the plans produced by each planner (the mean is taken over all 100 tasks). The normalized path distance is a path's Euclidean distance (in configuration space), divided by the Euclidean distance between the start and goal. For example, the straight line from start to goal has a (unitless) normalized path distance of 1. RTD's MNPD is 24% smaller than CHOMP's when using the straight line HLP, which may be because CHOMP's cost rewards path smoothness, whereas RTD's cost rewards reaching an intermediate waypoint at each planning iteration (note, path smoothness could be included in RTD's cost function).

For the Hard Scenarios, the results are as follows, and summarized in Table 9.10. With the straight-line HLP, RTD does not complete any of the tasks but also has no collisions. With the RRT* HLP [KF11], RTD completes $5/7$ scenarios. CHOMP converges to trajectories with collisions in all of the Hard Scenarios.

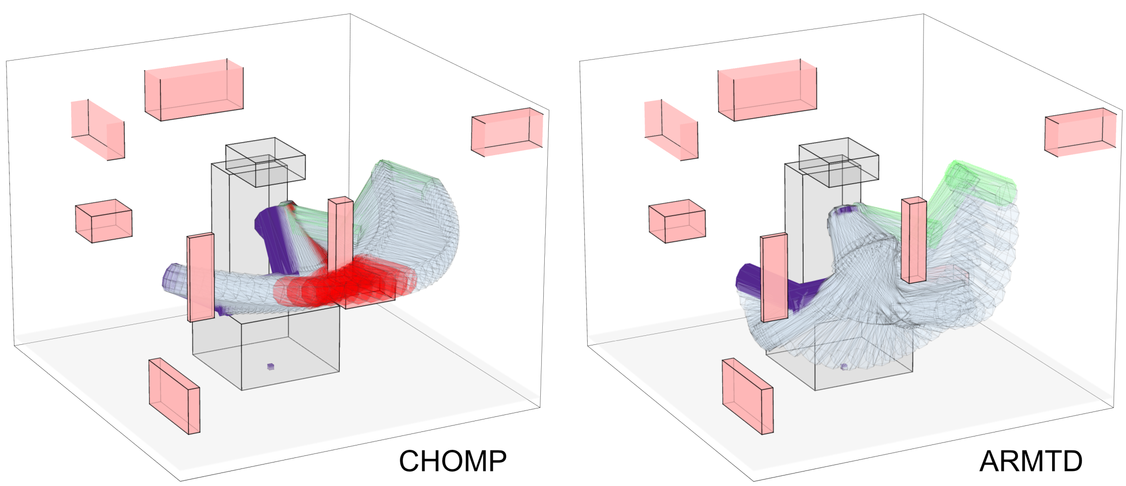


Figure 9.14: A Random Obstacles trial with 8 obstacles in which CHOMP [ZRD+13] converged to a trajectory with a collision (collision configurations shown in red), whereas RTD successfully navigated to the goal (green); the start pose is shown in purple. CHOMP fails to move around a small obstacle close to the front of the Fetch.

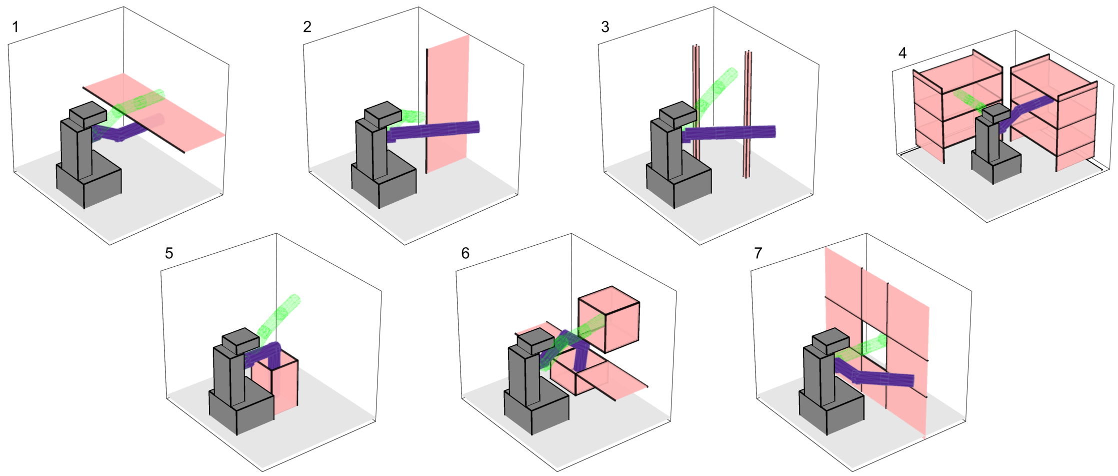


Figure 9.15: The set of seven Hard Scenarios (number in the top left), with start pose shown in purple and goal pose shown in green. There are seven tasks in the Hard Scenarios set: (1) from below to above a table, (2) from one side of a wall to another, (3) between two vertical posts, (4) from one set of shelves to another, (5) from inside to outside of a box on the ground, (6) from a sink to a cupboard, (7) through a small window.

