Space-indexed Dynamic Programming: Learning to Follow Trajectories

J. Zico Kolter KOLTER@CS.STANFORD.EDU
Adam Coates ACOATES@CS.STANFORD.EDU
Andrew Y. Ng ANG@CS.STANFORD.EDU
Yi Gu GUYINET@STANFORD.EDU
Charles DuHadway DUHADWAY@STANFORD.EDU
Computer Science Department, Stanford University, CA 94305

Abstract

We consider the task of learning to accurately follow a trajectory in a vehicle such as a car or helicopter. A number of dynamic programming algorithms such as Differential Dynamic Programming (DDP) and Policy Search by Dynamic Programming (PSDP), can efficiently compute non-stationary policies for these tasks — such policies in general are well-suited to trajectory following since they can easily generate different control actions at different times in order to follow the trajectory. However, a weakness of these algorithms is that their policies are *time-indexed*, in that they apply different policies depending on the current time. This is problematic since 1) the current time may not correspond well to where we are along the trajectory and 2) the uncertainty over states can prevent these algorithms from finding any good policies at all. In this paper we propose a method for *space-indexed* dynamic programming that overcomes both these difficulties. We begin by showing how a dynamical system can be rewritten in terms of a spatial index variable (i.e., how far along the trajectory we are) rather than as a function of time. We then use these space-indexed dynamical systems to derive space-indexed version of the DDP and PSDP algorithms. Finally, we show that these algorithms perform well on a variety of control tasks, both in simulation and on real systems.

Appearing in *Proceedings of the 25th International Conference on Machine Learning*, Helsinki, Finland, 2008.

1. Introduction

We consider the task of learning to accurately follow a trajectory, for example in a car or helicopter. This is one of the most basic and fundamental problems in reinforcement learning and control. One class of approaches to this problem uses dynamic programming. These algorithms typically start at the last time-step T of a control task, and compute a simple (say, linear) controller for that time-step. Then, they use dynamic programming to compute controllers for time-steps $T-1$, $T-2$ and so on down to time-step 1. Some examples of algorithms in this family include (Jacobson & Mayne, 1970; Bagnell et al., 2004; Atkeson & Morimoto, 2003; Lagoudakis & Parr, 2003), and all of them output time-varying/non-stationary policies that choose the control action as a function of time. Given that following a trajectory requires one to choose very different control actions at different parts of the trajectory — for example, the controls while driving a car on a straight part of the trajectory are very different from the controls needed during a turn – these dynamic programming algorithms therefore initially seem well-suited for trajectory following.

However, a weakness in the naive dynamic programming approach is that the control policies are *time-indexed*. That is, these algorithms output a sequence of controllers $\pi_1, \pi_2, \ldots, \pi_T$ and execute controller π_t at time t. However, as time passes, the uncertainty over the state increases, and this can greatly degrade controller performance. For example, suppose we are driving a car around a track with both straight and curved portions, and suppose that the controller at time t assumed the car was on a curved portion. If, due to the natural stochasticity of the environment, the car was actually on a straight portion of the track at this time, the resulting controller would perform very poorly, and this problem increases over time. This

problem can be alleviated slightly be "re-indexing" the controllers by state during execution. That is, at time t we do not necessarily execute controller π_t, but instead we examine *all* the controllers $\pi_1, \ldots, \pi_T$, and execute the controller whose corresponding state is closest to the current state — several variations on this approach exist, and we will discuss them further in Section 5. However, there are two fundamental limitations of this general method. First, because we are executing a different policy from the one learned by dynamic programming, it is difficult to provide any of the performance guarantees that often accompany the purely time-indexed dynamic programming algorithms. Second, and more fundamentally, the uncertainty over states in the distant future often make it extremely difficult to learn *any* good policy using the time-indexed algorithms. This means that regardless of how we re-index the controllers during execution, we are unlikely to obtain good performance.

In this paper we propose a method for *space-indexed* dynamic programming that addresses both these concerns. More precisely, we will define a spatial index variable d that measures how far we have traveled along the target trajectory. Then, we will use policies π_d that depend on d — where we are along the trajectory — rather than the current time t. In order to learn such policies, we define the notion of a *space-indexed dynamical system*, and show how various dynamical systems can be rewritten such that their dynamics are indexed by d instead of by time t. This then allows us to extend various dynamic programming algorithms to produce space-indexed policies — in particular, we develop a space-indexed versions of the Differential Dynamic Programming (DDP) (Jacobson & Mayne, 1970) and Policy Search by Dynamic Programming (PSDP) algorithms (Bagnell et al., 2004). Finally, we successfully apply this method to several control tasks, both in simulation and in the real world.

