# Robot Learning from Demonstration by Constructing Skill Trees

George Konidaris[1,2] Scott Kuindersma[2,3]
Roderic Grupen[3] Andrew Barto[2]

[1]Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge MA
USA
gdk@csail.mit.edu

[2]Autonomous Learning Laboratory
University of Massachusetts Amherst
Amherst MA
USA

[3]Laboratory for Perceptual Robotics
University of Massachusetts Amherst
Amherst MA
USA

**Abstract**

We describe CST, an online algorithm for constructing skill trees from demonstration trajectories. CST segments a demonstration trajectory into a chain of component skills, where each skill has a goal and is assigned a suitable abstraction from an abstraction library. These properties permit skills to be improved efficiently using a policy learning algorithm. Chains from multiple demonstration trajectories are merged into a skill tree. We show that CST can be used to acquire skills from human demonstration in a dynamic continuous domain, and from both expert demonstration and learned control sequences on the uBot-5 mobile manipulator.

# 1 Introduction

If we are to fulfill the promise of robotics and achieve ubiquitous general purpose automation, we must move beyond designing specialized task-specific robots, and toward general purpose commercial machines suitable for a wide range of potentially user-specified applications. Such flexibility, however, poses a challenge: how can such machines be programmed by end-users, rather than engineers?

Learning from demonstration (or LfD) [Argall et al., 2009] is a promising methodology that offers a natural and intuitive approach to robot programming: rather than investing effort into writing a detailed control program, we simply *show* the robot how to achieve a task. This has the immediate advantage of requiring no (or very little) specialized skill or training, and makes use of a human demonstrator's existing procedural knowledge both to identify *which* control program to acquire, and to avoid having to learn a control program from scratch—which is presently infeasible for most tasks of interest on real robots. LfD has consequently received a great deal of attention in recent years.

An important goal of current research in LfD is deriving controllers that are robust to noise and initial conditions, and generalizable so that the resulting controllers can handle reasonable variations in task automatically. To this end we present CST (for *Constructing Skill Trees*), an LfD algorithm with four properties that, taken together, distinguish it from previous work:

1. Rather than converting a demonstration trajectory into a single controller, CST segments demonstration trajectories into a sequence of controllers (which we term *skills*, but are often also called behaviors, motion primitives or options in the literature). Breaking a complex policy into components results in skills that can be reused in other problems, or replayed in a different sequence.

2. CST extracts skills which have *goals*—in particular, the objective of skill $n$ is to reach a configuration where skill $n + 1$ can be successfully executed. Given a cost function, we can therefore use a policy learning algorithm to improve the robot's performance for each skill individually. This is especially important in cases where it is easy to provide a "poor" demonstration but difficult to provide a "good" one. More generally, this is essential in cases where we wish to improve upon the demonstrator's performance. For example, it may be possible to manually control a robot to demonstrate catching only balls traveling very slowly. Ideally, we would like such a robot to use demonstration to obtain an initial baseline policy and then improve it using its own experience to become able to catch fast-moving balls.

3. CST supports the use of *skill-specific abstractions*, where each skill policy is defined using only a small number of relevant state and motor variables. Given a library of candidate abstractions, CST selects the appropriate

abstraction for each component skill. This can greatly reduce the sample complexity of policy improvement and facilitate skill transfer.

4. CST merges skill chains from multiple demonstrations into a *skill tree*, allowing it to deal with collections of trajectories that use different component skills to achieve the same goal, while also determining which trajectory segments are instances of the same policy.

Given an abstraction library, CST segments demonstration trajectories into sequences of skills, where the skill boundaries are determined using *changepoint detection*—a principled statistical method that detects points on the trajectory where either the most relevant abstraction changes, or where the trajectory on both sides of the point is too complex to represent as a single skill. We restrict its per-step computational complexity to a constant using a particle filter, resulting in an online algorithm that processes each data point as it occurs and stores only constant-sized sufficient statistics.

Skill-specific abstractions (though optional, because the abstraction library could simply contain a single "whole problem" abstraction) aid in acquiring policies that are too high-dimensional to be feasibly learned as a single skill. By breaking them into sequences of much simpler policies, we can define each subpolicy using only a small number of relevant variables. Thus, a complex task such as waking up in the morning and driving to work—which requires far too many state variables to acquire efficiently as a single problem—could be broken down into a series of subtasks (getting out of bed, taking a shower, leaving the apartment, starting the car, etc.) that require sufficiently few state variables to be feasible to learn.

# 2 Background

This work relies on several tools and frameworks to model sequential skill execution. We adopt the options framework [Sutton et al., 1999]—a hierarchical reinforcement learning formalism—for representing and reasoning about skills. Our approach builds on prior work based on the notion that each skill has its own policy (represented implicitly by a *value function*) defined using an associated *abstraction* that includes only the relevant features of the environment and ignores the others. Finally, we use a statistical *changepoint detection* algorithm to detect changes in demonstrated policy, which occur when a trajectory segment is best represented using more than one value function, or when the most relevant abstraction changes.

The remainder of this section briefly covers each of these areas in turn, before we proceed to the derivation of the CST algorithm in Sections 3 and 4.

## 2.1 Reinforcement Learning

In the reinforcement learning (RL) [Sutton and Barto, 1998] setting, we usually model the problem of learning a robot control policy as a continuous-state Markov Decision Process (MDP), which can be described as a tuple:

$$M = (S, A, P, \mathcal{R}, \gamma), \tag{1}$$

where $S \subset \mathbb{R}^d$ is a set of possible state vectors, $A$ is either a set of available actions (the discrete action case) or a subset of $\mathbb{R}^n$ (the continuous action case), $P$ is the transition model (with $P(s'|s,a)$ modeling the probability distribution of next states $s'$ given the agent executing action $a$ in state $s$), and $R$ is the reward function (with $R(s,a,s')$ giving the reward obtained from executing action $a$ in state $s$ and transitioning to state $s'$). The objective is to learn a policy $\pi$ that maps state vectors to actions so as to maximize *expected return* from each state $s$:

$$R_\pi(s) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i \middle| s_0 = s\right], \tag{2}$$

where $0 < \gamma \leq 1$ discounts future rewards. Note that although the agent receives a reward $r_i$ for each transition, it aims to maximize the expected discounted sum of future rewards. This mismatch between the agent's immediate feedback and its objective is what makes policy learning difficult in general.

In *episodic* MDPs, some set of states (termed *absorbing states*) cause interaction with the environment to cease and for the agent to receive no further reward. When the agent reaches such a state, we consider it to have completed an episode, and allow it to begin interaction again. Learning then consists of a series of episodes. In this case, we model the absorbing states as states that only allow self-transitions with a reward of zero, and we obtain from each episode a finite-length sample of $R_\pi(s)$, called a Monte Carlo sample of return from state $s$ [Sutton and Barto, 1998]:

$$\bar{R}_\pi(s) = \sum_{i=0}^{n} \gamma^i r_i. \tag{3}$$

We will make extensive use of such samples in this paper.

Two broad classes of policy learning algorithms have found application in robotics. The first class of algorithms, *value function methods*, aim to learn a policy indirectly by representing a value function $V$ that maps a given state vector to expected return, and then selecting actions that result in the state with highest $V$. This suffices for control when $P$ is known. When it is not available, the agent must either learn it, or instead learn an *action-value function*, $Q$, that maps state-action pairs to expected return.[1] $V$ is commonly approximated as a

[1] Because the theory underlying the two cases is similar we consider only the value function case in this paper.

weighted sum of a given set of basis functions $\phi_1, ..., \phi_k$:

$$\hat{V}(\mathbf{x}) = \sum_{i=1}^{k} w_i \phi_i(\mathbf{x}). \tag{4}$$

This is termed *linear value function approximation* because the value function is linear in the vector of weights $\mathbf{w} = [w_1, ..., w_k]$. Learning entails finding the weights corresponding to an approximate optimal value function $\hat{V}^*$. Linear function approximation methods are attractive because they result in simple update rules (often using gradient descent) and possess a quadratic error surface that (except in degenerate cases) has a single minimum. In addition, they can represent complex value functions because the basis functions themselves can be arbitrarily complex.

We use the Fourier basis, a generic basis that generally exhibits good performance, throughout this paper. The $k$th order Fourier Basis for $d$ state variables is the set of basis functions defined as:

$$\phi_i(\mathbf{x}) = \cos\left(\pi \mathbf{c}^i \cdot \mathbf{x}\right), \tag{5}$$

where $\mathbf{c}^i = [c_1, ..., c_d]$, $c_j \in \{0, ..., k\}$, $1 \leq j \leq d$. Each basis function is obtained using a vector $\mathbf{c}^i$ that attaches an integer coefficient (between 0 and $k$, inclusive) to each variable in $\mathbf{x}$ (after that variable has been scaled to $[0, 1]$). The set of basis functions is obtained by enumerating all such $\mathbf{c}^i$ vectors. For more details please see Konidaris et al. [2011b].

The second class of methods, *policy gradient* algorithms, represent $\pi$ directly as a parametrized policy and then ascend the gradient of expected return with respect to its parameters. These methods have found wide applicability in robot applications [Ng et al., 2003, Peters et al., 2003, Kohl and Stone, 2004, Tedrake et al., 2004, Peters and Schaal, 2008, Neumann et al., 2009, Kober and Peters, 2010] because they can easily represent policies defined over continuous actions and can include task constraints or policy structure in $\pi$ directly. When $\pi$ is differentiable, an approximate value function can be used to obtain a low-variance estimator of the policy gradient [Sutton et al., 2000], so that value function approximation is a key step in most policy gradient algorithms. Even when it is not, an approximate value function is still a useful guide to the structure of the policy and contains richer information for use in segmentation than the policy itself (which is often piecewise constant). Therefore, we focus on segmentation using value function approximation.

