Simulation-Based LQR-Trees with Input and State Constraints

Philipp Reist and Russ Tedrake

Abstract— **We present an algorithm that probabilistically covers a bounded region of state space with a sparse tree of feedback stabilized trajectories leading to a goal state. The generated tree serves as a lookup table control policy to get any initial condition within the bounded region to the goal state. The approach combines motion planning with reasoning about the set of states around the trajectories for which the feedback policy of the trajectory is able to stabilize the system. We propose approximating this set by sampling and simulation, which allows enforcing input and state constraints. By taking the approximated sets into account, we only add trajectories to the tree where needed. In practice, we obtain trees of trajectories with a low number of nodes compared to state space discretizing methods. We show simulation results obtained with model systems and analyze the performance and robustness of the algorithm.**

I. INTRODUCTION

The proposed algorithm bases on the ideas presented in [1]. By combining motion planning and formally verifying the stabilizable set around a trajectory, the algorithm in [1] produces a sparse tree of locally optimal trajectories that can bring any initial condition of a bounded region of state space to a goal state. Each trajectory in the tree has an associated time varying linear quadratic regulator (TVLQR) policy. For each trajectory, the set of states which can be stabilized by the linear controller is verified by a formal method, using a sum of squares (SoS) optimization [2]. Reasoning about the coverage of a trajectory allows to only generate new trajectories where needed, resulting in a sparse tree.

The tree is generated offline and can be used to control nonlinear dynamic systems (e.g. robotic balancing tasks, walking and flying robots, etc.) in real time as a lookup table policy. The class of algorithms proposed here and in [1] aim at generating control policies for complicated nonlinear systems when a linear controller is insufficient or the number of states prohibits the application of discretization-based methods like policy iteration.

In contrast to [1], we replace the formal verification of the stabilizable set with SoS around a given trajectory by approximating it using sampling and simulation. The formal approach can take input constraints into account, but the approximation becomes too conservative; also, it is not straightforward to incorporate state constraints in the formal approach. However, both state and input constraints are present in robotic systems; using simulation, the algorithm generates a tree enforcing these constraints.

We only replace the estimation along the trajectories by the simulation approach; we still verify the approximation to the basin of attraction of the goal state using formal methods.

P. Reist and R. Tedrake are with CSAIL at MIT, Cambridge MA, USA.

We obtain the set of states around the goal state for which the nonlinear closed loop system is asymptotically stable from this approximation.

It is remarkable that even though we use a simulation-based method to estimate the stabilizable set around the trajectories, we can still maintain all the probabilistic coverage guarantees that are obtained using the formal method. The coverage mechanism remains the same for both the formal and the sampling approach.

For this algorithm, the notion of a stabilizable set is more general than asymptotic convergence. We not only approximate the set of states around a trajectory that can be brought to the goal, but also make sure that the states in this set reach the goal without violating state and input constraints. In the following, we use a more general term for this set and call it the 'funnel' of a trajectory (inspired by [3] and [4]).

Assume a tree consisting of a set of TVLQR stabilized trajectories leading to a goal state. Each node of this tree consists of a nominal state belonging to one of the trajectories, the respective TVLQR control policy and a local description of the funnel. The proposed algorithm works as follows: First, we generate a random sample from the subspace $\mathcal{R}$ we want to cover. Next, we find the nodes of the tree whose local description of the funnel contain the sample. Each node provides the starting point of a time-varying nominal trajectory and feedback policy. We simulate the system using these policies while applying input saturation from the random sample as initial condition. A simulation is successful if the sample reaches the approximated basin of the goal state without violating any state constraints. If the simulation is successful, we proceed to the next sample. If the simulation fails, we shrink the funnel of the nominal trajectory we executed to not include the trajectory generated in the simulation. If the sample never reaches the goal basin with the existing policies, we use motion planning to connect the failed sample to the tree. We generate the TVLQR policy for the newly generated nominal trajectory and add it to the tree, initializing it's funnel to cover the whole subspace $\mathcal{R}$. If motion planning fails to find a feasible trajectory to connect the sample to the tree, we determine the sample to be (temporarily) unreachable. The algorithm terminates after a fixed number of random samples consecutively reached the goal using the existing tree, ignoring samples that are not reachable.

In the following, we first review related work and compare it to our approach. Next, we present the key concepts of the algorithm and a detailed description of the algorithm's different steps. Finally, we show the performance of the

algorithm in simulation on two model systems.

A. Related Work

The algorithm is inspired by randomized motion planning, i.e. RRTs [5]. Randomized motion planning demonstrated that it scales well to higher dimensional systems, sacrificing optimality but generating feasible motion plans.

