Efficient Search for Robot Skill Learning: Simulation and Reality

Jeff G. Schneider and Roger F. Gans
University of Rochester
Rochester, NY 14627-0226, USA

Abstract

Table lookup with interpolation is used for many learning and adaptation tasks. Redundant mappings capture the important concept of "motor skill" in real, behaving systems. Few robot skill implementations have dealt with redundant mappings, in which the space to be searched to create the table has much higher dimensionality than the table. A practical method for inverting redundant mappings is important in physical systems with limited time for trials. We present the "Guided table Fill In" algorithm, which uses data already stored in the table to guide search through the space of potential table entries. The algorithm is illustrated and tested on a skill learning task using a robot with a flexible link. Our experiments show that the ability to search high dimensional action spaces efficiently allows skill learners to find new behaviors that are *qualitatively* different from what they were presented or what the system designer may have expected. Thus the use of this technique can allow researchers to seek higher dimensional action spaces for their systems rather than constraining their search space at the risk of excluding the best actions. We also present a model for the robot arm, flexible link dynamics, and release mechanism of our robot. Our experiments suggest that the use of even a crude simulation model can be helpful for learning on the real robot.

1 Introduction

Memory-based models such as table lookup with interpolation have been used for many robotic learning tasks [Raibert 77, Atkeson 88, Atkeson 91, Mukerjee & Ballard 85, Moore 90, Moore 91]. The block diagram for a general learning task, and a specific task example (throwing a ball) are shown in Fig. 1. A table residing in the box marked "Skill" holds values for a mapping from a task parameter space to a plant command space. The plant command space is all possible vectors that could be stored in the table. The task parameters are used to index into the table. In our 1-d throwing task, the plant command space is the set of possible joint velocity sequences that can be sent to the robot controller and the single task parameter is the distance the ball travels.

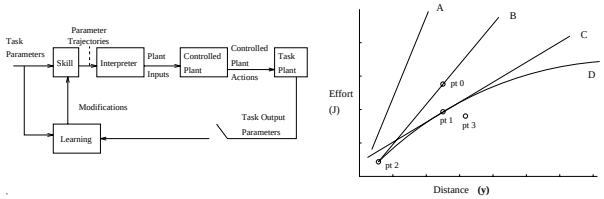

Figure 1: Skill learning system and 1d throwing task

We assume that our learning system operates in two modes: training and operational. It may train first and then remain in operational mode, or it may switch between the two frequently. In either case our goal is one of optimization: to minimize the amount of time required in training to attain a certain performance level in operation, or to maximize the performance level given a certain amount of training time. Performance may be measured with respect to accuracy, range of operation achieved, or a control effort metric.

Often, memory-based learning systems have relied on random search to fill in the table with the necessary information. This works when: the action space is inherently the same size as the task result space, the system designer has explicitly constrained the action space to be of moderate size using partial task models, or despite the size of the action space the system is interested in learning the results of all possible actions. Robot kinematic and dynamic learning systems often fall into the first and/or last categories. Skill learning systems often fall into the second category. In these kinds of system configurations random search is acceptable. Moore [Moore 90] considered tasks whose action space is of moderate size, but whose desirable actions make up a small portion of the space. He proposed an efficient search strategy for these tasks.

We consider tasks whose space of task parameters has low dimensionality (a small table), but whose space of plant commands has high dimensionality (large vectors stored in the table slots). Tasks of this type arise with

open-loop control or planning, when an entire sequence of actions is derived from a single current state or goal. Each action in the sequence makes up a dimension of the space of possible action sequences, and different sequences can achieve the goal at different costs: there is redundancy in the mapping. Discrete closed-loop control avoids high-dimensional action spaces by choosing a single action at each time step. Three reasons for open-loop control are: 1) the action is fast with respect to sensor or control time constants. This problem could also be addressed by increasing sensing, computation, and control bandwidth. 2) there is a lack of sensors or controls for state during the task (e.g. during the flight of a thrown ball). 3) delay, which can destabilize a feedback system [Brown & Coombs 91]. The "Guided Fill In" algorithm given here addresses the problem of high-dimensional search to fill a small table. We test the algorithm with an open-loop robot throwing task both in simulation and on a real system.