## 2.2 Hierarchical Reinforcement Learning with Options

The options framework [Sutton et al., 1999] adds to the standard RL framework methods for hierarchical planning and learning using temporally-extended actions. Rather than restricting the agent to selecting actions that take a single time step to complete, it models higher-level decision making using *options*:

actions that have their own policies and which may require multiple time steps to complete. An option, $o$, consists of three components: an *option policy*, $\pi_o$, giving the probability of executing each action in each state in which the option is defined; an *initiation set* indicator function, $I_o$, which is 1 for states where the option can be executed and 0 elsewhere; and a *termination condition*, $\beta_o$, giving the probability of option execution terminating in states where the option is defined. Given an *option reward function* (often just a cost function with a termination reward), determining the option's policy can be viewed as just another RL problem, and an appropriate policy learning algorithm can be applied. Once acquired, a new option can be added to an agent's action repertoire alongside its primitive actions, and the agent chooses when to execute it in the same way.

An option is a useful model for a robot controller: it contains all the information required to determine when a controller can be run (its initiation set), when it is done (its termination condition), and how it performs control (its policy). The use of an option reward function allows us to model robot controllers that can be improved through experience. In the remainder of this paper, we will use the terms *skill* and *option* interchangeably.

## 2.3 Skill Chaining and Abstraction Selection

CST builds on two ideas from hierarchical RL: skill chaining and abstraction selection.

Skill chaining is a skill discovery method for continuous RL domains similar in spirit to pre-image backchaining [Lozano-Perez et al., 1984, Burridge et al., 1999, Tedrake, 2009]. Given a continuous RL problem where the policy is either too difficult to learn directly or too complex to represent monolithically, skill chaining constructs a skill tree such that the agent can obtain a trajectory from every start state to a solution state by executing a sequence (or chain) of acquired skills.

This is accomplished by first creating a skill to reach the problem goal. This skill's policy is learned using an RL algorithm, and its initiation set is learned using a classifier, with training examples obtained by trial and error (states from which the skill successfully executes are used as positive examples, and those from which it executes and fails are used as negative examples). Once that skill is learned, another skill is created whose goal is to reach the initiation set of the first skill. This process is repeated, so that an agent applying it along a single trajectory will create a chain of skills that grows backward from the task goal toward the start region (as depicted in Figure 1). More generally, multiple solution trajectories, noise in control, or stochasticity in the environment will result in skill trees rather than skill chains because more than one option will be created to reach some target events. Eventually, the acquired skills will cover the areas of the state space repeatedly visited by the agent. A more detailed description can be found in Konidaris and Barto [2009b].

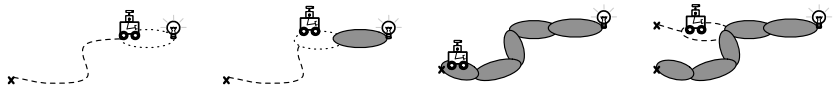



Figure 1: An agent creates options using skill chaining. (a) First, the agent encounters a target event and creates an option to reach it. (b) Entering the initiation set of this first option triggers the creation of a second option whose target is the initiation set of the first option. (c) Finally, after many trajectories the agent has created a chain of options to reach the original target. (d) When multiple options are created that target a single initiation set, the chain splits and the agent creates a skill tree.

Skill chaining provides a mechanism for adaptively representing a complex policy using a collection of simpler policies. Abstraction selection [Konidaris and Barto, 2009a] extends this approach by providing an algorithm for assigning an abstraction to each option. Abstractions are one formalization of the idea that policies often need not depend on *all* of the perceptual and motor resources available to the robot; instead, many policies can be expressed using only a small number of relevant features.

We define an abstraction $M$ to be a pair of functions $(\sigma_M, \tau_M)$, where

$$\sigma_M : S \rightarrow S_M \tag{6}$$

is a *state abstraction* mapping the overall state space $S$ to a smaller state space $S_M$ (often simply a subset of the variables in $S$, but potentially a more complex mapping involving significant feature processing), and

$$\tau_M : A \rightarrow A_M \tag{7}$$

is a *motor abstraction* mapping the full action space $A$ to a smaller action space $A_M$ (often simply a subset of $A$). When performing policy learning using an abstraction, the agent's sensor input is filtered through $\sigma_M$ and its policy $\pi$ maps from $S_M$ to $A_M$.

In addition, we assume that each abstraction has an associated set of basis functions $\Phi_M$ defined over $S_M$ which we can use to define a value function. Therefore, using an abstraction amounts to representing the relevant value function using only that abstraction's set of basis functions.

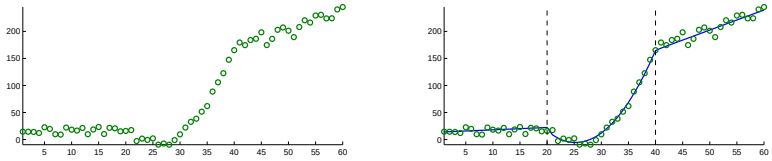
Given a library of candidate abstractions and a set of sample trajectories, abstraction selection finds the abstraction best able to represent the value function inferred from the sample trajectories. It can be combined with skill chaining to acquire a skill tree where each skill has its own abstraction.

## 2.4 Changepoint Detection

Unfortunately, performing skill chaining iteratively is slow because it creates skills incrementally and sequentially. This requires several episodes for a new skill to be learned followed by another several episodes to learn its initiation set by trial-and-error execution of the skill. Rather than creating a skill tree through repeated interaction with the environment, in this work we aim to do so by segmenting demonstration trajectories.

The algorithm we introduce in the following sections is based on statistical changepoint detection, which we now describe in a general regression setting.

We are given observed data and a set of candidate models. We assume that the data are sequentially generated by an instance of a single model, occasionally switching between models at certain points in time, called *changepoints*. We are to infer the number and positions of the changepoints and select and fit an appropriate model instance for each segment. Figure 2 shows a simple example.



(a) (b)

Figure 2: Artificial example data with multiple segments. The observed data (a) are generated by three different models (b) plus noise. The models are shown using solid lines, and the changepoints using dashed lines. The first and third segments are generated by a linear model, whereas the second is quadratic.

Because our data are received sequentially and possibly at a high rate, we would like to perform changepoint detection online—processing transitions as they occur and then discarding them. Fearnhead and Liu [2007] introduced online algorithms for both Bayesian and maximum a posteriori (MAP) changepoint detection. We use the simpler method that obtains the MAP changepoints and models via an online Viterbi algorithm.

Their algorithm proceeds as follows. A set, $Q$, of models is given with prior $p(q)$ for each $q \in Q$. Data tuples $(\mathbf{x}_t, y_t)$ are observed for times $t \in \{1, 2, \ldots, T\}$. The marginal probability of a segment length $l$ is modeled with probability mass function $g(l)$ and cumulative distribution function $G(l) = \sum_{i=1}^{l} g(i)$. Finally, a segment from time $j+1$ to $t$ can be fit using model $q$ to obtain $P(j, t, q)$, the probability of the data segment conditioned on $q$.

This results in a Hidden Markov Model where the hidden state at time $t$ is the model $q_t$ and the observed data is $y_t$ given $\mathbf{x}_t$. The hidden state transition

probability from any model $q_i$ at time $i$ to model $q_j$ at time $j$ is given by:

$$T(q_i, q_j) = g(j - i - 1)p(q_j), \tag{8}$$

representing the probability of a segment of length $j - i - 1$ times the prior for model $q_j$. Note that a transition between two instances of the same model (but with different parameters) is possible.

Similarly, the probability of an observed data segment starting at time $i+1$ and continuing through $j$ using $q$ is given by:

$$P(y_{i+1} : y_j|q) = P(i, j, q)(1 - G(j - i - 1)), \tag{9}$$

modeling the probability of the segment data given model $q$ (obtained by fitting the model to the data) times the probability of a segment lasting at least $j-i-1$ steps.

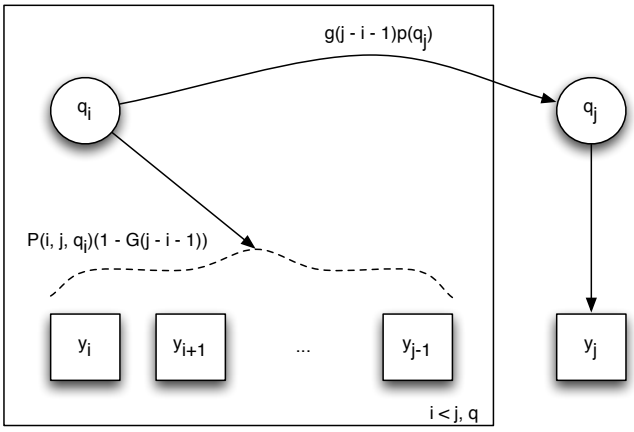



Figure 3: The Hidden Markov Model for changepoint detection. The model $q_t$ at each time $t$ is hidden, but produces observable data $y_t$. Transitions occur when the model changes, either to a new model or the same model with different parameters. The transition from model $q_i$ to $q_j$ occurs with probability $g(j - i - 1)p(q_j)$, while the emission probability for observed data $y_i, ..., y_{j-1}$ is $P(i, j, q_i)(1 - G(j - i - 1))$. These probabilities are considered for all times $i < j$ and models $q_i, q_j \in Q$.

