Introduction

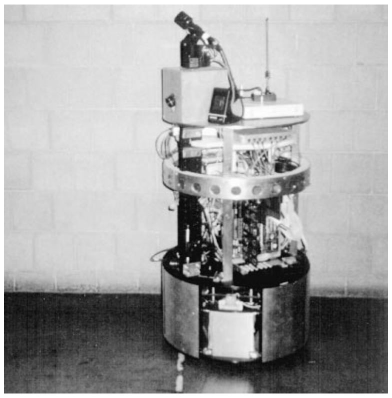
Robot navigation methods that do not explicitly represent uncertainty have severe limitations. Consider for example how commonly used navigation techniques utilize distance information. Landmark based approaches use no distance information at all and consequently suffer from the problem that sensors occasionally miss landmarks and are not able to distinguish between similar landmarks. On the other hand, approaches that use purely metric maps are vulnerable to inaccuracies in both the map making and dead reckoning abilities of robots. To make navigation more reliable, we have developed a navigation technique that explicitly represents uncertain metric information, actuator uncertainty, and sensor uncertainty.

In particular, we use partially observable Markov decision process (POMDP) models to robustly track a robot's position and direct its course. The POMDP estimates the position of the robot in the form of probability distributions. The probabilities are updated when the robot reports that it has moved or turned, and when it observes features such as walls and corridor junctions. To direct the robot's behavior, a decision theoretic planner associates a directive (e.g., turn or stop) with every Markov state. Whenever the probability distribution of the POMDP is updated, the total probability mass for each directive is calculated, and the robot executes the directive with the largest probability mass. A learning module adapts the POMDP to the environment of the robot and, in the process, reduces the initial distance uncertainty and improves the action and sensor models. These improved models are subsequently used by the path planner and lead to better robot performance.

Probabilistic reasoning is known to be time consuming, but an important aspect of its application to robot navigation is that the algorithms must run in real time on board of an actual robot. To make probabilistic reasoning feasible for robot navigation, one therefore has to deal with memory and time constraints and, for learning from the environment, with the problem that collecting training examples is time consuming.

Figure 1: Architecture of the navigation system

In the following, we give an overview of our probabilistic navigation approach and then discuss the methods we use for model learning and path planning. Particular emphasis is placed on the techniques we use to address limited computational resources. All of our algorithms have been implemented and tested on our autonomous mobile office robot Xavier, pictured to the right.



Overview of the Navigation System Architecture

The overall system architecture consists of seven main components (Figure 1). The task of the **robot controller** is to head in a given direction while using sonar information to avoid obstacles. The directives supplied to the controller are to make it stop, go, and change heading. The **sensor interpretation component** converts the continual motion of the robot into discrete action reports (heading changes and distance traveled), and produces sensor reports from the raw sensor data that indicate the observation of high level features, such as walls and corridor openings of various sizes (small, medium, and large) in front of the robot and to its immediate left and right. The **position estimation component** uses these reports and the POMDP to maintain a belief about the current location of the robot by updating the probability distribution over the Markov states. The **action selection component** uses this probability distribution, along with a goal directed policy produced by the **decision theoretic planner**, to choose directives which are sent to the controller. These directives are also fed back to the sensor interpretation component, since the interpretation of features is often heading dependent. In addition to these components, a **model learning component** improves the initial POMDP, that was generated by the **POMDP compiler**.

In [Simmons and Koenig, 1995], we reported our work on the position estimation and action selection component. In the following, we describe the planning and learning components.

Constructing the POMDPs

The POMDP models used by the robot are constructed from three sources of information: the topology of the environment (which we presume can be easily obtained), general knowledge about office environments (such as that corridors are straight and perpendicular to each other), and approximate metric knowledge (obtained either from rough measurements or experience). Before describing how we construct them, we introduce some terminology.

A finite Markov decision process model consists of a finite set of states S, a finite set of actions A, a set of actions $A(s) \subseteq A$ for each state $s \in S$, that can be executed in that state, and transition probabilities $p(s'|s, a)$ for all $s, s' \in S$ and $a \in A(s)$ (the probability that the new state is s' if action a is executed in state s). We also define a set of sensors $i \in I$. The sensors are characterized by observation probabilities $p_i(o|s)$ for all $s \in S$ and $o \in O(i)$ (the probability that sensor i reports feature o when the robot is in state s).

