# Scheduling of Costly Measurements for State Estimation using Reinforcement Learning

by

Keith Eric Rogers

B.S.E. Aerospace Engineering
Princeton University, 1992

S.M. Aeronautical and Astronautical Engineering
Massachusetts Institute of Technology, 1994

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY IN AERONAUTICS AND ASTRONAUTICS
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 1999

Signature of Author ______________________________
Department of Aeronautics and Astronautics
June 18, 1999

Certified by ______________________________
Wallace E. Vander Velde, Thesis Advisor
Professor of Aeronautics and Astronautics

Certified by ______________________________
Dr. Rami S. Mangoubi, Thesis Supervisor
Member of Technical Staff, Charles Stark Draper Laboratory

Certified by ______________________________
John N. Tsitiklis
Professor of Electrical Engineering

Certified by ______________________________
Eric Feron
Associate Professor of Aeronautics and Astronautics

Accepted by ______________________________
Professor Jaime Peraire
Chairman, Departmental Graduate Committee

# Scheduling of Costly Measurements for State Estimation using Reinforcement Learning

by

Keith Eric Rogers

Submitted to the Department of Aeronautical and Astronautical Engineering
on May 18, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Aeronautical and Astronautical Engineering

## Abstract

There has long been a significant gap between the theory and practice of measurement scheduling for state estimation problems. Theoretical papers tend to deal rigorously with small-scale, linear problems using methods that are well-grounded in optimization theory. Practical applications deal with high-dimensional, nonlinear problems using heuristic policies. The work in this thesis attempts to bridge that gap by using reinforcement learning (RL) to treat real-world problems. In doing so, it makes contributions to the fields of both measurement scheduling and RL.

On the measurement scheduling side, a unified formulation is presented which encompasses the wide variety of problems found in the literature as well as more complex variations. This is used with RL to handle a series of problems of increasing difficulty. Both continuous and discrete action spaces are treated, and RL is shown to be effective with both. The RL-based methods are shown to beat alternative methods from the literature in one case, and are able to consistently match or beat heuristics for both high-dimensional linear problems and simple nonlinear problems. Finally, RL is applied to a high-dimensional nonlinear problem in radar tracking and is able to outperform the best available heuristic by as much as 35%. In treating these problems, it is shown that a useful synergy exists between learned and heuristic policies, with each helping to verify and improve the performance of the other.

On the reinforcement learning side, the contribution comes mainly from applying the algorithms in an extremely adverse environment. The measurement scheduling problems treated involve high-dimensional, continuous input spaces and continuous action spaces. The nonlinear cases must use sub-optimal nonlinear filters and are hence non-Markovian. Cost feedback comes in terms of internally propagated states with a sometimes tenuous connection to the environment. In a field where typical applications have both finite state spaces and finite action spaces, these problems test the limits of its usability.

Some advances are also made in the treatment of problems where the cost differential is much smaller in the action direction than the state direction. Learning algorithms are presented for a class of transformations to Bellman's equation, of which Advantage Learning represents a special case. Conditions under which Advantage Learning may diverge are described, and an alternative algorithm – called G-Learning – is given which fixes the problem for a sample case.

Thesis Supervisor: Rami Mangoubi
Title: Draper Laboratory Technical Staff

Thesis Supervisor: Wallace VanderVelde
Title: Professor of Aeronautics and Astronautics

# Acknowledgments

The road to completion of this thesis has been a long one, and a number of people deserve thanks for helping me stay on it. First of all, I'd like to thank my supervisor at Draper, Dr. Rami Mangoubi, for putting up with me. I know it wasn't an easy task. I'd also like to thank my committee chairman, Professor Wallace VanderVelde, for always keeping an open door and an open ear, and especially for sharing his experience with real-world applications to help me develop a believable application for my research. My mother and brother deserve credit simply for providing a sort of doctoral existence proof: if they had done it, so could I. And of course thanks go to my wife, Harumi, for her constant love and support.

Professor Tsitsiklis at MIT deserves credit for catching various errors in my exposition on reinforcement learning. Professor Sutton of the University of Massachusetts at Amherst (now at AT&T labs) was generous with his time, giving me valuable insights into some of the subtleties of the field. A number of people at Draper lent a helping hand at various points in my research. Among them were Brent Appleby, David Carter, Chris Dever, and Larry McGovern.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under IR&D project number 18612.

Publication of this thesis does not constitute approval by Draper of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

(author's signature)

This page intentionally left blank.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In this thesis we will examine the problem of state estimation with costly measurements. This type of problem shows up in many real-world situations but has seen surprisingly little attention from the engineering community.

It is relatively easy to come up with examples of situations where measurements are costly. For some measurements it is possible to associate actual monetary costs. Recently geologists mounted an expedition to Mt. Everest in order to place a GPS unit at the top. Needless to say, this was an expensive measurement. Every measurement taken by the Hubble Space Telescope is costly when one considers the large expense of putting the instrument in orbit and maintaining it.

The field of radar provides examples of a variety of different measurement costs. The cost of a radar measurement might come from monetary expense of the power usage, which can easily reach into megawatts for the larger systems. Power might also come at a premium on a mobile radar platform such as an aircraft, which has only a limited power supply available to meet demands from a variety of sources. Stealth considerations can make the use of radar expensive, as often the measurer does not wish to be revealed to the target. The necessity of balancing a large number of tasks can give rise to a sort of measurement cost. A radar system might be forced to choose between monitoring an existing target or searching for a new target, with attention given to either task coming at the expense of the other.

Similarly, it might be forced to choose between transmission of two types of waveforms, each specialized for a specific task.

There are examples to be found in a variety of other fields as well. In chemical and manufacturing processes some measurements may be difficult to take, and the actual act of taking them may have an adverse effect on the processes themselves. In systems with distributed sensor networks, computing capacity may become a factor as the number of available measurements exceeds the computer's ability to process them. In space applications one can imagine that power comes at a premium, so that the power to run one sensing system comes at the expense of not using some other instrument. Communications bandwidth also can become a problem as a spacecraft reports its observations back to Earth.

In the literature for this problem (covered in detail in Section 1.1), there is a clear distinction between the papers produced by those in academia and those in industry. The academic papers generally formulate a problem in terms of costs and constraints, suggest a solution method appropriate to the particular formulation, and then attempt to make rigorous statements about the problem. They address issues such as optimality, stability, and convergence so that their results can be applied to an entire class of problems, not just a single application. Usually these papers work with scalar and/or linear systems, as they are more amenable to analysis. The focus is on the theory, not the practice, so applications (if they are given at all) are usually for small, tractable systems. Examples of this type of paper include [4], [30], [35], [41], and [51].

People in industry are usually more intent on getting practical results with real systems. Since many real-world problems are too complex to be treated rigorously with even the most modern theoretical techniques, their papers are ususally more concerned with heuristic rules and algorithms than mathematically correct problem statements and proofs. Only in these papers do we see extensive treatment of nonlinear systems. Often this type of paper will include a reference to a complex (perhaps proprietary) simulation used for Monte-Carlo analysis of algorithms. As these simulations are rarely available to anyone other than the paper's authors, critical evaluation of the results becomes difficult. By this we do not mean to imply any criticism; we are simply describing a well known reality. Examples of this type of paper include [15], [18], [20], [22], [46], and [66]. It is interesting to note that of the two papers by Mehra we reference, one fits the description of academic papers given above and was written while he was at Harvard [35]; the other fits the description of application-oriented papers and was written while he was in industry [34].

One of the reasons for the wide discrepancy between these two categories of papers is that the problem they all address is a difficult one. The academics tend to use a series of assumptions to reduce the problem to one that is simple enough that it can be dealt with analytically or by using a numerical technique like dynamic programming (DP), which we describe briefly in Section 1.2. The practitioners don't have the freedom to make such assumptions, so they tend to rely on less formal, more heuristic algorithms. Recent advances in the field of reinforcement learning (RL) have made it possible to break out of the limitations of traditional dynamic programming techniques and treat complex nonlinear problems in a more rigorous way. Section 1.3 provides a brief overview of RL and describes the features which allow it to handle problems DP cannot.

The contribution of this thesis will come in two areas. For the measurement with cost problem we will present a general formulation applicable to a wide variety of problems and a single, unified solution technique which we will apply to some real-world examples. For RL the contribution will be in its application to problem types that have not yet been closely examined. To see the value of these endeavors, we will begin by looking at what has been done in these areas to date.

## 1.1 Literature Review: Measurement Scheduling

Though the literature in the area of measurement scheduling is not large, it is quite diverse. One can find a wide range of cost formulations, constraints, solution methods, and applications with no overarching framework to connect one to another. In the paragraphs below we will take a quick look at each of these aspects of the measurement with cost problem.

### 1.1.1 Problem Formulation

Within the overall subject of measurement scheduling it is possible to formulate a wide variety of problems, and the literature reflects this. There are however, a few aspects of the papers in this area that seem to hold relatively constant. As mentioned above, the academic papers all deal with linear systems, while the industry papers consider nonlinear plants or measurements. Similarly, the vast majority of papers deal with finite horizon formulations, with the only treatment of infinite horizon or "steady state" problems coming in a brief paper by Feron [24] and the more applied algorithms of the industry papers [15], [22], [46], and [66]. We will examine both finite and infinite horizon formulations for both linear and nonlinear cases.

Regardless of whether a cost function is finite or infinite horizon, it can almost always be divided into two parts: one dealing with performance, the other with the control – in this case, the measurement commands. All the papers in the literature agree that the appropriate measure of performance for estimation problems is the error covariance, though exactly how this quantity is used varies from paper to paper. The most common choice is the use of a weighted sum of traces of error covariance matrices evaluated at a specific set of times. Variations on this cost formulation can be found in [4], [5], [6], [22], [28], [36], [41], and [51]. Mehra [35] and Blackman [15] base their costs on the inverse of the Fischer information matrix, a similar quantity. Tanaka [58] formulates the problem in terms of minimizing the maximum of the error variance over a finite horizon. Chen and Blair [20] use the error covariance as a constraint at the final timestep. In the papers concerned with simultaneous control and estimation, the performance measure is naturally based on the system's state, not on the estimate.