This model is depicted in Figure 3. Notice that all of its transition probabilities are known or computed directly from the data. Rather than attempting to learn the transition probabilities of the hidden states, we are instead trying to compute the maximum likelihood sequence of hidden states given their transition probabilities and the data. We can therefore use an online Viterbi algorithm to compute $P_t(j, q)$, the probability of the changepoint previous to time $t$ occurring at time $j$ using model $q$:

$$P_t(j, q) = (1 - G(t - j - 1))P(j, t, q)p(q)P_j^{MAP}, \tag{10}$$

for each $j < t$, which represents the probability of a segment continuing for at least $t - j + 1$ steps, multiplied by the probability of the segment data given model $q$, the prior of $q$, and the probability of the MAP changepoint at time $j$:

$$P_j^{MAP} = \max_{i,q} \frac{P_j(i,q)g(j-i)}{1 - G(j-i-1)}. \tag{11}$$

At time $j$, the $i$ and $q$ maximizing Equation 11 are the MAP changepoint position and model for the current segment, respectively. This procedure is repeated for time $i$ and continued until time 1 is reached to obtain the changepoints and models for the entire sequence. Thus, at each time step $t$ the algorithm computes $P_t(j,q)$ for each model $q$ and changepoint time $j < t$ (using $P_j^{MAP}$) and then computes and stores $P_t^{MAP}$.[2] This requires $O(T)$ storage and $O(TL|Q|)$ time per timestep, where $L$ is the time required to compute $P(j,t,q)$.

Because most $P_t(j,q)$ values will be close to zero, we can employ a particle filter to discard most combinations of $j$ and $q$ and retain a constant number per timestep. Each particle then stores $j$, $q$, $P_j^{MAP}$, sufficient statistics and its Viterbi path. We use the Stratified Optimal Resampling algorithm of Fearnhead and Liu [2007] to filter down to $M$ particles whenever the number of particles reaches $N$.

In addition, for most models of interest $L$ can be reduced to a constant by storing a small sufficient statistic and updating it incrementally in time independent of $t$, obtaining $P(j,t,q)$ from $P(j,t-1,q)$. This results in a time complexity of $O(NL)$ and storage complexity of $O(Nc)$, where there are $O(c)$ changepoints in the data. The resulting algorithm is given in Figure 4.

# 3 Segmenting a Trajectory into a Skill Chain

Our aim in this work is to develop a method that can segment demonstration trajectories into skill chains and merge skill chains from multiple demonstrations into a skill tree. Recall that a problem is broken into multiple skills in skill chaining because it either consists of policies that use different abstractions, or it consists of policies that are too complex to be approximated using a single function approximator.

These conditions are analogous to the conditions with which we segment data using changepoint detection: we wish to detect when either the model *(i.e., abstraction)* changes or a when trajectory segment is too complex to fit using a single instance of the same model *(i.e., using a single value function)*.

Therefore, we propose segmenting a trajectory into a skill chain by performing changepoint detection: we form $Q$ using the set of basis functions associated with each abstraction as each candidate model, and the sample return $R_t = \bar{R}(s_t)$ (see Equation 3) at time $t$ as the target variable $y_t$.

[2] In practice all equations are computed in log form to ensure numerical stability.

```
Initialization
particles = ∅
Process each incoming data point
for t = 1:T do
    Compute fit probabilities for all particles
    for p ∈ particles do
        p_tjq = (1 - G(t - p.pos - 1)) × p.fit_prob() × prior(p.model) ×
        p.prev_MAP
        p.MAP = p_tjq × g(t - p.pos) / (1 - G(t - p.pos - 1))
    Filter if necessary
    if |particles| >= N then
        particles = filter(particles, particles.MAP, M)
    Determine the Viterbi path
    if t == 1 then
        max_path = []
        max_MAP = 1/|Q|
    else
        max_particle = max_p p.MAP
        max_path = max_particle.path ∪ max_particle
        max_MAP = max_particle.MAP
    Create new particles for a changepoint at time t
    for q ∈ Q do
        new_p = create_particle(model = q, pos = t, prev_MAP =
        max_MAP, path = max_path)
        particles = particles ∪ new_p
    Update all particles
    for p ∈ particles do
        p.update_particle(x_t, y_t)
Return the most likely path to the final point.
return max_path
```

Figure 4: Fearnhead and Liu's online MAP changepoint detection algorithm.

This is a natural mapping to RL because we are thus performing changepoint detection *on the value function sample obtained from the trajectory*. Segmentation thus breaks that value function sample up into simpler segments, or detects a change in model (and therefore abstraction). This is depicted in Figure 5.

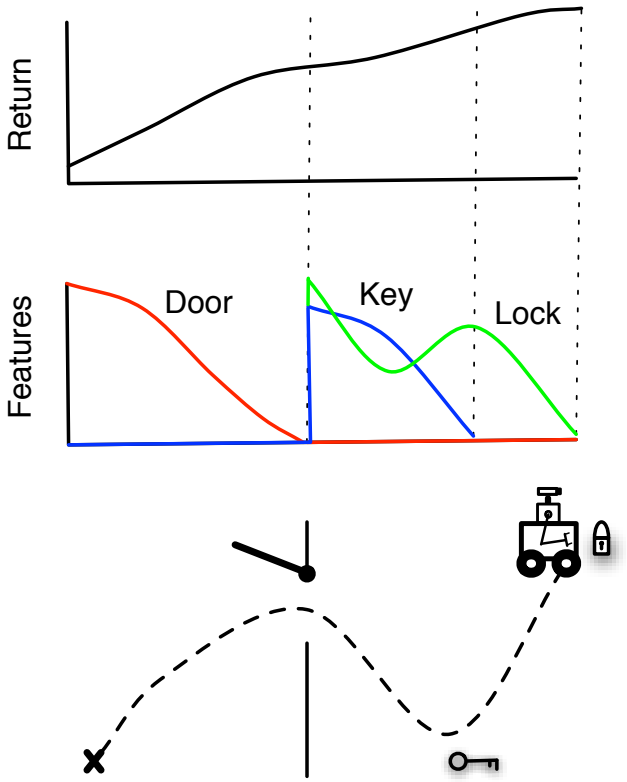



Figure 5: An illustration of a trajectory segmented into skills by CST. A robot executes a trajectory where it goes through a door, approaches and picks up a key, and then takes the key to a lock (bottom). The robot is equipped with three possible abstractions: state variables giving its distance to the doorway, the key, and the lock, respectively. The values of these variables change during trajectory execution (middle) as the distance to each object changes while it is in the robot's field of view. The robot also obtains a sample of return for each point along the trajectory (top). CST splits the trajectory into segments by finding a MAP segmentation such that the return estimate is best represented by a piecewise linear value function where each segment is defined over a single abstraction. Changepoints are indicated by dashed vertical lines.

This requires that an appropriate model of expected skill (segment) length, and an appropriate model for fitting the data. We assume a geometric distribution for skill lengths with parameter $p$, so that $g(l) = (1-p)^{l-1}p$ and $G(l) = (1-(1-p)^l)$. This gives us a natural way to set $p$ via $k = 1/p$, the expected skill length.[3]

[3]Although we assume all skills have the same expected length, the model could also be modified to have different expected skills lengths for different abstractions. For example, skills

Because RL in continuous state spaces usually employs linear function approximation, it is natural to use a linear regression model with Gaussian noise as our model of the data. Following Fearnhead and Liu [2007], we assume conjugate priors: the Gaussian noise prior has mean zero and an inverse gamma variance prior with parameters $\frac{v}{2}$ and $\frac{u}{2}$.[4] The prior for each weight is a zero-mean Gaussian with variance $\sigma^2\delta$. Integrating the likelihood function over the parameters results in:

$$P(j,t,q) = \frac{\pi^{-\frac{n}{2}}}{\delta^m}|(\mathbf{A}_q + \mathbf{D})^{-1}|^{\frac{1}{2}} \frac{u^{\frac{v}{2}}}{(y_q+u)^{\frac{n+v}{2}}} \frac{\Gamma(\frac{n+v}{2})}{\Gamma(\frac{v}{2})}, \tag{12}$$

where $n = t - j$, $q$ has $m$ basis functions, $\Gamma$ is the Gamma function, $\mathbf{D}$ is an $m$ by $m$ matrix with $\delta^{-1}$ on the diagonal and zeros elsewhere, and:

$$\mathbf{A}_q = \sum_{i=j}^{t} \Phi_q(\mathbf{x_i})\Phi_q(\mathbf{x_i})^T \tag{13}$$

$$y_q = (\sum_{i=j}^{t} R_i^2) - \mathbf{b}_q^T(\mathbf{A}_q + \mathbf{D})^{-1}\mathbf{b}_q, \tag{14}$$

where $\Phi_q(\mathbf{x_i})$ is a vector of the $m$ basis functions associated with $q$ evaluated at state $\mathbf{x_i}$, $R_i = \sum_{j=i}^{T} \gamma^{j-i} r_j$ is the return sample obtained from state $i$, and $\mathbf{b}_q = \sum_{i=j}^{t} R_i \Phi_q(\mathbf{x_i})$. $P(j,t,q)$ from Equation 12 is the probability of the data segment from time $t$ to time $j$ conditioned on model $q$, and can be used in Equation 10.