The approach is similar to the work of Atkeson [6], who uses trajectories as a sparse representation of the global value function of a system. The authors propose building a library of trajectories leading to a goal state and approximating the global value function with quadratic models along these trajectories. Trajectories are added based on how well two adjacent trajectories agree on the value function in between them. The resulting policy is not time based like in our approach, but transformed to a state dependent policy using a nearest neighbor lookup. The policy of the closest state in the trajectory library is executed, where the closeness is measured using a weighted Euclidean norm [7].

In contrast, the proposed algorithm aims not at producing globally optimal trajectories or estimating the global value function; we are more interested in designing a scalable algorithm that yields feasible policies.

As Atkeson points out, an advantage of using trajectories to represent a policy is that one avoids the problems that a discretization of state space generates. It is remarkable that in practice, we realized policies for the cart-pole (4 states) swing-up with trees of between 20000 and 200000 nodes. The equivalent resolution of a state space discretization would be between 12 and 21 grid points per dimension, which is quite poor.

II. Key Concepts

A. Discrete-Time TVLQR

We consider sampled-data feedback control of continuous time dynamical systems. Therefore, we derive the controllers and goal state basin of attraction directly in discrete-time. The sampled trajectories stored in the tree are stabilized using a time varying linear quadratic regulator (TVLQR). In the following, we derive the optimal feedback policy and cost-to-go. The system dynamics of a smoothly differentiable system are given by

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), \quad (1)$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state of the system and $\mathbf{u} \in \mathbb{R}^m$ is the input to the system. Given a sampled nominal trajectory

$$\mathbf{x}_{0k}, \mathbf{u}_{0k}, \quad k = 0, 1, ..., N, \quad (2)$$

with $\mathbf{x}_{0k} = \mathbf{x}_0 (t = k \cdot dt)$, dt is the sample time and $N+1$ is the number of elements. We linearize and discretize the system (1) around this trajectory to obtain the discrete linear system dynamics

$$\bar{\mathbf{x}}_{k+1} = \mathbf{A}_k \bar{\mathbf{x}}_k + \mathbf{B}_k \bar{\mathbf{u}}_k, \quad (3)$$

with

$$\begin{aligned} \bar{\mathbf{x}}_k &= \mathbf{x}_k - \mathbf{x}_{0k} \\ \bar{\mathbf{u}}_k &= \mathbf{u}_k - \mathbf{u}_{0k}. \end{aligned} \quad (4)$$

We derive the TVLQR trajectory stabilizer according to Bertsekas [8]. The quadratic objective function to be minimized is

$$J_k(\bar{\mathbf{x}}_k) = \bar{\mathbf{x}}_N^{\mathrm{T}} \mathbf{S}_N \bar{\mathbf{x}}_N + \sum_{n=k}^{N-1} \left[\bar{\mathbf{x}}_n^{\mathrm{T}} \mathbf{Q} \bar{\mathbf{x}}_n + \bar{\mathbf{u}}_n^{\mathrm{T}} \mathbf{R} \bar{\mathbf{u}}_n \right] \quad (5)$$

$$\mathbf{S}_N \geq 0, \ \mathbf{Q} \geq 0, \ \mathbf{R} > 0, \quad (6)$$

with $\mathbf{S}_N$, $\mathbf{Q}$ and $\mathbf{R}$ being the penalty matrices on the final state deviation and state and input deviation from the nominal trajectory, respectively. We assume the optimal cost-to-go to have a quadratic form

$$J_k^*(\bar{\mathbf{x}}_k) = \bar{\mathbf{x}}_k^{\mathrm{T}} \mathbf{S}_k \bar{\mathbf{x}}_k, \quad (7)$$

with cost-to-go matrix $\mathbf{S}_k \geq 0$. Applying the dynamic programming update rule combined with the assumption on the optimal cost and linearized system dynamics, we obtain the optimal input

$$\begin{aligned} \bar{\mathbf{u}}_k^* &= -\left(\mathbf{R} + \mathbf{B}_k^{\mathrm{T}} \mathbf{S}_{k+1} \mathbf{B}_k\right)^{-1} \mathbf{B}_k^{\mathrm{T}} \mathbf{S}_{k+1} \mathbf{A}_k \bar{\mathbf{x}}_k \\ &= -\mathbf{K} \bar{\mathbf{x}}_k, \end{aligned} \quad (8)$$

where $\mathbf{K} \in \mathbb{R}^{m \times n}$ is the compensator matrix. We further obtain the update rule for the discrete time TVLQR optimal cost-to-go

$$\begin{aligned} \mathbf{S}_k =& \mathbf{Q} + \mathbf{A}_k^{\mathrm{T}} \Big(\mathbf{S}_{k+1} \\ &- \mathbf{S}_{k+1} \mathbf{B}_k \left(\mathbf{R} + \mathbf{B}_k^{\mathrm{T}} \mathbf{S}_{k+1} \mathbf{B}_k\right)^{-1} \mathbf{B}_k^{\mathrm{T}} \mathbf{S}_{k+1} \Big) \mathbf{A}_k. \end{aligned} \quad (9)$$

In case the trajectory ends at an equilibrium point of the system (1), we obtain the boundary condition $\mathbf{S}_N$ from the stabilizing solution to the time invariant version of the discrete algebraic Riccati equation (9) ($\mathbf{S}_k = \mathbf{S}_{k+1} = \mathbf{S}_N$).