The remainder of this paper is organized as follows. In Section 2 we show how to transform a standard (time-indexed) dynamical system into a space-indexed dynamical system. Using this transformation, in Section 3 we develop space-indexed versions of the DDP and PSDP algorithms. In Section 4 we present experimental results on several control tasks. Finally, in Sections 5 and 6 we discuss related work and conclude the paper.

2. Space-indexed Dynamical Systems

Standard dynamic programming algorithms are very efficient because they know in advance that the policies $\pi_1, \ldots, \pi_T$ will be executed in a certain sequence (and that each policy will be executed only once), and can thus solve for them in reverse order. The key difficulty of generalizing a dynamic programming algorithm to the space-indexed setting is that it is difficult to know in advance where in space (i.e., how far along the trajectory) the vehicle will be at each step, and thus which space-indexed policy will be executed when. For example, if the vehicle is currently at space-index d, then there is no guarantee that executing policy π_d for one time-step will put the vehicle in space-index $d+1$. But if π_d might be executed multiple times before switching to π_{d+1}, then in general its parameters cannot be solved for in closed form during the dynamic programming backup step, and require some complex policy search instead. In this section we discuss a method for addressing this problem. Specifically, we will rewrite the dynamics of a system so that the states and transitions are indexed by the spatial-index variable d rather than by the time t.

Suppose we are given a general non-linear dynamics model in the form of a (possibly stochastic) differential equation $\dot{s} = f(s, u)$, where $s \in \mathbb{R}^n$ denotes the state vector $u \in \mathbb{R}^m$ denotes the control input, and $\dot{s}$ denotes the derivative of the state vector with respect to time. While some classical control algorithms operate directly on this differential equation, a common technique in reinforcement learning and control is to create a discrete-time model of the system

$$s_{t+1} = F(s_t, u_t) + w_t$$

by numerical integration, where s_t and u_t denote the state and input at time t respectively, and w_t is a zero-mean IID noise term (typically taken to be Gaussian with some prespecified covariance, for example). A simple but very common method for achieving this discretization is by Euler integration. In this case the state evolves as

$$s_{t+\Delta t} = s_t + f(s_t, u_t)\Delta t + w_t$$

where Δt is the integration time constant (the variance of w_t will scale linearly with the time constant as well). Note that even though the system evolves in continuous time, by making the decision to model it as a discrete-time system, we have made a decision to explicitly represent the state only at certain *instants* in time ($t = \Delta t, t = 2\Delta t, \ldots$).

When transitioning to a space-indexed dynamical system, we instead will explicitly represent the state only when it is at certain *points* along the trajectory. We begin by representing the time-indexed state as $s_t = [x_t, \theta_t]^T$, where $x \in \mathbb{R}^p$ represents what we refer to as the *spatial portions* of the state (in this paper we typically consider the spatial portions of the state to be the 2D or 3D position). Now, assume we are

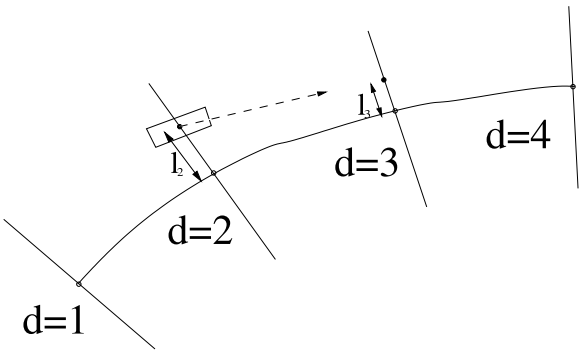

Figure 1. Figure illustrating space-indexed dynamics.

given a target trajectory in the space of x, such as the curved path shown in Figure 1. We choose a total of D discrete points along the trajectory, and designate the target state at these points as $x_1^\star, x_2^\star, \ldots, x_D^\star$. In the space-indexed dynamical system, we will explicitly represent the state only when the state lies on a hyperplane which is orthogonal to the target direction of travel and which passes through the one of the target points $x_d^\star$. More formally, we let $\dot{x}_d^\star$ be the instantaneous direction of motion (along the target trajectory) at point d. We will then explicitly represent the state only when $(x - x_d^\star)^T \dot{x}_d^\star = 0$. This situation is depicted in Figure 1.

Because we constrain the state in this manner, our space-indexed state will have a different set of variables as our time-indexed state. In particular, we represent the space-indexed state as $\tilde{s}_d = [t_d, \ell_d, \theta_d]^T$ where $t_d \in \mathbb{R}$ denotes the time of the system, $\ell_d \in \mathbb{R}^{p-1}$ denotes the lateral deviation from the target trajectory — for $x \in \mathbb{R}^p$ a point satisfying the constraint that $(x - x_d^\star)^T \dot{x}_d^\star = 0$ can be represented using $p-1$ dimensions and this gives the lateral deviation term — and θ_d denotes the non-spatial portions of the state as before. The time variable t_d is kept for completeness, but it can be ignored if the policies do not depend on time.

