Policies Based on Trajectory Libraries

Martin Stolle
Christopher G. Atkeson
Robotics Institute,
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
{mstolle, cga}@andrew.cmu.edu

***Abstract*— We present a control approach that uses a library of trajectories to establish a global control law or policy. This is an alternative to methods for finding global policies based on value functions using dynamic programming and also to using plans based on a single desired trajectory. Our method has the advantage of providing reasonable policies much faster than dynamic programming can provide an initial policy. It also has the advantage of providing more robust and global policies than following a single desired trajectory. Trajectory libraries can be created for robots with many more degrees of freedom than what dynamic programming can be applied to as well as for robots with dynamic model discontinuities. Results are shown for the "Labyrinth" marble maze, both in simulation as well as a real world version. The marble maze is a difficult task which requires both fast control as well as planning ahead.**

I. Introduction

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

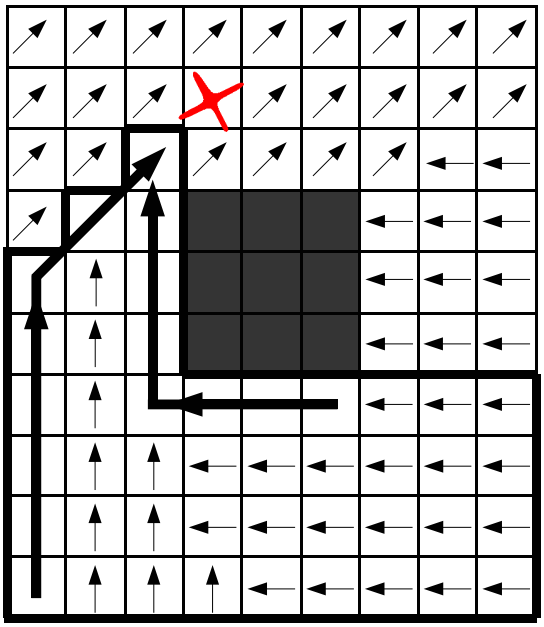

Fig. 1. A library of trajectories (thick arrows) is used to create global policy (thin arrows) by nearest-neighbor look up

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 or time varying linear control law. The desired trajectory can be generated manually, generated by a path planner [1], or generated by trajectory optimization [2]. 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 [3] 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 1 shows a simple grid world example with eight possible actions (N, E, S, W, NE, SE, SW, NW). The cross marks the goal and the dark 3x3 region is an obstacle. The thick arrows show the two trajectories which make up the library. These paths can be created very quickly using forward planners such as A* or Rapidly exploring Random Trees (RRT) [1]. These 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. In the discrete example environment of Figure 1 we have designated the resulting policy for all states using thin arrows. For large parts of the environment, marked by the thick borders, the resulting policy leads to the goal.

II. 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 [4], [5]. 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, ignoring disturbances or inaccuracies.

Another related technique in path planning is the creation of Probabilistic Roadmaps (PRMs) [6]. 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 must 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.

Prior versions of a trajectory library approach, using a modified version of Differential Dynamic Programming (DDP) [7] to produce globally optimal trajectories can be found in [8], [9]. This approach reduced the cost of dynamic programming, but was still quite expensive and had relatively dense coverage. The approach of this paper uses more robust and cheaper trajectory planners and strives for sparser coverage. Good (but not globally optimal) policies can be produced quickly.

III. Case Study: Marble Maze

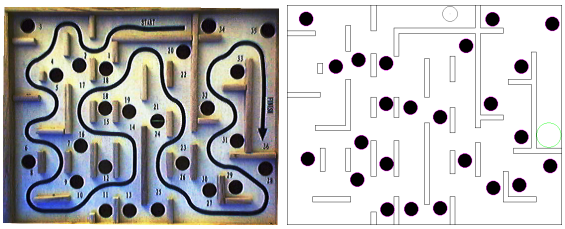

Fig. 2. Original (left) and simulation (right)

The domain used for gauging the effectiveness of the new policy representation and generation is the "Labyrinth" marble maze domain (Figure 2). It 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. The simulation used in this project 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. In order to provide for a more realistic simulator and to gauge the robustness of the new type of policy, Gaussian noise, scaled by the speed of the marble, was added to the applied force in the simulator. A higher-dimensional marble maze simulator was used by Bentivegna [10]. In Bentivegna's simulator the current tilt of the board is also part of the state representation.