Method	Goals [%]	Collisions [%]	MST [s]	MNPD
RTD + SL	84	0	0.273	1.076
RTD + RRT*	62	0	0.466	1.417
CHOMP	82	18	0.177	1.511

Table 9.9: Simulation results for the Fetch mobile manipulator on the 100 Random Obstacles trials. RTD uses the straight-line (SL) and RRT* HLPs; CHOMP [ZRD+13] uses the default settings from MoveIt [CSCC14]. MST is mean solve time (per planning iteration for RTD, and total for CHOMP) and MNPD is mean normalized path distance. MNPD is only computed for trials where the task was successfully completed, i.e. the path was valid.

9.7.4 Hardware Demonstration

RTD completes arbitrary tasks while safely navigating the Fetch arm around obstacles in scenarios similar to Hard Scenarios (1) and (4). We demonstrate real-time planning by suddenly introducing obstacles (a box, a vase, and a quadrotor) in front of the moving arm. The obstacles are tracked using motion capture, and are treated as static in each planning iteration. Since RTD performs receding-horizon planning, it can react to the sudden obstacle appearance and continue planning without collisions. A video is available: youtu.be/ySnux2owlAA.

Method	1	2	3	4	5	6	7
RTD + SL	S	S	S	S	S	S	S
RTD + RRT*	O	O	O	S	O	S	O
CHOMP	C	C	C	C	C	C	C

Table 9.10: Simulation results for the seven Hard Scenario simulations. RTD uses the straight-line (SL) and RRT* HLPs. The entries are "O" for task completed, "C" for a crash, or "S" for stopping safely without reaching the goal.

9.8 Chapter Review

The takeaway of this chapter is that RTD provides safe, real-time planning over thousands of simulations and dozens of hardware demonstrations, on seven different robots with three different morphologies. An additional wheeled robot, the Turtlebot, is demonstrated in the open-source tutorial [KV19]. Additional manipulators are available in the manipulator RTD repository [HZKR20].

9.8.1 Chapter Summary

We demonstrated RTD on four different wheeled robots: the Segway, Rover, Fusion, and EV. We then demonstrated RTD on two different quadrotors: the Hummingbird and the Mambo. Finally, we demonstrated RTD on a Fetch manipulator.

9.8.2 What is Missing?

For wheeled robots, we have not yet considered significant tire slip during aggressive maneuvers, where an accurate high-fidelity model may not be available. For aerial robots, we have not yet considered wind. For manipulators, we have yet to treat the full dynamics with, e.g., Coriolis forces and joint torque limits. We have also not yet treated the multi-agent planning case.

CHAPTER 10

Conclusion and Future Directions

This dissertation has developed Reachability-based Trajectory Design (RTD) as a general framework for provably-safe, real-time motion planning. The method is demonstrated on wheeled, aerial, and manipulator robots; across thousands of simulations and dozens of hardware demonstrations, RTD has enabled these robots to safely and successfully complete tasks that are challenging or impossible for other state-of-the-art approaches.

In this chapter, we provide a review of the dissertation's contributions, and discuss future research directions.

10.1 Dissertation Review and Contributions

We first briefly review the structure of this dissertation. In §3, we developed a generic theoretical framework for RTD; in particular, we formally specified notions of safety and fault, and showed how mathematical objects called reachable sets can be used to formulate safe motion planning. To implement this theory, in §4, we developed an offline sums-of-squares polynomial approach to compute the robot's Forward Reachable Set (FRS) as a polynomial, and showed how to use this polynomial at runtime to enable provably-safe motion planning. Then, in §5, we specified a novel discretized obstacle representation for arbitrary planar (i.e., wheeled) robots that enables safe and real-time planning with the polynomial FRS. To extend RTD to robots outside the plane, in §6, we then developed an FRS representation using zonotopes, a special type of convex polytope. We showed how to incorporate tracking error into the polynomial and zonotope FRSes in §7. To conclude the development of RTD, in §8 we introduced rotatotopes, an extension of zonotopes that make RTD tractable for multi-link robots such as manipulators, whereas the polynomial and zonotope methods were restricted to single rigid-body robots. Finally, in §9, we showed that RTD is practical for wheeled, aerial, and manipulator robots with a large variety of simulations and hardware demonstrations.