The use of the error covariance as a performance metric is somewhat problematic, however. Since we have no way of directly measuring this quantity, we will always be relying to a greater or lesser degree on guesswork to evaluate our current state and our performance. This problem is alleviated to some extent when we deal with linear systems, as we can propagate the error covariance from step to step using the Kalman Filter. This is probably one of the primary reasons why the academic literature deals with linear systems. One of the issues that we will address in this thesis is what to do when faced with a nonlinear system, for which this problem becomes more significant.

The portion of the cost relating to the measurement can be expressed directly or enter indirectly as a constraint. Such a constraint might be to fix the total number of measurements to be taken as in [30], [36], [50], [57], and [58], with the only degree of freedom coming in their timing. Another possible constraint seen in [5], [28], [24], [36], and [41] would be to allow only a single measurement at each step out of a discrete set of possible choices. In [6], [35], and [51], measurements are chosen from a continuum, the cost of a measurement is seen as proportional to its accuracy, and the constraint is phrased in terms of total cost rather than the number of measurements. Andersland [3] includes a cost for switching sensors on or off as well. Cooper and Nahi [21] and Olgac et al. [40] come closest to the measurement cost formulation which will be adopted in this thesis, assigning an explicit cost to each measurement and minimizing over a finite horizon without posing any additional constraints.

### 1.1.2 Solutions

The range of solutions presented in the papers we have mentioned is quite diverse, so we will not go over all of them here. Suffice it to say that analytical solutions are provided for only the most simple cases, with the vast majority relying on constrained optimization techniques. A few, [21], [36], and [40] use an approach based on dynamic programming. These latter papers all use simplifications of one sort or another to make the dynamic programming approach practical. Casta et al. [18] use reinforcement learning, the technique we will focus on in this thesis, for a sensor fusion type problem for military aircraft.

### 1.1.3 Applications

In the literature for this subject, the level of consideration given to applications varies widely from paper to paper. Some papers treat the problem in the context of combined estimation and control, others purely in terms of estimation.

Of the authors who look at the former problem, Aoki [4], Andersland [3], Olgac et al. [40], Cooper and Nahi [21], and Tanaka and Okita [57] make no reference at all to possible applications. They deal purely with abstract linear systems. Kushner [30] makes a vague mention of possible application to space navigation, Mellefont [37] makes a reference to applications in chemical processes, and Sawaragi et al. [50] talk about applications in pollution control. Only Meier [36] discusses possible areas of application and provides an actual example.

The range of detail is similar on the pure estimation side. Mehra [35] provides no discussion of possible applications. Tanaka [58] brings up Sawaragi's pollution control application, while Shakeri, et al. [51] note possible applications to active perception and Oshman [41] discusses applications in the area of large space structures. Athans [5] gives a wide range of possible applications and provides a radar-related numerical example.

Radar seems to get the most attention of all the applications mentioned in the literature. The most specific examples are also found in this area. In addition to Athans, Avitzour and Rogers [6], Blackman [15], Chen and Blair [20], Daum [22], Feron [24], Heffes [28], Meier et al. [36], and Van Keuk and Blackman [66] all discuss the problem of costly measurements with applications to radar tracking in varying levels of detail. Popoli [46] and Casta et al. [18] also discuss radar applications, but in the more general context of the sensor fusion problem.

Our examples will also come from the field of radar based surveillance and tracking. After working our way through a few simple examples, we will apply our RL-based algorithms to tracking of ballistic reentry vehicles using a complex nonlinear model based on the one developed by Mehra [34] but including more detail both in the dynamics and in the modeling of the measurement system.

## 1.2 Dynamic Programming

This section will provide only a brief overview of the concepts of dynamic programming. Readers wishing a more in-depth treatment of the subject should see [12]. Readers who are already aquainted with the subject may want to glance over this section to familiarize themselves with the notation that will be used in this thesis.

Dynamic programming embodies a large variety of problems, but in the discussion below we are primarily concerned with those relating to the optimal control of finite-state, finite-action Markov decision processes with known models. This implies the following assumptions:

- There are a finite number of states $x \in \mathcal{S} = \{0, 1, \ldots, N_s\}$, where $N_s < \infty$.
- Transitions between states depend on an action which comes from a finite set $a_k \in \mathcal{A} = \{\alpha_1, \ldots, \alpha_{N_a}\}$, where $N_a < \infty$ and obey the Markov property such that $p(x_{k+1} = j \mid x_k = i, x_{k-1}, \ldots, x_0, a_k) = p(x_{k+1} = j \mid x_k = i, a_k)$, where the subscripts on $x$ and $a$ indicate the time sequencing.
- The transition probabilities $P_{ij}(a_k) = p(x_{k+1} = j \mid x_k = i, a_k)$ are known.
- The transition costs $g(x_k, a_k, x_{k+1})$ are known.

Given all these constraints, dynamic programming can be used to find the best action at each step as a function of the state. This is expressed in terms of a policy $\pi$, with an associated cost function $J^\pi$. At each state the best action $a^*$ is given by the optimal policy $\pi^*$, for which

$$J^{\pi^*}(x) \leq J^\pi(x) \quad \forall x \in \mathcal{S}, \pi \in \Pi \tag{1.1}$$

where $\Pi$ is the set of possible policies. Thus

$$a^* = \pi^*(x). \tag{1.2}$$

The cost function (sometimes also referred to as a value function) $J^\pi$ is simply the expectation of the discounted sum of all the future transition costs under policy $\pi$,

$$J^{\pi}(x) = \mathrm{E}\left[\sum_{k=0}^{\infty} \gamma^{i} g(x_k, \pi(x_k), x_{k+1}) \mid x_0 = x\right] \tag{1.3}$$

where the discount factor $\gamma$ lies in the interval $[0, 1)$ and ensures that the sum remains finite. Equation (1.3) is what is known as an *infinite horizon* cost function. It can also be used to handle *finite horizon* cost functions such as

$$J^{\pi}(x) = \mathrm{E}\left[\sum_{k=0}^{N} g(x_k, \pi(x_k), x_{k+1}) \mid x_0 = x\right] \tag{1.4}$$

by adding an accepting state (by convention state 0) with $P_{00} = 1$ and $g(0, a, 0) = 0$ for all $a \in \mathcal{A}$ and setting $\gamma = 1$.[1]

Equation (1.3) can also be expressed in recursive form:

$$J^{\pi}(x) = \mathrm{E}\left[g(x_k, \pi(x_k), x_{k+1}) + \gamma J^{\pi}(x_{k+1}) \mid x_k = x\right]. \tag{1.5}$$

Replacing $\pi$ with $\pi^*$, we arrive at the Bellman equation,

$$J^{*}(x) = \min_{a \in \mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J^{*}(x_{k+1}) \mid x_k = x\right], \tag{1.6}$$

where $J^*$ is convenient shorthand for $J^{\pi^*}$. Dynamic programming provides an iterative method for solving the Bellman equation for $J^*$. Once we have $J^*$, we can easily solve for $\pi^*$ using

$$\pi^{*}(x) = \arg\min_{a \in \mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J^{*}(x_{k+1}) \mid x_k = x\right]. \tag{1.7}$$

Dynamic programming provides two basic methods for solving the Bellman equation: value iteration and policy iteration. Value iteration solves for the optimal cost function directly, starting with an arbitrary $J$ and updating $J(x)$ at all states simultaneously according

[1] We may also need to replace the state $x$ with a new state $x' = (x, t)$ so as to capture terminal costs.

to

$$J(x) \leftarrow \min_{a \in \mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J(x_{k+1}) \mid x_k = x\right]. \tag{1.8}$$

Under certain restrictions, this is guaranteed to converge to $J^*$, though it may take an infinite number of iterations to get there [12]. The optimal policy is extracted at the end using Equation (1.7).

Policy iteration, on the other hand, works in two steps. It starts with an arbitrary policy $\pi_0$. Then, in what is known as a policy evaluation step, the cost $J^{\pi_n}$ is calculated for each policy $\pi_n$ by solving Equation (1.5). Under the restrictions set forth above, this is a set of $N_s + 1$ linear equations in $N_s + 1$ unknowns. This can always be solved in finite time. Once $J^{\pi_n}$ has been calculated, a new policy can be constructed in a policy update step using a greedy update

$$\pi_{n+1}(x) = \arg\min_{a \in \mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J^{\pi_n}(x_{k+1}) \mid x_k = x\right] \tag{1.9}$$

for every state. Under the appropriate restrictions, policy iteration is guaranteed to converge to the optimal policy in a finite number of iterations [12].

In practice, however, neither value iteration nor policy iteration is often used. What is more common are variations of what Sutton calls Generalized Policy Iteration (GPI) [56] and Bertsekas and Tsitsiklis call asynchronous policy iteration [13]. The general idea is that instead of solving the linear equations for $J^{\pi_n}$ exactly, an $O(N^3)$ operation, some number of value iteration steps are used to approximate $J^{\pi_n}$, after which a new policy is calculated based on this approximate value. Neither the policy evaluation step nor the policy update step need be carried out to completion; eventual convergence is still guaranteed.

## 1.3 Reinforcement Learning

In some areas dynamic programming can be fairly useful, but the range of problems to which it can be applied is actually extremely limited. Three major problems make it hard to apply dynamic programming to real world problems. The first of these is the requirement that an accurate model of the system be available. The second is known as the "curse of dimensionality". Simply put, the calculations necessary to execute a dynamic programming algorithm quickly become prohibitively expensive as the size of the state space increases. Many problems with finite but large state spaces become infeasible, and direct application of DP to problems with continuous state or action spaces is impossible. The third major problem is the presence of the Markov assumption. Because of this, application of DP requires full state knowledge, which is rarely present in real-world situations.

The field of reinforcement learning contains methods for addressing all of these problems, though the literature relating to the third problem is still somewhat limited. For a survey of the field and history of its development, see [29]. For a good introduction to the concepts and methods with plenty of examples, see [56]. For a deeper treatment of the theoretical aspects of the subject, see [13].