In addition to standard table lookup methods, local function approximation methods like Kohonen maps [Ritter et al. 92], CMACs [Albus 75, Miller et al. 89], radial basis functions [Poggio & Girosi 89], and back propagation neural networks [Rumelhart et al. 86] store, retrieve, and interpolate between given data points. [Mel 90] combines a neural network approach with a depth first search of possible reaching strategies. Each method develops an efficient representation once a suitable set of input-output pairs has been found. However, none of these addresses the problem of efficiently obtaining the data to be learned. Often, the method is to let the system execute random plant commands and observe the results, which works well when the space of possible plant commands is not unreasonably large.

There is other work that attempts to perform the types of robot skills used to test the "Guided Fill In" algorithm. The use of global function approximation methods for robot skill learning was reported in [Schneider & Brown 93]. Work on throwing and juggling is reported in [Schaal & Atkeson 93, Rizzi 92]. In contrast to our work the task is usually constrained to remove redundancy or accurate models are used to approximate the desired mapping.

2 Guided Table Fill In

The Guided table Fill In algorithm is a modification of the SAB controller in Moore's thesis [Moore 90]. He was concerned with the efficient search of action spaces, but did not specifically address the issue of inverting a redundant action to task result mapping. Because of time constraints in real systems, it is often impractical to search the entire plant command space. Therefore, skills with redundant mappings have an increased need for search efficiency during training.

Fields of a Table Entry			
p^{act}	p^{res}	p^{eff}	p^{good}
action	task results	control effort	goodness value

Table 1: Table entry for skill learning

The unique inversion of a redundant mapping from some m-space of plant commands to an n-space of task parameters ($m > n$) requires a penalty function to optimize. For example, our 1-d ball throwing robot has two joints controlled via a sequence of six joint velocity settings ($m = 12$, $n = 1$). The penalty function measures the accuracy of the throw and the control effort (sum of squared joint accelerations). The goal of the system is to find the n-subspace of the plant command space that optimally maps onto the n-dimensional task parameter space.

The result of each system execution is stored in a table. The fields of each entry are listed in Table 1. P^{act} is the action sequence tried and p^{res} the result in task space. P^{eff} is a measure of the control effort required by the action sequence. P^{good} is a goodness value for the entry (its computation is described below).

Guided table Fill In is summarized in Fig. 2. In the first step existing models or teachers can determine points from the plant command space to become the first entries into the table. Random choice is a worst case, but possible. There are two parts to a table entry's evaluation. First (done only once): given a point in plant command space, the system executes the corresponding action (p^{act}) and observes the output (p^{res}) and its control effort value (p^{eff}). The task output parameters determine where the point is recorded in the table. Second (executed once each iteration): compare each point against its neighbors in the table (nearest points in the output space). A point's goodness (p^{good}) is the percentage of neighbors whose effort value is worse than its own.

Step 3 randomly chooses a goal from part of the task space. There are several ways the desired portion of the task space may be specified (discussed later). Step 4 generates candidate actions to accomplish the goal task result. Some of the actions are generated by local random modifications to existing "good" points. The table is searched for the entry that best accomplishes the desired goal considering both accuracy and control effort. The action for that entry is altered with small random changes. Several alterations are done to produce a set of candidate actions.

Moore advocates generating some actions from a uni-

Guided Table Fill In Algorithm

1. Initialize table using approximate task model, teacher, or random choice.
2. Evaluate each point in memory according to its control effort. Assign a "goodness" value based on a comparison of a point's control effort value to its neighbors'
3. Randomly choose goal from desired portion of task parameter space.
4. Generate candidates from plant command space using these methods:
   - Find the table entry whose penalty function is lowest for the desired goal (considering accuracy of task result and control effort). Make random modifications to the action sequence of that entry.
   - For a n-d task space choose $n+1$ "good" samples from the table and interpolate or extrapolate to produce an action for the new goal.
5. Evaluate the probability of success for each of the candidate points.
6. Execute the action with the highest probability of success and store the results in the table (Optionally, readjust the desired range of performance). Goto 2.