In our case, the Markov model is partially observable because the robot may never know exactly which state it is in. Instead, it maintains a belief of its current state in form of a probability distribution $p(s)$ over the states $s \in S$. The probability distribution is updated in two ways: When an action report a is received, indicating a move or turn, the new probabilities become:

$$p_{posterior}(s) = K \times \sum_{s' \in S | a \in A(s')} p(s|s', a) \times p_{prior}(s')$$

Figure 2: Group of four Markov states representing one location

where K is a normalization factor to ensure that the probabilities all sum to one (this is necessary because not all actions are defined for all states). When a sensor report o is received from sensor i, indicating that a feature has been detected, the probabilities become:

$$p_{posterior}(s) = K \times p_i(o|s) \times p_{prior}(s)$$

Rather than characterizing the observation probabilities for each individual state, we characterize classes of states, such as `wall`, `open` (corridor junctions), `closed-door`, `open-door`, and `near-wall`. For example, a sensor that reports on the immediate left of the robot can be characterized by conditional probabilities such as p(`wall`|`open`) = 0.05, which indicates the probability of observing a wall given that there is really an opening.

Each Markov state encodes both the orientation and location of the robot. Since our corridors are straight and perpendicular to each other, it is sufficient to discretize orientation into the four compass directions: North, South, East, West. The spatial locations of the robot are also discretized. We found a resolution of one meter to be sufficient. Three actions are modeled: turning right 90 degrees (r), turning left 90 degrees (l), and going forward one meter (f). Right and left turn actions are defined for every state (Figure 2). Some states also have forward actions defined for transitioning from location to location (note that forward actions are not defined for states that face walls). Dead reckoning uncertainty is modeled by a self transition, that is, the forward action transitions with some probability from a state into itself.

Our representation of corridors (or, to be more precise, the corridor parts between two adjacent corridor junctions or doors) is a key to our approach. We model corridors as sets of parallel Markov chains, each corresponding to one of the possible lengths of the corridor (Figure 3). The transition probabilities into the first state of each chain are the same as the estimated probability distribution over the possible corridor lengths. Each forward transition after that is deterministic

Figure 3: Markov model for a corridor of length 2 to 4 meters.

(modulo dead reckoning uncertainty — note that the identity transitions are not shown in the figure).

Doorways are modeled with a single exact length Markov chain that leads through a door into a room. Similarly to [Nourbakhsh *et al.*, 1995], doorways have an associated probability *p* that the door is open. Then, the observation probabilities associated with seeing a doorway are:

$$p_i(o|\texttt{door}) = p \times p_i(o|\texttt{open-door}) + (1 - p) \times p_i(o|\texttt{closed-door})$$

We actually use a more complicated POMDP than we have described here. For example, we employ a more space efficient way to encode corridors, use more than one group of Markov states to encode positional uncertainty in corridor junctions, and are able to model foyers and rooms, see [Simmons and Koenig, 1995] for a more comprehensive discussion.

Model Learning

The task of the model learning component is to improve the initial distance estimates and to learn better action and sensor models. Learning better models increases the precision of the position estimation component and consequently the performance of the planning and action selection components as well.

While it seems plausible that Xavier can use experience gained from operating in its environment to adapt the initial POMDP to its surroundings, it cannot simply measure corridor lengths by first moving to the beginning of the corridor and then to its end while measuring the distance traveled. This is due to Xavier's positional uncertainty. A similar problem arises when one attempts to learn better action and sensor models. It is tedious to learn them under controlled conditions, because this requires one to determine the exact location of the robot manually, during the whole training run. Also, the action and sensor models of a robot can be different in different environments, and the training environment might not be representative for the environments in which the robot has to operate after training. The turn action models depend, for example, on the width of the corridors, and the models of sonar based sensors depend on the reflectance characteristics of the walls.

The model learning component takes the existing POMDP and a sequence of action and sensor reports ("execution trace") as inputs and outputs an improved POMDP. It uses a variant of the Baum-Welch algorithm, a simple expectation maximization algorithm for learning partially observable Markov models from observations. In computer science, the Baum-Welch algorithm is best known for its application to speech recognition, but it has also been applied in robotics, for example to interpret tele operation commands [Hannaford and Lee, 1991, Yang *et al.*, 1994].

