Feedback-motion-planning with simulation-based LQR-trees

Feedback-Motion-Planning with Simulation-Based LQR-Trees

Journal Title
XX(X):1–26

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Philipp Reist[1], Pascal V. Preiswerk[1], and Russ Tedrake[2]

Abstract

The paper presents the simulation-based variant of the LQR-Tree feedback-motion-planning approach. The algorithm generates a control policy that stabilizes a nonlinear dynamic system from a bounded set of initial conditions to a goal. This policy is represented by a tree of feedback-stabilized trajectories. The algorithm explores the bounded set with random state samples and, where needed, adds new trajectories to the tree using motion planning. Simultaneously, the algorithm approximates the funnel of a trajectory, which is the set of states that can be stabilized to the goal by the trajectory's feedback policy. Generating a control policy that stabilizes the bounded set to the goal is equivalent to adding trajectories to the tree until their funnels cover the set. In previous work, funnels are approximated with sums-of-squares verification. Here, funnels are approximated by sampling and falsification by simulation, which allows the application to a broader range of systems and a straightforward enforcement of input and state constraints. A theoretical analysis shows that in the long run, the algorithm tends to improve the coverage of the bounded set as well as the funnel approximations. Focusing on the practical application of the method, a detailed example implementation is given that is used to generate policies for two example systems. Simulation results support the theoretical findings, while experiments demonstrate the algorithm's state-constraints capability, and applicability to highly-dynamic systems.

Keywords

Feedback motion planning, random sampling, feedback policy, nonlinear dynamic system, trajectory library

1 Introduction

The proposed algorithm generates a feedback policy that stabilizes a nonlinear dynamic system (for example, a running or flying robot) from a wide range of initial conditions to a goal. While standard linear policies can stabilize initial conditions near a goal (or a nominal trajectory), they usually fail for initial conditions "far" from the goal, where the system's nonlinear dynamics, and input- and state constraints must be considered by the policy. In order to address this issue, the simulation-based LQR-Tree algorithm, a variant of the algorithm introduced in Tedrake (2009), generates a policy that consists of a tree of feedback-stabilized trajectories that lead to the goal (the trajectories are stabilized with time-varying linear quadratic regulator (LQR) policies, hence the name).

The algorithm combines concepts from randomized motion planning and feedback control: The bounded set of initial conditions that should be stabilized to the goal is explored using random state samples, and feedback-stabilized trajectories are added to the tree-policy where needed. Key to the algorithm is the approximation of the set of states that can be stabilized to the goal by the feedback policy of a trajectory. This set is called the *funnel* of a trajectory, inspired by Mason (1985) and Burridge et al. (1998). Generating a policy that stabilizes the bounded set to the goal is equivalent to covering the set with funnels of trajectories leading to the goal. When the policy is applied, the tree is queried to find the trajectory whose funnel contains the initial state of the system; then, by definition of a funnel, the system reaches the goal when the trajectory's feedback-policy is applied. The assignment of initial conditions to trajectory-policies is one of the funnels' key purposes.

[1]Institute for Dynamic Systems and Control, ETH Zurich, Switzerland
[2]Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, Cambridge, MA, USA

Corresponding author:
Philipp Reist, Institute for Dynamic Systems and Control, ETH Zurich, Sonneggstrasse 3, 8092 Zurich, Switzerland.
Email: pr@ethz.ch

In previous work (Tedrake 2009; Tedrake et al. 2010), the funnels are approximated by verifying invariant sets using Lyapunov function candidates and sums-of-squares (SoS) programming (Parrilo 2003). Here, we propose an alternative method that, instead of a formal verification, approximates the funnels with a falsification mechanism that uses sampling and simulation: Each trajectory has an associated funnel hypothesis; if a random state sample is within the funnel hypothesis of a trajectory, we simulate the system with the sample as initial condition, while applying the trajectory's feedback policy. If the simulation does not reach the goal, the funnel hypothesis is falsified and is shrunk to exclude the sample. With more samples, the hypothesis shrinks to a tighter approximation to the stabilizable set of the trajectory.

The falsification mechanism renders the funnel approximations nonconservative, which is a major difference to previous work, where the conservative approximations provide sufficient conditions for stabilizability (Tedrake et al. 2010). The stability guarantees of the formal verification are traded-off for: 1) The applicability to a broader class of systems: Simulation-based tree-policies can be generated for (almost) any system that can be simulated on a computer. For example, the approach can handle feedback policies that are not a straightforward function of time and state, such as model-predictive control (MPC) policies (Garcia et al. 1989). 2) The implementation of the simulation-based approach is less involved compared to the SoS method. For example, it is straightforward to enforce input and state constraints in a simulation, and the system dynamics do not have to be approximated by polynomials, as typically performed for SoS-verification (Tedrake et al. 2010; Majumdar et al. 2013).

An analysis of the algorithm's behavior as the number of iterations tends to infinity shows that coverage guarantees similar to probabilistic feedback coverage as defined in Tedrake et al. (2010) can be obtained. However, while the analysis shows that in the long run, the algorithm tends to improve the generated policies, there are no guarantees for finite iterations. Therefore, we introduce a straightforward, statistical method to assess generated policies in practice.

The second part of the paper focuses on the practical application of the algorithm. We provide a detailed example implementation, available in Extension 1, and discuss design considerations. The implementation is used to generate tree-policies for two example nonlinear systems: A simple pendulum and a cart-pole system. The cart-pole features both state- and input-constraints and is an example of a system where the original algorithm using SoS-verification is not straightforward to apply.

The algorithm can be applied to control systems that require high-bandwidth controllers, since a policy is generated offline and is then used as a lookup-table. This is demonstrated in experiments with a laboratory cart-pole setup.

1.1 Structure and Focus of the Paper

The algorithm is presented in a top-to-bottom approach: After a review of related work and some preliminaries, the key ideas of the LQR-Tree algorithm are reviewed on a conceptual level in Section 3. In the following Section 4, we introduce the simulation-based version and its key mechanism, the funnel falsification. In the same section, we discuss the theoretical properties of the algorithm. In Section 5, we present a practice-oriented, detailed example implementation of the algorithm. Finally, in Section 6, the implementation is used to generate tree-policies for the simple pendulum and the cart-pole, which are evaluated in simulation and experiments.

The paper should allow the reader to implement the algorithm in practice for experimental robotic systems. Feedback policies are usually implemented on computers and are therefore executed in discrete time. For this reason, and because it makes the presentation of some of the concepts more straightforward, the algorithm description and example implementation are in discrete time, such that the resulting policies can be directly deployed to experimental systems. The time-discretization steps required for generating policies for continuous-time systems are discussed in Section 5.

1.2 Related Work

This paper is an extension to preliminary simulation results presented in Reist and Tedrake (2010), providing the following additional contributions: Experimental results with a cart-pole system; a systematic termination condition and a statistical method to assess the quality of the generated policies; a discussion of theoretical, asymptotic properties; and a detailed example implementation of the algorithm.

The simulation-based LQR-Tree algorithm is a variant of the LQR-Tree algorithm introduced in Tedrake (2009), where the funnels of trajectories are approximated with sums-of-squares (SoS) programming. Recent advances in funnel verification with SoS include the verification of funnels around trajectories of an experimental Acrobot system (Majumdar et al. 2013),

and verification of funnels of limit cycles of walking robots (Manchester et al. 2011). The LQR-Tree algorithm has also been applied to controlling a fixed-wing glider to perch on a string (Moore et al. 2014). Related, an approach to combine simulations with SoS-programming for finding a Lyapunov function that verifies a region of attraction of stable system equilibria is presented in Topcu et al. (2008). Further related is the approach in Gillula et al. (2014) that computes the viability-kernel, i.e. the set of states that can be kept within state constraints by constrained control inputs, for high-dimensional, sampled-data LTI systems: An inner approximation to the kernel is built from states on the kernel's boundary that are found in a randomized approach and point-wise viability-tests.

The LQR-Tree algorithm is inspired by randomized motion planning, such as probabilistic road maps (PRMs) (Kavraki et al. 1996) and rapidly-exploring randomized trees (RRTs) (LaValle 2006), which have proven to be effective in challenging environments that are high-dimensional, nonlinear, and nonconvex (Kuffner et al. 2001; Frazzoli et al. 2002; Plaku et al. 2005; Shkolnik and Tedrake 2009). The algorithms have further been extended to return asymptotically optimal paths (Karaman and Frazzoli 2011) by continuously improving the generated graphs and trees.

An LQR-Tree policy is related to a PRM, which is an undirected graph where the nodes are robot configurations (i.e. system states) and the edges are paths that connect the configurations. The PRM is constructed by randomly drawing configurations from the free space of an environment with obstacles, with the goal of sparsely covering the free space with a connected graph, which is then used to efficiently plan a path between two robot configurations at run-time: If both configurations can be connected to two graph nodes with a fast, local motion planner, a graph search provides a path between the two nodes, and a full path is found. The tree of feedback-stabilized trajectories in simulation-based LQR-Tree policies serves a similar purpose, as it provides the basis for obtaining a control policy that stabilizes an initial condition to a goal, which implicitly generates a motion plan from the initial condition to the goal. Since the goal is fixed, however, the tree-policies are single-query. Another difference to PRMs is that the samples used to generate an LQR-Tree are not equilibria, as typically used in PRMs (Kavraki et al. 1996; Agha-mohammadi et al. 2014).

A recent, related approach is presented in Levine and Koltun (2013) and Mordatch and Todorov (2014), where open-loop trajectories obtained with motion-planning are used together with linear feedback policies stabilizing the trajectories to learn a global feedback policy represented by a neural network. The optimization used for motion-planning is adapted such that the generated trajectories can be represented by the neural network, and the policy is obtained in an iterative procedure that alternates between motion-planning and policy-learning. A possible issue with this approach is that the generalization of the resulting policy is not systematically tested, which may cause the policy to perform poorly for states that are "far" from the trajectories used for policy-learning. In contrast, the algorithm proposed here constantly verifies the generalization of trajectory-policies, i.e. the funnels, through sampling and simulation (or rigorously through sums-of-squares verification in the original version (Tedrake et al. 2010)). Another advantage is that the choice of the motion-planning tool, the design of the trajectory-policies, and the generalization mechanism are decoupled. Furthermore, the global tree-policy does not have to be parametrized a priori (e.g. by choosing the number of hidden layers), since the algorithm determines the required policy-complexity at runtime, adding trajectories wherever needed for coverage. The disadvantage is that the memory-requirement for the tree-policy cannot be fixed a priori, and may be higher than using function-approximation.

Adding trajectories only where needed is comparable to variable-resolution discretization, e.g. as in Munos and Moore (2002), which addresses the curse of dimensionality encountered when applying standard uniform state-space-discretization to dynamic programming (Bertsekas 2005) or reinforcement-learning problems (Sutton and Barto 1998). Indeed, we find that the tree-policies generated for the example-systems in this paper contain a low number of nodes (i.e. discretized states in the trajectories) compared to a corresponding uniform state-space discretization, see the discussion in Section 6.3. Another benefit of representing feedback policies with trajectories instead of a state-space discretization is that interpolation issues are avoided. This is also highlighted in Stolle and Atkeson (2010), where the authors propose, similar to LQR-Trees, building a library of trajectories leading to a goal state to represent a control policy: Given the current system state, the nominal control input of the closest state in a trajectory in the library is executed, where closeness is measured using a weighted Euclidean norm. The library is initialized with a single trajectory, and more trajectories are added to the library when failures are observed in experiments,

or simulations with process noise. Trajectory-library policies are applied to a simulated biped walking robot in Liu et al. (2013), where, in addition to the nominal control input, a linear feedback input is applied, similar to the trajectory-stabilizers used here. Trajectories are added when the robot stumbles after a push, using the robot's state immediately after the push as an initial condition for motion planning. The additional trajectories then increase the robustness of the policy against even larger pushes.

The estimation of funnels of trajectories can be used in high-level motion planning. One of the first approaches is introduced in Burridge et al. (1998), where the authors present a cascaded control design for a robotic arm that juggles balls in the presence of obstacles. The basic idea behind the approach is that each controller in the cascade stabilizes a set of states to a goal set (picture a funnel). This goal set is contained in the funnel of a subsequent controller, and the controllers are switched once the system state reaches the funnel of the next controller. Motion planning is then reduced to stringing together a set of policies and their funnels, such that the last funnel ends in the goal region. A related, randomized feedback-motion-planning approach for systematically building controller concatenations to move kinematic robots through environments with obstacles is presented in Yang and LaValle (2004).

In Gillula et al. (2011), controller concatenation is used to achieve a quadcopter backflip maneuver. Since the dynamic-game approach Gillula et al. (2011) use to calculate the regions of attraction of the individual controllers can take disturbances and model uncertainty into account, the resulting controller sequence also provides safety guarantees. Another systematic approach for the concatenation of controllers has been proposed in Colledanchise and Ogren (2014) based on ideas from Burridge et al. (1998) and the concept of Behavior Trees in AI design for computer games (Isla 2005). A challenge in the composition of controllers is to characterize their regions of attraction, which are key to switching between controllers. The region of attraction of a tree-policy is the union of its funnels, which could be used, in addition to being a feedback-law, in the formal construction of robot behaviors by concatenating feedback policies.

2 Preliminaries

First, we introduce some notation and definitions underlying the algorithm description.

2.1 System Dynamics

We consider systems with discrete-time, time-invariant, nonlinear dynamics

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k). \tag{1}$$

At time k, the state of the system is $\mathbf{x}_k \in \mathbb{R}^{\mathrm{n}}$, and the control input is $\mathbf{u}_k$, which is an element of a bounded, open set of admissible inputs: $\mathbf{u}_k \in \mathcal{U} \subset \mathbb{R}^{\mathrm{m}}$, where $\subset$ denotes a proper subset. State constraints are described by the bounded, open set of admissible states $\mathcal{X} \subset \mathbb{R}^{\mathrm{n}}$.

2.2 Policies and Solutions

Next, we define a few concepts related to the control and evolution of the system:

2.2.1 Policy: Let $\boldsymbol{\pi}_k(\mathbf{x})$ be a policy that maps a given time k and state $\mathbf{x}$ to an admissible control input $\mathbf{u} \in \mathcal{U}$. A special class of policies are finite, open-loop input sequences, which are denoted by $\{\bar{\mathbf{u}}_k\}_N := \{\bar{\mathbf{u}}_0, \bar{\mathbf{u}}_1, \ldots, \bar{\mathbf{u}}_{N-1}\}$, where $\bar{\mathbf{u}}_k \in \mathcal{U}$, $\forall\, k$. Note that input constraints are satisfied by definition.

2.2.2 Solutions: Given the state of the system at time n, $\mathbf{x}_n$, and policy $\boldsymbol{\pi}_k(\mathbf{x})$, we can calculate the state at time l, $\mathbf{x}_l$, by recursion of the dynamics (1). We denote the solution with

$$\mathbf{x}_l =: \boldsymbol{\phi}^{l-n}(\boldsymbol{\pi}, \mathbf{x}_n). \tag{2}$$

2.2.3 Open-Loop Trajectories: An open-loop, or nominal system trajectory is the pair $\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N$, where $\{\bar{\mathbf{x}}_k\}_N := \{\bar{\mathbf{x}}_0, \bar{\mathbf{x}}_1, \ldots, \bar{\mathbf{x}}_N\}$ is the state trajectory that results from applying $\{\bar{\mathbf{u}}_k\}_N$ to the system with initial state $\bar{\mathbf{x}}_0$: $\{\bar{\mathbf{x}}_k\}_N = \{\bar{\mathbf{x}}_0, \boldsymbol{\phi}^1(\{\bar{\mathbf{u}}_k\}_N, \bar{\mathbf{x}}_0), \ldots, \boldsymbol{\phi}^N(\{\bar{\mathbf{u}}_k\}_N, \bar{\mathbf{x}}_0)\}$.

2.3 State Sets

We further define the following sets of states:

2.3.1 Goal Set: $\mathcal{G} \subseteq \mathcal{X}$, where $\subseteq$ denotes that $\mathcal{G}$ is a subset of $\mathcal{X}$, but may be equal to $\mathcal{X}$. The open set $\mathcal{G}$ contains the states to which the system state should be stabilized to by the generated feedback policy. This set can be user-defined, e.g. as in Moore et al. (2014), or can be the approximated basin of attraction of a feedback-stabilized goal state as in the examples shown in Section 6.

2.3.2 Stabilizable Set: $\mathcal{S} \subseteq \mathcal{X}$: The set of states $\mathbf{x}$ that can be driven to the goal set $\mathcal{G}$ in finite time without violating state constraints. If $\mathbf{x} \in \mathcal{S}$, there exists a finite open-loop trajectory $\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N$, where $\bar{\mathbf{x}}_0 = \mathbf{x}$, such that the final state is in the goal set, $\bar{\mathbf{x}}_N \in \mathcal{G}$,

and the state trajectory does not violate any state constraints: $\bar{\mathbf{x}}_k \in \mathcal{X}$, $\forall k$. In the following, we say that an initial condition $\mathbf{x}$ is "stabilized to the goal set" by a feedback policy if the closed-loop trajectory arrives in the goal set in finite time without violating any state constraints.

2.3.3 Design Set: $\mathcal{D} \subseteq \mathcal{X}$: The user-defined, open set of states $\mathbf{x}$ for which the algorithm should generate a feedback policy that stabilizes $\mathbf{x}$ to the goal set $\mathcal{G}$ if $\mathbf{x} \in \mathcal{D} \cap \mathcal{S}$. Therefore, the set of states to be stabilized to the goal is $\mathcal{S}_\mathcal{D} := \mathcal{S} \cap \mathcal{D}$.

In practice, a user defines the set of admissible states and inputs $\mathcal{X}$, $\mathcal{U}$ based on physical limitations of the underlying mechanical system, and the design and goal sets $\mathcal{D}$, $\mathcal{G}$ based on the task to be achieved, e.g. perching of a glider on a string from a set of initial states produced by a launching mechanism (Moore et al. 2014). The stabilizable sets $\mathcal{S}$ and $\mathcal{S}_\mathcal{D}$ are typically unknown and not straightforward to determine.

2.4 Motion Planning, Trajectory Stabilization, and Funnels

The algorithm combines motion planning and feedback control to generate control policies. The simulation-based variant of the LQR-Tree algorithm allows a modular implementation using different, existing techniques from the fields of motion-planning and feedback-control research. For example, we use a direct optimal control method (Betts 2010) for motion planning and time-varying LQR-feedback policies to stabilize nominal trajectories. However, the algorithm could, for example, also be implemented with an RRT-based motion-planning method (LaValle 2006) and MPC trajectory stabilizers (Garcia et al. 1989). Due to this modularity, we use generic motion-planning and feedback-control modules in the algorithm description.

2.4.1 Motion-Planning Module: Given a state in the design set, $\mathbf{x} \in \mathcal{D}$, the module attempts to find an open-loop trajectory $\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N$, where $\bar{\mathbf{x}}_0 = \mathbf{x}$, such that the final state is in the goal set, $\bar{\mathbf{x}}_N \in \mathcal{G}$, and the state trajectory does not violate any state constraints: $\bar{\mathbf{x}}_k \in \mathcal{X}$, $\forall k$. We assume that the motion planner finds a valid system trajectory if the state is in the stabilizable set, $\mathbf{x} \in \mathcal{S}_\mathcal{D}$, and fails if $\mathbf{x} \notin \mathcal{S}_\mathcal{D}$. *Remark:* Whether this assumption is valid in practice is not straightforward to determine and depends on the specific motion-planner and system dynamics. We further discuss this assumption for the example implementation in Section 5.4.2.