Now we can rewrite the dynamics so that they are indexed by space rather than time; this will give us an equation for computing $\tilde{s}_{d+1}$ from $\tilde{s}_d$. For simplicity, we develop an Euler integration-like method, but the technique could also be extended to higher-order numerical integration methods. Rather than simulating the system forward by a fixed time step, we *solve* for Δt such that the next state will lie exactly on the $d+1$ plane. Temporarily ignoring the noise term, we solve

$$(\dot{x}_{d+1}^\star)^T (x + \dot{x}\Delta t - x_{d+1}^\star) = 0$$

for Δt. Note that a positive solution for Δt may not always exist, but it typically does except in degenerate cases, when the vehicle starts moving perpendicular or backward with respect to the desired direction, and this is unlikely given any reasonable controller. When $\Delta t > 0$ does exist, we can compute $s_{t+\Delta t} = s_t + f(s_t, u_t)\Delta t$ and use this to find the next space-indexed state

$$\tilde{s}_{d+1} = [t_d + \Delta t, \ell_{d+1}, \theta_{t+\Delta t}]^T$$

where ℓ_{d+1} is $x_{t+\Delta t} - x_{d+1}^\star$ expressed in the $\mathbb{R}^{p-1}$ subspace defined by the plane through $x_{d+1}^\star$. This gives us our final space-indexed simulator in the form

$$\tilde{s}_{d+1} = \tilde{F}(\tilde{s}_d, u_d) + \tilde{w}_d \tag{1}$$

where $\tilde{w}_d$ is a noise term. Although the distribution of $\tilde{w}_d$ is in general quite complex, we can apply methods from stochastic calculus to efficiently draw samples from this distribution. However, in practice we find that a simpler approximate approach works just as well: we compute $s_{t+\Delta t}$ as above, assuming no noise, then add noise as in the time-indexed model. This will result in a point that may no longer lie exactly on the space-index plane, so lastly we form the line between the states s_t and $s_{t+\Delta t}$ and let $\tilde{s}_{d+1}$ be the point where this line intersects the $d+1$ plane.

3. Space-indexed Dynamic Programming

In this section we use the techniques presented in the previous section to develop space-indexed versions of the Differential Dynamic Programming (DDP) and Policy Search by Dynamic Programming (PSDP) algorithms. We begin by defining notation. Let S be the state space and A be the action space (so that in the context of the dynamical system above, $S = \mathbb{R}^n$ and $A = \mathbb{R}^m$). Since, as described above, there exists a one-to-one mapping from time-indexed states to space-indexed states, all the quantities below can be equivalently expressed in terms of the space-indexed state. A reward function is a mapping $R : S \to \mathbb{R}$ and a policy is a mapping $\pi : S \to A$. Given a non-stationary sequence of policies $(\pi_t, \ldots, \pi_T)$ we define the value function

$$V_{\pi_t, \ldots \pi_T}(s) = \tfrac{1}{T} E[\textstyle\sum_{i=t}^T R(s_i) | s_t = s; (\pi_t, \ldots, \pi_T)].$$

3.1. Space-indexed DDP

We first review (time-indexed) DDP briefly. DDP approximates the dynamics and cost function of a system along a specific sequence of states. Given an initial controller π_{init}, DDP simulates the system to generate a sequence of states $s_1, \ldots, s_T$. It then linearizes the dynamics around these points to obtain a time-varying linear dynamical system, and forms a second-order (quadratic) approximation to the reward function. This system can then be solved by the Linear

Quadratic Regulator (LQR) algorithm (Anderson & Moore, 1989), which results in a new controller and a new sequence of states. This process is repeated until convergence.

Space-indexed DDP proceeds in the same manner. Given some initial controller π_{init} and space-indexed dynamical system of the form (1)[1], we simulate the system forward for D space-indexes, resulting in a set of states $\tilde{s}_1, \ldots, \tilde{s}_D$. Using our dynamics model, we form the first order Taylor expansion of the dynamics at each point along the trajectory, which results in the (space-indexed) linearized dynamics:

$$\tilde{s}_{d+1} = A_d \tilde{s}_d + B_d u_d.$$

As in standard DDP, we form a quadratic approximation of the reward function $R_d(\tilde{s}) = -\tilde{s}^T Q_d \tilde{s}$, where Q_d is some (usually PSD) matrix. This reduces the problem to an LQR problem, which can be solved efficiently using a backward recursion.