The Baum-Welch algorithm operates as follows: Given a partially observable Markov model and a sequence of observations ("observation trace"), it estimates a new partially observable Markov model that better fits the trace, in the sense that the probability with which the Markov model generates the trace is increased. This is done in two steps: First, it calculates – for every observation in the trace – the probability distribution over the states at the point in time when the observation was obtained. These estimates are not only based on the observations made prior to the observation in question, but also on all subsequent observations. Second, it estimates new transition and observation probabilities based on these state probabilities, by counting frequencies. This two step hill climbing process can then be repeated with the same trace and the new Markov model. (See [Rabiner, 1986] for details.)

It is known that this process converges eventually to a Markov model which locally, but not necessarily globally, best fits the trace. In order to prevent the Baum-Welch algorithm from over fitting the trace (i.e. adapting to the "noise" in the trace, which would result in a poor generalization performance), one can split the trace into a training trace and a test trace. The test trace is then used for cross validation, stopping the Baum-Welch algorithm when the fit of generated POMDPs starts to decrease on the test trace.

It is trivial to extend the Baum-Welch algorithm from learning partially observable Markov models to learning POMDPs, i.e. to take into account that the robot can choose from several actions in every state. In fact, the Baum-Welch algorithm is an ideal candidate for an implementation of the learning component, because the topology of the POMDP is given and only its probabilities have to be learned. The learning component has the following desirable properties:

- It does not need to have control of Xavier at any time, since it learns by passively observing Xavier's interactions with the world. Thus, training data are obtained whenever Xavier operates.
- The learning component runs in the background while Xavier operates and updates the POMDP from time to time. It does not interfere with the other components, which do not even know of its existence. This is possible because the learning component does not change the structure of the POMDP, but only its probabilities.
- All four kinds of uncertainty can be reduced simultaneously, if desired: the action models, the sensor models, the distance uncertainty, and the uncertainty whether doors are open or closed.
- The calculations of the learning component are not time consuming: it can output a new POMDP in seconds or, for large models, in at most a couple of minutes.
- The learning component can provide a refined POMDP at any time (even if Xavier has only collected a small amount of training data), and the quality of the produced POMDPs improves over time.
- The basic functionality of the learning component does not depend on the topology of the POMDPs. Modifications or augmentations of the structure of the POMDPs do not require major changes to the learning component.

Despite its advantages, the Baum-Welch algorithm has a couple of deficiencies that we need to address before we can use it to implement the learning component.

Memory Requirements: A straight forward implementation of the Baum-Welch algorithm needs arrays of floating point numbers whose sizes are on the order of the product of the number of Markov states and the number of action executions contained in the execution trace. **Training Data Requirements:** The Baum-Welch algorithm requires a large amount of training data, which is time consuming to collect. **Geometric Constraints:** The Baum-Welch algorithm does not know about relationships between probabilities, such as constraints imposed by geometric properties. It may for example learn different distances for traversing a corridor in opposite directions.

We address these problems by augmenting the Baum-Welch algorithm. The nature of our learning problem allows us to apply common machine learning techniques.

To reduce the amount of memory required by the Baum-Welch algorithm, we limit the amount of look ahead that it uses to estimate the probability distributions over the states. We have implemented a version of the algorithm that uses a sliding "time window" on the execution trace to estimate probabilities. The window can be scaled in size to use more or less memory, independent of the size of the execution trace. Using "time windows" adds a small overhead in the run time and causes some loss in precision of the state probabilities, but allows the memory requirements to be scaled to the available memory.

To reduce the amount of training data needed to learn a good POMDP, we reduce the number of model parameters that the Baum-Welch algorithm has to learn. The larger the degree of freedom that the Baum-Welch algorithm has available to fit the training data, the larger is the likelihood that it will over fit the training data, requiring more data to get good generalization. Of course, when we reduce the number of model parameters, we restrict the set of POMDPs that the Baum-Welch algorithm can learn, and thus we have to be careful not to eliminate all POMDPs that closely reflect reality. We use initial knowledge to reduce the number of model parameters. If the knowledge is only approximately correct, we trade off model quality for the amount of training data required to learn a good model. The modified Baum-Welch algorithm is no longer guaranteed to converge, but this is not a problem in practice. We use two different kinds of knowledge:

First, we do not adjust the parameters that we believe to be approximately correct. Action and sensor models, for example, are often similar in different environments and consequently one might have confidence in their initial probabilities. In this case, we do not update them. The POMDPs that are generated during each iteration of the Baum-Welch algorithm only reflect the updated distance and door uncertainties; their action and sensor models remain unchanged.