2.4.2 Feedback-Control Module: Let $\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N$ be a finite open-loop trajectory, as produced by the motion-planning module. Then, we assume that the feedback module produces a (possibly time-varying) policy $\boldsymbol{\pi}_k(\mathbf{x})$ that stabilizes a neighborhood of states around the state trajectory to the goal set $\mathcal{G}$ without violating any state constraints: Let

$$\mathcal{B}(\bar{\mathbf{x}}_n, \epsilon_n) := \{\mathbf{x} : d(\mathbf{x}, \bar{\mathbf{x}}_n) < \epsilon_n\} \tag{3}$$

be an open ball of radius $\epsilon_n > 0$ that is centered at the state $\bar{\mathbf{x}}_n$, where $d(\cdot,\cdot)$ is a distance metric on $\mathbb{R}^n$. Then, for each $\bar{\mathbf{x}}_n$ in the trajectory, we assume there exists an open ball that the policy $\boldsymbol{\pi}_k(\mathbf{x})$ stabilizes to the goal set $\mathcal{G}$ in $N-n$ steps, without violating any state constraints:

$$\begin{aligned} &\forall n \in \{0,\dots,N-1\}, \exists \epsilon_n > 0 \\ &\quad \text{s.t. } \mathbf{x}_n \in \mathcal{B}(\bar{\mathbf{x}}_n, \epsilon_n) \\ &\text{implies } \boldsymbol{\phi}^{N-n}(\boldsymbol{\pi}, \mathbf{x}_n) \in \mathcal{G}, \\ &\quad \text{and } \boldsymbol{\phi}^{k-n}(\boldsymbol{\pi}, \mathbf{x}_n) \in \mathcal{X}, \ \forall k \in \{n,\dots,N-1\}. \end{aligned} \tag{4}$$

A feedback-stabilized trajectory is denoted by the sequence $\mathcal{J} = \{\mathcal{N}_0, \dots, \mathcal{N}_{N-1}\}$, where the elements $\mathcal{N}_k := \{\boldsymbol{\pi}_k, \bar{\mathbf{u}}_k, \bar{\mathbf{x}}_k\}$ are called *nodes* and contain the policy, and nominal state and input at time k. Note that since the final state of the trajectory is in the goal $\mathcal{G}$, it does not need a corresponding node.

2.4.3 Funnels: Finally, we define the set $\mathcal{S}_\mathcal{J} \subseteq \mathcal{S}$ as the set of states around a feedback-stabilized trajectory $\mathcal{J}$ that can be stabilized to the goal set $\mathcal{G}$ by its feedback policy $\boldsymbol{\pi}_k(\mathbf{x})$ without violating any state constraints. We call $\mathcal{S}_\mathcal{J}$ the *funnel* of the trajectory, inspired by Mason (1985) and Burridge et al. (1998).

2.4.4 Remarks about the Feedback-Control Module: A trivial policy that may satisfy the "stabilizable balls" property (4) of the feedback-control module is the open-loop input sequence $\{\bar{\mathbf{u}}_k\}_N$ itself: For example, if the dynamics $\mathbf{f}$ are locally Lipschitz, we can, for a given final trajectory state $\bar{\mathbf{x}}_N$ in the open goal set $\mathcal{G}$, find an open ball centered at $\bar{\mathbf{x}}_{N-1} \in \mathcal{X}$, in which all states are stabilized to the goal: $\exists \epsilon_{N-1} > 0 : \mathbf{x} \in \mathcal{B}(\bar{\mathbf{x}}_{N-1}, \epsilon_{N-1}) \Rightarrow \mathbf{f}(\mathbf{x}, \bar{\mathbf{u}}_{N-1}) \in \mathcal{G}$. Then, given ϵ_{N-1} for the ball around $\bar{\mathbf{x}}_{N-1}$, we can proceed analogously to find an open ball with radius ϵ_{N-2} around $\bar{\mathbf{x}}_{N-2}$, and so on.

The goal of the algorithm is to generate a policy by covering the stabilizable set $\mathcal{S}_\mathcal{D}$ with the funnels of trajectories. Therefore, even though an open-loop input may be sufficient for stabilizing a neighborhood of a trajectory to the goal, one would implement more advanced feedback policies in practice: 1) For "long" trajectories and typically unstable system dynamics, the stabilizable balls and, analogously, the funnels are "small" using an open-loop policy. A more

advanced feedback policy enlarges the funnels around the trajectories, which improves the "sparsity" of the tree-policy since fewer trajectories are needed for coverage of $\mathcal{S}_\mathcal{D}$. 2) Stabilizing a system using open-loop trajectories only works in theory. The underlying physical system has unmodeled dynamics that are likely to cause the open-loop policies to fail in practice. Stabilizing the open-loop trajectories using feedback helps to compensate for these unmodeled dynamics.

3 Conceptual LQR-Tree Algorithm

The key idea, introduced in Tedrake (2009), is that the algorithm generates a tree of feedback-stabilized trajectories $\mathcal{T} := \{\mathcal{J}_1, \mathcal{J}_2, \ldots\}$ that cover the set of states $\mathcal{S}_\mathcal{D}$ that are to be stabilized to the goal $\mathcal{G}$ with the union of their funnels $\mathcal{S}_{\mathcal{J}_j}$. This set of trajectories represents the tree-policy that stabilizes states in $\mathcal{S}_\mathcal{D}$ to the goal $\mathcal{G}$: Given a state $\mathbf{x} \in \mathcal{S}_\mathcal{D}$, we apply the feedback policy of the trajectory $\mathcal{J}_j \in \mathcal{T}$ whose funnel $\mathcal{S}_{\mathcal{J}_j}$ contains $\mathbf{x}$.

The tree is generated iteratively with a randomized sampling approach: In each iteration of the algorithm, a random sample $\mathbf{x}_\mathrm{S}$ is drawn i.i.d. from $\mathcal{D}$ with a user-defined probability density function that is positive on $\mathcal{D}$. If there is a trajectory in the current tree whose feedback policy can stabilize $\mathbf{x}_\mathrm{S}$ to the goal $\mathcal{G}$, i.e. $\mathbf{x}_\mathrm{S} \in \mathcal{S}_\mathcal{J}$ of that trajectory, the algorithm proceeds to the next iteration. If there is no such trajectory, the motion-planner module attempts to find a trajectory from $\mathbf{x}_\mathrm{S}$ to the goal $\mathcal{G}$. If motion-planning is successful, a feedback policy is generated by the feedback-control module, and the new trajectory is added to the tree. If motion-planning fails, $\mathbf{x}_\mathrm{S}$ is not in the stabilizable set, $\mathbf{x}_\mathrm{S} \notin \mathcal{S}_\mathcal{D}$, and the algorithm proceeds to the next iteration. These steps are iterated until the set $\mathcal{S}_\mathcal{D}$ is covered by the funnels of the trajectories in the tree, i.e. until the tree-policy can stabilize all states in $\mathcal{S}_\mathcal{D}$ to the goal $\mathcal{G}$. This iterative procedure is outlined in pseudo-code in Algorithm 1, and is illustrated in a sketch in Fig. 1. The pseudo-code uses descriptive function names that correspond to steps described above, but that are not specifically defined.

Finally, note that we defined the motion-planning module to generate trajectories from the sample to the goal, which is motivated by the strategy we use to initialize the motion-planner in the example implementation, see Section 5.4.1. This is another variation on the original algorithm introduced in Tedrake (2009), where the motion-planner attempts to connect a failed sample to states in existing trajectories in the tree. The following concepts and the simulation-based algorithm could be adapted straightforwardly to the original motion-planning strategy.

Algorithm 1 Conceptual LQR-Tree Algorithm

$\mathcal{T} \leftarrow$ `empty set` {Initialize the tree}
while `isNotCovered`$(\mathcal{S}_\mathcal{D}, \mathcal{T})$ **do**
  $\mathbf{x}_\mathrm{S} \leftarrow$ `getRandomSample`$(\mathcal{D})$
  if `isInAnyFunnel`$(\mathbf{x}_\mathrm{S}, \mathcal{T})$ **then**
    `continue` {$\mathbf{x}_\mathrm{S}$ is already in a funnel}
  else
    $\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N \leftarrow$ `motionPlan`$(\mathbf{x}_\mathrm{S})$ {Attempt to plan an open-loop trajectory from $\mathbf{x}_\mathrm{S}$ to the goal $\mathcal{G}$}
    if `motionPlanSuccessful` **then**
      $\boldsymbol{\pi} \leftarrow$ `getFeedbackPolicy`$(\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N)$
      $\mathcal{T} \leftarrow$ `addTrajectory`$(\mathcal{T}, \boldsymbol{\pi}, \{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N)$
    else
      `continue` {$\mathbf{x}_\mathrm{S}$ is not in $\mathcal{S}_\mathcal{D}$}
    end if
  end if
end while

3.1 Probabilistic Feedback Coverage

We show in the Appendix B.2 that the conceptual LQR-Tree algorithm achieves *probabilistic feedback coverage* of $\mathcal{S}_\mathcal{D}$ as defined in Tedrake et al. (2010). The property implies that, as the number of algorithm iterations tends to infinity, the tree-policy is able to stabilize all states in $\mathcal{S}_\mathcal{D}$ to the goal set $\mathcal{G}$, except possibly a set of states with Lebesgue measure zero. Since in the original algorithm (Tedrake et al. 2010), provably conservative approximations to the funnels are obtained, we use an analogous proof to Sec. 6 in Tedrake et al. (2010) to show the coverage property for the conceptual algorithm. We restate the proof for completeness, and since it inspired the proofs for the simulation-based algorithm.

4 Simulation-Based Algorithm

Some of the steps in the conceptual description of the algorithm are not straightforward to implement. For example, the set $\mathcal{S}_\mathcal{D}$ is usually unknown and, therefore, it is not straightforward to determine whether $\mathcal{S}_\mathcal{D}$ is covered by the funnels of the tree. Practical implementation details are discussed throughout the following. First, we introduce the simulation-based variant of the algorithm and its key mechanism: The approximation of funnels using simulation and falsification. Then, in Section 5, we present a detailed, practical example implementation of the algorithm.

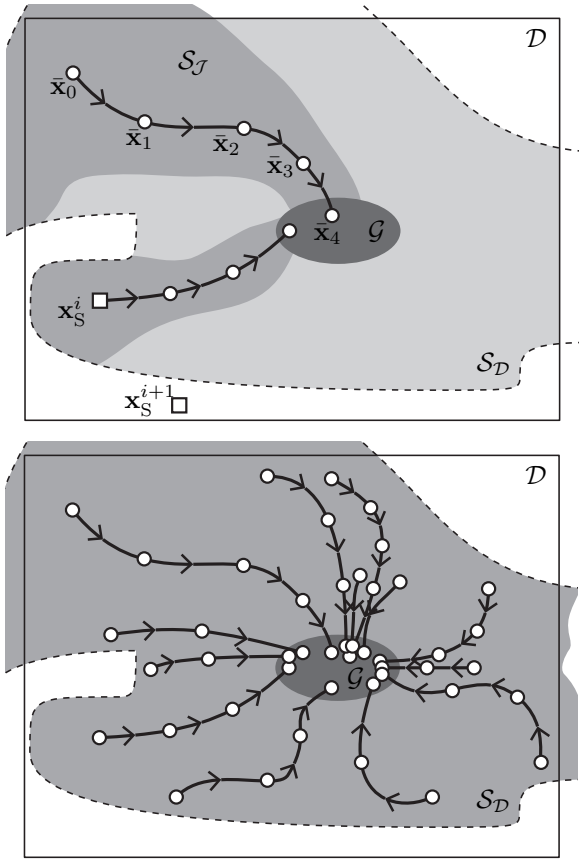

Figure 1. Sketch of the algorithm. *Top:* The tree $\mathcal{T}$ consists of a single trajectory $\mathcal{J}$ with nominal states $\{\bar{\mathbf{x}}_0, \dots, \bar{\mathbf{x}}_4\}$. The funnel $\mathcal{S}_\mathcal{J}$ of $\mathcal{J}$ is shown in medium dark gray. The intersection of the stabilizable set $\mathcal{S}$ (dashed outline) with the design set $\mathcal{D}$ (box), $\mathcal{S}_\mathcal{D} = \mathcal{S} \cap \mathcal{D}$, is shown in light gray. The random sample $\mathbf{x}_\mathrm{S}^i$ drawn at iteration i is not in $\mathcal{S}_\mathcal{J}$ and the algorithm adds a trajectory to the tree that connects $\mathbf{x}_\mathrm{S}^i$ to the goal set $\mathcal{G}$ (dark gray). The next random sample $\mathbf{x}_\mathrm{S}^{i+1}$ is not in $\mathcal{S}_\mathcal{D}$, causing the motion-planner to fail, and the algorithm proceeds to the next iteration. *Bottom:* The algorithm terminates after enough trajectories were added to cover $\mathcal{S}_\mathcal{D}$ with their funnels.

4.1 Funnel Approximation with Simulation and Falsification

Key to the implementation of the LQR-Tree algorithm are the funnels of the trajectories in the tree. The funnel $\mathcal{S}_\mathcal{J}$ of the trajectory $\mathcal{J}$ is the set of states that can be stabilized to the goal $\mathcal{G}$ by its feedback policy $\boldsymbol{\pi}_k(\mathbf{x})$ without violating state constraints. Funnels are, in general, not straightforward to estimate as they depend on the system dynamics, the feedback policy, the state and input constraints, and the goal set. In the original algorithm (Tedrake et al. 2010), funnels are approximated using a formal approach where Lyapunov function candidates are verified with sums-of-squares programming (Parrilo 2003). Here, we present an alternative method: The approximation of funnels using simulation and falsification.

The method is straightforward: For each trajectory in the tree, keep track of a *funnel hypothesis*, which is a parametrized set of states. When a sample $\mathbf{x}_\mathrm{S}$ is within the hypothesis, simulate the sample with the policy of the trajectory. If the simulation fails because either the system state does not reach the goal set $\mathcal{G}$ or state constraints are violated, adapt the funnel parameters to shrink the hypothesis to exclude $\mathbf{x}_\mathrm{S}$. With growing numbers of samples and simulations, the funnel hypotheses in the tree shrink to a nonconservative approximation of the true funnels $\mathcal{S}_\mathcal{J}$.

4.1.1 Node-Policies and Node-Funnels: Before we describe the mechanism, we discuss how a trajectory's feedback policy can be applied to a sample $\mathbf{x}_\mathrm{S}$, and the implications on its funnel. Standard feedback policies that stabilize trajectories are time-varying (for example, we use time-varying LQR policies in the example implementation in Section 5). Let $\boldsymbol{\pi}_k(\mathbf{x})$ be the time-varying policy that stabilizes the nominal trajectory $\{\bar{\mathbf{x}}_k\}_N, \{\bar{\mathbf{u}}_k\}_N$, and let $\mathbf{x}_\mathrm{S} \in \mathcal{D}$ be a random sample that is the initial condition for a simulation with the trajectory-policy. The sample $\mathbf{x}_\mathrm{S}$ has no specific trajectory-index, i.e. time-index assigned with respect to the time-varying trajectory-policy. We can assign different trajectory-indices to $\mathbf{x}_\mathrm{S}$ that lead to different simulations: For example, we can assign $\mathbf{x}_0 := \mathbf{x}_\mathrm{S}$ and obtain the final state $\mathbf{x}_N = \boldsymbol{\phi}^N(\boldsymbol{\pi}, \mathbf{x}_0)$, or we assign $\mathbf{x}_6 := \mathbf{x}_\mathrm{S}$ and obtain $\mathbf{x}'_N = \boldsymbol{\phi}^{N-6}(\boldsymbol{\pi}, \mathbf{x}_6)$. Two different tails of the trajectory-policy are applied to $\mathbf{x}_\mathrm{S}$ that stabilize the nominal trajectory starting at index 0 and at index 6, respectively. In the following, we consider each tail of a trajectory-policy as a distinct policy, and call the policy starting at index n the *node-policy* of node $\mathcal{N}_n$.

Returning to the example, it may be that $\mathbf{x}_N \in \mathcal{G}$, but $\mathbf{x}'_N \notin \mathcal{G}$: The node-policy of $\mathcal{N}_0$ can stabilize $\mathbf{x}_\mathrm{S}$ to the goal, but the node-policy of $\mathcal{N}_6$ cannot. Therefore, the funnel of a trajectory should be time-varying, i.e. node-dependent, and we introduce the *node-funnel*, i.e. the set of states that can be stabilized to the goal by the node-policy of $\mathcal{N}_n$ in trajectory $\mathcal{J}$:

$$\mathcal{S}_\mathcal{N}^n := \{\mathbf{x}_n : \boldsymbol{\phi}^{N-n}(\boldsymbol{\pi}, \mathbf{x}_n) \in \mathcal{G} \text{ and } \boldsymbol{\phi}^{k-n}(\boldsymbol{\pi}, \mathbf{x}_n) \in \mathcal{X},\ \forall k \in \{n, \dots, N-1\}\}. \tag{5}$$

Note that the nodes, policies, node-funnels, and nominal trajectories should all carry a trajectory index j, since they belong to a specific trajectory $\mathcal{J}_j$ in the tree. For readability, however, we

omit trajectory indices wherever possible without introducing ambiguity.

4.1.2 Funnel Hypotheses: Note the similarity between the time-varying funnel definition (5) and the stabilizable balls property of the feedback-control module (4). Any stabilizable ball around the nominal state $\bar{\mathbf{x}}_n$ of node $\mathcal{N}_n$ is a subset of its funnel. Since balls are straightforward to parametrize by their radius, they are a natural (but not the only) choice for the funnel hypotheses used in the simulation-based funnel approximation.

We define the *funnel hypothesis of node* $\mathcal{N}_n$ as an open ball with radius $\hat{\epsilon}_n > 0$, centered at $\bar{\mathbf{x}}_n$

$$\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n) := \{\mathbf{x}_n : d(\mathbf{x}_n, \bar{\mathbf{x}}_n) < \hat{\epsilon}_n\} \tag{6}$$

where $d(\cdot,\cdot)$ is some metric on $\mathbb{R}^n$. The hypothesis to be falsified is that the funnel hypothesis is contained in the node-funnel: $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n) \subseteq \mathcal{S}^n_{\mathcal{N}}$. In other words, the funnel hypothesis claims that all states $\mathbf{x}_n \in \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n)$ can be stabilized to the goal by the policy of node $\mathcal{N}_n$ without violating state constraints.

The funnel hypothesis parameters $\hat{\epsilon}_n$ are key to the generation and application of the tree-policy. Therefore, we redefine the nodes of a trajectory to $\mathcal{N}_n := \{\pi_n, \bar{\mathbf{u}}_n, \bar{\mathbf{x}}_n, \hat{\epsilon}_n\}$. Note that while the policy, and nominal state and input are constant, the funnel parameter is a variable that changes as the algorithm is executed due to the falsification mechanism described below. However, for readability, we omit an additional superscript that indexes the funnel parameter adaptations.