In standard dynamic programming, a model is needed primarily to carry out greedy policy updates as per Equation (1.9). As long as some method is available for feeding back the transition costs, policy evaluation can be carried out without a model by using sample averages. Sample averaging will not work for the policy update step, however, as multiple courses of action must be compared.

Reinforcement learning algorithms bypass this problem by using what Sutton calls a state-action value function instead of a normal value function.[2] This function is designated with a $Q$ instead of a $J$, and hence is known as the Q-function. It is defined to be the quantity minimized on the right hand side of the Bellman equation, or

$$Q^{*}(x,a) = \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J^{*}(x_{k+1}) \mid x_k = x\right] \tag{1.10}$$

and can be thought of as the cost of taking action $a$ at step $k$ and then following the optimal policy thereafter. Similarly, $Q^{\pi}$ is defined as

$$Q^{\pi}(x,a) = \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J^{\pi}(x_{k+1}) \mid x_k = x\right], \tag{1.11}$$

[2] Though RL techniques bypass the need for a model in conducting greedy updates, some sort of simulation model is still needed in order to interactively generate experience.

the cost of taking action $a$ at step $k$ and then following policy $\pi$ thereafter. Note that

$$Q^{\pi}(x,\pi(x)) = J^{\pi}(x) \tag{1.12}$$

so that Equation (1.10) can also be expressed as

$$Q^{*}(x,a) = \mathrm{E}\left[g(x_k,a,x_{k+1}) + \gamma Q^{*}(x_{k+1},\pi^{*}(x_{k+1})) \mid x_k = x\right]. \tag{1.13}$$

Once the Q-function has been defined, an optimal policy can be defined as

$$\pi^{*}(x) = \arg\min_{a\in\mathcal{A}} Q^{*}(x,a), \tag{1.14}$$

and the policy update of Equation (1.9) can be simplified to

$$\pi_{n+1}(x) = \arg\min_{a\in\mathcal{A}} Q^{\pi_n}(x,a) \tag{1.15}$$

Thus, if the Q-function is known, policy updates can be completed without the use of a model. Incidentally, this allows us to write Bellman's equation in yet other forms:

$$\begin{aligned} J^{*}(x) &= \min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k,a,x_{k+1}) + \gamma J^{*}(x_{k+1}) \mid x_k = x\right] && (1.16)\\ &= \min_{a\in\mathcal{A}} Q^{*}(x,a) && (1.17)\\ &= \min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k,a,x_{k+1}) + \gamma Q^{*}(x_{k+1},\pi^{*}(x_{k+1})) \mid x_k = x\right] && (1.18)\\ &= \min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k,a,x_{k+1}) + \gamma \min_{\alpha\in\mathcal{A}} Q^{*}(x_{k+1},\alpha) \mid x_k = x\right]. && (1.19) \end{aligned}$$

Each of these variations of the Bellman equation corresponds to a different update technique. The differences between these updates will be discussed in detail in Chapter 2.

The second major problem with dynamic programming, the "curse of dimensionality", has several facets. When the size of the state space makes full updates impractical, other methods must be used to estimate the value function. Classically, Monte-Carlo simulation methods have filled this role. Unfortunately, these can be impractical even with relatively small state spaces, due to the necessity of completing an entire trajectory to get a single sample of the cost. In the field of reinforcement learning, however, the method of temporal differences (TD) has become the standard for estimating value functions. Like Monte-Carlo methods, TD algorithms use sample trajectories through the environment to determine the relative costs of states and actions. Unlike Monte-Carlo methods, however, TD methods feed back cost information incrementally, updating the approximate values of all the states encountered in a trajectory instead of just the first. A sort of secondary discount factor called an "eligibility trace", designated by the symbol $\lambda$ (as in TD($\lambda$)), ensures that each cost sample has little effect on the estimated cost of states which are far removed from it in time. The main references for TD methods are [23], [55], and [63].

Even if efficient use is made of experience by using temporal difference methods, some sort of compact representation for the value function is necessary. Reinforcement learning algorithms address this facet of the problem of the "curse of dimensionality" by using function approximation. The use of function approximators such as neural networks instead of DP's tables allows the compact storage of the information gathered about a system, and makes treatment of continuous state and action spaces possible. There are a wealth of options to choose from in this area, but the most common choice has long been the use of a global nonlinear function approximator such as the Multilayer Perceptron (MLP). The popularity of this function approximator comes partly from its familiarity as the "standard" in the field of neural networks (see [27]) and partly from the remarkable success of Tesauro's backgammon player [59] [60] [61]. MLP's using backpropagation learn relatively slowly, however, and in many applications can run into serious problems with local minima. As alternatives, people have turned to linear and localized function approximators.

By linear function approximators we mean approximators constructed from linear combinations of basis functions. The basis functions themselves need not be linear, only the manner of combining them to get an overall function. This allows quick training through linear least squares methods and avoids problems with local minima.

Local function approximation architectures allocate different amounts of resources to different regions of the state space. By putting more resources into areas where the function being approximated has greater complexity, these architectures in many cases use their resources (the architecture's free parameters) more efficiently than global architectures. The down side comes with the additional work required to distribute the approximation elements appropriately through a high dimensional state space. Most local function approximators (for example, Radial Basis Functions) are linear approximators with nonlinear basis functions. There are exceptions, however; [19] uses nonlinear combinations of linear basis functions.

Considerable attention has been given in the literature to the limitations of function approximation with regard to Reinforcement Learning. Further information can be found in [63] [62] [8] [68] [53].

The last major problem with dynamic programming has yet to be addressed adequately by the reinforcement learning field. Two critical assumptions underly most reinforcement learning algorithms. The first is that the state of the system in question evolves in time according to a Markov decision process (MDP). Essentially this means that the system's future behavior depends only on the present state; the past history need not be considered to find the optimal action. The second assumption is that the algorithm has full access to the state at each moment in time. This latter assumption is somewhat unrealistic in many situations, however, as an agent generally obtains information about its surroundings through noisy sensors. It is this consideration that has given rise to the study of Partially Observable Markov Decision Processes (POMDP's) in the RL community. For articles on POMDP's in the RL literature, see [32] [43] [52].

Whereas dynamic programming is a well developed field with many strong convergence results, reinforcement learning comes with fewer assurances. Convergence proofs have been given for policy evaluation using TD($\lambda$) [23] and for Q-Learning [65], but both of these assume finite state spaces. Further results have been published on policy evaluation and control using function approximation, and are summarized in Appendix B. These results are fairly limited, however. The strongest result for TD($\lambda$) using function approximation requires that the approximator be linear and the underlying Markov chain at most countably infinite. So far, convergence results for continuous state and action spaces are limited to a single special case. Most applications of RL techniques, including the ones we will present in this thesis, are to situations that do not meet the requirements for existing convergence proofs, and so no guarantees exist.

## 1.4 RL for measurement with cost problems

The problem of scheduling costly measurements for state estimation will test the limits of what can be achieved with this technique. We will not only be dealing with non-Markovian systems, but also surveying continuous, multi-dimensional state spaces and choosing from a potentially continuous range of possible actions. Moreover, we will have no direct performance measure for reinforcement. Successful application of RL to this problem will show that this is a technique which can be used with real-world problems. In the paragraphs below we give a short preview of the challenges we will be facing.

Both dynamic programming and reinforcement learning rely heavily on the use of the Markov property. Even research on partially observable Markov processes (POMDP's) relies on the existence of an underlying Markov chain. When we deal with nonlinear estimation problems, however, we are working with non-Markovian observation processes even if the state transition process is Markovian (i.e. the $y_k$'s are not Markovian even if the $x_k$'s are). At any given point the optimal estimate will depend on the entire past history. Since optimal nonlinear filters are infinite-dimensional, we will be forced to use suboptimal filters and base our decisions on estimated quantities. If our estimators perform well our overall system should be *almost* Markovian, but it remains to be seen if this will be good enough.

Instead of dealing with the standard small, finite state spaces, to handle this problem we will be forced to work with continuous, multi-dimensional state spaces. The state estimate's error covariance matrix, which we will be using as one of our primary inputs, has $O(n^2)$ independent elements. For a seven-dimensional state vector there are 35 continuously varying inputs needed to propagate an extended Kalman filter. Coping with state spaces of this size will be a non-trivial problem.

Most reinforcement learning problems in the literature allow only a few possible action choices at each step. We will examine this sort of problem as well, but we will also treat problems with continuous action spaces. Some of the issues relating to learning with continuous action spaces have been discussed in the literature, but relatively few examples of its actual use are available. We will be treating problems that not only have continuous action spaces, but multi-dimensional continuous state spaces and non-Markovian state propagation as well.

Finally, standard reinforcement learning problems often work without the benefit of a model, relying on some sort of physical feedback (or reinforcement) for learning. The estimation problems we examine, on the other hand, will rely extensively on models. We will use the filter's error covariance, a largely fictional quantity, for our performance metrics. Instead of getting feedback from the outside world, our cost measures will be internally generated and propagated from step to step.

Clearly we will be pushing the limits of what can be accomplished using current reinforcement learning techniques. In doing so we hope to generate insights that will be useful to others forced to deal with these difficult issues.

## 1.5 Organization

Chapter 2 will present a general formulation of the measurement with cost problem. This will provide a framework able to accommodate the wide variety of problems which have been examined in the literature along with other, as yet untreated variations. The problem will be posed in such a way as to allow easy application of reinforcement learning.

This chapter will also include a description of the algorithms that will be used later in the thesis. It will address issues associated with function approximation and with various update methods. A distinction will be drawn between simple and complex action spaces, and methods for dealing with each will be discussed.

Chapter 3 will continue the algorithmic discussion by addressing issues related to learning when cost variations with respect to the action are small compared to those in the state direction. A class of transformations to the Bellman equation capable of accentuating small differences in costs will be described, and new methods for applying reinforcement learning to the resulting functions will be presented. Baird's advantage learning algorithm will be discussed in this context. Previously unreported limitations to the algorithm will be described along with an example which shows how these limitations can cause the algorithm to diverge. An alternative algorithm called G-Learning, which corrects this defect for the example problem, will be presented.

