# Finding and Transferring Policies Using Stored Behaviors

Martin Stolle

CMU-RI-TR-08-27

*Submitted in partial fulfillment of the*
*requirements for the degree of*
*Doctor of Philosophy in Robotics*

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

May 2008

Thesis Committee:
Christopher Atkeson
James Kuffner
Drew Bagnell
Rüdiger Dillmann, University of Karlsruhe

# Abstract

We present several algorithms that aim to advance the state-of-the-art in reinforcement learning and planning algorithms. One key idea is to transfer knowledge across problems by representing it using local features. This idea is used to speed up a dynamic programming based generalized policy iteration.

We then present a control approach that uses a library of trajectories to establish a control law or policy. This approach is an alternative to methods for finding policies based on value functions using dynamic programming and also to using plans based on a single desired trajectory. Our method has the advantages of providing reasonable policies much faster than dynamic programming and providing more robust and global policies than following a single desired trajectory.

Finally we show how local features can be used to transfer libraries of trajectories between similar problems. Transfer makes it useful to store special purpose behaviors in the library for solving tricky situations in new environments. By adapting the behaviors in the library, we increase the applicability of the behaviors. Our approach can be viewed as a method that allows planning algorithms to make use of special purpose behaviors/actions which are only applicable in certain situations.

Results are shown for the "Labyrinth" marble maze and the Little Dog quadruped robot. The marble maze is a difficult task which requires both fast control as well as planning ahead. In the Little Dog terrain, a quadruped robot has to navigate quickly across rough terrain.

# Acknowledgments

I would like to thank my thesis advisor, Chris Atkeson, for his insightful guidance and his ability to look at research from unconventional view points. Chris' ability to "think big picture" and break down mental barriers as to what is possible and what is not has helped me tremendously to grow intellectually and as a researcher.

I would also like to thank the members of my thesis committee for their insightful comments and discussions. In particular, I would like to thank James Kuffner for his insights into and discussions about the intricacies and corner cases of path planning algorithms. I would like to thank Drew Bagnell for his discussions and insights into machine learning and in particular transfer learning algorithms. Finally, I am grateful for having had the opportunity to work with Professor Rüdiger Dillmann and Dr. Tamim Asfour on their Armar III humanoid robot at the University of Karlsruhe. It has helped me widen my horizon to robotic systems that I would not have had the opportunity to work with otherwise.

Additionally, thanks go out to all my colleagues from the Little Dog team. In particular, I would like to thank Joel Chestnutt for letting me use his foot step planner and for sharing his insights into low-level controllers for Little Dog. I would also like to thank Matt Zucker for his collision checking library that I am using extensively and the many opportunities to bounce ideas off of him. Thanks also go out to Hanns Tappeiner for his initial work on training tricks to Little Dog and to Nathan Ratliff. Although not on Little Dog, I would also like to thank my officemate Mike Stilman for lively discussions and his insights into Dynamic Programming. I also thank Suzanne Lyons Muth who has been great at helping with any administrative questions I had as well as Sumitra Gopal for her endless help with packages, speaker phones and the many friendly conversations.

Finally, a big thank you goes out to my parents who have always been

very supportive and have helped me through easy as well as difficult times. Special thanks also go out to my fiancée Sarah Haile whose love and attention has been wonderful and uplifting and for giving me unfailing support. I also thank all my friends from "Potluck Pirates" for all the awesome gatherings, making life in Pittsburgh enjoyable and giving moral support.

# Contents

# List of Figures

# List of Tables

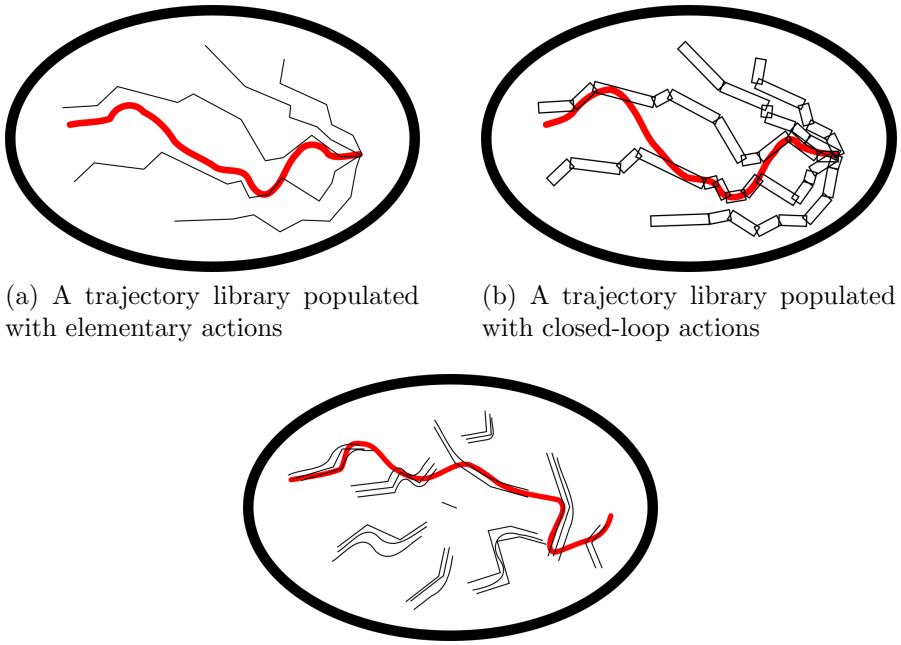
# Chapter 1

# Introduction

## 1.1 Overview

The aim of this thesis is to advance the state-of-the-art in reinforcement learning and planning algorithms so they can be applied to realistic, high-dimensional problems. Our approach is two-pronged:

One part is to enable the reuse of knowledge across different problems in order to solve new problems faster or better, or enable solving larger problems than are currently possible. The key to attaining these goals is to use multiple descriptions of state in a given domain that enable transfer of knowledge as well as learning and planning on different levels of details. In particular, while most domains have a standard state representation such as Cartesian position and velocity with respect to a fixed origin or joint position and velocities, it is sometimes beneficial to also consider ambiguous descriptions using local features of the agent or task. While only short term predictions of the future are possible due to their local and ambiguous nature, they are powerful tools to generalize knowledge across different parts of the environment or to new problems. In order to avoid the limitations of state aliasing, it is however important to use local features in conjunction with a global state representation.



(a) A trajectory library populated with elementary actions

(b) A trajectory library populated with closed-loop actions

(c) A library populated with incongruent behaviors

Figure 1.1: Different types of libraries. The red line illustrates a possible path taken by the agent using the library.

The second key idea is to take advantage of libraries of stored behaviors. In particular, when solving stochastic control problems, often great computational resources are spent on computing globally optimal control laws using Dynamic Programming (DP) or Policy Search. The reasoning is that given knowledge on how to act from any state, one can still behave well under uncertainty: even after an unexpected state transition, the robot knows what is the best action to pick. In some domains, it is possible to avoid this upfront computation by using path planners to quickly obtain a valid solution. In case the original plan becomes infeasible due to the stochasticity of the environment, replanning is performed to find a new solution. However, this much simpler approach is only possible if computers are fast enough so that delays due to replanning are small enough to be permissible. We aim to

close the gap between computing globally optimal policies and replanning by leveraging libraries of trajectories created by path planners. By storing many trajectories in a library, we avoid or reduce replanning while at the same time avoiding the computation required by more traditional methods for finding a globally optimal control law.

Libraries can be implemented in different ways. In the simplest type of library, the planner used to populate the library is reasoning at the level of elementary actions that are directly executed by the agent or sent to motors (figure 1.1(a)). In more complex domains, a library can be populated by planning at the level of more abstract actions (figure 1.1(b)). Low level controllers or heuristics are used to generate the elementary actions when a particular abstract action is executing. Yet another type of library, which is not necessarily created directly from planning, consists of knowledge about behaviors that can be executed in specific parts of the state space (figure 1.1(c)). This third type of library encodes knowledge about possible behaviors, but not all behavior possibilities are necessarily desirable in attaining a specific goal. Hence, a high-level search is necessary to ensure goal-directed behavior.

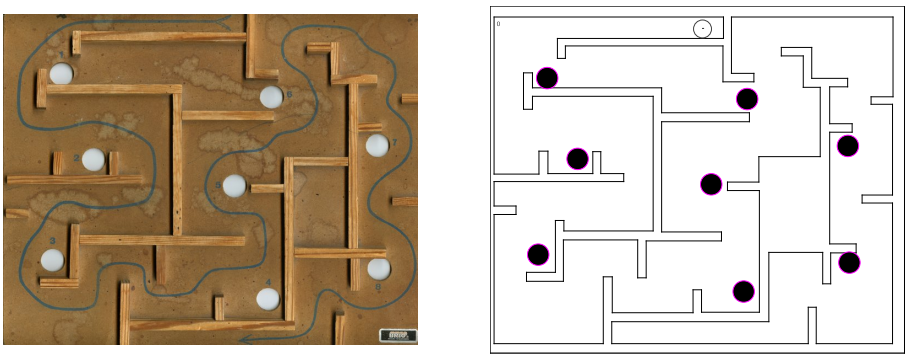
One way the third kind of library is created is when a library of behaviors is transferred to a new environment. Searching through the resulting library of behaviors to find a path can be viewed as augmenting a path planner with external knowledge contained in the library of behaviors. This library of behaviors, describing possibilities for executing special purpose behaviors, enables the path planner to find solutions to hard problems where the models available to the planner are not sufficient to find these behaviors autonomously. Using local features to transfer libraries of behaviors to new problems combines the two key ideas of transfer using local features and libraries of stored behaviors.

This thesis is organized as follows: In the next section, we introduce and describe the experimental domains we are using to validate the ideas and

algorithms. In chapter 2, we describe an algorithm to speed up the creation of global control laws using dynamic programming by transferring knowledge from previously solved problems. Results are presented for simulations in the marble maze domain. In chapter 3, we describe a representation for control laws based on trajectory libraries. Results are shown on both simulated and actual marble maze. Finally, in chapter 4, we propose ways of transferring the libraries presented in chapter 3.

## 1.2 Experimental Domains

### 1.2.1 Marble Maze



(a) original (b) simulation

Figure 1.2: A sample marble maze

Two domains are used to validate and assess the proposed algorithms: the marble maze domain and the Little Dog domain. The marble maze domain (figure 1.2) is also known as "Labyrinth" and consists of a plane with walls and holes. A ball (marble) is placed on a specified starting position and has to be guided to a specified goal zone by tilting the plane. Falling into holes has to be avoided and the walls both restrict the marble and can help it in avoiding the holes. Both a hardware based, computer controlled setup as well as a software simulator are designed and implemented.

The *simulation* uses a four-dimensional state representation $(x, y, dx, dy)$ where $x$ and $y$ specify the 2D position on the plane and $dx$, $dy$ specify the 2D velocity. Actions are also two dimensional $(fx, fy)$ and are force vectors to be applied to the marble. This is not identical but similar to tilting the board. The physics are simulated as a sliding block (simplifies friction and inertia). Collisions are simulated by detecting intersection of the simulated path with the wall and computing the velocity at the time of collision. The velocity component perpendicular to the wall is negated and multiplied with a coefficient of restitution of 0.7. The frictional forces are recomputed and the remainder of the time slice is simulated to completion. Some of the experiments use Gaussian noise, scaled by the speed of the marble and added to the applied force in order to provide for a more realistic simulator and to gauge the robustness of the policies. This noise roughly approximates physical imperfections on the marble or the board. Other noise models could be imagined. A higher-dimensional marble maze simulator was used by Bentivegna [4]. In Bentivegna's simulator the current tilt of the board is also part of the state representation. An even higher fidelity simulator could be created by taking into account the distance of the marble to the rotation axes of the board, because the rotation of the board causes fictitious forces such as centripetal and centrifugal forces on the marble.

The experiments that were performed on the *physical* maze used hobby servos for actuation of the plane tilt. An overhead Firewire 30fps, VGA resolution camera was used for sensing. The ball was painted bright red and the corners of the labyrinth were marked with blue markers. After camera calibration, the positions of the blue markers in the image are used to find a 2D perspective transform for every frame that turns the distorted image of the labyrinth into a rectangle. The position of the red colored ball within this rectangle is used as the position of the ball. Velocity is computed from the difference between the current and the last ball position. Noise in the velocity is quite small compared to the observed velocities so we do not perform

Figure 1.3: The physical maze

filtering. This avoids adding latency to the velocity. As in the simulator, actions are represented internally as forces. These forces are converted into board tilt angles, using the known weight of the ball. Finally, the angles are sent to the servos as angular position.

### 1.2.2 Little Dog

Figure 1.4: Little Dog environment

Another domain we will use for assessing the effectiveness of the algorithms is the Little Dog domain. Little Dog is a quadruped robot developed by Boston Dynamics for DARPA (figure 1.4). It has four legs, each

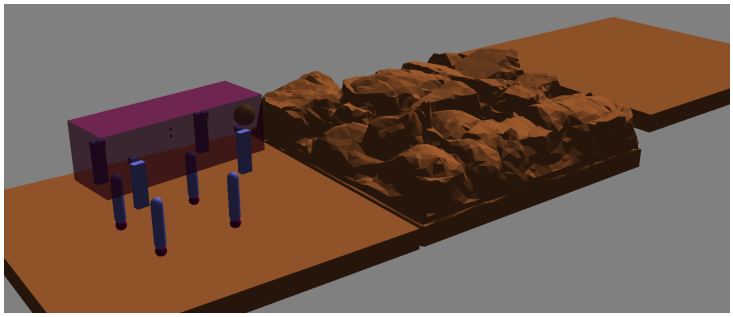


Figure 1.5: Little Dog simulator

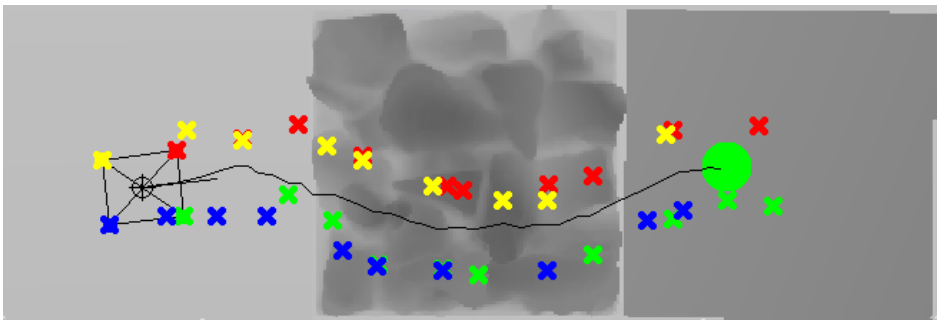


Figure 1.6: Little Dog graphical interface with plan. The black line shows a hypothetical trajectory for the body while the colored crosses correspond to stance locations of the feet (red=front left, green=front right; yellow=hind left, blue=hind right). The dog moves from left to right and the green circle marks the goal for the plan

with three actuated degrees of freedom. Two degrees of freedom are at the hip (inward–outward, forward–backward) and one at the knee (forward–backward). Torque can be applied to each of the joints. This results in a 12 dimensional action space (three for each of the four legs). The state space is 36 dimensional (24 dimensions for the position and velocity of the leg joints and 12 dimensions for the position, orientation, linear velocity and angular velocity of the center of mass). The task to be solved in this domain is to navigate a small-scale rough terrain.

The robot is controlled by sending desired joint angles to an on-board proportional-derivative (PD) controller for each joint. A PD controller sends desired torques to a motor proportional to the error of the joint angle while

subtracting torque proportional to the speed at which the error is decreasing. The desired joint angles can be updated at 100Hz. The on-board PD controller computes new torque outputs at 500Hz. The robot is localized using a Vicon motion capture system which uses retro-reflective markers on the robot in conjunction with a set of six infrared cameras. Additional markers are located on the terrain boards. The proprietary Vicon software provides millimeter accuracy location of the robot as well as the terrain boards. We are supplied with accurate 3D laser scans of the terrain boards. As a result, no robot sensor is needed to map the terrain.

The user interface shown in figure 1.6 is used for monitoring the controllers and drawing plans as well as execution traces. Program data is superimposed over a heightmap of the terrain.

# Chapter 2

# Transfer of Policies Based on Value Functions[1]

## 2.1 Introduction

Finding policies, a function mapping states to actions, using dynamic programming (DP) is computationally expensive, especially in continuous domains. The alternative of computing a single path, although computationally much faster, does not suffice in real world domains where sensing is noisy and perturbations from the intended paths are expected. When solving a new task in the same domain, planning algorithms typically start from scratch. We devise an algorithm which decreases the computation needed to find policies for new tasks based on solutions to previous tasks in the same domain. This is accomplished by initializing a policy for the new task based on policies for previous tasks.

As policies are often expressed using state representations that do not generalize across tasks, policies cannot be copied directly. Instead, we use local features as an intermediate description which generalizes across tasks. By way of these local features, policies can be translated across tasks and

[1] Published in [45]

used to seed planning algorithms with a good initial policy.

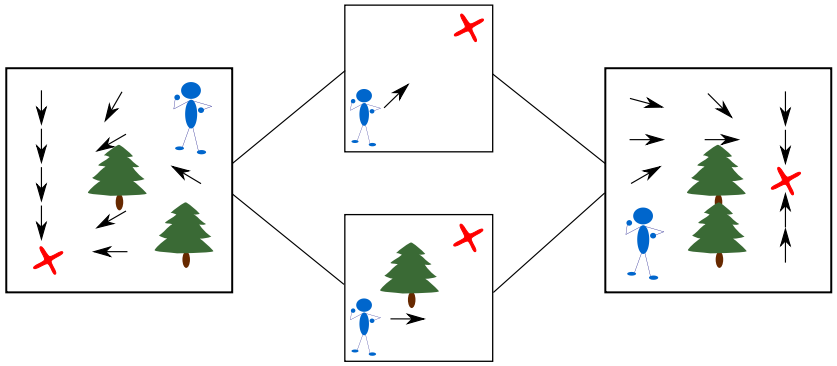


Figure 2.1: Example navigation domain, left: original terrain, middle: feature-based policy, right: new terrain

For example, in a navigation domain, a policy is usually defined in terms of $(x, y)$ coordinates. If the terrain or goal changes, the same $(x, y)$ position will often require a different action. For instance, on the left terrain in figure 2.1, the policy of the upper left corner is to go down, whereas in the right terrain the policy of the same position is to go right. However, one can represent the policy in terms of local features that take into account the position of the agent with respect to the goal and obstacles. A new policy is initialized by looking up what the local features are for each state and setting the action of that state to the action that is associated with the local features. By reverting back to the global $(x, y)$-type state representation, the policy can be refined for the new task without being limited by the local state description.

## 2.2 Related Work

The transfer of knowledge across tasks is an important and recurring aspect of artificial intelligence. Previous work can be classified according to the type of description of the agent's environment as well as the variety of environments the knowledge can be transferred across. For symbolic planners and problem

solvers, the high level relational description of the environment allows for transfer of plans or macro operators across very different tasks, as long as it is still within the same domain. Work on this goes back to STRIPS [16], SOAR [26], Maclearn [22] and analogical reasoning with PRODIGY [48]. More recent relevant work in planning can be found in [14, 51].

In controls, work has been done on modeling actions using local state descriptions [9,32]. Other work has been done to optimize low-level controllers, such as walking gaits, which can then be used in different tasks [8, 25, 40, 50]. That work focuses on cyclic policies that typically don't take into account features of the terrain and are meant to be used as a low-level behavior with a higher-level process governing direction of the walk.

Some work has been done in automatically creating macro-actions in reinforcement learning [33, 34, 41, 46], however those macro actions could only transfer knowledge between tasks where only the goal was moved. If the environment was changed, the learned macro actions would no longer apply as they are expressed in global coordinates, a problem we are explicitly addressing using the local state description. Another method for reusing macro actions in different states by discovering geometric similarities of state regions (homomorphism) can be found in [37].

