Kinodynamic Planning with $\mu$-Calculus Specifications

by

Luc Larocque

A thesis presented to the University of Waterloo in fulfillment of the thesis requirement for the degree of Master of Mathematics in Applied Mathematics

Waterloo, Ontario, Canada, 2018

© Luc Larocque 2018
Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Stephen Smith
Associate Professor, Dept. of Electrical and Computer Engineering

Supervisor: Jun Liu
Associate Professor, Dept. of Applied Mathematics

Internal Member: Brian Ingalls
Associate Professor, Dept. of Applied Mathematics
I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.
Abstract

Motion planning problems involve determining appropriate control inputs to guide a system towards a desired endpoint. Sampling-based motion planning was developed as a technique for discretizing the state space of systems with complex environments. This makes the sampling-based method especially useful in robotics, where robots are expected to perform tasks in unknown, changing, or cluttered environments. On the other hand, temporal logic presents a means of prescribing the desired behaviour of a system. In the area of formal methods, researchers seek to solve problems in such a way that synthesized solutions provably satisfy a given temporal logic specification. In this thesis, we investigate combining the flexibility of sampling-based planning with the ability to specify the high-level behaviour of an autonomous system with the temporal logic known as $\mu$-calculus.

While using temporal logic specifications with motion planning has been heavily researched, reliance on an available steering function is often impractical and suited only to basic problems with linear dynamics. This is because a steering function is a solution to an optimal two-point boundary value problem (OBVP); thus far, mathematicians have yet to find analytic solutions to such problems in all but the simplest of cases. Addressing this issue, we have developed a means of using the motion planning algorithm SST* in combination with a local model checking procedure to solve kinodynamic planning problems with deterministic $\mu$-calculus specifications without using a steering function. The procedure involves combining only the most pertinent information from multiple Kripke structures in order to create one abstracted Kripke structure storing the best paths to all possible proposition regions of the state-space. A linear-quadratic regulator (LQR) feedback control policy is then used to track these best paths, effectively connecting the trajectories found from multiple Kripke structures. Simulations demonstrate that it is possible to satisfy a complex liveness specification involving infinitely often reaching specified regions of state-space using only forward propagation of the system dynamics.

We proceed to repurpose this tool for real-time quadrotor motion planning with temporal logic specifications. The dynamical system is derived, and a real-time planning framework is presented based on a variant of the FMT* planning algorithm. Despite requiring a steering function, an argument is presented which allows finding OBVP solutions only for an approximation of the full dynamics. The notion of an abstracted Kripke structure is then applied in the context of quadrotor kinodynamic planning, allowing for rapid model checking and ensuring high-quality feasible solutions satisfying a given deterministic $\mu$-calculus specification.
Acknowledgments

Many thanks to my ever-supportive and helpful supervisor, Jun Liu. Your positivity and guidance gave me the motivation I needed to succeed, while allowing me the independence to follow the research path that suited me best. Thank you also to all of the members of the Hybrid Systems Lab. Yinan Li, Milad Farsi, Chuanzheng Wang, Riley Brooks, and Kevin Church, you have all contributed so much to my Master’s experience with insightful conversation, interesting presentations, and utmost kindness. Lastly, to Stephen Smith and Brian Ingalls, I am very grateful for the time you dedicated and the comments you provided as members of my defence committee.
Dedication

This is dedicated to my officially-expanded family and to Maša: you have made my journey through graduate studies an absolute pleasure.
# Table of Contents

List of Figures ix

Abbreviations x

1 Introduction 1

1.1 Motion Planning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.1.1 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.1.2 Sampling-Based Kinodynamic Planning . . . . . . . . . . . . . . . . . 3

1.2 Temporal Logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.4 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2 Preliminaries 8

2.1 $\mu$-Calculus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.1.1 Modal $\mu$-Calculus . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.1.2 Deterministic $\mu$-Calculus . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.1.3 Tarski-Knaster Theorem . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.1.4 Specification Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.2 SST* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.3 FMT* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>RRT sampling and connection procedure</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of a Kripke structure</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>$\mu$-Calculus semantics example</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>SST Flappy Bird Example</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>FMT* Example</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>FMT* Diagram</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Abstracted Kripke Structure</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Double Integrator Example — Three Trees using SST*</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Double Integrator Example — Simulation Results</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Inertial and body-fixed frames</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Quadrotor simulation tree</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>Quadrotor simulation waypoints</td>
<td>66</td>
</tr>
<tr>
<td>4.4</td>
<td>Quadrotor simulation smooth trajectory</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>Quadrotor step-response</td>
<td>67</td>
</tr>
<tr>
<td>4.6</td>
<td>Quadrotor tracking errors</td>
<td>68</td>
</tr>
<tr>
<td>4.7</td>
<td>Quadrotor smooth path tracking</td>
<td>69</td>
</tr>
<tr>
<td>4.8</td>
<td>Quadrotor planning with temporal logic specifications</td>
<td>71</td>
</tr>
<tr>
<td>4.9</td>
<td>Abstracted Kripke structure for quadrotor paths</td>
<td>72</td>
</tr>
</tbody>
</table>
Abbreviations

CTL  Computation Tree Logic 5, 18, 28

EST  Expansive Space Trees 19

FMT* Fast Marching Tree 6–8, 24–27, 40, 71, 73

LQR  Linear Quadratic Regulator 6, 7

LTL  Linear Temporal Logic 5, 13, 18, 28, 29, 31

NED North-East-Down 40, 62

OBVP optimal boundary value problem 6, 19, 30, 46–51, 53, 64, 71

PRM Probabilistic Roadmap 19, 24

QP quadratic program 54

RRT Rapidly-exploring Random Tree 4, 19, 24

SLAM simultaneous localization and mapping 3

SST Stable Sparse RRT 6–8, 19, 20, 22–24, 30, 31, 33–37, 40, 49, 52, 70, 73, 74

SVM Support Vector Machine 47, 51, 52

UAV unmanned aerial vehicle 2, 3, 42
Chapter 1

Introduction

1.1 Motion Planning

Planning is a fundamental problem in robotics: mobile robots must be able to determine how to move in order to perform tasks. According to LaValle in his titular book on planning algorithms, converting high-level specifications into low-level descriptions of how a robot ought to move is what is generally referred to as motion planning [26]. He further states motion planning, in modern control theory literature, refers to the generation of inputs to a dynamical system which drive it from an initial state to a specified goal state (or set).

In general, motion planning solves problems involving a state space, which is the set of all possible states in which a system could find itself. Such a space could be finite, like in the case of Rubik’s cube with finitely many configurations, or infinite, such as train with both a position and velocity that can vary continuously in the domain of real numbers. Note that time also plays a crucial role in both of these examples: the Rubik’s cube allows moves in succession, in some order, and the train’s location and current velocity depend on its past position and velocity. Lastly, in order to plan, one must be able to affect the system, i.e., change the state, in the same way. Some systems, like the train, abide by a set of dynamics which govern how the system changes. A train on a steep hill will roll down the hill if it does not have sufficient momentum to crest over the top. However, if we allow the system to accept an input or control, then the system can be altered to act in a desirable way. Being able to set the engine to full-throttle may make the difference between getting to the destination and ending up stuck at the foot of the hill. For continuous time systems, the dynamics are modeled with ordinary differential equations. Even for systems without dynamics, like with the Rubik’s cube (assuming we are not concerned with how
the faces of the cube are rotated), there must be a way to specify exactly how an action affects the state of the system.

This thesis will focus only on continuous-time motion planning problems, where the dynamics are modeled by a control system that consists of a set of ordinary differential equations modeling the dynamics of the system, an initial condition, a set of allowed states, and a set of admissible controls. Note that this does not rule out the possibility of uncertainties. Dynamical systems model reality but do not necessarily do so with complete accuracy, and disturbances in the environment may have an effect on the behaviour of a system. A well-designed motion planning algorithm is robust under uncertainties, using feedback control to reject disturbances.

1.1.1 Examples

Motion planning has an immense variety of applications in both virtual and robotic systems. Moreover, the potential for autonomous robots to improve human living conditions is vast, and as yet not fully understood. One especially disruptive emerging technology is autonomous vehicles: cars and trucks that are able to drive from an initial position to a goal location with minimal or no human input [42]. These autonomous vehicles have been gaining popularity in recent years, especially with the media hype of Google and Tesla bringing self-driving cars into the public eye; however, the first research on this topic began in the 20th century. By 1995, Todd Jochem and Dean Pomerleau completed a 2,797 mile journey across America in a van using neural networks to design a vision-based partially automated driving system\(^1\) [18]. This accomplishment demonstrated level 2 automation under the SAE International Standard J3106 [15], which falls short of being described as an “automated driving system” as a human driver is still essential. More recent advances have brought autonomous vehicles to level 3 with Google’s self-driving car having over 500,000 miles of autonomous driving in 2012 [32]. Level 3 is labeled as “conditional automation”, meaning for certain driving modes the automated driving system can control all aspects of driving with the caveat that a human driver be on standby to respond to requests to intervene. In jumping from level 2 to level 3, a human driver becomes non-essential to the driving task, except as a fallback.

Another practical example of motion planning is controlling an unmanned aerial vehicle (UAV) system, such as a quadrotor. Due to their scalable size and high maneuverability, quadrotors are used for an ever-increasing range of tasks, from aerial photography, to

---
\(^1\)The trip was titled “No Hands Across America” since the only human input involved braking and accelerating; steering was performed completely autonomously.
mapping dangerous, cluttered, or unexplored regions with simultaneous localization and mapping (SLAM), to light shows from a fleet of Intel’s quadrotors, and even for quickly delivering small parcels [49, 36]. Each of these tasks requires a means of determining where and how the quadrotor should move, and advances in motion planning will allow for even more complex and intricate maneuvers, and therefore more applications. However, planning for quadrotors can be quite difficult because, like cars, they are non-holonomic and subject to differential constraints. Since they can maneuver in 3D space instead of being restricted to a 2D surface, as is the case for ground vehicles, quadrotors are described by a 12-dimensional state space (position, velocity, orientation, and rotational velocity). This often means that much computational effort is involved in motion planning for UAV systems. Furthermore, given that the dynamics governing quadrotor motion are highly nonlinear, motion planning is all the more difficult, from path generation to trajectory tracking. Many of these issues will be addressed in Chapter 4, which presents a framework for real-time motion planning of quadrotors, based on work by Allen et al. [2].

Overall, the field of robotics is continuing to grow, and with it, motion planning is becoming all the more relevant and important in day-to-day life. Forklifts are being automated to move merchandise around in warehouses, and vacuum cleaners have become small disks that roam around the home like robotic pets. In the field of agriculture, robots are improving the lives of farmers by autonomously targeting and removing weeds that negatively impact crops [45]. The recurring theme in all of these examples is that automation, along with the essential component of motion planning, is leading the way in reducing the need for human labour. Robots are becoming increasingly capable of performing complex tasks, especially with the concurrent rise of machine learning and artificial intelligence, and advances in motion planning are leading to improved performance [11], online reactivity to dynamic obstacles [2], and better guarantees [30] in all areas of robotics.

### 1.1.2 Sampling-Based Kinodynamic Planning

The sampling-based approach to motion planning has become particularly popular in robotics applications since it avoids having to explicitly represent the environment and its obstacles. In essence, the strategy is to discretize the state space by sampling, thereby constructing a graph which explores a representative portion of the space. Such problems can be solved geometrically, meaning solutions only find a collision-free path in space and do not take into account feasibility, or using kinodynamic planning, which considers the system dynamics and differential constraints to ensure the robot can feasibly accomplish the planned task. In this thesis, emphasis is placed on kinodynamic planning.
The general approach to sampling-based motion planning begins with choosing a sampling scheme; that is, samples of the state space are to be taken either deterministically, or according to some probability distribution. These samples are then used to generate new nodes on a graph, where the details involved in connecting each new node to the existing graph vary based on the sampling-based algorithm being applied. For example, as depicted in Figure 1.1, the popular motion planning algorithm Rapidly-exploring Random Tree (RRT) samples a random point in state space, finds the nearest existing point of the tree, and steers from said nearest point towards the sample up to some maximum distance, $\delta_{\text{max}}$. The result is a newly added edge (transition from one state to another) and node (state) in the tree. The fact that steering is required means there must exist a function which optimally connects two states. This requirement is is quite strict, and will be discussed in Chapter 2 and Chapter 3.
An important concept in motion planning literature is completeness, and we say that an algorithm is complete if, for any input, it correctly returns in finite time whether or not there exists a solution to the path planning problem [26]. Clearly, due to the nature of sampling-based algorithms, completeness cannot be achieved. There are, however, weaker notions of completeness which can be useful to study. One common alternative when using a random sampling scheme is the notion of probabilistic completeness, which means that the probability that the algorithm finds a solution — if one exists — converges to 1 as the number of samples tends to infinity. Similarly, we say that a motion planning algorithm is asymptotically optimal if the probability of finding an optimal solution (based on a chosen cost function) approaches one as the number of samples approaches infinity [20].

1.2 Temporal Logic

Motion planning occurs on various levels of abstraction. A hierarchical control structure would typically have some “vague”, high-level description of what the robot or system is supposed to do, guiding the desired behaviour. Below that there is a model, usually encompassing system dynamics, with a control architecture that prescribes how information is passed along in a series of inputs and outputs, which ultimately generates a motion plan to be followed. At the lowest levels, algorithms are used to put the motion plan in action, using a tracking controller — with some form of feedback to improve robustness — that determines exactly what inputs ought to be applied to maneuver along the planned path trajectory.

Temporal logics are the language used to express the high-level specifications that preside over the control hierarchy, dictating the user-desired behaviour. Most instances of motion planning seek simply to move from one location to another, without any other instructions (except to avoid obstacles). This goal is often hard-coded into the algorithms designed for solving motion planning problems. However, there exists a much broader realm of possibility that uses model checking to determine whether a solution satisfies a given specification. Allowing users to specify the high-level expectations of robot behaviour opens many avenues for real-world applications, not to mention the benefit of having performance guarantees from the use of formal methods [30]. We choose to work with \( \mu \)-calculus, a highly expressive temporal logic which permits more diverse and complex specifications than the most widely used temporal logics, including Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and extensions thereof [19]. One further advantage proffered by the use of (deterministic) \( \mu \)-calculus is the relative ease and elegance of model checking that it permits. An in-depth background on \( \mu \)-calculus is provided Chapter 2.
1.3 Contributions

Summarizing the contributions of this thesis, we have successfully solved a kinodynamic planning problem with temporal logic specifications without the need for a solution to an optimal boundary value problem (OBVP), also called a steering function. While using temporal logic specifications with motion planning has been heavily researched, e.g., [3, 4, 19, 30, 47], it is often difficult or impossible to find a steering function, allowing motion planning only for simple dynamical systems. Addressing this issue, we have combined the probabilistically complete and asymptotically optimal variant of the kinodynamic planning algorithm Stable Sparse RRT (SST), called SST*, (see Section 2.2) from [28] with the model checking procedure from [19] to create a motion planning algorithm with deterministic μ-calculus specifications that does not rely on a steering function. We further introduce the notion of an abstracted Kripke structure which stores the most cost-efficient paths reaching the desired proposition regions as a single edge in a relatively small graph. This is achieved by merging information obtained from multiple Kripke structures, containing all of the relevant detail, into a single simplified structure. The solution trajectories necessary in satisfying the given specification are then combined and tracked using Linear Quadratic Regulator (LQR) feedback controller.

Furthermore, we use the same notion of an abstracted Kripke structure to solve the highly nonlinear quadrotor planning problem with temporal logic specifications. A similar approach to the above is used, however we leverage the real-time framework proposed in [2] which uses Fast Marching Tree (FMT*) to open the possibility of generating trajectories satisfying a deterministic μ-calculus specification while online. This is important because online motion planning is not feasible with the incremental SST* method.

Further contributions are made in providing much-needed detail and educationally significant clarifications to some literature in this field. An introductory overview of μ-calculus is conducted in Chapter 2 which is intended to be both detailed and understandable. Many early developments on the subject, e.g., [9, 7, 5, 8], are amalgamated into one section providing readers with a deep intuition on the subject of temporal logic using μ-calculus. In addition, an original example is included to illustrate the semantics of μ-calculus specifications.

We also provide further detail regarding quadrotor motion planning, including quadrotor dynamics [34], real-time planning [2], and geometric control methods [27], in Chapter 4. We take a deeper look at the mathematics involved in deriving many of the equations that arise in kinodynamic planning for quadrotors, deriving and justifying many equations that appear without explanation in the above cited works. Moreover, the provided descrip-
tions should be sufficient in guiding the implementation of the algorithms and frameworks discussed herein.

1.4 Overview

The rest of this thesis is structured as follows. Chapter 2 introduces many of the important and recurring concepts discussed throughout. The syntax and semantics of $\mu$-calculus are provided, and a crucial theorem used in model checking is stated with proof. Then, the fragment of $\mu$-calculus we will use, called deterministic $\mu$-calculus, is defined. Many common specifications are described in detail to provide the foundations of the language of our temporal logic specifications. We then move to a description of the kinodynamic planning algorithm SST* which we will use in Chapter 3. The final section of this chapter provides a detailed look at the FMT* algorithm, which we use for quadrotor motion planning.

In Chapter 3, we present novel research on solving kinodynamic planning problems while satisfying deterministic $\mu$-calculus specifications without requiring a steering function. A problem statement is provided along with a detailed description of the meta-algorithm KinoSpecPlan which combines using SST* with a local deterministic model checking algorithm as well as LQR tracking. A complex liveness specification is then shown to be satisfied in an example that uses this method.

Chapter 4 focuses on the application of kinodynamic planning to quadrotors. The dynamics of a quadrotor system are derived, and the problem of planning with deterministic $\mu$-calculus specifications is stated. We then outline all of the necessary components for a real-time motion planning framework and demonstrate some simulation results for the task of transiting from an initial state to a goal state. Lastly, the ideas from Chapter 3 are used in the context of quadrotor planning to produce trajectories satisfying temporal logic specifications.

Lastly, Chapter 5 offers closing remarks as well as directions for future work.
Chapter 2

Preliminaries

In this chapter, we introduce some core background concepts that arise throughout the thesis. First, we detail the notation and specify the semantics of our choice of temporal logic, \( \mu \)-calculus, and provide the model checking tools necessary to use it. A description of the incremental kinodynamic planning algorithm SST* follows, along with a description of the various functions used to implement the planning algorithm. This is contrasted with FMT*, which is another motion planning algorithm that is not incremental.

2.1 \( \mu \)-Calculus

We begin by defining the syntax and semantics of the full, modal \( \mu \)-calculus. A simple example Kripke structure is provided to build some intuition surrounding the notation and meaning of some common \( \mu \)-calculus formulas. We then describe a fragment of \( \mu \)-calculus called deterministic \( \mu \)-calculus, which we will be using for the motion planning procedure in Chapter 3. Finally, we present many examples of typical deterministic \( \mu \)-calculus formulas, describing how to parse and thoroughly understand each one.

2.1.1 Modal \( \mu \)-Calculus

First, an atomic proposition is a declarative statement that is either true or false, and which cannot be further split into smaller statements. An example of an atomic proposition might be “in free space”, where a state satisfies the atomic proposition if and only if it lies in the defined free space for a problem. On the other hand, the statement “in free space AND
not in goal region” is not an atomic proposition, as it can be further deconstructed into the simpler statements “in free space” and “in goal region”. As this example demonstrates, more complex statements use logical connectives such as conjunction (∧, logical AND), disjunction (∨, logical OR), and negation (¬, logical NOT). We will also be using the modal operators ♦ and □, representing the existential and universal successor operators, respectively. Lastly, we use the symbols μ and ν as least and greatest fixed point operators, respectively.