4.1.3 Funnel Hypothesis Test: The funnel approximation mechanism using simulation and falsification is straightforward. Let the sample $\mathbf{x}_S \in \mathcal{D}$ be inside the funnel hypothesis of node $\mathcal{N}_n$ in trajectory $\mathcal{J}$, $\mathbf{x}_S \in \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n)$. We then test if the hypothesis holds for $\mathbf{x}_S$, i.e. we check if $\mathbf{x}_S$ is in the funnel $\mathcal{S}^n_{\mathcal{N}}$ of node $\mathcal{N}_n$. This test is straightforward when $\mathbf{x}_S$ is simulated with the policy of node $\mathcal{N}_n$:

1. Set $\mathbf{x}_n = \mathbf{x}_S$, simulate the system applying the policy of node $\mathcal{N}_n$, and obtain the state trajectory

$$\begin{aligned}\{\mathbf{x}_n = \mathbf{x}_S,\ \mathbf{x}_{n+1} = \boldsymbol{\phi}^1(\boldsymbol{\pi}, \mathbf{x}_S),\\ \ldots,\ \mathbf{x}_N = \boldsymbol{\phi}^{N-n}(\boldsymbol{\pi}, \mathbf{x}_S)\}.\end{aligned} \tag{7}$$

2. Check if the funnel conditions given in (5) hold, i.e. the state trajectory (7) must satisfy

$$\mathbf{x}_N \in \mathcal{G} \ \text{ and } \ \mathbf{x}_k \in \mathcal{X},\ \forall\, k \in \{n, \ldots, N\}. \tag{8}$$

   Recall that input constraints are satisfied by definition by policies. We declare a simulation *successful* if the conditions (8) are satisfied, and *failed* otherwise.

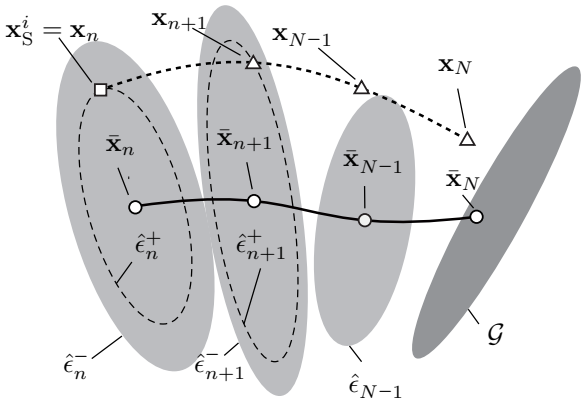

Figure 2. Adjusting funnel hypotheses after the failed simulation of random sample $\mathbf{x}^i_S$ (□) with the policy of node $\mathcal{N}_n$. The final state $\mathbf{x}_N$ of the simulation trajectory (△) failed to reach the goal set $\mathcal{G}$. However, the funnel hypotheses of nodes $\mathcal{N}_n$ and $\mathcal{N}_{n+1}$, described by the funnel parameters $\hat{\epsilon}^-_n$ and $\hat{\epsilon}^-_{n+1}$, claim that a simulation would be successful. Therefore, we set $\hat{\epsilon}^-_n, \hat{\epsilon}^-_{n+1}$ to $\hat{\epsilon}^+_n, \hat{\epsilon}^+_{n+1}$ according to (9), resulting in the dashed hypotheses that do not include $\mathbf{x}^i_S$ and $\mathbf{x}_{n+1}$ anymore. The simulation state $\mathbf{x}_{N-1}$ is not inside the hypothesis of node $\mathcal{N}_{N-1}$, and thus $\hat{\epsilon}_{N-1}$ remains unchanged.

3. If the simulation is successful, do not adjust the funnel hypothesis. If the simulation fails, i.e. the hypothesis is falsified, shrink the funnel hypothesis of node $\mathcal{N}_n$ such that it no longer includes $\mathbf{x}_S$. Furthermore, the trajectory generated in the failed simulation can be used to test the funnel hypotheses of the nodes that follow $\mathcal{N}_n$. The trajectory $\{\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N\}$ is the result of applying the policy of node $\mathcal{N}_{n+1}$ to $\mathbf{x}_{n+1}$. If $\mathbf{x}_{n+1}$ is contained in the funnel hypothesis of node $\mathcal{N}_{n+1}$, $\mathbf{x}_{n+1} \in \hat{\mathcal{B}}(\bar{\mathbf{x}}_{n+1}, \hat{\epsilon}_{n+1})$, this hypothesis is also falsified. Therefore, after a failed simulation, set

$$\hat{\epsilon}_k \leftarrow \min\left(d(\mathbf{x}_k, \bar{\mathbf{x}}_k), \hat{\epsilon}_k\right), \forall\, k \in \{n, \ldots, N-1\}. \tag{9}$$

   The operation $\min(\cdot,\cdot)$ returns the smaller of its two scalar arguments and ensures that a hypothesis is only adjusted if it contains $\mathbf{x}_k$, i.e. the hypotheses can only shrink, and never expand.

This key mechanism of the algorithm is illustrated in Fig. 2. Finally, when a new trajectory is added to the tree, all funnel hypotheses are initialized to cover the whole design set $\mathcal{D}$: $\hat{\epsilon}_k \leftarrow \infty, \forall\, k \in \{0, \ldots, N-1\}$ (the last node in the trajectory is in $\mathcal{G}$). Then, with a growing number of samples and simulations,

the hypotheses are shrunk to a nonconservative approximation of the true node-funnels.

4.2 Application of the Tree-Policy: Query Phase

Before summarizing an iteration of the algorithm, we clarify how the tree-policy $\mathcal{T}^i$ at iteration i is applied to a sample $\mathbf{x}_{\mathrm{S}}^i \in \mathcal{D}$ in the *query phase*. The same query phase is used when the policy is deployed to stabilize initial conditions $\mathbf{x}_{\mathrm{IC}} \in \mathcal{D}$ to the goal $\mathcal{G}$. Therefore, one may read the random sample $\mathbf{x}_{\mathrm{S}}^i$ as initial condition $\mathbf{x}_{\mathrm{IC}}$ in the following.

The idea is to apply the policy of a node $\mathcal{N}_n$ in the tree whose funnel hypothesis $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n)$ contains $\mathbf{x}_{\mathrm{S}}^i$. There may be multiple nodes in the tree whose funnel hypotheses contain $\mathbf{x}_{\mathrm{S}}^i$, which introduces ambiguity in what policy to apply to $\mathbf{x}_{\mathrm{S}}^i$. Therefore, we use the following rule to assign a node-policy to a sample $\mathbf{x}_{\mathrm{S}}^i$: Apply the policy of the node $\mathcal{N}_n$ whose nominal state $\bar{\mathbf{x}}_n$ is closest to $\mathbf{x}_{\mathrm{S}}^i$, measured by the distance metric $d(\cdot,\cdot)$ that defines its funnel hypothesis (6). Specifically, we apply the policy of node $\mathcal{N}^*$ in the tree $\mathcal{T}^i$ according to

$$\begin{aligned} \mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i) = \underset{\mathcal{N}_n \in \mathcal{T}^i}{\operatorname{argmin}} \quad & d(\mathbf{x}_{\mathrm{S}}^i, \bar{\mathbf{x}}_n) \\ \text{subject to: } & \mathbf{x}_{\mathrm{S}}^i \in \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n), \quad \bar{\mathbf{x}}_n, \hat{\epsilon}_n \in \mathcal{N}_n. \end{aligned} \tag{10}$$

Since it is not relevant what specific trajectory the node $\mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i)$ is an element of, we omit the trajectories for clarity and refer to all nodes in the tree with $\mathcal{N}_n \in \mathcal{T}^i$.

4.3 An Iteration of the Algorithm: Testing Samples

An iteration of the algorithm can be summarized as follows:

1. Draw an i.i.d. random sample $\mathbf{x}_{\mathrm{S}}^i$ from $\mathcal{D}$ using a probability density function that is positive on $\mathcal{D}$. We use a uniform distribution in the examples in Section 6; a better option is to use a distribution derived from measured initial conditions of the real system.
2. Find $\mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i)$ according to the minimization (10). If no feasible $\mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i)$ can be found, i.e. if $\mathbf{x}_{\mathrm{S}}^i$ is outside all funnel hypotheses, attempt to add a new trajectory to the tree, and proceed to the next iteration.
3. If there is a feasible $\mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i)$, simulate $\mathbf{x}_{\mathrm{S}}^i$ with the policy of $\mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i)$, and check the success conditions (8). If successful, proceed to the next iteration. If failed, adjust the funnel parameters (9) and go back to Step 2: Since the funnel of $\mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i)$ does not include $\mathbf{x}_{\mathrm{S}}^i$ anymore, there may exist other feasible $\mathcal{N}^*(\mathbf{x}_{\mathrm{S}}^i)$ in the tree that could stabilize $\mathbf{x}_{\mathrm{S}}^i$ to $\mathcal{G}$.

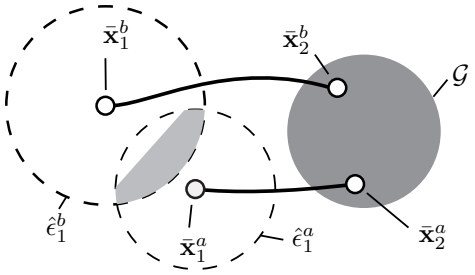

Figure 3. Overlap of funnel hypotheses (dashed circles) and policy assignment. The metric $d(\cdot,\cdot)$ is the Euclidean distance and there are two trajectories a and b. If the funnel hypothesis of node $\mathcal{N}_1^a$ in Trajectory a never shrinks, the gray set of states in the hypothesis of $\mathcal{N}_1^b$ in Trajectory b is never tested since the states are closer to $\bar{\mathbf{x}}_1^a$ than $\bar{\mathbf{x}}_1^b$. In this case, it may be that some states in the gray region cannot be stabilized by the node-policy of $\mathcal{N}_1^b$, but they are never tested because the hypothesis of $\mathcal{N}_1^a$ hides them from the algorithm.

4.3.1 Interpretation of Funnel Hypotheses: The specific steps taken in an iteration of the algorithm have an implication on the funnel hypotheses that is not obvious, but that is useful to point out for the interpretation of the generated tree-policy: First, note that a sample $\mathbf{x}_{\mathrm{S}}$ can be in multiple funnel hypotheses due to possible overlap. Second, note that the algorithm immediately proceeds to the next iteration after the first successful stabilization of the sample $\mathbf{x}_{\mathrm{S}}$. The consequence of both observations is that some subsets of funnel hypotheses may never be tested, and therefore may contain states that cannot be stabilized by the respective node-policy. This is illustrated in Fig. 3.

Better approximations to the actual funnels could be achieved by testing a sample with the policies of *all* nodes whose funnel hypotheses contain the sample, such that no "unstabilizable" subsets of the hypotheses remain undetected. The downsides of this strategy are: 1) A single iteration of the algorithm becomes computationally more expensive, since a sample is typically in many funnel hypotheses (in a test with the simple pendulum system introduced later in Section 6, the runtime increase was about 7 fold); and 2) the algorithm adds more nodes to the tree (in the test with the simple pendulum, the node-increase was about 50%): Assume that all states in the funnel of node $\mathcal{N}_1^b$ in Fig. 3 are stabilizable, except for the gray states. Then, if the gray states are detected and the funnel of $\mathcal{N}_1^b$ shrinks, additional nodes are added to compensate for the lost coverage. Furthermore, the goal of the

algorithm is to generate a tree-policy that stabilizes initial conditions to the goal, and not to approximate the funnels of individual trajectories as well as possible.

These reasons lead to the design decision of proceeding immediately after the first successful stabilization, which implies that: 1) The main purpose of the funnel hypotheses is to speed up an iteration of the algorithm and to assign policies to initial conditions; and 2) the tree-policy is a unit, where all nodes must be considered in the query phase, since their interplay is important.

4.4 Asymptotic Properties

We analyze the theoretical properties of the simulation-based algorithm as the number of iterations and samples tends to infinity. The focus is on two key points: 1) The coverage of the stabilizable set $\mathcal{S}_\mathcal{D}$ by the tree-policy, similar to the probabilistic-feedback-coverage property of the original algorithm (Tedrake et al. 2010); and 2) whether the funnel hypotheses provide correct assignments of initial conditions to stabilizing node-policies. The proofs are based on the assumptions, summarized in Appendix B.1, made in Section 2.4 and Section 4.3 about the motion-planning and feedback modules, and the probability density function used to generate state samples.

The analysis is more involved than for the original algorithm: We need to consider the interactions between the funnel hypotheses, and that the hypotheses are outer approximations to the funnels. For the same reasons, the conclusions that can be drawn from the results are not as strong as for the original algorithm, where conservative inner-approximations to the funnels are obtained by verifying Lyapunov-function-candidates (Tedrake et al. 2010). However, the results still indicate that in the long run, the simulation-based algorithm tends to improve both coverage and node-policy assignments, which is key to generating acceptable policies in practice.

4.4.1 Coverage: We can show that as the number of algorithm iterations tends to infinity, there exists a stabilizing node-policy in the tree for all states in the stabilizable set $\mathcal{S}_\mathcal{D}$, except possibly a set of states with Lebesgue measure zero. A proof that uses a similar approach to the proof of probabilistic feedback coverage in Tedrake et al. (2010) is given in Appendix B.3. Roughly speaking, the uncovered set of states in $\mathcal{S}_\mathcal{D}$ must have measure zero in the limit, because 1) the algorithm can always detect if a state $\mathbf{x}$ is in the uncovered set, since then, either there exists no feasible $\mathcal{N}^*(\mathbf{x})$, or all candidate node policies fail to stabilize $\mathbf{x}$ to the goal $\mathcal{G}$, and, in addition, the motion planner finds a trajectory from $\mathbf{x}$ to $\mathcal{G}$; and 2) the algorithm keeps adding trajectories starting at uncovered states until it cannot find uncovered states anymore.

However, since the algorithm uses outer approximations to the funnels, this result does not allow to conclude that in the limit, all initial conditions (except a set of measure zero) in $\mathcal{S}_\mathcal{D}$ can be stabilized to the goal $\mathcal{G}$. In addition to coverage, we must show that states are correctly assigned to stabilizing node-policies by the funnel hypotheses.

4.4.2 Correct Policy Assignment: In the query phase, the tree-policy relies on the funnel hypotheses to provide a correct assignment of initial conditions $\mathbf{x}_{\text{IC}} \in \mathcal{D}$ to node-policies: If $\mathbf{x}_{\text{IC}}$ is assigned to node $\mathcal{N}_n = \mathcal{N}^*(\mathbf{x}_{\text{IC}})$ according to (10), this should imply that $\mathbf{x}_{\text{IC}}$ can be stabilized to the goal $\mathcal{G}$ by the policy of node $\mathcal{N}_n$. A failure can be due to either 1) a stabilizable $\mathbf{x}_{\text{IC}} \in \mathcal{S}_\mathcal{D}$ is wrongly assigned to $\mathcal{N}_n$, and additional shrinking of the hypothesis of $\mathcal{N}_n$ is required to correct the assignment to another (possibly newly added) node that can stabilize $\mathbf{x}_{\text{IC}}$; or 2) an unstabilizable $\mathbf{x}_{\text{IC}} \notin \mathcal{S}_\mathcal{D}$ is assigned to $\mathcal{N}_n$, and additional shrinking is required to exclude unstabilizable states from the hypothesis of the node.

In Appendix B.4, we show that as the iterations $i \to \infty$, the tree-policy provides a correct assignment of states to a node $\mathcal{N}_n$, i.e. in the limit, $\mathcal{N}^*(\mathbf{x}) = \mathcal{N}_n$ implies that the policy of $\mathcal{N}_n$ can stabilize $\mathbf{x} \in \mathcal{D}$ to the goal $\mathcal{G}$ (again possibly except a set of states with measure zero). The key to this property is that the funnel hypotheses can only shrink, which permanently excludes wrongly assigned states: Roughly speaking, the algorithm keeps shrinking a node's funnel hypothesis after sampling states that are assigned to the node and that fail to stabilize, until it cannot find such wrongly assigned states anymore.

The result implies that in the long run, the algorithm keeps improving the assignment of initial conditions to a node. Note that the result does not imply that all states in the node's funnel hypothesis $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n)$ can be stabilized. It may well be that there are states in the hypothesis that cannot be stabilized, but that are hidden from the algorithm by an overlapping hypothesis of another node, see Fig. 3 and the discussion in Section 4.3.1.

The result further implies that if coverage of $\mathcal{S}_\mathcal{D}$ is achieved by the tree-policy at a finite iteration and therefore a finite number of nodes in the tree, the policy assignments are correct in the limit, and all states in $\mathcal{S}_\mathcal{D}$ can be stabilized to the goal $\mathcal{G}$. However, since funnel hypotheses provide an outer approximation, a correct policy-assignment cannot be guaranteed in case coverage is only achieved as the number of tree nodes

tends to infinity together with the number of iterations: Arbitrarily often, a new node is added to the tree, and this nodes' funnel hypothesis may contain wrongly assigned states (a trivial example is when $\mathcal{S}_\mathcal{D} \subset \mathcal{D}$ is strict, since a node's funnel hypothesis initially covers $\mathcal{D}$). A possible solution is to limit the number of nodes that can be added to the tree; then, in the limit, a correct policy assignment is achieved at the price of the loss of the coverage guarantee.

While the theoretical analysis shows that the algorithm tends to improve the policy in the long run, it does not provide guarantees for tree-policies generated in practice with finite iterations: Both coverage and node assignments are likely to be imperfect. Therefore, we next discuss a heuristic termination condition that we found to produce tree-policies with acceptable coverage and node-assignments, and a method to assess these two key measures of generated policies.

4.5 Termination Condition and Assessment of Policy Quality

We terminate the algorithm in either of two cases: 1) a pre-determined iteration limit is reached, which limits the maximal run time; or 2) a heuristic termination condition is fulfilled. In order to assess the generated tree-policy, we propose a simulation-based, statistical method to estimate two quality measures: 1) An estimate of the coverage of the tree-policy (i.e. the funnels) of the design set $\mathcal{D}$; and 2) an estimate of the likelihood that an initial condition is stabilized to the goal by the tree-policy, given that the initial condition is inside a funnel. These measures can be used to make a decision about whether the generated policy is acceptable, or requires further refinement, i.e. more iterations.

4.5.1 Heuristic Termination Condition: The heuristic we propose is similar to the termination condition presented in Yang and LaValle (2004), and resulted in tree-policies with acceptable coverage and node-assignments for the example systems considered in Section 6: The algorithm terminates after a consecutive sequence of M samples $\mathbf{x}_\mathrm{S}^i \in \mathcal{D}$ does not cause the tree-policy to change: no funnels are adjusted and no new trajectories are added. Therefore, a sample $\mathbf{x}_\mathrm{S}^i$ of the sequence is either successfully stabilized to the goal by the policy of $\mathcal{N}^*(\mathbf{x}_\mathrm{S}^i)$, or is not in any funnel and motion planning fails (i.e. we assume $\mathbf{x}_\mathrm{S}^i \notin \mathcal{S}_\mathcal{D}$). Note that even though the algorithm may attempt to stabilize a sample $\mathbf{x}_\mathrm{S}^i$ with more than one node-policy in an iteration, see Section 4.3, a sample $\mathbf{x}_\mathrm{S}^i$ of the success sequence must be stabilized by the *first* policy $\mathcal{N}^*(\mathbf{x}_\mathrm{S}^i)$ that is applied, otherwise funnels are adjusted.