The key contribution of this work is RTD as a general theoretical framework. We developed several mathematical formulations of reachable sets and obstacle representations to make RTD numerically tractable. We have also made our code available and accessible [KVV19, KV19, HZKR20]. Finally, we demonstrated the practicality of RTD with extensive evaluation and comparison against other state-of-the-art methods.

10.2 Future Research Directions

Several gaps remain that, if filled, can make RTD even more widely applicable and practical. We now discuss these gaps, and potential ways forward in addressing them. We then specify which ones shall be filled by the completion of this dissertation.

Probabilistic Obstacles Currently, all obstacle representations are expected to be deterministic overapproximations for RTD applied to any robot morphology. This is a conservative approach, and can lead to the robot stopping frequently. Instead, we can consider obstacles (and robot dynamics) described with probability distributions (e.g., [JHJRV17, BBR$^+$19]), and use a risk measure such as Conditional Value-at-Risk (CVaR) [CLT$^+$19] to define probabilistic collision-avoidance constraints at runtime.

Tracking Error Computation So far, we have computed tracking error offline via sampling and conservatively fitting polynomials [KVB$^+$20] or zonotopes [KHV19] to these samples. We have leveraged information about the high-fidelity model to choose these samples to maximize tracking error. In cases where the high-fidelity model changes slowly (such as a drone's battery losing charge) or is unknown, we still want to be able to capture tracking error, without simply choosing a large number to encompass all possible tracking error *a priori*. In other words, we should be able to learn the model error at runtime, perhaps similar to a learning-based MPC approach [AGST13].

Trajectory Optimization During online receding-horizon planning, we have so far always used a nonlinear solver for trajectory optimization, such as MATLAB's `fmincon` [Mat19b], or IPOPT [WLMK20]. These solvers can be fast, but have no certification of finding globally optimal solutions, or even finding feasible solutions if they exist. To address this, we have begun exploring branch-and-bound techniques to find optimal plans in real time [KZZV20].

Interaction Modeling In dynamic environments, we have not yet considered interactions. That is, we have not included how other actors would react to our motion plan in the predictions of their behavior. Ideally, we could include interaction as part of the cost or constraints for online trajectory optimization, akin to a differential game formulation (e.g., [MBT05], but this application would require a much faster solver).

Tire Friction For wheeled robots, we have not yet explicitly incorporated tire friction limits (instead, we have implicitly included them as tracking error). This approach means we have not yet handled cases such as driving on snow or ice. A possible intermediate step towards treating this driving regime is to consider stable drifting trajectories such as in [GG16].

Drone Aerodynamics For quadrotor drones, we have not yet included aerodynamic disturbances such as wind or obstacle wake, which we identified as issues with the Parrot Mambo hardware demonstration in §9.6. A potential way forward is to use higher-dimensional, linearized versions of the dynamics to capture these nonlinear effects, since our quadrotor planning model is linear [KHV19]. Such an approach has been shown successfully for hard-to-model systems such as soft robots [BRV19].

Manipulator Dynamics For manipulator arms, we have not yet considered dynamics and the resulting trajectory-dependent bounds on actuator torque limits. Since our approach for arms uses zonotope reachability, we may be able to incorporate these limits as zonotopes that bound tracking error [GA17], then add them to our reachable set at runtime as is done with our approach for drones [KHV19]. Furthermore, we have not yet performed grasping tasks with RTD; but, our current approach can tolerate runtime changes to the arm's geometry.

10.3 Final Remarks

The key takeaway of this work is the practicality of safe, real-time robot motion planning. We have shown this on a variety of robots in real-world demonstrations, and have made our code and implementations available and accessible. However, in addition to the future research directions mentioned above, it is critical for robots to have *perception* guarantees, which are not addressed in this work. To make robots in general, practical, the major next step lies in closing the loop between perception and planning in a robust and fast way. We believe that RTD is an important step towards this goal.

BIBLIOGRAPHY

[ACE+19] Aaron D Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath, and Paulo Tabuada. Control barrier functions: Theory and applications. In *2019 18th European Control Conference (ECC)*, pages 3420–3431. IEEE, 2019.

[AGLP19] M. Althoff, A. Giusti, S. B. Liu, and A. Pereira. Effortless creation of safe robots from modules through self-programming and self-verification. *Science Robotics*, 4(31), 2019.

[AGST13] Anil Aswani, Humberto Gonzalez, S Shankar Sastry, and Claire Tomlin. Provably safe and robust learning-based model predictive control. *Automatica*, 49(5):1216–1226, 2013.

[Alt10] Matthias Althoff. *Reachability analysis and its application to the safety assessment of autonomous cars*. PhD thesis, Technische Universität München, 2010.

[Alt13] Matthias Althoff. Reachability Analysis of Nonlinear Systems Using Conservative Polynomialization and Non-Convex Sets. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, page 173–182, New York, NY, USA, 2013. Association for Computing Machinery.