Second, we aggregate similar states and constrain some of their probabilities to be identical:

- We constrain the observation probabilities of a sensor to be the same for all Markov states in which an ideal sensor would report the same observation. For example, we assume that the probabilities with which the front sensor reports a wall are the same for all Markov states in which there is actually a wall in front of Xavier. Similarly, we constrain the conditional observation probabilities of the left and right sensors to be identical.
- We constrain the turn probabilities to be independent of the Markov state. For example, we assume that the probability of success of a left turn action does not depend on the Markov state in which it is executed. We also constrain the left and right turn probabilities to be symmetrical.
- We constrain the transition probabilities of the forward action to be identical for the initial Markov states in both directions of the same corridor. This forces the length estimates for a corridor to be the same in both directions, thus enforcing geometric constraints.

According to our experience, the Baum-Welch algorithm can learn extremely good models of the environment with three or four traversals of each corridor. Even with only one or two corridor traversals the learned POMDPs are still very good, despite Xavier's noisy sensors.

Decision Theoretic Planning

The use of Markov models for position estimation and action execution suggests to use POMDP algorithms on the same models for goal-directed path planning. At present, however, it is infeasible to determine optimal POMDP solutions given the size of our state spaces and our real time constraints [Lovejoy, 1991, Cassandra *et al.*, 1994]. Our planner therefore associates a directive $d(s) \in D$ with each Markov state (note: these should not be confused with the set of actions $A(s)$

defined for the Markov model). The four directives are: change heading by 90 degrees (turn right), -90 degrees (turn left), 0 degrees (go forward), and stop. The action selection component chooses new directives greedily by selecting the directive with the highest total probability mass:

$$\arg\max_{d \in D} \sum_{s \in S | d(s)=d} p(s)$$

This directive can be calculated very efficiently, making the action selection component very reactive.

Recent advances in approximate algorithms for solving POMDP problems [Parr and Russell, 1995] suggest that it might eventually become feasible to use them for path planning. Consequently, we are pursuing planning and action selection algorithms that approximate optimal POMDP policies. The promise of such methods is that they allow the amount of computation to be traded off against the quality of the resulting plan. How to perform this tradeoff is critical to the performance of the robot. This is one of the research issues that we have considered when designing the navigation architecture of our robot.

We model the path planning problem as a decision problem: Our planner generates a set of alternative plans, evaluates the likelihood of possible outcomes for each alternative, evaluates the desirability of each outcome and, finally, picks the alternative with the best expected value (here: the smallest expected travel time). Generating all possible alternatives is prohibitively time consuming for all but the smallest problems. To approximate a solution, we use several techniques: we use abstractions of the state space when generating plans, we generate and evaluate only a subset of all possible plans, and we do not require plans to be complete before we evaluate them. To decide how closely to approximate a solution, we use methods ranging from simple rules compiled into the planner to more sophisticated on line strategies that monitor and direct the planning activities.

The initial version of our planner was adapted from the planner used in our landmark based navigation system [Simmons, 1994]. It uses A* search in a topological map augmented with approximate metric information (an abstraction of the POMDP) to find a shortest path to the goal. The resulting path is used to assign preferred headings to the edges and nodes in the topological map. Edges and nodes adjacent to the planned path are assigned headings to lead the robot back to the desired path. Directives are then associated with the Markov states: a