Figure 2.1: Example of a Kripke structure, with obstacles shown in red, free space shown in white, and the goal set shown in green.

Let Π be the set of atomic propositions, and let VAR be the set of variables, where a variable is a placeholder that often appears bound to a fixed point operator. We define the following structure as in [19].

**Definition 1.** A *Kripke structure* \( K \) over the set of atomic propositions \( \Pi \) is a tuple \((S, S_0, R, \mathcal{L})\) where \( S \) is a finite set of states, \( S_0 \subseteq S \) is a set of initial states, \( R \subseteq S \times S \) is a binary relation which indicates a means of transiting from one state to another (usually
represented as the edges in a directed graph), and \( \mathcal{L} : S \rightarrow 2^\Pi \) is a labeling function, mapping each state to the subset of propositions that it satisfies.

In essence, a Kripke structure is a graph with edges representing available transitions between nodes, which represent states that have been added to the graph via some sampling scheme. The distinction to be made between a Kripke structure and any other graph representation is that every node on the graph has an associated label which defines which atomic propositions that node satisfies. Figure 2.1 is an example of a graph that could represent a Kripke structure. Here, the set of states, \( S \), is represented by the nodes of the graph, the set of initial states is given by the singleton containing the bottom-left node, and the set of relations is represented by the edges. Define \( \pi_f \) to be the atomic proposition “in free space”, \( \pi_g \) to mean “in goal”, and \( \pi_o \) to mean “in obstacle”. The labeling function would then simply specify, for every node, the subset of \( \Pi = \{ \pi_f, \pi_g, \pi_o \} \) that it satisfies.

Define the set of valid \( \mu \)-calculus formulas \( (L_\mu) \) inductively, as follows:

- the symbols \( \text{True} \) and \( \text{False} \) are formulas;
- every \( p \in \Pi \) and \( X \in \text{VAR} \) is a formula;
- if \( \phi, \psi \) are formulas, then \( \neg \phi, \phi \land \psi, \text{and } \phi \lor \psi \) are also formulas;
- if \( \phi \) is a formula then \( \Diamond \phi \) and \( \Box \phi \) are formulas;
- if \( \phi[X] \) is a formula where \( \phi[X] \) is syntactically monotone in \( X \), then \( \mu X.\phi \) and \( \nu X.\phi \) are formulas.

As in [12], we write \( \phi[X] \) to indicate that \( \phi \) contains an occurrence of \( X \), i.e., the variable \( X \) appears in formula \( \phi \). In the last bullet above, \( \mu X.\phi \) (similarly, \( \nu X.\phi \)) binds the variable \( X \) in \( \phi \) to the fixed point operator. Note that the notation for fixed point operators always takes the form \( \sigma X.\phi \), where \( \sigma \) is one of \( \mu \) or \( \nu \), \( X \) is any variable name, and a dot separates the operator from the formula on which it acts. In this general case, the formula \( \phi = \phi[X] \) is a mu-calculus formula that is syntactically monotone in the variable \( X \). We define syntactic monotonicity using the following definitions.

**Definition 2.** Given a \( \mu \)-calculus specification, \( \phi \), we say that a variable \( X \) is *positive* (respectively, *negative*) if it occurs under the scope of an even (respectively, odd) number of negations in formula \( \phi \).

**Definition 3.** A subformula \( X \) is *pure* in \( \mu \)-calculus specification \( \phi \) if all of its occurrences have the same polarity (i.e., all occurrences of \( X \) are positive or all occurrences of \( X \) are negative).
Definition 4. The formula $\phi[X]$ is **syntactically monotone** in $X$ if and only if $X$ occurs with pure polarity in $\phi$.

$\mu$-calculus formulas are interpreted with respect to a Kripke structure $K = (S, S_0, R, \mathcal{L})$ and an environment (also called an *evaluation*), $e : \text{VAR} \to 2^S$, which initializes (i.e., assigns a subset of $S$ to) each free variable, where a free variable is defined in the usual sense and a variable is otherwise said to be bound by a fixed point operator. For every $L_\mu$ formula $\phi$, given a Kripke structure, $K$, define $\lceil \phi \rceil_K \subseteq S$ to be the set of states of $S$ satisfying proposition $\phi$ (note that the subscript $K$ is often omitted when the Kripke structure being used is clear, and the superscript $e$ is omitted when $\phi$ contains no free variables). The semantics of the formulas is determined inductively by the following, where $p \in \Pi$, $X \in \text{Var}$, and $\phi, \psi \in L_\mu$ are arbitrary formulas [46].

$$
\begin{align*}
[\text{False}]_K &= \emptyset \\
[\text{True}]_K &= S \\
[p]_K &= \{s \in S : p \in L(s)\} \\
[\neg p]_K &= S \setminus [p] \\
[\phi \lor \psi]_K^e &= [\phi]_K^e \cup [\psi]_K^e \\
[\phi \land \psi]_K^e &= [\phi]_K^e \cap [\psi]_K^e \\
[\lozenge \phi]_K^e &= \text{Pre}^e_{K,\exists}(\phi) \\
[\Box \phi]_K^e &= \text{Pre}^e_{K,\forall}(\phi) \\
[\mu X. \phi]_K^e &= \bigcap \{A \subseteq S : [\phi]_K^{e[X \leftarrow A]} \subseteq A\} \\
[\nu X. \phi]_K^e &= \bigcup \{A \subseteq S : [\phi]_K^{e[X \leftarrow A]} \supseteq A\}
\end{align*}
$$

The existential and universal *predecessor* functions $\text{Pre}^e_{K,\cdot} : L_\mu \to S$ map a $\mu$-calculus formula to the set of states which immediately precede (i.e., have transitions to) states satisfying said formula in the Kripke structure $K$ and under evaluation $e$:

$$
\begin{align*}
\text{Pre}^e_{K,\exists}(\phi) &= \{s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \land s' \in [\phi]_K^e\} \\
\text{Pre}^e_{K,\forall}(\phi) &= \{s \in S : \forall s' \in S, (s, s') \in R \implies s' \in [\phi]_K^e\}
\end{align*}
$$

In words, $\text{True} = (p \lor \neg p)$ holds for all states in $S$, and $\text{False} = (p \land \neg p)$ does not hold for any state in $S$; disjunction and conjunction of formulas is equivalent to the union and intersection of the sets which satisfy them, respectively; $\lozenge$ is the *existential successor* (or “next”) operator, and $\Box$ is the *universal successor* operator; lastly, $\mu$ and $\nu$ are the *least* and *greatest fixed-point* operators, respectively, where $e[X \leftarrow A]$ is a modified evaluation function which maps $X$ to $A$, i.e., $e[X \leftarrow A](X) = A$. To help build a more intuitive understanding of these last four semantic definitions, a simple example is provided (Example 1).
Example 1. This example illustrates the semantics presented above. Refer to Figure 2.2. Define atomic proposition \( p \) to be the boolean value corresponding to “in gray region”. We can specify the labeling function for each node, writing \( \mathcal{L}(s_p) = \{ p \} \) for nodes \( s_p \in \{ a, b, c \} \), and \( \mathcal{L}(s) = \emptyset \) for nodes \( s \in \{ d, e, f, g \} \). Note that the environment superscript \( e \) is omitted as we are not using any free variables, and we use the generic Kripke structure \( K = (S, S_0, R, \mathcal{L}) \).

\[
\begin{align*}
[p]_K &= \{ s \in S : p \in \mathcal{L}(s) \} = \{ a, b, c \} \\
[\neg p]_K &= S \setminus \{ p \} = \{ d, e, f, g \} \\
[\Diamond p]_K &= \text{Pre}^e_{K,\exists}(p) = \{ a, c, d, f \} \\
[\Box p]_K &= \text{Pre}^e_{K,\forall}(p) = \{ a, c \} \\
[\mu X.(p \lor \Diamond X)]_K &= \bigcap \{ A \subseteq S : [p \lor \Diamond X]_K^{[X \rightarrow A]} \subseteq A \} = \{ a, b, c, d, f \} \\
[\nu X.(p \land \Diamond X)]_K &= \bigcup \{ A \subseteq S : [p \land \Diamond X]_K^{[X \rightarrow A]} \supseteq A \} = \{ a, c \}
\end{align*}
\]

- \([p]_K\) is the set of nodes satisfying \( p \) (nodes in the gray region).
- \([\neg p]_K\) is the complement of \([p]_K\) in \( S \) (nodes that are not in the gray region).
- \([\Diamond p]_K\) contains those nodes that have a transition to a node in the gray region.
- \([\Box p]_K\) contains only nodes such that every transition leads to a node in the gray region.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{example1}
\caption{\( \mu \)-Calculus semantics Example 1.}
\end{figure}

12
[µX.(p ∨ ♦X)]_K is the reachability specification which determines whether there exists a sequence of transitions reaching a state satisfying p (see Section 2.1.4). In this case, the resulting set consists of the nodes that are in the gray region or that can follow transitions to eventually reach a node in the gray region.

[νX.(p ∧ ♦X)]_K is the safety specification which determines whether there exists a sequence of transitions guaranteeing that proposition p is always satisfied (see Section 2.1.4). In this case, the resulting set consists of the nodes that are in the gray region and can always take a transition to a node that is also in the gray region.

In order to develop a means of evaluating expressions containing fixed-point operators, we rely on the Tarski-Knaster theorem from set theory. The next sections introduce a fragment of modal µ-calculus called deterministic µ-calculus, which we will use in the formulation of the required theorem.

2.1.2 Deterministic µ-Calculus

We will focus our efforts on deterministic µ-calculus, which admits efficient model checking algorithms and is more expressive than the commonly used temporal logic LTL [19, 8]. Deterministic µ-calculus imposes some restrictions on syntax, notably that only atomic propositions may be negated, conjunction can only occur between a formula and an atomic proposition, and the universal successor operator □ is omitted. We may write this syntax succinctly in Backus-Naur form as follows:

\[ \phi ::= p | ¬p | X | p \land \phi | ¬p \land \phi | \phi \lor \phi | ♦\phi | µX.\phi | νX.\phi \]

where \( p \in \Pi \) and \( X \in \text{VAR} \). We denote the set of all deterministic µ-calculus formulas by \( L_1 \). Note that by restricting negation to atomic propositions only, we ensure that every formula in \( L_1 \) is syntactically monotone in its variables. Furthermore, this fragment of µ-calculus does not express any temporal branching; all found solutions flow forward in time, as is necessary for a motion planning trajectory.

2.1.3 Tarski-Knaster Theorem

The Tarski-Knaster fixed point theorem (Theorem 1) we introduce in this section makes an important statement about complete lattices and their fixed-points [40]. We make use of this theorem to formulate an algorithm which computes the least or greatest fixed
points for the set of states satisfying a $\mu$-calculus proposition containing a least or greatest fixed point operator. Note that if a proposition does not contain fixed point operators or the existential successor operator, it is easy to evaluate whether any given state satisfies the proposition. Formulas including the existential successor operator are not much more difficult, although they do require searching for successor states along the set of relations. However, what is truly interesting is the fact that we can use the Tarski-Knaster theorem to easily perform model checking for specifications that involve fixed point operators [19]. We begin by defining the necessary concepts.

**Definition 5.** Let $(X, \leq)$ be a partially ordered set, and let $A \subseteq X$. Then $\bigvee A$ ($\bigwedge A$) denotes the least upper bound (respectively, greatest lower bound) of $A$ with respect to $\leq$, if it exists. We say that $X$ is a complete lattice if, for every $A \subseteq X$, both $\bigvee A$ and $\bigwedge A$ exist in $X$.

**Example 2.** Consider $(\mathcal{P}(X), \subseteq)$ for any set $X$, where $\mathcal{P}(X)$ denotes the power set of $X$. Note that $\forall A \subseteq \mathcal{P}(X), \bigvee A = \bigcup A$, since the union of all the elements of $A$ gives the smallest set which completely contains the elements of each subset in $A$, and this union must be an element of $\mathcal{P}(X)$. Similarly, $\bigwedge A = \bigcap A \in \mathcal{P}(X)$. Thus, $(\mathcal{P}(X), \subseteq)$ is a complete lattice.

**Definition 6.** Let $(A, \leq_A)$ and $(B, \leq_B)$ be partially ordered sets. A function $f : A \to B$ is monotone if $a_1 \leq_A a_2 \implies f(a_1) \leq_B f(a_2)$. A point $a \in A$ is a fixed point of a function $f : A \to B$ if $f(a) = a$, and we denote the set of all fixed points of $f$ by $\text{fix}(f)$.

**Theorem 1.** (Tarski-Knaster Fixed Point Theorem)
Let $\mathbb{L}$ be a complete lattice and let $F : \mathbb{L} \to \mathbb{L}$ be monotone. Then

1. $\bigvee \{x \in \mathbb{L} \mid x \leq F(x)\} \in \text{fix}(F),$
2. $\bigwedge \{x \in \mathbb{L} \mid F(x) \leq x\} \in \text{fix}(F),$ and
3. $\text{fix}(F)$ is a complete lattice.

**Proof.**
Let $\mu F := \{u \in \mathbb{L} \mid u \leq F(u)\}$, and let $\zeta = \bigvee \mu F$ ($\zeta$ exists since $\mu F \subseteq \mathbb{L}$, a complete lattice). For all $u \in \mu F$, $u \leq \zeta$, so $u \leq F(u) \leq F(\zeta)$, by monotonicity. Thus, $F(\zeta)$ is an upper bound for $\mu F$, and $\zeta$ is the least upper bound, so $\zeta \leq F(\zeta)$. By monotonicity, $F(\zeta) \leq F(F(\zeta))$, so $F(\zeta) \in \mu F$. This implies that that $F(\zeta) \leq \zeta$. Therefore, $\zeta = F(\zeta) \in \text{fix}(F)$. A similar argument can be made for $\bigwedge \{u \in \mathbb{L} \mid F(u) \leq u\}$.

14
Now we show that an arbitrary subset $A \subseteq \text{fix}(F)$ has a least upper bound and a greatest lower bound, thereby proving $\text{fix}(F)$ is a complete lattice. Define $a := \bigvee A$, and $1_L = \bigvee L$. Consider the interval $[a, 1_L] := \{ x \in L \mid a \leq x \leq 1_L \}$, which is a complete lattice. Then if $A$ has a least upper bound in $\text{fix}(F)$, it must lie in $[a, 1_L]$. Note that it suffices to show that $F$ can be restricted to act as a monotone function $F : [a, 1_L] \to [a, 1_L]$, so that we may apply the first result: a monotone function on the complete lattice $[a, 1_L]$ has a least fixed point, and this point is therefore the least upper bound of $A \subseteq \text{fix}(F)$.

Let $x \in A$. Then $x \leq a$ and $x = F(x) \leq F(a) = a$ by monotonicity, so $a \leq F(a)$. Letting $y \in [a, 1_L]$, we can see that $a \leq y$ and $a = F(a) \leq F(y) \leq 1_L$ by monotonicity. This implies $F(y) \in [a, 1_L]$, so we conclude that $F([a, 1_L]) \subseteq [a, 1_L]$ which further implies that we may restrict the domain and co-domain, $F : [a, 1_L] \to [a, 1_L]$. We have thus shown that an arbitrary subset $A \subseteq \text{fix}(F)$ has a least upper bound in $\text{fix}(F)$. It is true for all lattices $L$ that if every sublattice $S \subseteq L$ has a least upper bound $\bigvee S$, then $S$ has a greatest lower bound defined by

$$\bigwedge S = \bigvee \left( \bigcap_{s \in S} \{ x \in L : x \leq s \} \right).$$

Therefore, $\text{fix}(F)$ is a complete lattice.

We will use this theorem to perform model checking over a Kripke structure on propositions with a fixed point operator, $\phi = \sigma X.\psi$, where $\sigma$ is used as a generic symbol representing one of $\mu$ or $\nu$.

**Corollary 2** presents an intuitive algorithm for finding the least and greatest fixed points satisfying a given deterministic $\mu$-calculus proposition with fixed point operators. Note that the complete lattice in question is the power set of the set of states ordered by set inclusion, $(\mathcal{P}(S), \subseteq)$, as in **Example 2**. The proof relies on the fact that deterministic $\mu$-calculus propositions are inherently *syntactically monotone* and therefore monotone in their variables [12], as negation is allowed only on atomic propositions. This implies that for any formula of the form $\sigma X.\psi[X]$, $A \subseteq B$ implies $[\psi]_K^{e[X \leftarrow A]} \subseteq [\psi]_K^{e[X \leftarrow B]}$.

**Corollary 2.** Let $K = (S, S_0, R, L)$ be a Kripke structure, $e : \text{VAR} \to 2^S$ be an evaluation, and $\phi \in L_1$ be a deterministic $\mu$-calculus formula. Define $Q^\mu_i$ and $Q^\nu_i$ recursively as follows:

\[
\begin{align*}
Q^\mu_0 &= \emptyset \\
Q^\mu_i &= [\phi]_K^{e[X \leftarrow Q^\mu_{i-1}]} \\
Q^\nu_0 &= S \\
Q^\nu_i &= [\phi]_K^{e[X \leftarrow Q^\nu_{i-1}]}
\end{align*}
\]

then
∀i ∈ N, Q^µ_{i-1} ⊆ Q^µ_i and Q^ν_i ⊆ Q^ν_{i-1},

(ii) ∃n, m ∈ N such that Q^µ_{n-1} = Q^µ_n, Q^ ν_{m-1} = Q^ ν_m, and

(iii) Q^µ_n = \llbracket \mu X.\phi \rrbracket^e_K, Q^ ν_m = \llbracket \nu X.\phi \rrbracket^e_K.

Summarizing the algorithm, to evaluate \llbracket \mu X.\psi \rrbracket^e_K, it suffices to set Q^µ_1 to be the set of states satisfying the formula \psi where X = Q^µ_0 is initialized to be the empty set. We then proceed inductively, letting each subsequent Q^µ_i be the result of finding the states which satisfy \psi where X is replaced by Q^µ_{i-1}. After a finite number n of iterations (since the set of states S of a Kripke structure is finite, see Definition 1), a fixed point Q_{n-1} = Q_n = \llbracket \mu X.\psi \rrbracket^e_K will be reached. An analogous algorithm is applied when we wish to evaluate \llbracket \nu X.\phi \rrbracket^e_K, where Q^ ν_0 = X is initialized to be S.

2.1.4 Specification Examples

This section will introduce several examples of commonly used deterministic µ-calculus specifications. Each example is named, described, and explained in detail to provide some intuition to the reader [19].

(i) Reachability: \phi = \mu X.(p \lor \diamond X)

The reachability specification is used to ensure that the system eventually reaches a state which satisfies atomic proposition p. The resulting set \llbracket \phi \rrbracket^e_K is the winning set, that is, the set of all initial states for which the proposition \phi holds. In this case, the winning set consists of states which satisfy p or for which there exists a sequence of transitions in R which lead to a state satisfying p. In the context of the Kripke structure K = (S, S_0, R, \mathcal{L}), we seek only to show that S_0 is contained in the winning set \llbracket \phi \rrbracket^e_K.

In this example, we look for the least fixed point because we will start with the empty set and grow through all states that satisfy p or that can reach \llbracket p \rrbracket^e_K with one transition, then two transitions, and so on until the entire winning set is found. This process is guaranteed to terminate since the formula (p \lor \diamond X) is monotone (\phi is a deterministic µ-calculus formula), and since the Kripke structure contains finitely
many states. Let us apply the algorithm from Corollary 2 to elucidate the procedure:

\[ Q_0^\mu = \emptyset \]
\[ Q_1^\mu = [p \lor \Box X]_K^{[X \rightarrow Q_0^\mu]} \]
\[ = [p]_K \cup [\Box X]_K^{[X \rightarrow \emptyset]} \]
\[ = [p]_K \cup \text{Pre}_{[X \rightarrow \emptyset]}(X) \]
\[ = [p]_K \cup \{ s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \land s' \in \emptyset \} \]
\[ = [p]_K \]
\[ Q_2^\mu = [p \lor \Box X]_K^{[X \rightarrow Q_1^\mu]} \]
\[ = [p]_K \cup \{ s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \land s' \in [p]_K \} \]

This process continues until the least fixed point is reached. See Example 1 for a concrete illustration of the reachability specification.