[Alt15] M. Althoff. An Introduction to CORA 2015. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

[Asc19] Ascending Technologies. AscTec Hummingbird, 3 2019.

[BAC06] José Manuel Bravo, Teodoro Alamo, and Eduardo F Camacho. Robust MPC of constrained discrete-time nonlinear systems based on approximated reachable sets. *Automatica*, 42(10):1745–1751, 2006.

[BBB+19] Andrea Bajcsy, Somil Bansal, Eli Bronstein, Varun Tolani, and Claire J Tomlin. An efficient reachability-based framework for provably safe autonomous navigation in unknown environments. *arXiv preprint arXiv:1905.00532*, 2019.

[BBR+19] Somil Bansal, Andrea Bajcsy, Ellis Ratner, Anca D Dragan, and Claire J Tomlin. A Hamilton-Jacobi reachability-based framework for predicting and analyzing human motion for safe planning. *arXiv preprint arXiv:1910.13369*, 2019.

[BK00] Robert Bohlin and Lydia E Kavraki. Path planning using lazy PRM. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on*

Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065), volume 1, pages 521–528. IEEE, 2000.

[BL91] Jerome Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649, 1991.

[BM99] Alberto Bemporad and Manfred Morari. Robust model predictive control: A survey. In *Robustness in identification and control*, pages 207–226. Springer, 1999.

[BPA17] D. Beckert, A. Pereira, and M. Althoff. Online verification of multiple safety criteria for a robot trajectory. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 6454–6461, Dec 2017. View online.

[BRV19] Daniel Bruder, C David Remy, and Ram Vasudevan. Nonlinear system identification of soft robot dynamics using koopman operator theory. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6244–6250. IEEE, 2019.

[BWWN19] Julie Stephany Berrio, James Ward, Stewart Worrall, and Eduardo Nebot. Identifying robust landmarks in feature-based maps. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1166–1172. IEEE, 2019.

[C$^+$13] Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15(49):5, 2013.

[CHV$^+$18] M. Chen, S. L. Herbert, M. S. Vashishtha, S. Bansal, and C. J. Tomlin. Decomposition of Reachable Sets and Tubes for a Class of Nonlinear Systems. *IEEE Transactions on Automatic Control*, 63(11):3675–3688, 11 2018.

[CLS16] J. Chen, T. Liu, and S. Shen. Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1476–1483, 5 2016.

[CLT$^+$19] Margaret P Chapman, Jonathan Lacotte, Aviv Tamar, Donggun Lee, Kevin M Smith, Victoria Cheng, Jaime F Fisac, Susmit Jha, Marco Pavone, and Claire J Tomlin. A Risk-Sensitive Finite-Time Reachability Approach for Safety of Stochastic Dynamic Systems. In *2019 American Control Conference (ACC)*, pages 2958–2963. IEEE, 2019.

[CMO14] Elena Celledoni, Håkon Marthinsen, and Brynjulf Owren. An introduction to Lie group integrators–basics, new developments and applications. *Journal of Computational Physics*, 257:1040–1061, 2014.

[CPG17] Yuxiao Chen, Huei Peng, and Jessy Grizzle. Obstacle avoidance for low-speed autonomous vehicles with barrier function. *IEEE Transactions on Control Systems Technology*, 26(1):194–206, 2017.

[CSCC14] David Coleman, Ioan Alexandru Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study. *CoRR*, abs/1404.3785, 2014.

[CSS15] J. Chen, K. Su, and S. Shen. Real-time safe trajectory generation for quadrotor flight in cluttered environments. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1678–1685, Dec 2015.

[CSW$^+$19] Andrea Censi, Konstantin Slutsky, Tichakorn Wongpiromsarn, Dmitry Yershov, Scott Pendleton, James Fu, and Emilio Frazzoli. Liability, ethics, and culture-aware behavior specification using rulebooks. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8536–8542. IEEE, 2019.

[DGZD15] Wei Dong, Guo-Ying Gu, Xiangyang Zhu, and Han Ding. Development of a quadrotor test bed—modelling, parameter identification, controller design and trajectory generation. *International Journal of Advanced Robotic Systems*, 12(2):7, 2015.

[DS16] Christopher M. Dellin and Siddhartha S. Srinivasa. A Unifying Formalism for Shortest Path Problems with Expensive Edge Evaluations via Lazy Best-first Search over Paths with Edge Selectors. In *Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'16, pages 459–467. AAAI Press, 2016.

[Dub57] Lester E Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of mathematics*, 79(3):497–516, 1957.

[Fer00] Christophe Ferrier. Computation of the distance to semi-algebraic sets. *ESAIM: Control, Optimisation and Calculus of Variations*, 5:139–156, 2000.