B. Approximation to the Goal State Basin of Attraction

After simulating a random sample, we check if the tree managed to bring the sample inside a region around the goal state, which we describe by a hyper-ellipsis defined by the final cost matrix $\mathbf{S}_N$.

In the following examples, we use goal states $\mathbf{x}_G$ which are stabilizable. This implies that we can design a linear time invariant linear quadratic regulator (LTI-LQR) controller for the goal state. Furthermore, we can approximate the basin of attraction of the closed-loop nonlinear system at the goal state using a formal method. In the following, we present the principle described in [1], in a discrete-time setting.

We are given discrete-time, nonlinear closed-loop system dynamics of the form

$$\begin{aligned} \bar{\mathbf{x}}_{k+1} &= \mathbf{f}(\bar{\mathbf{x}}_k) \\ \bar{\mathbf{x}}_k &= \mathbf{x}_k - \mathbf{x}_G, \end{aligned} \quad (10)$$

where $\mathbf{x}_G$ is an equilibrium of the system, i.e. $\mathbf{f}(\mathbf{0}) = \mathbf{0}$. We define the basin as the largest set of states for which the optimal cost-to-go decreases with every step

$$J^*(\mathbf{f}(\bar{\mathbf{x}}_k)) - J^*(\bar{\mathbf{x}}_k) < 0, \quad (11)$$

which represents a discrete-time formulation of a Lyapunov function. Every state within the final basin should always

take a step towards the goal, reducing it's cost-to-go with every step and eventually reaching $\mathbf{x}_G$, implying asymptotic stability. We require this to hold over the domain $\mathcal{B}(\rho_N)$ defined as

$$\mathcal{B}(\rho_N) := \left\{\mathbf{x}|\mathbf{x}^{\mathrm{T}}\mathbf{S}_N\mathbf{x} \le \rho_N\right\}, \tag{12}$$

with $\mathbf{S}_N$ being the LQR cost-to-go matrix. The goal is now to search for the largest ρ_N for which (11) holds, i.e. we are looking for the largest $\mathcal{B}(\rho_N)$ completely contained in the basin of attraction of the nonlinear closed-loop system. We add $\mathcal{B}(\rho_N)$ to the problem using the Lagrange multiplier $\mathrm{h}(\bar{\mathbf{x}}_k)$

$$J^*\left(\mathbf{f}\left(\bar{\mathbf{x}}_k\right)\right) - J^*\left(\bar{\mathbf{x}}_k\right) + \mathrm{h}(\bar{\mathbf{x}}_k)\left(\rho_N - \bar{\mathbf{x}}_k^{\mathrm{T}}\mathbf{S}_N\bar{\mathbf{x}}_k\right) < 0 \tag{13}$$
$$\mathrm{h}(\bar{\mathbf{x}}_k) = \mathbf{m}^T(\bar{\mathbf{x}}_k)\mathbf{H}\mathbf{m}(\bar{\mathbf{x}}_k), \quad \mathbf{H} \ge 0,$$

where $\mathbf{m}(\bar{\mathbf{x}}_k)$ is a vector of monomials of order zero to N_m in the components of $\bar{\mathbf{x}}$. Note that with $<$, we state that the left hand side of (13) needs to be a negative definite function of $\bar{\mathbf{x}}_k$.

We test (13) for negative definiteness using a SoS program [2]. For a given value of ρ_N, the SoS program is feasible if it can find values for the elements of the matrix $\mathbf{H}$ such that (13) holds. In order to execute the program, (13) has to be a polynomial expression. All summands in (13) except $J^*\left(\mathbf{f}\left(\bar{\mathbf{x}}_k\right)\right)$ are already polynomials. To make the whole expression polynomial, we approximate $J^*\left(\mathbf{f}\left(\bar{\mathbf{x}}_k\right)\right)$ using a Taylor expansion

$$\hat{J}^*\left(\mathbf{f}\left(\bar{\mathbf{x}}_k\right)\right) = J^*\left(\mathbf{f}\left(\bar{\mathbf{x}}_k = 0\right)\right) + \left.\frac{\partial J^*\left(\mathbf{f}\left(\mathbf{x}\right)\right)}{\partial \mathbf{x}}\right|_{\mathbf{x}=0} \bar{\mathbf{x}}_k + \dots \tag{14}$$

to sufficiently high order, omitting the higher order terms. We now perform a binary search on ρ_N to find the maximal ρ_N that results in a feasible SoS program.