(ii) **Safety:** \( \phi = \nu X. (p \land \Box X) \)

We use the the term “safety” as this specification guarantees that a given atomic proposition will hold on some state trajectory; the atomic proposition may be concerned with being in an obstacle-free space, or it may ensure that constraints on speed or acceleration are observed, for instance. The formula \( \phi \) will hold for all states which satisfy \( p \) and which have transitions to states which will themselves satisfy \( p \) and in turn have transitions to other states that will satisfy this same condition. For this reason, we start with the entire set of states, \( S \), and repeatedly find intersections to narrow down the states until the greatest fixed point is reached.

\[ Q_0^\nu = S \]
\[ Q_1^\nu = [p \land \Box X]_K^{[X \rightarrow Q_0^\nu]} \]
\[ = [p]_K \cap [\Box X]_K^{[X \rightarrow S]} \]
\[ = [p]_K \cap \{ s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \land s' \in S \} \]
\[ = [p]_K \]
\[ Q_2^\nu = [p \land \Box X]_K^{[X \rightarrow Q_1^\nu]} \]
\[ = [p]_K \cap \{ s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \land s' \in [p]_K \} \]

See Example 1 for a concrete illustration of the safety specification.
(iii) **Reaching a Region Safely**: \( \phi = \mu X. (\neg q \land (p \lor \Diamond X)) \)

One way of combining the above two specifications is to ensure that a safety condition is met while trying to reach an objective. This specification is commonly used for motion planning problems, wherein a planner searches for a trajectory which avoids obstacles (represented by states satisfying \( q \)), and reaches a given goal (represented by states satisfying \( p \)). Obstacle avoidance is guaranteed by the conjunction of the usual reachability subformula with \( \neg q \) so that at each iteration, we keep only those states which satisfy the reachability criterion and are not obstacles.

(iv) **Reaching a Safe Region**: \( \phi = \mu X. ((\nu Y. (p \land \Diamond Y)) \lor \Diamond X) \)

Another way to combine safety and reachability is the specification to reach a region whose states always satisfy a property \( p \). The subformula \( \psi = \nu Y. (p \land \Diamond Y) \) is identical to the safety specification listed above; \( \psi \) is satisfied by all initial states which give rise to trajectories that always satisfy \( p \). This safety specification is wrapped in the reachability specification \( \mu X. (\psi \lor \Diamond X) \), meaning states which satisfy \( \phi \) are either in the safe region already, or can reach the safe region using a finite number of transitions.

(v) **Ordering**: \( \phi = \mu X. (q \lor (p \land \Diamond X)) \)

In both LTL and CTL, there is a temporal operator \( U \) denoting “until”, so that \( pUq \) is satisfied provided \( p \) holds at least until \( q \) holds; that is, after a finite number of transitions, \( q \) must hold, and \( p \) may or may not continue to hold. In \( \mu \)-calculus, we can formulate this specification by building up a set of states where either \( q \) is already satisfied, or \( p \) is satisfied and there exists a transition to a state satisfying \( q \lor (p \land \Diamond X) \). Another way to interpret the formula is to distribute the disjunction to obtain \( \mu X. ((q \lor p) \land (q \lor \Diamond X)) \). In this form, we observe the reachability subformula \( q \lor \Diamond X \) (in the scope of a least fixed point operator, as usual), so we may conclude that the winning set contains states which eventually satisfy \( q \), and along the way must satisfy \( p \) (otherwise \( q \) is already satisfied, so \( p \) may or may not be satisfied).

(vi) **Liveness**: \( \phi = \nu Y. \mu X. ((p \land \Diamond Y) \lor \Diamond X) \)

Liveness, is a specification which guarantees that atomic proposition \( p \) is satisfied infinitely often. This example is the first with an alternation depth greater than one, where alternation depth refers to the level of mutually recursive least and greatest fixed point operators [46]. A more formal definition of alternation depth is presented in [5], though the definition seen here is sufficient for our purposes.

Consider the largest proper subformula of \( \phi \), \( \psi = \mu X. ((p \land \Diamond Y) \lor \Diamond X) \). This can be seen as a reachability specification where the goal is to eventually reach states
satisfying $\eta = p \land \Diamond Y$, which itself looks like a safety specification, remarking that $\eta$ is in the scope of a greatest fixed point operator. We may parse the liveness specification $\phi = \nu Y.\psi = \nu Y.\mu X.(\eta \lor \Diamond X)$ as follows: $\psi$ ensures that a region satisfying $\eta = p \land \Diamond Y$ is reached, and $\eta$ ensures that $p$ is satisfied and that there is a transition to a state satisfying $\psi$. Having mutually recursive greatest and least fixed point operators in this way allows for this more complex combination of reachability and safety, where $\phi$ is satisfied by all states which have paths that always eventually reach $[p]_K$. Note that liveness is also sometimes called the Büchi objective in the context of infinite parity games, a topic closely related to $\mu$-calculus [7, 21, 46].

2.2 SST*

In this section, we discuss a probabilistically complete sampling-based kinodynamic motion planning algorithm called SST along with its asymptotically optimal variant, SST*. For completeness, the planner is summarized in this section, although further details and proofs can be found in [28].

One very useful property of SST is that it does not rely on having the solution to an OBVP for the relevant system. Such a solution is called a steering function in the literature, and many motion planning algorithms are contingent upon its availability. For example, some planning algorithms that necessitate using a steering function include RRT* [20] and PRM [22]. The steering function provides, as the name suggests, optimal inputs to control or steer the system from one given state to another; for many planners, the necessity of such a function arises when sampled states (nodes) must be connected together with directed edges, representing that there is a known set of inputs to control the system between such states. The problem is, just as finding analytic solutions to nonlinear differential equations is very difficult or impossible, so too is finding the related solution to an associated OBVP.

Although scarce, there are a small number of sampling-based planning algorithms that do not require a steering function, most notably RRT-Extend [25] and Expansive Space Trees (EST). While EST has been shown to be asymptotically optimal (RRT-Extend is not), the rate of convergence to the (near-) optimal solution is logarithmic, making it

---

1Note that motion planning algorithms whose acronyms end with an asterisk (*) are usually asymptotically optimal variants of their associated algorithm.

2Some authors use different terminology for the steering/OBVP problem. For instance, in their paper on Probabilistic Roadmap (PRM), Kavraki et al. refer to a “local planner” which is used to connect neighbouring nodes in the case of holonomic robots.
impractical at reliably finding high-quality paths. On the other hand, the advantages offered by SST* include being provably asymptotically optimal as well as having good (linear) convergence to high-quality solutions. SST* is further improved by its use of a sparse data structure, where a pruning operation is used to accelerate nearest neighbours searches, thereby ameliorating computational efficiency.

We will now describe the implementation details of the SST* sampling-based planning algorithm. SST* employs MonteCarlo_Prop which, as the name suggests, forward propagates a selected node, $x_{\text{selected}}$, using random sampling. Specifically, a random control vector is sampled from the allowed control-space, $U$, and supplied as input to the system dynamics for a random duration, up to some specified maximum time, $T_{\text{prop}}$, resulting in trajectories with piecewise constant control inputs (see Algorithm 1).

Algorithm 1 MonteCarlo_Prop($x_{\text{selected}}, U, T_{\text{prop}}$)

1: $t \leftarrow \text{Sample}([0,T_{\text{prop}}])$
2: $\Upsilon \leftarrow \text{Sample}(U)$
3: return $x_{\text{new}} \leftarrow x_{\text{selected}} + \int_0^t f(x(\tau), \Upsilon) \, d\tau$

SST* also uses a best-first selection strategy to forward integrate from the least-cost node found within a specified radius of the sampled state, thereby improving convergence to high-quality solutions. To elaborate, we use Algorithm 2 called Best_First_Selection to sample a random state, $x_{\text{rand}}$, from the state space, $X$. Then, the set of all states within a $\delta_{\text{BN}}$ radius of $x_{\text{rand}}$ is stored in $X_{\text{near}}$. If $X_{\text{near}}$ is empty, we simply return the nearest node to $x_{\text{rand}}$ in the set of all nodes, $V$. Otherwise, the state in $X_{\text{near}}$ with the least cost is selected. In either case, the selected state, $x_{\text{selected}}$, is the state from which we propagate and grow the tree using MonteCarlo_Prop.

Algorithm 2 Best_First_Selection($X, V, \delta_{\text{BN}}$)

1: $x_{\text{rand}} \leftarrow \text{Sample}(X)$
2: $X_{\text{near}} \leftarrow \text{Near}(V, x_{\text{rand}}, \delta_{\text{BN}})$
3: if $X_{\text{near}} == \emptyset$ then
4: return Nearest($V, x_{\text{rand}}$)
5: else
6: return $\arg \min_{x \in X_{\text{near}}} \text{Cost}(x)$

Furthermore, SST* applies a pruning operation to maintain a sparse data structure. Pruning removes high-cost nodes to improve run-time by accelerating nearest neighbour searches. With this in mind, a graph of witness nodes is maintained, where each witness
Algorithm 3 \texttt{Is\_Locally\_Best}(x_{\text{new}}, S, \delta_s)

1: \texttt{s}_{\text{new}} \leftarrow \texttt{Nearest}(S, x_{\text{new}})
2: \textbf{if} \texttt{dist}(x_{\text{new}}, \texttt{s}\_{\text{new}}) > \delta_s \textbf{then}
3: \quad S \leftarrow S \cup \{x_{\text{new}}\}
4: \quad \texttt{s}_{\text{new}} \leftarrow x_{\text{new}}
5: \quad \texttt{s}_{\text{new}}.r\text{ep} \leftarrow \text{NULL}
6: \quad \texttt{x}_{\text{peer}} \leftarrow \texttt{s}_{\text{new}}.r\text{ep}
7: \textbf{if} \texttt{x}_{\text{peer}} \neq \text{NULL} \textbf{or} \texttt{Cost}(x_{\text{new}}) < \texttt{Cost}(x_{\text{peer}}) \textbf{then}
8: \quad \textbf{return} \text{True}
9: \textbf{return} \text{False}

keeps track of an optimal-cost representative node within a $\delta_s$-radius of the witness. Correspondingly, we must determine whether a new node is the “best” in its neighbourhood in order to decide whether or not to keep it. For this purpose, we use \texttt{Is\_Locally\_Best} presented in Algorithm 3. The algorithm finds the nearest witness state, $s_{\text{new}}$, to the newly propagated state $x_{\text{new}}$. If $s_{\text{new}}$ is not within distance $\delta_s$ of $x_{\text{new}}$, $x_{\text{new}}$ is deemed locally best by default and becomes a new witness node. In this way, $x_{\text{new}}$ is added to the set of witness nodes, $S$. If $x_{\text{new}}$ does have a neighbour within a $\delta_s$ radius, the chosen cost function, Cost, is used to check whether the new state is better than the locally best representative of the witness node.

Algorithm 4 \texttt{Prune}(x_{\text{new}}, V_{\text{active}}, V_{\text{inactive}}, E)

1: \texttt{s}_{\text{new}} \leftarrow \texttt{Nearest}(S, x_{\text{new}})
2: \texttt{x}_{\text{peer}} \leftarrow \texttt{s}_{\text{new}}.r\text{ep}
3: \textbf{if} \texttt{x}_{\text{peer}} \neq \text{NULL} \textbf{then}
4: \quad V_{\text{active}} \leftarrow V_{\text{active}} \setminus \{x_{\text{peer}}\}
5: \quad V_{\text{inactive}} \leftarrow V_{\text{inactive}} \cup \{x_{\text{peer}}\}
6: \quad \texttt{s}_{\text{new}}.r\text{ep} \leftarrow x_{\text{new}}
7: \textbf{while} \texttt{Is\_Leaf}(x_{\text{peer}}) \textbf{and} x_{\text{peer}} \in V_{\text{inactive}} \textbf{do}
8: \quad \texttt{x}_{\text{parent}} \leftarrow \texttt{x}_{\text{peer}}.p\text{arent}
9: \quad E \leftarrow E \setminus \{x_{\text{parent}} \rightarrow x_{\text{peer}}\}
10: \quad V_{\text{inactive}} \leftarrow V_{\text{inactive}} \setminus \{x_{\text{peer}}\}
11: \quad x_{\text{peer}} \leftarrow x_{\text{parent}}

Now that there is a means of determining whether a state is locally best, we are ready to define the algorithm which enforces sparsity, \texttt{Prune} (Algorithm 4). First, the nearest
witness state, $s_{\text{new}}$, to the newly propagated state, $x_{\text{new}}$, is found. We want to set $x_{\text{new}}$ to be the representative of $s_{\text{new}}$ since it has been deemed locally best if the SST* algorithm has reached this point (see Algorithm 5), but first we must check whether or not $s_{\text{new}}$ already has a representative. Since Prune is only executed if Is_Locally_Best returns True, then if $s_{\text{new}}$ has a representative, it must have higher cost than $x_{\text{new}}$, and it must now be moved from $V_{\text{active}}$ to $V_{\text{inactive}}$. The next step is to remove inactive leaf nodes recursively, so that if the previous representative of $s_{\text{new}}$ is an inactive leaf node, it is removed entirely from the set of all nodes, $V$, and the check is performed again with the parent of the removed leaf node.

Algorithm 5 SST*(X, U, $x_0$, $T_{\text{prop}}$, N, $\delta_{\text{BN}}, \delta_s, \xi$)

1: $V_{\text{active}} \leftarrow \{x_0\}$; $V_{\text{inactive}} \leftarrow \emptyset$
2: $E \leftarrow \emptyset$
3: $s_0 \leftarrow x_0$; $s_0.rep = x_0$; $S \leftarrow \{s_0\}$
4: $j \leftarrow 0$
5: while True do
6:   for N iterations do
7:      $x_{\text{selected}} \leftarrow \text{Best\_First\_Selection}(X, V_{\text{active}}, \delta_{\text{BN}})$
8:      $x_{\text{new}} \leftarrow \text{MonteCarlo\_Prop}(x_{\text{selected}}, U, T_{\text{prop}})$
9:      if CollisionFree($x_{\text{selected}} \rightarrow x_{\text{new}}$) then
10:         if Is_Locally_Best($x_{\text{new}}, S, \delta_s$) then
11:            $V_{\text{active}} \leftarrow V_{\text{active}} \cup \{x_{\text{new}}\}$
12:            $E \leftarrow E \cup \{x_{\text{selected}} \rightarrow x_{\text{new}}\}$
13:            Prune($x_{\text{new}}, V_{\text{active}}, V_{\text{inactive}}, E$)
14:         $\delta_s \leftarrow \xi \cdot \delta_s$; $\delta_{\text{BN}} \leftarrow \xi \cdot \delta_{\text{BN}}$
15:      $j \leftarrow j + 1$
16:   $N \leftarrow N(1 + \log(j)) \xi^{-j(d+l+1)}$
17: $V \leftarrow V_{\text{active}} \cup V_{\text{inactive}}$
18: return $(V, E)$

With the necessary helper functions defined, the pseudocode for SST* is outlined in Algorithm 5. Note that the nested for-loop constitutes the core of SST, and the addition of the update step for $\delta_s$ and $\delta_{\text{BN}}$ is all that is necessary to make the algorithm asymptotically optimal instead of being merely asymptotically near-optimal. This is because, if these two parameters do not tend to zero as the number of iterations goes to infinity, then there will always be some degree of sparsity, which means that not every possible solution is explored. In terms of notation, the overline indicates an edge (trajectory from one state
to another), and $d, l$ denote the number of dimensions of the state space and the control space, respectively.

It is important to note that proper tuning of the parameters used in SST* is crucial for effective path planning. The most significant parameters are $\delta_{BN}$ and $\delta_s$, which are the radii for selecting low-cost nearby nodes and for pruning in the vicinity of witness nodes, respectively. Choosing a large value of $\delta_s$ can decrease the time it takes to find an initial solution at the cost of solution quality, while choosing a large value of $\delta_{BN}$ can improve initial solution quality while increasing the time it takes to find an initial solution. Moreover, the SST* algorithm is incremental, meaning states are not sampled all at once, rather during each iteration a new state is sampled to incrementally grow a tree. A consequence of the incremental nature of SST* is that there is no clear way to precompute solution trajectories for online use unless (necessarily static) obstacles and the initial state are specified exactly.

![Figure 2.3: An example of a tree produced by SST for the flappy bird problem. The initial state is in the top-left, the point travels at a constant horizontal velocity, and the control-space is a set containing only two options: “do nothing”, or “accelerate up”, each for a random duration. The least-cost solution for this execution is highlighted in pink.](image)
2.3 FMT*

FMT* is an asymptotically optimal sampling-based motion planning algorithm that was specifically developed for use in high-dimensional systems [16]. In simulations, Janson et al. consistently found that FMT* converges to high-quality solutions faster than PRM* and RRT* for systems with dimension from 2D to 7D. The improvements were more noticeable in higher dimensions, making FMT* a very promising algorithm for complex motion planning problems such as for those involving quadrotors, which operate in a 12D state space. Note that, unlike SST*, FMT* is not an incremental algorithm. This means that the set of samples is fixed (as opposed to constantly growing), which allows for the possibility of precomputing useful information on the set of samples while offline.

Algorithm 6 \texttt{kinoFMT}(\texttt{x\textsubscript{init}}, \texttt{X\textsubscript{goal}}, \texttt{X}, n, J\textsubscript{th})

1: \texttt{V} $\leftarrow$ \{\texttt{x\textsubscript{init}}\} $\cup$ \texttt{Sample}(n, \texttt{X})
2: \texttt{E} $\leftarrow$ $\emptyset$
3: \texttt{W} $\leftarrow$ \texttt{V} \setminus \{\texttt{x\textsubscript{init}}\}; \texttt{H} $\leftarrow$ \{\texttt{x\textsubscript{init}}\}
4: \texttt{z} $\leftarrow$ \texttt{x\textsubscript{init}}
5: \texttt{while} \texttt{z} $\notin$ \texttt{X\textsubscript{goal}} \texttt{do}
6: \texttt{N\textsubscript{out}}\textsubscript{z} $\leftarrow$ \texttt{Near\_Forward}(\texttt{z}, \texttt{V} \setminus \{\texttt{z}\}, J\textsubscript{th})
7: \texttt{X\textsubscript{near}} $=$ \texttt{N\textsubscript{out}}\textsubscript{z} $\cap$ \texttt{W}
8: \texttt{for} \texttt{x} $\in$ \texttt{X\textsubscript{near}} \texttt{do}
9: \texttt{N\textsubscript{in}}\textsubscript{x} $\leftarrow$ \texttt{Near\_Backward}(\texttt{x}, \texttt{V} \setminus \{\texttt{x}\}, J\textsubscript{th})
10: \texttt{Y\textsubscript{near}} $\leftarrow$ \texttt{N\textsubscript{in}}\textsubscript{x} $\cap$ \texttt{H}
11: \texttt{y\textsubscript{min}} $\leftarrow$ \texttt{arg\_min}_{\texttt{y} \in \texttt{Y\textsubscript{near}}} \{\texttt{y}\texttt{.cost} + \texttt{Cost}(\texttt{y} \rightarrow \texttt{x})\}
12: \texttt{if} \texttt{Collision\_Free}(\texttt{y\textsubscript{min}} \rightarrow \texttt{x}) \texttt{then}
13: \texttt{E} $\leftarrow$ \texttt{E} $\cup$ \{\texttt{y\textsubscript{min}} $\rightarrow$ \texttt{x}\}
14: \texttt{x\textsubscript{cost}} $\leftarrow$ \texttt{y\textsubscript{min}\textsubscript{.cost}} + \texttt{Cost}(\texttt{y\textsubscript{min}} \rightarrow \texttt{x})
15: \texttt{H} $\leftarrow$ \texttt{H} $\cup$ \{\texttt{x}\}
16: \texttt{W} $\leftarrow$ \texttt{W} \setminus \{\texttt{x}\}
17: \texttt{H} $\leftarrow$ \texttt{H} \setminus \{\texttt{z}\}
18: \texttt{if} \texttt{H} $==$ $\emptyset$ \texttt{then}
19: \texttt{return} \texttt{Failure}
20: \texttt{z} $\leftarrow$ \texttt{arg\_min}_{\texttt{y} \in \texttt{H}} \{\texttt{y}\texttt{.cost}\}
21: \texttt{return} \texttt{Path}(\texttt{z}, \texttt{V}, \texttt{E})

We consider the kinodynamic variant of FMT* by Ross Allen and Marco Pavone called \texttt{kinoFMT}. The primary distinction lies in the fact that, due to considerations of
differential constraints, there is a fundamental difference between searching for nearest backward-reachable and forward-reachable states. In the case of backward reachability, \texttt{Near\_Backward}(x, V, J_{th}) finds states in V that can reach state x without exceeding a threshold cost, J_{th}. In contrast, \texttt{Near\_Forward}(x, V, J_{th}) finds states in V that can be reached by x itself without exceeding J_{th}. See Section 4.2.4 for further details. In fact, using a cost threshold is another deviation from the standard FMT* algorithm, which uses a distance threshold. Since kinodynamic planning is concerned with feasibility under kinematic and dynamic constraints, a simple distance metric is insufficient for determining which states are easily reachable from another state. Algorithm 6 presents the pseudocode of \texttt{kinoFMT} which is similar to the algorithm shown in [2] except for some slight changes for clarity and implementational simplicity.