[FHW12] Efi Fogel, Dan Halperin, and Ron Wein. *Minkowksi Sums and Offset Polygons*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[FKS20] Davide Falanga, Kevin Kleber, and Davide Scaramuzza. Dynamic obstacle avoidance for quadrotors with event cameras. *Science Robotics*, 5(40), 2020.

[FS75] H. Freeman and R. Shapira. Determining the Minimum-area Encasing Rectangle for an Arbitrary Closed Curve. *Commun. ACM*, 18(7):409–413, July 1975.

[GA17] A. Giusti and M. Althoff. Efficient Computation of Interval-Arithmetic-Based Robust Controllers for Rigid Robots. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 129–135, 4 2017.

[GG16] Jonathan Y Goh and J Christian Gerdes. Simultaneous stabilization and tracking of basic automobile drifting trajectories. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 597–602. IEEE, 2016.

[GKM10] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous UAV guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65, 2010.

[GNZ03] Leonidas J Guibas, An Nguyen, and Li Zhang. Zonotopes as bounding volumes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 803–812. Society for Industrial and Applied Mathematics, 2003.

[GPM89] Carlos E Garcia, David M Prett, and Manfred Morari. model predictive control: theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.

[GPMN15] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi. A review of motion planning techniques for automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1135–1145, 2015.

[Hau12] Kris Hauser. On responsiveness, safety, and completeness in real-time motion planning. *Autonomous Robots*, 32(1):35–48, 1 2012.

[HCH+17] S. L. Herbert, M. Chen, S. Han, S. Bansal, J. F. Fisac, and C. J. Tomlin. FaSTrack: A modular framework for fast and guaranteed safe motion planning. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 1517–1522, 12 2017.

[HGK10] Thomas M Howard, Colin J Green, and Alonzo Kelly. Receding horizon model-predictive control for mobile robot navigation of intricate paths. In *Field and Service Robotics*, pages 69–78. Springer, 2010.

[HHWT11] Gabriel M Hoffmann, Haomiao Huang, Steven L Waslander, and Claire J Tomlin. Precision flight control for a multi-vehicle quadrotor helicopter testbed. *Control engineering practice*, 19(9):1023–1036, 2011.

[HKMV16] Patrick Holmes, Shreyas Kousik, Shankar Mohan, and Ram Vasudevan. Convex estimation of the α-confidence reachable set for systems with parametric uncertainty. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 4097–4103. IEEE, 2016.

[HKRA16] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-Time Loop Closure in 2D LIDAR SLAM. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.

[HKZ+20] Patrick Holmes, Shreyas Kousik, Bohao Zhang, Daphna Raz, Corina Barbalata, Matthew Johnson-Roberson, and Ram Vasudevan. Reachable Sets for Safe, Real-Time Manipulator Trajectory Design, 2020.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HWMZ20] Lukas Hewing, Kim P Wabersich, Marcel Menner, and Melanie N Zeilinger. Learning-based model predictive control: Toward safe learning in control. *Annual Review of Control, Robotics, and Autonomous Systems*, 3:269–296, 2020.

[HZKR20] Patrick Holmes, Bohao Zhang, Shreyas Kousik, and Daphna Raz. Autonomous Reachability-based Manipulator Trajectory Design Repository, 2020.

[JHJRV17] Henry O Jacobs, Owen K Hughes, Matthew Johnson-Roberson, and Ram Vasudevan. Real-time certified probabilistic pedestrian forecasting. *IEEE Robotics and Automation Letters*, 2(4):2064–2071, 2017.

[KF11] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[KFT+08] Yoshiaki Kuwata, Gaston A Fiore, Justin Teo, Emilio Frazzoli, and Jonathan P How. Motion planning for urban driving using RRT. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1681–1686. IEEE, 2008.

[KG02] Hassan K Khalil and Jessy W Grizzle. *Nonlinear systems*, volume 3. Prentice hall Upper Saddle River, NJ, 2002.

[KHV19] Shreyas Kousik, Patrick Holmes, and Ram Vasudevan. Safe, Aggressive Quadrotor Flight via Reachability-Based Trajectory Design. *Dynamic Systems and Control Conference*, 3, 10 2019. V003T19A010.

[KMO+12] Shahab Kaynama, John Maidens, Meeko Oishi, Ian M Mitchell, and Guy A Dumont. Computing the viability kernel using maximal reachable sets. In *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pages 55–64, 2012.

[KQCD15] Christos Katrakazas, Mohammed Quddus, Wen-Hua Chen, and Lipika Deka. Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. *Transportation Research Part C: Emerging Technologies*, 60:416–442, 2015.