Note that we are using each $R_t$ as the target regression variable in this formulation, even though we only observe $r_t$ for each state. However, to compute Equation 12 we need only retain sufficient statistics $\mathbf{A}_q$, $\mathbf{b}_q$ and $(\sum_{i=j}^{t} R_i^2)$ for each model. Each can be updated incrementally using $r_t$ (the latter two using traces). Thus, the sufficient statistics required to obtain the fit probability can be computed incrementally and online at each timestep, without storing any transition data. The algorithm for this update is given in Figure 6.

Note that $\mathbf{A}_q$ and $\mathbf{b}_q$ are the same matrices used for performing a least-squares fit to the data using model $q$ and $R_t$ as the regression target. They can thus be used to produce a value function fit (equivalent to a least-squares Monte Carlo estimate) for the skill segment if so desired; again, without the need to store the trajectory.

In practice, segmenting a sample trajectory should be performed using a lower-order function approximator than is to be used for policy learning, because the

that involve manipulation objects with a gripper might be expected to take less time than skills that involve the robot traveling across a building.

[4] These parameters may seem cryptic, but they can be set indirectly using an expected variance $\sigma_v^2$ and scaling parameter $\beta_v$. The scaling parameter controls how sharply the distribution is peaked around $\sigma_v^2$; values closer to zero indicate a flatter distribution. We can then set $u = \sigma_v^2 + \beta_v$ and $v = \frac{\beta_v}{\sigma_v^2} - 1$.

**input** : $\mathbf{x}_t$: the current state
$r_t$: the current reward

*Initialization*
**if** *t == 0* **then**
  $\mathbf{A}_q$ = zero_matrix(q.m, q.m)
  $\mathbf{b}_q$ = zero_vector(q.m)
  sum_r$_q = 0$
  $\mathbf{z}_q$ = zero_vector(q.m)
  tr_1$_q$, tr_2$_q = 0$;
*Compute the basis function vector for the current state*
$\Phi_t = \Phi_q(\mathbf{x_t})$
*Update sufficient statistics*
$\mathbf{A}_q = \mathbf{A}_q + \Phi_t \Phi_t^T$
$\mathbf{z}_q = \gamma \mathbf{z}_q + \Phi_t$
$\mathbf{b}_q = \mathbf{b}_q + r_t \mathbf{z}_q$
tr_1$_q = 1 + \gamma^2$ tr_1$_q$
sum_r$_q$ = sum_r$_q + r_t^2$ tr_1$_q + 2\gamma r_t$ tr_2$_q$
tr_2$_q = \gamma$ tr_2$_q + r_t$tr_1$_q$

Figure 6: Incrementally updating the changepoint detection sufficient statistics for model $q$.

agent sees merely a single trajectory sample rather than a dense sample over the state space.

# 4 Merging Skill Chains into a Skill Tree

The above algorithm segments a single trajectory into a skill chain. Given multiple skill chains from different trajectories, we would like to merge them into a skill tree by determining which pairs of trajectory segments belong to the same skills and which are distinct.

Because we wish to build skills that can be sequentially executed, we can only consider merging two segments when they have the same target—which means that their goals are either to reach the initiation set of the same target skill, or to reach the same final goal. This means that we can consider merging the final segment of each trajectory, or two segments whose successor segments have been merged. Thus, two chains are merged by starting at their final skill segments. Each pair of segments are merged if they are a good statistical match. This process is repeated until a pair of skill segments fail to merge, after which the remaining skill chains branch off on their own. This process is depicted in

Figure 7. A similar process can be used to merge a chain into an existing tree by following the chain with the highest merge likelihood when a branch in the tree is reached.

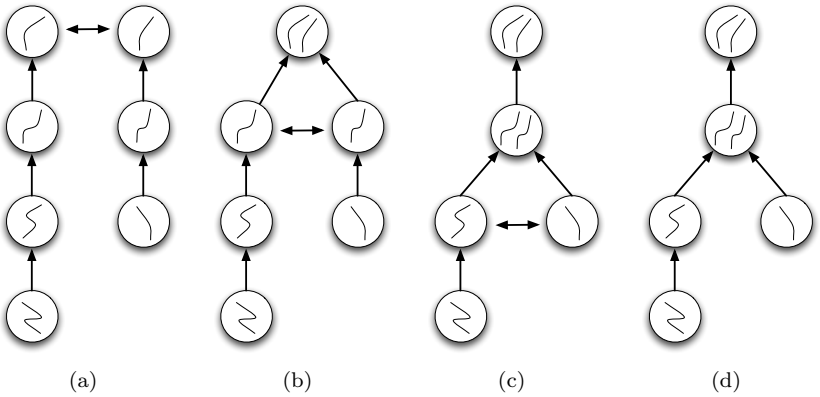



Figure 7: Merging two skill chains into a skill tree. First, the final trajectory segment in each of the chains is considered (a). If these segments use the same model, overlap, and can be well represented using the same function approximator, they are merged and the second segment in each chain can be considered (b). This process continues until it encounters a pair of segments that should not be merged (c). Merging then halts and the remaining skill chains form separate branches of the tree (d).

Because $P(j, t, q)$ as defined in Equation 12 is the integration over its parameters of the likelihood function of model $q$ given segment data, we can reuse it as a measure of whether a pair of trajectories are better modeled as one skill or as two separate skills. Given sufficient statistics $\mathbf{A}_a, \mathbf{b}_a$ and sum of squared return $R_a$ from segment $a$ (having $n_a$ transitions) and $\mathbf{A}_b, \mathbf{b}_b$ and $R_b$ from segment $b$ (having $n_b$ transitions), the probability of both data segments given a single skill model can be computed by evaluating Equation 12 using the sum of these quantities: $\mathbf{A}_{ab} = \mathbf{A}_a + \mathbf{A}_b$, $\mathbf{b}_{ab} = \mathbf{b}_a + \mathbf{b}_b$, $R_{ab} = R_a + R_b$, and $n_{ab} = n_a + n_b$. Note that this model uses the same number of basis functions ($m$) as either model in isolation.

The probability of data segments $a$ and $b$ coming from two different skill models can be evaluated using Equation 12 with $R_{ab} = R_a + R_b$ and $n_{ab} = n_a + n_b$ as before, but with:

$$\mathbf{A}_{ab} = \begin{bmatrix} \mathbf{A}_a & 0 \\ 0 & \mathbf{A}_b \end{bmatrix}, \text{ and } \mathbf{b}_{ab} = \begin{bmatrix} \mathbf{b}_a \\ \mathbf{b}_b \end{bmatrix}. \tag{15}$$

This models the situation where each segment is given its own set of basis functions. It is equivalent to using a larger set of basis functions, $\Phi_{ab} = [\Phi_a, \Phi_b]^T$, where the basis functions from each segment are each non-zero only in their own

segments. Although we may find a better fit (in terms of error) using the two sets of basis functions independently, because we have a higher number of basis functions ($m$ is twice as large), we obtain a higher probability only when the two segments really are much better fit separately. This occurs because the overall probability of fit contains a natural Bayesian penalty for higher-dimensional models.

When considering a merge between more than two segments (as occurs when merging a chain into a tree at points where the tree has already split), a similar operation is performed that evaluates the total probability of the data in all segments given that a pair of segments have merged and the remainder are independent, evaluated for all candidate merging pairs and the case where no merge occurs. This is necessary so that the probabilities obtained from Equation 12 for each case are over the same data, and therefore comparable.

Before merging, a fast boundary test is performed to ensure that the trajectory pairs actually overlap in state space—if they are completely spatially disjoint, we will often be able to represent them both simultaneously with very low error and hence this metric may incorrectly suggest a merge.

If we are to merge skills obtained over multiple trajectories into trees we require the component skills to be aligned, meaning that the changepoints occur in roughly the same places. This will occur naturally in domains where changepoints are primarily caused by a change in relevant abstraction. When this is not the case, the changepoint positions may vary because segmentation is then based on function approximation boundaries, and hence two trajectories segmented independently may be poorly aligned. Therefore, when segmenting two trajectories sequentially in anticipation of a merge, we may wish to include a bias on changepoint locations in the second trajectory. We model this bias as a Mixture of Gaussians, centering an isotropic Gaussian at each location in state-space where a changepoint previously occurred. This bias can be included during changepoint detection by multiplying Equation 10 with the resulting PDF evaluated at the current state.

Note that, although segmentation is performed using a lower-order function approximator than skill policy learning, merging should be performed using the same function approximator used for learning. This necessitates the maintenance of two sets of sufficient statistics during segmentation. Fortunately, the major computational expense is computing $P(j, t, q)$, which during segmentation is only required using the lower-order approximator.

# 5 Acquiring Skills from Human Demonstration in the Pinball Domain

In this section, we evaluate the performance benefits obtained using a skill tree generated from a pair of human-provided solution trajectories in the Pinball

domain. Because the domain is relatively small (4 continuous state variables), we do not use an abstraction library.

## 5.1 The Pinball Domain

The Pinball domain is a continuous domain with dynamic aspects, sharp discontinuities, and extended control characteristics that make it difficult for control and function approximation.[5] Previous experiments have shown that skill chaining is able to find a very good policy while flat learning finds a poor solution [Konidaris and Barto, 2009b]. In this section, we evaluate the performance benefits obtained using a skill tree generated from a pair of human-provided solution trajectories, as opposed to performing skill chaining incrementally.