![Figure 2.4: Example of a tree generated by FMT* using 2000 nodes. The initial state is shown at the top center as a green hexagon, the goal region lies in the bottom left and is shown in green, and obstacles are represented by red rectangles. The optimal path is highlighted in blue. This example does not use any dynamic model, so the cost function is simply the Euclidean distance.](image)
The main idea behind FMT* and kinoFMT is to use forward dynamic programming on a predetermined number of sampled states. The algorithms perform graph construction and graph search simultaneously, thus the final least-cost node lying in the goal region is already known when the algorithm terminates. In other words, by the nature of the expansion of the tree structure, the least-cost nodes on the frontier of expansion are always tracked, so that when the goal is reached, it is not necessary to search the entire tree for the optimal terminal node. Moreover, since tree structures contain no cycles, there exists a unique path from the starting node to the terminal node. Figure 2.4 demonstrates an example of a solution path found via FMT*.

In more detail, kinoFMT takes as input an initial state, \( x_{\text{init}} \), and goal region, \( X_{\text{goal}} \), the state space, \( X \), the number of nodes to sample, and a cost threshold, \( J_{th} \). The Sample function uniformly samples the entire state space\(^3\) (usually with some samples selected directly from the goal region) and stores these sampled states in \( V \) along with \( x_{\text{init}} \) (line 1). The set \( W \) contains unexplored nodes, and the set \( H \) contains the nodes forming the frontier of expansion of the tree, which initially includes only \( x_{\text{init}} \) (line 3). Every iteration, the least-cost node \( z \in H \) becomes the pivot about which expansion occurs (lines 6, 20). With the pivot known, we define \( X_{\text{near}} \) to be the set of unexplored nearest nodes in the forward-reachable set of \( z \) (lines 5–7). Then, for each element \( x \in X_{\text{near}} \), define \( Y_{\text{near}} \) to be the set of nodes in the frontier that are also in the backward-reachable set of \( x \) (lines 8–10). The least-cost state \( y_{\text{min}} \) is then found from the set of all \( y \in Y_{\text{near}} \), where the cost is determined to be the sum of the cost of reaching \( y \) (along its unique path from \( x_{\text{init}} \)) and the cost to travel from \( y \) to \( x \), recalling that \( y \) is in the backward-reachable set of \( x \) (line 11). This is the dynamic programming step, where we try to minimize the cost-to-come and the cost-to-go. Once the cost minimizer is found, a collision check is performed (line 12), and if a collision is detected along the transition from \( y_{\text{min}} \) to \( x \), then the algorithm simply discards this iteration and proceeds to the next. Otherwise, the edge (transition) is added to the tree, the newly connected node’s cost is updated and the node is added to the frontier set, \( H \), and it is simultaneously removed from the set of unexplored nodes, \( W \) (lines 13–16). Once the for-loop has iterated through the entire set of forward-reachable nodes, \( X_{\text{near}} \), the pivot, \( z \), is removed from the frontier and the procedure repeats until either the frontier is empty, at which point “Failure” is returned (lines 18–19), or until \( z \) lies within the goal region, at which point the algorithm terminates. The Path function simply returns the optimal path, that is, the unique path from \( x_{\text{init}} \) to the final least-cost goal state, \( z \).

\(^{3}\)The FMT* algorithm specifies that samples are taken only from the free space (i.e., not including obstacles,) however, sampling from the entire state space allows for a much more general motion planning framework, as discussed in Section 4.2.
Figure 2.5. In part a) of the diagram, of the two nodes in the frontier, $H$, the one with least cost is selected as $z$. For this simple example, distance is used as the cost with a search radius of $J_{th}$, and three unvisited nodes are found to lie within this distance from $z$. These are the nodes belonging to $X_{\text{near}}$. The algorithm will iterate through all three nodes in $X_{\text{near}}$, but we focus on one for this example. In b), we add all nodes whose search radius includes node $x$ to $N^i_{x}$. This set is not necessarily equivalent to the set of all nodes within the search radius of $x$; for non-symmetric cost functions, this distinction could have a significant impact (e.g., the kinoFMT algorithm). In c), of the nodes in $N^i_{x}$, only those that are also in $H$ are included in the set $Y_{\text{near}}$. For each node $y \in Y_{\text{near}}$, the cost of each, $y.\text{cost} + \text{Cost}(y \rightarrow x)$, is computed and the least-cost node, $y_{\text{min}}$, is determined. Provided there is no collision when connecting $y_{\text{min}}$ to $x$, the edge is added to the tree and $x$ is added to the set $H$. Once this entire process is repeated for every $x \in X_{\text{near}}$, the next iteration begins with a new least-cost frontier node $z$, or returns “Failure” if $H$ is empty.
Chapter 3

Sampling-Based Motion Planning with $\mu$-Calculus Specifications without Steering

This chapter presents the work published by the author of this thesis in [24].

3.1 Introduction

Motion planning in complex environments has seen a shift towards using sampling-based planning algorithms. By using a predetermined sampling scheme, a random snapshot of the workspace can be taken and incrementally improved upon without any prior knowledge of the environment. Furthermore, combining sampling-based motion planning with temporal logic specifications grants users a much more sophisticated toolbox of high-level behaviors that can be specified.

Temporal logics present a means of formally expressing high-level specifications for use in various problems in mathematics, robotics, and computer science. In particular, temporal logic specifications are well-suited for motion planning problems, allowing a user-defined specification to describe the desired behavior of an autonomous vehicle or robot (e.g., [30, 47, 6]). The modal $\mu$-calculus is a highly expressive temporal logic which permits more diverse and complex specifications than the most widely used temporal logics, including LTL, CTL, and extensions thereof. On the other hand, it is typically much easier to understand LTL specifications at a glance, whereas $\mu$-calculus formulas can be
much more difficult to intuit. For example, the LTL formula for reachability is $\Diamond p$, which is equivalent to the more complicated $\mu$-calculus expression $\mu X . (p \lor \Diamond X)$. The added complexity of $\mu$-calculus formulas is not without benefit, however, as a major advantage lies in its predisposition for simple model checking. While LTL specifications are a useful tool for high-level planning, they must usually be translated into automata for the purposes of model checking. The equivalent $\mu$-calculus specification, however, can be checked directly without any intermediate steps via the Tarski-Knaster fixed point theorem [8, 40].

In this work, a fragment of the full $\mu$-calculus called deterministic $\mu$-calculus [19] is used, allowing for efficient model checking while maintaining the ability to specify complex tasks. Some such specifications include reaching a goal while avoiding obstacles, and a property known as liveness, which involves satisfying one or many propositions infinitely often. Furthermore, using the model checking algorithm discussed in Section 3.3, it is possible to formally synthesize a control policy that provably satisfies a given deterministic $\mu$-calculus specification.

3.2 Problem Formulation

Consider a time-invariant continuous dynamical control system given by the differential equation

$$\dot{x}(t) = f(x(t), u(t)), \quad x(0) = x_0 \tag{3.1}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the control input, and $f : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$ is a locally Lipschitz function. Let $\Pi$ be a set of atomic propositions, which are the simplest form of declarative statements that are either true or false. Lastly, let $L : \mathbb{R}^n \rightarrow 2^\Pi$ be a labeling function which assigns to a state all of the atomic propositions that it satisfies.

The goal is to design a controller such that the possibly infinite state trajectory $x(t)$ satisfies a given temporal logic specification, $\Phi$. We choose to work with deterministic $\mu$-calculus for reasons discussed in Section 2.1.2. It is important to note that model checking of the temporal logic specification must be performed on a finite model of the dynamical system. To this end, a sampling-based motion planner is used to generate a discretization of the set of possible trajectories from the initial condition in the form of a Kripke structure.

Definition 7. A Kripke structure $K = (S; \{s_0\}, R, L)$ models the dynamical control system (3.1) with initial state $x(0)$ if (i) $S \subseteq \mathbb{R}^n$; (ii) $s_0 = x(0) \in S$; (iii) $(s, s') \in R$ only if there exist $t_0, t_1 \in \mathbb{R}$, $t_1 > t_0$, and a control signal $u : [t_0, t_1] \rightarrow \mathbb{R}^m$ such that $s = x(t_0) \in \mathbb{R}^n$ and $s' = x(t_1) = x(t_0) + \int_{t_0}^{t_1} f(x(t), u(t))dt$; (iv) $L(s) = L(s)$ for all $s \in S$. 

29
This definition provides a concrete way of evaluating whether or not a Kripke structure sufficiently models any given dynamical system. We use this definition in the problem statement that follows.

3.2.1 Problem Statement

A precise formulation of the problem to be solved can now be made. We seek to use a dynamical control system together with the notions of Kripke structures and deterministic $\mu$-calculus specifications, the aim being to determine whether a given specification is satisfied by a Kripke structure that models the dynamical control system.

**Definition 8.** A continuous-time dynamical control system of the form (3.1) is said to satisfy a deterministic $\mu$-calculus specification $\Phi$ at some initial state $x_0$ if and only if there exists a Kripke structure $K^* = (S^*, \{x_0\}, R^*, L^*)$ modeling the system, and such that $x_0 \in [\Phi]_{K^*}$.

In contrast to [19], we allow the dynamical control system to be continuous in time. The problem statement is as follows:

**Problem Statement.** Given a continuous-time dynamical control system (3.1) with initial state $x_0$ and a deterministic $\mu$-calculus formula $\Phi$, return a control policy $u$ which gives rise to a trajectory satisfying $\Phi$ obtained from a Kripke structure that models the system, or return failure if such a trajectory is not found.

3.3 Kripke Structures and Model Checking

The main goal of this work is to design an algorithm that finds near-optimal trajectories satisfying a given temporal logic specification without relying on an OBVP (steering function). The motivation for such an algorithm is that in a differentially constrained system, finding an optimal trajectory between two states is difficult in general. Some research has been done to first linearize system dynamics to find a solution to the BVP [44], while others have found some success in numerical solutions to BVPs with nonlinear dynamics using sequential quadratic programming [48]. However, neither approach fully addresses the crux of the problem: some planning problems, such as systems simulated on a physics engine, allow only forward propagation. We opt to use the asymptotically optimal variant of SST called SST* by Li et al. [28] which does not require a steering function to plan high-quality trajectories.
3.3.1 Model Checking with $\mu$-Calculus Specifications

It is necessary to determine whether or not the $\mu$-calculus specification $\Phi$ is satisfied during the incremental tree expansion (refining the discretized state space) with SST*. To perform such a verification, we will use a local model checking algorithm tailored specifically to run efficiently for deterministic $\mu$-calculus. The procedure requires as input a Kripke structure, $K$, the $\mu$-calculus specification, $\Phi$, an initial state, $s$, and the subformula to be checked in the current execution, $\phi$. We further require a function $\text{succ}(s)$, which returns all states in $K$ which may be reached from $s$ via one relation (edge of the tree), and $\text{BoundFormula}(X)$, which maps the input variable $X$ to the subformula of $\Phi$ of the form $\sigma X.\psi$, that is, the smallest subformula that binds the variable $X$ to a least or greatest fixed-point operator. Note that the first two arguments of $\text{ModelCheck}$ are omitted in recursive calls for brevity as they remain unchanged. The local model checking algorithm presented here is based on Algorithm 3 presented in [19].

Algorithm 7 is a recursive function which returns a boolean value without having to create any intermediary graphs unlike the proposed incremental model checking algorithms from [8] and [19], and also without the need for generating an automaton on which to perform model checking, unlike when using LTL. Since model checking will be performed on a very small abstracted Kripke structure, using the local, non-incremental model checking algorithm is sufficient. The algorithm presented here is based on a similar algorithm from [38], wherein correctness is proved.

3.3.2 Abstracted Kripke Structure and Planning

The primary contribution we make is to simplify model checking over several data structures by creating an abstracted Kripke structure. We generate this new abstracted structure $\tilde{K}_{\Pi^+(\Phi)} = (\tilde{S}, \{\tilde{s}_0\}, \tilde{R}, \tilde{L})$ from $p = |\Pi^+(\Phi)|$ Kripke structures of the form $K_i = (S_i, \{s_{i0}\}, R_i, L), i = \{1, \ldots, p\}$, generated via SST*, where $\Pi^+(\Phi) \subseteq \Pi$ is the set of atomic propositions which appear positively in specification $\Phi$. The reason for restricting the abstracted Kripke structure to use only the positively-appearing atomic propositions is that these are the specifications we explicitly want to fulfill (e.g., reach a certain goal), as opposed to something we do not want our system to do (e.g., run into obstacles). In this way, a directed edge is added to the abstracted Kripke structure only if a trajectory in one of the $K_i$ is found to initially satisfy one atomic proposition, and reaches a state satisfying another atomic proposition, all the while ensuring any negatively-appearing atomic propositions are respected so that the associated regions of state space are avoided.
Algorithm 7 ModelCheck($K, \Phi, s, \phi$)

1: switch $\phi$ do
2:   case $p$ where $p \in \Pi$ do
3:     return $p \in \mathcal{L}(s)$
4:   case $\neg p$ where $p \in \Pi$ do
5:     return $p \notin \mathcal{L}(s)$
6:   case $p \land \varphi$ do
7:     return $p \land \text{ModelCheck}(s, \varphi)$
8:   case $\neg p \land \varphi$ do
9:     return $\neg p \land \text{ModelCheck}(s, \varphi)$
10:  case $\psi \lor \varphi$ do
11:    return $\text{ModelCheck}(s, \psi) \lor \text{ModelCheck}(s, \varphi)$
12:  case $\Diamond \varphi$ do
13:    for $s' \in \text{succ}(s)$ do
14:      if $\text{ModelCheck}(s', \varphi)$ then
15:        return True
16:    return False
17:  case $\sigma X. \varphi$ where $\sigma \in \{\mu, \nu\}$ do
18:    set $\leftarrow$ set $\cup \{(s, \varphi)\}$
19:    value $\leftarrow$ ModelCheck($s, \varphi$)
20:    set $\leftarrow$ set $\setminus \{(s, \varphi)\}$
21:    return value
22:  case $X$ where $X \in \text{VAR}$ do
23:    if $(s, \text{BoundFormula}(X)) \in \text{set}$ then
24:      switch BoundFormula($X$) do
25:        case $\mu X. \varphi$ do
26:          return False
27:        case $\nu X. \varphi$ do
28:          return True
29:      else
30:        return $\text{ModelCheck}(s, \text{BoundFormula}(X))$
The `KinoSpecPlan` algorithm takes as input the dynamics $f$ from (3.1), the initial condition $x_0$, the deterministic $\mu$-calculus specification $\Phi$, an array called `regions` containing $p$ subsets of the state space (one for each positively-appearing atomic proposition in $\Phi$), and the labeling function $L$. The function `ChooseInit` samples a state from the input region to use as the initial state for Kripke structure $K_i$, or $x_0$ if it exists in the given region. $\text{SST}^*$ takes a Kripke structure and incrementally grows the structure using forward propagation, thereby updating its set of states and relations. $\text{AbstractUpdate}$ verifies the existence of any paths spanning from one region to another in the list $K$ of all Kripke structures. If so, the corresponding relation is added to the list of relations maintained in the abstracted Kripke structure $\tilde{K}$. Furthermore, the first time it is run, $\text{AbstractUpdate}$ adds the necessary states from the initial states of each of the $K_i$ to its own set of states. Using the abstracted Kripke structure and the list of all Kripke structures, $\text{ConstructPath}$ creates a single time-parameterized state trajectory along the least-cost path satisfying $\Phi$, where each individual candidate path from the Kripke structures is combined head-to-tail as necessary.

In essence, the algorithm works as follows. First, the abstracted Kripke structure $\tilde{K}$ is initialized with only the initial condition and an empty set of relations, and $K$, the list of Kripke structures, is initialized to be empty (lines 1–3). In lines 4–7, the for-loop initializes each Kripke structure $K_i$, $i \in \{1, \ldots, p\}$, to be the trivial graph consisting only of the initial condition $s_{i0}$, which is chosen from among any of the states in the proposition region $[\pi_i]$ associated with atomic proposition $\pi_i \in \Pi^+$; that is, the initial node $s_{i0}$ of $K_i$ satisfies $\pi_i \in L(s_{i0})$. 

---

**Algorithm 8 KinoSpecPlan($f,x_0,\Phi,\text{regions},\mathcal{L}$)**

1: $\tilde{K} \leftarrow (\{x_0\}, \{x_0\}, \emptyset, \mathcal{L})$
2: $K \leftarrow \emptyset$
3: $p \leftarrow \text{length}(\text{regions})$
4: for $i \in \{1, \ldots, p\}$ do
5:   $s_{i0} \leftarrow \text{ChooseInit}(\text{regions}[i], x_0)$
6:   $K_i \leftarrow (\{s_{i0}\}, \{s_{i0}\}, \emptyset, \mathcal{L})$
7:   $K \leftarrow K \cup \{K_i\}$
8: while $\neg \text{ModelCheck}(\tilde{K}, \Phi, x_0, \Phi)$ do
9:   for $i \in \{1, \ldots, p\}$ do
10:      $K_i \leftarrow \text{SST}^*(K_i)$
11: $\tilde{K} \leftarrow \text{AbstractUpdate}(\tilde{K}, K)$
12: return $\text{ConstructPath}(\tilde{K}, K)$
Next, each Kripke structure is expanded in parallel using the SST* motion planning algorithm (lines 8–10). The abstracted Kripke structure $\tilde{K}_{\Pi^+}$ is then updated in line 11 so that each of its $p$ nodes $\tilde{s}_i \in \tilde{S}$ corresponds to a positively-appearing atomic proposition, so $\pi_i \in \mathcal{L}(\tilde{s}_i)$, $i \in \{1, \ldots, p\}$ (this particular procedure only occurs the first time AbstractUpdate is executed). The crucial aspect of this step is that the relations in $\tilde{R}$ are also updated, and relation $(\tilde{s}_a, \tilde{s}_b)$ is added if and only if there exists a path in the Kripke structure $K_a$ from $s_{a0} \in S_a$ to a node $s_b \in S_a$ satisfying $\pi_b \in \mathcal{L}(s_b)$.