At the intersection of planning and control, work in relational reinforcement learning creates policies that operate on relational domain state descriptions [15, 53]. Applying the policy to the planning domain is expected to either solve the planning query (which explicitly encodes the goal state) or guide a search method. The policy is learned only once for a given domain and is reused for different planning queries. Similar to traditional work in relational planning, the policies derive their ability to generalize from the relational domain description. Work that is more closely related to ours in solving relational Markov Decision Processes (MDP) can be found in [20]. Like in our approach, a domain expert creates an alternative state description. This state description allows for the creation of a factored MDP over

classes. Every class has a value function associated with it that depends on the state of an instance of that class. For a particular environment, the value of a state is the sum of the value of each instance of every class. This allows for generalization to new problems, assuming the state of the instances contains the information necessary to generalize.

Finally, a related area of research is multi-task learning [7]. The idea behind multi-task learning is that a machine learning algorithm (originally neural networks) can learn faster if it learns multiple related tasks at the same time. There are two ways to look at this: One way is that the input to the machine learning algorithm is very high dimensional and by learning multiple tasks at the same time, the machine learning algorithm can learn which state features are relevant. When learning new tasks, the algorithm can focus learning on those features. Alternatively, one can hope for the machine learning algorithm to compute new relevant features from the given input features. In navigational domains, this would require the whole map to be part of the input state, which would dramatically increase the size of the state space. It is unclear what kind of relationships between original state (such as position) and maps would be learned.

## 2.3 Case Study: Marble Maze

We used the marble maze domain (figure 1.2) to gauge the effectiveness of our knowledge transfer approach. The model used for dynamic programming is the simulator described in section 1.2.1. The reward structure used for reinforcement learning in this domain is very simple. Reaching the goal results in a large positive reward. Falling into a hole terminates the trial and results in a large negative reward. Additionally, each action incurs a small negative reward. The agent tries to maximize the reward received, resulting in policies that roughly minimize the time to reach the goal while avoiding holes.

Solving the maze from scratch was done using value iteration. In value iteration, dynamic programming sweeps across all states and performs the following update to the value function estimate $V$ for each state $s$:

$$V^{t+1}(s) = \max_a \{r(s,a) + V^t(s(a))\} \tag{2.1}$$

where $a$ ranges over all possible actions, $r(s,a)$ is the reward received for executing $a$ in state $s$ and $s(a)$ is the next state reached after $a$ is executed in state $s$.

The simulator served as the model for value iteration. The state space was uniformly discretized and multi-linear interpolation was used for the value function [13].

The positional resolution of the state space was 3mm and the velocity resolution was 12.5mm/s. The mazes were of size 289mm by 184mm and speeds between -50mm/s to +50mm/s in both dimensions were allowed, resulting in a state space of about 380,000 states. This resolution is the result of balancing memory requirements and accuracy of the policy. At coarser resolution, values in some parts of the state space were inadequately resolved, resulting in bad policies. Variable resolution methods such as [35] could be used to limit high-resolution representation to parts of the space where it is strictly necessary. The maximum force on the marble in each dimension was limited to 0.0014751N and discretized into -.001475N, 0 and +.001475N in each dimension, resulting in 9 possible actions for each state. With a simulated mass of the marble of .0084kg, maximal acceleration was about 176mm/s$^2$ in each dimension. Time was discretized to 1/60th of a second.

### 2.3.1 Local State Description

The local features, chosen from the many possible local features, depicts the world as seen from the point of view of the marble, looking in the direction it is rolling. Vectors pointing towards the closest hole, the closest wall as well

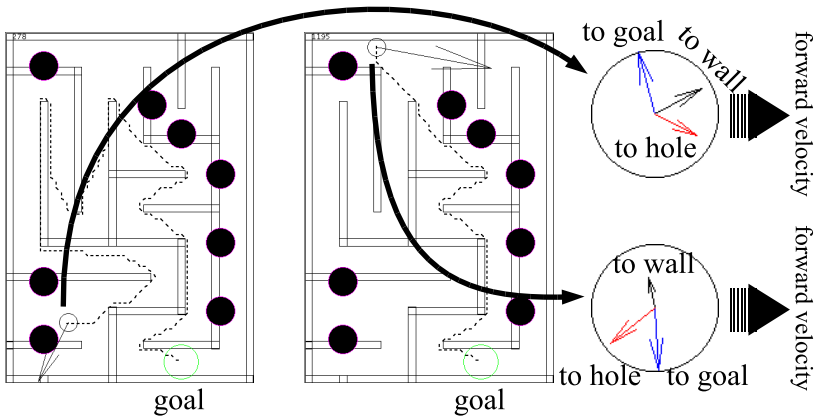



Figure 2.2: Local state description

as along a path towards the goal (dashed line in figure 2.2) are computed. These vectors are normalized to be at most length 1 by applying the logistic function to them. The path towards the goal is computed using A* on a discretized grid of the configuration space (**position only**). A* is very fast but does not take into account velocities and does not tell us what actions to use. Two examples of this local state description can be seen in figure 2.2. In the circle representing the relative view from the marble, the forward velocity is towards the right. In the first example, the marble is rolling towards a hole, so the hole vector is pointing ahead, slightly to the right of the marble, while the wall is further to the left. The direction to the goal is to the left and slightly aft. This results in a state vector of (.037; -.25, -.97; .72, -.38; .66, .34), where .037 is the scalar speed of the marble (not shown in figure), followed by the relative direction to the goal, relative direction to the closest wall and relative direction to the closest hole. The second example has the closest hole behind the marble, the closest wall to the left and the direction to the goal to the right of the direction of the marble, resulting in a state vector of (.064; .062, .998; -.087, -.47; -.70, .58). As all vectors are relative to the forward velocity, the velocity becomes a scalar speed only. Actions can likewise be *relativized* by projecting them onto the same forward velocity

vector. For comparison purposes, we show results for a different local state description in the discussion section (section 2.5).

### 2.3.2 Knowledge Transfer

The next step is to transfer knowledge from one maze to the next. For the intermediate policy, expressed using the local state description, we used a nearest neighbor classifier with a kd-tree as the underlying data structure for efficient querying. After a policy has been found for a maze, we iterate over the states and add the local state description with their local actions to the classifier. It is possible to use this intermediate policy directly on a new maze. For any state in the new maze, the local description is computed and the intermediate policy is queried for an action. However, in practice this does not allow the marble to complete the maze because it gets stuck. Furthermore, performance would be expected to be suboptimal as the local description alone does not necessarily determine the optimal action and previous policies might not have encountered states with local features similar to states that now appear on the new task.

Instead, an initial policy based on global coordinates is created using the classifier by iterating over states of the new maze and querying the classifier for the appropriate action based on the local features of that state. This policy is then refined.

### 2.3.3 Improving the Initial Policy

Originally, we wanted to use policy evaluation to create a value function from the initial policy which could then be further optimized using value iteration. In policy evaluation, the following update is performed for every state to update the value function estimate:

$$V_{\pi}^{t+1}(s) = r(s,a) + V_{\pi}^{t}(s(a)) \tag{2.2}$$

where $a = \pi(s)$, the action chosen in state $s$ by the policy $\pi$.

Compared to value iteration (equation 2.1), policy evaluation requires fewer computations per state because only one action is evaluated as opposed to every possible action. We hoped that the initial value function could be computed using little computation and that the subsequent value iterations would terminate after a few iterations.

However, some regions of the state space had a poor initial policy so that values were not properly propagated through these regions. In goal directed tasks such as the marble maze, the propagation of a high value frontier starting from the goal is essential to finding a good policy as the agent will use high valued states in its policy. If high values cannot propagate back through these bad regions, the values behind these bad regions will be uninformed and value iteration will not be sped up. Similarly, if a policy improvement step was used to update the policy in these states, the policy of states behind these bad regions would be updated based on an uninformed value function. An intuitive example of this might be the case of a tightrope walk: The optimal policy includes a tightrope walk. However, if in one of the states on the tightrope a wrong action is chosen, the states leading up to it might change their policy to take a long, suboptimal detour.

We overcame these two problems by creating a form of generalized policy iteration [47]. The objective in creating this dynamic programming algorithm was to efficiently use the initial policy to create a value function while selectively improving the policy where the value function estimates are valid. Our algorithm performs sweeps over the state space to update the value of states based on a fixed policy. In a small number of randomly selected states, the policy is updated by checking all actions (a full value iteration update using equation 2.1). As this is done in only a small number of states (on the order of a few percent), the additional computation required is small.

In order to avoid changing the policy for states using invalid values, the randomly selected states are filtered. Only those states are updated where

the updated action results in a transition to a state which has been updated with a value coming from the goal. This way we ensure that the change in policy is warranted and a result of information leading to the goal. This can easily be implemented by a flag for each state that is propagated back with the values. Note that as a result, we do not compute the value of states that cannot reach the goal.

## 2.4 Simulation Results

In order to gauge the efficiency of the algorithm, a series of simulated experiments was run. First, pools of 30 training mazes and 10 test mazes where created using a random maze generator (mazes available from [42]). We trained the intermediate classifier with an increasing number of training mazes to gauge the improvement achieved as the initial policy becomes more informed. The base case for the computation required to solve the test mazes was the computation required when using value iteration.

**Computational effort was measured by counting the number of times that a value backup was computed** before a policy was found that successfully solved the maze. The procedure for measuring the computational effort was to first perform 200 dynamic programming sweeps and then performing a trial in the maze based on the resulting policy. Following that, we alternated between computing 50 more sweeps and trying out the policy until a total of 1000 dynamic programming sweeps were performed.

When performing a trial, the policy was to pick the best action with respect to the expected reward based on the current estimate of the value function. Figure 2.3 shows the quality of the policy obtained in relation to the number of value backups. The right most curve represents value iteration from scratch and the other curves represent starting with an initial policy based on an increasing number of training mazes. The first data points show a reward of -2100 because policy execution was limited to 2100 time steps.

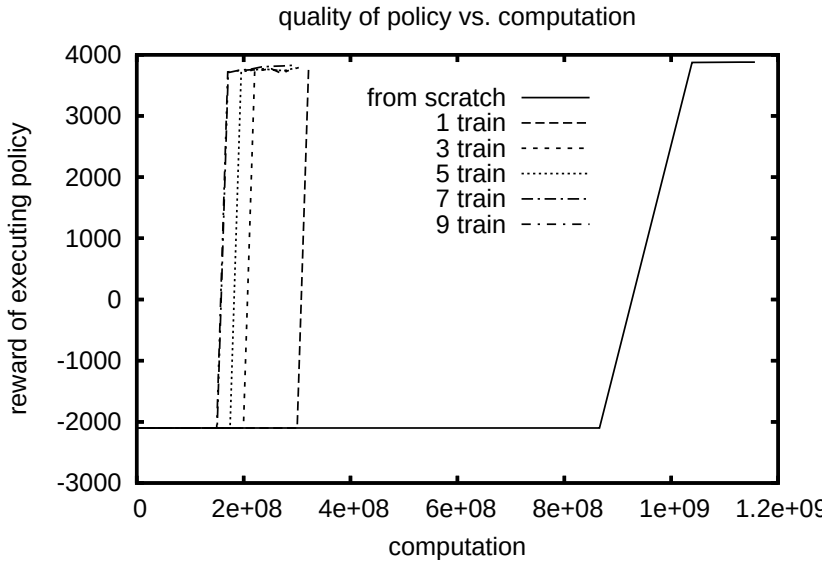



Figure 2.3: Results for one test maze

The trials were aborted if the goal was not yet reached.

Clearly, an initial policy based on the intermediate policy reduces the computation required to find a good policy. However, final convergence to the optimal policy is slow because only a small number of states are considered for policy updates. This results in a slightly lower solution quality in our experiments.

In order to ensure that the savings are not specific to this test maze, we computed the relative computation required to find a policy that successfully performs the maze for ten different test mazes and plotted the mean in figure 2.4 (solid). Additionally, in order to exclude the peculiarities of the training mazes as a factor in the results, we reran the experiments with other training mazes. The results can be seen in figure 2.4 (dashed). Clearly, the individual training mazes and their ordering do not influence the results very much.

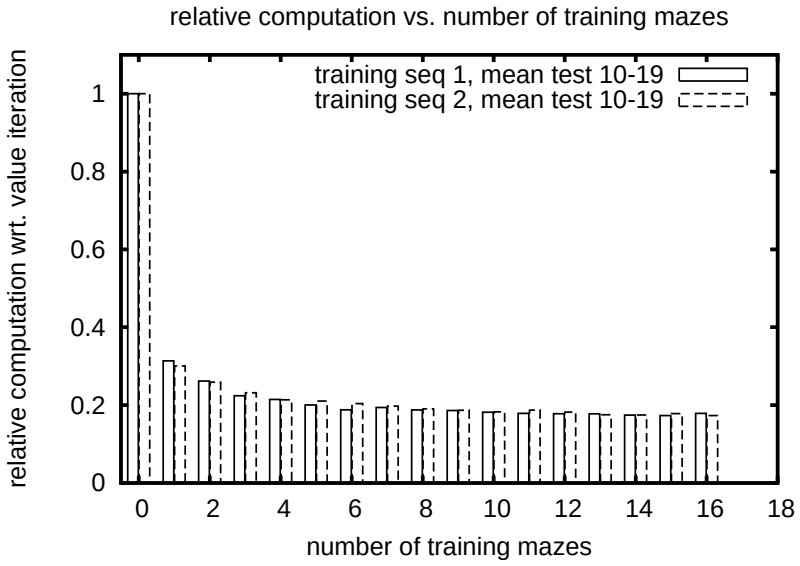



Figure 2.4: Relative computation, averaged over 10 test mazes, using two different sequences of training mazes

## 2.5 Discussion

**State Description:** The local features that we are proposing as a solution to this problem are intuitively defined as features of the state space that are in the immediate vicinity of the agent. However, often the agent is removed from the actual environment and might even be controlling multiple entities or there may be long-range interactions in the problem. A more accurate characterization of the features we are seeking are that they influence the results of the actions in a consistent manner across multiple tasks and allow, to a varying degree, predictions about the relative value of actions. These new features have to include enough information to predict the outcome of the same action across different environments and should ideally not include unnecessary information that does not affect the outcome of actions. They are similar in spirit to predictive state representation [31]. These conditions will preclude features such as position on a map, as this alone will not predict

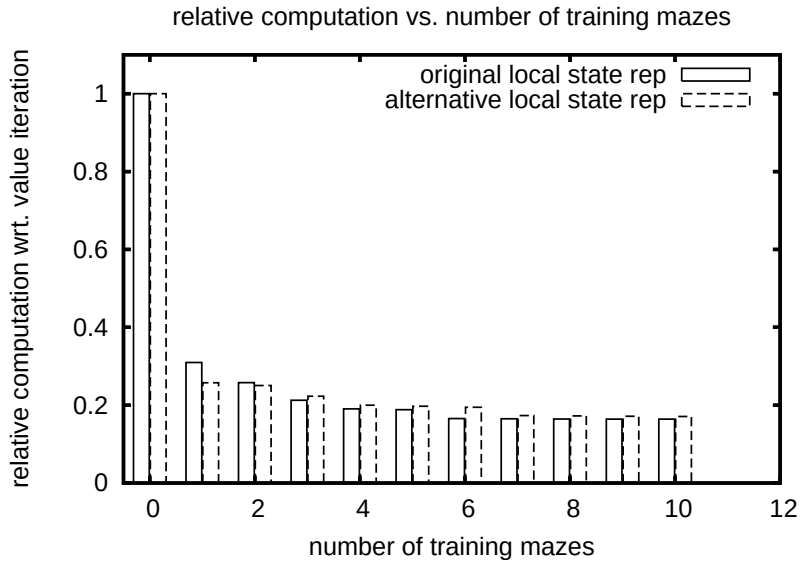



Figure 2.5: Relative computation required for one test maze for two different local state descriptions

the outcome of actions – obstacles and goals are much more important.

In order to gauge the effect of different local state descriptions, we created an alternative state description. In this alternative description, the world is described again as seen from the marble, but aligned with the direction to the goal instead of the direction of the movement. Furthermore, the view is split up into 4 quadrants: covering the 90 degrees towards the path to the goal, 90 degrees to the left, to the right and to the rear. For each quadrant, the distance to the closest hole and closest wall are computed. Holes that are behind walls are not considered. The velocity of the marble is projected onto the path towards the goal. The resulting state description is less precise with respect to direction to the walls or holes than the original local description but takes into account up to four holes and walls, one for each quadrant. As can be seen in figure 2.5, the results are similar for both state descriptions. The new state description performs slightly better with fewer training mazes but loses its advantage with more training mazes.

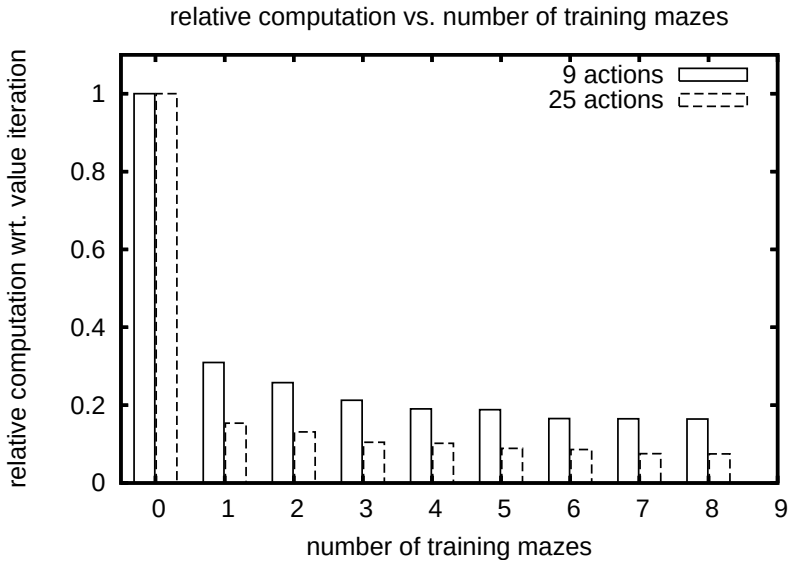



Figure 2.6: Relative computation required for one test maze and different number of actions

**Computational Saving:** There are several factors that influence the computational saving one achieves by using an informed initial policy. The computational reduction results from the fact that our generalized policy evaluation only computes the value of a single action at each state, whereas value iteration tries out all actions for every state. As a result, if the action space is discretized at high resolution, resulting in many possible actions at each state, the computational savings will be high. If on the other hand there are only two possible actions at each state, the computational saving will be much less. The computation can be reduced at most by a factor equal to the number of actions. However, since in a small number of states in the generalized policy evaluation we also try all possible actions, the actual savings at every sweep will be less. In order to show the effects of changing the number of actions, we reran the experiments for one maze with actions discretized into 25 different actions instead of 9. As seen in figure 2.6, the relative computational saving becomes significantly larger, as was expected.

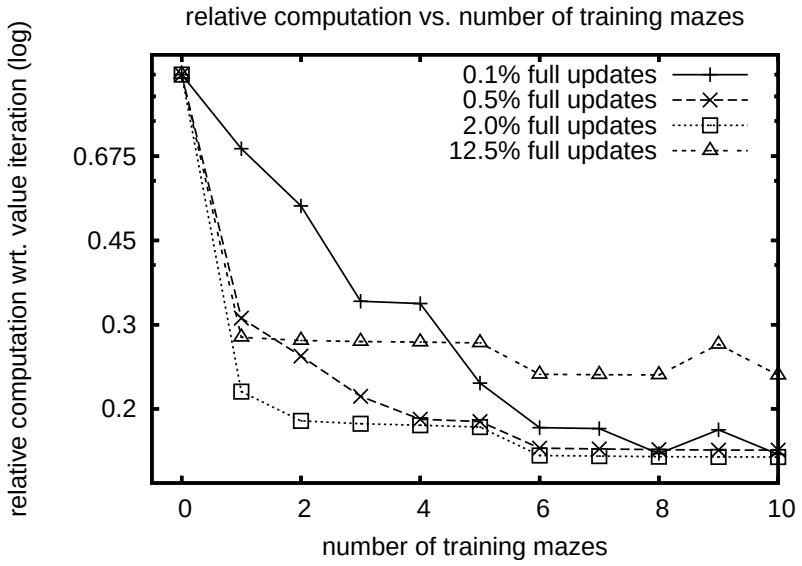



Figure 2.7: Relative computation required for one test maze and different percentages of full updates

We also ran experiments to determine the effect of performing policy updates on a varying number of states. If many states are updated at every sweep, fewer sweeps might be necessary, however each sweep will be more expensive. Conversely, updating fewer states can result in more sweeps, as it takes longer to propagate values across bad regions which are now less likely to be updated. The results are presented in figure 2.7. When reducing the percentage of states updated to 0.1%, the computational saving is reduced as it now takes many more sweeps to find a policy that solves the maze, unless the initial policy is very good (based on several mazes). The savings become more pronounced as more states are updated fully and are the greatest when 2.0% of the states are updated, performing better than our test condition of 0.5%. However, increasing the number of states updated further results in reduced savings as now the computational effort at every sweep becomes higher. Comparing the extreme cases shows that when updating few states, the initial policy has to be very good (many training

mazes added), as correcting mistakes in the initial policy takes longer. On the other hand, if many states are updated, the quality of the initial policy is less important – many states are updated using the full update anyways.