The goal of PinBall is to maneuver the small ball (which starts in one of two places) into the large red hole. The ball is dynamic (drag coefficient 0.995), so its state is described by four variables: $x$, $y$, $\dot{x}$ and $\dot{y}$. Collisions with obstacles are fully elastic and cause the ball to bounce, so rather than merely avoiding obstacles the agent may choose to use them to efficiently reach the hole. There are five primitive actions: incrementing or decrementing $\dot{x}$ or $\dot{y}$ by a small amount (which incurs a reward of $-5$ per action), or leaving them unchanged (which incurs a reward of $-1$ per action). Reaching the goal obtains a reward of $10,000$. We use the Pinball domain instance shown in Figure 8 with 5 pairs (one trajectory in each pair for each start state) of human expert demonstration trajectories.

Figure 8: The Pinball instance used in our experiments, and a representative solution trajectory pair.

## 5.2 Implementation Details

Initiation sets were learned using a logistic regression classifier. For CST agents, we used only the training examples from the demonstration trajectories for

[5] Java source code for Pinball can be downloaded at: `http://www-all.cs.umass.edu/~gdk/pinball`

learning initiation sets (positive examples are those in the skill segment, negative examples all others). After this initial phase the initiation set classifiers were considered to be learned and only updated incrementally when new negative training examples were received. All skill chaining parameters were as in Konidaris and Barto [2009b].

We used an expected skill length of $k = 100$, $\delta = 0.0001$, particle filter parameters $N = 30$ and $M = 50$, and a first-order Fourier Basis (16 basis functions) for segmentation. We used expected variance mean $\sigma_v^2 = 15^2$ and scaling parameter $\beta_v = 0.0001$ for the noise prior. After segmenting the first trajectory we used isotropic Gaussians with variance $0.5^2$ to bias the segmentation of the second. The full 3rd-order Fourier basis representation was used for merging. To obtain a fair comparison with the linear-time online learning algorithm used in the incremental skill chaining case, we initialized the CST skill policies using 10 episodes of experience replay [Lin, 1991] of the demonstrated trajectories, rather than using the sufficient statistics to perform a batch least-squares value function fit.

## 5.3 Results

Trajectory segmentation was successful for all demonstration trajectories, and all pairs were merged successfully into skill trees when the alignment bias was used to segment the second trajectory in the pair (two of the five could not be merged due to misalignments when the bias was not used). Example segmentations and the resulting merged trajectories are shown in Figure 9, and the resulting initiation sets are shown in their tree structure in Figure 10.

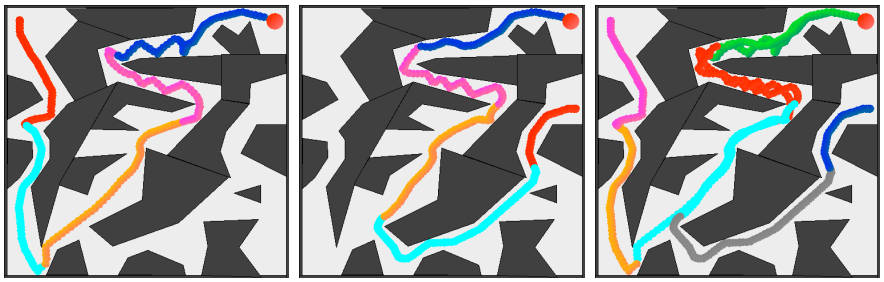


Figure 9: Segmented skill chains from the sample pinball solution trajectories shown in Figure 8, and the trajectory assignments obtained when the two chains are merged.

The learning curves obtained by reinforcement learning agents given the resulting skill trees, averaged over 100 runs (20 runs using each demonstrated skill tree) are shown in Figure 11. We compare these agents against two baselines: one where the agents acquire skills from scratch (using skill chaining), and one where the agents are pre-equipped with all of the skills acquired by skill chaining

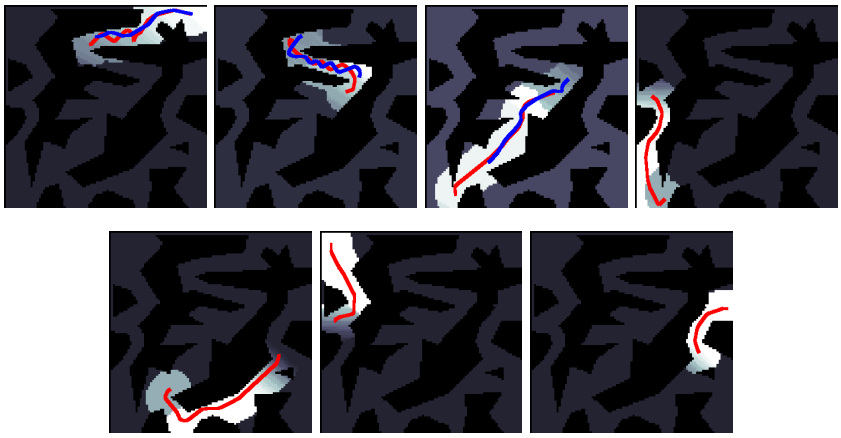


Figure 10: The initiation sets for each option in the tree shown in Figure 9.

over 250 episodes, but not given an overall task policy (and so must learn how to sequence the skills they have been given). The results show that the CST policies are not good enough to use immediately, as the agents do worse than those given pre-learned skills for the first few episodes (although they immediately do better than skill chaining agents). However, very shortly thereafter—by the 10th episode—the CST agents are able to learn excellent policies, immediately performing much better than skill chaining agents, and shortly thereafter actually temporarily exceeding the performance of agents with pre-learned skills. This is likely because the skill tree structure obtained from demonstration has fewer but better skills than that learned incrementally by skill chaining agents, resulting in a faster initial startup while the agents given pre-learned skills learn to correctly sequence them.

In addition, segmenting demonstration trajectories into skills results in much faster learning than attempting to acquire the entire demonstrated policy at once. Agents that perform experience replay using the demonstration trajectories to initialize their overall task value function and then proceed using skill chaining have virtually identical learning curves (not shown) to those of agents performing skill chaining from scratch.

# 6 Acquiring Mobile Manipulation Skills from Human Demonstration

In the previous section, we showed that CST is able to segment demonstration trajectories in Pinball and merge them into a tree suitable as a basis for further learning. However, Pinball is a relatively small domain and therefore did not require the use of skill-specific abstractions.

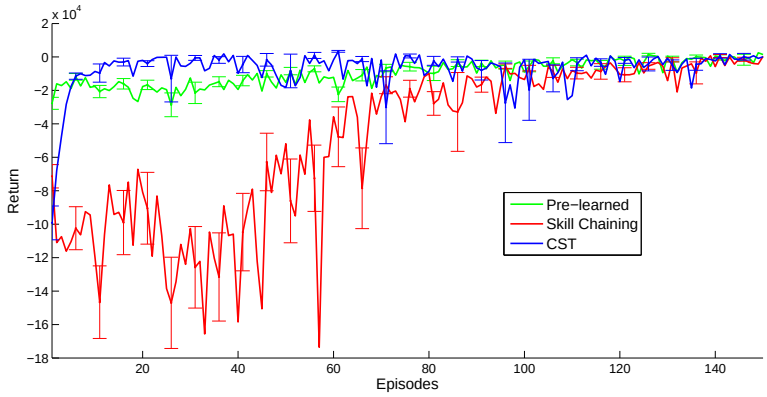



Figure 11: Learning curves in the PinBall domain, for agents employing skill trees created from demonstration trajectories, skill chaining agents, and agents starting with pre-learned skills.

In this section we show that CST can scale up to much higher-dimensional domains using skill-specific abstractions. We apply CST along with an abstraction library to create skill chains from human demonstration on the uBot-5 [Deegan et al., 2006, Kuindersma et al., 2009], a dynamically balancing mobile manipulator. The robot's task (illustrated in Figure 12) in this section is to enter a corridor, approach a door, push the door open, turn right into a new corridor, and finally approach and push on a panel. Twelve demonstration trajectories were obtained from an expert human operator.

## 6.1 Implementation Details

To simplify perception, purple, orange and yellow colored circles were placed on the door and panel, beginning of the back wall, and middle of the back wall, respectively, as perceptually salient markers. The distances (obtained using onboard stereo vision) between the uBot to each marker were computed at 8Hz and filtered. The uBot was able to engage one of two motor abstractions at a time: either performing end-point position control of its hand, or controlling the speed and angle of its forward motion.