The deterministic $\mu$-calculus model checking algorithm ModelCheck verifies whether the abstracted Kripke structure $\tilde{K}$ satisfies the given specification, $\Phi$. If the specification is not satisfied, the while-loop repeats lines 8–11. Once the specification is found to be satisfied by the abstracted Kripke structure $\tilde{K}$, a time-parameterized trajectory is created from the combination of the best paths found among the relevant Kripke structures $K_i$ (line 12).

Note that in order to use the path returned by Algorithm 8, we track it using an LQR controller obtained by linearizing the dynamical system (3.1) at the current state. The state error can then be obtained and the appropriate feedback control can be applied until the next time step, at which point the LQR algorithm must be run again.

### 3.3.3 LQR Tracking

Once the abstracted Kripke structure $\tilde{K}_{\Pi^+}(\Phi)$ is generated and the model checking algorithm confirms the satisfaction of the $\mu$-calculus specification $\Phi$, it is necessary to connect the candidate trajectories from the various $K_i$ structures. Accordingly, for all paths in $\tilde{K}_{\Pi^+}(\Phi)$ satisfying $\Phi$, the corresponding candidate trajectories in the $K_i$ structures are bridged together within the appropriate proposition region, i.e., the region in state space where every state $x$ satisfies a particular proposition $p \in \Pi^+(\Phi)$. To do so, we apply an LQR controller as in [41].

First, the dynamical system is linearized to be of the form $\dot{x} = Ax + Bu$. We then define the quadratic cost function over the time interval $[t_0, t_1]$ to be

$$J = \int_{t_0}^{t_1} (\bar{x}^T Q \bar{x} + \bar{u}^T R \bar{u}) \, dt \quad (3.2)$$

where $Q$ is a symmetric positive semi-definite state cost matrix, $R$ is a symmetric positive definite control cost matrix, the tracking error is $\bar{x} = x - x_c$ where $x_c$ is the time-parameterized total candidate trajectory found by patching together the individual candidate trajectories in sequence, and $\bar{u} = u - u_c$ where $u_c$ is the corresponding control signal.
for $x_c$. To proceed, the steady-state solution $P$ to the continuous algebraic Ricatti equation (CARE)

$$A^T P + PA - PBR^{-1}B^T P + Q = 0$$  \hspace{1cm} (3.3)

must be found. The feedback control is then given by $u = u_c - K\bar{x}$, where $K = R^{-1} B^T P$.

### 3.4 Example

To demonstrate the effectiveness of our method, we provide the following pertinent example. We use continuous double integrator dynamics on two spatial dimensions, resulting in a 4D state space. State and control vectors take the form

$$\bar{x} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}, \quad \bar{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix}$$  \hspace{1cm} (3.4)

and the dynamical system is given by

$$\dot{x} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \bar{u},$$  \hspace{1cm} (3.5)

with initial condition $\bar{x}_0 = [100, 400, 0, 0]$. We choose the cost function for SST* to be the duration of the trajectory, $T$, along with a control cost term, with cost matrix $R_{sst}$:

$$J_{SST} = \int_0^T (1 + u^T R_{sst} u) \, dt.$$  \hspace{1cm} (3.6)

The specification we wish to satisfy is to visit three distinct regions of the state space, $R_a, R_b,$ and $R_c$ infinitely often while avoiding the obstacle regions collectively called $R_o$. Define atomic propositions $p_i, i \in \{a, b, c, o\}$, such that $p_i \in \mathcal{L}(s)$ if and only if $s \in R_i$. We write the deterministic $\mu$-calculus formula $\Phi$ as follows

$$\mu M.[(\neg o \land \Diamond M) \lor \\
\nu W.((a \land \mu X.\neg o \land (((b \lor c) \land W) \lor \Diamond X)) \lor \\
(b \land \mu Y.\neg o \land (((a \lor c) \land W) \lor \Diamond Y)) \lor \\
(c \land \mu Z.\neg o \land (((a \lor b) \land W) \lor \Diamond Z))].$$

35
Upon running SST* in three parallel instances starting in the center of each proposition’s associated region in state space, we obtained Figure 3.2. Note that there are six candidate trajectories, each representing the best path from one region to another in terms of the cost, $J_{SST}$. In general, for $p$ positively-appearing atomic propositions in the $\mu$-calculus specification, $p$ trees are incrementally updated in parallel, and we search for $p - 1$ candidate trajectories for each tree.

The model checking algorithm ensures that at least one cycle of length three may be formed in the abstracted Kripke structure, which itself is constructed with three nodes (one for each proposition region), and whose directed edges represent the candidate trajectories that begin in one proposition region and end in another one. The procedure was run until all six candidate trajectories successfully reached their appropriate goal regions, allowing for the abstracted Kripke structure to be fully connected\(^1\). The resulting structure contains two infinite paths satisfying proposition $\Phi$ above, noting that the initial condition is situated in the blue region: (i) head to the green region first, then the burgundy region, and returning to the blue region to repeat, or (ii) head to the burgundy region first, then the green region, and repeating upon returning to the blue region.

In order to determine which of the cycles to take, a simulation is run using the LQR controller discussed in Section 3.3.3, tracking each of the proposed paths and bridging the gap between the end of one trajectory and the beginning of another. The state-cost matrix $Q$ was set to $\text{diag}(25, 25, 25, 25)$ and the control-cost matrix $U$ was chosen to be $\text{diag}(1, 1)$. Using total cost to determine the better of the two proposed solutions, it is found that

\[^1\]This was done in order to compare the cost associated with each direction of the possible 3-cycle, although Algorithm 8 as it is written would return as soon as any satisfactory path is found.
Figure 3.2: SST* is performed three times, producing a Kripke structure for each of the regions shown here in blue (left), burgundy (bottom center), and green (top right). Obstacles are represented as pink rectangles. The color of each line matches the color of the region of the Kripke structure to which it belongs, and the bolded curves are the lowest-cost trajectories that reach another region.

option (i) results in a faster circuit between the three regions. See Figure 3.3.
Figure 3.3: The six candidate trajectories are shown, where curves of the same color are selected from the same Kripke structure. The solution trajectory is shown in yellow, starting in the center of the blue region and tracking the infinite path with least cost that satisfies specification $\Phi$. 

38
Chapter 4

Quadrotor Motion Planning

4.1 Quadrotor Model

Quadrotors, as discussed in Chapter 1, are growing in popularity for their many uses. Autonomous navigation for quadrotors is still in the early stages of development, although there are already some basic autonomous behaviours in use commercially; for example, many drones now support following a moving person to capture video footage. What makes quadrotor motion planning a truly interesting challenge is the fact that they occupy a 12D state space, and they are non-holonomic vehicles that are governed by nonlinear dynamics. The combination of high-dimensionality and nonlinearity render many current methods ineffective in the context of motion planning, for example the interval method [17, 29], which discretizes the state and control space with intervals, and suffers from the curse of dimensionality [14].

Before proceeding to the mathematics involved, it may be of interest to the reader to discuss our choice to use the term “quadrotor”. But first, we begin by seeing that the word “helicopter” can be broken down into “helico”, which is itself a combining form of “helix” (screw), and “pter” meaning “wing”. Some people have referred to four-rotor helicopter UAVs as “quadcopters”, but this terminology is ignorant of the Ancient Greek etymology of such words, as “heli/copter” is not the correct partitioning of the English word. So, the term “quadrotor” avoids the issue altogether, and its meaning is self-evident.

Now, in order to overcome the hurdles inherent in motion planning for such a complex system as a quadrotor, Ross Allen and Marco Pavone put forth a “full-stack approach” for real-time kinodynamic planning of such aerial vehicles [2]. First and foremost, a sampling-based planning approach was deemed necessary to deal with unknown environments online,
and while SST* can handle the high-dimensionality and nonlinearity of the quadrotor system without needing a steering function, the incremental nature cripples its ability to plan effectively in an online setting. On the other hand, FMT*, with its pre-sampled states and its flexibility in allowing to perform much of the necessary pre-computation offline, is much more suited to online planning. For this reason, the authors centre their planning framework around a variant of FMT* called kinoFMT, introduced in Chapter 2. Once an approximate trajectory is found using this planning algorithm, trajectory smoothing is applied, and due the differentially flat nature of the quadrotor dynamics, it is then feasible to track the smooth trajectory with the proposed controller.

This chapter begins by analyzing the dynamical system that models quadrotor dynamics. Then, much of the work from [2] is described in various levels of detail, and conclusions are drawn from the efficacy of this approach. Finally, the main idea of using high-level temporal logic specifications using an abstracted Kripke structure, presented in Chapter 3, is applied to quadrotor kinodynamic planning with steering under the aforementioned real-time planning framework.

4.1.1 Background

Before delving into the laws of motion for quadrotor systems, we begin by defining the underlying coordinate systems used in our analysis. In order to describe the position and velocity of the quadrotor, we need a fixed inertial reference frame (sometimes called the world frame) where Newton’s laws hold. We will use the usual $x, y, z$ coordinates with basis vectors $\{e_1, e_2, e_3\}$, where $e_1 = [1, 0, 0]^T$, $e_2 = [0, 1, 0]^T$, and $e_3 = [0, 0, 1]^T$. It is worth noting that much of the aviation/aeronautics literature uses the North-East-Down (NED) configuration, so equations of motion found in that frame will differ slightly from those that we will present here. Along with the inertial frame, we define a body-fixed frame with basis $\{b_1, b_2, b_3\}$ whose origin coincides with the centre of mass of the quadrotor. The vectors $b_1$ and $b_2$ lie in the plane formed by the four rotors, and $b_3$ points in the direction opposite the applied thrust. See Figure 4.1.

Controlling a quadrotor involves adjusting the thrust applied to each of the four rotors. Note that opposite rotors rotate in the same direction, and adjacent rotors rotate in opposite directions; consequently, if all four rotors apply the same amount of thrust, the quadrotor will fly directly upward, and the angular momentum contributed by each rotor cancels so that there is zero rotational motion. Quadrotor motion is described in the space of all rigid body transformations, namely the special Euclidean group $\text{SE}(3)$. This space has six degrees of freedom: translation in three dimensions, and rotation about each of the
three body-fixed axes. It bears mentioning that, since there are only four control inputs compared with six degrees of freedom, the quadrotor system is underactuated.

Rotational motion is often described by the Euler angles measuring yaw (about $b_3$), pitch (about $b_2$), and roll (about $b_1$). The use of Euler angles as state variables is not ideal, though, as singularities and jump-discontinuities arise as a result of restricting the domain of such angles. Recent work by Taeyoung Lee et al. instead takes a geometric control approach with a globally defined model to avoid many of the issues inherent to Euler angles [27], and we will make use of their work in modeling quadrotor dynamics. The important change they make is to replace the three Euler angle state variables with a single $3 \times 3$ matrix in the special orthogonal group, $SO(3)$, defined as follows:

$$SO(3) = \{ R \in \mathbb{R}^{3 \times 3} \mid R^T R = I, \, \det(R) = 1 \}.$$  \hspace{1cm} (4.1)

The elements $R \in SO(3)$ are called rotation matrices, and they are orthogonal matrices that describe the attitude of the quadrotor. Note that the restriction on the determinant of the matrices excludes orthogonal matrices with determinant equal to $-1$, which have
the effect of transforming via reflection as opposed to rotation. As we are concerned only with physically possible transformations, reflections are removed from the set of allowed transformation matrices, and the qualifier “special” is prepended to the orthogonal group.

The rotation matrices in SO(3) are linear transformations that act on vectors via multiplication to produce a rotated vector. Given a vector \( \vec{v} \in \mathbb{R}^3 \) in the inertial frame (i.e., \( \vec{v} = ae_1 + be_2 + ce_3 \) for some \( a, b, c \in \mathbb{R} \)) and rotation matrix \( R \in \text{SO}(3) \), \( w = R\vec{v} \) is the result of rotating \( \vec{v} \) by \( R \), where \( w \) is expressed in the inertial frame. A useful interpretation of such rotation matrices is that the matrix \( R \) represents the current orientation of a rigid body. That is to say, the body-fixed axes are obtained, as above, by applying the rotation \( R \) to each of the basis vectors \( e_1, e_2, \) and \( e_3 \). In this way, we need not consider the rotation an active change in the quadrotor’s orientation, but rather as the current orientation obtained by rotating the axes of the inertial frame.

4.1.2 Dynamics

We are now sufficiently equipped to outline the equations of motion of a quadrotor UAV. The first equation states the relationship between the position of the centre of mass, \( x = [x_1, x_2, x_3]^\top \in \mathbb{R}^3 \), and the velocity of the centre of mass, \( v = [v_1, v_2, v_3]^\top \in \mathbb{R}^3 \), together constituting the first six state variables. The equation is simply

\[
\dot{x} = v. \tag{4.2}
\]

To develop the next, more interesting, equation, we begin by noticing that \( b_3 = Re_3 \). If we express the magnitude of the thrust generated by the \( i^{th} \) propeller as \( f_i \), \( i \in \{1, 2, 3, 4\} \), then the total thrust in the body-fixed frame is given by \( fb_3 \), where \( f = \sum_{i=1}^{4} f_i \). Therefore, in the inertial frame, the total thrust is written as \( fRe_3 \). The only other force acting on the quadrotor (ignoring disturbances) is the force of gravity, which pulls along the \(-e_3\) axis, and we write the force as \(-mge_3\), where \( m \) is the mass of the quadrotor, and \( g \) is the magnitude of the force of gravity (\( g \approx 9.8 \) on the surface of the Earth). Using Newton’s Second Law, we may now put these forces together to write our second equation of motion,\[

m\dot{v} = fRe_3 - mge_3. \tag{4.3}
\]

The final two equations of motion for the quadrotor system describe the rotational dynamics. Define \( \Omega \in \mathbb{R}^3 \) to be the angular velocity of the quadrotor in the body-fixed frame. \( R \) and \( \Omega \) constitute the remaining state variables, and they appear together in
an interesting way in the third equation of motion, which describes the rate at which the rotation matrix changes with time.

Before proceeding, let us first introduce \( \mathfrak{so}(3) \), the Lie algebra associated with the Lie group \( \text{SO}(3) \). A Lie algebra contains the elements of the tangent space of the Lie group at the identity, and in this case, we have that \( \mathfrak{so}(3) \) is simply the set of skew-symmetric matrices,

\[
\mathfrak{so}(3) = \{ X \in \mathbb{R}^{3 \times 3} \mid X^\top + X = 0 \}.
\] (4.4)

The skew-symmetric matrices represent infinitesimal rotations. Consider rotating a vector \( x \) about some unit vector, \( v \), by angle \( \theta \) in the counterclockwise direction. In the limit as \( \theta \) approaches 0, the rotation occurs normal to the plane containing both vectors, in the direction \( v \times x \). This motivates the definition of the hat map, \( \hat{\cdot} : \mathbb{R}^3 \to \mathfrak{so}(3) \) which satisfies \( \hat{a}b = a \times b \) for all \( a, b \in \mathbb{R}^3 \).

\[
\hat{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}
\] (4.5)

Tying these concepts together, the infinitesimal generator of rotation in the given scenario is the matrix \( \hat{\Omega} \), since \( \hat{\Omega}x \) yields the direction of the rotation of \( x \) about \( v \) by an infinitesimal angle.

Recall that an element \( \hat{v} \in \mathfrak{so}(3) \) is in the tangent space of \( \text{SO}(3) \) at the identity. In order to produce an infinitesimal rotation at an arbitrary element \( R \in \text{SO}(3) \), we simply left-multiply the appropriate infinitesimal rotation (element of \( \mathfrak{so}(3) \)) by \( R \). In this case, \( \hat{\Omega} \) is the appropriate element of \( \mathfrak{so}(3) \) since \( \Omega \) describes the rotational motion, and thus the axis of rotation, of the quadrotor. Therefore, we conclude that the third equation of motion is:

\[
\dot{R} = R\hat{\Omega}.
\] (4.6)

Finally, the last equation of motion governs how \( \Omega \) changes over time. Given the moment of inertia matrix, \( J \in \mathbb{R}^{3 \times 3} \), of our quadrotor, we can write an equation for the total torque in the body-fixed frame, \( \tau \in \mathbb{R}^{3 \times 3} \) (the rotational analog to the second equation of motion, Eq. (4.3)). Following the same line of reasoning in deriving Eq. (4.6), we can see that the rate of change of any one body-fixed frame basis vector, \( u \in \{b_1, b_2, b_3\} \), due to the angular velocity is given by

\[
\frac{du}{dt} = \Omega \times u = \hat{\Omega}u.
\] (4.7)
For any differentiable vector-valued function $f(t) = f_x(t)b_1 + f_y(t)b_2 + f_z(t)b_3$, we can use Eq. (4.7) to find the time-derivative of $f$ as follows [23]:

\[
\frac{df}{dt} = \frac{df_x}{dt}b_1 + f_x \frac{db_1}{dt} + \frac{df_y}{dt}b_2 + f_y \frac{db_2}{dt} + \frac{df_z}{dt}b_3 + f_z \frac{db_3}{dt} \\
= \left(\frac{df_x}{dt}\right)_b + \frac{df_y}{dt}b_2 + \frac{df_z}{dt}b_3 + \Omega \times (f_x(t)b_1 + f_y(t)b_2 + f_z(t)b_3) \\
= \left(\frac{df}{dt}\right)_b + \Omega \times f(t),
\]

where $\left(\frac{df}{dt}\right)_b$ indicates the derivative of $f$ as seen in the body-fixed frame. Note that an observer in the body-fixed frame does not perceive The last remaining concept that needs to be defined to obtain the remaining equation of motion is the angular momentum, $L$, which satisfies $L = J\Omega$. The time-derivative of angular momentum is equal to the total torque, $\tau = [\tau_1, \tau_2, \tau_3]^T$ (in the body-fixed frame), which is exactly what we use to determine an equation for $\dot{\Omega}$.

\[
\tau = \frac{dL}{dt} \\
= \frac{d}{dt}J\Omega \\
= \left(\frac{d}{dt}J\Omega\right)_b + \Omega \times J\Omega \\
= J\dot{\Omega} + \Omega \times J\Omega
\]

Note that in the last step, since the derivative is taken in the body-fixed frame, the moment of inertia, $J$, does not change, so the term $J\dot{\Omega}$ resulting from the product rule vanishes. Rearranging, we obtain our final equation of motion:

\[
J\dot{\Omega} = \tau - \Omega \times J\Omega. \tag{4.8}
\]

We summarize the nonlinear dynamics of the deterministic model for quadrotor motion below [33].

\[
\begin{align*}
\dot{x} &= v \\
\dot{v} &= \frac{f}{m}Re_3 - ge_3 \\
\dot{R} &= R\dot{\Omega} \\
\dot{\Omega} &= J^{-1}(\tau - \Omega \times J\Omega)
\end{align*}
\tag{4.9}
\]
The inputs to this system are \( u = [f, \tau_1, \tau_2, \tau_3]^\top \in \mathbb{R}^m \) with dimension \( m = 4 \), and the state vector is given by \( X = [x, v, R, \Omega]^\top \in \mathbb{R}^3 \times \mathbb{R}^3 \times \text{SO}(3) \times \mathbb{R}^3 \).