Application of the general formulation and algorithms described in Chapter 2 will begin in Chapter 4. This chapter will treat four linear problems, two of which come from the literature. In describing the application of reinforcement learning to these problems a large amount of effort will be spent in showing the methodology involved. Detailed lists of the specific parameters used will be included as well as the reasons for which they were chosen. The intent is to showcase the wide variety of problem formulations which can be tackled using reinforcement learning, but also to describe the design decisions which must be made in order to implement these algorithms successfully.

Chapter 5 will up the bar by examining a simple nonlinear problem. This is intended as preparation for the more complex application which will be addressed in Chapter 6. By looking at a problem which is simple enough to allow some basic analysis to be applied, we can compare learned policies with an intelligently designed heuristic. This should give us an idea of whether our algorithms will work in this new domain. Both discrete (binary) and continuous action spaces will be examined.

The culmination of our efforts will come in Chapter 6. Here we will develop a complex nonlinear model for a radar tracking system monitoring intermediate-range ballistic missile reentry. We will then see whether reinforcement learning can yield concrete improvements over heuristic policies in real-world situations. Again, both discrete and continuous action spaces will be examined.

We will conclude with Chapter 7. This will include a discussion of the results from the previous chapters. Both successes and failures will be examined for insights that may help in future work. Conclusions will be drawn with regard to the utility of reinforcement learning for complex problems, and possibilities for future work will be discussed.

# Chapter 2

# Problem Formulation and Solution Approach

As we have already seen from the literature, there are a variety of different ways in which the measurement with cost problem can be formulated. Similarly, there are a variety of different ways in which its solution can be sought. In this thesis, we would like to show that a wide spectrum of these problems can be handled by a single general formulation, and that all of them may be solved using one basic approach. The unified formulation will be presented in terms of functional blocks and their connections. Once this has been done, the generation of each specific formulation merely becomes a matter of filling in the contents of the blocks. In later sections we will go through this process of filling in the blocks for several linear examples, two of which come from the literature. Later, we will fill in the blocks for nonlinear examples, showing the versatility of the formulation.

## 2.1 Formulation

The problem with which we are faced is the estimation of some function of the state of an evolving physical system. We assume that the system can be modeled by a stochastic differential equation of the form

$$\dot{x}(t) = f(x(t), u(t), w(t), t) \tag{2.1}$$

where $x(t)$ is the state, $u(t)$ is a (known) control, $w(t)$ is a stochastic signal representing the process noise, and $t$ is time. This is the first of our blocks. In some cases, it may be more advantageous to think in terms of discrete time state propagation. In these instances, (2.1) will be replaced with the difference equations

$$x_{k+1} = f_d(x_k, u_k, w_k, t_k) \tag{2.2}$$

where the subscript $d$ stands for discrete.

Regardless of the manner in which the state is propagated, at each timestep $t_k$ we are able to take a measurement $y_k$ of some aspect of the system. Here we have some freedom of choice: we have a set $\mathcal{A}_k$ of possible actions that we may take, and the form and accuracy of each measurement depends on the particular action $a_k \in \mathcal{A}_k$ that we choose.

$$y_k = h(x_k, u_k, a_k, v_k, t_k) \tag{2.3}$$

where $x_k = x(t_k)$, etc., and the statistics of the sensor noise $v_k$ are a function of the action chosen.

At this point it may help to give an example of what might actually go into these blocks. Equations (2.1) and (2.3) will simplify substantially for most problems of interest; in the linear case, for instance, they become

$$f_d(x_k, u_k, w_k, t_k) = Ax_k + Bu_k + w_k \tag{2.4}$$

and

$$h(x_k, u_k, a_k, v_k, t_k) = Cx_k + Du_k + v_k. \tag{2.5}$$

In order to estimate the (unknown) state of the system, we maintain a filter which has its own (known) state. Combined with the known external input, the actions, and the measurements themselves, the filter state – which we shall designate $\zeta_k$ – contains all the information necessary to propagate estimates from timestep to timestep. We will distinguish between the state of the filter before and after the information from $y_k$ is incorporated by using $\zeta_k^-$ and $\zeta_k^+$ respectively[1]. The filter state will typically include the time, the state estimates themselves, and some measure of the current estimate's error statistics.

The filter itself then comes in the form of two blocks. The first specifies how each measurement is incorporated into the estimate,

$$\zeta_k^+ = \Phi\left(\zeta_k^-, y_k, u_k, a_k\right), \tag{2.6}$$

and the second specifies the manner in which the estimate is propagated from timestep to timestep

$$\zeta_{k+1}^- = \Psi\left(\zeta_k^+, u_k\right). \tag{2.7}$$

The actions at each timestep are chosen according to a policy $\pi$ such that

$$a_k = \pi\left(\theta_k\right) \tag{2.8}$$

where $\theta_k$ is a feature vector derived from the filter state, $\zeta_k^-$, and the currently known portion of the control history, $\varsigma_k$:

$$\theta_k = \Gamma(\zeta_k^-, \varsigma_k). \tag{2.9}$$

In many cases, $\theta_k$ will simply be a subvector of $\zeta_k^-$. The second argument of $\Gamma$ may be a bit more complex. It's character depends on the level of knowledge available about the external input $u_k$. At the very least we assume that the current input $u_k$ is known, and therefore all past inputs $u_0 \ldots u_{k-1}$ are also known. In some cases, however, future values of the input may also be known. Since this knowledge could have an effect on the measurement choice at the current time, it must be included in $\varsigma_k$.

[1] Superscripted + and - symbols will be used with this same meaning for a variety of variables throughout the thesis.

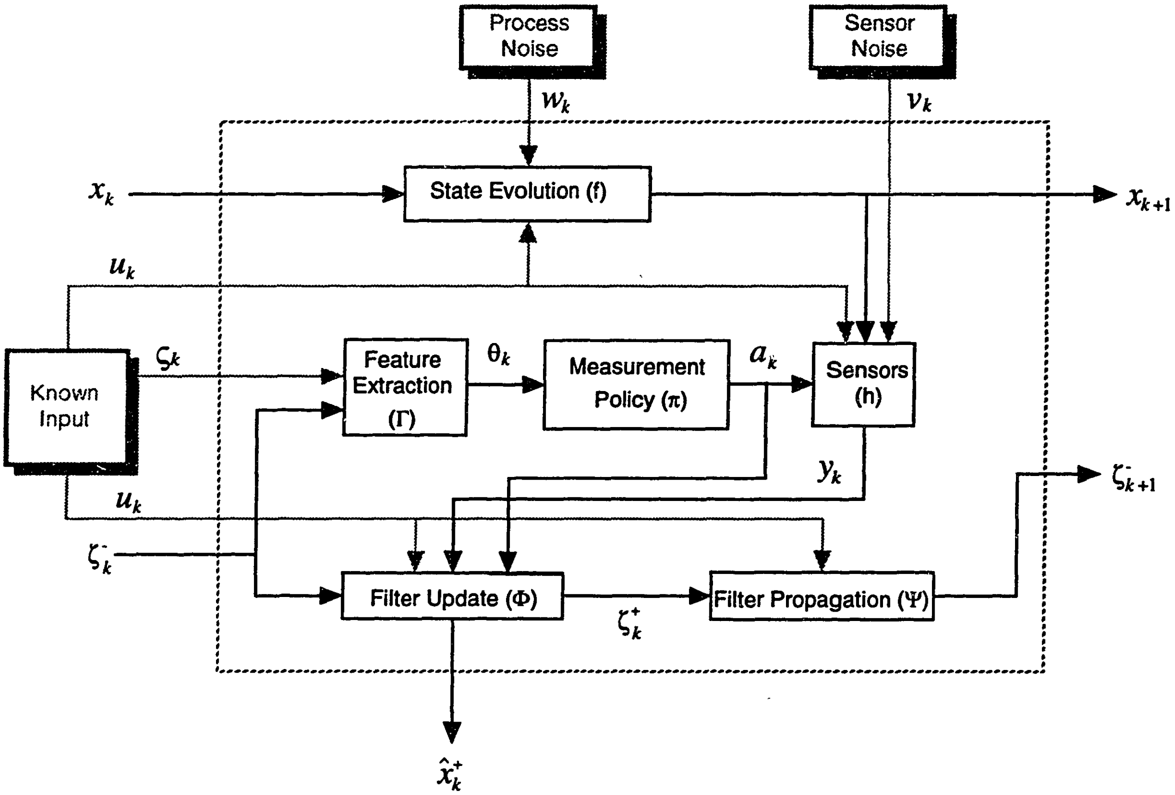


Figure 2-1: Block diagram for combined plant/filter

The combination of plant, policy, and filter is shown diagramatically in Figure 2-1. At each timestep, performance is evaluated as a function of the filter state, action, and measurement:

$$c_k = g\left(\zeta_k^-, a_k, y_k\right). \tag{2.10}$$

The estimation error covariance often enters into the cost through the filter state. Overall performance of the policy $\pi$ over the interval $[k, k+N]$ is evaluated through a cost function

$$J^\pi\left(\zeta_k^-\right) = \mathrm{E}\left[\sum_{i=0}^{N} \gamma^i c_{k+i}\right] \tag{2.11}$$

where we take the limit as $N \to \infty$ for the discounted infinite horizon case and set $\gamma = 1$ for the finite horizon case. The expectation is taken with regard to the underlying plant, measurement, and filter models described in Equations (2.1)–(2.7). The problem, then, is to find a policy which minimizes the expected cost over the chosen horizon.

## 2.2 Approach

The main benefit derived from combining the various possible measurement with cost problems into a single general formulation is that it allows us to use the same basic tools to solve all of them. Reinforcement learning provides a single unified approach to the solution of the problems posed above, though each individual implementation may require its own slightly different algorithm.

In this section we begin by presenting the algorithmic framework that makes up this unified approach. Then, just as the previous section described the blocks which make up the problem formulation, this section will describe the blocks which will be used to seek a near-optimal solution.

### 2.2.1 Algorithmic Framework

At its heart, a reinforcement learning algorithm is concerned with two functions: a value function $J$ or state-action value function $Q$ (in this thesis, we will only consider the latter), and a policy $\pi$. The algorithm begins with some initial policy and some estimate of $Q$ and tries to find the optimal functions $Q^*$ and $\pi^*$ by interacting with the environment. The details of the algorithm lie in how that interaction takes place and how the estimates of $Q$ and $\pi$ are updated.