[KRSV10] T. Kunz, U. Reiser, M. Stilman, and A. Verl. Real-time path planning for a robot arm in changing environments. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5906–5911, 10 2010.

[KS12] Tobias Kunz and Mike Stilman. Time-Optimal Trajectory Generation for Path Following with Bounded Acceleration and Velocity. In *Robotics: Science and Systems*, 2012.

[KSLO96] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[KV19] Shreyas Kousik and Sean Vaskov. RTD Tutorial, 2019.

[KVB+20] Shreyas Kousik*, Sean Vaskov*, Fan Bu, Matthew Johnson-Roberson, and Ram Vasudevan. Bridging the Gap Between Safety and Real-Time Performance in Receding-Horizon Trajectory Design for Mobile Robots. *The International Journal of Robotics Research*, 8 2020. In press.

[KVJRV17] Shreyas Kousik, Sean Vaskov, Matthew Johnson-Roberson, and Ram Vasudevan. Safe trajectory synthesis for autonomous driving in unforeseen environments. In *ASME 2017 Dynamic Systems and Control Conference*. American Society of Mechanical Engineers Digital Collection, 2017.

[KVL19] Shreyas Kousik, Sean Vaskov, and Hannah Larson. simulator, 2019.

[KVV19] Shreyas Kousik, Sean Vaskov, and Ram Vasudevan. Reachability-based Trajectory Design Repository, 2019.

[KZZV20] Shreyas Kousik, Bohao Zhang, Pengcheng Zhao, and Ram Vasudevan. Safe, Optimal, Real-time Trajectory Planning with a Parallel Constrained Bernstein Algorithm. *arXiv preprint arXiv:2003.01758*, 2020.

[Las10] Jean-Bernard Lasserre. *Moments, positive polynomials and their applications*, volume 1. World Scientific, 2010.

[LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006.

[LDM15] Alexander Liniger, Alexander Domahidi, and Manfred Morari. Optimization-based autonomous racing of 1: 43 scale RC cars. *Optimal Control Applications and Methods*, 36(5):628–647, 2015.

[LKJ01] Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[LLM10] Taeyoung Lee, Melvin Leok, and N Harris McClamroch. Control of complex maneuvers for a quadrotor UAV using geometric methods on SE (3). *arXiv preprint arXiv:1003.2005*, 2010.

[Mat19a] MathWorks. Parrot Minidrones Support from Simulink, Nov 2019.

[Mat19b] Mathworks. MATLAB Optimization Toolbox, 2019. The MathWorks, Natick, MA, USA.

[MBT05] Ian M Mitchell, Alexandre M Bayen, and Claire J Tomlin. A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. *IEEE Transactions on automatic control*, 50(7):947–957, 2005.

[McG19] Amy McGovern. PyParrot Documentation, Nov 2019.

[McN11] Matthew McNaughton. *Parallel algorithms for real-time motion planning*. PhD thesis, Citeseer, 2011.

[Mec18] Mechanical Simulation. CarSim, 2018.

[MFJQ+16] Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J. Sorin, and George Konidaris. Robot Motion Planning on a Chip. In *Robotics: Science and Systems*, 2016.

[MHD15] M. W. Mueller, M. Hehn, and R. D'Andrea. A Computationally Efficient Motion Primitive for Quadrocopter Trajectory Generation. *IEEE Transactions on Robotics*, 31(6):1294–1310, 12 2015.

[MK11] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE international conference on robotics and automation*, pages 2520–2525. IEEE, 2011.

[Mos10] Mosek ApS. The MOSEK optimization software, 2010.

[MSS18] Aditya Mandalika, Oren Salzman, and Siddhartha Srinivasa. Lazy receding horizon A* for efficient path planning in graphs with expensive-to-evaluate edges. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

[MT16] Anirudha Majumdar and Russ Tedrake. Funnel libraries for real-time robust feedback motion planning. *arXiv preprint arXiv:1601.04037*, 2016.

[Mun00] James Munkres. *Topology (2nd Edition)*. Pearson, 2 edition, January 2000.

[MVTT14] Anirudha Majumdar, Ram Vasudevan, Mark M Tobenkin, and Russ Tedrake. Convex optimization of nonlinear feedback controllers via occupation measures. *The International Journal of Robotics Research*, 33(9):1209–1230, 2014.

[OF15] Michael Otte and Emilio Frazzoli. RRT-X: Real-Time Motion Planning/Replanning for Environments with Unpredictable Obstacles. In *Algorithmic Foundations of Robotics XI*, pages 461–478. Springer, 2015.

[PA15] A. Pereira and M. Althoff. Safety control of robots under computed torque control using reachable sets. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 331–338, May 2015. View online.

[Par19] Parrot Drones. Parrot Mambo FPV, Nov 2019.