Note that the low-level quadrotor controller must convert the input values to the individual torques to be applied to each propeller. We assume that the first and third propellers rotate clockwise, the second and fourth propellers rotate counterclockwise, and that the torque is directly proportional to the thrust generated by a propeller, with proportionality constant \( c_\tau \). Recall that \( f_i, i \in \{1, 2, 3, 4\} \) denote the thrusts generated, and define \( d \) to be the distance in the \( b_1b_2 \)-plane from the centre of mass to the centre of each propeller. Then, we can write the inputs as follows [27, 33]:

\[
\begin{bmatrix}
f \\
\tau_1 \\
\tau_2 \\
\tau_3 \\
\end{bmatrix} =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & -d & 0 & d \\
d & 0 & -d & 0 \\
-c_\tau & c_\tau & -c_\tau & c_\tau \\
\end{bmatrix}
\begin{bmatrix}
f_1 \\
f_2 \\
f_3 \\
f_4 \\
\end{bmatrix}.
\] (4.10)

Since this matrix is invertible provided \( d, c_\tau \neq 0 \), it suffices to invert the matrix and multiply by the vector of inputs, \( u \), in order to obtain the necessary thrusts, and therefore torques, for the individual propellers.

The remainder of this chapter solves the following problem, as similarly stated in [2], with the addition of satisfying a temporal logic specification.

**Problem Statement.**
Let \( \mathcal{X}_{\text{free}} \subseteq \mathbb{R}^3 \times \mathbb{R}^3 \times \text{SO}(3) \times \mathbb{R}^3 \) be the subset of state space that is unobstructed, and let \( U \subseteq \mathbb{R}^4 \) be the set of admissible control inputs. Furthermore, let \( \Phi \) be a deterministic \( \mu \)-calculus specification. Then, given the continuous-time quadrotor dynamical system,

\[
\dot{X}(t) = f(X(t), u(t)), \quad X(0) = X_0,
\] (4.11)

where \( f \) is given by the equations of motion in Eq. (4.9), determine a control signal,

\[
u(t) = [f(t), \tau_1(t), \tau_2(t), \tau_3(t)]^\top \in U,
\] (4.12)

and corresponding state trajectory,

\[
X(t) = [x(t), v(t), R(t), \Omega(t)]^\top \in \mathcal{X}_{\text{free}}
\] (4.13)

that satisfies specification \( \Phi \), or return failure if such a trajectory is not found.
4.2 Real-Time Motion Planning

As discussed in Chapter 2 and Chapter 3, many motion planning algorithms require knowledge of a steering function. kinoFMT (Algorithm 6) is one such algorithm, but as previously noted, the nonlinear quadrotor dynamics make it difficult (or perhaps impossible) to find an analytic solution to the OBVP. In order to apply the kinoFMT algorithm to the problem of kinodynamic planning for a quadrotor, we must therefore use an approximation to the 12D nonlinear system that has a known steering function. The crucial property involved in using an approximation to the full dynamics is called differential flatness, which allows any sufficiently smooth path to be tracked. The method employed by Allen et al. [2] involves two further important steps: reachable set approximation, and trajectory smoothing. To elaborate, the reachable set approximation is arguably the key step that allows for online planning, as it is used to rapidly connect the initial state and goal states to the preexisting tree that is to be computed beforehand offline. Once a path is planned between the newly added initial state and goal states, a smooth path is generated called a minimum-snap trajectory, and we leverage the differential flatness property of the quadrotor dynamics to be able to track this smooth path in the full dynamics. The details involved in tracking are provided, and simulations are performed to demonstrate the effectiveness of this method.

4.2.1 Framework Overview

We now outline the high-level real-time planning framework proposed in [2]. The entire process is broken down into two parts: the offline precomputation phase, where as much information as possible is gathered before knowing any of the specific details that will become clear during real-time trials, and the online planning phase, which takes into account the actual initial position and goal region and performs the various planning steps described in later sections.

Offline Phase

Before online planning can begin, it is desirable to perform as much precomputation as possible to minimize the computational effort, and therefore the time, required when online. With this in mind, there are three steps to perform offline: sampling the state-space, constructing a cost roadmap, and training a classifier that determines nearest nodes.

The sampling step simply stores a user-defined number of samples, $N$, in a set, $V$. The samples are drawn uniformly from the unobstructed state-space of the approximated
system, without any regard for potential obstacles (as such information is as yet unknown). This manner of sampling is a boon to the flexibility of the proposed method, as it can be applied online in a very general setting.

Next, for every pair of states in $V$, the OBVP is solved. The optimal time and cost are then stored as values in a look-up table (or dictionary), called Cost, associated with the corresponding pair of states. In this way, no computation for cost is required while running kinoFMT online, except when it comes to pairs involving the states known only when online: the initial state and sampled goal states. This issue is addressed in the final step of the offline phase.

Note that as $N$ gets large, the number of pairs grows quadratically as $N(N-1)$, and the bisection optimization method used to compute the optimal time for each pair converges linearly. For $N > 2000$, this can be rather expensive, so one could choose to instead sample some number of pairs on which to perform the precomputation. Furthermore, the ordering of the pairs matters, so one cannot use a symmetry argument to halve the number of computations. To illustrate this point, consider two states in one dimension (with 2D state-space: position and velocity), each with some positive velocity. The state that is behind has a fairly straightforward means of reaching the other state, whereas the state that is ahead would be forced to change direction, get to the appropriate position and accelerate in the positive direction once again to reach the desired positive velocity.

In order to avoid having to compute the cost between each of the initial or goal states and the existing $N$ samples, which would involve solving the OBVP $O(N)$ times, a machine learning approach is implemented. A Support Vector Machine (SVM) is used, learning from the data in the Cost look-up table to rapidly classify a pair of points as “near” or “not near” in terms of the cost incurred by traveling from one state to the other. As such, the OBVP must be solved only for those points that are estimated to fall within a certain cost threshold.

**Online Planning**

Upon beginning a trial with a quadrotor, the first step is to sample $N_{goal}$ states from the now-known goal region. Then, given the initial state of the quadrotor, we determine the outgoing nearest neighbours from the initial state using the trained SVM and store them in $N_{out}^{init}$, and we similarly determine the incoming neighbours for each of the goal states, storing them in $N_{in}^{goal}$. At this point, the OBVP is solved between the initial state (goal states) and the states in its neighbourhood, $N_{out}^{init}$ ($N_{in}^{goal}$), and the appropriate entries are added to the cost roadmap, Cost. Note that using the SVM to estimate cost-limited
reachable sets provides an immense reduction in the number of online OBVP solutions required, from $O(N)$ down to $O(1)$.

Now that the Cost look-up table is complete, the kinoFMT algorithm (Algorithm 6) is run on the set of samples, $V$ along with the initial and goal states. The algorithm quickly returns the optimal path from start to goal, excluding any paths that are found to collide with obstacles. However, given that the path is found using an approximated linear model, it must be smoothed so that it may be tracked by leveraging the differential flatness of the quadrotor system. With this aim, a smoothing algorithm takes the waypoints (states) from the path outputted by kinoFMT and produces a set of four high-degree polynomials in the flat output variables that are smooth up to fourth order. Finally, all that remains is to track the smooth path using an appropriately tuned feedback controller.

4.2.2 Differential Flatness

We say of a system that it is differentially flat if the states and the inputs can be written as functions of the system’s flat outputs and their derivatives. We state the definition more precisely as follows [11]:

**Definition 9.** Consider a continuous-time nonlinear system $\dot{x}(t) = f(x(t), u(t))$, $x(0) = x_0$ where $t \in \mathbb{R}$, $x(t) \in \mathbb{R}^n$ is the state, $u(t) \in \mathbb{R}^m$ is the input, and $f$ is a smooth function. This nonlinear system is *differentially flat* if there exists $\zeta(t) \in \mathbb{R}^m$, whose components are differentially independent, such that the following hold [10]:

\[
\begin{align*}
\zeta &= \Lambda(x, u, \dot{u}, \ldots, u^{(δ)}) \\
x &= \Phi(\zeta, \dot{\zeta}, \ldots, \zeta^{(ρ−1)}) \\
u &= \Psi^{-1}(\zeta, \dot{\zeta}, \ldots, \zeta^{(ρ)})
\end{align*}
\]

where $\Lambda, \Phi, \Psi^{-1}$ are smooth functions, $\zeta = [\zeta_1, \ldots, \zeta_m]^T$ is the vector of flat outputs, and $δ$ and $ρ$ are the maximum orders of the derivatives of $\zeta$ and $u$ necessary in defining the flat outputs and their relation to $x$ and $u$.

The concept of differential flatness is useful for many reasons, although two in particular seem eminently popular in the field of motion planning. The first is that differential flatness can be used in the process of feedforward (or feedback) linearization to separate a nonlinear system into a linear flat model and a nonlinear transformation, so that is possible to consider the control problem only on the linear part, and the resulting flat states and flat inputs can be used to correct for the nonlinear part via inversion of the
nonlinear transformation [11, 43]. The second consequence of differential flatness, and the one we focus on here, is that any smooth trajectory in the flat output space, subject to reasonably bounded derivatives, can be tracked [34]. The significance of this fact is not to be understated, as even the underactuated quadrotor can track a sufficiently smooth path generated from the flat outputs of the system.

In the case of quadrotors, the flat outputs can be chosen to be the position of the centre of mass in the inertial frame and the yaw angle, \( \zeta = [x_1, x_2, x_3, \psi]^\top \). Mellinger and Kumar prove that this choice of flat outputs does indeed admit a way to write the state and input as a function of \( \zeta \) and its derivatives up to fourth order.

### 4.2.3 Quadrotor Dynamics Approximation

There exist many known approaches to handling nonlinear dynamics when solving motion planning problems, such as using a planner that avoids the steering problem altogether (e.g., SST). Another approach, when dealing with differentially flat nonlinear systems, is to use feedforward linearization, as mentioned in Section 4.2.2. In the method presented here, as in [2], the quadrotor system is first approximated as a linear double integrator system (Eq. (4.14)). The approximation assumes the quadrotor can accelerate in any direction at any time, which, while crude, is sufficient to generate high-quality trajectories from an initial state to a goal region. The issue of sampling from a 12D system is also greatly reduced, as samples are instead taken from the approximated 6D system. The main idea is that, upon using \texttt{kinoFMT} to generate an ordered set of waypoints on this simplified system, a smooth trajectory in the flat outputs can be generated and tracked in the full dynamics.

\[
\dot{\tilde{x}}(t) = \begin{bmatrix} 0_{3 \times 3} & I_{3 \times 3} \\ 0_{3 \times 3} & 0_{3 \times 3} \end{bmatrix} \tilde{x}(t) + \begin{bmatrix} 0_{3 \times 3} \\ I_{3 \times 3} \end{bmatrix} \tilde{u}(t) - \begin{bmatrix} 0_{5 \times 1} \\ g \end{bmatrix}
\]  

(4.14)

Here, \( \tilde{x} = [x_1, x_2, x_3, \dot{x}_1, \dot{x}_2, \dot{x}_3]^\top \in \mathbb{R}^6 \) is simply a truncated representation of the full state including only position and velocity, and \( \tilde{u} = [\ddot{x}_1, \ddot{x}_2, \ddot{x}_3]^\top \in \mathbb{R}^3 \) is the new control. We denote the matrix multiplying \( \tilde{x}(t) \) by \( A \), the matrix multiplying \( \tilde{u}(t) \) by \( B \), and the constant vector by \( c \).

Now that we are working with a linear system, we solve the \texttt{OBVP} as in [37]. Given any two (sampled) states, we seek an analytic solution to the problem of finding an optimal path between them, as well as the optimal control signal used to generate such a path. We
begin by defining the cost function
\[
J(\tilde{u}, t_f) = \int_0^{t_f} 1 + \tilde{u}(t)^\top R_u \tilde{u}(t) dt
\] (4.15)
where \(R_u \in \mathbb{R}^{3 \times 3}\) is symmetric positive definite, and \(t_f\) is the fixed final time. This cost function prioritizes minimum-time solutions while also penalizing control effort. A suitable choice for the control penalty weighting matrix is \(R_u = w_R I_{3 \times 3}\), for some \(w_R \in \mathbb{R}\).

Without derivation, the optimal cost for the double integrator OBVP from initial state \(\tilde{x}_0\) at time \(t = 0\) to \(\tilde{x}_1\) at time \(t = t_f\) is given by [37, 2]
\[
J^*(t_f) = t_f + (\tilde{x}_1 - \bar{x}(t_f))^\top G(t_f)^{-1} (\tilde{x}_1 - \bar{x}(t_f)).
\] (4.16)
The state and control trajectories which achieve this optimal cost are given by
\[
\tilde{x}(t) = \bar{x}(t) + G(t) \exp(A^\top [t_f - t]) G(t_f)^{-1} (\tilde{x}_1 - \bar{x}(t_f))
\] (4.17)
\[
\tilde{u}(t) = R_u^{-1} B^\top \exp(A^\top [t_f - t]) G(t_f)^{-1} (\tilde{x}_1 - \bar{x}(t_f))
\] (4.18)
where
\[
\bar{x}(t) = \exp(At)x_0 + \int_0^t \exp(As)c \, ds
\] (4.19)
\[
= \exp(At)x_0 - \begin{bmatrix} 0 \\ 0 \\ g t^2/2 \\ 0 \\ 0 \\ g t \end{bmatrix}
\] (4.20)
\[
G(t) = \int_0^t \exp(As)BR_u^{-1}B^\top \exp(A^\top s) \, ds
\] (4.21)
\[
= \frac{1}{w_R} \begin{bmatrix} 
\frac{t^3}{3} & 0 & 0 & t^2/2 & 0 & 0 \\
0 & \frac{t^3}{3} & 0 & 0 & t^2/2 & 0 \\
0 & 0 & \frac{t^3}{3} & 0 & 0 & t^2/2 \\
\frac{t^2}{2} & 0 & 0 & t & 0 & 0 \\
0 & \frac{t^2}{2} & 0 & 0 & t & 0 \\
0 & 0 & \frac{t^2}{2} & 0 & 0 & t
\end{bmatrix}.
\] (4.22)

The only remaining unknown is the final time \(t_f = \arg\min_{t>0} J^*(t)\), which can be found via the bisection method performed on the derivative of the convex function \(J^*\).
This involves choosing an initial interval \([a, b]\) in which to check for an optimal solution (e.g., \([0.0001, 100]\)) as well as some error tolerance, \(\epsilon\). The bisection method is a recursive algorithm that checks to see whether the derivative of the function at \(a\) has the same sign as the derivative of the function at the midpoint of the interval, \(q\). If it is not the same, recurse on the interval \([a, q]\) since a turning point (i.e., a minimum) exists therein. Similarly, if the derivative at \(q\) has a different sign from the derivative at \(b\), recurse on \([q, b]\). The algorithm terminates and returns the midpoint once the length of the interval, \(b - a\), is less than \(\epsilon\), or when the derivative at the midpoint is zero.

### 4.2.4 Reachable Set Approximation

The problem of finding a state \(\tilde{x}_b\) that is nearest to state \(\tilde{x}_a\) in a kinodynamic framework is not as simple as finding the state \(\tilde{x}_b\) which minimizes the Euclidean distance between \(\tilde{x}_a\) and \(\tilde{x}_b\). Due to the issue of drift, including the consideration of momentum and angular momentum, distance is insufficient in determining the actual difficulty involved in transiting from one state to another. Instead, the cost is given by Eq. (4.15), used as the optimality condition for the OBVP. In the same vein, while geometric planners may use some maximal, distance-based search radius, our kinodynamic planner instead uses a cost-limited reachable set when looking for nearby states. We define such a reachable set from a state \(\tilde{x}_a\) with maximum cost \(J_{th}\), in the subset of unobstructed state space of the approximate dynamics, \(\tilde{X}_{free}\), and given the set of admissible input signals, \(\tilde{U}\), as follows [1]:

\[
R(\tilde{x}_a, \tilde{U}, J_{th}) = \{\tilde{x}_b \in \tilde{X}_{free} \mid \exists \tilde{u} \in \tilde{U}, t \in [0, t_f] \text{ s.t. } \tilde{x}(t) = \tilde{x}_b \text{ and } J(\tilde{u}, t) \leq J_{th}\}. \tag{4.23}
\]

In words, the cost-limited reachable set contains all unobstructed states that can be reached before the final time, \(t_f\), using admissible controls and without exceeding the cost threshold.

The issue posed by real-time planning is that solving the OBVP from the previously unknown initial state to each of the \(N\) sampled states is computationally expensive. This would then have to be repeated for each of the newly sampled goal states, rendering the task of online kinodynamic planning practically infeasible. Even worse, if the number of samples is large and the look-up table \(\text{Cost}\) does not include the cost between every pair of the \(N\) samples, then querying for nearby states requires even more OBVP solutions. What has been proposed in [2] is to use an SVM classifier to estimate whether any given state lies within the cost threshold, \(J_{th}\), of another state. That is, when calling \(\text{Near Forward}(x, V, J_{th})\) from Algorithm 6, the SVM is used to estimate, for each \(v \in V\), whether or not \(v \in R(x, \tilde{U}, J_{th})\). Similarly, \(\text{Near Backward}(x, V, J_{th})\) returns the set of states \(v\) such that \(x \in R(v, \tilde{U}, J_{th})\).

51
An SVM works by consuming a large array of training data, `arr_train`, with `n_{train}` entries called feature vectors, as well as an array of `n_{train}` labels. The idea is to train the supervised learning algorithm on the correct labels to be able to partition the space of feature vectors in such a way as to be able to accurately classify new feature vectors. This partitioning can be accomplished in many ways, such as using a simple linear hyperplane as a boundary, or creating more complex nonlinear boundaries. The boundary that is used is determined by the chosen kernel function.

Our implementation uses the scikit-learn package “svm” for Python. We begin by determining what should be included in the feature vector. One simple choice for the feature vector that is sufficient for our purposes is to concatenate the pair of states. Given a pair of states \((\tilde{x}_a, \tilde{x}_b)\), \(\tilde{x}_a, \tilde{x}_b \in \tilde{X}_{free} \subseteq \mathbb{R}^6\) (in the approximate dynamics), let the \(i\)th feature vector of `arr_train` be \(p_i = [\tilde{x}_a^\top, \tilde{x}_b^\top]^\top\). The corresponding label \(y_i\) is equal to 1 if the actual optimal cost from \(\tilde{x}_a\) to \(\tilde{x}_b\) is less than \(J_{th}\), and 0 otherwise. In our work, we found the most success using a polynomial kernel of degree 3. All that remains is to choose an error penalty parameter, \(C\), as input to the SVC function from the scikit-learn svm package. As with many machine learning techniques, this step is subject to trial and error. Once chosen, the SVC function can be used to train a classifier. Given a new pair of vectors, one can then construct the appropriate feature vector and run it through the trained classifier to check whether or not the pair satisfies the cost-limited reachability condition.

We encourage interested readers to refer to [39] for further details regarding SVMs.

4.2.5 Trajectory Smoothing

The path generated by `kinoFMT` cannot be used directly as it is based on the double integrator dynamics approximation\(^1\). To use the generated path, we must first create a smooth trajectory in each of the four flat outputs: the three components of position and the yaw angle. We opt to use polynomial interpolation on the waypoints of the path generated by our planning algorithm. This section is primarily based on the work of Richter et al. [36], though many details that are missing from these papers are provided.

To accomplish the task of smoothing, we will use \(M\) polynomials of order \(N_p\) for each of the four flat output variables. To introduce the topic, we will begin by analyzing how the interpolation task is accomplished for one polynomial segment between two waypoints.

---

\(^1\)Note that if we were to apply a different method that could plan on the full nonlinear dynamics, such as with SST (Chapter 3), smoothing would not be a necessary step, although it can be useful in improving trajectory quality.
for a single flat output variable. We will then extend the result to create $M$ polynomials, and the procedure may be repeated for each of the other flat output variables.