Figure 2: See text for a detailed explanation

form distribution over the entire space of actions. The purpose of these candidates is to keep the learner from converging to local minima. Experiments with redundant mappings showed that these candidates were not useful. The probability of a random action in a high dimensional space being useful proved to be too small. However, it is still necessary to generate candidate actions far from the existing set of actions in the table. This is done by using linear combinations of existing "good" points in the table. Several sets of points are chosen randomly with the "best" points having a higher probability of being chosen. Interpolation or extrapolation is done from the chosen points to generate new actions. These actions may be in completely unexplored regions of the action space, but are likely to be more useful than completely random actions because of the way they are generated.

Step 5 evaluates the probability of success as Moore suggests. For each candidate action, p^{act}, the table entry whose action, p^{act}_{near}, is nearest the candidate action is determined and used to estimate this probability. When several task result dimensions are considered, a probability is computed for each dimension and the product of them is the probability for the whole task result. When considering redundant mappings, reducing control effort is also a goal. Therefore, a goal effort is selected (usually a constant, unattainable, low value) and control effort is considered to be another dimension in the above computation. Finally, the candidate with the highest probability of success is executed at step 6. The results of the execution are recorded and a new table entry is made. Steps 2-6 are iterated during the training process.

3 Experiments on Learning without a Model

Since the Guided Fill In algorithm does not require a model, our first experiments involved learning to throw a ball accurately without one. The skill goal is a vector describing the position of a target and its output is sequences of joint velocities for a throwing robot. Here, the robot is the controlled plant and the forces affecting the ball's flight after it leaves the robot make up the uncontrolled plant. Skills are called n-dimensional where n is the number of parameters in the output space of the task: thus a 2-d throwing task has a target lying in a plane, such as the floor.

Experiments were done with 1-d and 2-d throwing tasks. For the 1-d task the robot consists of two joints in a vertical plane (Fig. 1). The control signal is a sequence of joint velocities to be executed over 6 time steps (also called a trajectory), thus making a total of 12 input command parameters. The single output dimension is the distance the ball lands from the base of the robot. In the terms of table lookup, a 12-d space must be searched to fill in a 1-d table. The 2-d throwing task is done with a three joint robot configured like the first 3 joints of a PUMA 760; a rotating base is added to the 1-d thrower. The additional joint yields an 18-d search space. The two task parameter dimensions are the cartesian coordinates of the ball's landing position on the floor.

The penalty function includes the approximate amount of energy required to execute the throwing trajectories (sum of squared joint accelerations) and the average task output error. The approximate energy measure has a second purpose. Robots have limits on their joint velocities and the metric prefers trajectories that stay away from those limits. The average value of the penalty function over the task parameter space will be referred to as the *performance* and the two terms will be referred to separately as *error* and *effort*.

3.1 Rigid Simulation

The results of skill learning on a simulated rigid robot thrower (as in 1) are summarized in Table 3.1 and described in detail in [Schneider 93, Schneider 94]. The sum-

Task	Opt. value	GFI 200 tries	RFI 200 tries	RFI tries to catch up
1d 2.5k	24.4	55.0	267.6	over 100k
1d 5k	164.6	254.4	817.7	10k
1d 10k	467.1	1072.3	2443.8	10k
2d 2.5,1.2k	39.2	155.5	1246.9	over 100k

Table 2: Summary of rigid simulation results: The task description lists the dimensionality and range of distances desired. The optimal is numerically estimated. GFI is the performance of the new algorithm after 200 iterations using automatic range expansion when it produces improved results. RFI is from traditional table lookup with random trials. The last column indicates how many additional trials RFI needs to equal GFI's performance after 200.

mary compares a random algorithm for choosing practice trials to fill in the lookup table (RFI) with the Guided Fill In algorithm (GFI). This simulation has the advantage that it is possible to compute the optimal throws numerically. It is also a fast simulation because no expensive dynamics computations are necessary. This allows us to run many trials quickly.