Since this step does not take input constraints into account, we check if any of the states on the surface of the hyper-ellipsis violate the input constraints and reduce ρ_N if necessary.

III. The Algorithm

In the following, we explain the steps of the algorithm in more detail. An overview of the algorithm in pseudocode is stated in Algorithm 1.

A. The Tree

Each node i in the tree consists of 6 elements:

1) $\mathbf{x}_{0i}$: Nominal state.
2) $\mathbf{u}_{0i}$: Nominal input.
3) $\mathbf{S}_i$: Cost-to-go matrix. $J_i(\bar{\mathbf{x}}) = \bar{\mathbf{x}}^{\mathrm{T}}\mathbf{S}\bar{\mathbf{x}}$.
4) $\mathbf{K}$: TVLQR compensator matrix. $\bar{\mathbf{u}} = -\mathbf{K}\bar{\mathbf{x}}$.
5) ϕ: Local description of funnel. A state $\mathbf{x}$ is in node i's funnel if $(\mathbf{x} - \mathbf{x}_{0i})^{\mathrm{T}}\mathbf{S}_i(\mathbf{x} - \mathbf{x}_{0i}) < \phi_i$.
6) p: Pointer to parent node (next node in trajectory if time index advances by 1).

For the goal node $\mathbf{x}_G$, $\phi = \rho_N$ and $p = NULL$. From each node starts a time varying nominal input and state trajectory that leads to the goal: $\mathbf{u}_{0k}, \mathbf{x}_{0k}, k = 0, \dots, N$ with $\mathbf{x}_{0N} = \mathbf{x}_G$. In the following, we call the nominal trajectory starting at a node combined with the TVLQR compensator matrices the node's *policy*.

Algorithm 1 Simulation-Based LQR-Trees

```
1:  [A, B] ⇐ linearization around x_G, u_G and discretization
2:  [K, S] ⇐ dlqr(A, B, Q, R)
3:  ρ_N ⇐ Approximated basin of x_G
4:  T(1) ⇐ {x_G, u_G, S, K, ρ_N, NULL}
5:  for j = 1 to maxIter do
6:    xSample ⇐ random sample
7:    Γ ⇐ build simulation priority array
8:    for i = 1 to |Γ > 0| do
9:      xSim ⇐ simTree(T, i, xSample)
10:     if isInGoalBasin and constraintsOK then
11:       continue; break if enough successes
12:     else
13:       T ⇐ adjustFunnel(T, i, xSim)
14:     end if
15:   end for
16:   if noSimSuccessful then
17:     [iNear] ⇐ distanceMetric(T, xSample)
18:     [T, foundTraj] ⇐ growTree(T, iNear, xSample)
19:     if foundTraj then
20:       Reset success counter
21:     end if
22:   else
23:     Increment success counter; terminate if converged
24:   end if
25: end for
```

B. Sampling and Simulation

The goal of the algorithm is to cover a predefined subspace $\mathcal{R}$ of the state space. We sample uniformly from $\mathcal{R}$ to generate the random sample $\mathbf{x}_S$. This sample serves as an initial condition of the system for the subsequent simulations.

After generating a random sample, we simulate the existing tree $\mathbf{T}$. First, we build the array Γ, which determines the simulation priority of the tree nodes. It's elements are

$$\gamma_i = \phi_i - \left(\mathbf{x}_S - \mathbf{x}_{0i}\right)^{\mathrm{T}}\mathbf{S}_i\left(\mathbf{x}_S - \mathbf{x}_{0i}\right), \; i = 1, \dots, |\mathbf{T}| \tag{15}$$

with $|\mathbf{T}|$ being the total number of nodes in the tree. The γ_i represent a measure of how 'far' the sample lies within the ellipsis of the respective node in the tree. Any node with a positive γ_i should be able to get the sample to the goal basin $\mathcal{B}(\rho_N)$ when the node's policy is applied.

We sort Γ in descending order and simulate the sample if any element of Γ is positive. Only simulating the sample with policies of nodes with $\gamma_i > 0$ speeds up the simulation step as many nodes that are likely to fail are ignored. If there are no nodes with $\gamma_i > 0$, we proceed to the motion planning step, see Section III-C. We start simulating the sample using the policy of the node q with the largest γ_q in Γ, i.e. the first element of Γ. This results in an efficient shrinking of ϕ_i. Since we are shrinking the funnel around a trajectory from