Based on the proof that the quadrotor system is differentially flat [34], it is shown that four derivatives of the flat outputs are required to express the state and the input. For this reason, we require that each polynomial we construct is continuous up to the fourth derivative. Furthermore, each polynomial must have equal derivatives at shared intermediate waypoints to ensure smoothness of the full trajectory. Despite these restrictions, there remain infinitely many possible polynomials that join successive waypoints. We choose the “best” option which we define to be the unique polynomial that minimizes the integral of the square of the snap (fourth derivative) of the polynomial, as shown in Eq. (4.24).

$$J_{\text{snap}}(T) = \int_0^T P^{(4)}(t) dt = p^\top Q(T) p$$

(4.24)

Here, $T$ is the fixed final time for the given polynomial segment, which was found when solving the OBVP and subsequently recording the optimal duration and cost in the Cost roadmap, and $Q$ is the Hessian matrix for the integral expression with respect to the vector of polynomial coefficients, $p$. $P(t)$ is given by

$$P(t) = \sum_{i=0}^{N_p} p_i t^i, \quad p = [p_0, p_1, \ldots, p_{N_p}]^\top.$$  

(4.25)

Considering a generic polynomial of order $N_p$, we can take four derivatives and compute the integral of the square of the result to determine an expression for $Q$ from Eq. (4.24), as in [2]:

$$Q_{ij}(T) = \begin{cases} 2 \left( \frac{i! j!}{(i-j)! j!} T^{i+j-\gamma} \right), & i \geq 4 \land j \geq 4 \\ 0, & \text{otherwise.} \end{cases}$$

(4.26)

Note that the indexing used in this section will follow the computer science convention, so the top-left entry of a matrix has index $(i, j) = (0, 0)$.

Interpolation requires the trajectory to pass through its terminal endpoints. Let $d$ be a vector of the derivatives at the initial waypoint ($d_0$) concatenated with the derivatives at the following waypoint ($d_T$). Some of the derivative values may not be known, however, particularly at intermediate waypoints. Let $\beta$ represent the number of unknown derivatives, and let $\delta$ be the total number of derivatives kept in each of $d_0, d_T$. We will determine an appropriate value for $\delta$ when working on the extended problem involving all $M$ polynomial segments. Note that the unknown derivatives can be left free so that our procedure assigns
them optimal values, but they must still satisfy continuity. We can encode the continuity constraints as follows:

\[ Ap = d, \quad \text{with} \quad A = \begin{bmatrix} A_0 \\ A_T \end{bmatrix}, \quad d = \begin{bmatrix} d_0 \\ d_T \end{bmatrix} \] (4.27)

where

\[ A_{0,ij} = \begin{cases} j! , & i = j \\ 0 , & \text{otherwise} \end{cases} \quad d_{0i} = P^{(i)}(0) \] (4.28)

\[ A_{T,ij} = \begin{cases} \frac{j!}{(j-i)!}T^{j-i} , & j \geq i \\ 0 , & j < i \end{cases} \quad d_{Ti} = P^{(i)}(T). \]

The expressions for \( A_0 \) and \( A_T \) are simply the result of differentiating \( P(t) \) and evaluating at \( t = 0 \) and \( t = T \), respectively, to determine the appropriate coefficients by which to multiply each of the polynomial coefficients in \( p \).

Now, we are left with the problem of minimizing \( J_{\text{snap}} \) subject to the constraint given in Eq. (4.27). This is called a constrained quadratic program (QP), and the one we are working with here tends to be numerically unstable when the problem is extended to multiple segments. However, it is possible to reformulate the problem as an unconstrained QP by optimizing over the vector of derivatives instead of the polynomial coefficients. All that is required is to invert Eq. (4.27) to obtain \( p = A^{-1}d \), so that we can rewrite Eq. (4.24) as

\[ J_{\text{snap}}(T) = d^\top A^{-\top} Q(T) A^{-1} d \] (4.29)

where we use the notation \( A^{-\top} \) to denote the transpose of the inverse of matrix \( A \). The unconstrained problem does not suffer from the same numerical instability as the constrained problem, so we proceed by extending the optimization over \( M \) polynomials with this reformulated problem.

Define \( A_{1..M} \) and \( Q_{1..M} \) to be block diagonal matrices containing the \( A \) and \( Q \) matrices (respectively) corresponding to the appropriate polynomial segment; that is, the first block in \( A_{1..M} \) (\( Q_{1..M} \)) contains the matrix \( A \) (\( Q \)) for the polynomial between the initial waypoint and the second waypoint, then the next block on the diagonal corresponds to the polynomial between the second and third waypoint, and so on.

We could proceed in a similar fashion for vector \( d \), concatenating all of the derivative vectors to obtain \( d_{1..M} \), but this results in a vector with an unorganized mix of known and free derivative values. Instead, we will reorder the vector such that the all of the fixed
(known) derivative values appear first, followed by all of the free derivatives that remain to be optimized,

\[ d_{\text{order}} = \begin{bmatrix} d_{\text{fix}} \\ d_{\text{free}} \end{bmatrix}, \quad d_{\text{order}} = Cd_{1..M}, \tag{4.30} \]

where \( C \) is a pseudo-ordering matrix that also encodes continuity. Reordering of the concatenated vector, \( d_{1..M} \), works by letting \( C \) be the identity matrix with appropriately swapped rows. Encoding continuity involves carefully choosing some entries in \( C \) to have value \(-1\), so that pairs of elements of \( d_{1..M} \) that ought to be equal are subtracted. The difference is forced to evaluate to 0 by placing half of the free derivatives in the vector of fixed derivatives, \( d_{\text{fix}} \), with value 0, since the derivatives at the end point of one polynomial must be equal to the derivatives of the starting point of the next polynomial. See Eq. (4.31).

Without loss of generality, we can assume that the unknown values of \( d_T \) for polynomial segment \( j \) are “known” and set to be 0 in \( d_{\text{fix}} \), while the corresponding unknown elements in \( d_0 \) for polynomial \( j + 1 \) remain as unknown values in \( d_{\text{free}} \).

\[ d^j_{Ti} - d^{j+1}_{0i} = 0, \quad \text{since} \quad A_T^j p^j = A_0^{j+1} p^{j+1} \tag{4.31} \]

where the subscript \( i \) denotes the \( i^{th} \) component of the vector, and the superscript \( j \) denotes the polynomial segment to which the matrix or vector corresponds, \( j \in \{0, \ldots, M - 1\} \).

We now return to the question: what is \( \delta \), the number of derivatives that we keep in each \( d_0, d_T \)? Consider the extended constraint equation,

\[ A_{1..M}p_{1..M} = d_{1..M}, \tag{4.32} \]

where \( p_{1..M} \) is the vector that results from concatenating all of the polynomials coefficients for each of the \( M \) polynomials. Eq. (4.32) is a system of \( 2\delta M \) equations, since each \( d^j = [d^j_0, d^j_T]^T \) contains \( 2\delta \) derivatives of the \( j^{th} \) polynomial, \( P_j(t) \). We assume we know all derivatives at the initial and final waypoint, and that we know \( \beta \) derivatives at every intermediate waypoint. There are a total of \( M + 1 \) waypoints, and we know all derivatives for two of them, leaving \( (M - 1) \cdot 2(\delta - \beta) \) unknown derivatives. As discussed, half of these unknowns are identical to the other half, so in total there are in fact only \( (M - 1)(\delta - \beta) \) unknown derivatives. Moreover, there are \( M(N_p + 1) \) unknown polynomial coefficients. Ensuring the problem is never over-constrained, we use the number of equations and the total number of unknown values to solve for \( \delta \), yielding the inequality:

\[ \delta \leq \left\lfloor \frac{M(N + 1) - \beta(M - 1)}{M + 1} \right\rfloor. \tag{4.33} \]
Continuing, we may rewrite $J_{\text{snap}}$ in this extended form,

$$J_{\text{snap}}(T) = \begin{bmatrix} d_{\text{fix}} \\ d_{\text{free}} \end{bmatrix}^\top C^{-\top} A_{1..M}^{-\top} Q_{1..M}(T) A_{1..M}^{-1} C^{-1} \begin{bmatrix} d_{\text{fix}} \\ d_{\text{free}} \end{bmatrix},$$

(4.34)

and we note that, since $C$ is not strictly a permutation matrix, $C^\top \neq C^{-1}$ in general.

Defining $H = C^{-\top} A_{1..M}^{-\top} Q_{1..M}(T) A_{1..M}^{-1} C^{-1}$, we can write an expression for $J_{\text{snap}}$ in block matrix form:

$$J_{\text{snap}} = \begin{bmatrix} d_{\text{fix}} \\ d_{\text{free}} \end{bmatrix}^\top \begin{bmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{bmatrix} \begin{bmatrix} d_{\text{fix}} \\ d_{\text{free}} \end{bmatrix}$$

(4.35)

$$\frac{dJ_{\text{snap}}}{d(d_{\text{free}})} = d_{\text{fix}}^\top H_{00} d_{\text{fix}} + d_{\text{fix}}^\top H_{01} d_{\text{free}} + d_{\text{free}}^\top H_{10} d_{\text{fix}} + d_{\text{free}}^\top H_{11} d_{\text{free}}$$

(4.36)

Since each $Q^j$ is symmetric by its definition (Eq. (4.26)), $Q_{1..M}$ is symmetric, so $H$ is also symmetric. Setting Eq. (4.37) to 0, we can then simplify to obtain an expression for the optimized values of the free derivatives,

$$d_{\text{free}}^* = -H_{11}^{-1} H_{00}^\top d_{\text{fix}}.$$

(4.38)

All that remains is to invert the extended constraint equation to solve for the $M(N_p + 1)$ polynomial coefficients:

$$p_{1..M} = A_{1..M}^{-1} d_{1..M}^*,$$

where $d_{1..M}^* = C^{-1} \begin{bmatrix} d_{\text{fix}} \\ d_{\text{free}} \end{bmatrix}$.

(4.39)

**Smoothing Example**

Consider smoothing a trajectory over three waypoints ($M = 2$ polynomials): the initial state, the final state, and one intermediate state. Suppose we want to interpolate with polynomials of degree $N_p = 3$, and assume that we know $\beta = 1$ derivatives (i.e., only the actual value; we have no knowledge of the first or any subsequent derivatives) at the intermediate state. Using Eq. (4.33), we determine $\delta = 2$. Then, Eq. (4.32) becomes,

$$\begin{bmatrix} A^0 & 0 \\ 0 & A^1 \end{bmatrix} \begin{bmatrix} p^0 \\ p^1 \end{bmatrix} = \begin{bmatrix} d^0 \\ d^1 \end{bmatrix}$$

(4.40)
where $T_0, T_1$ are the final times for the first and second segments, respectively, and we start the clock back at $t = 0$ for each new segment. The polynomials we seek to find are of the form $P_j(t) = \sum_{i=0}^{N_p} p_j^i t^i$, for $j \in \{0, 1\}$. Please note that the subscripts $T_0, T_1$ of the derivative values do not refer to the final times, but rather to the indices of vectors $d^0_T$ and $d^1_T$.

Determining the reordering matrix involves first deciding how to order $d_{1:M}$ into a vector of the form $[d_{fix}, d_{free}]^\top$. Recalling that $\beta = 1$, and since there is only one intermediate waypoint, only two values are unknown (but equal): $d_{T_1}^0$ and $d_{0_1}^1$. As discussed, we can set $d_{T_1}^0$ to be a “known” value, 0, so that

$$d_{order} = \begin{bmatrix} d_{fix} \\ d_{free} \end{bmatrix} = \begin{bmatrix} d_{T_0}^0 \\ d_{0_1}^0 \\ d_{T_1}^0 \\ d_{0_1}^1 \\ d_{T_0}^1 \\ d_{T_1}^1 \end{bmatrix} \begin{bmatrix} d_{fix} \\ d_{free} \end{bmatrix}$$

(4.41)
and the corresponding pseudo-ordering matrix is given by,

\[
C = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
\]  

(4.42)

which satisfies Eq. (4.30). The \(-1\) entry corresponds to the single unknown, \(d_{0_1}^1\), which must be equal to \(d_{0_1}^0\). Upon multiplying \(Cd_{1..M}\), the two values are subtracted, and the corresponding value in \(d_{fix}\) is chosen to be 0 to enforce the continuity constraint. This same method can be applied for any number of unknown but equal pairs at intermediate waypoints.

We could then proceed to apply Eq. (4.38) to compute \(d_{\text{free}}^*\), which ultimately allows us to find the polynomial coefficients, \(p_{1..M}\), using Eq. (4.39). Given that this is a toy example, however, the small degree of polynomials used \((N_p = 3)\) implies that the snap, i.e., fourth derivative, is always zero, so there is nothing meaningful to minimize. The purpose of this example is to elucidate the method used. On the other hand, larger problems, while more pertinent, lead to unwieldy matrix equations best left for a programming environment.

A reasonable concern is that the new smooth trajectory may now intersect with obstacles. During the original path planning performed by kinoFMT, any transitions that would lead to a collision were discarded, meaning there must exist a path that successfully navigates the obstacles. To handle the issue of possible collisions, we can therefore successively add waypoints at the midpoints of the segment in which a collision is detected, and the smoothing operation is performed every time a new waypoint is added. In this way, the smooth trajectory can lie as close as is necessary to the path found using kinoFMT [36].

As a final remark, recall that the smoothing procedure must produce a set of polynomials for each of the flat outputs: \(x_d = [x_{1d}, x_{2d}, x_{3d}]^\top\), the desired \(x, y, z\) position, and \(\psi_d\), the desired yaw angle. While the position of the waypoints is contained directly in the waypoint states of the approximated system, the yaw angle is not. In practice, the yaw angle can be specified by the user. Some common choices include maintaining a constant heading, \(\psi_d = \psi_0 \in \mathbb{R}\), or facing in the direction of travel, using \(\psi_d = \arctan(\dot{x}_{2d}/\dot{x}_{1d})\).
4.2.6 Tracking Controller

Leveraging the differential flatness of the quadrotor system, we can use a feedback controller to track any smooth path with reasonably bounded derivatives, such as the path generated using the trajectory smoothing technique above. In this section, the geometric tracking controller developed in [27] is presented along with some clarifying details.

**Tracking Errors**

The first step involved in designing our tracking controller is to define the tracking errors. Recall that the full state of the quadrotor dynamical system is given by $X = [x, v, R, \Omega]^T$. The errors for the position and velocity are simply:

$$e_x = x - x_d, \quad (4.43)$$
$$e_v = v - v_d, \quad (4.44)$$

where the vector $x_d = [x_{1d}, x_{2d}, x_{3d}]^T$ is the desired position, as determined by the polynomial trajectories generated in the smoothing step, and $v_d = \dot{x}_d$ is the desired velocity.

Determining an error vector for the orientation, $R$, requires more careful consideration due to the nonlinear nature of the spaces in which $R$ and $R_d$ evolve, where $R_d$ represents the desired rotation matrix, i.e., the desired orientation of the quadrotor. Consider the following error function on $SO(3)$,

$$\Psi(R, R_d) = \frac{1}{2} \text{tr}(I - R_d^T R). \quad (4.45)$$

Let us investigate some properties of this function in order to provide some guarantees about the derivation of the tracking controller that follows. Let $S = R_d^T R \in SO(3)$. Then by orthogonality, all eigenvalues of $S$ lie on the complex unit circle, and since $S$ is real, any complex eigenvalues come as conjugate pairs. In the case that all eigenvalues are real, since $\det(S) = 1$ and since, for any matrix, the determinant is the product of the eigenvalues, it must be that $\lambda_i = 1$ or $-1$, $i \in \{1, 2, 3\}$, where $\lambda_i$ are the eigenvalues of $S$. Since $\text{tr}(S) = \lambda_1 + \lambda_2 + \lambda_3$, and because there must be an even number of negative eigenvalues to ensure the determinant is positive, we therefore have that $-1 \leq \text{tr}(S) \leq 3$. Now consider the case where there is a pair of complex conjugate eigenvalues, $\lambda_2 = \bar{\lambda}_1$. Then $1 = \det(S) = \lambda_1 \lambda_2 \lambda_3 = |\lambda_1|^2 \lambda_3 = \lambda_3$. Thus, $\text{tr}(S) = 2\Re(\lambda_1) + \lambda_3$ which implies $-1 \leq \text{tr}(S) \leq 3$ [13]. In both cases, we obtain the same conclusion, and this result implies that the error function provided in Eq. (4.45) satisfies $0 \leq \Psi(R, R_d) \leq 2$. From this we can glean
that $\Psi(R, R_d)$ is always positive if $R \neq R_d$. In the other extreme, when $\text{tr}(R_d^T R) = -1$ so that $\Psi(R, R_d) = 2$, we can see that $R, R_d$ must differ by a rotation angle of $180^\circ$. The tracking controller developed in [27] guarantees that the zero equilibrium of the tracking errors is exponentially attractive provided that, initially, $\Psi(R, R_d) < 2$.

Our goal is to find an expression for $R$ that minimizes the error, $\Psi(R, R_d)$. We begin by defining the Frobenius inner product for real matrices $A, B$ of equal dimension:

$$\langle A, B \rangle_F = \text{tr}(A^T B). \quad (4.46)$$

This inner product satisfies the product rule from differential calculus, and we use it to rewrite $\Psi(R, R_d)$ as

$$\Psi(R, R_d) = \frac{1}{2} (3 - \langle R_d, R \rangle_F), \quad (4.47)$$

by linearity of the trace. Then, proceeding with the usual calculus procedure of minimization, with $dR = R \hat{\eta}$ for arbitrary unit vector $\eta \in \mathbb{R}^3$ ($\hat{\eta} \in \mathfrak{so}(3)$) as the axis of rotation, as discussed in Section 4.1.2, we compute the infinitesimal element of $\Psi$ with respect to $R$:

$$D_R \Psi(R, R_d) = -\frac{1}{2} \langle (0, R, +)_F \langle R_d, dR \rangle_F \rangle \quad (4.48)$$

$$= -\frac{1}{2} \text{tr}(R_d^T R \hat{\eta}) \quad (4.49)$$

$$= -\frac{1}{2} \text{tr}(\hat{\eta}^T R^T R_d), \quad \text{tr}(A) = \text{tr}(A^T) \quad (4.50)$$

$$= \frac{1}{2} \text{tr}(\hat{\eta} R^T R_d), \quad \hat{\eta}^T = -\hat{\eta} \quad (4.51)$$

$$= \frac{1}{2} \text{tr}(R^T R_d \hat{\eta}), \quad \text{tr} ABC = \text{tr} BCA. \quad (4.52)$$

By the equality of Eq. (4.49) and Eq. (4.52), we proceed to obtain

$$D_R \Psi(R, R_d) = -\frac{1}{4} [\underbrace{\text{tr}(R_d^T R - R^T R_d) \hat{\eta}}_{\hat{e}_R}] \quad (4.53)$$

$$= \frac{1}{2} \langle \hat{e}_R, \hat{\eta} \rangle_F, \quad -\frac{1}{2} \text{tr}(\hat{x} \hat{y}) = x^T y \quad (4.54)$$

where we defined the quantity

$$\hat{e}_R = R_d^T R - R^T R_d, \quad (4.55)$$

which is necessarily an element of $\mathfrak{so}(3)$ since $\hat{e}_R^T = -\hat{e}_R$. Setting $D_R \Psi(R, R_d) = 0$, and since $\eta$ is an arbitrary axis of rotation, we see that $e_R$ is an appropriate choice of error
vector for the attitude of the quadrotor [27]. In defining \( e_R \), we use the inverse of the hat map, the \( \vee : \mathfrak{so}(3) \to \mathbb{R}^3 \), which acts on a skew symmetric matrix and returns a vector, \( a \), satisfying \( ab = a \times b \), for all \( b \in \mathbb{R}^3 \).

\[
e_R = (R_d^T R - R_d^T R_d)^\vee (4.56)
\]

Lastly, we seek an error vector for the angular velocity. Begin by noticing that \( \dot{R} \in T_R \text{SO}(3) \) and \( \dot{R}_d \in T_{R_d} \text{SO}(3) \) lie in different tangent spaces. Left-multiplying an element of \( T_{R_d} \text{SO}(3) \) by the inverse of the element at which we centre the tangent bundle, \( R_d^T \), results in an element of the tangent space at the identity, \( T_1 \text{SO}(3) \). Left-multiplying this result by \( R \), we obtain an element of \( T_R \text{SO}(3) \). So, in order to compare \( \dot{R} \) and \( \dot{R}_d \) in the same tangent space, we proceed as follows:

\[
\dot{R} - RR_d^T \dot{R}_d = R(\dot{\Omega} - RR_d^T \dot{R}_d \dot{\Omega}_d), \quad \dot{R} = R \dot{\Omega} (4.57)
\]
\[
= R(\dot{\Omega}_d), \quad R_d^T R_d = I (4.58)
\]
\[
= R(\Omega - \Omega_d)^\wedge (4.59)
\]

where the last line follows from the fact that the hat map is distributive. From this, we can see that the expression is 0 if the vector \( \Omega - R_d^T \dot{R}_d \Omega_d = 0 \), so this becomes the error vector for the angular velocity:

\[
e_{\Omega} = \Omega - \Omega_d (4.60)
\]

Note that, while this seems like the obvious choice, some subtlety is involved in ensuring it is sensible. This same procedure is used by Lee et al. in [27], except that the authors converted \( R_d \) to the tangent space centered at \( R \) using right-multiplication, resulting in a slightly more complex expression for \( e_{\Omega} \).