Fig. 3. The real world maze

The experiments that were performed on the real world 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 filtering to avoid adding latency to the velocity signal. 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.

IV. Library Details

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 [11] 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 [11]. 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.

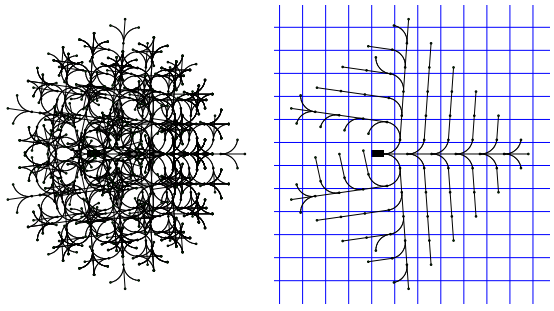

Fig. 4. An example of pruning

The basic A* algorithm is adjusted to continuous domains as described in [12]. 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 4 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 ± 0.007N 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 [7] or DIRCOL [2].

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. In order to speed up nearest-neighbor look ups, the mapping is stored using a kd-tree [13].

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 [10]. 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.

V. Experiments

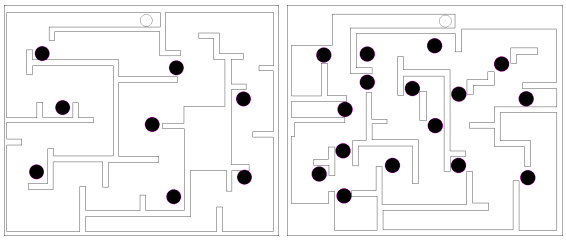

Fig. 5. The two mazes, maze A and maze B, used for testing

We performed experiments on two different marble maze layouts (Figure 5). 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.

The first set of experiments was run in simulation. 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 at the top of Figure 6. 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 7. Initially, many trajectories are added. Once the marble is guided through the maze 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 at the bottom of Figure 6. 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

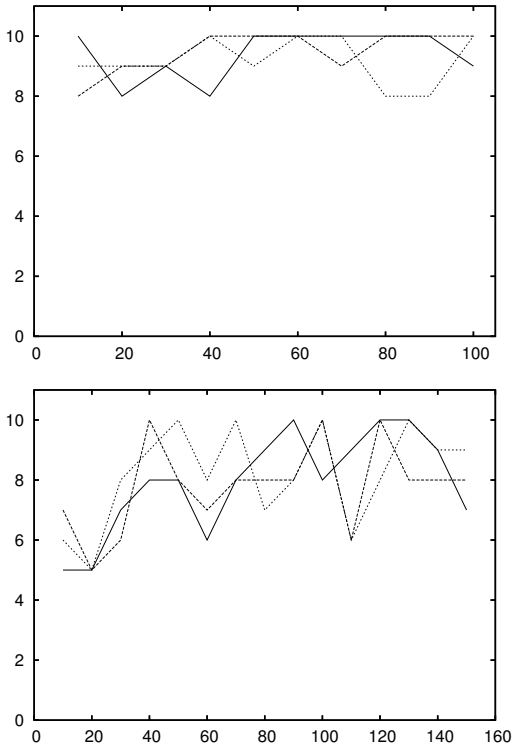

Fig. 6. Learning curves for simulated trials on maze A (top) and maze B (bottom). The x axis is the number of starts and the y axis is the number of successes in 10 starts (optimal performance is a flat curve at 10). We restarted with a new (empty) library three times.

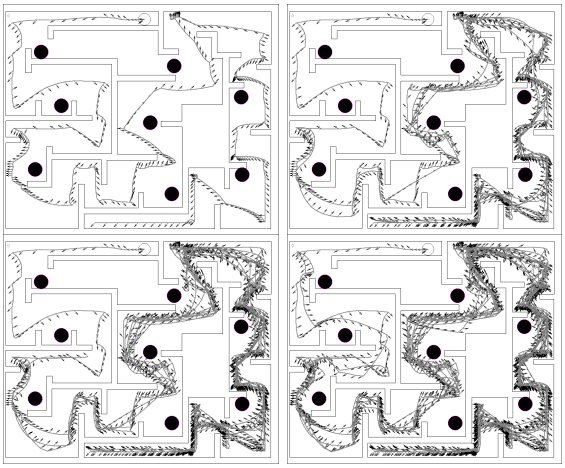

Fig. 7. Evolution of library of trajectories: original single trajectory (before first run), after 10, 30 and 100 runs respectively. (The trajectories (thick lines) are shown together with their actions (thin arrows))

latencies. We used the simulator as 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 8).