[PJ87] F. Pfeiffer and R. Johanni. A concept for manipulator trajectory planning. *IEEE Journal on Robotics and Automation*, 3(2):115–123, 4 1987.

[PKA16] L. Palmieri, S. Koenig, and K. O. Arras. RRT-based nonholonomic motion planning using any-angle path biasing. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2775–2781, May 2016.

[PKA19] Christian Pek, Markus Koschi, and Matthias Althoff. An online verification framework for motion planning of self-driving vehicles with safety guarantees. In *AAET-Automatisiertes und vernetztes Fahren*, 2019.

[PKK09] Mihail Pivtoraiko, Ross A Knepper, and Alonzo Kelly. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(3):308–333, 2009.

[PLM06] R. Pepy, A. Lambert, and H. Mounier. Path Planning using a Dynamic Vehicle Model. In *2006 2nd International Conference on Information Communication Technologies*, volume 1, pages 781–786, 2006.

[PMK+13] Caitlin Powers, Daniel Mellinger, Aleksandr Kushleyev, Bruce Kothmann, and Vijay Kumar. Influence of aerodynamics and proximity effects in quadrotor flight. In *Experimental robotics*, pages 289–302. Springer, 2013.

[Pol12] Elijah Polak. *Optimization: algorithms and consistent approximations*, volume 124. Springer Science & Business Media, 2012.

[PPM12] Chonhyon Park, Jia Pan, and Dinesh Manocha. ITOMP: Incremental Trajectory Optimization for Real-Time Replanning in Dynamic Environments. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[PPM20] Chonhyon Park, Jae Sung Park, and Dinesh Manocha. Fast and bounded probabilistic collision detection for high-dof robots in dynamic environments. In *Algorithmic Foundations of Robotics XII*, pages 592–607. Springer, 2020.

[PR14] Michael A Patterson and Anil V Rao. GPOPS-II: A MATLAB software for solving multiple-phase optimal control problems using hp-adaptive Gaussian quadrature collocation methods and sparse nonlinear programming. *ACM Transactions on Mathematical Software (TOMS)*, 41(1):1–37, 2014.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3.2, page 5. Kobe, Japan, 2009.

[QK93] Sean Quinlan and Oussama Khatib. Elastic bands: Connecting path planning and control. In *[1993] Proceedings IEEE International Conference on Robotics and Automation*, pages 802–807. IEEE, 1993.

[R+64] Walter Rudin et al. *Principles of mathematical analysis*, volume 3. McGraw-hill New York, 1964.

[Raj11] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[RBR16] Charles Richter, Adam Bry, and Nicholas Roy. Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. In *Robotics Research*, pages 649–666. Springer, 2016.

[RLMK04] Stephane Redon, Ming C Lin, Dinesh Manocha, and Young J Kim. Fast continuous collision detection for articulated models. *ACM Symposium on Solid Modeling and Applications*, 2004.

[SA19] Nils Paul Stephanus Smit-Anseeuw. *Robust and Economical Bipedal Locomotion*. PhD thesis, University of Michigan, 2019.

[SCH+18] Sumeet Singh, Mo Chen, Sylvia L Herbert, Claire J Tomlin, and Marco Pavone. Robust tracking with model mismatch for fast and safe planning: an SOS optimization approach. *arXiv preprint arXiv:1808.00649*, 2018.

[SDH+14] John Schulman, Yan Duan, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, Jia Pan, Sachin Patil, Ken Goldberg, and Pieter Abbeel. Motion planning with sequential convex optimization and convex collision checking. *The International Journal of Robotics Research*, 33(9):1251–1270, 2014.

[SHB14] Dieter Schramm, Manfred Hiller, and Roberto Bardini. *Vehicle Dynamics Modeling and Simulation*. Springer, 2014.

[SNGA19] Andrew Singletary, Petter Nilsson, Thomas Gurriet, and Aaron D Ames. Online active safety for robotic manipulators. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 173–178. IEEE, 2019.

[SSSS17] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374*, 2017.

[Str82] Gilbert Strang. The Width of a Chair. *The American Mathematical Monthly*, 89(8):529–534, 1982.

[SVBT14] Victor Shia, Ram Vasudevan, Ruzena Bajcsy, and Russ Tedrake. Convex computation of the reachable set for controlled polynomial hybrid systems. In *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pages 1499–1506. IEEE, 2014.

[SYA19] Stanley W Smith, He Yin, and Murat Arcak. Continuous abstraction of nonlinear systems using sum-of-squares programming. *arXiv preprint arXiv:1909.06468*, 2019.

[TK20] Ezra Tal and Sertac Karaman. Accurate tracking of aggressive quadrotor trajectories using incremental nonlinear dynamic inversion and differential flatness. *IEEE Transactions on Control Systems Technology*, 2020.