**Intermediate Policy Representation:** Another issue that arose during testing of the knowledge transfer was the representation of the intermediate policy representation. We chose a nearest neighbor approach, as this allows broad generalization early on, without limiting the resolution of the intermediate policy once many training mazes were added to the intermediate policy. However, after adding many mazes, the data structure grew very large (around 350,000 data points per maze, around 5 million for 15 mazes). While the kd-trees performed well, the memory requirements became a problem. Looking at the performance graph, adding more than 5 mazes does not seem to make sense with the current state description. However, if a richer state description was chosen, it might be desirable to add more mazes and then pruning of the kd-tree becomes essential.

The nearest neighbor algorithm itself is modifiable through the use of different distance functions. By running the distances to the closest hole and wall through a logistic function, we have changed the relative weight of different distances already. However, instead one could imagine rescaling distance linearly to range from 0 and 1, where 1 is the longest distance to either hole or wall observed.

**Dynamic Programming on Local State Space:** As we are using the local state space to express an intermediate policy, it might be interesting to perform dynamic programming in this state space directly. Due to the possible aliasing of different states to the same local state, the problem becomes a partially observable Markov decision process (POMDP). This is aggravated if one keeps the value function across multiple tasks, as now even more states are potentially aliased to the same local state. A policy is less sensitive to this aliasing, as the actions might still be similar while the values could be vastly different. An example can be seen in figure 2.8. Both positions with

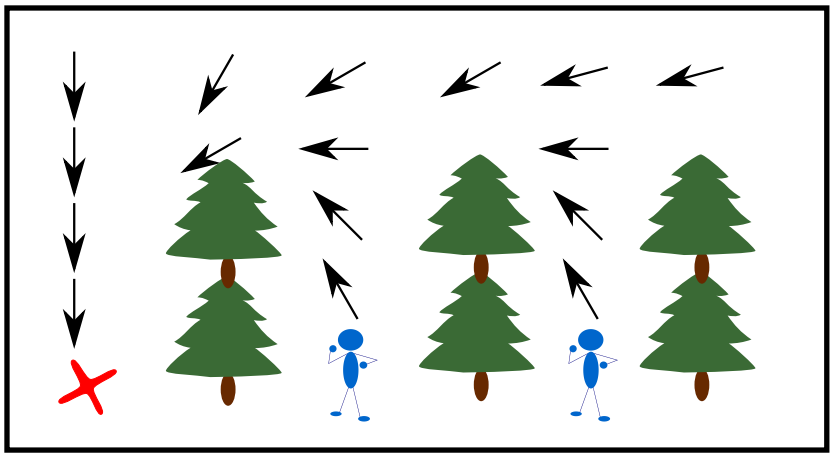


Figure 2.8: Aliasing problem: same local features, same policy but different values

the agent have the same features and the same policy, but the value would be different under most common reward functions which favor short paths to the goal (either with discounting or small constant negative rewards at each time step). We avoid this problem by expanding the intermediate feature-based policy back into a global state-based policy and performing policy iteration in this state space (see also figure 2.1). For similar reasons, it is tricky to transfer the value function directly: the same local features might have different values depending on their distance to the goal.

## 2.6 Conclusion

We presented a method for transferring knowledge across multiple tasks in the same domain. Using knowledge of previous solutions, the agent learns to solve new tasks with less computation than would be required without prior knowledge. Key to this knowledge transfer was the creation of a local state description that allows for the representation of knowledge that is independent of the individual task.

# Chapter 3

# Policies Based on Trajectory Libraries[1]

## 3.1 Introduction

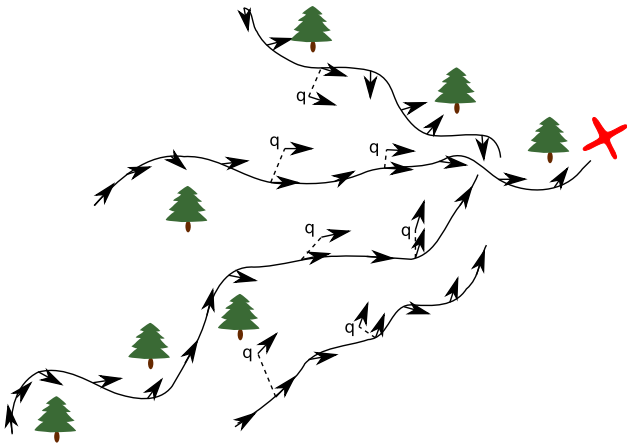



Figure 3.1: Illustration of a trajectory library. When queried at any point (e.g. 'q'), the action (indicated by arrows) of the closest state on any trajectory is returned

Finding a policy, a control law mapping states to actions, is essential in

[1]Partially published in [44]

solving many problems with inaccurate or stochastic models. By knowing how to act for all or many states, an agent can cope with unexpected state transitions. Unfortunately, methods for finding policies based on dynamic programming require the computation of a value function over the state space. This is computationally very expensive and requires large amounts of fast memory. Furthermore, finding a suitable representation for the value function in continuous or very large discrete domains is difficult. Discontinuities in the value function or its derivative are hard to represent and can result in unsatisfactory performance of dynamic programming methods. Finally, storing and computing this value function is impractical for problems with more than a few dimensions.

When applied to robotics problems, dynamic programming methods also become inconvenient as they cannot provide a "rough" initial policy quickly. In goal directed problems, a usable policy can only be obtained when the value function has almost converged. The reward for reaching the goal has to propagate back to the starting state before the policy exhibits goal directed behavior from this state. This may require many sweeps. If only an approximate model of the environment is known, it would be desirable to compute a rough initial policy and then spend more computation after the model has been updated based on experience gathered while following the initial policy.

In some sense, using dynamic programming is both too optimistic and too pessimistic at the same time: it is too optimistic because it assumes the model is accurate and spends a lot of computation on it. At the same time, it is too pessimistic, as it assumes that one needs to know the correct behavior from any possible state, even if it is highly unlikely that the agent enters certain parts of the state space.

To avoid the computational cost of global and provably stable control law design methods such as dynamic programming, often a single desired trajectory is used, with either a fixed, time varying linear or state dependent

control law. The desired trajectory can be generated manually, generated by a path planner [29], or generated by trajectory optimization [49]. For systems with nonlinear dynamics, this approach may fail if the actual state diverges sufficiently from the planned trajectory. Another approach to making trajectory planners more robust is to use them in real time at fixed time intervals to compute a new plan from the current state. For complex problems, these plans may have to be truncated (N step lookahead) to obey real time constraints. It may be difficult to take into account longer term outcomes in this case. In general, single trajectory planning methods produce plans that are at best locally optimal.

To summarize, we would like an approach to finding a control law that, on the one hand, is more anytime [6] than dynamic programming - we would like to find rough policies quickly and expend more computation time only as needed. On the other hand, the approach should be more robust than single trajectory plans.

In order to address these issues, we propose a representation for policies and a method for creating them. This representation is based on libraries of trajectories. Figure 3.1 shows a simple navigational domain example. The cross marks the goal and the trees represent obstacles. The black lines are the trajectories which make up the library and the attached arrows are the actions. These trajectories can be created very quickly using forward planners such as A* or Rapidly exploring Random Trees (RRT) [29]. The trajectories may be non-optimal or locally optimal depending on the planner used, in contrast to the global optimality of dynamic programming.

Once we have a number of trajectories and we want to use the agent in the environment, we turn the trajectories into a state-space based policy by performing a nearest-neighbor search in the state-space for the closest trajectory fragment and executing the associated action. For example in figure 3.1, when queried in states marked with 'q', the action of the closest state on any trajectory (shown with the dashed lines) is returned.

## 3.2 Related Work

Using libraries of trajectories for generating new action sequences has been discussed in different contexts before. Especially in the context of generating animations, motion capture libraries are used to synthesize new animations that do not exist in that form in the library [27, 30]. However, since these systems are mainly concerned with generating animations, they are not concerned with the control of a real world robot and only string together different sequences of configurations, often ignoring physics, disturbances or inaccuracies.

Another related technique in path planning is the creation of Probabilistic Roadmaps (PRMs) [24]. The key idea of PRMs is to speed up multiple planning queries by precomputing a roadmap of plans between stochastically chosen points. Queries are answer by planning to the nearest node in the network, using plans from the network to get to a the node closest to the goal and then planning from there to the node. The method presented here and PRMs have some subtle but important differences. Most importantly, PRMs are a path planning algorithm. Our algorithm, on the other hand, is concerned with turning a library of paths into a control law. Internally, PRMs precompute bidirectional plans that can go from and to a large number of randomly selected points. However, the plans in our library all go to the same goal. As such, the nature of the PRM's "roadmap" is very different than the kind of library we require. Of course, PRMs can be used as a path planning algorithm to supply the paths in our library. Due to the optimization for multiple queries, PRMs might be well suited for this and are complementary to our algorithm.

Libraries of low level controllers have been used to simplify planning for helicopters in Frazzoli's Ph.D. thesis [17]. The library in this case is not the solution to the goal achievement task, but rather a library of controllers that simplifies the path planning problem. The controllers themselves do not use libraries. Older works exists on using a library of pregenerated control inputs

together with the resulting path segments to find a sequence of control inputs that follow a desired path [3, 19]. These are examples of motion primitives where a library of behaviors is used as possible actions for planning instead of low level actions. These motion primitives encode possible behaviors of the robot and are not used as reactive controllers like the work presented here. Typically, motion primitives assume that their applications will result in the same trajectory *relative to the starting position* no matter which position they are applied from. Bouncing into a wall would be grounds for disqualifying the use of a particular motion primitive. An exception to this is the work by Howard and Kelly [21], where a library of motion primitives (generated on flat ground) is used to seed an optimizer that takes into account interactions with rough terrain for particular instantiations of a primitive.

Prior versions of a trajectory library approach, using a modified version of Differential Dynamic Programming (DDP) [23] to produce globally optimal trajectories can be found in [1,2]. This approach reduced the cost of dynamic programming, but was still quite expensive and had relatively dense coverage. The approach of this chapter uses more robust and cheaper trajectory planners and strives for sparser coverage. Good (but not globally optimal) policies can be produced quickly.

Other related works in planning do not attempt to use libraries of trajectories but exploit geometrical properties of the state space and carefully analyze the model of the environment to create vector fields. These feedback motion plans [11, 12, 39] can be hard to compute and it is unclear how to make use of discontinuities in the model, such as bouncing into a wall in the case of the marble maze. Sampling based methods such as [52] have been introduced to simplify the construction of feedback motion plans.

## 3.3 Case Study: Marble Maze

The first domain used for gauging the effectiveness of the new policy representation and generation is the marble maze domain (figure 1.2). The model used for creating trajectories is the simulator described in section 1.2.1. The hardware described in the same section was used for the experiments on the actual maze. The actions in the library are tilts that are directly sent to the maze, which makes this a library of elementary actions (see figure 1.1(a)).

### 3.3.1 Trajectory Libraries

The key idea for creating a global control policy is to use a library of trajectories, which can be created quickly and that together can be used as a robust policy. The trajectories that make up the library are created by established planners such as A* or RRT. Since our algorithm only requires the finished trajectories, the planner used for creating the trajectories is interchangeable. For the experiment presented here, we used an inflated-heuristic [36] A* planner. By overestimating the heuristic cost to reach the goal, we empirically found planning to proceed much faster because it favors expanding nodes that are closer to the goal, even if they were reached sub-optimally. This might not be the case generally [36]. We used a constant cost per time step in order to find the quickest path to goal. In order to avoid risky behavior and compensate for inaccuracies and stochasticity, we added a cost inversely proportional to the squared distance to the closest hole on each step. As basis for a heuristic function, we used distance to the goal. This distance is computed by a configuration space (position only) A* planner working on a discretized grid with 2mm resolution. The final heuristic is computed by dividing the distance to the goal by an estimate of the distance that the marble can travel towards the goal in one time step. As a result, we get a heuristic estimate of the number of time steps required to reach the goal.

The basic A* algorithm is adjusted to continuous domains as described

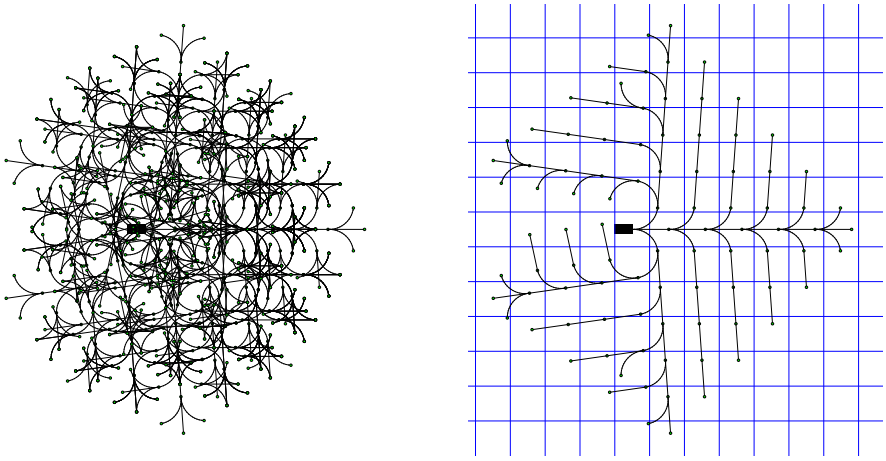


Figure 3.2: An example of pruning [28]

in [28]. The key idea is to prune search paths by discretizing the state space and truncating paths that fall in the same discrete "bin" as one of the states of a previously expanded path (see figure 3.2 for an illustration in a simple car domain). This limits the density of search nodes but does not cause a discretization of the actual trajectories. Actions were limited to physically obtainable forces of up to $\pm 0.007$N in both dimension and discretized to a resolution of 0.0035N. This resulted in 25 discrete action choices. For the purpose of pruning the search nodes, the state space was discretized to 3mm spatial resolution and 12.5mm/s in velocity resolution.

The A* algorithm was slightly altered to speed it up. During search, each node in the queue has an associated action multiplier. When expanding the node, each action is executed as many times as dictated by the action multiplier. The new search nodes have an action multiplier that is incremented by one. As a result, the search covers more space at each expansion at the cost of not finding more optimal plans that require more frequent action changes. In order to prevent missed solutions, this multiplier is halved every time none of the successor nodes found a path to the goal, and the node is re-expanded using the new multiplier. This resulted in a speed up in finding trajectories

(over 10x faster). The quality of the policies did not change significantly when this modification was applied.

As the policy is synthesized from a set of trajectories, the algorithms for planning the trajectories have a profound impact on the policy quality. If the planned trajectories are poor, the performance of the policy will be poor as well. While in theory A* can give optimal trajectories, using it with an admissible heuristic is often too slow. Furthermore, some performance degradation derives from the discretization of the action choices. RRT often gives "good" trajectories, but it is unknown what kind of quality guarantees can be made for the trajectories created by it. However, the trajectories created by either planning method can be locally optimized by trajectory optimizers such as DDP [23] or DIRCOL [49].

Currently, no smoothness constraints are imposed on the actions of the planners. It is perfectly possible to command a full tilt of the board in one direction and then a full tilt in the opposite direction in the next time step (1/30th second later). Only imposing constraints on the plans would not solve the problem as the policy look up might switch between different trajectories. However, by including the current tilt angles as part of the state description and have the actions be changes in tilt angle, smoother trajectories could be enforced at the expense of adding more dimensions to the state space.

In order to use the trajectory library as a policy, we store a mapping from each state on any trajectory to the planned action of that state. During execution, we perform a nearest-neighbor look up into this mapping using the current state to determine the action to perform. We used a weighted Euclidean distance which tries to normalize the influence of distance (measured in meters) and velocity (measured in meters per second). As typical velocities are around 0.1m/s and a reasonable neighborhood for positions is about 0.01m, we multiply position by 100 and velocities by 10, resulting in distances around 1 for reasonably close data points.

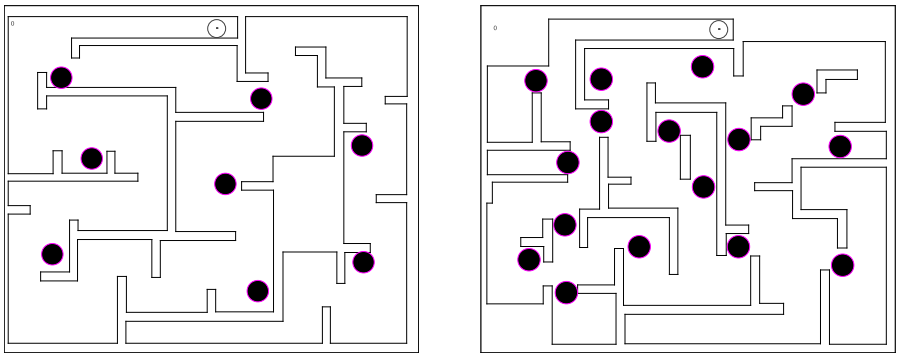
We speed up the nearest-neighbor look ups by storing the state-action mappings in a kd-tree [18]. Performance of the kd-tree was very fast. After 100 runs, the library contained about 1200 state-action pairs and queries took about 0.01ms on modest hardware (Pentium IV, 2.4GHz). Query time is expected to grow logarithmically with the size of the library.

Part of the robustness of the policies derives from the coverage of trajectories in the library. In the experiments on the marble maze, we first created an initial trajectory from the starting position of the marble. We use three methods for adding additional trajectories to the library. First, a number of trajectories are added from random states in the vicinity of the first path. This way, the robot starts out with a more robust policy. Furthermore, during execution it is possible that the marble ceases making progress through the maze, for example if it is pushed into a corner. In this case, an additional path is added from that position. Finally, to improve robustness with experience, at the end of every failed trial a new trajectory is added from the last state before failure. If no plan can be found from that state (for example because failure was inevitable), we backtrack and start plans from increasingly earlier states until a plan can be found. Computation is thus focused on the parts of the state space that were visited but had poor coverage or poor performance. In later experiments, the model is updated during execution of the policy. In this case, the new trajectories use the updated model. The optimal strategy of when to add trajectories, how many to add, and from which starting points is a topic of future research.

Finally, we developed a method for improving an existing library based on the execution of the policy. For this purpose, we added an additional discount parameter to each trajectory segment. If at the end of a trial the agent has failed to achieve its objective, the segments that were selected in the policy leading up to the failure are discounted. This increases the distance of these segments in the nearest-neighbor look up for the policy and as a result these segments have a smaller influence on the policy. This

is similar to the mechanism used in learning from practice in Bentivegna's marble maze work [4]. We also used this mechanism to discount trajectory segments that led up to a situation where the marble is not making progress through the maze.