When the system being examined is a small finite-state Markov decision problem (MDP), we can interact with the environment (or a model of the environment) in two fundamentally different ways. One way is to generate trajectories through simulation or experiments, where states are encountered according to the underlying probabilities of the MDP. An alternative method is to simply list the states and test each one, either sequentially or according to some predetermined pattern. The problem with the former method is that some states may be encountered only rarely, slowing convergence dramatically. The latter method can be much quicker, but has been shown to admit the possibility of divergence when combined with function approximation [63].

In systems with low-dimensional, continuous states, it may be possible to distribute tests over the entire state space by gridding it off. However, as the dimension of the state increases this grid grows exponentially in size, so that uniformly distributed interaction with the environment becomes impractical. We are left with trajectory-based interaction as our only reasonable alternative. This type of interaction will be used throughout the thesis, except for one example where a grid-based interaction is used on a simple system for illustrative purposes. Of course, with the decision to use a trajectory-based algorithm come a host of issues relating to the exploration vs. exploitation trade-off, but these will be addressed in a later section.

The rest of the algorithm's details revolve around updating of the Q-function and policy. We must specify how the cost estimates are generated within each trajectory, how the Q-function is updated based on the cost estimates, and how the policy is updated based on the revised Q-function. Each of these steps involves a number of trade-offs and design decisions.

The overall algorithm used throughout this thesis is outlined in Figure 2-2. Before entering the main learning loop, various algorithm parameters must be set, function approximator weights must be initialized, and some preliminary training may be conducted based on an initial policy guess or other available background information. This is what occurs in the "Initialize Algorithm" block. Once the algorithm has been initialized, the main loop is entered and a series of trajectories are generated either through simulation or direct interaction with the environment. Each trajectory uses a learning technique, either SARSA($\lambda$) or Q-learning, to generate a set of estimates $\{\hat{Q}\}$ of the Q-function. This is the "Run Trajectory" block. The $\hat{Q}$'s are then used to update the parameters of the Q-function approximator $\tilde{Q}(\theta, a, r)$ ($r$ is the approximator's weight vector), and the updated $\tilde{Q}$ is in turn used to update the policy $\pi$. After adjusting learning and exploration rates as appropriate, the next trajectory is run as the cycle continues. There are a variety of ways in which the algorithm can be terminated, but we have chosen merely to stop after a fixed number of iterations, $N_{eg}$, have been completed. The "eg" stands for "epsilon-greedy".

It is important to understand that the algorithm framework shown in Figure 2-2 is but one of several possibilities. The updating of $\tilde{Q}$ and $\pi$ need not be relegated to separate blocks reached only upon conclusion of a complete trajectory. Updates for both of these can be conducted in an on-line, incremental fashion as well. The frequency with which updates are conducted is an important design parameter available to the user, and different choices can produce dramatically different results. In this thesis we have chosen to conduct our updates for both policy and Q-function in batches after each trajectory for the following reasons:

- Separating out the updates increases modularity and decreases the overall complexity of the algorithm.

- Some function approximators converge relatively slowly. This means that a large number of target points must be presented, which is often more easily achieved by presenting the same data points multiple times than by generating large numbers of new points. This favors batch training operations.

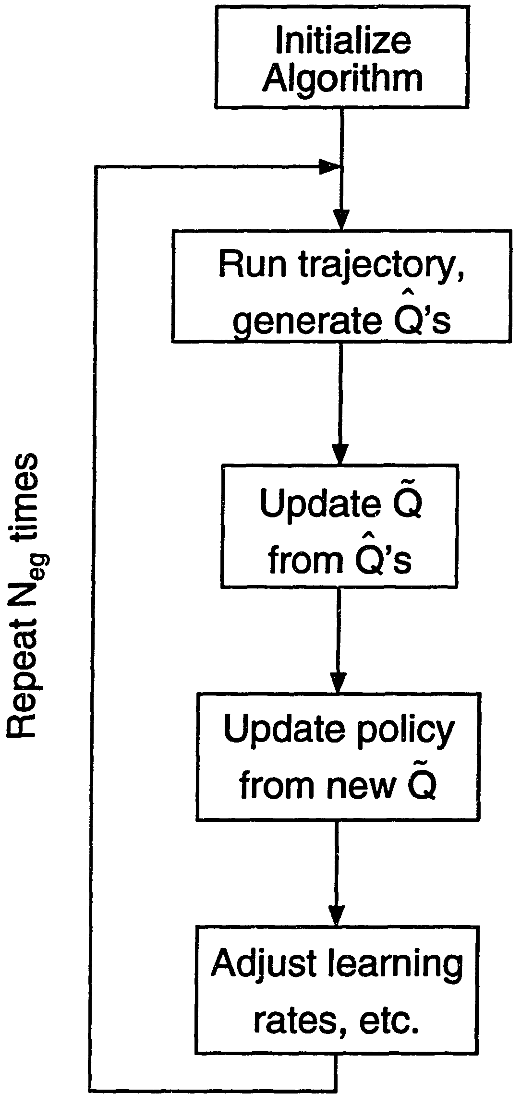

Initialize Algorithm
Run trajectory, generate $\hat{Q}$'s
Update $\tilde{Q}$ from $\hat{Q}$'s
Update policy from new $\tilde{Q}$
Adjust learning rates, etc.
Repeat $N_{eg}$ times

Figure 2-2: Flowchart of basic learning algorithm.

- Overly frequent policy updating has been shown to cause convergence to non-optimal policies under certain conditions [13].
- A single framework that addresses both continuous and discrete action spaces is desired, and frequent updating is computationally impractical for systems with continuous action spaces. The reasons for this will be explained in more detail below.

### 2.2.2 Trajectory generation

Logically, it would make sense to begin our discussion of the algorithm blocks with the algorithm initialization. However, there are elements of the initialization which will make more sense after the rest of the algorithm has been described in detail. Thus, the initialization phase will be described last in Section 2.2.7, not first, and we instead begin with trajectory generation.

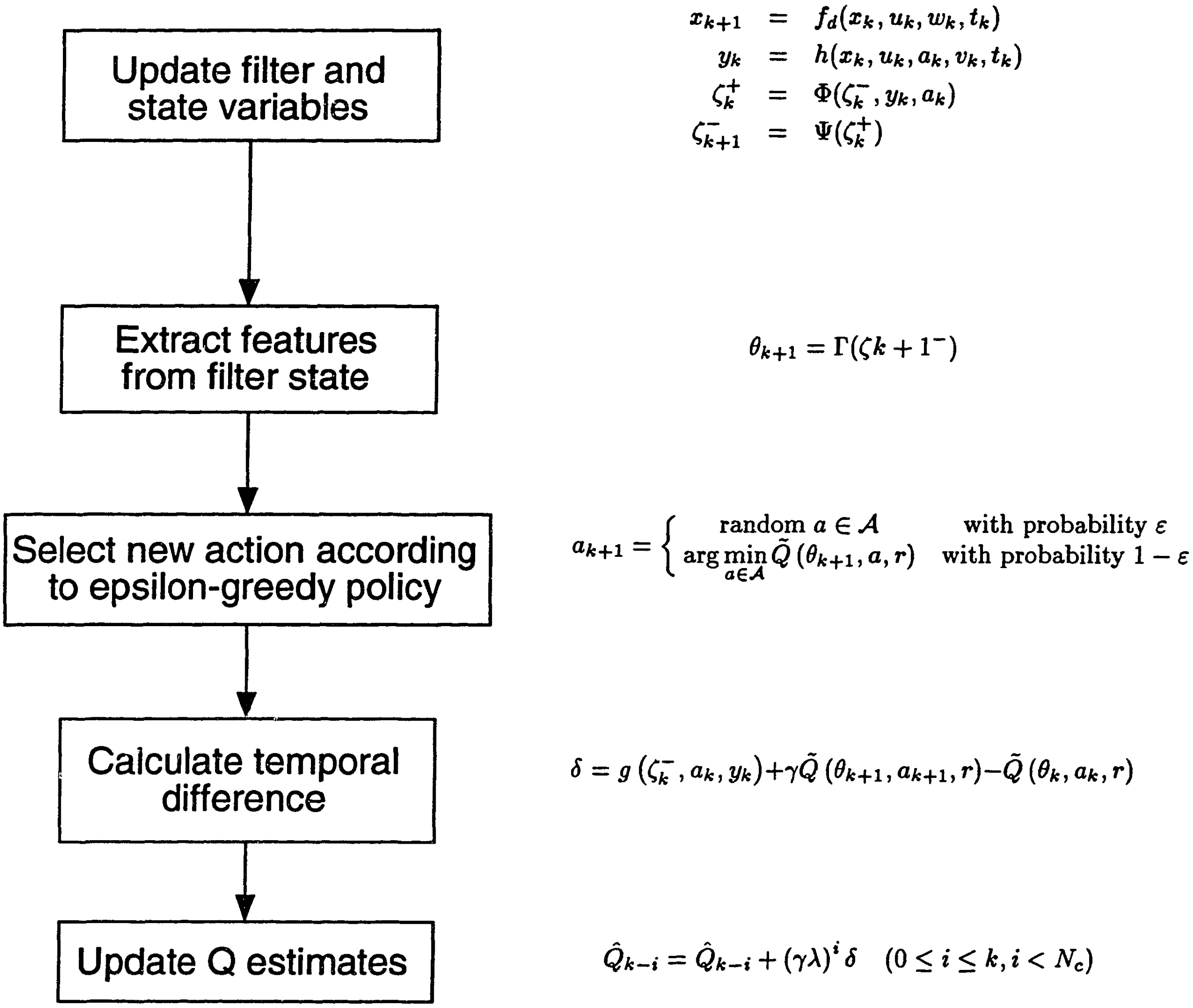
There are two major classes of reinforcement learning algorithms: those based on the concepts of temporal differences (represented by the SARSA($\lambda$) algorithm in this thesis), and those based on the concepts of Q-Learning. Both types of algorithm generate the actual trajectories in the same way; they differ in how they take statistics to generate the $\hat{Q}$ samples which will be used to train the $\tilde{Q}$ approximator. In their standard form, both methods generate trajectories by following the current policy with occasional random actions added for exploratory purposes. But while SARSA($\lambda$) algorithms take their statistics based on the actions actually chosen after the random element is added, Q-Learning algorithms take their statistics based on the estimate of what would have been the best action to take, regardless of whether that action was actually taken or not. So the SARSA($\lambda$) algorithms are training $\tilde{Q}$ to match the best estimate of the cost of the current randomized policy $Q^{\pi_\epsilon}$, while the Q-Learning algorithms are training $\tilde{Q}$ to match the best estimate of the optimal policy $Q^*$. The differences between the two algorithms and how they are applied to our particular problem are explained in more detail in the following two sections.