3.2. Space-indexed PSDP

We now briefly review PSDP. As input, PSDP takes a time horizon T, a restricted policy class Π, and a sequence of baseline distributions over the states space $\mu_1, \ldots, \mu_T$, where we can informally think of μ_t as providing a distribution over which states would be visited at time t by a "good" policy. Given policies $\pi_{t+1}, \ldots, \pi_T$, PSDP computes (or approximates via Monte-Carlo sampling and parameter search)

$$\pi_t = \arg\max_{\pi \in \Pi} E_{s \sim \mu_t}[V_{\pi, \pi_{t+1} \ldots \pi_T}(s)]. \qquad (2)$$

By starting with $t = T$ and proceeding down to $t = 1$, the algorithm is able to generate a sequence of policies that can perform well on the desired task. The space-indexed version of PSDP proceeds exactly as above, replacing the time t with the space index d and using the space-indexed simulator to generate the Monte-Carlo samples.

Just as in the time-indexed version, the space-indexed version of PSDP comes with nontrivial performance guarantees, formalized by the theorem below. The theorem follows immediately from the equivalent theorem for the time-indexed version of PSDP, and from the fact that the space-indexed dynamics and reward function do not depend on time.

Theorem 3.1 [following (Bagnell et al., 2004)] *Suppose $\pi = (\pi_1, \ldots, \pi_D)$ is a policy returned by an ϵ-approximate version of state-indexed PSDP where on each step the algorithm obtains π_d such that*

$$E_{s \sim \mu_d}[V_{\pi_d, \pi_{d+1}, \ldots, \pi_D}(s)] \geq \arg\max_{\pi \in \Pi} E_{s \sim \mu_d}[V_{\pi, \pi_{d+1} \ldots \pi_D}(s)] - \epsilon$$

Then for all $\pi_{\text{ref}} \in \Pi^D$,

$$V_\pi(s_0) \geq V_{\pi_{\text{ref}}}(s_0) - D\epsilon - D d_{\text{var}}(\mu, \mu_{\pi_{\text{ref}}})$$

where μ is the baseline distribution over space-index states (without the time component) provided to SI-PSDP, d_{var} denotes the average variational distance, and $\mu_{\pi_{\text{ref}}}$ is the state distribution induced by π_{ref}.

This bound not only provides a performance guarantee for the space-indexed PSDP algorithm, it also helps to elucidate the advantage of space-indexing over time-indexing. The bound implies that to make PSDP and SI-PSDP perform as well as possible, it would be best to provide them with $\mu_{\pi^\star}$, the baseline distribution of the optimal controller, as the baseline distribution. But for time-indexed PSDP, the natural stochasticity of the environment can cause this distribution to be highly spread out over the state space, even for the optimal policy. Therefore, when performing the maximization (2), it is likely that *no* policy in the class will perform very well, since this would require a policy that could operate well over many different regions of the state space. Thus, regardless of whether or not we re-index the resulting controllers by state during execution, the time-indexed version of PSDP would fail to find a good policy. In contrast, if we are doing a good job following the trajectory, then we would expect that the distribution over states at each space-index would be much tighter, allowing the space-indexed PSDP to perform much better.

4. Experiments

4.1. Autonomous Driving

We begin by considering the problem of autonomously and accurately following a trajectory with a car, such as that shown in Figure 4. Our first set of experiments were carried out in a simulator of the vehicle, built following (Rossetter & Gerdes, 2002) (with model parameters such as the vehicle dimensions, total weight, etc). To follow the desired trajectory, we applied the space-indexed DDP algorithm described above.

Prior to the work presented in this paper, significant engineering effort went into a hand-designed trajectory following controller; this was an initial version of the controller described in (Hoffmann et al., 2007), which was a hand-optimized, linear, regulation controller that computes its actions as a function of state

[1] For the DDP algorithm, we ignore the noise term $\tilde{w}_d$ because by the principle of certainty equivalence, the optimal controller for LQR does not depend on the variance/magnitude of the noise (Anderson & Moore, 1989).

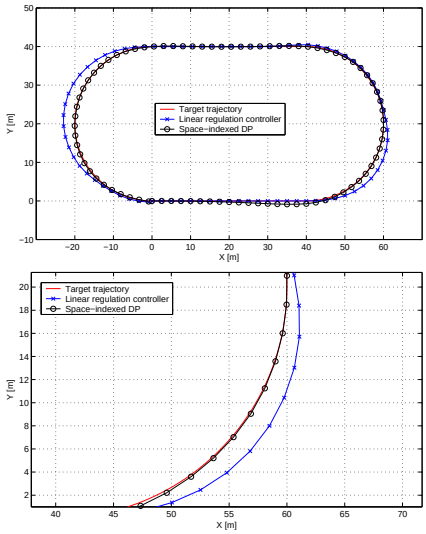

Figure 2. Comparison of the regulation controller and space-indexed controller. The figure below is a magnification of one of the turns.