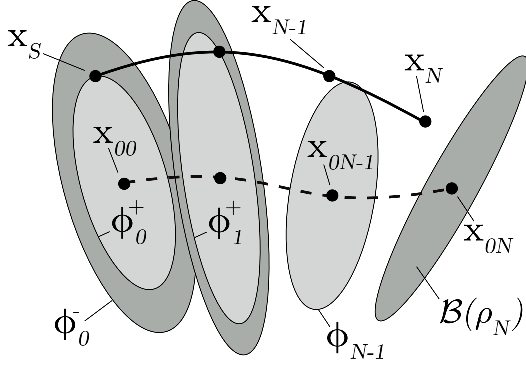


Fig. 1. Adjusting the Funnel after a Failed Simulation. The simulated trajectory (solid black) failed to reach the goal basin $\mathcal{B}(\rho_N)$ using the policy starting at node q : $\mathbf{x}_{00}, \phi_0$. However, the funnel described by the darker grey ellipses defined by $\phi^-_{0,1}$ around the first two nodes of the policy's nominal trajectory (dashed line) predicted a successful simulation. Therefore, we adjust $\phi^-_{0,1}$ to $\phi^+_{0,1}$ according to (17), resulting in the light grey ellipses. The simulated state at time index $N-1$ was not inside the ellipsis of node $N-1$ and thus ϕ_{N-1} remains unchanged.

covering the whole subspace $\mathcal{R}$, the γ_i are a measure of how likely it is that the respective ϕ_i needs more shrinking.

We simulate the nonlinear system using the random sample $\mathbf{x}_S$ as initial condition and the policy of node q for $t \in [0, N \cdot dt]$. The simulation generates a sampled trajectory of the system with states $\mathbf{x}_0 = \mathbf{x}_S, \mathbf{x}_1, ..., \mathbf{x}_N$. After the simulation, we check if the system reached the approximated basin of attraction of the goal state $\mathcal{B}\left(\rho_N\right)$:

$$\left(\mathbf{x}_N - \mathbf{x}_G\right)^{\mathrm{T}} \mathbf{S}_N \left(\mathbf{x}_N - \mathbf{x}_G\right) \leq \rho_N. \tag{16}$$

The input constraints are enforced using a saturation on the input; we then only have to check whether the trajectory of the system generated in the simulation violates any state constraints.

If the policy of node q is successful, we continue with the next node in Γ. After a fixed number (we use 10) of successful simulations, we proceed to the next random sample; it would be sufficient to proceed after the first success, however, we set the threshold higher to efficiently sample more funnels with a single sample.

If the policy of node q fails, we adjust the funnel of the nominal trajectory starting at q setting

$$\phi_k = \bar{\mathbf{x}}_k^{\mathrm{T}} \mathbf{S}_k \bar{\mathbf{x}}_k, \quad k = \{0, 1, ..., N-1\}, \tag{17}$$

with $\bar{\mathbf{x}}_k = \mathbf{x}_k - \mathbf{x}_{0k}$ and $\mathbf{x}_{00}$ being the nominal state of node q. We only adjust ϕ_k if $\phi_k > \bar{\mathbf{x}}_k^{\mathrm{T}} \mathbf{S}_k \bar{\mathbf{x}}_k$. We only shrink the funnels and never expand them. This is illustrated in Fig. 1.

C. Growing the Tree

If a random sample $\mathbf{x}_S$ never reaches $\mathcal{B}(\rho_N)$ using the existing policies in the tree, we need to expand the tree. We find the closest node i_{near} to $\mathbf{x}_S$ in the tree using a distance metric and use motion planning to generate a trajectory connecting the sample to i_{near}. In the current implementation, we use the distance metric described in [1] and [9]. It bases on a linearization of the system dynamics around the sample point and calculating the optimal control cost-to-go from the nodes in the tree. Based on this cost, we choose the closest node to connect to.

The distance metric not only provides the closest node, but also an initial guess of the input $\hat{\mathbf{u}}^*$ to the system to reach the sample from the closest node. We further refine the open-loop trajectory given by $\hat{\mathbf{u}}^*$ with a direct collocation [10] implementation using SNOPT [11]. If motion planning fails, we discard the sample as (temporarily) unreachable. As the tree grows, we expect that sample to be included in a funnel of the tree if it is reachable.

After finding the connecting trajectory, we stabilize it using the discrete TVLQR controller derived in Section II-A, setting $\mathbf{S}_N$ to $\mathbf{S}_{i_{\mathrm{near}}}$. This generates a $\mathbf{K}_j, \mathbf{S}_j$ for every node in the new trajectory and we add the new nodes to the tree, setting $\phi_j \rightarrow \infty$, so that the funnel of the added trajectory covers the whole subspace $\mathcal{R}$.

D. Algorithm Termination Criteria

We terminate the algorithm after we encountered a fixed number of random samples that either successfully reached the goal or are unreachable, i.e. motion planning fails.