Results with the Automatic Range Expansion method are not shown explicitly in table. Step 3 of GFI calls for the selection of a goal. Automatic Range Expansion is a method that selects goals based on the robot's current level of performance. It only attempts long throws that are slightly outside the robot's current capabilities. This allows the system to learn the "easy" (or short) throws first and gives some performance benefits. Automatic Range Expansion is also useful when the possible range of performance is not known a priori and the learner will be expected to maximize its capability.

3.2 PUMA with a Flexible Link

The Guided Fill In algorithm was tested on a PUMA 760 with a flexible link attached (a metal meter stick). At the end of the meter stick a plastic cup is attached to hold a tennis ball for throwing experiments. The release point of the ball is not directly controlled. It comes out whenever it is moving faster than the cup. A CCD camera watches the landing area and determines where the ball lands with respect to the base of the robot. Most of the parameters of the experiment were set the same as the rigid 1-d throwing done in simulation. Two joints of the robot are controlled by specifying joint velocities over six 200 ms time steps. The low-level controller, RCCL, interpolates to produce a smooth velocity profile. As in the simulation, the effort function prefers trajectories that are far from the robot's physical limits.

The GFI algorithm was given three sample actions that resulted in throws ranging from 143 cm to 164 cm. Automatic range expansion was used because that option performed the best for 1-d throwing in simulation and because it was not possible to determine the robot's range of capability beforehand. After 100 iterations the robot had learned to throw accurately out to a range of 272 cm (a comparison execution of Moore's algorithm attained a maximum distance of 211 cm). Its accuracy is good enough that it consistently hits a 2 cm screw placed upright anywhere within its learned range of performance. The same cameras that watch the landing position of the ball during learning locate the target during performance mode.

The most interesting result of the learning was the type of action found to produce long throws. The three sample actions smoothly accelerate both joints forward. It seems reasonable that longer throws can be obtained by accelerating more quickly. The learning algorithm tried this and it worked up to a distance of approximately 210 cm (this is also what Moore's algorithm did). It was unable to produce longer throws with that type of velocity profile, though, because of the joint velocity limits on the PUMA. It finally learned to do the following "whipping" motion (shown in fig. 3): The joints are moved forward to flex the meter stick. Then the robot reverses the direction of its joints so that the stick is pushed forward past its flat state. Just as the stick begins to fall back again, the joints accelerate forward. This causes a large bend in the stick. Finally, the uncoiling of the stick combines with the large forward acceleration of the robot to produce a much higher ball release velocity than could be achieved by simple accelerating the joints forward.

The significant aspect of the long throws that are learned is that they are *qualitatively* different from any given to the system at the start. The sequence of events that led to the robot trying the action in Fig. 3 illustrates the GFI algorithm at its best. At iteration 15 the system was shooting for a goal of less than 200 cm (within its current range of operation). It chose an action created as a local random modification in step 3. That action had a significantly lower velocity at time step 2 for joint 3. The result was a throw for a distance of 164 cm with considerably lower control effort than any previous action for that distance. Later, at iteration 30, a similar thing happened with time step 3 of joint 5. The result was a low effort throw of 176 cm. Following that, the algorithm chose several actions that were generated by linear combinations of these unique actions. Large extrapolations from the new points created velocity profiles with the "whipping" motion shown in fig. 3. The penalty function, small random modifications, and extrapolation all worked together to

find new, useful actions in unexplored portions of a high dimensional space without having to resort to brute force search.

4 PUMA Simulations

We have begun the dynamic modeling necessary to simulate the ball-throwing robot, with a view to replacing most of the actual robot cycles with simulated robot cycles. This would make the learning process faster and cheaper. We have not completed this effort as yet, but we have made sufficient progress that it seems appropriate to report the preliminary results here. There are two aspects of the simulation that make it nontrivial. The first is the large displacement deformation exhibited by the meter stick; the second is the lack of a formal release mechanism for the ball. We will discuss these two questions in turn.