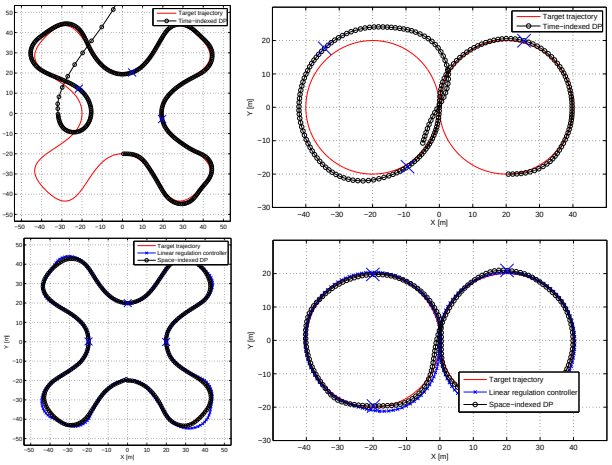

Figure 3. Two example trajectories where a time-indexed controller (above) performs significantly worse than the space-indexed controller (below).

features such as lateral error, orientation error, and so on. We used this controller to generate an initial trajectory for our space-indexed DDP algorithm; however, the results of space-indexed DDP were actually very insensitive to the choice of this initial controller.

Figure 2 shows the performance of the space-indexed DDP algorithm and the hand-tuned controller in simulation, when following an oval-like track at 30mph, along with a magnified view of the show the performance on one of the turns. We see that space-indexed DDP outperforms the hand-tuned controller; our controller has an RMS lateral error 0.26m, whereas the hand-tuned controller's RMS error is 1.18m.

Figure 3 shows a comparison between the performance of space-indexed and time-indexed dynamic programming. Due to the stochasticity of the simulator, the actual state s_t, for large t, is increasingly unlikely to be close to where the t-th step of the linearization occurred. Therefore the linearized approximation is less likely to be an accurate approximation of the "local" dynamics at time t. This is reflected in the figures: the time-indexed controllers initially perform well, but as time passes the controllers start to be executed at incorrect points along the trajectory, eventually leading the vehicle to veer off course. Using the space-indexed controller, however, the vehicle is able to accurately track the trajectory even for an arbitrarily long amount of time. For this relatively simple trajectory following task, re-indexing the time-indexed controllers by their spatial state, as described in the introduction, does perform well. However, as we will demonstrate in the next section, for more complex control tasks this is not the case.

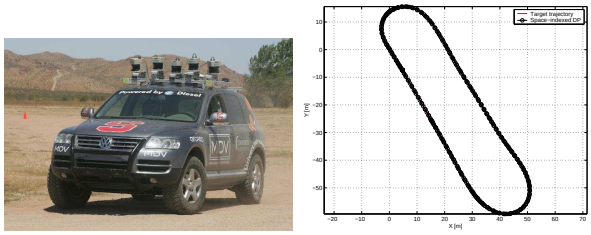

Figure 4. Picture of the vehicle used for experiments (left), and trajectory from a run on the actual car (right).

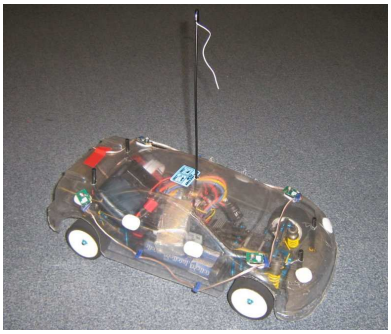

Figure 5. Autonomous RC car.

We also tested our method on the actual vehicle; the vehicle itself is described further in (Thrun & al., 2006). Figure 4 shown a typical result from an actual run on the vehicle moving at 10mph. The RMS error on the actual vehicle was about 0.17m, and the target and actual trajectories are indistinguishable in the figure.

4.2. Autonomous Driving with Obstacles

We next consider the more challenging control task of following a trajectory in the presence of obstacles. For this task we evaluated our methods on an RC car, shown in Figure 5. Since we want to learn a single controller that is capable of avoiding obsta-

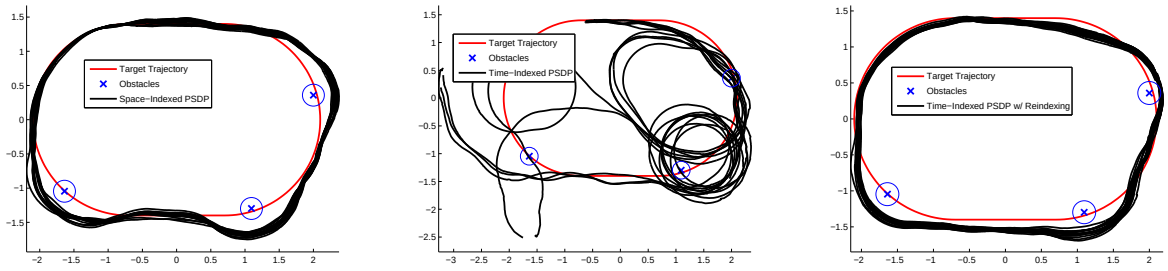

Figure 6. Trajectories taken by the real RC car on a course with obstacles, using space-indexed PSDP (left), time-indexed PSDP (middle), and time-indexed PSDP with re-indexing by space (right).