### 3.3.2 Experiments

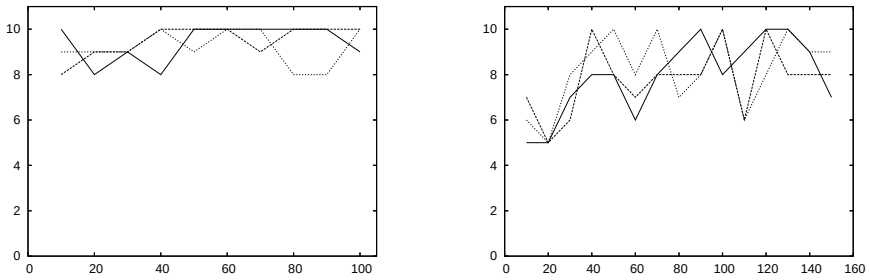


(a) Maze A (b) Maze B

Figure 3.3: The two mazes used for testing

We performed initial simulation experiments on two different marble maze layouts (figure 3.3). The first layout (maze A) is a simple layout, originally designed for beginners. The second layout (maze B) is a harder maze for more skilled players. These layouts were digitized by hand and used with the simulator.

For maze A, we ran 100 consecutive runs to find the performance and judge the learning rate of the algorithm. During these runs, new trajectories were added as described above. After 100 runs, we restarted with an empty library. The results of three sequences of 100 runs each are plotted in figure 3.4(a). Almost immediately, the policy successfully controls the marble through the maze about 9-10 times out of 10. The evolution of the trajectory library for one of the sequences of 100 runs is shown in figure 3.5. Initially, many trajectories are added. Once the marble is guided through the maze



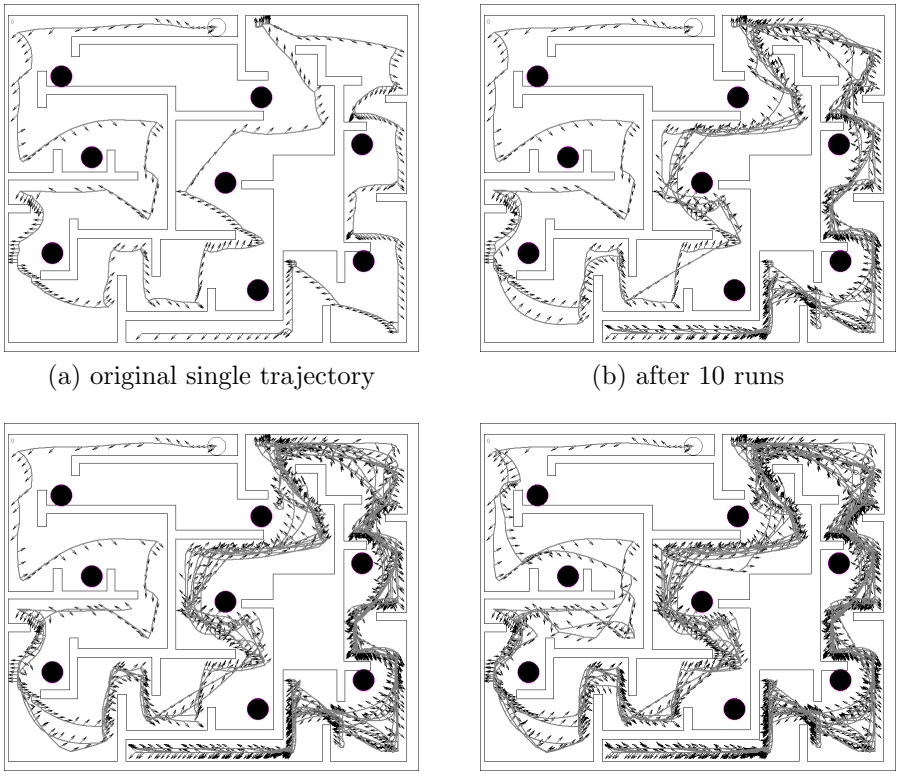

(a) Maze A (b) Maze B

Figure 3.4: Learning curves for simulated trials. The x axis is the number of starts and the y axis is the number of successes in 10 starts (optimal performance is a flat curve at 10). We restarted with a new (empty) library three times.

successfully most of the times, only few more trajectories are added. Similarly, we performed three sequences of 150 runs each on maze B. The results are plotted in figure 3.4(b). Since maze B is more difficult, performance is initially weak and it takes a few failed runs to learn a good policy. After a sufficient number of trajectories was added, the policy controls the marble through the maze about 8 out of 10 times.

We also used our approach to drive a real world marble maze robot. This problem is much harder than the simulation, as there might be quite large modeling errors and significant latencies. We used the simulator as the model for the A* planner. In the first experiment, we did not attempt to correct for modeling errors and only the simulator was used for creating the trajectories. The performance of the policy steadily increased until it successfully navigated the marble to the goal in half the runs (figure 3.6).

In figure 3.7 we show the trajectories traveled in simulation and on the real world maze. The position of the marble is plotted with a small round circle at every frame. The arrows, connected to the circle via a black line, indicate the action that was taken at that state and are located in the position for which they were originally planned for. Neither the velocity of the marble nor the velocity for which the action was originally planned for is



(a) original single trajectory (b) after 10 runs

(c) after 30 runs (d) after 100 runs

Figure 3.5: Evolution of library of trajectories. (The trajectories (thick lines) are shown together with their actions (thin arrows))

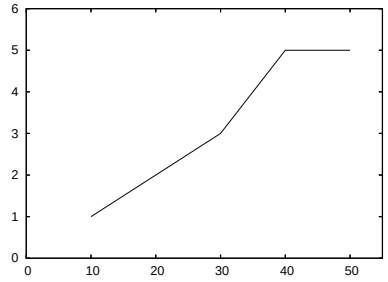
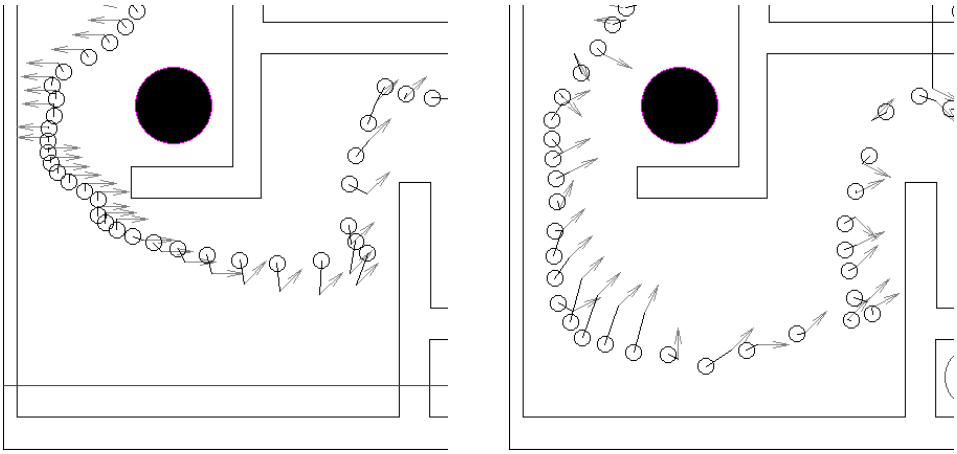



Figure 3.6: Learning curves for trials on hardware for maze A. The x axis is the number of starts and the y axis is the number of successes in 10 starts.



(a) simulation with simulated noise (b) real world

Figure 3.7: Actual trajectories traveled. The circles trace the position of the marble. The arrows, connected to the marble positions by a small line, are the actions of the closest trajectory segment that was used as the action in the connected state

plotted. Due to artificial noise in the simulator, the marble does not track the original trajectories perfectly, however the distance between the marble and the closest action is usually quite small. The trajectory library that was used to control the marble contained 5 trajectories. On the real world maze, the marble deviates quite a bit more from the intended path and a trajectory library with 31 trajectories was necessary to complete the maze.

Close inspection of the actual trajectories of the marble on the board revealed large discrepancies between the real world and the simulator. As a result, the planned trajectories are inaccurate and the resulting policies do not perform very well (only half of the runs finish successfully). In order to improve the planned trajectories, we tried a simple model update technique to improve our model. The model was updated by storing observed state-action-state change triplets. During planning, a nearest-neighbor look up in state-action space is performed and if a nearby tuplet is found, the stored state change is applied instead of computing the state evolution based on the simulator. In this nearest-neighbor look up, the weights for position and

velocity were the same as for the policy look up. Since the model strongly depends on the action, which is quite small (on the order of 0.007N), the weight in the distance metric for the actions is $1 \times 10^6$. The cutoff for switching over to the simulated model was a distance of 1.

Initial experiments using this method for updating the model did not result in significant improvements. Another factor that impacted the performance of the robot was the continued slipping of the tilting mechanism such that over time, the same position of the control knob corresponded to different tilts of the board. While the robot was calibrated at the beginning of every trial, sometimes significant slippage occurred during a trial, resulting in inaccurate control and even improperly learned models.

## 3.4 Case Study: Little Dog

Another domain to which we applied the algorithm to is the Little Dog domain (figure 1.4) described in section 1.2.2. Due to the complexity of the domain, we use a hierarchical approach to control the robot. At the high level, actions designate a foot and a new target location for that foot. A low level controller then controls the joints to move the body, lift the foot and place it at the desired location. The trajectory library operates at the high level and is responsible for selecting foot step actions, which makes this a library of abstract actions (see figure 1.1(b)) . Before explaining how the library works, we explain how the hierarchical controller works using a normal planner.

### 3.4.1 Planning and Feedback Control

In order to cross the terrain and reach a desired goal location, we use a footstep planner [10] that finds a sequence of steps going to some goal location. The planner operates on an abstract state description which only takes into account the global position of the feet during stance. Actions in this plan-

ner merely designate a new global position for a particular foot. We use a heuristic method to compute continuous body and foot trajectories in global coordinates that move the dog along the footsteps output by the planner. Details of this step execution are described in appendix A.

We implemented several feedback controllers to improve the performance of the step execution. One type of controller is a body controller which uses integrators on each foot to keep the body on the desired trajectory, even if the stance feet slip. In theory, as long as this controller performs well and the body only shows small servoing error, the swing leg will reach its intended location (see appendix B and C for details of two related implementations of the body controller). In practice, it was difficult to maintain stability of the body controller while keeping the body on the intended trajectory. As a result, we also added a global flight foot controller, which was designed to keep the flight foot on it's original trajectory in global coordinates, even if the body was not on its intended trajectory (see appendix D for details). Finally, we also added on-line trajectory modifications which modify the desired trajectories of the body and flight foot in order to maintain reachability of all feet as well as to avoid undesired contact with the ground (see appendix E).

Finally, we also implemented a safety monitor. After the execution of a step and before executing the next in a plan, we look at features of the stance combined with the next step to see if it is safe to execute the desired step. In particular, we look at the radius of the in-circle of the three stance feet for the next step. Given a standard sequence of flight feet, we can also hypothesize a probable next flight foot and look at the in-circle of the future step if the current step was executed as planned. If either of these in-circles has a radius smaller than 4cm, we deem the step to be unsafe. Furthermore, we also look at the expected distance of diagonal feet at the end of the chosen step. If this distance is larger than 36cm, we also deem the step unsafe, as the robot will usually not be stable with feet further apart.

In principle, a single plan suffices for the robot to reach the goal location, especially since the low-level step execution always tries to hit the next global target of the swing foot in the plan, regardless of how well the previous step was executed. In practice, if there is significant slip, the next foot hold might no longer be reachable safely. This is detected by the safety monitor and replanning is initiated.

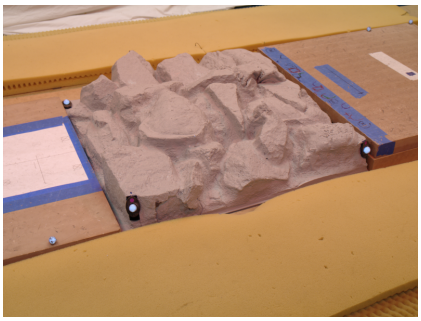
### 3.4.2 Trajectory Libraries

By using a trajectory library, the robot can start with multiple plans and reduce the likelihood of needing to replan. Even with slips, it might be close enough to execute a step of a previous plan safely. Only if the robot detects that the step selected from all the steps in the library is unreachable does it have to stop the execution and replan. The new plan is then merely added to the library and, if necessary in the future, actions from previous plans can still be reused.

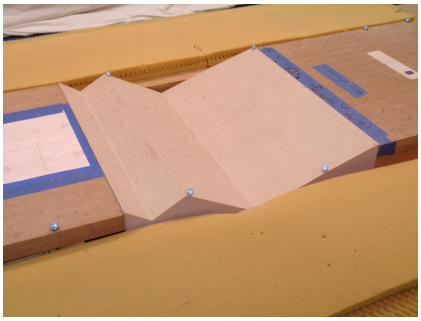
Similar to the marble maze, the robot uses a global state-based lookup into the trajectory library to select its next step. In the Little Dog case, this is the global position of the four feet. When started or after completing a step, the robot computes the position of the four feet in global coordinates. Then, for every step in the library, the sum of the Euclidean distances between the current position of the feet and their respective position at the beginning of the step is computed. The step with the minimum sum of the Euclidean distances is used as the next step to take. For large libraries, a kd-tree can be used to speed up this query. Since a plan in the Little Dog case consists of few, large steps, this was not necessary.

We also implemented a mechanism for improving the library based on experience. For this, we remember the last few steps taken. If one of the last steps taken from the library is selected again, we slow down the execution of this step. This is results in improving the execution of particularly difficult steps so that they are more likely to succeed. Additionally, we keep track

(a) modular rocks



(b) truncated rocks



(c) slopes

Figure 3.8: Terrains for gauging the advantage of the trajectory library

of how many times a particular step was selected for a particular trial. If a step is ever taken more than some number of times, we add a penalty to the step which is added to its Euclidean distance when selecting future step. The penalty is chosen such that if the library is queried from the same state again, it will chose the second best step instead of the previously selected step. This also prevents the robot from attempting the same step indefinitely.

### 3.4.3 Experiments

We performed several experiments to gauge the effectiveness of the library approach. In these experiments, we compared using a single-plan sequential execution with replanning to using the trajectory library approach. In both cases, we use the same safety monitor to check the safety of the next step

and replan as necessary. Experiments were run on three different terrains (see figure 3.8). For each terrain board, we started the robot in a number of different starting locations. As planning times of the footstep planner can vary wildly, even with nearly identical starting positions, we only look at the number of planning invocations and not at the time spent planning (total planning times were between 1/6th to 1/4th of the execution times for most boards. Due to peculiarities of the footstep planner, the total planning times for the slope board were between 1.2x to 2x the execution times). As the library approach keeps previous plans, we expect it to require fewer invocations of the planner.

Additionally, we also compare the time it takes for the robot to cross the board. As the library can slow down steps that are executed poorly and even avoid using particular steps, we expect the execution of the library to be slightly better.

As can be seen from the results in table 3.1, using a trajectory library reduces the number of planning invocations necessary to cross a terrain by roughly a factor of two. Furthermore, execution speed alone, without considering planning time, is slightly improved. When examining the logging output, we found that the library based executions sometimes skips steps. This is possible because the planner is using a fixed sequence of feet in deciding which foot to move next. However, sometimes it is not possible to move a foot forward so it is stepped in place. In such a case, the trajectory library can skip a step. Also, it is possible that the robot slips forwards and a future step becomes executable early.

Finally, the start states used were chosen to allow for similar paths and hence synergies between plans. For the modular rocks board, we also picked a number of start states deliberately chosen to be far apart. Although the advantage of the library is less pronounced in this case when compared to the original step sequence (see table 3.1d vs. table 3.1a), there is a still a noticeably positive effect. When looking at the resulting library in 3.9, this

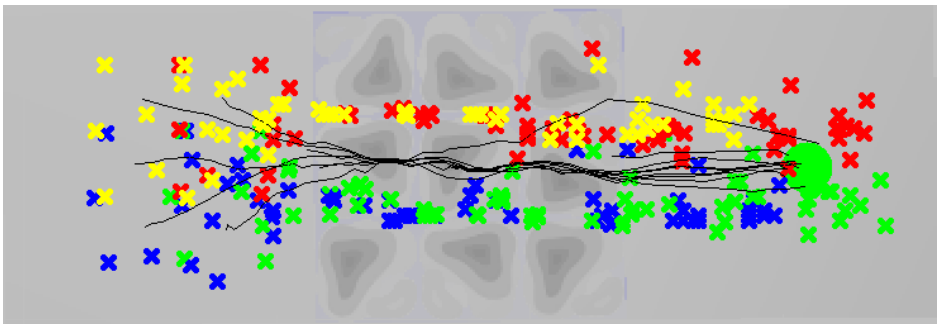


Figure 3.9: The library after executing 5 starts (second experiment) for the modular rocks terrain

| | sequential execution | | library | |
|---|---|---|---|---|
| start | plan. invocations | execution time | plan. invocations | execution time |
| 1 | 5 | 30.2s | 2 | 25.8s |
| 2 | 3 | 27.4s | 1 | 25.8s |
| 3 | 3 | 25.3s | 2 | 25.2s |
| 4 | 1 | 25.0s | 1 | 23.5s |
| 5 | 4 | 23.0s | 1 | 22.3s |
| sum | 16 | 130.9s | 7 | 122.6s |

(a) modular rocks results

| | sequential execution | | library | |
|---|---|---|---|---|
| start | plan. invocations | execution time | plan. invocations | execution time |
| 1 | 1 | 28.2s | 1 | 28.9s |
| 2 | 6 | 30.7s | 5 | 27.4s |
| 3 | 6 | 31.2s | 1 | 27.1s |
| sum | 13 | 90.1s | 7 | 83.4s |

(b) truncated rocks

| | sequential execution | | library | |
|---|---|---|---|---|
| start | plan. invocations | execution time | plan. invocations | execution time |
| 1 | 6 | 24.8s | 6 | 25.8s |
| 2 | 6 | 24.4s | 0 | 25.5s |
| 3 | 5 | 24.9s | 4 | 24.6s |
| 4 | 3 | 24.3s | 0 | 22.6s |
| sum | 20 | 98.4s | 10 | 98.5s |

(c) slopes

| | sequential execution | | library | |
|---|---|---|---|---|
| start | plan. invocations | execution time | plan. invocations | execution time |
| 1 | 4 | 30.8s | 1 | 25.2s |
| 2 | 6 | 31.2s | 1 | 24.2s |
| 3 | 4 | 29.7s | 2 | 26.4s |
| 4 | 2 | 25.4s | 2 | 23.4s |
| 5 | 4 | 28.3s | 5 | 20.2s |
| sum | 20 | 145.4s | 11 | 119.4s |

(d) modular rocks results, different start states

Table 3.1: Results comparing sequential execution and library execution on different terrains

is a result of the paths going over similar parts of the terrain, even when started from different start positions.

During these experiments, sometimes Little Dog catastrophically failed in executing a step and fell over. Neither replanning or a trajectory library can recover from this. In our experiments, this happened a small number of times and those runs were excluded. However, it is conceivable that reflexes can be implemented on Little Dog that interrupt the execution of a step when failure is imminent and attempt to return the robot to a stable pose. Most likely, this would require replanning, or in the case of the library, picking a step that wasn't the original successor to the previous step. Using reflexes to recover from step execution failure will hence benefit from using a library approach.

## 3.5 Discussion

The main advantage of a trajectory library is a reduction in needed computational resources. We can create an initial policy with as little as one trajectory. In the case of the marble maze, we avoid computing global policies and instead use path planning algorithms for creating robust behavior without the possibility of replanning. In the case of Little Dog, replanning is possible but undesirable and the use of the library reduces the need for replanning. The larger the library is, the less likely replanning is necessary. Due to the setup of the Little Dog domain, the robot is run over a terrain multiple times and often similar paths are found by the path planner, leading to an effective use of previous paths. Similarly, in the marble maze the library becomes more robust the more paths have been added.