4.1 Implementation

The meter stick clearly deflects beyond the range of linear beam theory, as can be seen in fig. 3 taken from an actual experiment. We suppose the deformation to be that of a dynamic elastica (large deformation, small strain) and have adopted the segmented model of Gans [Gans 1992], according to which the meter stick can be modeled by a set of rigid elements connected by linear torsional springs with spring constant NEI/L. N denotes the number of segments, E Young's modulus, I the section moment (so that EI represents the bending rigidity) and L is the effective cantilevered length of the meter stick (taken to be 90 cm). The model takes the form of 2N first order nonlinear ordinary differential equations in time:

$A_{ij}\dot{\Phi}_j = b_i + f_i$

$\dot{\Theta}_i = \Phi_i$

where the coupling matrix A and the vector b are functions of the segment angles Θ_i and their derivatives Φ_i, and the vector f denotes whatever inhomogeneous forcing is driving the system. The Lagrangian of this discrete model converges formally to the Lagrangian of the continuous elastica in the limit that $N \to \infty$, and tests of small oscillation frequency and buckling behavior show that the model is quite good for a relatively small number of segments.

We adopt $N = 3$ for our simulation studies, and we make some additional modifications. We suppose the meter stick to be rigidly attached to the end of the robot arm and the arm to have sufficient torque to drive whatever motions we desire. This compound hypothesis means that we can set $f_i = 0$ for all i and impose the motion of the first segment directly from the robot motion. This reduces the 6 dimensional system needed for a three segment beam to a four dimensional system by elimination of one Θ equation and one Φ equation. We further introduce a small dissipation to account for imperfect elasticity. This is done through a Rayleigh dissipation function in the Lagrangian, and the dissipation parameter is chosen to be proportional to N by analogy to the rigorous requirement for the spring constants. The magnitude of the dissipation is the only adjustable parameter in our current model. It is determinable in principle from measurements of the decay of free oscillations of our meter stick.

The ball is considered separately. We reduce it to a fourth order system of the form:

$m\ddot{x} = f$

$m\ddot{y} = g$

where the scalar force components f and g depend on gravity (possibly air resistance, omitted at this stage of our work) and any contact forces from the meter stick. We will discuss the nature of these forces, and the resulting criteria for flying and recapture shortly. We convert this pair of second order equations to a set of four first order equations, and we add them to the set governing the motion of the meter stick to obtain a coupled eighth order system. We integrate this system in time using the adaptive fourth (fifth) order Runge-Kutta scheme given in [Press et al. 1992]. We typically integrate adaptively over 100 identical subintervals so that we can accumulate simulated data at fixed, repeatable intervals of time. The simulated data presented below use a 2 second overall interval with subintervals of 20 ms. We will describe the complete algorithm shortly.

With this model the simulation seems to be somewhat less "whippy" than the real robot thrower. We do not have actual measurements of the motion of the real meter stick, so this difference cannot be quantified. Some of the difference must be attributable to the apparent extreme crudeness of the simulation model, however some is also attributable to the connection between the meter stick and the robot. Assumed rigid in the simulation, it is actually effected by bolting the stick to a rubber block that is in turn bolted to the robot. This additional degree of freedom surely contributes to larger flexion of the meter stick with respect to the robot.

To calculate a simulated throw we need to add a release criterion to the simulation. As we noted above, the experiment has no formal release mechanism: when the ball's momentum carries it out of the cup, it flies. A careful examination of frames 1-4 of the video (fig. 3) suggests that the ball leaves the cup more than once, being recaptured before it can escape. This phenomenon is indeed observed in the laboratory, and is reproduced in simulation. We simulate the action of this cup by resetting the ball if it

"passes through the bottom of the cup" when it is flying.