### 2.2.3 SARSA($\lambda$)

Figure 2-3 shows how a trajectory is generated using a SARSA($\lambda$) learning algorithm in the context of the formulation presented in the previous section. Equations showing the way the relevant quantities are calculated are given to the right of each block. At the beginning of each trajectory (of $N$ steps) the state and filter are reinitialized; the exact initial conditions may vary or remain constant. Then, at each step in the trajectory the filter is updated in accordance with the current action and then propagated to the next step. The feature vector is then calculated and the next action chosen.

The way in which this action is chosen is important. Throughout the learning process, the algorithm has access to the current policy $\pi$. This policy represents the current best guess at the optimal policy $\pi^*$. When actions are chosen, however, this policy is modified by introducing a random factor to encourage exploration of the state space. With probability $\epsilon$ the action is chosen randomly, and with probability $1 - \epsilon$ the action is chosen according to $\pi$. If $\pi$ is greedy, this modified policy is known as an $\epsilon$-greedy policy. Since we will be allowing approximated policies and partial policy updates in our algorithms, our policies will not necessarily be greedy, and so we will refer to the modified policy as $\pi_\epsilon$ instead.

It is after the action is chosen that we see the two distinguishing features of a SARSA($\lambda$) learning algorithm. First, it is an *on-policy* algorithm. By this we mean that instead of attempting to learn the value of following $\pi$, the current best guess at $\pi^*$, the algorithm attempts to learn the value of $\pi_\epsilon$. The $\hat{Q}$'s that are generated are estimates of $Q^{\pi_\epsilon}$, not $Q^{\pi}$. It is the estimation of $Q^{\pi_\epsilon}$ that allows the effective use of eligibility traces, the second distinguishing feature of SARSA($\lambda$) algorithms. The eligibility traces propagate cost information back through time (with a horizon determined by the parameter $\lambda$) in order to improve the estimates made at earlier points in the trajectory. Each temporal difference contributes a factor of $\delta$ to the calculation of the $\hat{Q}$ for the current state-action pair and a factor of $(\gamma\lambda)^i\,\delta$ to the calculation of the $\hat{Q}$ for the $i$th previous state-action pair. Information can be propagated backward in this way because the cost incurred at step $k$ always contains information about the value of $Q^{\pi_\epsilon}(\theta_{k-i}, a_{k-i})$, but if any random actions have been taken between steps


Update filter and state variables
Extract features from filter state
Select new action according to epsilon-greedy policy
Calculate temporal difference
Update Q estimates

$$\begin{aligned} x_{k+1} &= f_d(x_k, u_k, w_k, t_k) \\ y_k &= h(x_k, u_k, a_k, v_k, t_k) \\ \zeta_k^+ &= \Phi(\zeta_k^-, y_k, a_k) \\ \zeta_{k+1}^- &= \Psi(\zeta_k^+) \end{aligned}$$

$$\theta_{k+1} = \Gamma(\zeta k + 1^-)$$

$$a_{k+1} = \begin{cases} \text{random } a \in \mathcal{A} & \text{with probability } \varepsilon \\ \arg\min_{a \in \mathcal{A}} \tilde{Q}\left(\theta_{k+1}, a, r\right) & \text{with probability } 1 - \varepsilon \end{cases}$$

$$\delta = g\left(\zeta_k^-, a_k, y_k\right) + \gamma \tilde{Q}\left(\theta_{k+1}, a_{k+1}, r\right) - \tilde{Q}\left(\theta_k, a_k, r\right)$$

$$\hat{Q}_{k-i} = \hat{Q}_{k-i} + (\gamma\lambda)^i \delta \quad (0 \le i \le k, i < N_c)$$

Figure 2-3: Flowchart showing one step in a SARSA($\lambda$) trajectory. Expands the "Run trajectory, generate $\hat{Q}$'s" block from Figure 2-2.

$k-i$ and $k$ it contains no information about the value of $Q^{\pi}(\theta_{k-i}, a_{k-i})$.

The algorithm employed here differs slightly from the one described by Sutton and Barto [56] in that the Q-function is only updated at the end of each trajectory. Thus, in their terminology, we are using an off-line version of the algorithm.

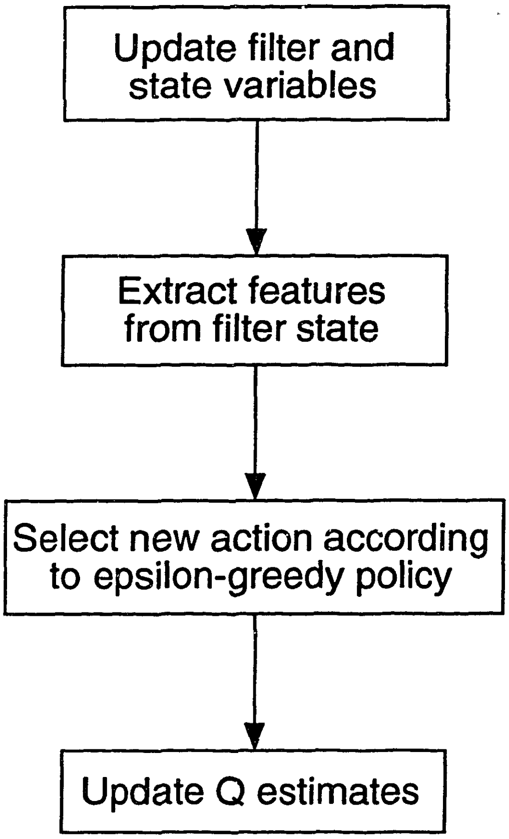
Convergence to the optimal policy comes through the process of generalized policy iteration. After working toward an estimate of $Q^{\pi_\epsilon}$ a policy improvement step is carried out and a new policy $\pi$ generated. This new policy is then evaluated, and another policy improvement step is carried out. If $\epsilon$ is gradually decreased throughout (in our algorithm, this takes place in the final block of Figure 2-2) this back and forth process should eventually converge to the optimal policy[2].

### 2.2.4 Q-learning

Figure 2-4 shows how a trajectory is generated using a Q-Learning algorithm. Again, the equations relating to each step are given at the right. The algorithm is identical to SARSA($\lambda$) until the action is chosen for the next step. At this stage, the Q-Learning algorithm calculates $\hat{Q}$ based on the cost of choosing the action according to $\pi$, even though $\pi_\epsilon$ may have generated a different action. This is why Q-Learning algorithms are classified as *off-policy.* Because $\pi$ and $\pi_\epsilon$ are likely to generate different actions at some point in the trajectory, eligibility traces cannot be used effectively to propagate cost information back and improve earlier $\hat{Q}$'s. Some researchers have experimented with $Q(\lambda)$ algorithms [67, 44, 48] where information is propagated back until $\pi$ and $\pi_\epsilon$ diverge, but such algorithms are not treated here.

Q-Learning algorithms work directly toward approximation of $Q^*$ without trying to approximate intermediate $Q^{\pi}$'s. For this reason, it is often said that Q-Learning algorithms are related to value iteration in dynamic programming, while SARSA($\lambda$) type algorithms are related to policy iteration.

[2] Convergence to the *optimal* policy is only actually guaranteed under fairly strict conditions. In most real-world situations, the best we can hope for is that we will get convergence to the policy which is as close to optimal as possible given the particular constraints we have put on the learning process.


Update filter and state variables
Extract features from filter state
Select new action according to epsilon-greedy policy
Update Q estimates

$$
\begin{aligned}
x_{k+1} &= f_d(x_k, u_k, w_k, t_k) \\
y_k &= h(x_k, u_k, a_k, v_k, t_k) \\
\zeta_k^+ &= \Phi(\zeta_k^-, y_k, a_k) \\
\zeta_{k+1}^- &= \Psi(\zeta_k^+)
\end{aligned}
$$

$$\theta_{k+1} = \Gamma(\zeta_{k+1}^-)$$

$$a_{k+1} = \begin{cases} \text{random } a \in \mathcal{A} & \text{with probability } \varepsilon \\ \arg\min_{a \in \mathcal{A}} \tilde{Q}\,(\theta_{k+1}, a, r) & \text{with probability } 1 - \varepsilon \end{cases}$$

$$\hat{Q}_k = g\,(\zeta_k^-, a_k, y_k) + \gamma \min_{a \in \mathcal{A}} \tilde{Q}\,(\theta_{k+1}, a, r)$$

Figure 2-4: Flowchart showing one step in a Q-Learning trajectory. Expands the "Run trajectory, generate $\hat{Q}$'s" block from Figure 2-2.

### 2.2.5 Updating $\tilde{Q}$, the Q-function approximator

This stage in the algorithm is relatively straightforward. The $\left(\theta, a, \hat{Q}\right)$ triples generated in the "Run Trajectory" block are used to train the function approximator, $\tilde{Q}\left(\theta_k, a_k, r\right)$, and produce a new weight vector $r$. The exact training method used will depend on the type of function approximator used, but will usually be some form of stochastic approximation. Each triple may be presented once or multiple times; again, the exact method used will depend on trade-offs conducted for the particular case at hand.

### 2.2.6 Updating the policy

In the policy update step, the current policy $\pi$ is replaced by a new policy that is greedy with respect to the new value function estimate generated in the previous step. In carrying out this update, decisions must be made about both the form of the update and the manner in which the update is to be completed. In the paragraphs below, we will discuss in general terms what we mean by this, and in Sections 2.2.6.1 and 2.2.6.2 we will give a more detailed description of the options and trade-offs involved.