E. Using the Generated Tree

After termination of the algorithm, we obtain a tree $\mathbf{T}$ that probabilistically covers $\mathcal{R}$. To decide which stabilized trajectory to apply to a given initial condition $\mathbf{x}_{IC}$, we search for the node i with

$$i = \underset{j}{\operatorname{argmin}} \left(\mathbf{x}_{IC} - \mathbf{x}_{0j}\right)^{\mathrm{T}} \mathbf{S}_j \left(\mathbf{x}_{IC} - \mathbf{x}_{0j}\right) \quad \text{subject to: } \gamma_j > 0 \tag{18}$$

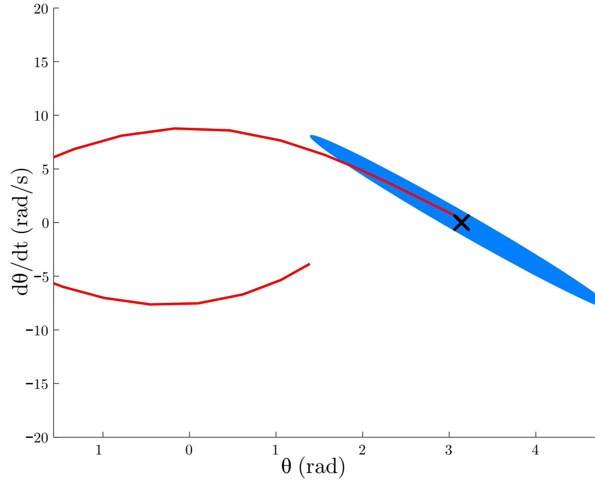
Thus we choose the policy starting at the node with the lowest cost-to-go for the initial condition whose funnel contains the initial condition. Since we approximate the funnel of a trajectory as we generate the tree, we can probabilistically guarantee that the policy we choose gets the initial condition to the goal.

IV. Simulation Examples

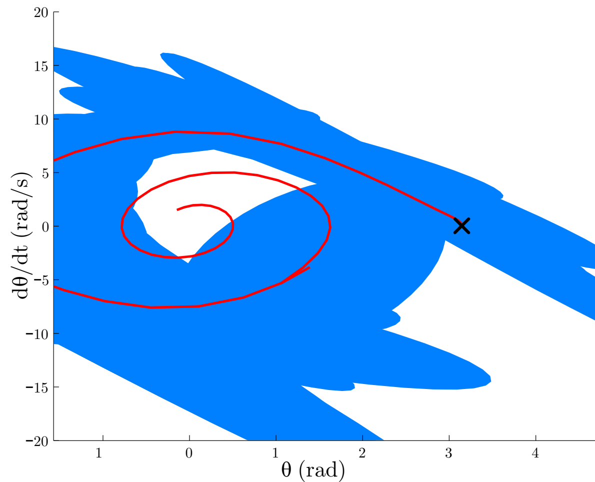
A. Simple Pendulum

We illustrate the algorithm using a simple, well visualizable system: the damped simple pendulum with input constraints. The parameters are $m = 1.0\,\mathrm{kg}$, $l = 0.5\,\mathrm{m}$, $g = 9.8\,\mathrm{m/s^2}$, $b = 0.1\,\mathrm{kgm^2/s}$ with the input constrained to $\pm 3\,\mathrm{Nm}$, requiring at least a single pump to swing up. We set the cost matrices for the goal state LQR controller to $\mathbf{Q} = \mathtt{diag}(10, 1)$ and $\mathbf{R} = 15$.

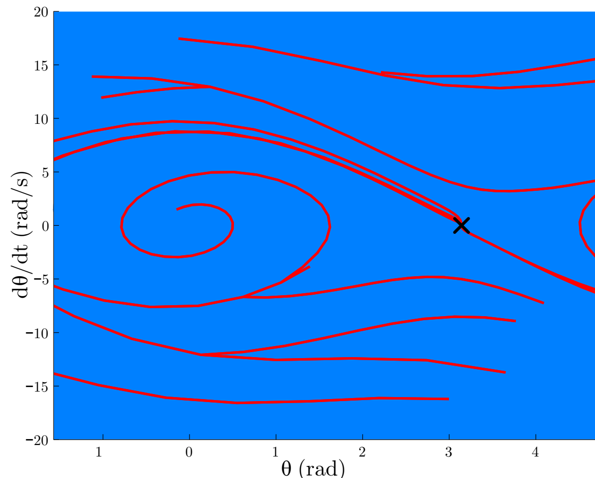
The evolution of the tree is illustrated in snapshots in Fig. 2. Note that wrapping of the coordinate system was not taken into account. The algorithm takes about half an hour to converge, with no attempts made to optimize run time. A large part of this time is due to the high threshold of 5000 consecutive successful simulations before the algorithm terminates. The average time for a random sample to be simulated for the whole tree was $0.27\,\mathrm{s}$ and stays almost constant over all iterations. Major spikes in time to simulate



(a) Iteration 2, 1 Trajectory, $|\mathbf{T}| = 41$. The final basin is plotted. The black cross is the goal state. The state space shown represents the subspace $\mathcal{R}$. Note that the funnel of the first added trajectory is not plotted, as it would cover the whole subspace.