Since the samples are drawn i.i.d. from $\mathcal{D}$, the probability p_α^i that a sample $\mathbf{x}_\mathrm{S}^i$ does not cause the tree-policy to change at iteration i is the underlying parameter of a (nonstationary) Bernoulli process. Every time the policy changes, p_α^i may change as well, since the funnel hypotheses and therefore node-assignments change. However, if the policy is constant, $p_\alpha^i =: \bar{\mathrm{p}}_\alpha$ is constant as well. We use this observation to derive the parameter M from the probability $\alpha = \bar{\mathrm{p}}_\alpha^\mathrm{M}$ of observing a sequence of M samples that do not cause the policy to change starting at iteration i. Both $0 < \alpha < 1$ and $0 < \bar{\mathrm{p}}_\alpha < 1$ are design parameters. Solving for M and rounding up (ceil), we obtain

$$\mathrm{M} = \mathrm{ceil}\left(\frac{\log(\alpha)}{\log(\bar{\mathrm{p}}_\alpha)}\right). \tag{11}$$

Note that M, and therefore the heuristic, is independent of the state and input dimensions.

As the algorithm runs, it keeps track of the current streak of successful samples and terminates when the "success count" reaches M. Since in the limit as $i \to \infty$, any finite length of consecutive, successful samples can eventually be observed for any positive underlying p_α, the termination condition does not provide guarantees for the quality of the generated policy in terms of p_α. However, we found that in practice, appropriately choosing the design parameters results in policies with acceptable quality measures, which we describe below.

With the termination condition, we discussed the final element of the simulation-based LQR-Tree algorithm. A summary in pseudo-code is given in Algorithm 2, where function and variable names are not explicitly defined, but correspond directly to descriptions in this section.

4.5.2 Statistical Assessment of Generated Tree-Policy: Let $\mathcal{T}$ be the tree-policy after the algorithm terminated. When applying the tree-policy $\mathcal{T}$ to initial conditions $\mathbf{x}_\mathrm{IC}$ from the design set $\mathcal{D}$, we care about: 1) The probability ρ that $\mathbf{x}_\mathrm{IC}$ is in a funnel hypothesis of the tree, i.e. that there exists a node $\mathcal{N}^*(\mathbf{x}_\mathrm{IC}) \in \mathcal{T}$ according to (10); and 2) given that there exists an $\mathcal{N}^*(\mathbf{x}_\mathrm{IC})$, the conditional probability p^* that the feedback policy of $\mathcal{N}^*(\mathbf{x}_\mathrm{IC})$ is able to stabilize $\mathbf{x}_\mathrm{IC}$ to the goal $\mathcal{G}$. We call ρ the *coverage ratio* and p^* the *success rate* of the tree-policy.

The two probabilities can be estimated by testing the tree-policy $\mathcal{T}$ in simulation with a set of N_s i.i.d. random samples $\mathbf{x}_\mathrm{S}^i \in \mathcal{D}$. The probability density function (PDF) used to generate the samples could be the same PDF used to generate the tree-policy, and may be designed to put more weight on frequently occurring initial conditions. For each sample $\mathbf{x}_\mathrm{S}^i$, there are three possible outcomes: 1) There is no feasible

Algorithm 2 Simulation-Based LQR-Tree Algorithm

$\mathcal{T} \leftarrow$ `empty set` {Init. the tree}
M $\leftarrow$ `ceil`$(\log(\alpha)/\log(\bar{\text{p}}_\alpha))$ {Term. heuristic}
`nSuccess`, `i` $\leftarrow 0$ {Init. success and iteration counters}
while `nSuccess` $<$ M and `i` $<$ `maxIterations` **do**
  {Iterate until termination heuristic is fulfilled or maximal number of iterations is reached}
  `i` $\leftarrow$ `i` $+ 1$ {Increment iteration}
  `isStabilized` $\leftarrow$ `false` {Reset stabilized boolean}
  $\mathbf{x}_\text{S} \leftarrow$ `getRandomSample`$(\mathcal{D})$
  while `isInAnyFunnelHypothesis`$(\mathbf{x}_\text{S}, \mathcal{T})$ **do**
    $\mathcal{N}^*(\mathbf{x}_\text{S}) \leftarrow$ `getNodeStar`$(\mathbf{x}_\text{S}, \mathcal{T})$
    `xTraj` $\leftarrow$ `simulateSystem`$(\mathbf{x}_\text{S}, \mathcal{N}^*(\mathbf{x}_\text{S}))$ {Apply policy of $\mathcal{N}^*(\mathbf{x}_\text{S})$, obtain state trajectory `xTraj`}
    `isStabilized` $\leftarrow$ `checkSimulation`$(\text{xTraj}, \mathcal{X}, \mathcal{G})$ {Check `xTraj` according to Eq. (8)}
    if `isStabilized` **then**
      `nSuccess` $\leftarrow$ `nSuccess` $+ 1$
      `break` {Proceed to next sample $\mathbf{x}_\text{S}^{i+1}$}
    else
      $\mathcal{T} \leftarrow$ `adjustFunnels`$(\text{xTraj}, \mathcal{T})$
      `nSuccess` $\leftarrow 0$ {Policy changed, reset `nSuccess`}
    end if
  end while
  if not `isStabilized` **then**
    {The sample $\mathbf{x}_\text{S}$ could not be stabilized, attempt to plan an open-loop trajectory from $\mathbf{x}_\text{S}$ to the goal $\mathcal{G}$}
    $\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N \leftarrow$ `motionPlan`$(\mathbf{x}_\text{S})$
    if `motionPlanSuccessful` **then**
      $\boldsymbol{\pi} \leftarrow$ `getFeedbackPolicy`$(\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N)$
      $\mathcal{J} \leftarrow$ `initTrajectory`$(\boldsymbol{\pi}, \{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N)$ {Set funnel hypothesis of trajectory to cover $\mathcal{D}$: $\hat{\epsilon}_k \leftarrow \infty, \forall\, k$}
      $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{J}$ {add trajectory to tree}
      `nSuccess` $\leftarrow 0$ {Policy changed, reset `nSuccess`}
    else
      {Do nothing; assume $\mathbf{x}_\text{S} \notin \mathcal{S}_\mathcal{D}$}
    end if
  end if
end while

$\mathcal{N}^*(\mathbf{x}_\text{S}^i) \in \mathcal{T}$, i.e. the sample is not in any funnel hypothesis in $\mathcal{T}$; 2) there exists an $\mathcal{N}^*(\mathbf{x}_\text{S}^i)$ and $\mathbf{x}_\text{S}^i$ is successfully stabilized to $\mathcal{G}$, or 3) there exists an $\mathcal{N}^*(\mathbf{x}_\text{S}^i)$ and $\mathbf{x}_\text{S}^i$ fails. During testing, no funnel hypotheses are adjusted and no new trajectories are added to the tree, such that the tree-policy and its underlying parameters p^* and ρ remain unchanged.

Let N_ρ be the number of samples $\mathbf{x}_\text{S}^i$ for which there exists an $\mathcal{N}^*(\mathbf{x}_\text{S}^i)$; and N_p be the number of samples $\mathbf{x}_\text{S}^i$ that are successfully stabilized to $\mathcal{G}$ by the policy of $\mathcal{N}^*(\mathbf{x}_\text{S}^i)$. We calculate the respective estimates $\hat{\rho}$ and $\hat{p}^*$ from

$$\hat{\rho} = \frac{N_\rho}{N_s}, \quad \text{and} \quad \hat{p}^* = \frac{N_p}{N_\rho}. \tag{12}$$

Assuming that when the tree-policy $\mathcal{T}$ is deployed, the initial conditions $\mathbf{x}_\text{IC}$ are drawn from the same PDF used for estimating $\hat{\rho}$ and $\hat{p}^*$, we expect the rate of $\mathbf{x}_\text{IC}$ for which there exists an $\mathcal{N}^*(\mathbf{x}_\text{IC})$ in $\mathcal{T}$ to be approximately $\hat{\rho}$. Similarly, the rate at which $\mathbf{x}_\text{IC}$ are stabilized to $\mathcal{G}$, given that there exists an $\mathcal{N}^*(\mathbf{x}_\text{IC})$, should be approximately $\hat{p}^*$. Finally, the product $\hat{p}^* \cdot \hat{\rho}$ estimates the probability of a given initial condition $\mathbf{x}_\text{IC}$ to be stabilized, and is a lower bound on the probability density contained in $\mathcal{S}_\mathcal{D}$, or, if a uniform PDF is used for testing, a lower bound on the relative volume of $\mathcal{S}_\mathcal{D}$ in $\mathcal{D}$.

How well the estimates approximate the true parameters can be analyzed with confidence intervals for the parameters ρ and p^*. Since N_ρ and N_p have a binomial distribution, one can use, for example, Clopper-Pearson intervals (Clopper and Pearson 1934). The intervals allow choosing the number N_s of samples to be tested, since more samples result in tighter confidence intervals; and making a decision about the tree-policy $\mathcal{T}$ being acceptable or not, for example by requiring that $\hat{p}^*$ is above some desired lower bound by a statistically significant amount. However, while we know that ideally, $p^* = 1$, the maximally achievable coverage ratio ρ is unknown, since $\mathcal{S}_\mathcal{D}$ is unknown.

In practice, we found a link between the success-rate quality measure and the termination heuristic: In the examples, choosing α sufficiently small (we used $\alpha = 0.01$) and setting $\bar{\text{p}}_\alpha$ to a desired success rate p^* resulted in success rate estimates $\hat{p}^*$ that were mostly above the desired p^* by a statistically significant amount.

5 Example Implementation

We present a detailed example implementation of algorithm and discuss practical design decisions. The implementation is used to generate tree-policies for the two example systems in Section 6: 1) A simple pendulum that is controlled by a torque at the pivot joint; and 2) a cart-pole system, where an actuated cart moving on a limited rail balances an inverted pendulum. The task for both systems is to stabilize the

system from a "large" design set $\mathcal{D}$ of initial conditions to a goal state $\mathbf{x}_{\mathrm{G}}$ where the pendulum is stabilized at its unstable equilibrium (and the cart is at rest at the center of the rail).

In order to achieve the task, the tree-policy stabilizes the design set $\mathcal{D}$ to a goal set $\mathcal{G}$ that is contained in the basin of attraction of the goal state $\mathbf{x}_{\mathrm{G}}$, which is stabilized with a linear controller (an LQR design). The basin of attraction of $\mathbf{x}_{\mathrm{G}}$ is the set of states that the linear controller can asymptotically stabilize to $\mathbf{x}_{\mathrm{G}}$, see the definition in Section 6.4 in Strogatz (2014). This allows a straightforward two-step control strategy to stabilize an initial condition $\mathbf{x}_{\mathrm{IC}} \in \mathcal{D}$ to the goal state $\mathbf{x}_{\mathrm{G}}$: First, apply the finite-time node-policy of $\mathcal{N}^*(\mathbf{x}_{\mathrm{IC}})$ to stabilize $\mathbf{x}_{\mathrm{IC}}$ to $\mathcal{G}$; then, switch to the policy that stabilizes the system at $\mathbf{x}_{\mathrm{G}}$. Since $\mathcal{G}$ is in the basin of attraction of the feedback-stabilized goal state, the system state converges to $\mathbf{x}_{\mathrm{G}}$. Before the motion-planning and feedback-control modules of the algorithm are described, we present the goal-state controller and a straightforward strategy to approximate a goal set $\mathcal{G}$ that is contained in the basin of attraction of $\mathbf{x}_{\mathrm{G}}$.

Since both systems are continuous-time, we present the necessary time-discretization steps that allow a straightforward implementation of the generated policies on digital control hardware, which is typically used for robot control. The goal of this section is to allow the reader to implement the algorithm in practice, together with the concepts introduced in Section 4. Furthermore, MATLAB code of the example implementation is available in Extension 1.

5.1 Continuous-Time System Dynamics

The systems considered in this section are both continuous-time (CT). In the following, we use the nonlinear, CT dynamics

$$\dot{\mathbf{x}}(t) = \mathbf{f}_c(\mathbf{x}(t), \mathbf{u}(t)) \tag{13}$$

to represent the time-invariant dynamics of the two example systems, where $\mathbf{x}(t) \in \mathbb{R}^{\mathrm{n}}$ and $\mathbf{u}(t) \in \mathbb{R}^{\mathrm{m}}$ are the system state and control input at time t, respectively. Analogous to the discrete-time (DT) definitions in Section 2, we define a solution to (13), i.e. the state $\mathbf{x}(t_1)$ at time t_1 as

$$\mathbf{x}(t_1) =: \boldsymbol{\phi}_c^{t_1 - t_0}(\mathbf{u}(t), \mathbf{x}(t_0)) \tag{14}$$

where $\mathbf{x}(t_0)$ is the initial state at time t_0. The solution is usually obtained by numerical integration of (13); for example, we use the MATLAB `ode45` function. For the discrete-time control approach presented here, we often require the solution (14) for a single sampling period $\bar{\tau}$ and a zero-order hold input (which is how the control input is generated in many experimental systems). Therefore, we define the discrete-time dynamics

$$\begin{aligned} \mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) &:= \boldsymbol{\phi}_c^{\bar{\tau}}(\mathbf{u}(t), \mathbf{x}(0) = \mathbf{x}_k) \\ \text{with } \mathbf{u}(t) &:= \mathbf{u}_k, \quad 0 \leq t < \bar{\tau} \end{aligned} \tag{15}$$

where we used the time-invariance of the dynamics (13). *Remark:* We use sampled state-trajectories obtained with (15) when testing funnels in simulations, and, specifically, when checking whether state constraints are violated, see (8). Therefore, it may be that state constraints are violated by the continuous trajectory in between samples. We ignored this issue in the example implementation since the chosen sampling times are small; for larger sampling times, one could appropriately pre-shrink the state constraints, similar to the approach presented in Gillula et al. (2014).

5.2 Control Design for Goal State Stabilization

Without loss of generality, we define the goal state and input to be at the origin, $\mathbf{x}_{\mathrm{G}} := 0$, $\mathbf{u}_{\mathrm{G}} := 0$. The control design involves three steps: 1) linearization of the CT nonlinear system dynamics (13) about the goal state and input; 2) discretization of the resulting CT linear time-invariant (LTI) system; and 3) design of the DT LQR controller. The first two steps are described in Section 4.3.6 in Franklin et al. (1998) and are omitted here, and the third in Section 3.3 in Anderson and Moore (2007).

5.2.1 Time-Invariant LQR Design: After linearization and discretization of the system (13) with a uniform sampling period $\bar{\tau}$, we obtain the DT LTI system

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k. \tag{16}$$

For both example systems, the pair $\mathbf{A}, \mathbf{B}$ of the DT LTI system (16) is controllable. Therefore, we can design time-invariant (TI), DT LQR policies to stabilize the goal states of the systems. The policies minimize the infinite-horizon cost-to-go from an initial state $\mathbf{x}_0$

$$J(\mathbf{x}_0) := \sum_{k=0}^{\infty} \left[\mathbf{x}_k^{\mathsf{T}} \mathbf{Q}_{\mathrm{G}} \mathbf{x}_k + \mathbf{u}_k^{\mathsf{T}} \mathbf{R}_{\mathrm{G}} \mathbf{u}_k\right] \tag{17}$$

where $^{\mathsf{T}}$ denotes the transpose; the matrix $\mathbf{Q}_{\mathrm{G}}$ penalizes state deviations and is positive semi-definite: $\mathbf{Q}_{\mathrm{G}} \geq 0$; and $\mathbf{R}_{\mathrm{G}}$ penalizes control effort and is positive definite: $\mathbf{R}_{\mathrm{G}} > 0$. Both $\mathbf{Q}_{\mathrm{G}}$ and $\mathbf{R}_{\mathrm{G}}$ are user-defined tuning parameters. It can be shown (Anderson and Moore 2007) that the optimal cost-to-go is

$$J^*(\mathbf{x}_k) = \mathbf{x}_k^{\mathsf{T}} \mathbf{S}_{\mathrm{G}} \mathbf{x}_k \tag{18}$$

with $\mathbf{S}_{\mathrm{G}} \geq 0$, and that the optimal linear feedback policy is

$$\boldsymbol{\pi}^{\mathrm{TI}}(\mathbf{x}_k) := -\mathbf{K}_{\mathrm{G}}\mathbf{x}_k. \tag{19}$$

Both $\mathbf{K}_{\mathrm{G}}$ and $\mathbf{S}_{\mathrm{G}}$ can be obtained using the MATLAB `dlqr` command. For both example systems, the input $\mathbf{u}_k$ is scalar and constrained to $|\mathbf{u}_k| \leq \mathbf{u}_{\max}$, where $\mathbf{u}_{\max}$ is a constant based on the respective actuator limits. Therefore, we define the goal-state feedback policy that satisfies the constraints as

$$\boldsymbol{\pi}^{\mathrm{G}}(\mathbf{x}_k) := \begin{cases} \boldsymbol{\pi}^{\mathrm{TI}}(\mathbf{x}_k), & |\boldsymbol{\pi}^{\mathrm{TI}}(\mathbf{x}_k)| \leq \mathbf{u}_{\max} \\ \mathrm{sgn}(\boldsymbol{\pi}^{\mathrm{TI}}(\mathbf{x}_k))\mathbf{u}_{\max}, & \text{otherwise} \end{cases} \tag{20}$$

where sgn is the signum function. For multi-input systems, the above is applied element-wise. Note that in order to keep the notation consistent in this section, we keep the bold-face vector notation for the systems' scalar inputs and policies.

For the LQR controllers in both example systems, we choose $\mathbf{Q}_{\mathrm{G}} > 0$, from which follows that the pair $\mathbf{A}, \mathbf{G}$ is observable, with $\mathbf{G}$ any matrix such that $\mathbf{Q}_{\mathrm{G}} = \mathbf{G}^{\mathrm{T}}\mathbf{G}$. From $\mathbf{A}, \mathbf{B}$ controllable and $\mathbf{A}, \mathbf{G}$ observable, it follows that the closed-loop dynamics

$$\mathbf{x}_{k+1} = (\mathbf{A} - \mathbf{K}_{\mathrm{G}}\mathbf{B})\mathbf{x}_k \tag{21}$$

are asymptotically stable and $\mathbf{S}_{\mathrm{G}} > 0$, see Section 3.3 in Anderson and Moore (2007). $\mathbf{S}_{\mathrm{G}} > 0$ allows using the optimal cost-to-go (18) as a distance metric to describe the goal as the sub-level set

$$\mathcal{G} := \{\mathbf{x} : J^*(\mathbf{x}) < \epsilon_{\mathrm{G}}\} \tag{22}$$

where ϵ_{G} is a constant, which we determine a priori as follows.

5.3 Approximating the Goal-State Basin of Attraction

We present a straightforward, sampling-based approach to approximate ϵ_{G} such that $\mathcal{G}$, defined in (22), is contained in the basin of attraction of the LQR-stabilized goal state.