cles placed at arbitrary points along the trajectory, DDP is a poor algorithm (DDP learns a controller for a single fixed trajectory, but in this task we need to follow different trajectories depending on the location of the obstacles). Therefore, we apply the space-indexed version of PSDP to this task. Videos of the resulting controllers for this task are available at: `http://cs.stanford.edu/~kolter/icml08videos`.

In greater detail, we applied the space-indexed PSDP algorithm as follows. The native action space for the car domain is a two-dimensional input specifying the velocity and steering angle (between -28 degrees and 28 degrees), but for our task we kept the velocity fixed at 1.5 m/s and discretized the commanded steering angle into five equally spaced angles. To generate the initial distribution for PSDP, $\mu_1, \ldots, \mu_D$, we sampled 2000 different trials in a simulator of the car, using a PD controller. Then, for each space index from $D-1$ down to 1, we (approximately) solved the optimization problem (2) by first trying each possible action for each of the 2000 sampled states, then executing the learned controller for all subsequent space-indices to compute the resulting cost of the policy.[2] We then tried to learn the optimal action as a linear function of various state features; in particular, each controller was of the form: $u_{\text{steer}} = \arg\max_i w_i^T \phi(s)$ where $w_i \in \mathbb{R}^n$ is a weight vector learned by a multi-class SVM-like algorithm[3], and $\phi(s) \in \mathbb{R}^n$ is a feature vector. In our setting the features comprised of 1) the x and y location of the car, 2) the sine and cosine of the current car orientation, 3) 16 exponential RBF functions, spaced uniformly around the car, indicating the presence of an obstacle, and 4) a constant term. In addition to the space-indexed version, we also evaluated the performance of a pure time-indexed version, and a time-indexed version where we re-index the controllers as follows: at time t rather than execute the controller π_t, we examine all the controllers $\pi_1, \ldots, \pi_T$ and execute the controller $\pi_{t'}$ with minimum distance from the current state to the mean of the distribution $\mu_{t'}$.

Table 1. Average costs and collision counts for the simulated RC car with obstacles, averaged over 1000 runs.

Algorithm	Cost	Collisions
SI-PSDP	**56.17 ± 0.43**	**51**
TI-PSDP	58.39 ± 0.40	181
TI-PSDP w/ re-indexing	58.19 ± 0.37	212
Hand-tuned PD Controller	58.35 ± 0.34	231

Table 1 shows the average cost incurred and total number of collisions for the different controllers in 1000 simulated trials, where each trial had three randomly placed obstacles on the trajectory. As can be seen, the space-indexed version outperforms all the other variants of the algorithm as well as a hand-tuned PD controller that we previously spent a good deal of time trying to tune. The performance benefits of the space-indexed controller become even more pronounced on the real system. Figure 6 shows typical resulting trajectories from the space-indexed controller, the pure time-indexed controller, and the time-indexed controller with re-indexing. Due to the stochasticity of

[2] We used a cost function (i.e., negative reward), of

$$C(s_d) = \theta_1 \ell^2 + \theta_2 (1 - dist/\theta_3) \mathbf{1}\{dist < \theta_3\}$$

where $dist$ denotes the distance to the nearest obstacle, $\mathbf{1}$ is the indicator function, and $\theta_1, \ldots, \theta_3$ are parameters that trade off the relative cost of deviating from the trajectory and getting close to obstacles. For our experiments, we used $\theta_1 = 1000$, $\theta_2 = 500$ and $\theta_3 = 0.5$.

[3] Algorithmic details: the PSDP algorithm with discrete actions leads to a cost-sensitive, k-class learning problem with examples form $\{\phi(s^{(i)}) \in \mathbb{R}^n, c^{(i)} \in \mathbb{R}^k\}$, $i = 1, \ldots m$ where $\phi(s^{(i)})$ are the features, and $c_j^{(i)}$ represents the cost of classifying example i as belonging to class j. We approximately solve this problem with a support vector machine-like algorithm, which bears some similarly to previous work in cost-sensitive SVM earning (Geibel et al., 2004). The algorithm finds the solution to the optimization problem:

$$\begin{aligned} \min_{w,\xi \geq 0} \quad & \textstyle\sum_{j=1}^k \frac{1}{2}\|w_j\|^2 + C \sum_{i=1}^m \sum_{j,l=1}^k (c_j^{(i)} - c_l^{(i)})^+ \xi_{i,j,l} \\ \text{s.t.} \quad & (w_l - w_j)^T \phi(s^{(i)}) \geq 1 - \xi_{i,j,l} \quad \forall i, j, k. \end{aligned}$$