(b) Iteration 28, 2 Trajectories, $|\mathbf{T}| = 81$. The funnel of the first trajectory was shrunken by unsuccessful simulations.



(c) Iteration 6086, 12 Trajectories, $|\mathbf{T}| = 477$. 5000 consecutive random initial conditions were successfully brought to the goal state.

Fig. 2. Tree Evolution Phase Plots for Simple Pendulum

a sample can be observed after the tree has been grown and many funnels are adjusted.

B. Cart-Pole

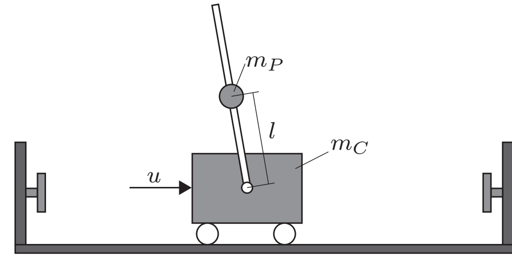


Fig. 3. The Cart-Pole: $m_C = 1.0$ kg, $m_P = 1.0$ kg, $l = 0.5$ m, $g = 9.8$ m/s^2 with the input u constrained to ± 30 N and the horizontal position x constrained to ± 0.45 m. We chose $\mathbf{Q} = \mathtt{diag}(50, 5, 40, 4)$ and $\mathbf{R} = 1$.

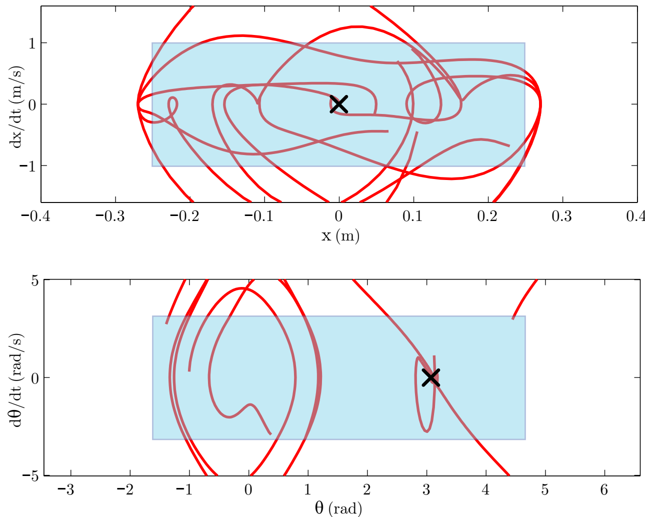
We show the state constraint capability with the cart-pole, see Fig. 3. It consists of an actuated cart with a passive pendulum attached. The state limitation is given by a limited rail on which the cart can move; a constraint often found in laboratory setups of this system.

Note that we chose the region $\mathcal{R}$ to be smaller in horizontal position x than the state constraints. We also set the constraints for the motion planning algorithm to be a shrunken version of the state constraints. This facilitates the algorithm in the sense that the controller can still take action on trajectories close to the state constraints without violating the constraints.

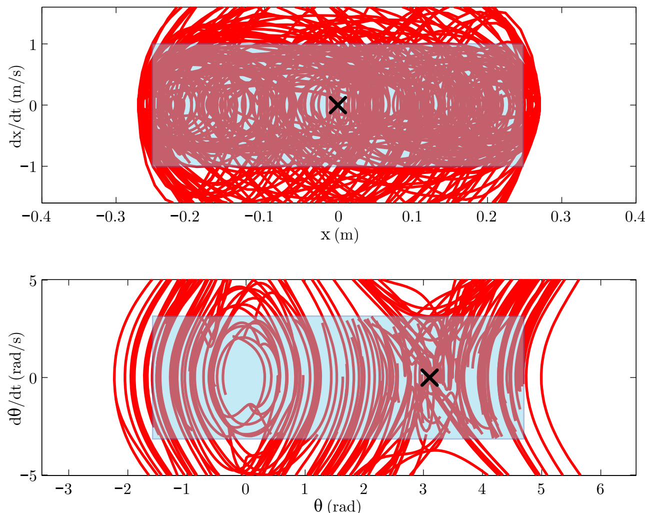
The algorithm takes significantly longer for the cart-pole than for the simple pendulum. It converged after 74375 iterations generating a tree with 11917 nodes and took about 32 hours on an average PC. For convergence, we required that the tree successfully brings 5000 consecutive random samples to the goal; we ignore unsuccessful samples that failed to connect to the tree. For this specific run, we had seven unreachable samples during the final 5000 iterations that could not be connected to the tree using motion planning. The large number of iterations needed is partly because of the doubled dimensionality compared to the pendulum, but a larger part is due to the increased complexity of adding the state constraints. Running the algorithm with the exact same parameters, only omitting the state constraints, it converged after 5821 iterations, producing a tree of 881 nodes in 2 hours.