For the given linear closed-loop dynamics (21), $J^*(\mathbf{x})$ is a Lyapunov function, which implies that

$$J^*((\mathbf{A} - \mathbf{K}_{\mathrm{G}}\mathbf{B})\mathbf{x}) - J^*(\mathbf{x}) < 0 \tag{23}$$

holds for all $\mathbf{x} \neq \mathbf{x}_{\mathrm{G}} = 0$, which implies global asymptotic stability of $\mathbf{x}_{\mathrm{G}}$ (Anderson and Moore 2007). For the nonlinear closed-loop dynamics (15)

$$J^*(\mathbf{f}(\mathbf{x}, \boldsymbol{\pi}^{\mathrm{G}}(\mathbf{x}))) - J^*(\mathbf{x}) < 0 \tag{24}$$

does not typically hold for all $\mathbf{x}$; however, it may hold for a neighborhood of $\mathbf{x}_{\mathrm{G}}$. Similar to previous work (Reist and Tedrake 2010; Tedrake et al. 2010), we find an ϵ_{G} for the goal set $\mathcal{G}$ according to (22) such that for all $\mathbf{x} \in \mathcal{G}$, the inequality (24) holds, which implies asymptotic stability of all $\mathbf{x} \in \mathcal{G}$. Instead of verifying the Lyapunov function on $\mathcal{G}$ using sums-of-squares programming as in Tedrake et al. (2010), we use, analogous to the funnel falsification approach, sampling and simulation to falsify that for all states $\mathbf{x}$ in the *goal basin hypothesis*

$$\hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}}) := \{\mathbf{x} : J^*(\mathbf{x}) < \hat{\epsilon}_{\mathrm{G}}\} \tag{25}$$

the inequality (24) holds. The parameter $\hat{\epsilon}_{\mathrm{G}}$ is determined with the straightforward procedure:

0. Initialize $\hat{\epsilon}_{\mathrm{G}} > 0$ such that $\hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}}) \supset \mathcal{D}$.
1. Draw a random sample $\mathbf{x}_{\mathrm{S}} \in \hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}})$ from a uniform distribution on $\hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}})$. We use the algorithm presented in Section 3.3.1 in Sun and Farooq (2002).
2. Check state constraints: If $\mathbf{x}_{\mathrm{S}} \in \mathcal{X}$, proceed to Step 3; otherwise, proceed to Step 4.
3. Calculate $\mathbf{f}(\mathbf{x}_{\mathrm{S}}, \boldsymbol{\pi}^{\mathrm{G}}(\mathbf{x}_{\mathrm{S}}))$ using numerical integration. Check if the Lyapunov test (24) holds. If yes, return to Step 1. If not, proceed to Step 4.
4. Shrink the hypothesis with

$$\hat{\epsilon}_{\mathrm{G}} \leftarrow \mathbf{x}_{\mathrm{S}}^{\mathrm{T}}\mathbf{S}_{\mathrm{G}}\mathbf{x}_{\mathrm{S}} \tag{26}$$

and return to Step 1.

The procedure terminates when a consecutive sequence of M samples fulfill the Lyapunov test, where M is designed analogous to the termination condition for the LQR-Tree algorithm (11). The quality measures presented in Section 4.5.2 can be straightforwardly adapted to test if the approximation $\hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}})$ is acceptable. Sample MATLAB code of the procedure is available in Extension 1.

In Fig. 4, we show the set $\hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}})$ that was approximated for the simple pendulum system. The approximation is compared to an estimated basin of attraction (the nonstriped area), which was obtained by simulating the nonlinear closed-loop system from discretized states and checking whether the system converges to $\mathbf{x}_{\mathrm{G}}$ in a user-defined, finite time. Also shown (in gray) are the discretized states for which the Lyapunov test (24) fails. While $\hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}})$ is a nonconservative approximation to the set for which $J^*(\mathbf{x})$ is a valid Lyapunov function, it turns out that $\hat{\mathcal{G}}(\hat{\epsilon}_{\mathrm{G}})$ appears to be a conservative approximation to the basin of attraction. The intuition is that verifying a Lyapunov function on a set of states is a sufficient, but not a necessary condition for the set to be contained in the basin of attraction. One may be able

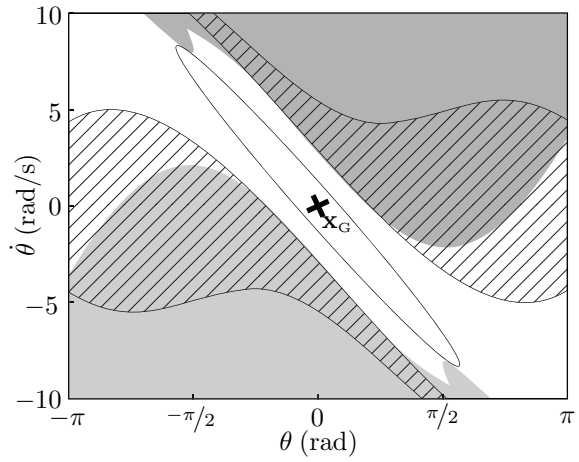

Figure 4. Approximated goal basin $\hat{\mathcal{G}}(\hat{\epsilon}_G)$ (ellipse) for the simple pendulum with states pendulum angle θ and angular velocity $\dot{\theta}$. The gray area are states for which the Lyapunov test (24) fails; and in the white area the test succeeds. The striped area marks the set of states that are not in the numerically estimated basin of attraction.

find candidate Lyapunov functions that result in a tighter approximation to the basin of attraction, see, for example, the approach presented in Moore et al. (2014).

An evident alternative approach is to falsify the basin hypothesis $\hat{\mathcal{G}}(\hat{\epsilon}_G)$ directly by checking whether the system converges to the goal state from a given initial condition. This approach would result in a tighter fit of the hypothesis to the basin of attraction, but would introduce additional design parameters such as a maximally allowed time for the system to converge to within a user-defined neighborhood of the goal. The Lyapunov approach allows to avoid defining these additional parameters and can be evaluated more efficiently, since only a single sampling period $\bar{\tau}$ is simulated. Furthermore, a conservative approximation is desirable for the two-step control strategy of first applying the finite-time tree-policy and then switching to the goal-state controller, since the strategy relies on $\hat{\mathcal{G}}(\hat{\epsilon}_G)$ being in the basin of attraction of $\mathbf{x}_G$. However, the conservativeness of $\hat{\mathcal{G}}(\hat{\epsilon}_G)$ in the example in Fig. 4 is a coincidence, and does not generalize: suppose $\mathbf{S}_G$ was identity in Fig. 4, then $\hat{\mathcal{G}}(\hat{\epsilon}_G)$ would overlap with both the gray and striped regions. In practice, we can accept if $\hat{\mathcal{G}}(\hat{\epsilon}_G)$ is not perfectly conservative: The approximation is based on an inaccurate model of the real system anyway, and one may, for example, tighten the input constraints in the simulation to obtain more conservative approximations, and to allow for extra control authority on the real system to compensate for modeling errors.

The goal set can also be a user-defined, task-specific set of states, e.g. the set of states for which a glider achieves perching on a string as shown in Moore et al. (2014); or it can be approximated with a method based on sum-of-squares verification (Tedrake et al. 2010; Topcu et al. 2008).

5.4 The Motion Planning Module

A key component of the algorithm is the motion-planning module that generates open-loop trajectories starting at a random sample and ending inside the goal set. In the example implementation, we use an optimal-control-based, direct-transcription method as described in Section 4.5 in Betts (2010). The method attempts to find an open-loop trajectory $\{\bar{\mathbf{u}}_k\}_N$, $\{\bar{\mathbf{x}}_k\}_N$ that is uniformly sampled with period τ, that starts at the sample $\bar{\mathbf{x}}_0 = \mathbf{x}_S$, and that ends at the goal state $\bar{\mathbf{x}}_N = \mathbf{x}_G \in \hat{\mathcal{G}}(\hat{\epsilon}_G)$ with the following minimization problem:

$$\text{Find } \tau,\ \{\bar{\mathbf{u}}_k\}_N,\ \{\bar{\mathbf{x}}_k\}_N \tag{27}$$

$$\text{that minimize } \sum_{k=0}^{N-1} \tau\,(\bar{\mathbf{x}}_k^\mathsf{T}\mathbf{Q}\bar{\mathbf{x}}_k + \bar{\mathbf{u}}_k^\mathsf{T}\mathbf{R}\bar{\mathbf{u}}_k) \tag{28}$$

$$\begin{aligned} \text{subject to: } & \bar{\mathbf{x}}_{k+1} = \mathbf{f}(\bar{\mathbf{u}}_k, \bar{\mathbf{x}}_k), \\ & \bar{\mathbf{x}}_0 = \mathbf{x}_S \text{ and } \bar{\mathbf{x}}_N = \mathbf{x}_G, \\ & |\bar{\mathbf{u}}_k| \le \mathbf{u}_{\max}^{\text{mp}},\ \forall k, \\ & \bar{\mathbf{x}}_k \in \mathcal{X}^{\text{mp}},\ \forall k, \\ & 0 < \tau \le \tau_{\max} \end{aligned} \tag{29}$$

where the matrices $\mathbf{Q} \ge 0$, $\mathbf{R} > 0$ are design parameters that, analogous to the TI LQR design, characterize the quadratic cost function to minimize. In order to reduce the number of design parameters, we use the same matrices as for the trajectory-stabilizing TV LQR controller described next in Section 5.5. The method allows introducing the initial and final states as state constraints, since it can explicitly handle input and state constraints. The system dynamics are introduced as a constraint on successive states in the trajectory (29), where the solution (15) is approximated by a standard 4th-order Runge-Kutta step. Finally, the variable sampling time τ allows the method to explore different time durations for the trajectory, up to a maximal time $N \cdot \tau_{\max}$.

Lastly, we introduce $\mathbf{u}_{\max}^{\text{mp}}$ as a design element with the intent of imposing stricter (element-wise) input constraints for motion-planning, i.e. $\mathbf{u}_{\max}^{\text{mp}} < \mathbf{u}_{\max}$; for example, we used $\mathbf{u}_{\max}^{\text{mp}} = 0.6\mathbf{u}_{\max}$ in the cart-pole example in Section 6.2. The stricter input constraints

result in more control authority for the trajectory-stabilizing policy. Analogously, if there are state constraints $\mathcal{X}$, it can help to use a reduced set $\mathcal{X}^{\text{mp}} \subset \mathcal{X}$ to allow for some deviations from nominal state trajectories. Both stricter input- and state-constraints should result in larger node-funnels.

The resulting nonlinear program is run with the sequential quadratic programming tool SNOPT (Gill et al. 2002). Since the problem is typically nonconvex, the resulting trajectories are not necessarily global minima of (28). Therefore, the example implementation does not generate an optimal policy, and an example of a suboptimal trajectory is shown in Section 6. However, the focus of the algorithm and example implementation is on generating a policy that can stabilize the system to the goal from a wide range of initial conditions.

The length N of the trajectory is set a priori; letting τ be a design variable allows the motion-planner to scale time to satisfy, if feasible, the initial and final state constraints. The disadvantage is that the motion-planner yields varying sampling times for the trajectories in the tree. If the tree-policy is to be implemented on digital control hardware, homogeneous sampling times are desirable. Therefore, we resample the trajectories to a uniform sampling time $\bar{\tau}$, which is identical to the sampling time used in the discretization step in the TI LQR design, see Section 5.2.1. If the time τ found by the motion planner is above or below a threshold, we rerun the optimization after the interpolation step to refine the interpolated trajectory. More details about the motion-planning approach we use may be found in Betts (2010) and in Extension 1.

5.4.1 Initializing Motion-Planning: We find that the motion-planner performs better if it is initialized with an appropriate initial guess for the open-loop trajectory. If available, we use one of the trajectories generated by simulations that failed to stabilize the sample $\mathbf{x}_{\text{S}}$. Specifically, assume that the j-th simulation trajectory with length N_j is described by the state and input sets $\{\mathbf{x}_k^j\}_{N_j}$, $\{\mathbf{u}_k^j\}_{N_j}$. Then, we initialize motion planning with the failed trajectory that minimizes

$$(\mathbf{x}_{N_j}^j)^{\mathsf{T}}\mathbf{S}_{\text{G}}\mathbf{x}_{N_j}^j + \sum_{k=0}^{N_j-1} (\mathbf{x}_k^j)^{\mathsf{T}}\mathbf{Q}\mathbf{x}_k^j + (\mathbf{u}_k^j)^{\mathsf{T}}\mathbf{R}\mathbf{u}_k^j \qquad (30)$$

which is the combination of the motion-planning (28) and TI LQR costs (18). The rationale of this heuristic is to initialize motion planning with a trajectory that got "close" to the goal. In case that the sample is not in any node funnel, we can obtain an initializing trajectory by simulating the sample with the node-policy that minimizes (10), ignoring the funnel constraint.

5.4.2 Remark related to the motion-planning assumption made in the algorithm introduction in Section 2.4: We argue that initializing the motion planner with failed trajectories helps the algorithm to cover parts of the stabilizable set $\mathcal{S}_{\mathcal{D}}$ for which motion planning is difficult without a "good" initial guess. As the algorithm grows the tree, there are trajectories added in the vicinity of the difficult region. If the policies of these trajectories are applied to samples from the difficult region, the resulting failed trajectories provide better initial guesses to the motion planner. Therefore, while the motion planner may not succeed for all samples initially, it may be that as the tree grows, the motion planner is able to find trajectories for previously failed samples. This reasoning is similar to the discussion about the motion-planning assumption in Tedrake et al. (2010).

5.5 Stabilizing Trajectories with Time-Varying LQR Policies

The open-loop trajectories obtained from the motion planner are stabilized with time-varying (TV) LQR policies. The design steps are similar to the TI LQR design: First, the nonlinear, CT dynamics (13) are linearized about the nominal states and inputs of the DT trajectory. Then, we discretize the linearized dynamics assuming a constant control input and a constant linearization during a single sampling period. This results in a DT, linear time-varying (LTV) system. Finally, we design an LTV LQR policy with a backward iteration of a Riccati equation as presented in Section 4.1 in Bertsekas (2005). Since the *discrete-time* TV LQR design is not common in the literature, we include more details in the following derivation.

5.5.1 Linearization and Discretization: Consider the open-loop trajectory $\{\bar{\mathbf{u}}_k\}_N, \{\bar{\mathbf{x}}_k\}_N$ that is to be stabilized by an LTV LQR policy. We first linearize the CT system dynamics (13) about the DT nominal trajectory and assume this linearization to be constant during a sampling period:

$$\dot{\tilde{\mathbf{x}}}(t) = \mathbf{A}_k^c\tilde{\mathbf{x}}(t) + \mathbf{B}_k^c\tilde{\mathbf{u}}(t), \text{ for } k\bar{\tau} \le t < (k+1)\bar{\tau} \qquad (31)$$

where

$$\mathbf{A}_k^c := \left.\frac{\partial \mathbf{f}_c(\mathbf{x},\mathbf{u})}{\partial \mathbf{x}}\right|_{\substack{\mathbf{x}=\bar{\mathbf{x}}_k \\ \mathbf{u}=\bar{\mathbf{u}}_k}}, \quad \mathbf{B}_k^c := \left.\frac{\partial \mathbf{f}_c(\mathbf{x},\mathbf{u})}{\partial \mathbf{u}}\right|_{\substack{\mathbf{x}=\bar{\mathbf{x}}_k \\ \mathbf{u}=\bar{\mathbf{u}}_k}} \qquad (32)$$

are the linearizations about the nominal state and input, and $\tilde{\mathbf{x}}(t) := \mathbf{x}(t) - \bar{\mathbf{x}}_k$, $\tilde{\mathbf{u}}(t) := \mathbf{u}(t) - \bar{\mathbf{u}}_k$ are the state and input deviations from the respective (assumed constant) nominal states and inputs. Assuming a zero-order hold input, $\tilde{\mathbf{u}}(t) := \tilde{\mathbf{u}}_k$ for $k\bar{\tau} \le t < (k+1)\bar{\tau}$, we discretize the LTV system (31) with

the matrix exponential, resulting in the DT LTV system dynamics

$$\tilde{\mathbf{x}}_{k+1} = \mathbf{A}_k \tilde{\mathbf{x}}_k + \mathbf{B}_k \tilde{\mathbf{u}}_k \tag{33}$$

with $\tilde{\mathbf{x}}_k := \mathbf{x}_k - \bar{\mathbf{x}}_k = \tilde{\mathbf{x}}(t = k\bar{\tau})$ and $\tilde{\mathbf{u}}_k := \mathbf{u}_k - \bar{\mathbf{u}}_k$.

5.5.2 TV LQR Design: For the DT LTV system, we design a TV LQR policy that minimizes

$$J_k\left(\tilde{\mathbf{x}}_k\right) := \tilde{\mathbf{x}}_N^\intercal \mathbf{Q}_N \tilde{\mathbf{x}}_N + \sum_{n=k}^{N-1} \left[\tilde{\mathbf{x}}_n^\intercal \mathbf{Q} \tilde{\mathbf{x}}_n + \tilde{\mathbf{u}}_n^\intercal \mathbf{R} \tilde{\mathbf{u}}_n\right] \tag{34}$$

where $\mathbf{Q}_N, \mathbf{Q} \geq 0$, and $\mathbf{R} > 0$ are penalty matrices on the final state deviation and on state and input deviations from the nominal trajectory, respectively. We use the cost-to-go matrix of the TI LQR design for $\mathbf{Q}_N = \mathbf{S}_\mathrm{G}$. It can be shown (Bertsekas 2005) that the optimal cost-to-go is

$$J_k^*\left(\tilde{\mathbf{x}}_k\right) = \tilde{\mathbf{x}}_k^\intercal \mathbf{S}_k \tilde{\mathbf{x}}_k \tag{35}$$

where $\mathbf{S}_k \geq 0$ is given by backwards-iterating the Riccati equation

$$\begin{aligned}\mathbf{S}_k = \mathbf{Q} + \mathbf{A}_k^\intercal \big(&\mathbf{S}_{k+1} - \\ &\mathbf{S}_{k+1}\mathbf{B}_k \left(\mathbf{R} + \mathbf{B}_k^\intercal \mathbf{S}_{k+1} \mathbf{B}_k\right)^{-1} \mathbf{B}_k^\intercal \mathbf{S}_{k+1}\big) \mathbf{A}_k\end{aligned} \tag{36}$$

with the terminal condition $\mathbf{S}_N = \mathbf{Q}_N$. The optimal TV policy is

$$\boldsymbol{\pi}_k^{\mathrm{TV}}(\mathbf{x}_k) := -\left(\mathbf{R} + \mathbf{B}_k^\intercal \mathbf{S}_{k+1} \mathbf{B}_k\right)^{-1} \mathbf{B}_k^\intercal \mathbf{S}_{k+1} \mathbf{A}_k \tilde{\mathbf{x}}_k. \tag{37}$$

Finally, we incorporate the input constraints to define the TV trajectory policy

$$\boldsymbol{\pi}_k^{\mathcal{J}}(\mathbf{x}_k) := \begin{cases} \boldsymbol{\pi}_k^{\mathrm{TV}}(\mathbf{x}_k), & |\boldsymbol{\pi}_k^{\mathrm{TV}}(\mathbf{x}_k)| \leq \mathbf{u}_{\max} \\ \mathrm{sgn}(\boldsymbol{\pi}_k^{\mathrm{TV}}(\mathbf{x}_k)) \mathbf{u}_{\max}, & \text{otherwise.} \end{cases} \tag{38}$$

5.6 Node Funnel Hypotheses

Analogous to the goal basin hypothesis, we use the cost-to-go obtained in the TV LQR design (35) to describe the funnel hypothesis of a tree node $\mathcal{N}_n$ with

$$\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n) := \{\mathbf{x}_n : (\mathbf{x}_n - \bar{\mathbf{x}}_n)^\intercal \mathbf{S}_k (\mathbf{x}_n - \bar{\mathbf{x}}_n) < \hat{\epsilon}_n\}. \tag{39}$$

Note that the geometry of the funnel hypothesis is a design parameter, and that input- and state-constraints are considered by the hypothesis by definition in Section 4.1.2.