The whole point of a reinforcement learning algorithm is to find the optimal policy $\pi^*$ by solving the Bellman equation

$$J^*\left(x_k\right) = \min_{a \in \mathcal{A}} Q^*\left(x_k, a\right). \tag{2.12}$$

Once we have the optimal cost, the optimal policy can be found by taking the action which is greedy with respect to $Q^*$:

$$\pi^*\left(x_k\right) = \arg\min_{a \in \mathcal{A}} Q^*\left(x_k, a\right). \tag{2.13}$$

There are alternate ways of expressing the right hand side of (2.13):

$$\pi^*\left(x\right) \quad = \quad \arg\min_{a \in \mathcal{A}} Q^*\left(x_k, a\right) \tag{2.14}$$

$$\begin{aligned} &= \arg\min_{a \in A} \mathrm{E}\left[g\left(x, a, x_{k+1}\right) + \gamma J^{*}\left(x_{k+1}\right)\right] && (2.15) \\ &= \arg\min_{a \in \mathcal{A}} \mathrm{E}\left[g\left(x_k, a, x_{k+1}\right) + \gamma \min_{\alpha \in \mathcal{A}} Q^{*}\left(x_{k+1}, \alpha\right)\right] && (2.16) \\ &= \arg\min_{a \in \mathcal{A}} \mathrm{E}\left[g\left(x_k, a, x_{k+1}\right) + \gamma Q^{*}\left(x_{k+1}, \pi\left(\zeta_{k+1}\right)\right)\right]. && (2.17) \end{aligned}$$

Each of these is equivalent when we are dealing with the optimal costs and policy. When we conduct a policy update in a reinforcement learning algorithm, however, we are *not* dealing with the optimal values of $Q$, $J$, and $\pi$. In this case the four greedy updates of Equations (2.14)-(2.17) are not the same. Each implies different decisions about model usage, cost sampling, and approximation of the value function. In choosing an update, we must consider

- The need for a full model. We will need some sort of black box capable of emulating the system in any case, but if any of the updates other than Equation (2.14) is to be used, we must be able to generate "what if" scenarios as well.

- Proximity of the approximation. The use of a function approximator will introduce inaccuracies to the system. These effects can be mitigated by including samples as in Equations (2.15)-(2.17).

- Sampling effects. While including sample values in the update equations can reduce the errors introduced due to function approximation, it can also introduce new errors due to the subsitution of sample values for expectations.

- Computational efficiency. All of the above considerations must be balanced with computational efficiency. The relative expense of evaluations of the function approximator and calculation of state propagations must be considered, along with the number of each required to produce $\hat{Q}$ or $\hat{\pi}$ samples.

The equation selected here determines the *form* of the policy update.

When we discuss the *manner* in which the update is to be conducted, we must begin to make distinctions based on the nature of the system's action space. There are a number of easy ways in which we could distinguish one action space from another: large or small, finite or infinite, continuous or discrete. In this thesis, however, we will distinguish between action spaces based on a characteristic that is somewhat less obvious. We will divide our attention between *simple* action spaces, for which minimization is computationally inexpensive, and *complex* action spaces, for which minimization is computationally expensive.

We make this distinction for the following reasons. If minimization over the action space is computationally inexpensive, we can define the policy implicitly in terms of the current value function estimate. This allows us to dispense with a separate policy update step, intertwining action selection with updating of the value function. If minimization over the action space is computationally expensive, however, we cannot afford to do it at every step. Instead, we must explicitly define our policy in terms of a function approximator, which we then update at intervals determined by our computational capability. The manner in which these updates are conducted is of prime importance.

In the following two sections, we will discuss options for policy updates of systems with simple and complex action spaces. Some of the distinctions are rather subtle; the "best" options may not always be clear, and will vary depending on the specifics of the system in question. A few of the options presented below will be compared in later chapters, but the number of possible variations is far too large to make much in the way of definitive statements.

### 2.2.6.1 Simple action spaces

For our purposes *an action space is* simple *if a policy defined implictly in terms of the cost function can be evaluated with a relatively minor computational effort.* In other words, if we define our policy as[3]

$$\pi\left(\theta_k\right) = \arg\min_{a\in\mathcal{A}} \tilde{Q}\left(\theta_k, a, r\right) \tag{2.18}$$

we must be able to carry out the requisite minimization of $\tilde{Q}$ quickly. Normally, this will mean that $\mathcal{A}$ is a small, finite set. In some cases, however, a continuous $\mathcal{A}$ could qualify as simple. For example, if it is known that for any given $\theta$, $\tilde{Q}\left(\theta, a\right)$ describes a parabola, the

[3] Remember that $\theta_k$ is the feature vector and a function of $\zeta_k$ and $\varsigma_k$.

| Update Method | | Fig | Evals | Props | Samples |
|---|---|---|---|---|---|
| Ideal | $\pi^*(x) = \arg\min_{a\in\mathcal{A}} Q^*(x,a)$ | (a) | $N$ | 1 | 1 |
| Practical | $\pi(\theta_k) = \arg\min_{a\in\mathcal{A}} \tilde{Q}(\theta_k, a, r)$ | | | | |
| Ideal | $\pi^*(x) = \arg\min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J^*(x_{k+1}) \mid x_k = x\right]$ | (b) | $N$ | $N$ | 1 |
| Practical | $\pi(\zeta_k^-) = \arg\min_{a\in\mathcal{A}} \left(g(\zeta_k^-, a, y_k) + \gamma\tilde{J}(\theta_k, r)\right)$ | | | | |
| Ideal | $\pi^*(x) = \arg\min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma \min_{\alpha\in\mathcal{A}} Q^*(x_{k+1}, \alpha) \mid x_k = x\right]$ | (c) | $N^2$ | $N$ | $1 + (N-1)$ |
| Practical | $\pi(\zeta_k^-) = \arg\min_{a\in\mathcal{A}} \left(g(\zeta_k^-, a, y_k) + \gamma \min_{\alpha\in\mathcal{A}} \tilde{Q}(\theta_k, \alpha, r)\right)$ | | | | |
| Ideal | $\pi^*(x) = \arg\min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma Q^*(x_{k+1}, \pi(x_{k+1})) \mid x_k = x\right]$ | (d) | $N + N$ | $N$ | $1 + (N-1)$ |
| Practical | $\pi(\zeta_k^-) = \arg\min_{a\in\mathcal{A}} \left(g(\zeta_k^-, a, y_k) + \gamma\tilde{Q}(\theta_{k+1}, \tilde{\pi}(\theta_{k+1}, r), r)\right)$ | | | | |

Table 2.1: Comparison of alternate greedy updates for simple action spaces. Note that for the non-ideal updates the arguments of the policy change depending on whether a single-step or two-step update is used.

minimum value of $a$ could be calculated from three samples with very little computational effort.

Since the policy is specified implicitly for a simple action space, the problem of updating the policy becomes intimately intertwined with the problem of updating the Q-function. With each action that is selected one or more samples of the Q-function are generated as a byproduct. Table 2.1 shows how the four options for greedy updates described in Equation (2.14) compare in computational efficiency. The updates are also compared graphically in Figure 2-5. The leftmost column of the table shows the ideal form of the quantity to be minimized and the practical reality using function approximation. The third column relates the update to the figure, and the remaining columns relate to the computational efficiency of the various update forms. They will be explained in more detail in the paragraphs below.

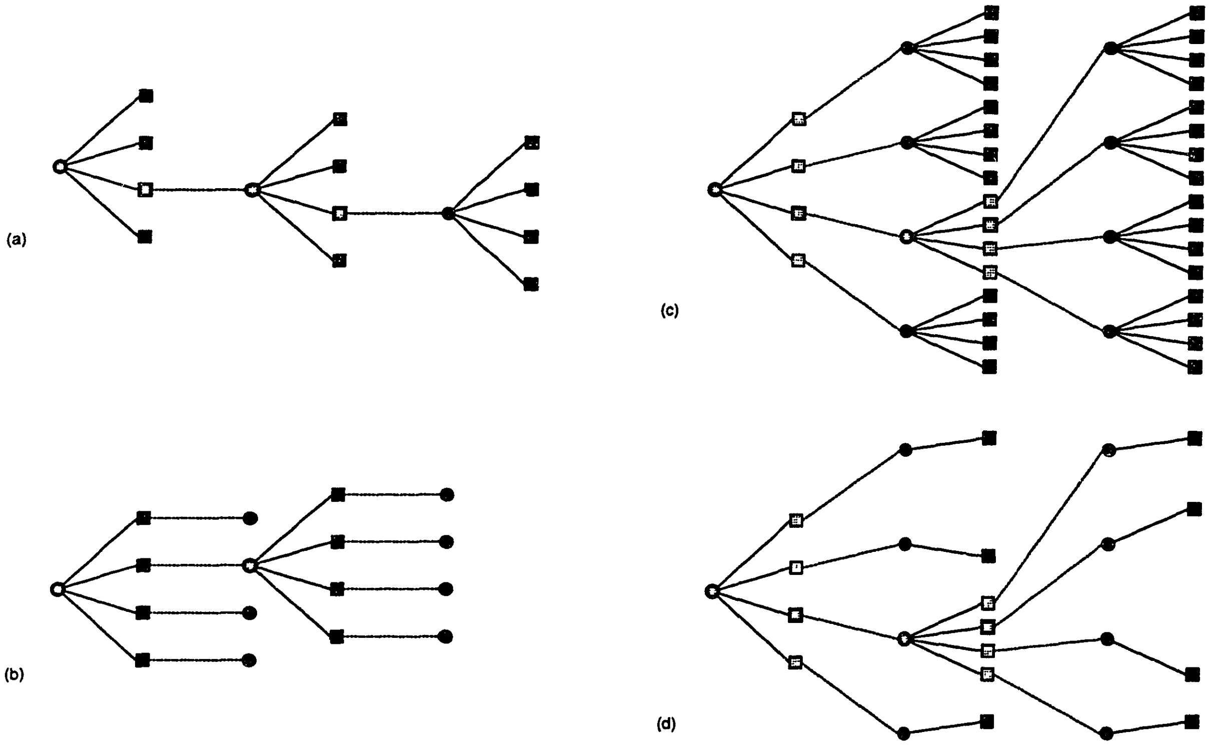


Figure 2-5: Diagram showing trajectories using various greedy updates. Circles represent states, squares actions. Circles and squares that are shaded represent states and actions that are connected with $\hat{Q}$'s or $\hat{J}$'s. Ones occurring on the actual trajectory taken are marked with a thicker line.