By scheduling the creation of new trajectories based on the performance of the robot or in response to updates of the model, policies based on trajectory libraries are easy to update. In particular, since the library can be continually improved by adding more trajectories, the libraries can be used in an anytime

algorithm [6]: while there is spare time, one adds new trajectories by invoking a trajectory planner from new start states. Any time a policy is needed, the library of already completely planned trajectories can be used. It is possible to plan multiple paths simultaneously for different contingencies or while the robot is executing the current paths. These paths can be added to the library and will be used as necessary. Planning simultaneously while executing the current plan leads to timing issues in the case of a sequential execution, as the robot will continue executing using the previous plan, while a new plan is created. When the new plan is finished, the robot is likely no longer near the start of the new plan and the problem of splicing the new plan into the current plan can be tricky. Especially in the Little Dog case, in our experience, the foot step planner can find very different paths even with similar starting conditions so it is possible that a plan started from near a state on a previous plan will be very different from the previous plan and it is not possible to start using the new plan by the time the planner has finished planning.

Compared to value function based methods for creating policies, trajectory planners have the advantage that they use a time index and do not represent values over states. Thus they can easily deal with discontinuities in the model or cost metric. Errors in representing these discontinuities with a value function can result in the divergence of DP algorithms. Additionally, no discretization is imposed on the trajectories - the state space is only discretized to prune search nodes and for this purpose a high resolution can be used. On the other hand, path planners use weaker reasoning than planning algorithms that explicitly take into account the stochasticity of the problem such as Dynamic Programming using a stochastic model. For example in the marble maze, a deterministic implementation of A* will result in plans that can go arbitrarily close to holes as long as they don't fall into the hole given the deterministic model that A* uses. In contrast to that, Dynamic Programming with a stochastic model will take into account the probabil-

ity of falling into a hole and result in policies that keep some clearance. Unfortunately, Dynamic Programming using stochastic models is even more computationally intensive than Dynamic Programming with a deterministic model. Similar limitations apply to Markov Games, where some information becomes available only in future states and might depend on those future state. This information is not available at the time of planning and planners typically do not take into account potential, future information.

Compared to replanning without remembering previous plans, trajectory libraries not only save computational resources, but also have the advantage of being able to use experience to improve the library. We did this both in the case of the marble maze as well as in the Little Dog case. In the first case, state-action pairs were penalized when they led up to failures. In the case of Little Dog, state-action pairs were penalized when they were invoked multiple times in a row. This results in a limited amount of exploration, as other state-action pairs with potentially different actions get picked in the same state in the future. However, when the planner is invoked, it might re-add the same action in the same state. In order to propagate the information from execution into the planner, one would have to update the model or maybe the cost function. Environments which require complex policies can also result in excessively large policies. For example, one could image arbitrarily complex environments in which small changes to the state require different actions. In this case, replanning might be more applicable as it is not affected by this complexity - it immediately forgets what the previous plan was.

Finally, it is unclear how to assess the quality of a library. Some distance-based metrics can be used to assess the coverage and robustness of the library: given some state, how far away is it from any state-action pair in the library? Given some radius around this state, how many state-action pairs can be found? (This could provide insights into robustness against perturbations.) Alternatively, one could assess the quality of the library by greedily following the reactive policy in simulation, assuming a deterministic model

and comparing this to a plan from the exact query state. Some research into assessing the quality of a library of motions in the context of synthesizing animations can be found in [38]. Finally, it is possible to imagine interference between two different trajectories which have different solutions (such as two different ways around an obstacle). However, in general picking an action from one trajectory will result in a state that's closer to states on the same trajectory. This is especially true in the Little Dog case where low-level controllers try to hit global foot hold locations. Using optimizers to improve the trajectories in the library will probably result in more congruent trajectories, too.

## 3.6 Conclusion

We have investigated a technique for creating policies based on fast trajectory planners. For the marble maze, experiments performed in a simulator with added noise show that this technique can successfully solve complex control problems such as the marble maze. However, taking into account the stochasticity is difficult using A* planners which result in some performance limitations on large mazes. We also applied this technique on a physical version of the marble maze. In this case, the performance was limited by the accuracy of the model.

In the case of Little Dog, we used a library in order to reduce the amount of replanning. Unlike the marble maze, it is possible to stop and replan in the Little Dog domain. However, it is desirable to reduce the time spent on replanning. Furthermore, by remembering steps from previous plans, experience from previous executions can be remembered in the library to improve future executions. This is not possible in the tabula rasa approach of planning from scratch every time.

# Chapter 4

# Transfer of Policies Based on Trajectory Libraries[1]

## 4.1 Introduction

The previous chapter introduced policies based on trajectory libraries. When controlling a system using a trajectory library as a policy, the current state is used to find the closest state on any trajectory. The action associated with the nearest state is used as the output of the policy (figure 3.1).

In many cases it is desirable to reuse an existing library of trajectories to solve new problems. This is especially true when the library is created manually, since there is no planner to fall back to. In other cases, it is desirable to augment a planner by adding special behaviors to a library that allow the agent to handle particularly tricky parts for which the planner alone cannot find satisfactory solutions. We would like to reuse these behaviors on new problems.

In this chapter, we present a transfer algorithm. The algorithm has two key ideas. One key idea is to represent the library in a feature-based space. The same idea was used in chapter 2 to speed up dynamic programming.

[1]Partially published in [43]

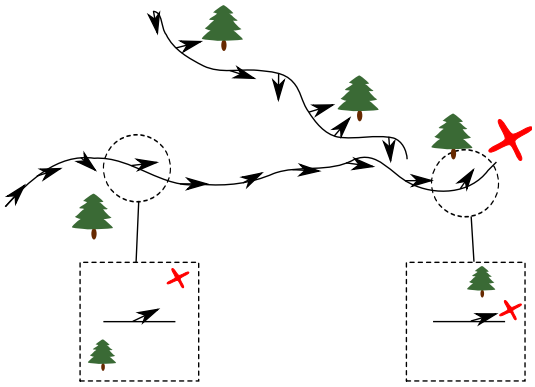


Figure 4.1: Illustration of feature space. On the circled states as examples, we show how state-action pairs could be specified in terms of local features such as the relative positions of obstacles and the goal.

When using features, instead of representing the state of the system using its default global representation, we use properties that describe the state of the system relative to local properties of the environment. For example in a navigational task, instead of using global Cartesian position and velocity of the system, we would use local properties such as location of obstacles relative to the system's position (figure 4.1). This allows us to reuse parts of the library in a new solution if the new problem contains states with similar features. The transfer to a new environment is done by looking for states in the new environment whose local features are similar to the local features of some state-action pair in the source environment. If a match is found, the state-action pair is added into the transferred library at the state where the match was found. The resulting library is a library of the type that encodes possibilities for behaviors in different parts of the environment (see figure 1.1(c)).

The other key idea is to ensure that the library produces goal-directed behavior by searching through the library. In the previous chapter, trajectory libraries were used in a greedy manner to pick an action based on the state of the system. If the library was transferred to a new problem, there is no guarantee that greedily picking actions will get to the goal. Parts of

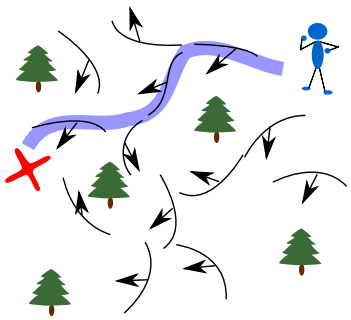


Figure 4.2: Illustration of search through a trajectory library. For the given start state, we find a sequence of trajectory segments that lead to the goal

the relevant state space might not even map to an appropriate goal-directed action. This is especially true if the features used for transfer do not take into account progress towards the goal. Even if individual state-action mappings are goal-directed, it is still possible that following such a greedy policy gets stuck or loops. We search through the library to ensure that following the library will lead to reaching the goal (figure 4.2).

## 4.2 Related Work

Transfer of knowledge across tasks is an important and recurring aspect of artificial intelligence. Previous work can be classified according to the type of description of the agent's environment as well as the variety of environments the knowledge can be transferred across. For symbolic planners and problem solvers, high level relational descriptions of the environment allow for transfer of plans or macro operators across very different tasks, as long as it is still within the same domain. Work on transfer of knowledge in such domains includes STRIPS [16], SOAR [26], Maclearn [22] and analogical reasoning with PRODIGY [48]. More recent relevant work in discrete planning can be found in [14, 51].

In controls, research has been performed on modeling actions using local state descriptions [9, 32]. Other work has been done to optimize low-level

controllers, such as walking gaits, which can then be used in different tasks [8, 25, 40, 50]. In contrast, our work focuses on finding policies which take into account features of the specific task. Some research has been performed to automatically create macro-actions in reinforcement learning [33, 34, 41, 46], however those macro actions could only transfer knowledge between tasks where only the goal was moved. If the environment was changed, the learned macro actions would no longer apply as they are expressed in global coordinates, a problem we are explicitly addressing using feature-based descriptions. Another method for reusing macro actions in different states using homomorphisms can be found in [37].

Bentivegna et al. [5] explore learning from observation using local features, and learning from practice using global state on the marble maze task. Our approach to learning from demonstration takes a more deliberate approach, since we perform a search after representing the learned knowledge in a local feature space.

Our approach is also related to the transfer of policies using a generalized policy iteration dynamic programming procedure in [45]. However, since trajectory libraries are explicitly represented as state-action pairs, it is much simpler to express them in a feature space.

## 4.3 Case Study: Little Dog

The domain to which we applied the algorithm is the Little Dog domain (figure 1.4) described in section 1.2.2. As described in the previous chapter, we can use a footstep planner [10] that finds a sequence of steps going to some goal location. We then use a heuristic method to compute body and foot trajectories that move the dog along the footsteps output by the planner. On some difficult terrains, we are unable to create good sequences of foot steps that can be executed by the heuristic foot step execution.

In order to increase the capability of the robot on difficult terrain, we use

learning from demonstration to navigate the robot across difficult terrain. We use a joystick together with inverse kinematics to manually drive the robot across the terrain and place feet. Sequences of joint angles together with body position and orientation are recorded and annotated with the stance configuration of the robot. The stance configuration describes which legs are on the ground and which leg is in flight. Once the robot has been driven across the terrain, the data can be automatically segmented into individual footsteps according to the stance configuration: a new step is created every time a flight foot returns onto the ground and becomes a stance foot. As a result, every step created in this way starts with a stance phase where all four feet are on the ground. If necessary, we can optionally suppress the automatic segmentation.

Once the steps have been segmented, they can be be used immediately on the same terrain without transfer. To do this, they are added to a trajectory library where the global position of the four feet, as recorded at the beginning of a step, is used to index the demonstrated actions. By using a trajectory library to pick which step to take, the robot can succeed in traversing a terrain even if it slips or if it is just randomly put down near the start of any step. The picking of an action from the library is based on the sum of Euclidean distances between the four feet, as described in the previous chapter.

After a recorded step has been picked from the library, we play back the recorded joint angles. In order to ensure smooth behavior, before the sequence of joint configurations of the chosen step can be played back, every joint is moved along a cubic spline from its current angle to its angle at the start of the chosen step. The time taken depends on how far the body is from its intended position. The speed is chosen conservatively to avoid slipping. Furthermore, if the joint angles are just played back as they were recorded, errors in the position and orientation of the robot would accumulate. In order to solve this problem, an integral controller is added during playback

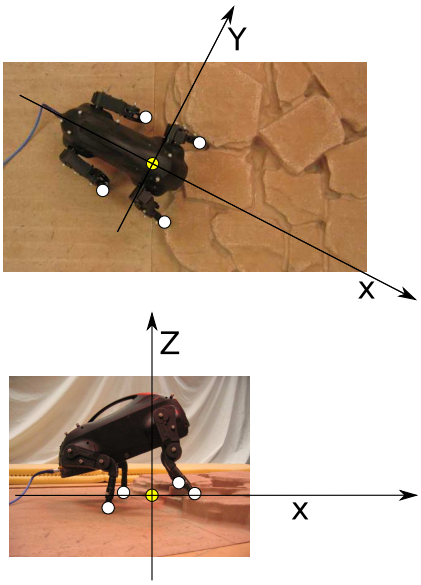
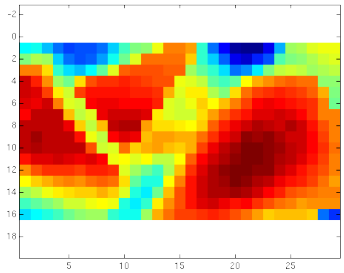



Figure 4.3: Local frame

(a) example pose



(b) resulting heightmap

Figure 4.4: Local heightmap

to correct for errors in body position and orientation. This controller is described in appendix B.

## 4.4 Library Transfer 1

### 4.4.1 Description

Clearly, indexing into the policy based on a global state description such as global position of the feet, limits it to a particular task. If the terrain is moved slightly, the steps will be executed incorrectly. Furthermore, if parts of the terrain have changed, the policy cannot adapt to even such a simple change. In order to solve these problems, we created an algorithm for transferring trajectory libraries to different environments. The transfer

**Algorithm 1** concise transfer and planning description

- transfer library
  - create height profile for every step
  - create height profile for a sampling of positions and orientation on the new map
  - for each step, find best match and transform step to the best match, discard if best match worse than some threshold
- find appropriate steps to get from $s_{start}$ to the goal $s_{goal}$
  if $p$ is a step, $s(p)$ is the start state of the step, $f(p)$ is the final state of the step.
  - add two steps, $p_{start}$ and $p_{goal}$ with $s(p_{start}) = f(p_{start}) = s_{start}$ and $s(p_{goal}) = f(p_{goal}) = s_{goal}$
  - $\forall p, p'$, find the Euclidean foot location metric between $f(p)$ and $s(p')$.
  - Define the successors of $p$, $succ(p)$ to be the n $p'$ with the smallest distance according to the metric.
  - Create footstep plans between all $f(p)$, $s(p')$, s. t. $p' \in succ(p)$
  - Perform a Best-First-Search (BFS) through the graph whose vertices are the footsteps $p$ and directional edges are defined from $p \rightarrow p'$ whenever $p' \in succ(p)$
  - The final library consists of all steps $p$ on the path determined by the BFS as well as all generated footsteps by the footstep planner on that path.

algorithm has to recognize appropriate terrain for applying the demonstrated steps successfully and effectively. The source of the transfer are multiple trajectories which were recorded when the robot traversed different terrain boards in varying directions.

As the first part of the algorithm, a local feature-based description of the environment is used to find appropriate places for state-action pairs. In the Little Dog domain, we create a local height profile (figure 4.4) for each step. The origin of the local frame (figure 4.3) for the profile is the centroid of the global foot positions at the beginning of the step. The x-axis is aligned with a vector pointing from the XY-center of the rear feet towards the XY-center of the front feet. The z-axis is parallel to the global z-axis (aligned with gravity). The height of the terrain is sampled at 464 positions on a regular grid (0.35m $\times$ 0.20m with .012m resolution) around this origin to create a length 464 vector. The grid is normalized so that the mean of the 464 entries is zero. In the same way, we then create local terrain descriptions for a sampling of all possible positions and rotations around the z-axis on the new terrain. The rotations around the z-axis are limited to rotations that have the dog pointing roughly to the right ($\theta \in [-\pi/4, \pi/4]$). For every step in the library, we then find the local frame on the new map that produces the smallest difference in the feature vector. If this smallest difference is larger than some threshold, the step is discarded. The threshold is manually tuned to ensure that steps do not match inappropriate terrain. Otherwise, the step is transferred to the new location by first representing the position and orientation of its start state and all subsequent states in the local frame of the step. We then translate these local positions and orientations back into global coordinates based on the best local frame in the new map.

For performance reasons, after creating the feature vectors for the matching of steps, we used principal component analysis (PCA) to project the vectors into a lower dimensional space. The PCA space was created beforehand by creating feature vectors for one orientation of all the obstacle boards we

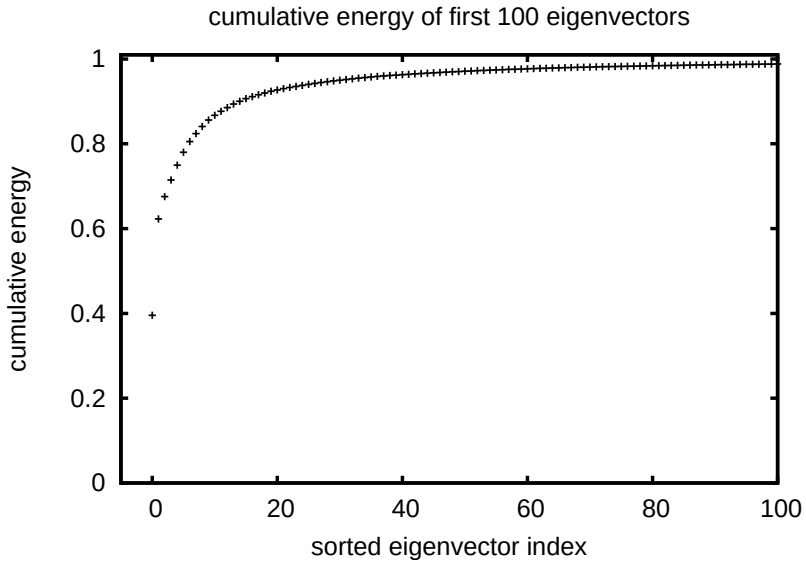



Figure 4.5: Cumulative energy of the first 100 eigenvectors

had. The first 32 eigenvectors, whose eigenvalues summed to 95% of the total sum of eigenvalues, were chosen as the basis for the PCA space (see figure 4.5 for the cumulative energy of the first 100 eigenvectors).

Once all steps have been discarded or translated to new appropriate positions, we perform a search through the library. Due to the relocation, there is no guarantee that the steps still form a continuous sequence. Depending on the size and diversity of the source library, the steps of the new library will be scattered around the environment. Even worse, some steps might no longer be goal directed. In some sense, the steps now represent capabilities of the dog. In places where a step is located, we know we can execute the step. However, it is unclear if we should execute the step at all or in what sequence. We solve this problem by performing a search over sequences of steps. In order to connect disconnected steps, we use a footstep planner [10]. Given the configuration of the robot at the end of one step and the beginning of another step, the footstep planner can generate a sequence of steps that will go from the first to the latter. The same heuristic walking algorithm

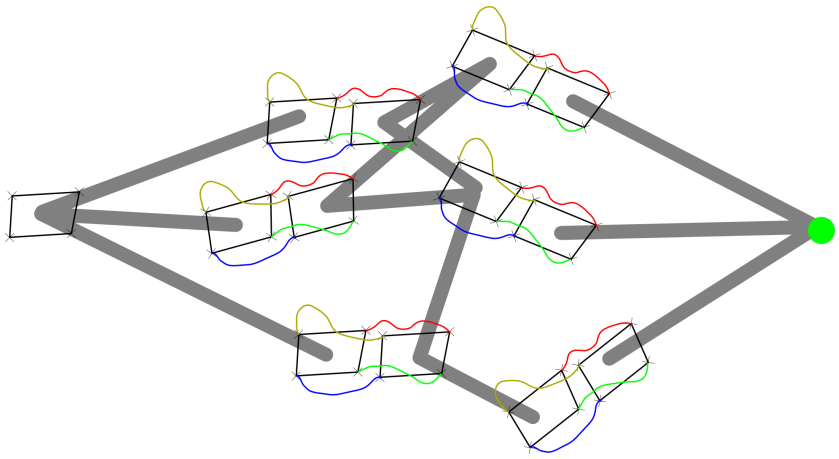


Figure 4.6: Illustration of topological graph after transfer of library. Each behavior is a node in the topological graph. The connections (gray lines) are made using the foot step planner.

as in the previous chapter was used for controlling the body and the actual leg trajectories while executing the footsteps from the footstep planner (see appendix A).