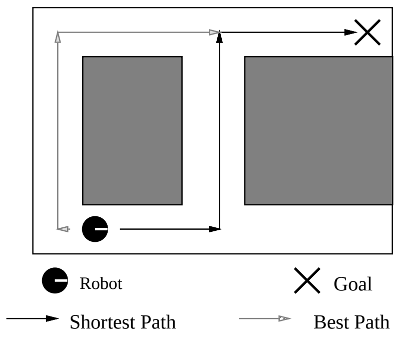


Figure 4: Trading off path length for the ability to follow a path

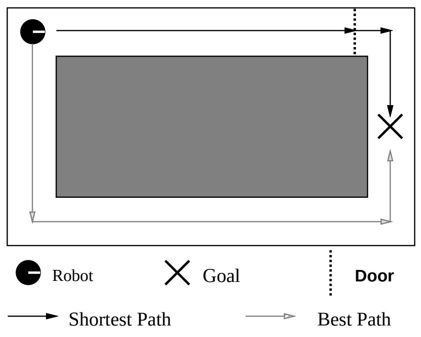


Figure 5: Taking the probability of blockages into account

"go forward" directive is assigned to each state whose orientation is the same as the preferred heading of its associated topological entity. The remaining states are assigned directives that will turn the robot towards the desired heading. Finally, a "stop" directive is assigned to the goal state.

We have extended the planner to make use of probabilistic information that characterizes how easy it is for the robot to follow a path and how likely it is for doors and corridors to be blocked. Our original A* planner considered only the shortest path to the goal (a simple meta level control strategy), but that choice is optimal only if the robot can faithfully follow the path. Similarly, it assumed that all corridors are passable and that it knows whether doors are open or closed. However, selecting a path through a door is optimal only if the door is likely to be open. To remedy these problems, the revised planner uses the A* search algorithm to iteratively generate possible paths to the goal from shortest to longest. As each path is generated, a forward simulation is performed to estimate the expected time needed to follow the path. The simulation takes into account the likelihood of missing turns and the subsequent recovery time. The result is a partial plan that does not specify what to do if the robot encounters a blockage. The planner uses bounds on the expected execution time given the blockage, (without explicitly planning what to do) in order to determine bounds on the expected execution time of the overall plan. If needed, it can narrow this range by creating contingency plans for the potentially blocked passages. The contingency plans may themselves be only partial plans because of the possibility of other blockages.

To understand how the revised planner works and what its advantages are, consider the following simplified navigation problems. Figure 4 shows a situation where the robot can take one of two paths to get to the goal (light and dark lines in the figure). The shorter path requires the robot to find the opening for the central corridor and turn into it. Given the robots noisy sensors, it may miss the opening and then has to backtrack to find it. The forward simulation takes the possible backtracking into account when estimating how long it takes to follow the path. The longer path contains turns only at the ends of corridors, and the geometry physically prevents the robot from overshooting a turn. These physical constraints are reflected in the estimate of the travel time generated by the forward simulation. The revised planner selects the path with the smallest expected travel time, in the case the longer path.

The method that we use for contingency planning is illustrated in Figure 5, which shows a situation where the robot again has two paths to get to the goal. The shorter path goes through a doorway that is closed with probability 0.5. The robot cannot open doors, so it must find another route if the door is closed. The expected execution time of this path is the average of the time it takes to follow this path weighted by the probability that the door is open, and the time it takes to get to the door plus the time it takes to get from there to the goal on the best alternative path weighted by the probability that the door is closed. The latter time can be bounded as follows: A lower bound is the time needed from the door to the goal along the path of the original (now infeasible) plan. An upper bound is the time that it takes to traverse all corridors twice. To see why this is true, imagine that the robot explores all corridors in a depth-first manner and returns to its initial location. It travels along every corridor twice (once in each direction) and – in the process – visits every reachable location, including the goal. Another upper bound is the time it takes the robot to return to its initial location plus the time needed to reach the goal on a different path. The planner uses whichever bound is lower. In Figure 5, only the execution time of the path that does not pass through the door can be computed without considering contingencies. The execution time of the other path can be bounded from below by the time it takes to follow this path and from above by the time it takes to follow the path to the door, return to the initial location (after it has been detected that the door is closed), and then follow the other path to the goal. This interval contains the execution time of the plan that does not pass through the door. To determine which plan is better, the planner has to reduce the range of the execution times of the plan that passes through the door. It therefore creates a contingency plan for the case that the door is closed.

In this simple example, only one such plan is available and there are no other contingencies to plan for, so the range is reduced to a point value. The planner can now compare the point values of both plans to choose the one with the smallest expected execution time and consequently routes the robot away from the door.