Figure 7. Tempest autonomous helicopter.

the real domain, the pure time-indexed approach performs very poorly. Re-indexing the controllers helps significantly, but the space-indexed version still performs substantially better (an incurred cost of 49.32 for space-indexed versus 59.74 for time-indexed with re-indexing, and the latter controller will nearly always hit at least one of the obstacles on the track). As seen in the figure, the space indexed version is able to track the trajectory well, while reliably avoiding obstacles.

4.3. Autonomous Helicopter Flight

We also apply these ideas to a simulated autonomous helicopter. This work used a stochastic simulator of the autonomous helicopter shown in Figure 7, and we considered the problem of making accurate, high-speed (5m/s) turns on this helicopter. We applied the space-indexed PSDP algorithm to this task due to the fact that the policy search setting allowed us to greatly restrict the class of control policies π_d under consideration (the space of all control policies for helicopter flight is very large, so we wanted to limit the risk of unexpected behavior). In particular, the "actions" of the controllers corresponded to picking a location of set point $x^\star$ which is then fed into a regulation controller, such as those described in (Bagnell & Schneider, 2001; Ng et al., 2004). Space constraints preclude a full description of the environment and algorithm, but the overall algorithm proceeds as in the previous section.

Figure 8 shows a typical result of applying the space-indexed PSDP algorithm to this task, along with the trajectory taken by a simple linear regulation controller. By varying the set point differently at different points along the trajectory, the space-indexed PSDP algorithm follows the trajectory much more accurately.

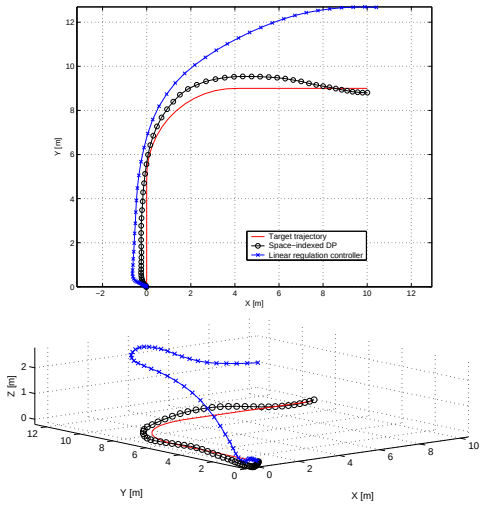

Figure 8. Comparison of the regulation controller and space-indexed controller in the helicopter simulation. The figure below shows the trajectories in three dimensions.

5. Related Work

The idea that a control policy should be dependent on the system's spatial state is by no means a new idea in the reinforcement learning and control literature. In the Markov Decision Process (MDP) formalism (Puterman, 1994), a policy is a mapping from states (which typically describe the *spatial* state of the system) to actions. In light of this observation, many classical dynamic programming algorithms such as value iteration or policy iteration can be viewed as performing dynamic programming on spatial states. However, in high-dimensional, continuous state spaces, the well-known "curse of dimensionality" renders a naive application of these algorithms intractable. Indeed, it is this reality that often motivates the jump to the trajectory following approach, where we want to find polices that perform well along the trajectory in particular.

There are also a number of methods for trajectory following. A common approach is to design a "regulation controller" that can keep the vehicle stable at a specified position $x^\star$. By smoothly moving $x^\star$ along the target trajectory, we can cause the vehicle to move along this path (Franklin et al., 1995; Dorf & Bishop, 2000). This approach works well when the regulation controller has very high bandwidth – i.e., if it can track $x^\star$ almost exactly as it varies — and is successful in application areas such as control of robot arms. But in more general settings in which the actual state x of the vehicle tends to lag well behind changes to $x^\star$, one often ends up manually and laboriously adjusting the regulation controller to try to obtain proper trajectory following performance. There are methods for compensating for this lag such as feedback linearization (Sastry, 1999), and there have also been many methods devised for trajectory following on specific systems (Egerstedt & Hu, 2000; Johnson & Calise, 2002). However, we know of no method for trajectory following in the general case of the nonholonomic, underaccuated vehicles that we consider.

The idea of partitioning the state space into regions, and using different controllers in the different regions, is a common practice in control, and often is referred to as gain-scheduling (Leith & Leithead, 2000). Taken

in the general sense, the algorithm we present in this paper can be viewed as a method for gain-scheduling, though more often the term is used for a particular application of this approach to the contexts of linear parameter varying systems. Such methods typically linearize the dynamical system around certain operating points, learn controllers at each point, and smoothly interpolate between controllers at various locations. However, the focus of this work is often to prove stability of such controllers using Lyapunov theory, and the overall approach is substantially different from what we consider here.