The matrices $\mathbf{S}_k$ must be positive definite in order to obtain a valid distance metric in (39). Since the final cost matrix $\mathbf{S}_N = \mathbf{S}_\mathrm{G} > 0$, and $\mathbf{Q} \geq 0$, $\mathbf{R} > 0$, it follows that all $\mathbf{S}_k > 0$ from the dynamic programming update, see the argumentation in Section 4.1 in Bertsekas (2005).

Remarks: Similar to the discussion about $\hat{\mathcal{G}}(\hat{\epsilon}_\mathrm{G})$ approximating the goal-state basin of attraction, it is unlikely that an ellipse is the best geometrical primitive to describe the funnel around a trajectory. Simulation-based approximation of funnels allows exploring different primitives that could potentially yield tighter fits to the real funnel, further improving the sparsity of the resulting tree. The advantages of the hyper-ellipses we use here are their simple geometry, and that their shape is based on the TV LQR design that is tied to the system dynamics.

In the conceptual discussion of the algorithm, the distance metric defining the funnel hypotheses is identical for all nodes. While this is not the case anymore with (39), it is not an issue for the algorithm. The ability to use different distance metrics for different nodes, and therefore also different regions of the state space, highlights another intuition behind the usefulness of funnel hypotheses: They help with the difficulty of choosing suitable metrics to measure distance between different states of a dynamic system (LaValle 2006; Glassman and Tedrake 2010). Typically, the hypotheses are "small", which means that the metric must only provide a reasonable heuristic for states "nearby" the nominal state trajectories.

5.7 Query-Phase Implementation: Tree-Policy

Given an initial condition $\mathbf{x}_\mathrm{IC}$ from $\mathcal{D}$, we find the node $\mathcal{N}^*(\mathbf{x}_\mathrm{IC})$ according to (10). Let the node $\mathcal{N}^*(\mathbf{x}_\mathrm{IC})$ be the n-th node in a trajectory of length N that is stabilized by the TV LQR policy $\boldsymbol{\pi}_k^{\mathcal{J}}(\mathbf{x}_k)$, and set $\mathbf{x}_n = \mathbf{x}_\mathrm{IC}$. The following feedback policy defines the tree-policy that is applied to $\mathbf{x}_\mathrm{IC}$:

$$\boldsymbol{\pi}_k^{\mathrm{Tree}}(\mathbf{x}_k) := \begin{cases} \boldsymbol{\pi}_k^{\mathcal{J}}(\mathbf{x}_k), & \text{for } n \leq k < N \\ \boldsymbol{\pi}^{\mathrm{G}}(\mathbf{x}_k), & \text{for } k \geq N \end{cases} \tag{40}$$

i.e. first apply the TV LQR policy of node $\mathcal{N}^*(\mathbf{x}_\mathrm{IC})$ and then apply the goal-state-stabilizing TI LQR policy. Finally, if there does not exist a feasible $\mathcal{N}^*(\mathbf{x}_\mathrm{IC})$ in the tree because $\mathbf{x}_\mathrm{IC}$ is not in any hypothesis, one may, for example, apply some fall-back policy that stabilizes the system to a "safe" region. Another possibility is to relax the in-funnel-constraint in (10) when determining $\mathcal{N}^*(\mathbf{x}_\mathrm{IC})$, which results in the application of the policy of the "closest" node as measured by the TV LQR cost-to-go.

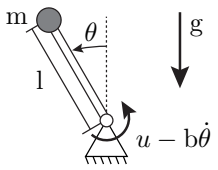

Figure 5. The simple pendulum.

6 Example Systems and Results

With the algorithm implemented as described above, we generated and evaluated tree-policies for two example systems.

6.1 Simulated Example: Simple Pendulum

Examples of tree-policies are illustrated with a well-visualizable system: the torque limited, damped simple pendulum, shown in Fig. 5. The system dynamics are

$$\ddot{\theta} = \frac{1}{\mathrm{ml}^2}\left(u + \mathrm{mgl}\sin\theta - \mathrm{b}\dot{\theta}\right) \tag{41}$$

where the pendulum angle is θ (rad) and the angular velocity is $\dot{\theta}$ (rad/s). The system state is $\mathbf{x} := (\theta, \dot{\theta})$. The input u ($\mathrm{N}\cdot\mathrm{m}$) is the torque applied to the pendulum at the pivot point and is limited to $|u| \leq 3\,\mathrm{N}\cdot\mathrm{m}$, requiring at least a single pump for a swing-up from the lower equilibrium. There are no state constraints. The system parameters are the same as in the example in Tedrake et al. (2010): mass $\mathrm{m} = 1.0\,\mathrm{kg}$, length $\mathrm{l} = 0.5\,\mathrm{m}$, gravity $\mathrm{g} = 9.8\,\mathrm{m/s^2}$, and damping term $\mathrm{b} = 0.1\,\mathrm{N}\cdot\mathrm{m}\cdot\mathrm{s}$. The pendulum is upright at rest when $\mathbf{x}_\mathrm{G} = \mathbf{0}$, $u_\mathrm{G} = 0$, which are the goal state and input, respectively. The box-shaped design set $\mathcal{D}$ is defined by the bounds $-3\pi/2 \leq \theta \leq \pi/2$ and $|\dot{\theta}| \leq 10\,\mathrm{rad/s}$. For simplicity, we do not take into account that the pendulum angle wraps around, i.e. that $\theta = 0\,\mathrm{rad}$ and $\theta = 2\pi$ result in the same pendulum position. Therefore, the state $\mathbf{x} = (2\pi, 0)$ is not a valid goal state, and the policy will swing the pendulum back to $\mathbf{x}_\mathrm{G} = \mathbf{0}$.

6.1.1 TI LQR Design and Goal Set: The feedback policy that stabilizes the goal state is designed as described in Section 5.2, with $\bar{\tau} = 0.05\,\mathrm{s}$, $\mathbf{Q}_\mathrm{G} = \mathrm{diag}(10, 1)$, and $\mathbf{R}_\mathrm{G} = 15$, where diag refers to a diagonal matrix with the corresponding entries starting at the (1,1)-entry. The goal set $\mathcal{G}$ was approximated with the procedure outlined in Section 5.3 using the termination heuristic parameters $\alpha = 0.01$ and $\bar{\mathrm{p}}_\alpha = 0.99$, requiring $\mathrm{M} = 459$ consecutive samples for which the Lyapunov condition (24) holds. The approximation takes less than 10 s, and the resulting $\hat{\mathcal{G}}(\hat{\epsilon}_\mathrm{G})$ is shown in Fig. 4, and as the dark gray ellipse in Fig. 6.

6.1.2 Motion Planning and TV LQR Design: Motion planning and trajectory stabilization are implemented as described in Section 5.4 and Section 5.5 using the same penalty matrices as for the TI LQR design: $\mathbf{Q} = \mathbf{Q}_\mathrm{G}$ and $\mathbf{R} = \mathbf{R}_\mathrm{G}$. The input constraints for motion planning are reduced to $u_{\max}^{\mathrm{mp}} = 2\,\mathrm{N}\cdot\mathrm{m}$, such that there is some control authority available for the TV LQR policy to stabilize open-loop trajectories. The trajectories generated by the motion planner are resampled to obtain a uniform sampling time of $\bar{\tau} = 0.05\,\mathrm{s}$ in the tree-policy.

6.1.3 Results: We generate the tree-policy according to Section 4 and Section 5. The termination heuristic is the same as for the goal set approximation, $\mathrm{M} = 459$. Two example tree-policies are shown in Fig. 6, along with their node, runtime, and iteration numbers. The runtimes were obtained with a MATLAB implementation on a desktop PC (CPU: i7-3770 with 3.4 GHz). On average over 100 generated tree-policies, we found a mean runtime of 2.0 min with standard deviation 36 s, and a mean number of tree-nodes of 227 with standard deviation 50.

A statistical policy evaluation according to Section 4.5.2 is performed with $N_s = 2000$ random initial conditions drawn uniformly from $\mathcal{D}$. Success was determined by checking for convergence to the goal state after simulating the system for another 3 s after the TV LQR policy ends and the goal TI LQR policy is activated according to Section 5.7 (to implicitly test the goal basin approximation). Both examples shown in Fig. 6 have coverage ratio estimates of 1.0, i.e. it is likely that all initial conditions in $\mathcal{D}$ are assigned to a node-policy. Both examples reached high success rate estimates that are well above the design parameter $\bar{\mathrm{p}}_\alpha = 0.99$ used in the termination condition. All of the failed stabilizations in the evaluations did not reach the goal $\mathcal{G}$, i.e. were not due to a poor approximation of the basin of attraction, which is not surprising given the conservativeness of $\mathcal{G}$ shown in Fig. 4. The coverage and success rates achieved in the two examples are representative: In the 100 generated policies used to assess runtime, only one policy did not achieve a coverage rate estimate of 1.0, and the success rate estimate mean was 0.996 with standard deviation 0.002.

A higher $\bar{\mathrm{p}}_\alpha$ or a lower significance α would likely result in a higher success rate at the cost of a longer runtime of the algorithm. For example, increasing $\bar{\mathrm{p}}_\alpha$ to 0.995 and re-generating the tree-policy shown on top in Fig. 6 using the same random-number-generator seed resulted in an almost perfect success rate of $\hat{p}^* = 0.9995$ with 99% CI $[0.9963, 1.0]$, still with a coverage of $\hat{\rho} = 1.0$, at the cost of a runtime increase to 5.6 min. We

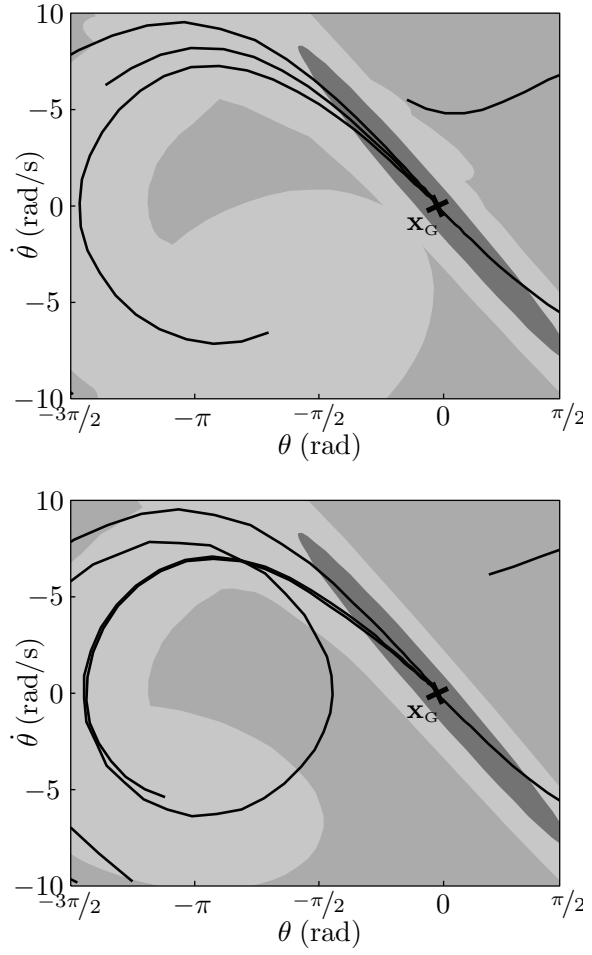

Figure 6. Simple pendulum tree-policy examples. The state-space shown corresponds to the design set $\mathcal{D}$. The funnel hypotheses are medium-gray, and for both examples, we highlight the final funnel hypothesis of a single trajectory in light gray. The goal set $\hat{\mathcal{G}}(\hat{\epsilon}_G)$ is shown in dark gray. *Top*: Policy with 198 nodes in 4 trajectories, generated in 4328 iterations that took 1.3 min; coverage estimate $\hat{\rho} = 1.0$ with 99% confidence interval (CI) $[0.997, 1.0]$; success rate estimate $\hat{p}^* = 0.996$ with 99% CI $[0.991, 0.999]$. *Bottom*: Policy with 238 nodes in 4 trajectories, generated in 4916 iterations that took 1.4 min; coverage estimate $\hat{\rho} = 1.0$ with 99% CI $[0.997, 1.0]$; success rate estimate $\hat{p}^* = 0.994$ with 99% CI $[0.987, 0.997]$.

would argue that the small increase in success rate does not justify the increase in runtime, since we find that in experiments, discussed below, a more likely source of failures are modeling errors.

Finally, the second example in Fig. 6 illustrates that the motion-planning method we use may add suboptimal trajectories to the tree-policy; see the state-trajectory that crosses itself. While the suboptimal trajectories do not impact the quality of the policy in terms of coverage and success rate, they result in "inelegant" system behavior; see also Experiment 111 with the cart-pole shown in Extension 2. An approach to improve the system behavior is to seed the algorithm with carefully designed, possibly optimal trajectories from typical initial conditions of the system; a similar approach is used for seeding an algorithm that generates a trajectory library in Stolle and Atkeson (2010).

A MATLAB function that animates simulations with the generated policy, and the files used to generate the policies can be found in Extension 1.

6.2 Experimental Example: Cart-Pole Swing-Up

The state-constraints capability of simulation-based LQR-Tree policies is demonstrated in experiments with a cart-pole system, sketched in Fig. 7. The task is to swing up the pole from a wide range of initial conditions to the goal state, where the cart is balancing the pole upright at the center of the rail. The rail of the experimental platform is limited, which imposes a state constraint on the cart position that the algorithm explicitly takes into account.

In order to demonstrate the application of tree-policies to highly dynamic systems, we choose motion-planning cost matrices $\mathbf{Q}, \mathbf{R}$ that result in a fast swing-up of the pendulum: The mean swing-up time, measured by the average duration of the open-loop trajectories in the tree, is just 1.75 s, compared to the period of about 1 s of the linearized pendulum dynamics. The speed of the swing-up maneuvers can also be observed in Extension 2. The continuous-time dynamics of the cart-pole are

$$\ddot{\xi} = \frac{u + \mathrm{m_P} \sin\theta \left(\mathrm{g}\cos\theta - \mathrm{l}\dot{\theta}^2\right)}{\mathrm{m_C} + \mathrm{m_P}(1 - \cos^2\theta)} \qquad (42)$$

$$\ddot{\theta} = \frac{\cos\theta \left(u - \mathrm{l}\,\mathrm{m_P}\dot{\theta}^2 \sin\theta\right) + \mathrm{g}\sin\theta\,(\mathrm{m_C} + \mathrm{m_P})}{\mathrm{l}\,(\mathrm{m_C} + \mathrm{m_P}(1 - \cos^2\theta))}$$

where the cart position is ξ (m) and the pendulum angle is θ (rad), and their respective velocities are $\dot{\xi}$ (m/s) and $\dot{\theta}$ (rad/s). The system state is $\mathbf{x} := (\xi, \theta, \dot{\xi}, \dot{\theta})$. Analogous to the simple pendulum, we did not consider wrap-around of the pendulum angle for simplicity. The model parameters for the experimental setup are: $\mathrm{m_C} = 1.5\,\mathrm{kg}$, $\mathrm{m_P} = 0.175\,\mathrm{kg}$, $\mathrm{l} = 0.28\,\mathrm{m}$, and $\mathrm{g} = 9.8\,\mathrm{m/s^2}$. The goal state and input are $\mathbf{x}_G = \mathbf{0}$ and $u_G = 0$, where the cart is at rest at the center of the rail and the pendulum is at rest, pointing up. The position is constrained to $|\xi| \leq 0.45\,\mathrm{m}$ by the limited rail. Friction in the pendulum joint is not modeled.

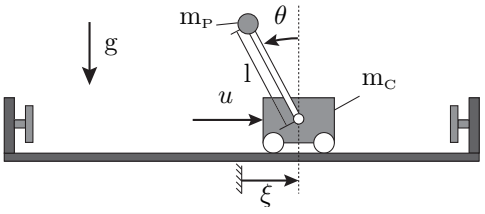

Figure 7. The cart-pole.

The control input u (N) acts on the cart in ξ-direction and is limited to $|u| \leq 60\,\mathrm{N}$. The box-shaped design set $\mathcal{D}$ is defined by the bounds $|\xi| \leq 0.25\,\mathrm{m}$, $|\dot{\xi}| \leq 2\,\mathrm{m/s}$, $-3\pi/2 \leq \theta \leq \pi/2$, and $|\dot{\theta}| \leq \sqrt{4\mathrm{g}/\mathrm{l}}$. The interval for $\dot{\theta}$ is derived from the maximal angular velocity that the pendulum reaches if the cart is fixed and the pendulum swings freely from its upright equilibrium.

6.2.1 Experimental Setup: The experimental setup is shown in Extension 2. The cart is actuated by a timing belt driven by a geared DC motor. The pendulum consists of a thin, hollow pole with a mass attached to its tip. Encoders at the pendulum's and the DC motor's pivot are used to measure the pendulum angle and cart position, respectively. Both the TV LQR trajectory policy and the TI LQR goal policy require full-state feedback. We estimate the velocities $\dot{\xi}$ and $\dot{\theta}$ from the position ξ and angle θ by numerical differentiation and filtering with a second-order, finite-impulse-response filter. The query-phase computations are implemented in MATLAB, and the resulting TV LQR node-policy is transferred to a dSpace DS1104 control-prototyping system. The dSpace system then applies the node-policy and activates the TI LQR policy after the finite-time node-policy ends, see Section 5.7. If the cart hits either bound of the rail, a limit switch cuts off the power to the DC motor to avoid damage.

6.2.2 TI LQR Design and Goal Set: The TI LQR policy is designed according to Section 5.2, with $\bar{\tau} = 0.01\,\mathrm{s}$, $\mathbf{Q}_\mathrm{G} = \mathrm{diag}(5000, 50, 0.5, 5)$, and $\mathbf{R}_\mathrm{G} = 0.1$. The specific values for $\mathbf{Q}_\mathrm{G}$ and $\mathbf{R}_\mathrm{G}$ were obtained by manually tuning the policy on the experimental setup. The goal set was determined with the same termination condition as for the simple pendulum ($\mathrm{M} = 459$), and was approximated in less than 10 s.

6.2.3 Motion Planning and TV LQR Design: Motion planning and trajectory stabilization are implemented as described in Section 5.4 and Section 5.5. Similar to the TI LQR design, we tuned the matrices $\mathbf{Q}$ and $\mathbf{R}$ of the TV LQR policy on the experimental system. First, we used motion planning to find a swing-up trajectory from the lower equilibrium at $\mathbf{x} = (0, -\pi, 0, 0)$, and then manually tuned $\mathbf{Q}$ and $\mathbf{R}$ to achieve acceptable tracking performance. We found that $\mathbf{Q} = \mathrm{diag}(1000, 300, 1000, 100)$ and $\mathbf{R} = 0.1$ work well.