For the search, we generate a topological graph (see figure 4.6). The nodes of the graph are the start state and the goal state of the robot, as well as every step in the transferred library. Edges represent walking from the end of the pre-recorded step represented by the start node to the beginning of the pre-recorded step represented by the target node. The cost of every edge is roughly the number of additional steps that have to be taken to traverse the edge. If the foot locations at the end of the source pre-recorded step are close to the foot locations at the beginning of the target pre-recorded step of the edge, no additional steps are necessary. In order to know the number of additional steps, the footstep planner is used at this stage to connect the gaps between steps when we generate the topological graph. Since the steps that are output by the planner are considered risky, we assign a higher cost to planned steps. (If the planner created reliable steps, we could just use the planner to plan straight from the start to the goal.) In order to reduce

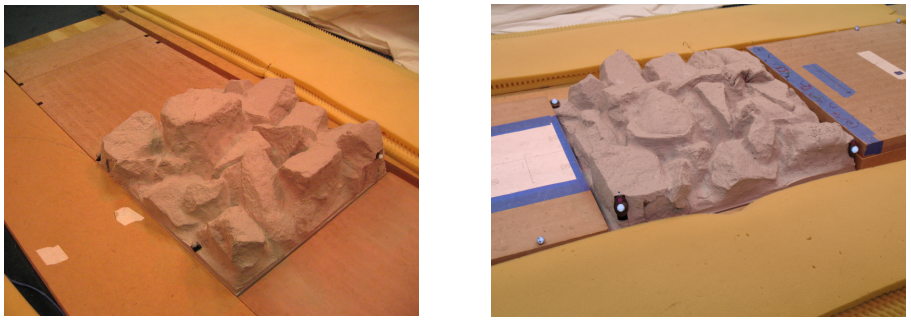


Figure 4.7: Terrains used to create trajectory library

the complexity of the graph, nodes are only connected to the n-nearest steps based on the sum of Euclidean foot location difference metric. We then use a best-first search through this graph to find a sequence of footstep-planner-generated and pre-recorded steps. This sequence is added to the final library.

### 4.4.2 Experiments

We performed several experiments to verify the effectiveness of the proposed algorithms. For all experiments we started with 7 libraries that were created from two different terrains. Using a joystick, one terrain was crossed in four different directions and the other terrain was crossed in three different directions (two examples can be seen in figure 4.8). The combined library contained 171 steps.

In order to test transfer using terrain features, we first looked at transferring the steps from these seven libraries to one of the original terrains. In theory, the steps from the library that was created on the same terrain should match perfectly back into their original location. Some spurious steps from the other terrains might also match. This is indeed the case as can be seen in figure 4.9. The spurious matches are a result of some steps walking on flat ground. Flat ground looks similar on all terrains.

When modifying the terrain, we expect the steps to still match over the unchanged parts. However, where the terrain has changed, the steps should

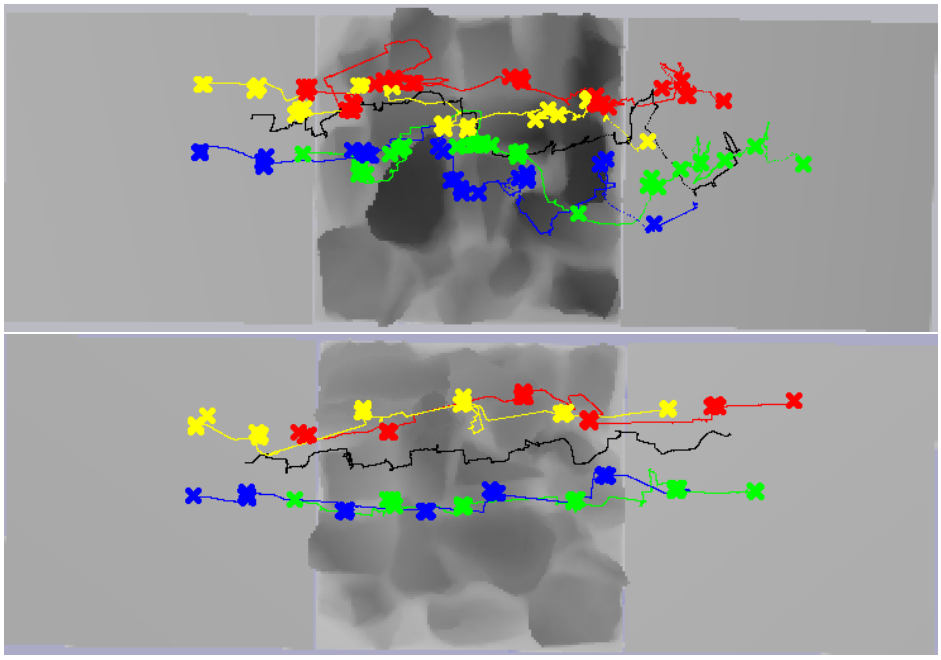


Figure 4.8: Excerpts from the trajectory library. Lines show the actual trajectories of feet and body (cf. figure 1.6). The dog moves from left to right

no longer match. For this experiment we modified the last part of a terrain to include new rocks instead of the previously flat part (figure 4.10). The matching algorithm correctly matches the steps that are possible and does not incorrectly match steps on the modified parts.

While the matching correctly identifies where to place steps in the library, the resulting library needs to be improved, as anticipated. There are large gaps between some steps. Moreover, some spuriously matched steps do not make progress towards the goal but can lead the robot away from it, if they happen to be matched greedily. We now use the search algorithm described earlier to postprocess the resulting library. The resulting plan should select the right steps, throwing out the spurious matches. Furthermore, by invoking the footstep planner to connect possible steps together, it will also fill in any gaps. This happens correctly for the modified map (figure 4.11).

Finally, in order to validate the transfer algorithm, we executed the re-

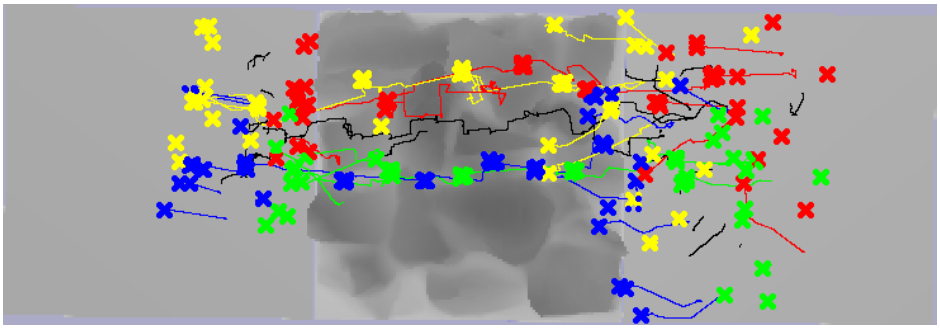


Figure 4.9: Library matched against one of the source terrains. Some crosses do not have traces extending from them, since they are the starting location for a foot from a step where one of the other three feet was moving.

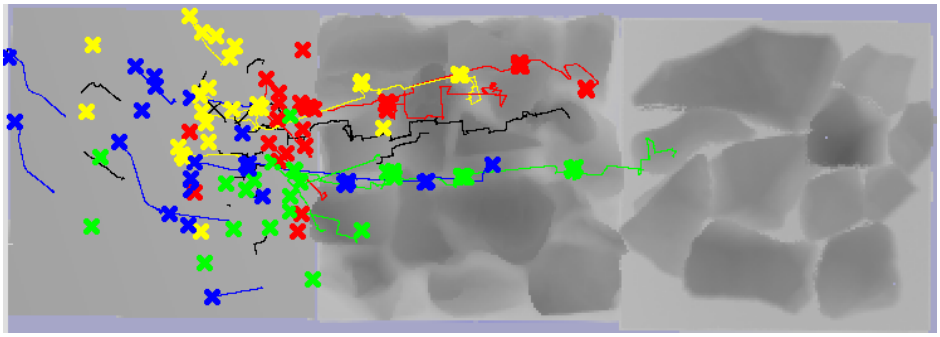


Figure 4.10: Library matched against new, modified terrain

sulting library on the terrain with the modified end board. A plan, from a slightly different start location but otherwise identical to figure 4.11, was executed on the robot and the robot successfully reached the goal, switching between steps from the source library that were created by joystick control and the synthetic steps created by the footstep planner (figure 4.12).

## 4.5 Library Transfer 2

### 4.5.1 Description

The first algorithm has a number of limitations that we worked on removing in a second transfer algorithm. One limitation is that we allowed only one

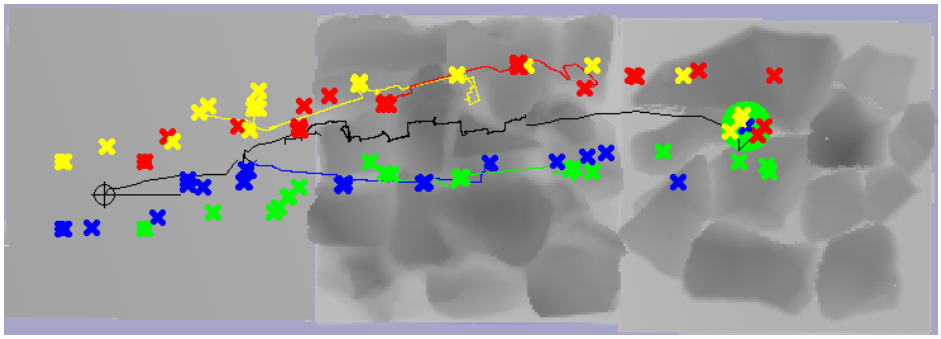
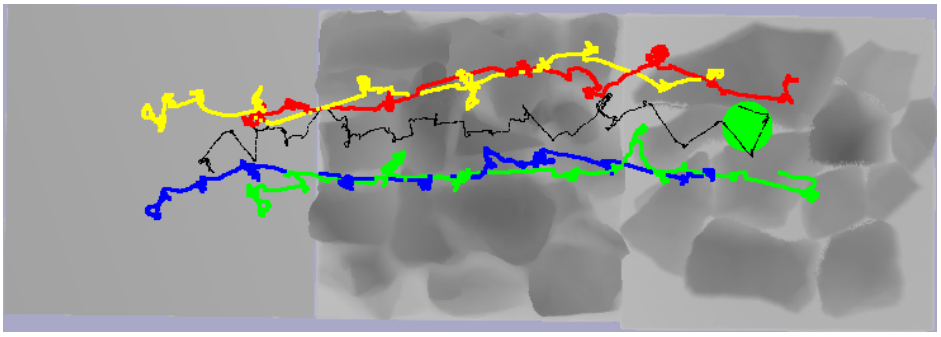


Figure 4.11: Result of searching through the library on modified terrain with the green spot as the goal. The steps coming from the footstep planner do not show traces from the starting places of the feet (crosses), since the foot trajectories are generated on the fly during execution. The body trajectory for planned steps are only hypothetical trajectories — the on-line controller is used for the actual trajectories during execution.

match for each state-action pair in the library of behaviors. In the new algorithm, we allow multiple matches of a step in different parts of the terrain. After a match has been found, we exclude a region around the match from being matched in the future. We then look for additional matches, excluding regions around each successful match, until the PCA error becomes larger than some threshold.

A more significant change was done in the matching algorithm itself. In the first algorithm, matching was done purely based on sum-of-squared error of the PCA feature vectors, which does not take into account the properties of a particular step. In particular for Little Dog, it is important that the terrain supports the stance feet and that neither the body nor the flight foot collide with the terrain. Hence, there are certain variations of the terrain (lower terrain in parts where the stance feet are not supported or higher terrain in parts which are not occupied by any part of the robot) that can be tolerated easily. On the other hand, if the terrain changes under the feet or changes so that parts of the body would collide, the match should no longer be allowed. In order to prevent matches in such cases, the cut-off on the PCA error has to be very low in the first algorithm. This precludes matching even if the terrain is different in tolerable ways.



(a) Trace from the actual execution. Lines show the actual trajectories of feet and body (cf. figure 1.6).

(b) Picture from the same execution

Figure 4.12: Trace of Little Dog executing the plan

In the second algorithm, we added additional checks after a PCA match to verify that a particular match is possible and does not result in collisions. This allows the use of a lower dimensional PCA representation with a higher error cut-off, as the PCA matching only has to recognize where the terrain has similar characteristics to the original training terrain. The PCA error is no longer used to judge if the relocation is valid.

Instead, the new algorithm uses a tiered approach to check the validity of a possible location. The first check is based on foot locations and is responsible to ensure that all feet are supported by the terrain. Before transferring a step,

**Algorithm 2** concise transfer description (algorithm 2)

- create height profile for every step, project into PCA space
- compute foot heights for touch-down and lift-offs
- create swept volume and clearance
- create height profile for a sampling of poses ($P_{\text{pca}}$) on the new map

**for all** steps **do**
  $P_{\text{cur}} \leftarrow P_{\text{pca}}$
  **for all** $p_{\text{pca}} \in P_{\text{cur}}$ sorted by PCA error **do**
    **if** pca_error($p_{\text{pca}}$) > threshold **then**
      break
    $P \leftarrow \text{nearby}(p_{\text{pca}})$
    $p* \leftarrow \arg\min_{p \in P} \text{foot\_error}(p)$
    **if** foot_error($p*$) < threshold **then**
      **if** clearance($p*$) > threshold **then**
        relocate_step($p*$)
        $P_{\text{cur}} \leftarrow P_{\text{cur}} - \text{nearby\_large}(p*)$
      **else**
        **if** variance(foot_error($p \in P$)) < threshold **then**
          $p* \leftarrow \arg\max_{p \in P} \text{clearance}(p)$
          **if** clearance($p*$) > threshold **then**
            relocate_step($p*$)
            $P_{\text{cur}} \leftarrow P_{\text{cur}} - \text{nearby\_large}(p*)$

we compute the location of touchdown and lift-off of all feet during the step in the coordinates of the local frame of the step (see figure 4.3). We then look up the height of the terrain under the foot and compute the height of the foot over the terrain. When checking a new location, we again compute the height under each foot, placing the local frame at the candidate location, and make sure that the height of the foot over the terrain does not change more than some threshold. This can be computed very quickly, as only a small number of foot locations have to be checked.

The foot check is designed to ensure that the feet have the necessary

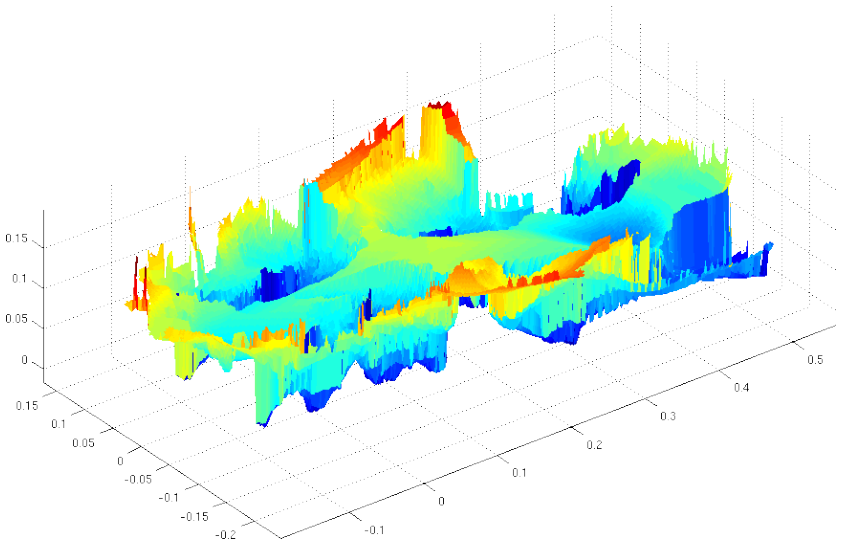



Figure 4.13: Example swept volume for crossing a barrier. The robot moves from left to right. This pictures shows for every point the lowest any part of the robot has been at that point while climbing over a barrier. The smooth surfaces in the center are from the bottom of the trunk. The protrusions downwards on both sides are caused by the legs and feet (which touch the ground while in support). Notice that near x=0.2m, no part of the robot is very low. This is where the barrier was placed.

ground support, however it does not check if the body would come in collision with the terrain. A second, more expensive check is performed to check for collisions if the foot check succeeded. To check for collision, before performing any matches, we compute the swept volume that the body sweeps through space during a step (see figure 4.13). We also compute the clearance (vertical distance) of the swept volume to the terrain. Due to inaccuracies in models and positioning, some parts of the swept volume can have a negative clearance. When checking a possible match, we recompute the clearance in the new location. If no part of the swept volume has a clearance that is worse than the smaller of the original clearance or zero, by some threshold, we allow the match. Otherwise the match is rejected.

As described in the previous section, the PCA matching is performed on a sampling of possible new reference frames for the step. For practical reasons,

the resolution of the PCA samples can be too coarse to find a good match. In order to increase the resolution of the possible matches, we perform a local search in the vicinity of the PCA match. In this local search, we search a range of positions and orientations (constraint to rotations around Z) around the original PCA match. For each possible placement of the reference frame, we first perform the foot check. Then we perform the swept-volume check on the best matched frame based on the foot check error. In case of success, the step is immediately matched.

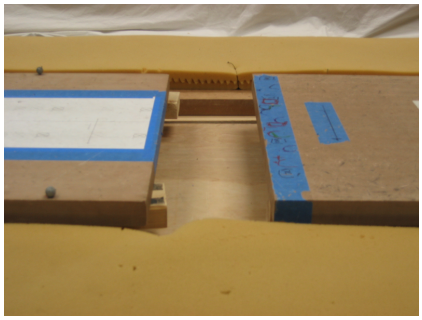
In case of a failure, it is still possible that a match in the vicinity of this PCA match is possible, but that it was not found because the foot error was not informed enough to find this location. For example, if an obstacle is surrounded by flat areas and the feet are only placed on the flat areas, many possible locations will have a good match based on the foot error, but they might still contain collisions with the obstacle. We look at the variance of the foot errors to determine if the foot errors were informative for finding a possible match. If the foot check has high variance, the match based on the foot check is considered informed and should have found terrain similar to the original terrain and the failure is final. However, if the foot check error had little variance, it is possible that the best match based on the foot error was not well informed. As a result we again search over nearby positions and orientations using the collision check instead of foot checking. If the location with the smallest clearance violation is above the collision threshold, the failure is final again. Otherwise the step is relocated to this location. For a concise description of the second matching algorithm, see algorithm 2.

This second algorithm allows matches to locations where the terrain is different in such a way that the unmodified behavior will succeed. However, we also wanted to increase the power of a library of behaviors by allowing simple modifications to the behaviors. In particular, we allow each stance foot to move up or down relative to its original location by a limited amount. The amount by which each stance moves is determined by minimizing the

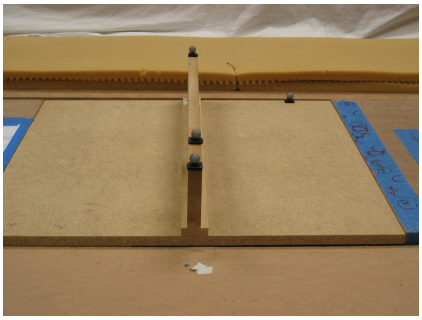
foot error metric. However, moving the stance of a foot does not guarantee the elimination of foot stance error: the foot stance error is computed based on the position of the foot relative to the terrain at touch down as well as lift off. Due to roll of the foot and possible slipping, these two locations are not usually the same. However, if a stance is moved, both lift off and touch down are moved together. If the terrain under the foot in the lift off position is lower than in the original location, but the terrain at the lift off location is higher, moving the stance cannot reduce both errors If the terrain under the foot in the lift off position is lower than in the original location, but the terrain at the lift off location is higher, moving the stance cannot reduce both errors If the terrain under the foot in the lift off position is lower than in the original location, but the terrain at the lift off location is higher, moving the stance cannot reduce both errors.

Once a delta for every stance throughout a behavior is determined, the behavior is modified as follows: Recall that a behavior is specified by a trajectory of desired body positions and desired joint angles. In order to apply the stance deltas, this information is used to compute desired foot positions in the local reference frame of the step. The stance deltas are interpolated through time by creating 1-d cubic splines of deltas between stances (from lift-off to touch-down). Then the interpolated deltas are applied to the trajectories of the desired foot positions and new joint angles are computed through inverse kinematics.