In Figure 9 we show the trajectories traveled in simulation and on the real world maze. The position of the marble is

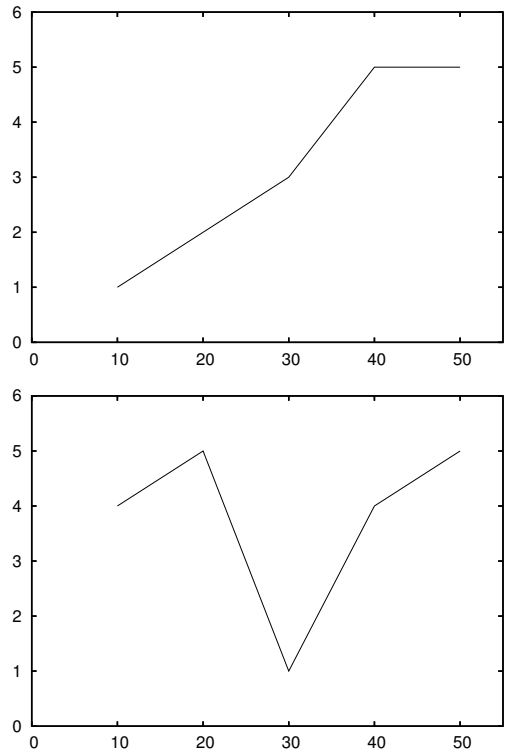

Fig. 8. Real world results for maze A, without (top) and with (bottom) model updates. The x axis is the number of starts and the y axis is the number of successes in 10 starts.

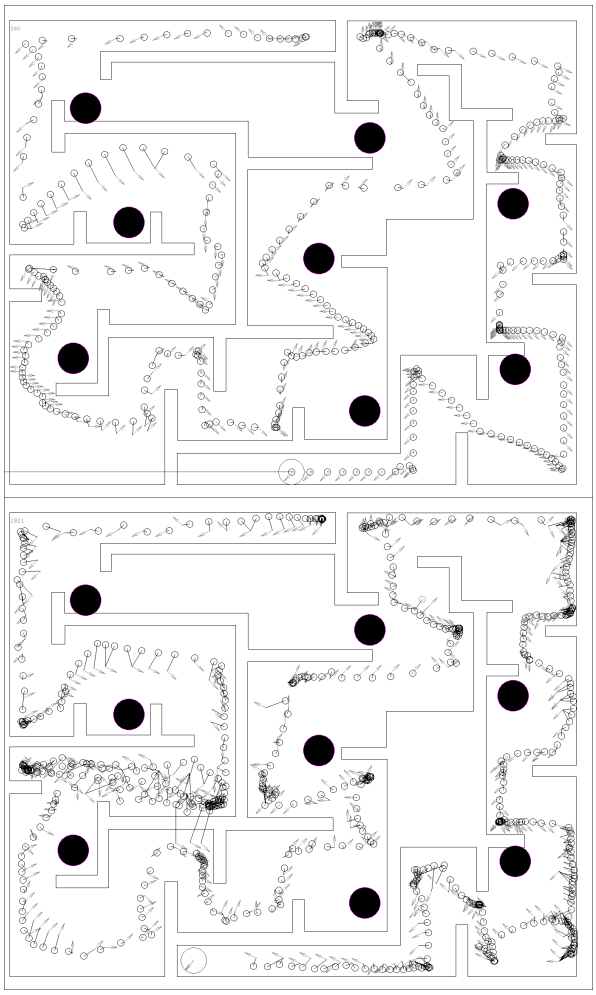

Fig. 9. Actual trajectories traveled in simulation (top) and on the real world maze (bottom). The circles trace the position of the marble. The arrows, connected to the marble positions by a small line, are the actions of the closest trajectory segment that was used as the action in the connected state

plotted 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 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. While initial results using this method are much better, overall the performance does not improve much over not using the model (Figure 8). Clearly, a better model update mechanism needs to be developed. 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.

VI. Discussion

We can create an initial policy with as little as one trajectory. Hence, creating this type of policy can be efficient in its use of computation resources. By scheduling the creation of new trajectories based on the performance of the robot or in response to updates of the model, these policies 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 [3]: 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.