The complete algorithm is as follows: We perform a simple Newton- Raphson iteration on the steady equations of motion to determine the initial "sag" of the beam. This typically converges in two steps. We then start an adaptive Runge-Kutta routine assuming the ball to be attached to the meter stick. At the end of each adaptive step, we check the contact force. If it is such that it must "pull" on the ball to keep it in contact with the meter stick we assume the ball to be flying for the next step. We remove the weight of the ball from the end of the stick, and we let the ball's position evolve ballistically. At the end of each flying step, we check the position of the ball. If it intersects the cup, we suppose it to have been recaptured, and set its position and velocity to that of the end of the meter stick for the next iteration. (We also check whether it has hit any of the other segments of the meter stick, but this does not happen in practice.) The recapture mechanism is crude, but seems to perform reasonably.

4.2 Performance of the PUMA Model

For the sequence shown in fig. 3, the simulation gave a throw distance of 247 cm, compared to the actual throw of 281 cm. The final release time in the simulation is later than that in the experiment. We believe the difference in throwing distance stems from: 1. the poor modeling of the meter stick-robot attachment, 2. the crude recapture mechanism, 3. the crudeness of the three segment approximation, and (perhaps) 4. an incorrect value for the damping constant.

Some comments regarding the effects of damping and release-recapture mechanisms are in order. We looked at both effects by suppressing the natural release-recapture model described above, and calculating the maximum possible throw for a given motion by controlling the release time independently. (We calculated the ballistic trajectory for release at the end of each time step and saved the longest. Release for the longest trajectory was typically very near the end of the robot motion, about 1043 ms from the start of the 1200 ms motion.) The longest throws were computed over a range of values for the damping parameter. These maximum simulated throws are all less than the corresponding real throw, leading us to believe that the major contribution to the discrepancy between simulation and experiment lies in the modeling of the attachment of the meter stick to the robot.

We believe that the fundamental problems of simulation have been solved for the flexible-meter stick ball-throwing robot, but that considerable work remains to be done on the details. In our case the experimental aspects of the work were completed before the simulation work was begun. A close working relation between the two aspects of the problem would allow a more accurate simulation of the problem.

4.3 Use of the Model for Learning

The ball thrower model as described above is not accurate enough to be directly applied to the skill learning problem. We believe it can be used in an iterative method that improves the model while learning without requiring a prohibitively large number of real robot executions. We propose the following algorithm (it is similar to the idea of combining real and "mental" practice as seen in [Sutton 90], for example):

1. Perform some learning trials on the real robot.
2. Use the results of these trials to improve the model of the robot.
3. Perform some learning trials in simulation.
4. Choose the best new trajectories discovered in simulation and execute them on the real robot.
5. Repeat.

Initial tests for this alorithm on the ball throwing robot look promising. We started with the 100 trial learning run on our PUMA described previously. Five throws from that set were chosen to calibrate the simulation model. The original model performed those throws with an average error of 33 cm. The model was best able to reproduce those throws accurately (as defined by a squared error criterion) by reducing the force of gravity by 50%, reducing the weight of the ball by 20% and speeding up time by 10%. These parameter settings brought the average distance error down to 10 cm. Unfortunately, changing the parameters by that much takes away their real, physical meaning. In general, it is preferable to improve the model and retain the meaning of the parameters.

The 100 trials on the PUMA took about 45 minutes to execute. In simulation, we did 2000 trials in about 45 minutes on a SPARC 10 (note that use of GFI is still important because time still limits the number of trials we perform). From those 2000, we selected the best 25 throws and tried them on the PUMA. In that set we found several throws that went up to 8 cm farther than our previous maximum. We also found throws for medium distances that reduced the necessary control effort by as much as 40% for those distances. Finally, we noticed that throws whose motion differed significantly from any of the throws used to calibrate the model were not well predicted by the model. Executing the throws on the real robot almost

always yielded a significantly shorter distance than the model predicted. Our conclusion is that the simulation can perfect throwing motions it is given, but is unlikely to find qualitatively different throws that actually work on the real robot.