A major hurdle in implementing the modifications into the transfer algorithm is the collision check. Previously, a particular behavior had one swept volume that could be precomputed. However, now every possible location in the local search can potentially have a difference swept volume, as the step is modified to adapt to the terrain. In order to make collision checking practical, we discretized the deltas for each stance to some resolution. We then compute swept volumes separately for the body and each stance-to-stance segment for every foot. Given a delta configuration, we can quickly



(a) gap (22cm wide)



(b) barrier (10cm high)

Figure 4.14: Terrains for gauging transfer with modifications

assemble a complete swept volume by computing the union of these mini-volumes. By caching swept volumes of stance-to-stance segments, we avoid recomputing of previously computed stance-to-stance swept volumes for the feet. Furthermore, we also cache complete swept volumes for a full configuration of discretized stance deltas. This cache is implemented as a Trie with the delta configuration as index. This allows essentially constant time access to a previously computed swept volume for a particular discretized delta configuration with a lower overhead than a hash table, since no hashes have to be computed. Due to this aggressive caching, we can compute collision checks for modified behaviors with little penalty over non-modified steps.

### 4.5.2 Experiments

In order to gauge the effectiveness of the new transfer algorithm, we choose two kinds of terrains that can be easily modified, both in ways that don't require changes to the step and in ways that do require changes to the step: a large gap and a tall barrier (see figure 4.14). Currently, our planning algorithms cannot cross the gap or reliably cross the barrier. In particular, in order to cross the gap, the robot has to fall forward onto a leg - a behavior that the planner cannot currently plan for. Using learning from demonstration, the robot can cross these terrain boards. In order to demonstrate simple

transfer that does not require modification of the original behavior, we modify the terrains by rotating the flat boards for the gap terrain (see figure 4.15) and changing the barriers (see figure 4.16).

When running the transfer algorithm on the modified terrains, multiple matches are found, as desired, for both the gap (figure 4.17) as well as the different kinds of barriers (figure 4.18). After performing the high level search through the transferred library, as described in section 4.4, the robot successfully executes a path made from heuristic steps and the behaviors matched on the terrain (figure 4.19, 4.20).

In a second test, we also modified the terrain in ways that required the behavior to adapt. For this, we raised some of the terrain as seen in figure 4.21. The new algorithm which allows modifications to the stored behavior again matches the terrain as expected and the robot executes the modified behavior successfully, both in the case of the large, modified gap (figure 4.22) and the tall barrier (figure 4.23).

## 4.6 Discussion

We have devised and implemented two algorithms for transferring stored behaviors to new, modified terrain. The first algorithm showed that it was possible to use terrain features to recognize when a step is applicable and where a step is not applicable. Combined with a high-level search over the matched steps, a sequence of heuristic, planned steps and demonstrated behaviors was used to reach the goal. However, the first algorithm could match steps only once and did not take into account the properties of a particular step. Its transfer potential was limited.

We then introduced a second algorithm that improves on the first algorithm in a number of ways. First of all, it allows a step to be matched multiple times. Because it explicitly checks for the applicability of a behavior based on foot support and collision-freeness, the terrain can be matched

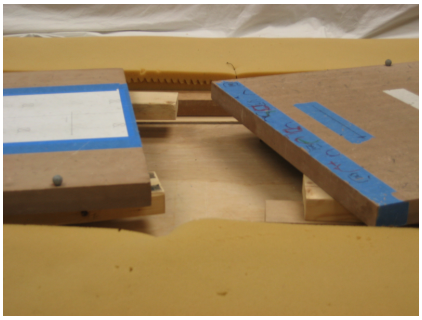
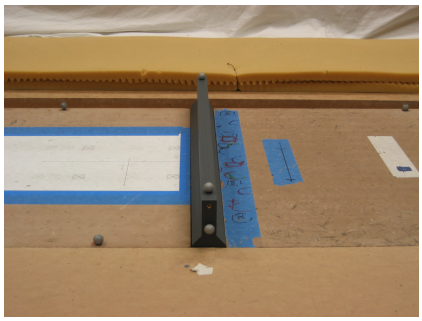


Figure 4.15: Simple modification to gap



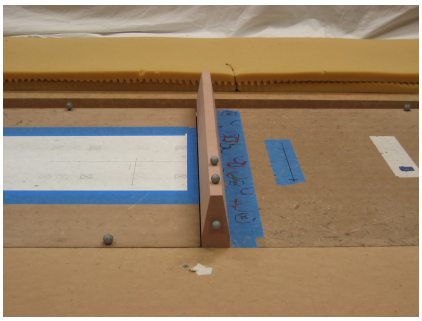


Figure 4.16: Simple modifications to jersey barrier

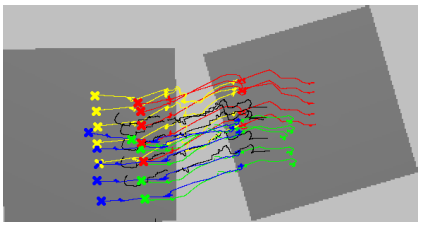


Figure 4.17: Matches on simple gap modification

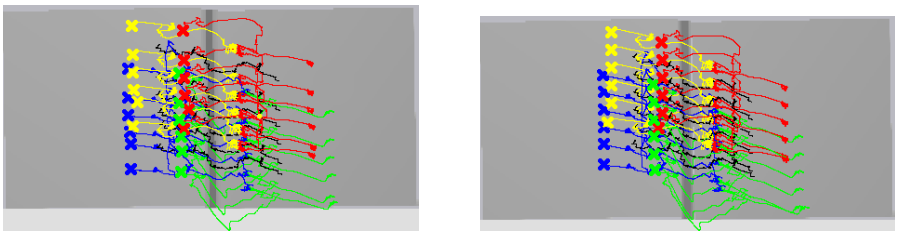


Figure 4.18: Matches on simple jersey barrier modification

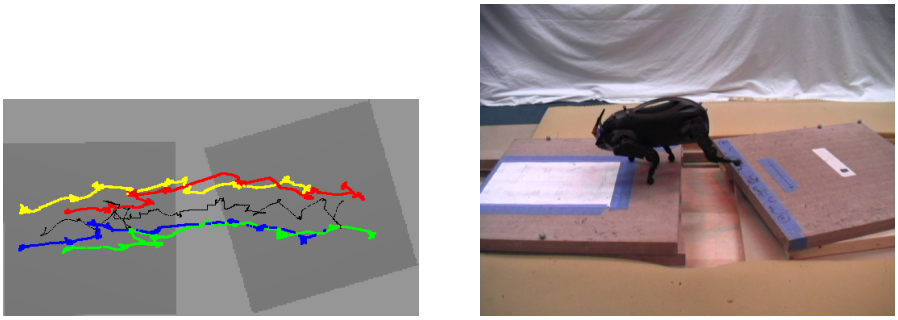


Figure 4.19: Little Dog crossing large gap

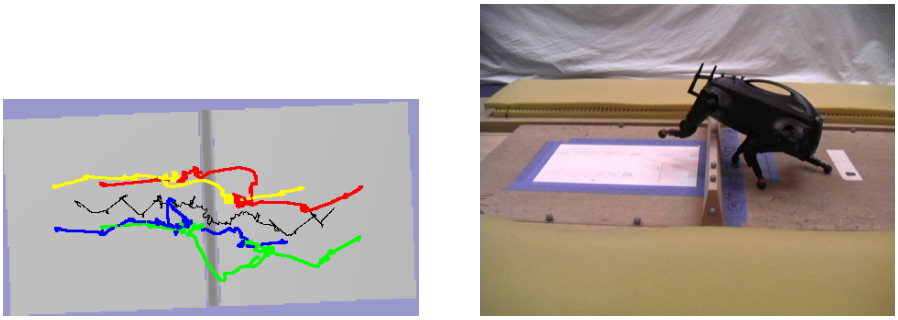
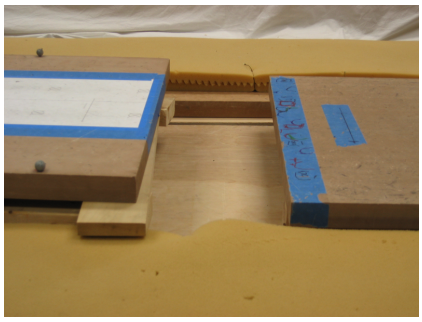
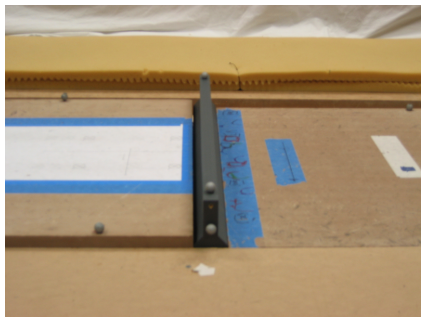


Figure 4.20: Little Dog climbing over barrier

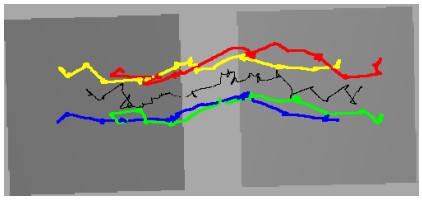


(a) modified gap



(b) modified jersey barrier

Figure 4.21: Modifications that require changes to the behavior



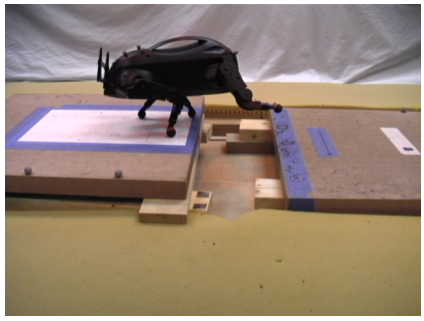
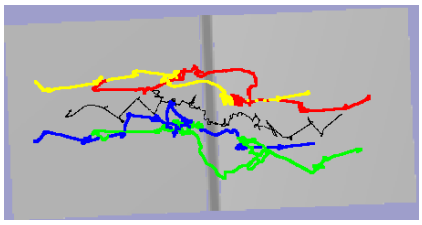


Figure 4.22: Little Dog crossing large gap with height difference using modified behavior



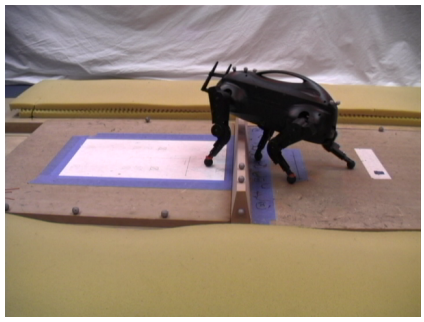


Figure 4.23: Little Dog climbing over barrier with with height difference using modified behavior

more liberally. However, in our experiments, we found it difficult to tune the parameters of the top-level PCA matching. In analyzing the principal components, it became clear that PCA might not be suitable to pick up the kind of terrain features that make a behavior "applicable" in a situation. In particular, in the case of the jersey barrier, the first few principal components with the highest energy resulted, when reprojected into the original space, showed an undulating terrain with no single "barrier" visible. Different low-level representations such as independent component analysis or wavelet decompositions could be explored as alternative representations for picking out "characteristics" that make a particular behavior appropriate.

A limitation of both algorithms is that given a start state, the search through the transferred library only yields a single trajectory. In order to increase the number of trajectories in the final library, one could perform multiple searches from different start states. Alternatively, a backwards search from the goal could be performed and the complete search tree added to the library. Finally, instead of searching once, it is possible to continuously search through the library during execution. Since the search is on a topological graph, this search would be much faster than the search performed by a path planning algorithm in the continuous state space. The gaps between steps are already filled in when creating the topological graph and do not have to be replanned during the continuous search process.

A more radical departure from the current algorithm would be to do away with explicitly finding global states where the features of the state-action pairs from the original library match. Instead, one could greedily match actions from the library based on local features of the start state and its vicinity. After executing the action, this can be repeated. Applying the library greedily based on local features does not allow for searching and might result in dead-ends. Also, it will not allow the robot to cross large gaps in the library if it is not in the vicinity.

Alternatively, one could search for a sequence of steps leading towards

the goal, performing a local search at every expansion to find one or more suitable successor steps in the vicinity of the termination of the previous step. However, this will not work if the local searches fail to find matching steps because of gaps — large areas where no steps in the library match. One could extend the local search area until, in the limit, the complete relevant state space is searched at every expansion. This would essentially be the algorithm that is presented here.

Both algorithms have in common that the computationally expensive operations of computing local features, matching and collision checking are performed as an off-line preprocessing step. The next step of planning through the transferred library of behaviors is also done before execution, but might need to be redone for multiple executions from different start states. Finally, during execution, one is just following a sequence of steps or, if multiple plans are available, performs state-based look ups in a trajectory library fashion (see chapter 3). If a nearest-neighbor look up is used, this can be performed quite fast.

While we believe it is possible to transfer behaviors in a large variety of domains, it might not always be advantageous to do so. In particular, when it is not possible to create useful local features, inappropriate transfer can happen where a behavior is transferred to a location where it is not appropriate to execute that behavior. Furthermore, the behaviors that are being transferred need to add additional capabilities to the system. If the behavior can be created from the action choices available to the planner for the domain, it might be more efficient to have the planner come up with them by itself.

## 4.7 Conclusion

We introduced two methods for transferring libraries of trajectories to new environments. We have shown that the method correctly transfers libraries

in the Little Dog domain based on terrain features. Furthermore, a search is used to effectively find relevant steps on the new terrain and fill in gaps in the library using a footstep planner.

# Chapter 5

# Conclusion and Future Work

We presented several algorithms that advance the state-of-the-art in reinforcement learning and planning algorithms. These algorithms make use of two key insights: using local features enables transfer of knowledge across tasks and libraries of discrete knowledge are a useful way to remember previous computation and avoid future computation. This enabled speeding up the creation of value function based policies, creating a new kind of policies based on libraries of trajectories and adding additional capabilities to path planning algorithms.

As future work, it would be interesting to improve the performance of the trajectory libraries in the marble maze case by learning improved models and applying trajectory optimization to improve trajectories in the library. One could also look for additional applications of local features to the marble maze: by representing a marble maze trajectory library in a feature-based space (c.f. chapter 2), it might be possible to use clustering to find high level "maneuvers" such as "rolling along a wall" or "roll into a corner". These high-level action representations can then be used to plan at a more abstract level.

In the case of transferring behaviors to new problems, it would be interesting to look at alternatives to Principal Component Analysis when looking

for similar features on new terrain. It would also be interesting to look at methods to more closely integrate the hierarchical planners. For example, it would be possible to search for paths to multiple successor behaviors simultaneously. Finally, it would be interesting to look at local features around each foot in order to make more informed transfer decisions.

# Appendix A

# Little Dog Step Execution

## A.1 Overview

Most of the Little Dog experiments described in this paper rely on what we call the "Step Execution". The Step Execution is responsible for creating sequences of desired joint angles that move the body and achieve a given sequence of foot holds. The resulting sequence of joint angles needs to be well coordinated so that the body describes a well-defined motion without the feet slipping. A very powerful method for generating joint angles is to imagine that there is a point on each leg, the foot, that is attached to the ground using a spherical joint and then to imagine turning off the motors and moving the body around by hand. For any position of the body, we can directly transform the position of the fixed foot into body-relative coordinates. Given the body-relative positions of the foot, we can compute corresponding joint angles using inverse kinematics. Using this method, we can compute joint angles for a sequence of body positions. Now we can turn around and instead of imagining moving the body by hand, we just turn the motors on and use the joint positions computed above as desired joint angles for the motors. Assuming the motor controllers achieve the desired joint angles exactly and in synchrony, and assuming that the feet don't slip, the body will describe

the desired motion without causing internal friction on the feet. If we want to move a foot, instead of keeping it's global position fixed on the ground, we can just have it follow a reference trajectory in global coordinates that takes it to the desired new foot hold and compute the inverse kinematics based on its position on that trajectory[1].

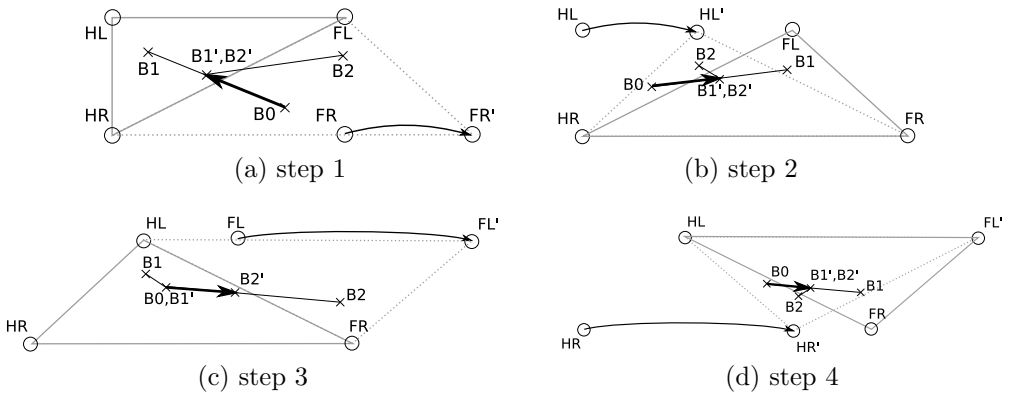
This approach has one important weakness: it assumes that there is a point on the leg that is attached to the ground via a spherical joint - effectively an ankle. However, Little Dog has no ankle and instead has ball feet that roll over the ground as the orientation of the lower leg changes with respect to the ground. As a result, there is no point in the lower leg frame that stays fixed, and using the above algorithm to generate desired joint angles can result in errors in the body pose and even some internal friction. There errors depend on the motion of the reference point in response to the changing orientation of the lower leg. The reference point we used in our experiments is the center of the spherical foot. This point has the advantage that, although it doesn't stay fixed, it will stay at the same height (assuming level ground): it will always be one foot-radius above the ground. Likewise, on flat ground this point is directly above the contact point between foot and ground. As a result, while the body will not exactly follow the desired trajectory, the body's coordinate system will follow the intended trajectory with respect to the support triangle. Furthermore, because the height of the reference point doesn't change, there will not be any orientation error in roll or pitch - only the heading can change. As a result, the position of the center of mass will follow its intended path with respect to the support triangle. This is important for the stability of the robot and the errors introduced due to the rolling foot contact will not cause the robot to become statically unstable. In practice, the internal forces generated by this method are small, as the feet will generally roll together in the same direction.

[1] we first learned about this method of generating joint angles from our colleague Joel Chestnutt

## A.2 Trajectory generation

We just described a method for generating joint angles so that the body and flight foot follow a desired trajectory in global coordinates. The main responsibility of the Step Execution is to generate good trajectories for the body that keep the robot stable and generate good trajectories for the flight foot so that it reaches the desired foot hold without colliding with (stubbing into) the terrain. Our method for generating body trajectories is designed to be statically stable by construction, assuming the robots center of mass is at the designed center of mass of the trunk. There are two ways the constructed trajectories can be unstable. Firstly, the mass of the legs is ignored. For the stance feet, this is not a problem as most of the mass of the leg is within the support polygon and cannot pull the center of mass out of the support polygon. However, when the flight leg is stretching out to reach a foot hold, it can put some mass outside the support polygon and potentially pull the center of mass outside the support polygon. This is especially true for the front legs and for sideways movements of the hind legs. Secondly, when the body is accelerating, dynamic effects can move the center of pressure so that it is no longer supported by the foot contacts. While our approach does not explicitly deal with these effects, we build a margin of error into the body trajectories to accommodate for them. This works very well in practice.

The method for generating body trajectories works by creating target locations for the center of mass inside the support triangle of the three stance feet for each step. We generate this point as a weighted average of the position of the "single" leg (the stance leg on the side of the moving leg), the "action" leg (the leg on the other side corresponding to the moving leg) and the "non-action" leg (the leg diagonally across from the moving leg). The weighting corresponds to how much of the weight rests on each leg. If all legs shared the load equally, we would always move to the centroid. This might seem desirable in order to maximize stability. However, in practice this causes a lot of sway and back-and-forth motion of the body which is not conducive to




(a) step 1 (b) step 2

(c) step 3 (d) step 4

Figure A.1: Sequence of four steps. B0: initial body position. B1: target location based on weighted average of the three support legs. B1': actual target of the body for stance phase, based on entering support triangle on way to B1. B2: target location for likely next step. B2': actual target of the body for the end of flight phase, based on remaining in the support triangle on the way to B2

walking smoothly and quickly. Instead, the weighting is adjusted to reduce sway and back-and-forth motion.