[TLEH20] Jesus Tordesillas, Brett T Lopez, Michael Everett, and Jonathan P How. FASTER: Fast and Safe Trajectory Planner for Flights in Unknown Environments. *arXiv preprint arXiv:2001.04420*, 2020.

[TMTR10] Russ Tedrake, Ian R Manchester, Mark Tobenkin, and John W Roberts. LQR-trees: Feedback motion planning via sums-of-squares verification. *The International Journal of Robotics Research*, 29(8):1038–1052, 2010.

[TPM13] Mark M Tobenkin, Frank Permenter, and Alexandre Megretski. Spotless polynomial and conic optimization, 2013.

[UAB+08] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[UMT15] UMTRI. Mcity Grand Opening. *Research Review*, 46(3), 2015.

[VDBGLM11] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer, 2011.

[VG18] Andreas Voelz and Knut Graichen. Computation of collision distance and gradient using an automatic sphere approximation of the robot model with bounded error. In *ISR 2018; 50th International Symposium on Robotics*, pages 1–8. VDE, 2018.

[VKL+19] Sean Vaskov*, Shreyas Kousik*, Hannah Larson, Fan Bu, James Ward, Stewart Worrall, Matthew Johnson-Roberson, and Ram Vasudevan. Towards Provably Not-at-Fault Control of Autonomous Robots in Arbitrary Dynamic Environments. In *Robotics: Science and Systems*, 2019.

[VLK+19] Sean Vaskov, Hannah Larson, Shreyas Kousik, Matthew Johnson-Roberson, and Ram Vasudevan. Not-at-Fault Driving in Traffic: A Reachability-Based Approach. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 2785–2790. IEEE, 2019.

[VSG+12] Ram Vasudevan, Victor Shia, Yiqi Gao, Ricardo Cervera-Navarro, Ruzena Bajcsy, and Francesco Borrelli. Safe semi-autonomous control with enhanced driver modeling. In *2012 American Control Conference (ACC)*, pages 2896–2903. IEEE, 2012.

[VSK+19] Sean Vaskov, Utkarsh Sharma, Shreyas Kousik, Matthew Johnson-Roberson, and Ramanarayan Vasudevan. Guaranteed safe reachability-based trajectory design for a high-fidelity model of an autonomous passenger vehicle. In *2019 American Control Conference (ACC)*, pages 705–710. IEEE, 2019.

[War89] Charles W Warren. Global path planning using artificial potential fields. In *Proceedings, 1989 International Conference on Robotics and Automation*, pages 316–321. Ieee, 1989.

[WB09] Yang Wang and Stephen Boyd. Fast model predictive control using online optimization. *IEEE Transactions on control systems technology*, 18(2):267–278, 2009.

[WFK+16] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. Fetch and freight: Standard platforms for service robot applications. In *Workshop on Autonomous Mobile Service Robots*, 2016.

[WLMK20] Andreas Waechter, Carl Laird, F Margot, and Y Kawajir. Introduction to IPOPT: A tutorial for downloading, installing, and using IPOPT. In *Revision*. COIN-OR, 2020.

[WS16] Guofan Wu and Koushil Sreenath. Safety-critical control of a planar quadrotor. In *2016 American Control Conference (ACC)*, pages 2252–2258. IEEE, 2016.

[XGTA17] Xiangru Xu, Jessy W Grizzle, Paulo Tabuada, and Aaron D Ames. Correctness guarantees for the composition of lane keeping and adaptive cruise control. *IEEE Transactions on Automation Science and Engineering*, 15(3):1216–1229, 2017.

[YLJS18] Sangyol Yoon, Dasol Lee, Jiwon Jung, and David Hyunchul Shim. Spline-based RRT* Using Piecewise Continuous Collision-checking Algorithm for Car-like Vehicles. *Journal of Intelligent and Robotic Systems*, 90(3-4):537–549, 2018.

[YMCA13] Shuyou Yu, Christoph Maier, Hong Chen, and Frank Allgöwer. Tube MPC scheme based on robust control invariant set with application to Lipschitz nonlinear systems. *Systems & Control Letters*, 62(2):194–200, 2013.

[Zha20] Pengcheng Zhao. *Fast, Safe, and Optimal Motion Planning for Bipedal Robots*. PhD thesis, University of Michigan – Ann Arbor, 2020.

[ZRD+13] Matt Zucker, Nathan Ratliff, Anca D Dragan, Mihail Pivtoraiko, Matthew Klingensmith, Christopher M Dellin, J Andrew Bagnell, and Siddhartha S Srinivasa. Chomp: Covariant hamiltonian optimization for motion planning. *The International Journal of Robotics Research*, 32(9-10):1164–1193, 2013.