The inaccuracy of the model in regions of the action space where it was not calibrated is no surprise. In fact, it seems unlikely that adjustment of gravity, ball mass, and time alone will produce a model accurate for all throws. There are two options to compensate for this. The first is to derive a better model of the robot arm dynamics as discussed previously. The second is to fit these parameters separately for different regions of the action space. As learning progresses the system finds appropriate parameter settings for larger areas of the action space.

5 Discussion and Conclusions

There are many important tasks with highly redundant plant command to task parameter mappings. When inverting redundant mappings it is necessary to optimize according to additional cost functions. This poses a problem for standard table lookup methods, which require a random or brute force search of the plant command space to optimize performance.

The Guided table Fill In algorithm extends lookup table learning methods to redundant mappings. It uses data already stored in the table to select plant command space candidates that are more likely to produce good data to add to the table. Linear interpolation and extrapolation between existing good points in the table will yield more good points if the mapping is reasonably smooth. The algorithm also allows natural modifications to learn the easy parts of a task first since it explicitly includes a desired range of task parameters in its decision process.

Experiments using a flexible manipulator for throwing demonstrate the power of the new learning algorithm. Previously, researchers applying learning to robotics attempted to constrain the action space to make the problem tractable. With efficient techniques for searching high dimensional spaces that step may not be necessary. More importantly, the ability to handle high dimensional spaces enables the learner to generate *qualitatively* different behaviors. Often these are the behaviors that the researcher would have eliminated by applying constraints based on poor intuition.

One of the disappointing aspects of work in learning is that it is often applied to tasks where the system designer "already knows the answer." In these situations learning functions more as a fine-tuner to improve accuracy or to fit model parameters. In the throwing experiments presented here, we had speculated that improvements could be made by storing energy in the manipulator. However, it was assumed that this would be done by making an initial backward motion, followed by a forward motion. Only through the use of the GFI algorithm was it revealed that a forward-backward-forward motion was the way to attain a high release velocity, given the constraints on joint velocity, the length of time allocated for the throwing motion, and the natural frequency of the meter stick.

It is now possible to model complex, nonlinear dynamic systems such as highly flexible beams and discontinuous dynamic systems such as the ball release process. We have demonstrated the use of these models for simulation of our ball throwing robot. Furthermore, our experiments indicate that even a crude model of a complex robot system can increase learning ability when there are constraints on real robot executions. Future work includes the construction of a model and flexible throwing system that more closely match each other and the continued experimentation on methods of learning by switching between simulated and real practice trials.

References

[1] J. Albus. A new approach to manipulator control: The cerebellar model articulation controller (cmac). *Journal of Dynamic Systems, Measures, and Controls*, 1975.

[2] C. Atkeson. Using associative content addressable memories to control robots. In *Proceedings of the 27th Conference on Decision and Control*, December 1988.

[3] C. Atkeson. Using locally weighted regression for robot learning. In *Proceedings of the 91 IEEE Int. Conference on Robotics and Automation*, April 1991.

[4] C. Brown and D. Coombs. Notes on control with delay. Technical Report 387, University of Rochester, 1991.

[5] R. Gans. On the dynamics of a conservative elastica pendulum. *ASME Journal of Applied Mechanics*, pages 425–430, June 1992.

[6] B. Mel. *Connectionist Robot Motion Planning: A Neurally Inspired Approach to Visually Guided Reaching.* Academic Press, 1990.

[7] W. Miller. Real-time application of neural networks for sensor-based controlof robots with vision. *IEEE Trans on Systems, Man, and Cybernetics*, July 1989.

[8] A. Moore. *Efficient Memory-Based Learning for Robot Control.* PhD thesis, University of Cambridge, November 1990.

[9] A. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In *Proceedings of the 8th International Workshop on Machine Learning*, 1991.

[10] A. Mukerjee and D. Ballard. Self-calibration in robot manipulators. In *Proceedings of the 85 IEEE Int. Conference on Robotics and Automation*, 1985.

[11] T. Poggio and F. Girosi. A theory of networks for approximation and learning. Technical Report 1140, MIT AI Lab, 1989.