**Control Laws**

Now that the tracking errors have been defined, we begin to formulate the the tracking controller as in [27, 33]. We define the desired thrust vector using the second equation of motion of the dynamical system, Eq. (4.3), which we solve for the total thrust term (with the desired acceleration). In designing the full desired thrust vector, we include the position and velocity errors, as the thrust vector is directly responsible for correcting the translational position and velocity. This yields:

\[
F = fRe_3 = m\ddot{x}_d + mg\varepsilon_3, \quad \text{total (desired) thrust} (4.61)
\]
\[
F_d = F - K_\varepsilon \varepsilon_\varepsilon - K_v \varepsilon_v, \quad \text{desired thrust with error} (4.62)
\]
where \( k_x, k_v \) are positive definite control gain matrices. Assuming \( \|F_d\| \neq 0 \), we can find the desired third body-fixed axis, \( b_{3,d} \), as the unit vector in the direction of \( F_d \),

\[ b_{3,d} = \frac{F_d}{\|F_d\|}. \tag{4.63} \]

Since we also have access to the desired yaw angle, \( \psi_d \), as it is a flat output for which we constructed a polynomial trajectory, we can write:

\[ b_{1,c} = [\cos(\psi_d), \sin(\psi_d), 0]^\top. \tag{4.64} \]

This is an intermediate value that we use to define the desired second body-fixed axis, relying on the right-handedness of our coordinate systems:

\[ b_{2,d} = \frac{b_{3,d} \times b_{1,c}}{\|b_{3,d} \times b_{1,c}\|}. \tag{4.65} \]

The desired first body-fixed axis can then be defined similarly as

\[ b_{1,d} = b_{2,d} \times b_{3,d}, \tag{4.66} \]

and the desired rotation matrix can simply be expressed by placing these three basis vectors into a \( 3 \times 3 \) matrix,

\[ R_d = [b_{1,d}, b_{2,d}, b_{3,d}]. \tag{4.67} \]

Next, we must find an expression for the desired angular velocity. The result is derived in [34] and reiterated in the NED frame in [2]. We include the pertinent results here for completeness.

\[ \Omega_d = \Omega_{1,d} b_{1,d} + \Omega_{2,d} b_{2,d} + \Omega_{3,d} b_{3,d} \tag{4.68} \]

where we define the following useful quantities

\[ u_{1,ff} = F \cdot Re_3 \tag{4.69} \]

\[ h_\Omega = \frac{m}{u_{1,ff}} [(x_d^{(3)} \cdot b_{3,d}) b_{3,d} - x_d^{(3)}]. \tag{4.70} \]

The feedforward thrust, \( u_{1,ff} \), projects the total desired thrust onto the current third body-fixed axis. Note that if the actual orientation is 90° from the total desired thrust (without the correcting error terms), \( u_{1,ff} \) is zero as the body must be rotated before the thrust can
be used to correct the error. We use $h_\Omega$ to define the components of the desired angular velocity as follows:

\[
\begin{align*}
\Omega_{1d} &= -h_\Omega \cdot b_{2d} \\
\Omega_{2d} &= h_\Omega \cdot b_{1d} \\
\Omega_{3d} &= \dot{\psi}_d (e_3 \cdot b_{3d}).
\end{align*}
\] (4.71)

The tracking controller we present here is almost identical to the one developed by Mellinger et al. [34]. Some small differences exist due to our choice of tracking error vectors and minor sign changes in some definitions above. We choose the control torque, $\tau$, used by Mellinger et al. as it omits some complexity from [27], which was found to be unnecessary in practice\footnote{All examples we have tried perform excellently with the simple expression for the control torque, although this does not necessarily imply that the complexity of the controller in [27] is never useful. Large errors may benefit from the feedforward terms involving angular acceleration as well as the consideration of a non-diagonal moment of inertia matrix.}. Instead, the control torque vector relies exclusively on the attitude and angular velocity tracking errors and their respective gain matrices, $K_R, K_\Omega$.

\[
\begin{align*}
f &= -(m\ddot{x}_d + mg e_3 - K_x e_x - K_v e_v) \cdot R e_3 \\
\tau &= K_R e_R + K_\Omega e_\Omega.
\end{align*}
\] (4.72, 4.73)

Recall that these four components of the controller, $u = [f, \tau]^T$, can be converted into the necessary rotor torques via inversion of Eq. (4.10).

Since the orientation error, $e_R$, is not based on problematic Euler angles, no small angle approximation is used and singularities are avoided. This means that tracking is feasible even for very large deviations in orientation, except when the quadrotor body is completely inverted (exactly 180° from the desired orientation). Proof of almost global exponential attractiveness of a similar controller is provided in [27], given that:

\[
\|e_\Omega(0)\|^2 < \frac{2}{\lambda_{\text{min}}(J)} k_R \left( 1 - \frac{1}{2} \text{tr}[I - R_d^T(0)R(0)] \right)
\] (4.74)

where $K_R = k_r I$.

### 4.2.7 Simulations

Now that the method has been introduced in detail, we are ready to present simulation results.
First, we perform all of the necessary offline computation on four $10 \times 10 \times 10$ environments, where we choose $N_p = 200, 200, 1000, 2000$ samples, and cost threshold $J_{th} = 300, 260, 200, 150$, respectively. We choose decreasing values of $J_{th}$ as the number of samples increases since more samples implies a higher probability of reachable states; as such, we attempt to keep the number of online OBVP computations to a minimum while still connecting to a sufficient number of points in the tree. Note that we have precomputed the entire Cost look-up table for each configuration, so that the solution to the OBVP for every pair of sampled points is known before online initiation.

When online, we choose the initial state to be $x_0 = (1.5, 1, 0, 0, 0, 0)$ (starting from rest) and the goal region is defined as the $1 \times 1 \times 2$ rectangular prism whose front-bottom-left corner is positioned at $(6,9,8)$. Moreover, each goal region is explicitly sampled $m = 5$ times to ensure a variety of possible goal states. We also include two large obstacles, shown in red, which the quadrotor must avoid. The quadrotor is constrained to stay within the bounding box, and aims to reach the goal, shown in cyan.

Upon running the online planning framework, and before smoothing is applied, we see the result of running kinoFMT in Figure 4.3. Each is the minimum cost path from initial state to goal region for the provided set of random samples. The case with $N_p = 200$ and the lower cost threshold, $J_{th} = 260$ was the fastest, taking 3.98 seconds, 0.58 seconds of which was incurred due to expensive collision checks which are considerably faster in physical experiments (sensors can be queried very rapidly). Interestingly, the path incurring the least cost resulted from the $N_p = 200, J_{th} = 300$ case, although it was nearly identical in cost to the path produced on the configuration over 2000 samples.

Trajectory smoothing is run on the same configurations to obtain Figure 4.4.

Finally, we demonstrate the effectiveness of the tracking controller presented in Section 4.2.6. The implementation of the tracking controller involves defining a function for the system dynamics from Eq. (4.9). Doing so allows us to forward integrate from the initial state using the control law from Eq. (4.73). Numerical integration is performed using the scipy.integrate.ode class with method set to “vode” which is the real-valued variable-coefficient ordinary differential equation solver. As we demonstrate only a simulation, the control inputs $f$ and $\tau$ can be used directly, so the actual rotor torques need not be computed.

The physical parameters are chosen to closely approximate a very small quadrotor, such as the one used in [31]. As such, the mass is set to $m = 0.3$ kilograms, and the moment of inertia matrix is diagonal, letting $J = 0.00002I$, with units $Kg \cdot m^2$. Upon tuning the controller with appropriate gain matrices $K_x, K_v, K_R, K_\Omega$, we are able to track a sequence including a unit step, a linear ramp, and a quadratic curve. Note that all subsequent
simulations in this section are performed on the set of 200 samples with $J_{th} = 260$. The tracking quality can be seen in Figure 4.5, and the tracking errors are displayed in Figure 4.6. It is clear that the controller successfully guides the controller toward the desired trajectory very rapidly, maintaining very small tracking errors except when the desired trajectory changes suddenly. We further demonstrate the efficacy of the tracking controller on a smooth path from the initial state to the goal in Figure 4.7.

Figure 4.2: Shown here are the tree structures generated for four different configurations. Starting from the top-left and reading across, we list the pairs $(N_p, J_{th})$ used: (200, 300), (200, 260), (1000, 200), (2000, 150). Edges represent collision-free paths between pairs of waypoints. Note that the visualization software displays an unobstructed view of at least one object, giving the false impression that no edges exist in front of the clearly visible obstacle.
Figure 4.3: Here we see the best-path waypoints as seen before smoothing is applied. \texttt{kinoFMT} returns a piecewise-linear path connecting the waypoints returned, shown in blue. The four configurations appear in the same order as in Figure 4.2.

Figure 4.4: Upon running the trajectory smoothing algorithm, we obtain the green curves shown here. Note that the curve always intersects the piecewise linear path (blue) at the waypoints.
Figure 4.5: The $x$, $y$, $z$ positions resulting from tracking the dashed green reference trajectory are plotted. Response to a unit step, linear ramp, and quadratic curve are shown.
Figure 4.6: The norm of each of the error vectors is plotted. These error vectors arise from tracking the piecewise reference of Figure 4.5.
Figure 4.7: Tracking the polynomial trajectory generated for the $N_p = 200$, $J_{th} = 260$ configuration (top-right instance of Figure 4.4). The nominal trajectory is shown as a dashed green line, though it is obscured by the simulated trajectory obtained using the tracking controller (black).
4.3 Abstracted Kripke Structures for Online Planning

In Chapter 3, we introduced the notion of an abstracted Kripke structure which acts as a simplified map, describing how the state-space can be traversed to reach various goal regions. It was shown that the motion planning algorithm SST* could be used to incrementally generate trajectories to multiple proposition regions\(^3\). These solutions could then each be stored as a directed edge in the abstracted Kripke structure, so that the details of the trajectory could be ignored while model checking, thereby greatly improving efficiency. The primary benefit, however, is being able to determine a sequence of trajectories that together can be used to satisfy a temporal logic specification.

This section will use the same concept of an abstracted Kripke structure to construct solution trajectories for a quadrotor system. Deviating from the method using the Monte-Carlo approach with SST*, we instead use the \textit{kinoFMT} planner with path smoothing to quickly determine a high-quality trajectory satisfying a given \(\mu\)-calculus specification, \(\Phi\).

4.3.1 Algorithm

The meta-algorithm we propose is similar to \textit{KinoSpecPlan} (Algorithm 8). Differences arise due to the fact that \textit{kinoFMT} is not an incremental algorithm; instead, samples are drawn during the offline phase so that the \texttt{Cost} look-up table can be precomputed. Another key difference is that, unlike SST*, the use of \textit{kinoFMT} assumes there is a way to locally steer between points, and the smoothing polynomials can be used to exactly traverse desired waypoints. Since it is only possible to reach sampled states with this method, there are a finite number of possible goal states for any given proposition region. We will use this fact to our advantage, reducing some of the uncertainty arising from tracking in the method presented in Chapter 3.

To begin, states are sampled from each of the proposition regions of state space until each such region contains exactly \(m\) samples. We first plan trajectories from the initial state to each of the proposition regions. Then, for each of the proposition regions, repeat this process, planning from each of the \(m\) samples of the region to every other proposition region. In the end, there will be at most \(mn(n-1)\) trajectories between proposition regions, and \(n\) possible trajectories from the initial state. See Figure 4.8 for an illustrative example.

Next, we construct the abstracted Kripke structure (Figure 4.9). A directed edge is added between proposition regions \([\pi_i],[\pi_j]\) only if each of the \(m\) samples in \([\pi_i]\) \(^3\)Recall that a proposition region consists of the set of states \([\pi_i]\), where \(\pi_i \in |\Pi_+(\Phi)|\), as defined in Section 3.3
successfully finds a trajectory to $[\pi_j]$. In this way, we guarantee that no matter which sample we start from in $[\pi_i]$, there will always be a path to $[\pi_j]$.

The idea remains the same whether the proposition regions are known online or offline, although repeating the entire real-time framework can become costly as the number of required OBVP solutions grows. If the proposition regions are known beforehand, the $m$ samples of each region can be added to the existing tree, and all of the necessary OBVP solutions can be found offline. In this way, the online planner need only connect the initial state to the existing graph without having to connect all of the proposition region states as well. However, not all computation can be done offline as obstacles remain unknown. The bottleneck while online is then caused by running \texttt{kinoFMT} up to $n + mn(n-1)$ times. Janson et al. showed in [16] that \texttt{FMT*} has time complexity $O(n \log(n))$, and by the same arguments, \texttt{kinoFMT} has the same computational complexity. Altogether, the time complexity contributed by calls of \texttt{kinoFMT} in this new framework is $O(mn^3 \log(n))$, which can be expensive if there are many positively-appearing propositions in specification $\Phi$. Note that path smoothing is required only for those paths which are required in satisfying the specification.

![Diagram](image)

Figure 4.8: This diagram illustrates the set of solution trajectories found when attempting to reach from every sampled node of each proposition region (and initial state $x_{init}$) to all other proposition regions. We use $m = 2$ samples in this example.
Figure 4.9: Shown here is the abstracted Kripke structure based on the trajectories found in Figure 4.8. Note that an edge is only included when a solution is found from all samples of a given region to another particular region.

It remains to determine whether or not the abstracted Kripke structure satisfies the given deterministic $\mu$-calculus specification. To this end, we once again use the local model checking algorithm from Chapter 3 (Algorithm 7). Once the appropriate set of transitions is determined, path smoothing can be applied. As we have not discussed how to generate smooth closed curves, which arise when any cycles are required in satisfying the specification, it suffices to plan each segment of the plan individually. At this point, the tracking controller from Eq. (4.73) can be used to follow the proposed (possibly infinite) path. The quadrotor travels to the first proposition region, arriving at one of $m$ samples, and since each is guaranteed to have a known trajectory to the next destination, the quadrotor simply tracks the appropriate smooth path, and the process repeats as necessary. Note that this algorithm is reactive, and any newly appearing obstacles may necessitate finding new sets of waypoints to avoid collisions.
Chapter 5

Conclusions

In this thesis, we offered motivation in the form of real-world applications for the study of motion planning and the use of temporal logic. Upon delving into the mathematical background of $\mu$-calculus and outlining the key features and uses of the motion planning algorithms SST* and FMT*, we proposed two frameworks for kinodynamic planning with temporal logic specifications.

The method presented in Chapter 3 is a novel use of SST*, a kinodynamic planning algorithm that avoids reliance on a steering function, and $\mu$-calculus model checking. We proceed to develop the idea of an abstracted Kripke structure to determine satisfaction of deterministic $\mu$-calculus specifications. Notably, construction of the abstracted Kripke structure is performed by creating multiple Kripke structures and merging the most cost-efficient solutions into one structure, and the appropriate trajectories satisfying the given specification are tracked using an LQR feedback control policy. Altogether, this is a general approach for generating trajectories satisfying a deterministic $\mu$-calculus proposition.

Meanwhile, the second approach is tailored to the application of quadrotor motion planning, although in theory a very similar methodology could be applied to any differentially flat system. Upon determining a set of flat output variables, we use double-integrator dynamics as an approximation on which to precompute approximate optimal cost and flight durations on a fixed set of random samples of the configuration space. This information is then stored in a look-up table for use online, where the kinoFMT planning algorithm is used to determine least-cost waypoints along which the quadrotor can travel to reach the goal. Finally, minimum-snap polynomial trajectories are calculated to construct a smooth path, and the result is tracked with feedback controller. Using an abstracted Kripke structure, we can store smooth trajectories between pairs of proposition regions and use a local
model checker to determine the paths necessary to satisfy a given deterministic $\mu$-calculus specification. The result can then be tracked using the proposed controller. One significant advantage of this approach is that planning time is reduced to mere seconds, especially if the proposition regions are known a priori so that much of the time-consuming computation may be done offline.

5.1 Future Work

This research opens many avenues for future work. Investigation into other types of feedback controllers may help to improve tracking when applying the SST method from Chapter 3, especially for nonlinear systems. Tracking also poses a problem for collision avoidance, since despite guaranteeing a collision-free trajectory with an appropriate $\mu$-calculus specification, tracking errors may yet cause collisions. This issue is partially addressed with the real-time framework for quadrotor kinodynamic planning in Chapter 4 which can detect collisions and plan a new trajectory. However, the time complexity for planning grows quadratically with the number of proposition regions, rendering the real-time planning task infeasible for specifications involving many atomic propositions.

Further on the topic of quadrotor kinodynamic planning with temporal logic specifications, we recognize that the proposed method is very conservative. We require that every sample of a proposition region must have a solution to another given region in order to add the corresponding directed edge to the abstracted Kripke structure. This constraint provides the guarantee that a path will always be found, independent of the sample at which the quadrotor arrives. On the other hand, it may be possible to locally steer towards a sample state for which there is a solution, and proceed along the known trajectory from there. This could conceivably be accomplished by prepending the set of waypoints from a sample with a known solution with the state for which a solution was not found. On the other hand, obstacles, significant differences in speed, and a number of other factors introduce difficulties. One further improvement to this planning method lies in developing a method for constructing closed curves when smoothing over the set of paths required to satisfy a given specification. In this way, individual trajectories need not be tracked, and a truly infinite-path solution can be immediately obtained.

Incidentally, Schoellig et al. at the University of Toronto tend to focus on path-following rather than trajectory tracking [11, 35]. This is because trajectory tracking involves maintaining pace with a time-parameterized trajectory, so that any disturbance which cause the system to fall behind requires catching up in a potentially undesirable manner. In path-following, only a geometric path is known, and the system merely follows along the
path at a prescribed velocity. With this method, disturbances merely cause the system to return to the nearest point along the geometric path. Path-following is therefore desirable for ensuring predictable behaviour, and seems to present a possible improvement over the time-parameterized smooth polynomials from Chapter 4.

Lastly, completeness of our method remains to be proven given the multilayer approach, as SST* and kinoFMT guarantee (probabilistic) completeness for each individual Kripke structure, however it is uncertain what can be said of the use of multiple Kripke structures in satisfying a single specification.
References