Model predictive control (MPC) (Garcia et al., 1989) (indirectly) addresses the issue of state uncertainty increasing over time, by explicitly computing new controllers at every time step in an online manner. However, MPC is generally orthogonal to the ideas we present here, since one could just as easily use a space-indexed dynamic programming method for the local controller in MPC. Furthermore, MPC can often times be computationally impractical to run real-time. An alternative approach is to use a local control method, such as DDP, in order to estimate the value function along several trajectories, and use these local estimates to build an approximate global model of the value function (Atkeson, 1994; Tassa et al., 2007). However, since these methods employ DDP, which is a time-indexed algorithm, they can potentially suffer the same problems as time-indexed methods in general.

6. Conclusions

In this paper we presented a space-indexed dynamic programming method for trajectory following. We showed how to convert standard time-indexed dynamical systems into equivalent space-indexed dynamical systems, and used this formulation to derive space-indexed versions of two well-known dynamic programming algorithms, DDP and PSDP. Finally, we successfully applied these methods to several control tasks, and demonstrated superior performance compared to their time-indexed counterparts.

Acknowledgments. This work was supported by the DARPA Learning Locomotion program under contract number FA8650-05-C-7261. We thank the anonymous reviewers for helpful comments, Sam Schreiber and Quan Gan for assistance with the RC car, Pieter Abbeel for assistance with the helicopter domain, Mark Woodward, Mike Montemerlo, Gabe Hoffmann, David Stavens and Sebastian Thrun for assistance with the DARPA Grand Challenge vehicle.

References

Anderson, B. D. O., & Moore, J. B. (1989). *Optimal control: Linear quadratic methods*.

Atkeson, C., & Morimoto, J. (2003). Nonparametric representation of policies and value functions: A trajectory-based approach. *NIPS 15*.

Atkeson, C. G. (1994). Using local trajectory optimizers to speed up global optimization in dynamic programming. *Neural Information Processing Systems 6*.

Bagnell, J., & Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. *Proceedings of the International Conference on Robotics and Automation*.

Bagnell, J. A., Kakade, S., Ng, A. Y., & Schneider, J. (2004). Policy search by dynamic programming. *Neural Information Processing Systems 16*.

Dorf, R., & Bishop, R. (2000). *Modern control systems, 9th edition*. Prentice-Hall.

Egerstedt, M., & Hu, X. (2000). Coordinated trajectory following for mobile manipulation. *Proceedings of the Internation Conference on Robotics and Automation*.

Franklin, G., Powell, J., & Emani-Naeini, A. (1995). *Feedback control of dynamic systems*. Addison-Wesley.

Garcia, C., Prett, D., & Morari, M. (1989). Model predictive control: theory and practice — a survey. *Automatica*, *25*, 335–348.

Geibel, P., Brefeld, U., & Wysotzki, F. (2004). Perceptron and SVM learning with generalized cost models. *Intelligent Data Analysis*, *8*.

Hoffmann, G., Tomlin, C., Montemerlo, M., & Thrun, S. (2007). Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing. *Proc. 26th American Control Conf.*

Jacobson, D., & Mayne, D. (1970). *Differential dynamic programming*. Elsevier.

Johnson, E., & Calise, A. (2002). A six degree-of-freedom adaptive flight control architecture for trajectory following. *Proceedings of the AIAA Guidance, Navigation, and Control Conference*.

Lagoudakis, M., & Parr, R. (2003). Reinforcement learning as classification: Leveraging modern classifiers. *Proceedings of the Int'l Conf on Machine Learning*.

Leith, D., & Leithead, W. (2000). Survey of gain-scheduling analysis and design. *International Journal of Control*, *73*, 1001–1025.

Ng, A. Y., Kim, H. J., Jordan, M., & Russell, S. (2004). Autonomous helicopter flight via reinforcement learning. *Neural Information Processing Systems 16*.

Puterman, M. L. (1994). *Markov decision processes: Discrete stochastic dynamic programming*. Wiley.

Rossetter, E., & Gerdes, J. (2002). Performance guarantees for hazard based lateral vehicle control. *Proceedings of the International Mechanical Engineering Conference and Exposition*.

Sastry, S. (1999). *Nonlinear systems*. Springer.

Tassa, Y., Erez, T., & Smart, W. (2007). Receding horizon differential dynamic programming. *NIPS 20*.

Thrun, S., & al. (2006). Winning the DARPA Grand Challenge. *J. of Field Robotics*. accepted for publication.