Furthermore, trajectory planners use a time index and do not require value functions. They can easily deal with discontinuities in the model or cost metric. 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.

Currently, the action selection for the policy is based on a nearest-neighbor look up in the library of trajectories. A poor choice of distance metric in this look up can result in a suboptimal selection of actions. In our experiments 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. The model learning technique that we used also relies on a nearest-neighbor look up. The same weights were used for position and velocity. Since the model strongly depends on the action, which are quite small (on the order of 0.007N), the weight in the distance metric for the actions is $1x10^6$. The cutoff for switching over to the simulated model was a distance of 1.

Currently, no smoothness constraints are imposed on the actions of the generated policies. 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). Imposing constraints on the trajectories would not solve the problem as the policy uses a nearest-neighbor look up to determine the closest trajectory and 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.

VII. Future Work

There are several topics for future research. One interesting area would be to use trajectories that were recorded from observing a human teacher. Adding these trajectories is an easy and potentially effective way to initialize the policy.

Furthermore, we would like to also use an RRT planner. This should make this algorithm even faster. Since RRT has been successfully applied to high-dimensional problems [14], we hope that this will enable us to use library-based policies on high-dimensional problems such as biped or quadruped walking. We could then use Differential Dynamic Programming (DDP) [7] or DIRCOL [2] to locally optimize the trajectories.

Also, different ways to use the segments in the library can be explored, such as the average of k-NN segments. Alternatively, we would like to combine this approach with the approach of having a local control law associated with each trajectory. This might result in more robust policies that require fewer trajectories. This approach is used in [8].

Finally, instead of or in addition to changing weights on the segments as a method for improving an existing library, one could imagine using DDP or DIRCOL after model updates to improve existing trajectories.

VIII. Conclusion

We have investigated a technique for creating policies based on fast trajectory planners. 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 real world version of the marble maze successfully. In this case, the performance was limited by the accuracy of the model. A simple model updating technique was used to improve the model with limited success.

References

[1] S. M. LaValle and J. J. Kuffner, Jr, "Randomized Kinodynamic Planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001. [Online]. Available: http://ijr.sagepub.com/cgi/content/abstract/20/5/378

[2] O. von Stryk, "DIRCOL," Internet/WWW, 2001. [Online]. Available: http://www.sim.informatik.tu-darmstadt.de/sw/dircol.html.en

[3] M. S. Boddy and T. Dean, "Deliberation scheduling for problem solving in time-constrained environments." *Artificial Intelligence*, vol. 67, no. 2, pp. 245–285, 1994.

[4] M. Lau and J. J. Kuffner, "Behavior planning for character animation," in *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. New York, NY, USA: ACM Press, 2005, pp. 271–280.

[5] J. Lee, J. Chai, P. S. A. Reitsma, J. K. Hodgins, and N. 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*. New York, NY, USA: ACM Press, 2002, pp. 491–500.

[6] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars., "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996. [Online]. Available: http://ai.stanford.edu/~latombe/pub.htm

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

[8] C. G. Atkeson, "Using local trajectory optimizers to speed up global optimization in dynamic programming," in *Advances in Neural Information Processing Systems*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds., vol. 6. Morgan Kaufmann Publishers, Inc., 1994, pp. 663–670. [Online]. Available: ftp://ftp.cc.gatech.edu/pub/people/cga/local.html

[9] C. G. Atkeson and J. Morimoto, "Nonparametric representation of policies and value functions: A trajectory-based approach," in *Advances in Neural Information Processing Systems*, S. Becker, S. Thrun, and K. Obermayer, Eds., vol. 15. Cambridge, MA: MIT Press, 2003, pp. 1611–1618. [Online]. Available: http://www-2.cs.cmu.edu/~cga/publications.html

[10] D. C. Bentivegna, "Learning from observation using primitives," Ph.D. dissertation, Georgia Institute of Technology, 2004. [Online]. Available: http://etd.gatech.edu/theses/available/etd-06202004-213721/

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

[12] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006, to appear. [Online]. Available: http://msl.cs.uiuc.edu/planning/

[13] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, September 1977. [Online]. Available: http://portal.acm.org/citation.cfm?doid=355744.355745

[14] J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue, "Motion planning for humanoid robots," in *Proceedings of the Eleventh International Symposium of Robotics Research (ISRR 2003)*, November 2003. [Online]. Available: http://www.ri.cmu.edu/pubs/pub_4747.html