Using these features, we constructed six sensorimotor abstractions, one for each pairing of salient object and motor command set. When a marker was paired with the hand, the abstraction's state variables consisted of the real-valued difference between the hand and the marker centroid in 3 dimensions. Actions were real-valued vectors moving the hand in 3 dimensions. When a marker was paired with the robot's torso, the abstraction's state variables consisted of two real values representing the distance and angle to the marker centroid. Actions were real-valued vectors controlling the translation and rotation of the robot

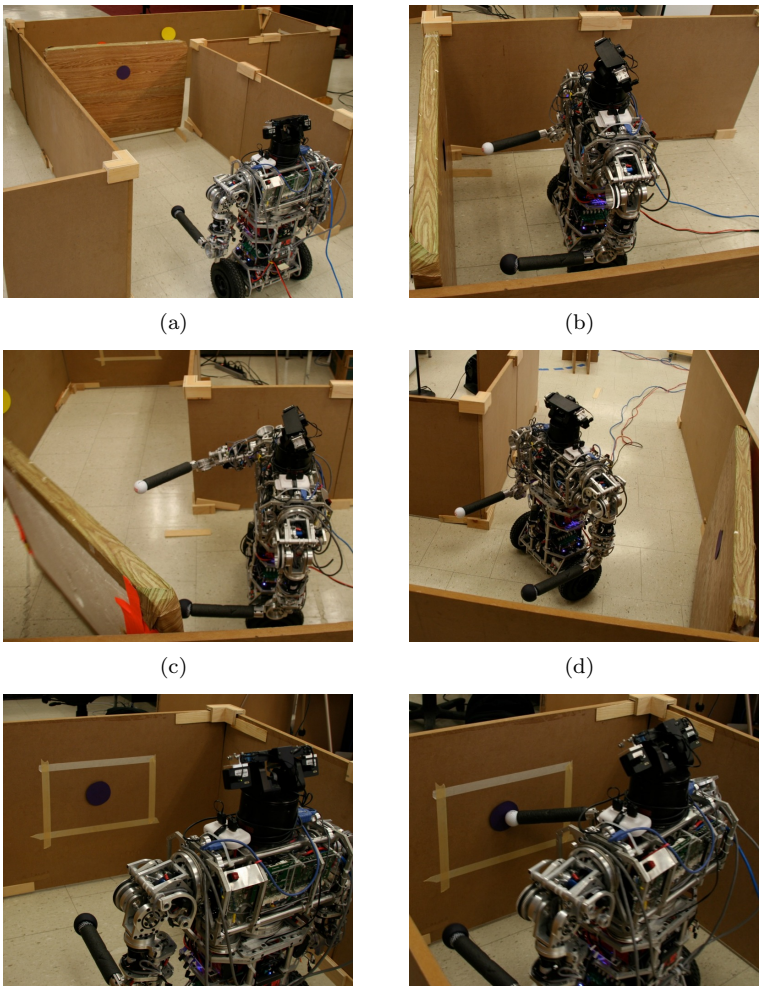


Figure 12: The task demonstrated on the uBot-5. Starting at the beginning of a corridor (a), the uBot approaches (b) and pushes open a door (c), turns through the doorway (d), then approaches (e) and pushes a panel (f).

torso using differential drive. We assumed a reward function of $-1$ received at 8Hz.

Particles were generated according to the currently executing motor abstraction, and a switch in motor abstraction always caused a changepoint.[6] The param-

[6]Informal experiments with removing this restriction did not seem to change the number or type of skills found but in some cases changed their starting and stopping positions by a few timesteps.

eters used for performing CST on the uBot were $k = 50$, $M = 60$, $N = 120$, $\sigma_v^2 = 8^2$ and $\beta_v = 0.00001$, using a 1st order Fourier basis. For merging, we used a 5th order Fourier basis with $\sigma_v^2 = 90^2$ and $\beta_v = 0.00001$.

When performing policy regression to fit the segmented policies for replay, we used ridge regression over a 5th order Fourier basis to directly map to continuous actions. The regularization parameter $\lambda$ was set by 10-fold cross-validation. We then tested how many demonstration trajectories were required to be able to robustly replay the skill policy by adding in one demonstration trajectory at a time, varying the starting point of the robot by hand, and observing policy execution. We used hand-coded stopping conditions corresponding to the initiation set of the subsequent skill.

## 6.2 Results

Of the 12 demonstration trajectories gathered from the uBot, 3 had to be discarded because of data loss due to excess perceptual noise. Of the remaining 9, all segmented sensibly and 8 were able to be merged into a single skill chain.[7]

Figure 13 shows a segmented trajectory obtained using CST, with Table 1 providing a brief description of the skills extracted along with their selected abstractions, and the number of demonstration trajectories required for each skill to be replayed successfully 9 times out of 10.

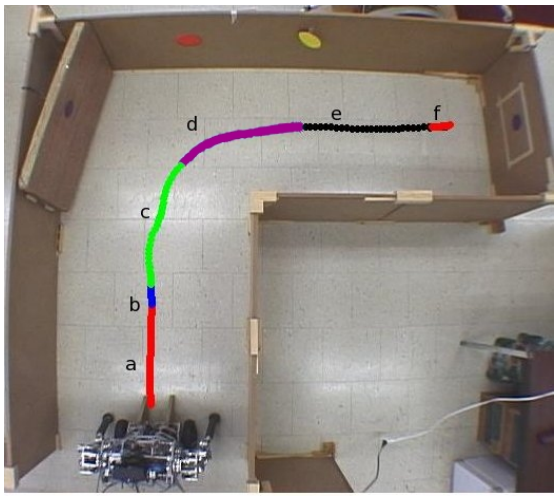



Figure 13: A demonstration trajectory from the uBot-5 segmented into skills.

[7] In four of these trajectories, a delay starting the robot moving after opening the door caused a "wait skill" to appear in the segmentation, where the robot did nothing. We excised these skills before merging.

| # | Abstraction | Description | Trajectories Required |
|---|---|---|---|
| a | torso-purple | Drive to door. | 2 |
| b | hand-purple | Push the door open. | 1 |
| c | torso-orange | Drive toward wall. | 1 |
| d | torso-yellow | Turn toward the end wall. | 2 |
| e | torso-purple | Drive to the panel. | 1 |
| f | hand-purple | Press the panel. | 3 |

Table 1: A brief description of each of the skills extracted from the trajectory shown in Figure 13, along with their selected abstractions, and the number of example trajectories required for accurate replay.

The different numbers of example trajectories required to accurately replay the demonstrated skills occurred because of the varying difficulty of each skill. Pushing the door open proved relatively easy—a general forward motion by the hand toward the purple circle will almost always succeed. By contrast, pushing the panel required greater precision, because the hand was required to be within the panel area when it touched the wall, even when the uBot was facing the wall at an angle. Finally, approaching a target turned out to be difficult in some cases because small mistakes in its angular velocity when the uBot neared the target could cause the target to drift out of the robot's narrow field of view.

A similar experiment that used CST in combination with model-based control methods for obtaining the skill policies was able to produce consistent replay using just a single demonstration trajectory [Kuindersma et al., 2010].

# 7 Acquiring Mobile Manipulation Skills from a Learned Policy

In the previous section we used CST to extract skills from trajectories obtained by expert human demonstration. We now describe a robot system that produces its own demonstration trajectories by learning to sequence existing controllers and extracts skills from the resulting learned solution. This was part of a larger experiment aimed at demonstrating that a robot could perform skill acquisition by sequencing existing skills and then extracting higher-level skills from raw sensorimotor experience [Konidaris et al., 2011a].

In the Red Room task, the uBot-5 is placed in a small room containing a button and a handle. When the handle is pulled after the button has been pressed, a door in the side of the room opens, allowing the robot access to a compartment which contains a switch. The goal of the task is to press the switch. Sensing and control for the objects in the room are performed using touch sensors, with

state tracked and communicated to the uBot via an MIT Handy Board [Martin, 1998]. A schematic and photographs of the domain are given in Figure 14.

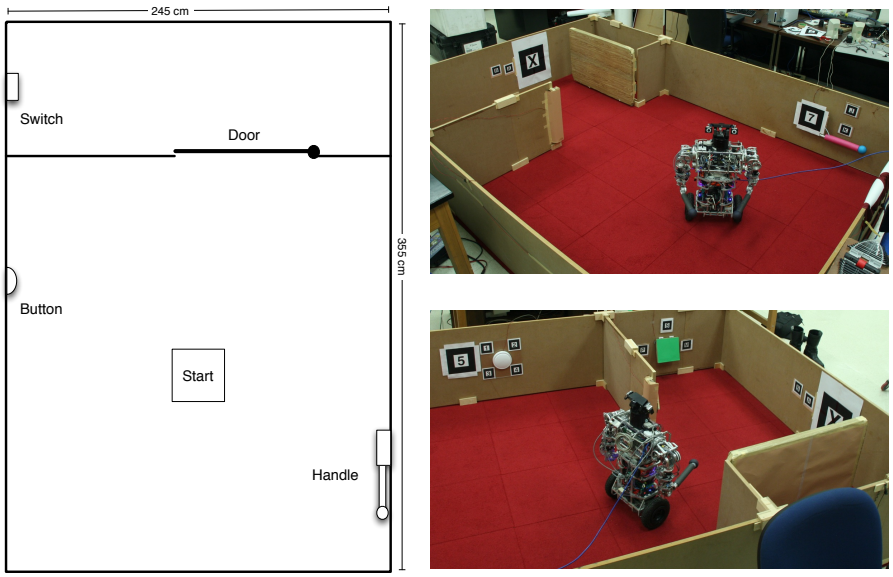



Figure 14: The Red Room Domain.

The robot is given a fixed set of innate controllers for navigating to visible objects of interest and for interacting with them using its end-effector (by extending its arm and moving it to the right, left, up, down, or forwards). In order to actuate the button and the switch, the robot must extend its arm and then move it outwards; in order to actuate the handle, it must extend its arm and then move it downwards. The robot constructed a transition model of the domain through interaction and used it to compute a policy using dynamic programming. The robot was able to find the optimal solution after five episodes, which required roughly one hour of cumulative interaction time with the environment. This reduces the interaction time required to complete one episode of the task from approximately 13 minutes (when the robot starts with no knowledge of the solution) to around 3 minutes. More details are available in Konidaris et al. [2011a].

The resulting optimal sequence of controllers are then used to generate 5 demonstration trajectories for use in CST (using a first-order Fourier Basis, $k = 150$, $M = 60$, $N = 120$, $\sigma_v = 60^2$, and $\beta_v = 0.000001$). The resulting trajectories all segment into the same sequence of 10 skills, and are all merged successfully (using a 5th order Fourier Basis, $\sigma_v = 500^2$, and $\beta_v = 0.000001$). An example segmentation is shown in Figure 15; a description of each skill along with its relevant abstraction is given in Table 2.