[12] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[13] M. Raibert. Analytical equations vs table look-up for manipulation: a unifying concept. In *Proceedings of the IEEE Conference on Decision and Control*, 1977.

[14] H. Ritter, T. Martinetz, and K. Schulten. *Neural Computation and Self-Organizing Maps.* Addison-Wesley, 1992.

[15] A. Rizzi and D. Koditschek. Progress in spatial robot juggling. In *Proceedings of the 92 IEEE Int. Conference on Robotics and Automation*, 1992.

[16] D. Rumelhart, G. Hinton, and R. Williams. *Learning Internal Representations by Error Propagation*, volume 1, chapter 8. MIT Press, 1986.

[17] S. Schaal and C. Atkeson. Open loop stable control strategies for robot juggling. In *Proceedings of the 93 IEEE Int. Conf. on Robotics and Automation*, 1993.

[18] J. Schneider. High dimension action spaces in robot skill learning. Technical Report 458, University of Rochester, June 1993.

[19] J. Schneider. High dimension action spaces in robot skill learning. In *Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

[20] J. Schneider and C. Brown. Robot skill learning, basis functions, and control regimes. In *Proceedings of the 93 IEEE Int. Conf. on Robotics and Automation*, pages 403–410, 1993.

[21] R. Sutton. First results with dyna, an intergrated architecture for learning, planning, and reacting. In *AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[22] C. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

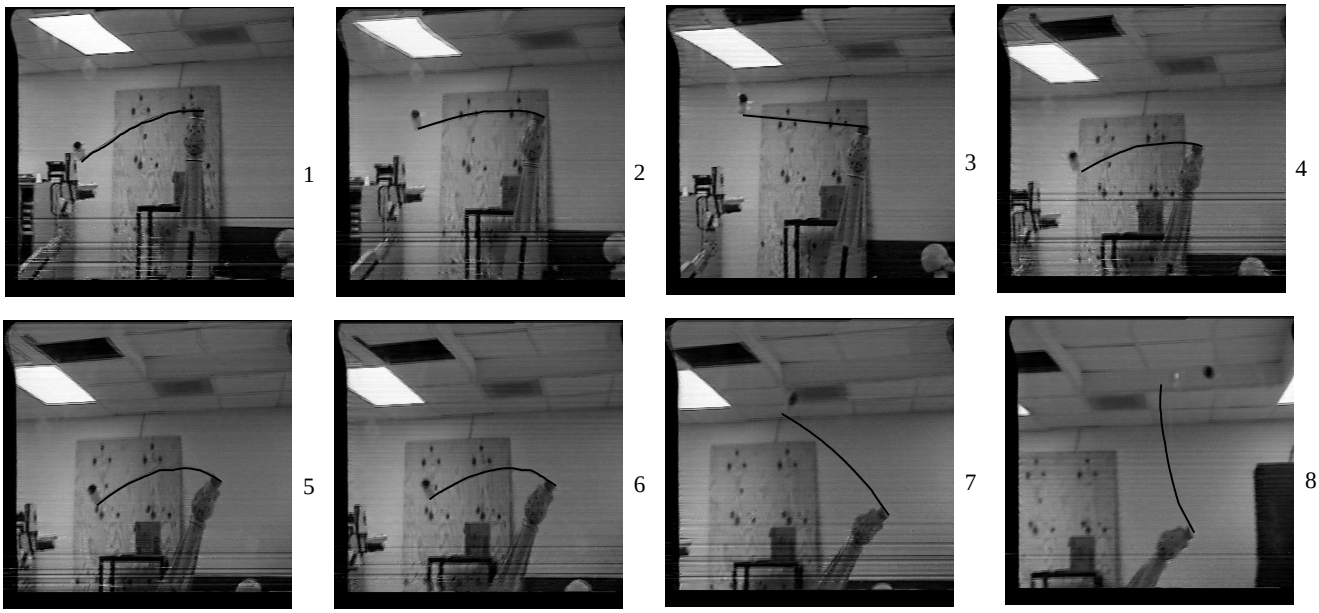

Figure 3: Learned whipping motion for long throws