V. Basic Robustness Analysis

We compared the performance of the generated policy for different model parameters than what the policy was designed for. For simplicity, we just introduced a scale factor that scales the mass of the cart and mass and length of the pendulum, e.g. $m_{P,sim} = \kappa \cdot m_P$, $m_{C,sim} = \kappa \cdot m_C$, $l_{sim} = \kappa \cdot l$. We ran simulations with the scaled parameters from 10000 random initial conditions uniformly drawn from the subspace $\mathcal{R}$ the algorithm was designed for. We further replaced the success criteria of ending up within the final basin by checking the actual simulated state for convergence



(a) Iteration 14, 5 Trajectories (solid red), $|\mathbf{T}| = 852$. The blue overlayed rectangle represents the subspace $\mathcal{R}$ that the algorithm iteratively covers with stabilized trajectories. The black cross is the goal state. Note that we do not plot the funnels as the projections from 4D to 2D can be misleading.



(b) Iteration 74375, 134 Trajectories (solid red), $|\mathbf{T}| = 11917$. The algorithm terminated after 5000 consecutive random initial conditions successfully reached the goal state.

Fig. 4. Generated Tree Phase Plots for the Cart-Pole

to the goal state. That means we simulated longer than the nominal time of the executed trajectory and applied the goal state controller for the exceeding simulation time.

We show the resulting percentages of successful simulations in Table I for the case of the tree generated with state constraints (A) and the tree without state constraints (B). We also ran the policy generated without state constraints on the constrained setup to compare the performance (C). Since we chose the weights in $\mathbf{Q}$ for the states x and $\dot{x}$ to be quite large, it could be that the controllers perform similarly. However, the percentages show that generating the tree with explicitly taking the state constraints into account results in a higher performance.

TABLE I

Simple Robustness Analysis for the Cart-Pole

κ	% Suc. (A)	% Suc. (B)	% Suc. (C)
0.8	2.82	13.57	4.31
0.85	4.92	99.63	14.50
0.9	78.48	100.00	46.15
0.95	99.34	100.00	43.40
1.0	99.75	100.00	46.59
1.05	99.27	100.00	63.07
1.1	95.92	99.98	53.06
1.15	83.56	99.80	25.31
1.2	51.10	97.89	14.67

VI. Discussion

It is unlikely that a hyper-ellipsis is the best geometrical primitive to describe the funnel around a trajectory. Simulation based approximation of the funnel would allow exploring different primitives that could potentially yield tighter fits to the real funnel, further improving the sparsity of the resulting tree. The advantage of the hyper-ellipsis we propose is that they are simple to reason about geometrically and are founded on the TVLQR design, thus are dynamically plausible.

We show a set of simulated trajectories from random initial conditions for the cart-pole in the video accompanying this submission. It is interesting that for some initial conditions the simulation using the tree generated for the unconstrained problem seems to outperform the tree for the constrained problem, even without violating the state constraints. This could be due to effects of the imperfect distance metric and local minima during the motion planning step. A possible way to improve the results is to seed the tree with some carefully designed trajectories from key initial conditions, e.g. a good swing up trajectory for the cart-pole. However, the results we show were obtained without seeding. Currently, we are working on evaluating the performance of the proposed algorithm on a laboratory setup cart-pole system.

References

[1] R. Tedrake, "LQR-trees: Feedback motion planning on sparse randomized trees," in *Proceedings of Robotics: Science and Systems*, Seattle, USA, June 2009.

[2] S. Prajna, A. Papachristodoulou, P. Seiler, and P. A. Parrilo, *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2004.

[3] M. Mason, "The mechanics of manipulation," in *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, vol. 2, Mar 1985, pp. 544–548.

[4] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, "Sequential Composition of Dynamically Dexterous Robot Behaviors," *I. J. Robotic Res.*, vol. 18, no. 6, pp. 534–555, 1999.

[5] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.

[6] C. Atkeson, "Using local trajectory optimizers to speed up global optimization in dynamic programming," in *Advances in Neural Information Processing Systems*. Morgan Kaufmann, 1994, pp. 663–670.

[7] M. Stolle and C. G. Atkeson, "Policies based on trajectory libraries," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 2006.

[8] D. P. Bertsekas, *Dynamic Programming and Optimal Control, Vol. I*. Athena Scientific, 2005.

[9] E. Glassman and R. Tedrake, "Lqr-based heuristics for rapidly exploring state space," *Submitted to ICRA 2010*, 2009.

[10] J. Betts, *Practical Methods for Optimal Control using Nonlinear Programming*. ASME, 2002.

[11] P. E. Gill, W. Murray, and M. A. Saunders, "Snopt: An sqp algorithm for large-scale constrained optimization," *SIAM Review*, vol. 47, no. 1, pp. 99–131, 2005.