Our initial meta level controller uses simple heuristics to limit the scope of the alternatives and contingencies that are considered during planning. The planner considers only a small, fixed number of alternatives (three to ten) and only creates contingency plans for closed doors, but not for other types of blockages. Its run time can still be exponential in the number of doors. However, since the planner only needs to consider doors on a small number of paths, planning time is negligible and the it can easily handle problems considerably more complex than those presented in the examples above.

To improve the performance of the planner and to enable it to deal with even more complex domains, we are currently replacing the simple heuristic based meta level control scheme. The new meta level controller monitors the performance of the planner to decide when to plan and when to begin execution. It uses a sensitivity analysis to decide when to generate more alternatives and which contingencies to plan for. The decision to delay action in favor of finding better plans is justified when the expected gain in plan efficiency outweighs the planning time. In making this decision, our meta level controller also takes into account the fact that the robot can continue to optimize its plan and plan for more contingencies after it begins to execute a plan. (See [Goodwin, 1994] for details.) To decide which contingencies to plan for, the planner uses a simplified sensitivity analysis to identify the contingencies to which the expected value of a plan is most sensitive. Planning for these contingencies first helps to most quickly reduce the range of the expected execution times of plans and thus helps to differentiate among plans. (Planning for likely contingencies also results in plans that are more likely to be completed without the need for replanning during plan execution.) In our office domain, for example, the robot is more likely to encounter a closed door than a blocked corridor, and the range of expected execution times of plans tends to be more sensitive to contingency plans that deal with closed doors. This observation forms the justification for the simple heuristic that only creates contingency plans for closed doors. An on line sensitivity analysis is a better informed version of this heuristic.

Conclusion

This paper has presented our first efforts at using partially observable Markov models (POMDPs) for autonomous office navigation. Our approach enables a robot to utilize all its sensor information, both positional and feature based, in order to robustly track its location. A model learning component, based on the Baum-Welch algorithm, is used to improve the initial distance estimates and adapt the action and sensor models to the environment of the robot. A decision theoretic path planner then uses the learned model to navigate the robot.

Advantages of our approach include the ability to account for uncertainty in the robot's initial position, actuator uncertainty, sensor noise, and uncertainty in the interpretation of the sensor data. Also, by integrating topological and metric information, the approach easily deals with uncertainty arising from incomplete descriptions of the environment. While Markov models are expressive and relatively efficient, they make strong independence assumptions that usually do not hold in the real world. Empirical evidence, however, shows that this is not a problem in practice – the implemented probabilistic navigation system has demonstrated its reliability on the Xavier robot. [Simmons and Koenig, 1995] report experimental results, compare our approach to other robot navigation approaches, and give pointers to the literature.

Acknowledgements

Thanks to Lonnie Chrisman and Joseph O'Sullivan for helping to implement parts of the navigation system and for many valuable discussions.

References

[Cassandra *et al.*, 1994] A.R. Cassandra, L.P. Kaelbling, and M.L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the AAAI*, pages 1023–1028, 1994.

[Goodwin, 1994] R. Goodwin. Reasoning about when to start acting. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS94)*, pages 86–91, 1994.

[Hannaford and Lee, 1991] B. Hannaford and P. Lee. Hidden Markov model analysis of force/torque information in telemanipulation. *International Journal of Robotics Research*, 10(5):528–539, 5 1991.

[Lovejoy, 1991] W.S. Lovejoy. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[Nourbakhsh *et al.*, 1995] I. Nourbakhsh, R. Powers, and S. Birchfield. Dervish: An office-navigating robot. *AI Magazine*, 16(2):53–60, 1995.

[Parr and Russell, 1995] R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the IJCAI*, 1995.

[Rabiner, 1986] L.R. Rabiner. An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–16, 1 1986.

[Simmons and Koenig, 1995] R.G. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of the IJCAI*, 1995.

[Simmons, 1994] R.G. Simmons. Becoming increasingly reliable. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 152–157, 1994.

[Yang *et al.*, 1994] J. Yang, Y. Xu, and C.S. Chen. Hidden Markov model approach to skill learning and its application to telerobotics. *IEEE Transactions on Robotics and Automation*, 10(5):621–631, 10 1994.