Every step is divided into two phases: a stance phase and a flight phase. During the stance phase, all four legs are on the ground and we move towards the target location (point B1 in figure A.1) in preparation for lifting up the flight leg. For stability, it is not necessary to actually reach the target location. Instead, it is sufficient for the body to have entered the support triangle by some margin. This is the actual target position for the stance phase (B1'). After reaching this point, we switch to the flight phase and lift up the flight foot. During the flight phase, we also start moving towards the next expected target location (B2). The actual target location for the center of mass is the last point on the line B1'-B2 that is still within the current support triangle by some margin (B2'). The trajectory used for generating joint angles is a cubic spline starting at the body position at the beginning of the step (B0), passing through B1' and ending in B2'.

The orientation of the body at the target positions (B1, B2) inside the support triangle is determined by the position of the feet on the ground.

The x-axis of the body's desired coordinate frame is aligned with the line connecting the centroid of the hind feet to the centroid of the front feet. The y-axis is determined by the vector pointing from the "single" foot to its closest point on this line. The z-component of the y-vector is halved to reduce body roll. If necessary to obtain a z-axis that is pointing up, we negate the y-vector. The orientation of the intermediate points B1' and B2' is obtained by performing slerp interpolation between the orientations at B0 and B1 for B1' and by interpolating between B1' and B2 for B2'. The desired height of the body is determined by averaging the height of the terrain under a number of points in the body's desired coordinate frame and adding a predetermined body height.

The flight foot trajectory is loosely based on a trapezoidal pattern determined by the start location, a point at a fixed height above the starting location, the target location and a point at a fixed height above the target location. Additionally, the terrain height is sampled every 5 mm between the start location and the end location and the 1-d convex hull of those points is determined. The trajectory is created as a cubic spline from the start location, the point above the starting location, points above the convex hull by the same fixed height, the point above the target location and finally the target location on the ground. This ensures a smooth, collision free path for the leg without unnecessary vertical movement.

The timing of the trajectories is computed from desired maximum speeds for the body and the flight foot. The length of the stance phase is determined by the larger of how much time it takes to move the body from B0 to B1' at its maximum speed and how much time it takes to rotate the body from the orientation at B0 to the orientation at B1' at its maximum rotation speed. Similarly, the flight phase time is determined by how much time it takes for the flight foot to follow its trajectory at its maximum speed and how much time the body needs to move and rotate from B1' to B2'. During execution, the Step Execution can slow down the flight time if the joint controllers have

errors resulting in the flight foot being too close to the ground or not being over the target location before it is moving down.

# Appendix B

# Little Dog Body Controller I

Every foot $f$ has a three dimensional vector integrator $\boldsymbol{I}_f$ associated with it. Instead of directly using the desired joint angles $\boldsymbol{j}_f$, we use the joint angles to compute the body-relative three-dimensional position of every foot: $\boldsymbol{x}_f = FK_f(\boldsymbol{j}_f)$. The integrated vector is added to the body-relative position of the foot to compute a new body-relative position of the foot: $\boldsymbol{x}'_f = \boldsymbol{x}_f + \boldsymbol{I}_f$. Inverse kinematics is used to compute appropriate joint angles for this new position: $\boldsymbol{j}'_f = IK_f(\boldsymbol{x}'_f)$. These are the new desired joint angles.

In order to update the integrators, we first compute the global terrain-relative position of every foot ($\boldsymbol{X}_f$) using the current joint positions and pose of the robot: $\boldsymbol{x}_f = FK_f(\boldsymbol{j}_f), \boldsymbol{X}_f = POSE(\boldsymbol{x}_f)$. We then hypothesize the robot being in the correct position and orientation and compute body-relative positions of the feet's current terrain-relative positions in the desired body frame: $\boldsymbol{x\_d}_f = POSE\_d^{-1}(\boldsymbol{X}_f)$. Some fraction of the difference between the actual body-relative position of the feet and the ideal body-relative position of the feet is added to the feet's integrators: $\boldsymbol{I}'_f = \boldsymbol{I}_f + k \cdot (\boldsymbol{x\_d}_f - \boldsymbol{x}_f)$. Assuming no further slippage occurs, this control will move the body towards its correct orientation and position. In order to have the feet step into their intended locations, the integrator for each foot is decayed to zero while the foot is in fight. Since the foot is no longer on the ground in this case, the foot

is no longer needed to correct the body position and orientation. Assuming the stance feet succeed in correcting the body's position and orientation, the flight foot, with zeroed integration term, will step into its intended location on the terrain.

# Appendix C

# Little Dog Body Controller II

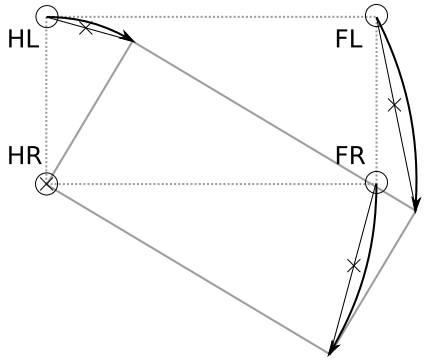



Figure C.1: Illustration of problem with Body Controller I

The Body Controller explained in appendix B suffers from a weakness: the gain is implemented via multiplying the correction vector for each foot. This can result in internal forces. Although the way the error vectors are computed for each foot guarantees that no internal forces are generated if the full correction is applied, a fractional correction can no longer guarantee this, if there is an orientation error. For example, looking at figure C.1, assume the body starts in the position outlined by the solid rectangle and the feet start under the circles. If the desired body position was as outlined by the dashed rectangle, we'd have to smoothly move the feet along the arcs

drawn on each foot. (For illustration purposes, these arcs are drawn with the original body position fixed. In reality the solid body would approach the dashed body, while the feet would stay fixed on the ground.) However, the integrator term added to the integrator is computed based on the total (straight line) vector from start to end position of the foot, which in this example would cause the distances between each foot and the HR foot to decrease, resulting in internal forces and slipping of the feet.

In order to alleviate this problem, we designed an alternative controller. It integrates an error vector for each foot, which is applied as described before. However, the gain is implemented by interpolating between current and the desired body pose, before the update for each foot is computed. Based on this intermediate body pose, the full correction is applied to each foot:

As before, we first compute the global terrain-relative position of every foot ($\boldsymbol{X}_f$) using the current joint positions and pose of the robot: $\boldsymbol{x}_f = FK_f(\boldsymbol{j}_f), \boldsymbol{X}_f = POSE(\boldsymbol{x}_f)$. We then compute an intermediate pose for the body: $POSE\_d' = k \cdot POSE\_d + (1-k) \cdot POSE$. Then we hypothesize the robot being in the intermediate position and compute body-relative positions of the feet's current terrain-relative positions in the intermediate body frame: $\boldsymbol{x\_d}_f = POSE'\_d^{-1}(\boldsymbol{X}_f)$. The difference between the actual body-relative position of the feet and the ideal body-relative position of the feet is added to the feet's integrators: $\boldsymbol{I}'_f = \boldsymbol{I}_f + (\boldsymbol{x\_d}_f - \boldsymbol{x}_f)$. As the gain is used to interpolate poses, all updates to the foot integrators are consistent with some body pose and do not cause internal forces.

# Appendix D

# Little Dog Global Flight Foot Controller

In the previous two appendices, we introduced two possibilities for integral controllers for correcting errors in the trunk pose. In principal, if a body controller is used and succeeds in canceling errors in the body pose, the flight foot will succeed in reaching its desired target location. Unfortunately, we do not have accurate estimates of the body velocity. Without the ability to dampen the body controller using a differential term, it is difficult to get good performance out of the body controllers without risking unstable behavior. While a conservative body controller helped in improving the fidelity of the robot's walk, it was also necessary to add a controller on the flight foot so that it would track its desired trajectory in global coordinates even in the presence of errors in the body pose.

The global flight foot controller is also a purely integral controller, like the body controller, as its function is to remove the steady-state error in the foot's position due to incorrect body position. In theory, it would be possible to compute the error in foot position due to error in the body position and directly apply this difference to the desired body-centric position in the foot before using inverse kinematics to compute desired joint angles. However,

due to noise in the position estimate of the body, magnified by the moment arm of the foot, this performs poorly in practice. Using an integral controller, noise is canceled out over time.

The controller works as follows: The flight foot has a three dimensional vector integrator $\boldsymbol{I}$ associated with it. As described in appendix A, we compute desired joint angles $\boldsymbol{j}$ by performing inverse kinematics $IK(\boldsymbol{x})$ on the body-relative desired position of the foot $\boldsymbol{x}$. The body-relative desired position of the foot $\boldsymbol{x}$ is based on the desired pose of the body and the desired global position of the foot: $\boldsymbol{x} = POSE\_d^{-1}(\boldsymbol{X\_d})$. When using the global flight foot controller, the body integrator is added to the local foot position ($\boldsymbol{x'} = \boldsymbol{x} + \boldsymbol{I}$) before computing the joint angles through inverse kinematics.

The vector integrator $\boldsymbol{I}$ is updated as follows: we first compute the body-centric coordinates for the desired global position based on the *actual* body pose, instead of our desired body pose: $\boldsymbol{x\_new} = POSE^{-1}(\boldsymbol{X\_d})$. We then compute the error in body coordinates of servoing to our original body-centric coordinates: $\boldsymbol{e} = \boldsymbol{x'} - \boldsymbol{x\_new}$. We could then use this error to update our integrator: $\boldsymbol{I'} = \boldsymbol{I} - k \cdot \boldsymbol{e}$. However, this update rule has a major problem: due to noise in the environment models and joint servos, it is possible that servoing to the original global position $\boldsymbol{X\_d}$ will put the foot underground. As this is not possible, trying to servo the foot to this position will cause the body to tilt. Usually, this is not a problem as the "desired" ground penetration is only a millimeter or two. However, when a global foot controller is used, the integrator will wind up, causing the body to tilt excessively, as the the controller tries to get the the foot to the impossible position underground. We eliminated this behavior by rotating the error $\boldsymbol{e}$ into global coordinates and examining it's z-component. If the z-component is smaller than zero (thereby causing the foot to lower it's height), we set the z-component to zero, unless it would decrease the size of the integrator to keep the z-component.

# Appendix E

# Little Dog Collision Avoidance

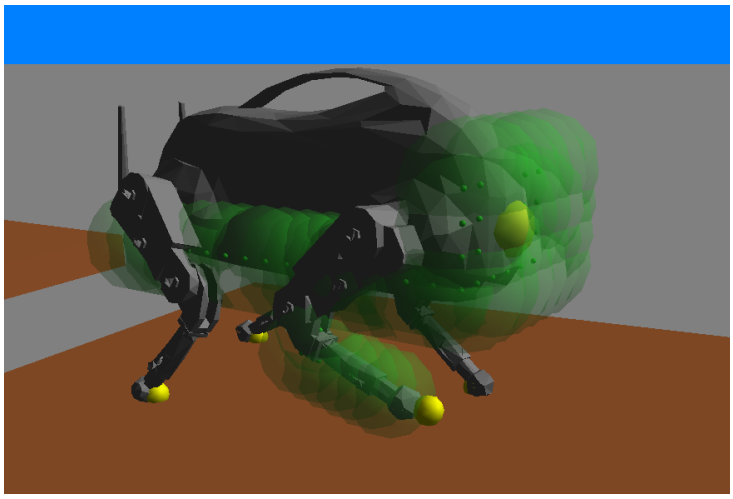


Figure E.1: Virtual bumpers on body and legs

The trajectory generation described in appendix A does not ensure that the trajectories for the body or legs are completely collision free. While the trajectory of the flight foot is designed to stay away from the ground, it is possible that other parts of the leg such as the knee collide with the ground. For most terrain, this is not a significant problem. However, in the case of a tall barrier, collisions with the terrain need to be explicitly avoided. We have

devised a collision avoidance controller that modifies the body trajectory during runtime to avoid collisions with the terrain. It works by creating spherical, virtual bumpers in a number of position on the legs and body (see figure E.1). For each of the spheres, we perform a bounded collision query, which tells us if the sphere is intersecting the terrain and which point on the terrain is closest to the center of the sphere. We then create a virtual force and torque for every sphere, proportional to the penetration distance and moment arm of the terrain into each sphere. The forces and torques are added up and the desired body position is moved and rotated proportional to the summed force and torque, respectively. However, changes in body position are restricted to be along the z-axis in order not to jeopardize the static stability of the center of mass. One important thing to note is that the desired joint angles for the flight foot are still generated based on the unmodified desired body position and desired global trajectory for the flight foot. As a result, if the body is rotated/moved, the flight foot is rotated/moved with it. Otherwise, the virtual forces are not effective in moving the flight leg out of collision. In order to have the flight foot still reach the intended target location, we interpolate between the original desired body position and the collision-avoidance controlled desired body position during the end of the flight phase when the foot is touching down in generating the body-relative desired coordinates for the flight foot.

We use the same approach for avoiding reachability problems. The body trajectory (see A) is generated without regard to reachability of the desired position of the feet. Furthermore, the collision avoidance described here can also generate body poses that make the desired foot positions unreachable. This is alleviated by adding additional forces, attached to the feet, that pull the foot from the closest possible position towards the desired position. These forces are then also used to modify the body position and orientation.

# Bibliography

[1] Christopher G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 663–670. Morgan Kaufmann Publishers, Inc., 1994.

[2] Christopher G. Atkeson and Jun Morimoto. Nonparametric representation of policies and value functions: A trajectory-based approach. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15, pages 1611–1618, Cambridge, MA, 2003. MIT Press.

[3] S. Bailey, R. L. Grossman, L. Gu, and D. Hanley. A data intensive computing approach to path planning and mode management for hybrid systems. In R. Alur, T. A. Henzigner, and E. Sontag, editors, *Hybrid Systems III, Proceedings of the DIMACS Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 485–495. Springer Verlag, 1996.

[4] Darrin C. Bentivegna. *Learning from Observation Using Primitives*. PhD thesis, Georgia Institute of Technology, 2004.

[5] Darrin C. Bentivegna, Christopher G. Atkeson, and Gordon Cheng. Learning similar tasks from observation and practice. In *Proceedings*

*of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems.*, pages 2677–2683, Beijing, China, October 2006.

[6] Mark S. Boddy and Thomas Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285, 1994.

[7] Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *International Conference on Machine Learning*, 1993.

[8] Sonia Chernova and Manuela Veloso. An evolutionary approach to gait learning for four-legged robots. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2004)*, September 2004.

[9] Sonia Chernova and Manuela Veloso. Learning and using models of kicking motions for legged robots. In *Proceedings of the International Conference on Robotics and Automation (ICRA 2004)*, May 2004.

[10] Joel Chestnutt, Manfred Lau, Kong Man Cheung, James Kuffner, Jessica K Hodgins, and Takeo Kanade. Footstep planning for the Honda ASIMO humanoid. In *Proceedings of the IEEE International Conference on Robotics and Automation*, April 2005.

[11] David C Conner, Alfred Rizzi, and Howard Choset. Composition of local potential functions for global robot control and navigation. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 4, pages 3546– 3551. IEEE, October 2003.

[12] C. Connolly and R. Grupen. The application of harmonic potential functions to robotics. *Journal of Robotic Systems*, 10(7):931–946, 1993.

[13] Scott Davies. Multidimensional interpolation and triangulation for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 9. Morgan Kaufmann Publishers, Inc., 1997.

[14] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 191–199, 2004.

[15] Alan Fern, Sungwook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *Journal of Artificial Intelligence Research*, 25:85–118, 2006.

[16] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[17] Emilio Frazzoli. *Robust Hybrid Control for Autonomous Vehicle Motion Planning*. Department of aeronautics and astronautics, Massachusetts Institute of Technology, Cambridge, MA, June 2001.

[18] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[19] R. Grossman, S. Mehta, and X. Qin. Path planning by querying persistent stores of trajectory segments. Technical Report LAC 93-R3, Laboratory for Advanced ComputingUniversity of Illinois at Chicago, September 1992.

[20] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of the Eigtheenth International Joint Conference on Artificial Intelligence*, 2003.

[21] Thomas Howard and Alonzo Kelly. Optimal rough terrain trajectory generation for wheeled mobile robots. *International Journal of Robotics Research*, 26(2):141–166, 2007.

[22] Glen A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989.

[23] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, 1970.

[24] L.E. Kavraki, P. Svestka, J.C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[25] Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 611–616, July 2004.

[26] J.E. Laird, P.S. Rosenbloom, and A. Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[27] Manfred Lau and James J. Kuffner. Behavior planning for character animation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 271–280, New York, NY, USA, 2005. ACM Press.

[28] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. to appear.

[29] Steven M. LaValle and James J. Kuffner, Jr. Randomized Kinodynamic Planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[30] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human motion data. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 491–500, New York, NY, USA, 2002. ACM Press.

[31] Michael L. Littman, Richard S. Sutton, and Satinder Singh. Predictive representations of state. In *Advances in Neural Information Processing*

*Systems*, volume 14, pages 1555–1561. Morgan Kaufmann Publishers, Inc., 2002.

[32] Sridhar Mahadevan. Enhancing transfer in reinforcement learning by building stochastic models of robot actions. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 290–299, 1992.

[33] Shie Mannor, Ishai Menache, Amit Hoze, and Uri Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

[34] Amy McGovern. *Autonomous Discovery Of Temporal Abstractions From Interaction With An Environment*. PhD thesis, University of Massachusetts Amherst, 2002.

[35] Remi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49, Numbers 2/3:291–323, November/December 2002.

[36] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[37] Balaraman Ravindran and Andrew G. Barto. SMDP homomorphisms: An algebraic approach to abstraction in semi Markov decision processes. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*. AAAI Press, August 2003.

[38] Paul S. A. Reitsma and Nancy S. Pollard. Evaluating motion graphs for character navigation. In *Proceedings of ACM SIGGRAPH / Eurographics 2004 Symposium on Computer Animation*, 2004.

[39] E. Rimon and D. E. Koditschek. Exact robot navigation using artificial potential fields. *IEEE Transactions on Robotics & Automation*, 8(5):501–518, October 1992.

[40] Thomas Röfer. Evolutionary gait-optimization using a fitness function based on proprioception. In *Eighth International Workshop on Robocup 2004*, 2005.

[41] Özgür Şimşek and Andrew G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

[42] Martin Stolle. Images of mazes used. Internet/WWW, 2007. `http://www.cs.cmu.edu/~mstoll/files/adprl2007-mazes.tar.gz`.

[43] Martin Stolle and Christopher Atkeson. Transfer of policies based on trajectory libraries. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2007)*, 2007.

[44] Martin Stolle and Christopher G. Atkeson. Policies based on trajectory libraries. In *Proceedings of the International Conference on Robotics and Automation (ICRA 2006)*, 2006.

[45] Martin Stolle and Christopher G. Atkeson. Knowledge transfer using local features. In *Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)*, 2007.

[46] Martin Stolle and Doina Precup. Learning options in reinforcement learning. *Lecture Notes in Computer Science*, 2371:212–223, 2002.

[47] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA., 1998.

[48] Manuela M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie Mellon University, 1992.

[49] Oskar von Stryk. DIRCOL. Internet/WWW, 2001. `http://www.sim.informatik.tu-darmstadt.de/sw/dircol.html.en`.

[50] Joel D. Weingarten, Gabriel A. D. Lopes, Martin Buehler, Richard E. Groff, and Daniel E. Koditschek. Automated gait adaptation for legged robots. In *International Conference in Robotics and Automation*, New Orleans, USA, 2004. IEEE.

[51] Elly Winner and Manuela Veloso. Automatically acquiring planning templates from example plans. In *Proceedings of AIPS'02 Workshop on Exploring Real-World Planning*, April 2002.

[52] L. Yang and S. M. LaValle. The sampling-based neighborhood graph: A framework for planning and executing feedback motion strategies. *IEEE Transactions on Robotics and Automation*, 20(3):419–432, June 2004.

[53] Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9:683–718, April 2008.