Motion planning uses the same cost matrices as the TV LQR design. The low value for $\mathbf{R}$ results in swing-up trajectories that are of short duration, see the discussion above. We used conservative input and state constraints: $|\xi^\mathrm{mp}| \leq 0.36\,\mathrm{m}$ and $u^\mathrm{mp}_\mathrm{max} = 36\,\mathrm{N}$. The stricter constraints leave more control authority for the TV LQR policy to stabilize trajectories, and should result in larger funnels. Finally, in order to obtain homogeneous sampling times of the trajectories for the TV LQR design, the trajectories were resampled to $\bar{\tau} = 0.01\,\mathrm{s}$ according to Section 5.4.

6.2.4 LQR-Tree Policy: The swing-up trajectory used to tune the TV LQR policy was re-optimized using the motion planner and the values determined for $\mathbf{Q}$ and $\mathbf{R}$. The trajectory was then used as initial seed the tree-policy, as discussed in Section 6.1.3. For the termination heuristic, we again used $\alpha = 0.01$ and $\bar{\mathrm{p}}_\alpha = 0.99$, which results in $\mathrm{M} = 459$.

The algorithm terminated after 95 595 iterations in 3.9 h, generating a tree-policy with 40 137 nodes in 218 trajectories. We assessed the generated tree-policy with $N_s = 2000$ random initial conditions drawn uniformly from $\mathcal{D}$, according to Section 4.5.2. Success was determined by checking for convergence to the goal state after simulating the system for another 3 s after the TV LQR policy ends and the goal TI LQR policy is activated. We obtained a coverage ratio estimate of $\hat{\rho} = 1.0$ with 99% confidence interval (CI) $[0.997, 1]$. Since the success rate estimate was just $\hat{p}^* = 0.991$ with 99% CI $[0.985, 0.996]$, we further refined the tree-policy with a lower $\alpha' = 0.008$, resulting in $\mathrm{M}' = 481$. The algorithm terminated after a cumulative total of 127 983 iterations in a cumulative total of 4.8 h, generating a policy with 45 054 nodes in 245 trajectories. The refined policy still achieves $\hat{\rho} = 1.0$ and a higher $\hat{p}^* = 0.998$ with 99% CI $[0.993, 0.999]$. All of the five failed stabilizations in the assessment are due to violated state-constraints, i.e. were not due to a poor approximation of $\mathcal{G}$ to the goal-state basin of attraction.

In Fig. 8, the evolution of the number of trajectories in the tree, and the time spent testing a sample $\mathbf{x}_\mathrm{S}$ with candidate node-policies is shown. The average time for testing a sample was 0.07 s, and the data shown in Fig. 8 suggests that the expected time to test a sample remains roughly constant even as the number of nodes in the tree increases. This highlights the efficiency that the funnel hypotheses add to the approach by limiting the number of node-policies that have to be tested. The spikes in testing time typically occur when a new

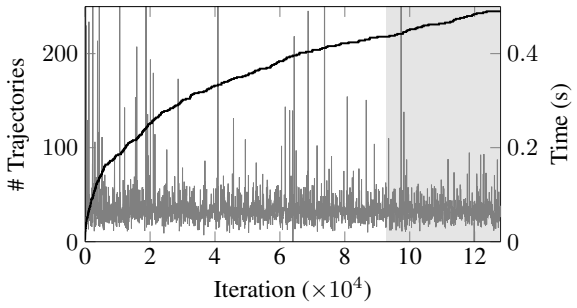

Figure 8. Number of trajectories (black) in the tree-policy for the state-constrained cart-pole and time spent testing a sample $\mathbf{x}_S$ per iteration (gray). The light gray region indicates the policy-refinement phase.

trajectory is added to the tree and many funnels are shrunk.

A MATLAB script that animates simulations with the tree-policy is available in Extension 1, which also contains MATLAB code that allows reproducing the policy.

6.2.5 Experiment Procedure: We generate initial conditions from $\mathcal{D}$ on the experimental setup either by manually exciting the pendulum with the cart at rest at different positions, or, in order to achieve nonzero cart velocities, by applying a step or sinusoidal open-loop input. The policy is activated manually, upon which the tree is queried for the node $\mathcal{N}^*(\mathbf{x}_{\mathrm{IC}})$ based on the initial condition $\mathbf{x}_{\mathrm{IC}}$, and the node-policy of $\mathcal{N}^*(\mathbf{x}_{\mathrm{IC}})$ is applied according to Section 5.7. We deal with initial conditions for which there does not exist an $\mathcal{N}^*(\mathbf{x}_{\mathrm{IC}})$ by applying the policy of the node that minimizes (10), ignoring the funnel-constraint (i.e. the "closest" node to $\mathbf{x}_{\mathrm{IC}}$ based on the TV LQR cost-to-go).

Finding $\mathcal{N}^*(\mathbf{x}_{\mathrm{IC}})$ and transferring its TV LQR policy from MATLAB to the dSpace system takes some time. This delay causes a difference between the measured initial condition $\mathbf{x}_{\mathrm{ICM}}$ and the actual initial condition of the system. We compensate for this by extrapolating $\mathbf{x}_{\mathrm{ICM}}$ by an empirically determined delay time. For the tree-policy with about 45 000 nodes, we used a delay of $\tau_{\mathrm{C}} = 0.017\,\mathrm{s}$ and obtained the extrapolated state $\mathbf{x}_{\mathrm{ICE}}$ from $\mathbf{x}_{\mathrm{ICM}}$ with the standard fourth-order Runge-Kutta method. The extrapolation is then used to find $\mathcal{N}^*(\mathbf{x}_{\mathrm{ICE}})$. The delay compensation improves the performance, since the extrapolated $\mathbf{x}_{\mathrm{ICE}}$ is typically closer to the actual initial condition when the TV LQR policy is activated. This is illustrated in Fig. 10. A single experiment is successful if the time-invariant (TI) LQR controller can stabilize the cart-pole at the goal state. Specifically, success is determined by measuring the average pendulum angle over 10 samples

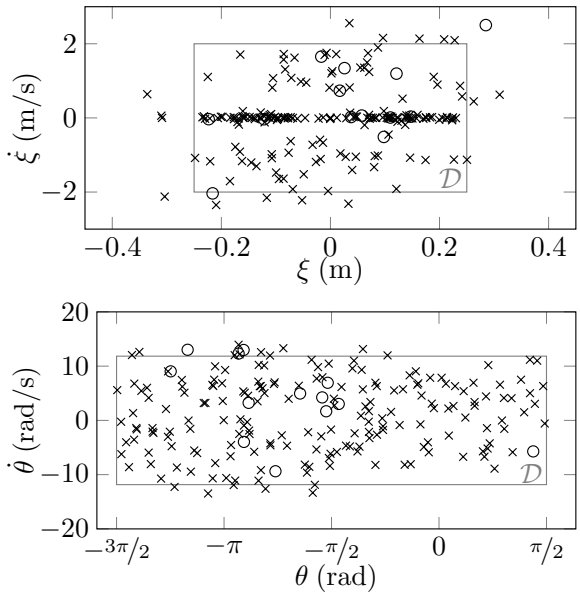

Figure 9. Extrapolated initial conditions of all 200 experiments. The gray rectangle represents the design set $\mathcal{D}$. Crosses indicate success while circles indicate failure. 34 initial conditions were not in $\mathcal{D}$. The success rate was 94.6% for the initial conditions in $\mathcal{D}$.

after three seconds have elapsed after the TI LQR policy is activated.

6.2.6 Results: We performed 200 experiments, of which 187 were successful. This results in a success rate of 93.5% with 95% CI [89.1%, 96.5%]. In 34 experiments, the extrapolated initial condition $\mathbf{x}_{\mathrm{ICE}}$ was outside the design region $\mathcal{D}$. If we discard these experiments, we find that 157 of 166 experiments were successful, which results in a success rate of 94.6% with 95% CI [90.0%, 97.5%]. In Fig. 9, all extrapolated initial conditions $\mathbf{x}_{\mathrm{ICE}}$ of the experiments are shown. In 102 of the total 200 experiments, a nominal trajectory was selected that was not used in any other experiment. In Fig. 10 , we show the phase plots of a successful experiment. Extension 2 is a video of selected experiments, and all measurement data and scripts to visualize and analyze the data are available in Extension 1.

We argue that the main reason for failures are modeling errors (for example, the unmodeled friction in the pendulum joint), which is supported by the difference between the experimental success rate and the almost perfect success rate obtained in simulation. The effect of modeling errors can also be observed in the phase plot of an experiment shown in Fig. 10: Towards the end, the cart has to perform a large corrective maneuver to keep the system state close

to the nominal trajectory. These corrective maneuvers can lead to the cart hitting the system bounds and activating the limit switches, which was observed in most failed experiments. Another contribution may be from extrapolation errors, i.e. the difference between the extrapolated initial condition $\mathbf{x}_{\text{ICE}}$ and the actual initial condition when the node-policy of $\mathcal{N}^*(\mathbf{x}_{\text{ICE}})$ is activated. We found that in 174 of the 200 experiments, a different node-policy than $\mathcal{N}^*(\mathbf{x}_{\text{ICE}})$ would be applied to the initial condition measured at policy-activation.

While the tree-policy appears to be quite robust against the modeling and extrapolation errors, the performance may be further improved by performing a more advanced system identification and regenerating the tree-policy, and by using a more sophisticated state observer that should lead to better extrapolations and control-performance. Another opportunity to pursue is the improvement of the tree-policy based on experimental data: Possible approaches are adjusting the funnels and growing the tree after a failed experiment, similar to the ideas presented in Stolle and Atkeson (2010) for adapting trajectory libraries; or iterative learning control, see, for example Schoellig et al. (2012), where the open-loop control input is adjusted based on experimental data, such that the nominal state trajectory is more accurately followed.

6.3 Remarks about Scalability

The significant increase in runtime and number of tree-nodes required to generate a policy for the cart-pole compared to the simple pendulum raises concerns about the scalability of the approach, i.e. the applicability to higher-dimensional problems. However, it is not straightforward to compare the runtimes and number of tree-nodes for different systems, as several factors besides the problem dimension influence them.

In the cart-pole example, a significant portion of both the runtime and the number of nodes can be attributed to the state-constraints: A tree-policy with identical parameters and quality measures, except with the state-constraints removed, can be generated in 2.6 h (−45%) and contains just 13 298 nodes (−70%) in 76 trajectories. Furthermore, the homogeneous sampling time can also have a significant impact on the number of nodes, and should not be chosen unreasonably low: When we halved the sampling time to $\tau = 0.005\,\text{s}$, we obtained, not surprisingly, about twice as many tree nodes. Therefore, the systems' time-constants and required control-bandwidth factor in the number of nodes and runtime. Related, if the feedback policies stabilizing the trajectories only have limited control authority due to system properties such as tight input constraints or poor local controllability,

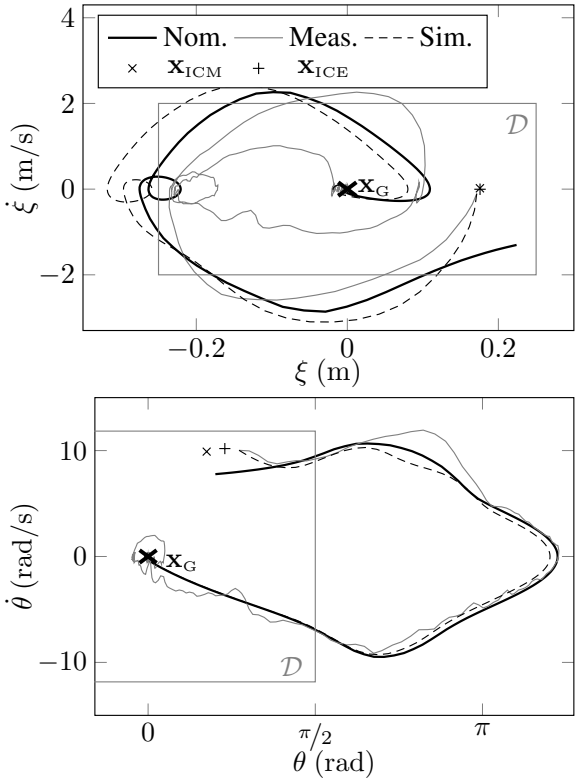

Figure 10. Phase plots of a successful experiment (Exp. #93). The extrapolated initial condition $\mathbf{x}_{\text{ICE}}$ is closer to the actual initial condition (beginning of gray and dashed lines) than the measured initial condition $\mathbf{x}_{\text{ICM}}$. Also shown is the trajectory obtained in simulation with the actual initial condition, and the box-shaped design set $\mathcal{D}$.

the resulting funnels are small and more nodes and runtime are required to generate a policy, see also the discussions in Section 2.4.4 and Section 5.4.

Another observation is that the total time spent testing samples was about 2.6 h, i.e. the algorithm spent about 45% of the runtime in motion planning. Furthermore, only about 30% of motion-planning attempts succeeded (i.e. 468 attempts failed). The failure-rate was likely due to poor initializations of the motion-planner, given the high coverage that the tree-policy ultimately achieved. A fast motion planner that is robust to poor initialization is therefore desirable. Furthermore, it may be beneficial to design the planner to fail quickly, for example by using low SQP-iteration limits in the method we use, especially when large regions of the design set $\mathcal{D}$ are not stabilizable.

In general, the design set $\mathcal{D}$ should be kept as small as possible, for example derived from typically observed initial conditions of the system. The LQR-Tree approach allows straightforward extension of an existing policy for a larger set $\mathcal{D}$, if required.

Choosing $\mathcal{D}$ unnecessarily large significantly increases the runtime and node-count by covering regions of the state space that may never be visited.

It is remarkable that the tree-policies are represented by just 45 000 or 13 000 nodes. The equivalent resolution of a uniform state-space discretization used in, for example, a standard value-iteration approach (Bertsekas 2005) is about $45\,000^{1/4} \approx 15$ or $13\,000^{1/4} \approx 11$ grid points per dimension, which is quite poor. The algorithm only adds trajectories/nodes where needed, similar to adaptive discretization approaches: The above node numbers are comparable to the numbers obtained with variable-resolution dynamic programming presented in Munos and Moore (2002) (note that the problem setup is different, though).

Both the runtime and node counts are problem-specific, and the scalability of the approach should be further explored using suitable systems that have comparable properties such as constraints, design sets $\mathcal{D}$, and time-constants.

7 Conclusion

We presented a variant of the LQR-Tree algorithm introduced in Tedrake (2009). The key difference to previous work is that stabilizable sets of trajectories are approximated with a simulation-based falsification method, instead of the formal, Lyapunov-function-based verification used in Tedrake (2009). The main advantage is that the approach allows generating tree-policies for a wider range of dynamic systems, and feedback designs for trajectory stabilization. Theoretical results showed that in the long run, the algorithm tends to improve both the coverage of the initial conditions to be stabilized, and the approximations to the stabilizable sets. This result is supported by simulation results, where the tree-policies achieved both high coverage- and stabilization-rates. The tree-policies for the example systems were generated using an example implementation of the algorithm that was described and discussed in detail. The applicability to highly-dynamic and state-constraint systems was demonstrated in experiments with a cart-pole, where a stabilization rate of 93% was achieved in 200 experiments. A comparison to the close to 100% stabilization rate achieved in simulation suggests that a key direction for future work is to develop strategies to adapt tree-policies based on experimental data, e.g. as in Schoellig et al. (2012) or Stolle and Atkeson (2010), in order to further improve experimental performance. Another interesting topic is developing approaches to post-process the generated tree-policies in order to reduce the number of tree-nodes by removing redundant trajectories, or to detect sub-optimal trajectories as shown in Fig. 6 by checking the consistency of a trajectory with nearby tree-trajectories. Finally, the scalability of the approach to higher-dimensional systems can be further analyzed by studying well-scalable systems.

Acknowledgements

The authors would like to thank Sebastian Trimpe, Markus Hehn, and Raffaello D'Andrea for their valuable suggestions and comments, and the anonymous peers for their time and effort spent reviewing this manuscript, and their helpful comments.

Funding

This work was supported by ETH Research Grant ETH-31 11-1.

A Index to multimedia extensions

Ext.	Media	Description
1	Code and Data	MATLAB code of example implementation, scripts to visualize simulations of the example systems, and scripts to analyze and plot the experimental data.
2	Video	Video of selected cart-pole experiments.

B Proofs of Asymptotic Algorithm Properties

B.1 Assumptions

We restate the assumptions underlying the following proofs from Section 2 and Section 3:

1. *Motion planner:* The motion planner is able to find an open-loop trajectory from any state $\mathbf{x} \in \mathcal{D}$ to the goal $\mathcal{G}$, given that $\mathbf{x}$ is stabilizable: $\mathbf{x} \in \mathcal{S}_{\mathcal{D}}$. Note that the following proofs also work with the assumption that the motion planner finds an open-loop trajectory only with some positive probability, which was used in Tedrake et al. (2010).
2. *Stabilizable balls:* For every nominal state $\bar{\mathbf{x}}_k$ in an open-loop trajectory, there exists an open ball centered at $\bar{\mathbf{x}}_k$, in which all states can be stabilized to the goal by the trajectory's feedback policy without violating state constraints.
3. *Probability density function (PDF) of random samples:* The state samples $\mathbf{x}_{\mathrm{S}}$ are drawn i.i.d.

from the design set $\mathcal{D}$ with a PDF that is positive on $\mathcal{D}$.

B.2 Probabilistic Feedback Coverage for the Conceptual Algorithm

The following proof closely follows Sec. 6 in Tedrake et al. (2010), which we adapted to the notation and definitions used in the conceptual algorithm. Let the *coverage set* $\mathcal{C}^i$ be the union of the funnels of the $|\mathcal{T}^i|$ trajectories in the tree at iteration i, $\mathcal{T}^i = \{\mathcal{J}_1, \dots, \mathcal{J}_{|\mathcal{T}^i|}\}$:

$$\mathcal{C}^i := \bigcup_{j=1}^{|\mathcal{T}^i|} \mathcal{S}_{\mathcal{J}_j} \cap \mathcal{D} \tag{43}$$

where the restriction to $\mathcal{D}$ implies $\mathcal{C}^i \subseteq \mathcal{S}_\mathcal{D}$. If a trajectory is added to the tree at iteration i, the coverage set $\mathcal{C}^i$ grows, which implies $\mathcal{C}^i \subseteq \mathcal{C}^{i+1}$. The limit set of the sequence $\mathcal{C}^i$ is

$$\mathcal{C}^\infty := \lim_{i\to\infty} \mathcal{C}^i \tag{44}$$

which is the set of states in $\mathcal{S}_\mathcal{D}$ that is covered by funnels as the number of iterations tends to infinity. The limit exists since the sequence $\mathcal{C}^i$ is ordered by inclusion, $\mathcal{C}^i \subseteq \mathcal{C}^{i+1}$, and, therefore, $\limsup \mathcal{C}^i = \liminf \mathcal{C}^i = \mathcal{C}^\infty$; see Chapter 4 in Rockafellar et al. (1998).