CST consistently extracted skills that corresponded to manipulating objects in

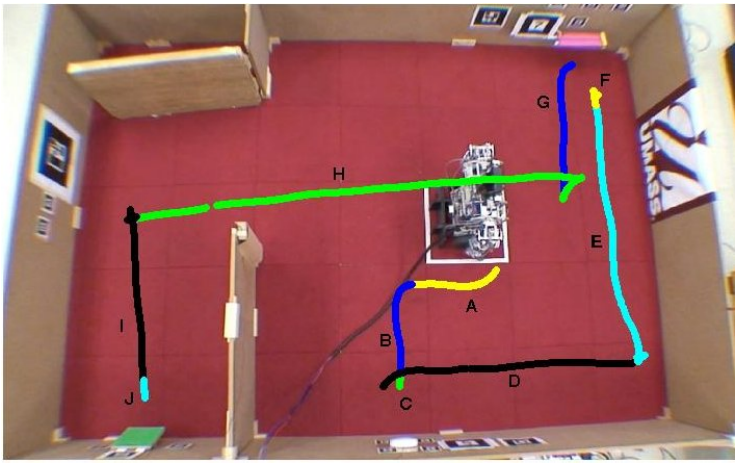



Figure 15: A trajectory from the learned solution to the Red Room task, segmented into skills.

| # | **Abstraction** | **Description** |
|---|---|---|
| A | torso-button | Align with the button. |
| B | torso-button | Turn and approach the button. |
| C | hand-button | Push the button. |
| D | torso-handle | Align with the handle. |
| E | torso-handle | Turn and approach the handle. |
| F | hand-handle | Pull the handle. |
| G | torso-entrance | Align with the entrance. |
| H | torso-entrance | Turn and drive through the entrance. |
| I | torso-switch | Approach the switch. |
| J | hand-switch | Press the switch. |

Table 2: A brief description of each of the skills extracted from the trajectory shown in Figure 15, along with their selected abstractions.

the environment, and navigating towards them. In the navigation case, each controller execution was split into two separate skills. These skills correspond exactly to the two phases of the navigation controller: first, aligning the robot with the normal of the target object, and second, moving the robot toward that feature. In the object-manipulation case, a sequence of two controllers is collapsed into a single skill: for example, extending the arm and then extending it further toward the object of interest is collapsed into a single skill which we might label *push the button*. We constructed closed-loop manipulation policies by fitting splines over relative spatial waypoints and obtained reliable replay using a single demonstration trajectory for each.

# 8 Discussion

Although CST is an online, incremental algorithm, in practice its parameters may require robot-specific tuning. In our experience its performance is robust to all of its parameters (including, within reason, the maximum number of particles) with the exception of the noise prior. When $\sigma_v^2$ is too low the segmentation is "brittle", and when it is too high it is too accommodating—far too high, and it prefers to fit a constant model and treat the trajectory data as noise. Fortunately, these parameters need only be manually set once per domain, although a model of the growth in value function variance as function of the length of the sample trajectory would improve the robustness of the method.

The availability of a suitable abstraction library is key to the application of CST in high-dimensional domains. This requires significant (though in principle once-off) design effort to create a set of abstractions suitable for representing anything the robot may decide to learn. We expect that for many robots and tasks, a small library consisting of pairings of motor abstractions and one or two visible objects will be sufficient. If necessary, abstraction selection could be paired with a feature selection method [Kolter and Ng, 2009, Johns et al., 2010] to augment the selected abstraction during policy learning. Future work may also consider approaches to acquiring the abstraction library from data or building skill-specific abstractions at runtime, although the sample complexity of such approaches may render them impractical.

Another important assumption is that each segmentation should form a chain and the merged chains should form a tree. In some cases, however, they may form a more general graph (e.g., when the demonstrated policy has a loop, or when sequences of corresponding skills are interleaved by sequences of distinct skills). Under such conditions the procedure to merge skill chains will incorrectly fail to merge some skills, although it could be straightforwardly generalized to better accommodate more cases.

Although CST is designed to handle multiple, unstructured demonstrations which begin at different states, we have made the assumption that the demonstrations all lead to the same goal. When this is not the case, we expect that the very first test during skill chain merging will fail, leading to a *skill forest* rather than a skill tree. Extending the procedure for merging trees to this case is straightforward—we can determine which of several possible trees is the most likely fit to a new skill chain in the same way that we determine which of several possible branches in an individual tree is the best match. The skill segmentation process itself would remain unchanged.

We also assume that the domain reward function is observed and that each option reward can be obtained from it by adding in a termination reward. By contrast, *apprenticeship learning* [Abbeel and Ng, 2004] methods infer the reward function from the demonstrated trajectory. Our approach assumes that the robot is minimizing an internal and known cost function (perhaps a measure of energy or time) while achieving a goal specified by the termination reward.

This captures many practical learning by demonstration problems, but not all.

Finally, we assume that the best abstraction for combining a pair of skills is the abstraction selected for representing both individually, from just a single demonstration trajectory. This may not always hold—two skills best represented individually by one abstraction may be better represented together using another (perhaps more general) abstraction; or, a single demonstration trajectory may not contain sufficient information to distinguish between two or more seemingly relevant abstractions (e.g., when the robot and two features are collinear along the entire trajectory). The likelihood of such a failure depends strongly on both the information contained in the demonstration trajectory and the basis function set used for each abstraction. However, because the correct abstraction would presumably be at least competitive during segmentation, such cases can be resolved by considering segmentations or abstractions other than the final MAP selection when merging. Ideally, the trajectories could be segmented jointly—however, it remains unclear how to achieve this without losing the incremental and online properties of CST; the fully Bayesian (as opposed to MAP) but still online changepoint detection algorithm given by Fearnhead and Liu [2007] may be a good starting point for such an extension.

The experiments reported here used three different ways to represent skill policies: value function regression in Pinball, regression on motor output for acquiring policies from a human expert on the uBot, and a trajectory following closed-loop controller for acquiring skills from learned controller sequences on the uBot. Broadly, these representations have progressed from useful in simple policy improvement algorithms and general to requiring more complex policy improvement algorithms but data-efficient and robust. For robust and reliable robot control from demonstration we expect that closed-loop trajectory-based controllers, such as those produced by dynamic motion primitives [Ijspeert et al., 2002, Schaal, 2003], will provide a good balance between learnability and efficiency.

# 9 Related Work

Several RL methods exist for skill acquisition from demonstration trajectories, but only in discrete domains. Representative recent methods are by Mehta et al. [2008] and Zang et al. [2009].

A great deal of work exists under the general heading of LfD (surveyed by Argall et al. [2009]). Most methods learn an entire policy monolithically from data, although some perform segmentation.

A sequence of policies represented using linear function approximators is similar to the notion of a switching (or sequenced) linear dynamical system, where the policy (rather than the value function) is represented as a sequence of linear systems. The approach most closely related to CST is by Dixon and Khosla [2004a], where a demonstration trajectory is segmented into a sequence of linear

dynamical systems using a heuristic measure that causes a segmentation when an error metric exceeds a threshold parameter. Each linear dynamical system is used to derive a convergent controller, with a small region around the final state considered its goal. The algorithm can be run online and was used in conjunction with several other methods to build a mobile robot system that performed LfD by tracking a human user [Dixon and Khosla, 2004b]. This system differs from CST in three ways. First, it does not use skill-specific abstractions, which makes it difficult to scale up to humanoid robots. Second, it segments demonstration trajectories into policies that are linear in the robot's state variables, which is a stronger condition than a value function that is linear in a set of basis functions. Finally, it uses a heuristic method for segmentation.

Chiappa et al. [2009] and Chiappa and Peters [2010] described a principled statistical approach to acquiring motion primitive libraries from data, where the demonstrated trajectories are modeled as Bayesian linear Gaussian state space models. This approach automatically segments the demonstrated data into policies, and it can handle multivariate target variables and models that repeat within a single trajectory. Although these systems have achieved impressive results on real robots, they use computationally intensive batch processing, do not use skill-specific abstractions, and do not result in skills with goals.

Other principled and sophisticated methods exist for learning switching linear dynamical systems from data [Xuan and Murphy, 2007, Fox et al., 2008]. These methods are more complex and computationally intensive than the much simpler changepoint detection method we use, and they have not been used in the context of skill acquisition.

Krüger et al. [2010] use an interesting segmentation method where demonstration trajectories are segmented according to the movement of the objects the robot interacts with, rather than its own motion. This can be viewed as trajectory segmentation using one specific kind of abstraction. The resulting segmentation is used to define a library of parametrized motion primitives.

Another closely related LfD framework is the Performance-Derived Behavior Vocabularies (PDBV) framework [Jenkins and Matarić, 2004], which segments demonstrated data into motion primitives and thereby builds a motion primitive library. Segments are compressed using a dimensionality reduction technique, and clustered into grouped exemplars, from which closed-loop controllers are derived. Our work differs from PDBV in four major aspects. First, PDBV is a batch method, which allows it to identify repeated skills, whereas CST is suitable for online data processing. Second, PDBV discovers a low-dimensional representation (a *motion-manifold*) for each motion primitive, which requires no prior knowledge but may be difficult to scale up to high-dimensional spaces. By contrast, CST selects skill-specific abstractions from a given library, which requires some design effort but eases scaling and transfer. Third, PDBV extracts motion *policies* rather than *goals*, resulting in a motion primitive library that is not amenable to automatic improvement. Finally, PDBV performs segmentation using Kinematic Centroid Segmentation, a heuristic specific to human-like

kinematic motions in free-space, whereas CST uses a more generally applicable and statistically principled but potentially more expensive segmentation method. However, more recent work has used computationally expensive but principled statistical methods [Grollman and Jenkins, 2010, Butterfield et al., 2010] to segment the data into multiple models as a way to avoid perceptual aliasing in the policy.