Adapting the definition of probabilistic feedback coverage in Tedrake et al. (2010) to the notation and definitions used here, it states: A feedback-motion-planning algorithm achieves *probabilistic feedback coverage* for the goal $\mathcal{G}$ if $\mathcal{C}^\infty = \mathrm{cl}(\mathcal{S}_\mathcal{D})$ with probability one, where $\mathrm{cl}(\cdot)$ is the closure of a set. Note that $\mathcal{C}^\infty$ is closed by definition (Rockafellar et al. 1998). We show that the conceptual algorithm achieves probabilistic feedback coverage by proving that

Lemma 1. *The set $\mathcal{F} := \mathcal{S}_\mathcal{D} \setminus \mathcal{C}^\infty$, which is the relative complement of $\mathcal{C}^\infty$ in $\mathcal{S}_\mathcal{D}$, has Lebesgue measure zero with probability one.*

We show by contradiction that $\mathcal{F}$ cannot have nonzero measure: Suppose that $\mathcal{F}$ has nonzero measure. Since the random samples used to generate the tree are drawn i.i.d. from a PDF that is positive on $\mathcal{D}$ (Ass. 3), and therefore also positive on $\mathcal{S}_\mathcal{D} \subseteq \mathcal{D}$, the probability of drawing a sample $\mathbf{x}_\mathrm{S} \in \mathcal{F} \subseteq \mathcal{S}_\mathcal{D}$ is nonzero at any iteration i. Therefore, as $i \to \infty$, the probability of drawing a sample $\mathbf{x}_\mathrm{S} \in \mathcal{F}$ tends to one. Since $\mathbf{x}_\mathrm{S} \in \mathcal{S}_\mathcal{D}$ by definition of $\mathcal{F}$, the motion-planning and feedback modules add a new trajectory $\mathcal{J}$ to the tree with nominal initial state $\bar{\mathbf{x}}_0 = \mathbf{x}_\mathrm{S}$ (Ass. 1). The funnel of the added trajectory is a subset of $\mathcal{C}^\infty$ by definition, and has a nonzero measure intersection with $\mathcal{F}$, as there is an open ball around $\bar{\mathbf{x}}_0 \in \mathcal{F}$ that can be stabilized to the goal by the feedback policy of the trajectory (Ass. 2). This contradicts the definition of $\mathcal{F}$, and we conclude that with probability one, the set $\mathcal{F}$ has Lebesgue measure zero. □

B.3 Simulation-Based Algorithm: Coverage

We show that the funnels $\mathcal{S}^n_\mathcal{N}$ of the nodes in the tree cover $\mathcal{S}_\mathcal{D}$ as $i \to \infty$. The interpretation is that in the limit, there exists a stabilizing feedback policy in the tree for all states in $\mathcal{S}_\mathcal{D}$, except possibly a set of states with Lebesgue measure zero. The node funnels $\mathcal{S}^n_\mathcal{N}$ are unknown, but can be tested in simulations: When the policy of node $\mathcal{N}_n$ with funnel hypothesis $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n)$ is applied to a sample $\mathbf{x}_\mathrm{S} \in \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n) \cap \mathcal{S}^n_\mathcal{N}$, the simulation succeeds; and the simulation fails if $\mathbf{x}_\mathrm{S} \in \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n) \setminus \mathcal{S}^n_\mathcal{N}$. In the following, we show that the sets $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n) \cap \mathcal{S}^n_\mathcal{N}$ of the tree nodes cover $\mathcal{S}_\mathcal{D}$ in the limit, which implies that the node funnels cover $\mathcal{S}_\mathcal{D}$. First, we define the set

$$\hat{\mathcal{C}}^i := \bigcup_{n=1}^{V^i} \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}^i_n) \cap \mathcal{S}^n_\mathcal{N} \tag{45}$$

which contains all states in $\mathcal{S}_\mathcal{D}$ for which there exists at least one stabilizing node-policy in the tree $\mathcal{T}^i$ at iteration i. We add an iteration superscript to the funnel hypotheses radii $\hat{\epsilon}^i_n$ to highlight that the radii may change (multiple times) in an iteration due to falsification(s). For the following, it is irrelevant what specific trajectory contains a node $\mathcal{N}_n$; therefore, we omit trajectories and define V^i to be the total number of nodes in $\mathcal{T}^i$.

Coverage of $\mathcal{S}_\mathcal{D}$ by $\hat{\mathcal{C}}^i$ in the limit follows from proving that the relative complement

$$\hat{\mathcal{F}}^i := \mathcal{S}_\mathcal{D} \setminus \hat{\mathcal{C}}^i \tag{46}$$

has measure zero as $i \to \infty$. However, unlike $\mathcal{C}^i$ (43), the sequence $\hat{\mathcal{F}}^i$ is not ordered by inclusion: At every iteration, the algorithm may both increase coverage by adding nodes, and decrease coverage by shrinking funnel hypotheses. Therefore, the existence of the limit of the sequence $\hat{\mathcal{F}}^i$ is not given, and we resort to considering its outer limit

$$\limsup_{i\to\infty} \hat{\mathcal{F}}^i = \bigcap_{n=1}^{\infty} \mathrm{cl}\left(\bigcup_{i=n}^{\infty} \hat{\mathcal{F}}^i\right) \tag{47}$$

which always exists (Rockafellar et al. 1998). The limit superior $\limsup \hat{\mathcal{F}}^i$ can be interpreted as the set of states that appear infinitely often but not necessarily

always in any tail of the sequence $\hat{\mathcal{F}}^i$. We show that $\limsup \hat{\mathcal{F}}^i$ has measure zero with probability one, from which follows that in the limit, the sequence $\hat{\mathcal{F}}^i$ also has measure zero:

Lemma 2. *The outer limit* $\limsup \hat{\mathcal{F}}^i$ *has Lebesgue measure zero with probability one.*

We define $\bar{\mathcal{F}} := \limsup \hat{\mathcal{F}}^i$. The proof is by contradiction: Assume that $\bar{\mathcal{F}}$ has nonzero measure. Since states in $\bar{\mathcal{F}}$ appear infinitely often in tails of $\hat{\mathcal{F}}^i \subseteq \mathcal{D}$, and samples $\mathbf{x}_\mathrm{S}$ are drawn i.i.d. from a PDF positive on $\mathcal{D}$ (Ass. 3), a sample $\mathbf{x}_\mathrm{S} \in \mathrm{int}(\bar{\mathcal{F}})$ is drawn with probability one, where $\mathrm{int}(\cdot)$ denotes the interior of a set. By definition of $\bar{\mathcal{F}}$, the simulation with initial condition $\mathbf{x}_\mathrm{S}$ fails: Either $\mathbf{x}_\mathrm{S}$ is not in any funnel hypothesis, or the policies of all nodes whose funnel hypotheses contain $\mathbf{x}_\mathrm{S}$ fail to stabilize $\mathbf{x}_\mathrm{S}$ to the goal. Since $\mathbf{x}_\mathrm{S} \in \mathcal{S}_\mathcal{D}$ by definition of $\bar{\mathcal{F}}$, the motion-planning and feedback modules add a new trajectory to the tree with nominal initial state $\bar{\mathbf{x}} = \mathbf{x}_\mathrm{S}$ (Ass. 1). Let $\hat{\mathcal{B}}(\bar{\mathbf{x}}, \hat{\epsilon})$ be the funnel hypothesis of the new node at $\bar{\mathbf{x}}$, $\mathcal{S}_\mathcal{N}$ its funnel, and $\mathcal{B} \subseteq \mathcal{S}_\mathcal{N}$ the stabilizable ball at $\bar{\mathbf{x}}$ (Ass. 2). The intersection $\hat{\mathcal{B}}(\bar{\mathbf{x}}, \hat{\epsilon}) \cap \mathcal{B}$ is a lower, nonzero measure bound on $\hat{\mathcal{B}}(\bar{\mathbf{x}}, \hat{\epsilon}) \cap \mathcal{S}_\mathcal{N}$, and all states in $\hat{\mathcal{B}}(\bar{\mathbf{x}}, \hat{\epsilon}) \cap \mathcal{B}$ can be stabilized to the goal. Since $\bar{\mathbf{x}} = \mathbf{x}_\mathrm{S} \in \bar{\mathcal{F}}$, $\hat{\mathcal{B}}(\bar{\mathbf{x}}, \hat{\epsilon}) \cap \mathcal{B}$ has a nonzero measure intersection with $\bar{\mathcal{F}}$, which contradicts the definition of $\bar{\mathcal{F}}$. We conclude that with probability one, the set $\bar{\mathcal{F}}$ has Lebesgue measure zero. □

B.4 Simulation-Based Algorithm: Policy Assignment

We show that as $i \to \infty$, the assignment of a state $\mathbf{x} \in \mathcal{D}$ to a node $\mathcal{N}_n$ by the assignment rule (10), i.e. $\mathcal{N}^*(\mathbf{x}) = \mathcal{N}_n$, implies that $\mathbf{x}$ can be stabilized to the goal by the policy of $\mathcal{N}_n$. At iteration i, we define the set of states $\mathbf{x} \in \mathcal{D}$ that are assigned to node $\mathcal{N}_n$, and that cannot be stabilized to the goal by the node-policy of $\mathcal{N}_n$ as

$$\mathcal{F}_n^i := \left\{ \mathbf{x} \in \mathcal{D} \cap \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n^i) \setminus \mathcal{S}_\mathcal{N}^n : \mathcal{N}^*(\mathbf{x}) = \mathcal{N}_n \right\}. \quad (48)$$

The sequence $\mathcal{F}_n^i$ is not ordered by inclusion due to the interaction of the funnel hypotheses in the node assignment: The sets $\mathcal{F}_n^i$ shrink and expand due to falsifications and changing policy assignments. For example, a node's funnel hypothesis that overlaps with the hypothesis of $\mathcal{N}_n$ may shrink in an iteration such that states are newly assigned to $\mathcal{N}_n$ that are not stabilizable by the policy of $\mathcal{N}_n$, see Fig. 3 and the discussion in Section 4.3.1. Therefore, the limit of $\mathcal{F}_n^i$ may not exist and we show, like above, that the outer limit $\limsup \mathcal{F}_n^i$ has measure zero with probability one, which implies that in the limit, the sequence $\mathcal{F}_n^i$ also has measure zero:

Lemma 3. *The outer limit* $\limsup \mathcal{F}_n^i$ *has Lebesgue measure zero with probability one.*

We define $\bar{\mathcal{F}}_n := \limsup \mathcal{F}_n^i$. The proof is by contradiction: Assume that $\bar{\mathcal{F}}_n$ has nonzero measure. Since states in $\bar{\mathcal{F}}_n$ appear infinitely often in tails of $\mathcal{F}_n^i \subseteq \mathcal{D}$, and the samples $\mathbf{x}_\mathrm{S}$ are drawn i.i.d. from a PDF positive on $\mathcal{D}$ (Ass. 3), a sample $\mathbf{x}_\mathrm{S} \in \mathrm{int}(\bar{\mathcal{F}}_n)$ is drawn with probability one. By definition of $\bar{\mathcal{F}}_n$, the simulation with initial condition $\mathbf{x}_\mathrm{S}$ fails and the funnel hypothesis of $\mathcal{N}_n$ is shrunk from $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n^-)$ to $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n^+)$ with $\hat{\epsilon}_n^+ < \hat{\epsilon}_n^-$, where $-, +$ denote pre- and post-shrinking, respectively. The shrinking makes $\mathbf{x}_\mathrm{S}$ a boundary point of $\hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n^+)$, and the set $\mathrm{int}(\bar{\mathcal{F}}_n) \setminus \hat{\mathcal{B}}(\bar{\mathbf{x}}_n, \hat{\epsilon}_n^+)$ has nonzero measure. This contradicts the definition of $\bar{\mathcal{F}}_n$, since all states in $\mathrm{int}(\bar{\mathcal{F}}_n)$ must be inside the funnel hypothesis of $\mathcal{N}_n$. We conclude that with probability one, the set $\bar{\mathcal{F}}_n$ has Lebesgue measure zero. □

References

Agha-mohammadi, A.-a., Chakravorty, S. and Amato, N. M. (2014), 'FIRM: Sampling-based Feedback Motion Planning Under Motion Uncertainty and Imperfect Measurements', *Int. J. Rob. Res.* **33**(2), 268–304.

Anderson, B. and Moore, J. (2007), *Optimal Control: Linear Quadratic Methods*, Prentice-Hall.

Bertsekas, D. P. (2005), *Dynamic Programming and Optimal Control, Vol. I*, Athena Scientific.

Betts, J. (2010), *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, 2nd edn, Siam.

Burridge, R. R., Rizzi, A. A. and Koditschek, D. E. (1998), 'Sequential Composition of Dynamically Dexterous Robot Behaviors', *Int. J. Rob. Res.* **18**(6), 534–555.

Clopper, C. and Pearson, E. (1934), 'The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial', *Biometrika* **26**(4), 404–413.

Colledanchise, M. and Ogren, P. (2014), How Behavior Trees Modularize Robustness and Safety in the Hybrid Control Systems of Computer Games, *in* 'Proc. IEEE/RSJ Int. Conf. Intell. Robot. Sys.', pp. 1482–1488.

Franklin, G., Powell, J. and Workman, M. (1998), *Digital Control of Dynamic Systems*, Vol. 3, Addison-Wesley.

Frazzoli, E., Dahleh, M. a. and Feron, E. (2002), 'Real-Time Motion Planning for Agile Autonomous Vehicles', *J. Guid. Control. Dyn.* **25**(1), 116–129.

Garcia, C. E., Prett, D. M. and Morari, M. (1989), 'Model Predictive Control: Theory and Practice – a Survey', *Automatica* **25**(3), 335–348.

Gill, P. E., Murray, W. and Saunders, M. A. (2002), 'SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization', *SIAM J. Optim.* **12**(4), 979–1006.

Gillula, J. H., Hoffmann, G. M., Vitus, M. P. and Tomlin, C. J. (2011), 'Applications of Hybrid Reachability Analysis to Robotic Aerial Vehicles', *Int. J. Rob. Res.* **30**(3), 335–354.

Gillula, J. H., Kaynama, S. and Tomlin, C. J. (2014), Sampling-based Approximation of the Viability Kernel for High-Dimensional Linear Sampled-data Systems, *in* 'Proc. 17th Int. Conf. Hybrid Syst. Comput. Control', pp. 173–182.

Glassman, E. and Tedrake, R. (2010), 'A Quadratic Regulator-Based Heuristic for Rapidly Exploring State Space', *Proc. IEEE Int. Conf. Robot. Autom.* pp. 5021–5028.

Isla, D. (2005), Handling Complexity in the Halo 2 AI, *in* 'Proc. Game Dev. Conf.'.

Karaman, S. and Frazzoli, E. (2011), 'Sampling-Based Algorithms for Optimal Motion Planning', *Int. J. Rob. Res.* **30**(7), 846–894.

Kavraki, L., Svestka, P., Latombe, J. and Overmars, M. (1996), 'Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces', *Robot. Autom. IEEE Trans.* **12**(4), 566–580.

Kuffner, J., Nishiwaki, K., Kagami, S., Inaba, M. and Inoue, H. (2001), 'Motion Planning for Humanoid Robots Under Obstacle and Dynamic Balance Constraints', *Proc. IEEE Conf. Robot. Autom.* pp. 692–698.

LaValle, S. M. (2006), *Planning Algorithms*, Cambridge University Press.

Levine, S. and Koltun, V. (2013), Guided Policy Search, *in* 'Proc. Int. Conf. Mach. Learn.', pp. 1–9.

Liu, C., Atkeson, C. G. and Su, J. (2013), 'Biped Walking Control Using a Trajectory Library', *Robotica* **31**(2), 311–322.

Majumdar, A., Ahmadi, A. A. and Tedrake, R. (2013), Control Design Along Trajectories with Sums of Squares Programming, *in* 'Proc. IEEE Int. Conf. Robot. Autom.', pp. 4054–4061.

Manchester, I., Tobenkin, M., Levashov, M. and Tedrake, R. (2011), Regions of Attraction for Hybrid Limit Cycles of Walking Robots, *in* 'Proc. IFAC World Congr.', pp. 5801–5806.

Mason, M. (1985), The Mechanics of Manipulation, *in* 'Proc. IEEE Int. Conf. Robot. Autom.', pp. 544–548.

Moore, J., Cory, R. and Tedrake, R. (2014), 'Robust Post-Stall Perching with a Simple Fixed-Wing Glider Using LQR-Trees', *Bioinspir. Biomim.* **9**(2), 025013.

Mordatch, I. and Todorov, E. (2014), Combining the Benefits of Function Approximation and Trajectory Optimization, *in* 'Robot. Sci. Syst.'.

Munos, R. and Moore, A. (2002), 'Variable Resolution Discretization in Optimal Control', *Mach. Learn.* **49**(2-3), 291–323.

Parrilo, P. A. (2003), 'Semidefinite Programming Relaxations for Semialgebraic Problems', *Math. Program.* **96**(2), 293–320.

Plaku, E., Bekris, K., Chen, B., Ladd, A. and Kavraki, L. (2005), 'Sampling-Based Roadmap of Trees for Parallel Motion Planning', *IEEE Trans. Robot.* **21**(4), 597–608.

Reist, P. and Tedrake, R. (2010), Simulation-Based LQR-Trees with Input and State Constraints, *in* 'Proc. IEEE Conf. Robot. Autom.', pp. 5504–5510.

Rockafellar, R., Wets, R. and Wets, M. (1998), *Variational Analysis*, Vol. 317, Springer.

Schoellig, A. P., Mueller, F. L. and D'Andrea, R. (2012), 'Optimization-Based Iterative Learning for Precise Quadrocopter Trajectory Tracking', *Auton. Robots* **33**, 103–127.

Shkolnik, A. and Tedrake, R. (2009), Path Planning in 1000+ Dimensions Using a Task-Space Voronoi Bias, *in* 'Proc. IEEE Conf. Robot. Autom.', pp. 2061–2067.

Stolle, M. and Atkeson, C. (2010), 'Finding and Transferring Policies Using Stored Behaviors', *Auton. Robots* **29**, 169–200.

Strogatz, S. H. (2014), *Nonlinear Dynamics and Chaos*, 2nd edn, Perseus Books Group.

Sun, H. and Farooq, M. (2002), Note on the Generation of Random Points Uniformly Distributed in Hyper-Ellipsoids, *in* 'Proc. Int. Conf. Inf. Fusion', pp. 489–496.

Sutton, R. S. and Barto, A. G. (1998), *Reinforcement Learning: An Introduction*, The MIT Press.

Tedrake, R. (2009), LQR-trees: Feedback Motion Planning on Sparse Randomized Trees, *in* 'Proc. Robot. Sci. Syst.'.

Tedrake, R., Manchester, I. R., Tobenkin, M. and Roberts, J. W. (2010), 'LQR-Trees: Feedback Motion Planning via Sums-of-Squares Verification', *Int. J. Rob. Res.* **29**(8), 1038–1052.

Topcu, U., Packard, A. and Seiler, P. (2008), 'Local Stability Analysis Using Simulations and Sum-of-Squares Programming', *Automatica* **44**(10), 2669–2675.

Yang, L. and LaValle, S. M. (2004), 'The Sampling-Based Neighborhood Graph: An Approach to Computing and Executing Feedback Motion Strategies', *IEEE Trans. Robot. Autom.* **20**(3), 419–432.