Kulić et al. [2009] use a principled, online and incremental method to perform segmentation, and use acquired motion primitives to build a hierarchy that can be used to improve later segmentation. Their method (unlike CST) is able to recognize and exploit repeated skills, but does not result in skills with goals and does not select skill-specific abstractions.

To the best of our knowledge, our method is the first that simultaneously performs statistically principled trajectory segmentation, employs skill-specific abstractions, and extracts skills that have goals (and are therefore suitable for further learning).

# 10 Conclusion

The four characteristics that, taken together, distinguish CST from existing algorithms are that it extracts skills, rather than individual controllers, from demonstration; that each skill has a goal; that each skill is allocated its own abstraction; and that multiple demonstrations can be segmented and merged into a skill tree.

In general we expect to obtain demonstration trajectories from unstructured demonstration. Segmenting a demonstration into component skills results in controllers that the robot can reuse in other contexts or in different sequences; skills with both goals and initiation sets are particularly well suited for use in backward chaining planners. In addition, unsuccessful or partial trajectories can still improve skills whose goals were nevertheless reached, and we can apply confidence-based learning methods [Chernova and Veloso, 2007] to each skill individually.

Extracting skills also enables the use of skill-specific abstractions, which are critical for efficient skill representation, policy improvement and generalization. In addition, skills represented using agent-centric features (such as in our uBot example) can be detached from a problem-specific setting and transferred to new problems [Konidaris and Barto, 2007].

Similarly, segmenting a demonstration trajectory into skills provides a natural way to merge multiple unstructured demonstration trajectories by considering merges at the level of each individual skill. Multiple, unstructured demonstrations where it is unclear (to the robot) where the demonstrated policy should really begin and where the demonstrations should overlap are the most natural LfD setting.

Finally, skills that have goals can be refined using policy improvement methods, and simplify the use of robust closed-loop controllers. A human will almost never be able to demonstrate a policy perfectly suited to a robot with a different morphology, motor scheme, and control regime. The goal of LfD should be to provide a satisficing initial policy that the robot can efficiently and autonomously improve over time.

Each of these aspects confers advantages that have enabled the successful use of CST in the three challenging LfD scenarios presented here. Together, they offer a promising avenue of development toward general-purpose robot learning from demonstration.

## Acknowledgments

We thank the members of the Laboratory for Perceptual Robotics for their technical assistance. Andrew Barto and George Konidaris were supported by the Air Force Office of Scientific Research under grant FA9550-08-1-0418. George Konidaris was additionally supported by the Singapore Ministry of Education under a grant to the Singapore-MIT International Design Center. Scott Kuindersma is supported by a NASA GSRP fellowship from Johnson Space Center. Rod Grupen was supported by the Office of Naval Research under MURI award N00014-07-1-0749.

## References

P. Abbeel and A.Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*, 2004.

B. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57:469–483, 2009.

R.R. Burridge, A.A. Rizzi, and D.E. Koditschek. Sequential composition of dynamically dextrous robot behaviors. *International Journal of Robotics Research*, 18(6):534–555, 1999.

J. Butterfield, S. Osentoski, G. Jay, and O.C. Jenkins. Learning from demonstration using a multi-valued function regressor for time-series data. In *Proceedings of the Tenth IEEE-RAS International Conference on Humanoid Robots*, 2010.

S. Chernova and M. Veloso. Confidence-based policy learning from demonstration using Gaussian mixture models. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2007.

S. Chiappa and J. Peters. Movement extraction by detecting dynamics switches and repetitions. In *Advances in Neural Information Processing Systems 23*, pages 388–396, 2010.

S. Chiappa, J. Kober, and J. Peters. Using Bayesian dynamical systems for motion template libraries. In *Advances in Neural Information Processing Systems 21*, pages 297–304, 2009.

P. Deegan, B. Thibodeau, and R. Grupen. Designing a self-stabilizing robot for dynamic mobile manipulation. In *Proceedings of the Robotics: Science and Systems Workshop on Manipulation for Human Environments*, August 2006.

K.R. Dixon and P.K. Khosla. Trajectory representation using sequenced linear dynamical systems. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3925–3930, 2004a.

K.R. Dixon and P.K. Khosla. Learning by observation with mobile robots: a computational approach. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 102–107, 2004b.

P. Fearnhead and Z. Liu. On-line inference for multiple changepoint problems. *Journal of the Royal Statistical Society B*, 69:589–605, 2007.

E.B. Fox, E.B. Sudderth, M.I. Jordan, and A.S. Willsky. Nonparametric Bayesian learning of switching linear dynamical systems. In *Advances in Neural Information Processing Systems 21*, 2008.

D.H. Grollman and O.C. Jenkins. Incremental learning of subtasks from unsegmented demonstration. In *International Conference on Intelligent Robots and Systems*, 2010.

A.J. Ijspeert, J. Nakanishi, and S. Schaal. Learning attractor landscapes for learning motor primitives. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1547–1554, 2002.

O.C. Jenkins and M. Matarić. Performance-derived behavior vocabularies: data-driven acquisition of skills from motion. *International Journal of Humanoid Robotics*, 1(2):237–288, 2004.

J. Johns, C. Painter-Wakefield, and R. Parr. Linear complementarity for regularized policy evaluation and improvement. In *Advances in Neural Information Processing Systems 23*, 2010.

J. Kober and J. Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203, 2010.

N. Kohl and P. Stone. Machine learning for fast quadrapedal locomotion. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 611–616, 2004.

J.Z. Kolter and A.Y. Ng. Regularization and feature selection in least-squares temporal difference learning. In *Proceedings of the 26th International Conference on Machine Learning*, pages 521–528, 2009.

G.D. Konidaris and A.G. Barto. Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, 2007.

G.D. Konidaris and A.G. Barto. Efficient skill learning using abstraction selection. In *Proceedings of the Twenty First International Joint Conference on Artificial Intelligence*, July 2009a.

G.D. Konidaris and A.G. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems 22*, pages 1015–1023, 2009b.

G.D. Konidaris, S.R. Kuindersma, R.A. Grupen, and A.G. Barto. Autonomous skill acquisition on a mobile manipulator. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*, pages 1468–1473, 2011a.

G.D. Konidaris, S. Osentoski, and P.S. Thomas. Value function approximation in reinforcement learning using the Fourier basis. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*, pages 380–385, 2011b.

V. Krüger, D.L. Herzog, S. Baby, A. Ude, and D. Kragic. Learning actions from observations. *IEEE Robotics and Automation Magazine*, 17(2):30–43, 2010.

S.R. Kuindersma, E. Hannigan, D. Ruiken, and R.A. Grupen. Dexterous mobility with the uBot-5 mobile manipulator. In *Proceedings of the 14th International Conference on Advanced Robotics*, June 2009.

S.R. Kuindersma, G.D. Konidaris, R.A. Grupen, and A.G. Barto. Learning from a single demonstration: Motion planning with skill segmentation. In *Proceedings of the NIPS Workshop on Learning and Planning from Batch Time Series Data*, December 2010.

D. Kulić, W. Takano, and Y. Nakamura. Online segmentation and clustering from continuous observation of whole body motions. *IEEE Transactions on Robotics*, 25(5):1158–1166, 2009.

L-J Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National conference on Artificial Intelligence*, pages 781–786, 1991.

T. Lozano-Perez, M.T. Mason, and R.H. Taylor. Automatic synthesis of fine-motion strategies for robots. *The International Journal of Robotics Research*, 3(1):3–24, 1984.

F.G. Martin. *The Handy Board Technical Reference*. MIT Media Lab, Cambridge MA, 1998.

N. Mehta, S. Ray, P. Tadepalli, and T. Dietterich. Automatic discovery and transfer of MAXQ hierarchies. In *Proceedings of the Twenty Fifth International Conference on Machine Learning*, pages 648–655, 2008.

G. Neumann, W. Maass, and J. Peters. Learning complex motions by sequencing simpler motion templates. In *Proceedings of the 26th International Conference on Machine Learning*, 2009.

A.Y. Ng, H.J Kim, M.I. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems 16*, 2003.

J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *Proceedings of the Third IEEE-RAS International Conference on Humanoid Robotics*, 2003.

S. Schaal. Dynamic movement primitives—a framework for motor control in humans and humanoid robots. In *Proceedings of the International Symposium on Adaptive Motion of Animals and Machines*, 2003.

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

R.S. Sutton, D. Precup, and S.P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, pages 1057–1063, 2000.

R. Tedrake. LQR-Trees: Feedback motion planning on sparse randomized trees. In *Proceedings of Robotics: Science and Systems*, pages 18–24, 2009.

R. Tedrake, T.W. Zhang, and H.S. Seung. Stochastic policy gradient reinforcement learning on a simple 3D biped. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, volume 3, pages 2849–2854, 2004.

X. Xuan and K. Murphy. Modeling changing dependency structure in multivariate time series. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, 2007.

P. Zang, P. Zhou, D. Minnen, and C.L. Isbell. Discovering options from example trajectories. In *Proceedings of the Twenty Sixth International Conference on Machine Learning*, pages 1217–1224, 2009.