**2.2.6.1.1 Single-step** The single-step update is described in the first row of Table 2.1 and part (a) of Figure 2-5. If there are $N$ possible actions[4] at a given state, then with the single-step update $N$ evaluations of $\tilde{Q}$ and 1 propagation are required for each cost sample $\hat{Q}$ that is generated. These are the numbers shown in the table, and will be compared to those for the other updates.

The single-step update is the only update which does not require the availability of a model of the system, and for this reason is the most commonly used. If propagations are computationally expensive, it may be the best choice even if a model is available. With regard to cost sample generation, the single-step update is focused: it only generates samples for the state-action pairs along the trajectory that is actually followed

Selection of actions based solely on the minimization of $\tilde{Q}$ has both advantages and disadvantages. As an advantage, $\tilde{Q}$ approximates the estimate of the *expected* cost of taking a particular action from a given state. The two-step updates, in contrast, replace the expectation over all the possible successor states with the cost of a single sample successor state. If the propagation from state to state is deterministic (as in the linear case of our problem) this does not make a difference, but if it is stochastic it may be significant. The main disadvantage of selecting actions based solely on $\tilde{Q}$ is that the particular architecture chosen may not do a good job of approximating the Q-function, especially in the early stages of learning. If the approximation is poor, the action which minimizes $\tilde{Q}$ may not be much of an improvement at all. A sample in this case may be preferable to an unreliable estimate of the expectation.

**2.2.6.1.2 Two-step, state only** If a sample model of the system is available, a two-step update may be desirable; the simplest of these is shown in the second row of Table 2.1 and part (b) of Figure 2-5. This update bases its calculations on $\tilde{J}$ instead of $\tilde{Q}$. Since the approximation is removed by a step, the effects of a poor approximator are reduced; the smaller number of inputs to the approximator should help in this area as well.

[4] This assumes a small, discrete action space. If the simple action space were actually continuous, $N$ would be the number of sample points necessary to find the greedy action (3 in the parabola example mentioned earlier).

This benefit is balanced by a greater computational burden: where the single-step update required $N$ evaluations and 1 propagation to produce one cost sample, this update requires $N$ evaluations and $N$ propagations. If propagations are expensive, the cost of the extra computations may outweigh the benefits gained from better cost approximation. Also, as mentioned above, the replacement of the expectation with a sample value may adversely affect the learning process.

The replacement of the expectation with sample values also means that the action taken from any given state will not always be the same. So even though this two-step update, like the single-step update, only provides cost samples for states that it actually encounters, the addition of this random element means that it is less focused than the single-step update. This increased exploration effect could be advantageous or not depending on the particular circumstances.

It should be noted here that almost all of the discussion in this thesis assumes that learning will be based on $\tilde{Q}$, not $\tilde{J}$. The changes that would be necessary to base an algorithm on $\tilde{J}$ are relatively minor and are left to the reader.

**2.2.6.1.3 Two-step, second stage minimization** The update shown in the third row of Table 2.1 would at first seem strictly worse than the one described immediately above. This update requires $N^2$ evaluations instead of $N$, and since it uses $\tilde{Q}$ instead of $\tilde{J}$ it has more inputs going into the approximator, which doesn't help matters at all. If all this extra computation still only produced a single $\hat{Q}$ this type of update would definitely not be worthwhile.

This update method does not just produce a single $\hat{Q}$ for its $N^2$ evaluations and $N$ propagations, however. In the process of calculating the action with the minimum cost, it produces estimates of the Q-function for each other action it checks. If these additional $N-1$ side-estimates are counted along with the primary cost estimate associated with the chosen action, our update becomes as efficient as the single-step update, which also requires $N^2$ evaluations and $N$ propagations to produce $N$ cost estimates.

The problem with the above calculation is that those extra $N-1$ cost estimates may not be as valuable as the one cost estimate associated with the minimum-cost action. The utility of these side-estimates will vary according to the specific problem formulation and the details of the learning algorithm implementation. It is relatively easy to come up with situations where they would be useful, but it is also easy to come up with situations where they would be useless. In this matter, as in many others, we must rely on the judgement of the engineer implementing the learning system.

**2.2.6.1.4 Two-step with policy approximation** If a model is available and we would like to use a two-step update, we can increase computational efficiency even further by maintaining a function approximator for the policy and conducting updates as per the fourth row of Table 2.1. The price comes from the additional effort required to train the approximator and the inaccuracies introduced by the approximation. The benefit we get is that instead of $N^2$ evaluations of the $\tilde{Q}$ function approximator, we only require $N$ evaluations of $\tilde{Q}$ and $N$ evaluations of $\tilde{\pi}$ to get our $1+(N-1)$ cost estimates.

It is important to understand here that even though we are maintaining an explicit policy through the $\tilde{\pi}$ approximator, actions are still being chosen based on the implicit minimization shown in the table. We are still dealing with a simple action space. In the case of a complex action space, where an explicit policy is required, the samples used to update $\tilde{\pi}$ are generated in a separate step after $\tilde{Q}$ has been updated. In this case, the samples for updating $\tilde{\pi}$ are generated in the main learning trajectories along with the $\hat{Q}$'s used to update $\tilde{Q}$. So if this two-step update is used, the policy update step would be a simple operation similar to the updating of $\tilde{Q}$, far simpler than the elaborate calculations used in updating the policy for a complex action space.

### 2.2.6.2 Complex action spaces

*If a system's action space does not qualify as simple according to the definition in Section 2.2.6.1, we call it* complex. Complex action spaces can be large finite sets, countably infinite sets, and uncountably infinite sets. Normally, continuous action spaces will be complex, though some potential exceptions have already been described.

The essential distinction is that with a complex action space the policy must be kept track of explicitly using a function approximator,

$$\pi(\theta_k) = \tilde{\pi}(\theta_k, r) \tag{2.19}$$

which is updated in a separate step (the policy update step currently being discussed) by generating a series of estimates $\{\hat{\pi}\}$ of a policy which is greedy with respect to the current best estimate of the value function ($\tilde{Q}$ or $\tilde{J}$). The same basic options are available for conducting greedy updates as with simple action spaces, but the comparisons are somewhat different. This is shown in Table 2.2.

The difference comes from the separation of the greedy calculation from the trajectory generation process. With a simple action space, the question was how many evaluations and how many propagations were necessary to produce a single cost estimate. With a complex action space, the generation of cost estimates is a completely separate process, and the question becomes how many evaluations and propagations are necessary to produce a single $\hat{\pi}$. With a simple action space, at least one propagation was always necessary in order to generate the next step in the trajectory, but with complex action spaces, since the policy update procedure is divorced from trajectory generation, the single-step update need not perform any propagations to generate a $\hat{\pi}$.

Still, with complex action spaces the major issue is not *how* to update the policy for each point, but *which* points to update. We normally will be paying a substantial penalty in accuracy by replacing direct minimization at each step with an approximated policy; to make this worthwhile we must realize correspondingly large computational savings by reducing the frequency with which minimizations are carried out.

The trade-offs in choosing sample points for policy updates are similar in many ways to those involved in choosing sample points for cost updates. If the state space is relatively low dimensional and bounded, one might decide to carry out the necessary minimizations at each point in a grid covering the entire space. For higher dimensional or unbounded

| Update Method | | $\tilde{Q}$ Evals | Props | $\tilde{\pi}$ Evals |
|---|---|---|---|---|
| Ideal | $\pi^*(x) = \arg\min_{a\in\mathcal{A}} Q^*(x, a)$ | $N$ | 0 | 0 |
| Practical | $\tilde{\pi}_k = \arg\min_{a\in\mathcal{A}} \tilde{Q}(\theta_k, a, r)$ | | | |
| Ideal | $\pi^*(x) = \arg\min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma J^*(x_{k+1}) \mid x_k = x\right]$ | $N$ | $N$ | 0 |
| Practical | $\tilde{\pi}_k = \arg\min_{a\in\mathcal{A}} \left(g\left(\zeta_k^-, a, y_k\right) + \gamma \tilde{J}(\theta_k, r)\right)$ | | | |
| Ideal | $\pi^*(x) = \arg\min_{a\in\mathcal{A}} \mathrm{E}\left[g(x_k, a, x_{k+1}) + \gamma Q^*(x_{k+1}, \pi(x_{k+1})) \mid x_k = x\right]$ | $N$ | $N$ | $N$ |
| Practical | $\tilde{\pi}_k = \arg\min_{a\in\mathcal{A}} \left(g\left(\zeta_k^-, a, y_k\right) + \gamma \tilde{Q}\left(\theta_{k+1}, \tilde{\pi}(\theta_{k+1}, r), r\right)\right)$ | | | |

Table 2.2: Comparison of alternate greedy updates for complex action spaces. For the non-ideal updates the $\tilde{\pi}$ samples that are gathered are used to train the $\tilde{\pi}$ approximator.

state spaces, one might use points that have been encountered in the current trajectory or previous trajectories or choose them according to some random distribution. If update points are chosen based on trajectories past or present there is a danger of overspecialization, but if they are chosen randomly there may be substantial wasted effort. This conflict between exploration and exploitation is a recurring theme in the implementation of reinforcement learning algorithms.

Even after the methods with which the updates are to be conducted and the sample points collected are determined, we must still choose the frequency with which to conduct updates and the number of sample points to gather for each update. This may be complicated by the use of function approximators to approximate the policy. Normally, the policy improvement theorem [13] allows us to conduct partial updates with complete freedom. This theorem states that as long as the policy is improved at each individual state, the policy as a whole is improved. But when function approximators with generalization are used, improvement in the policy at one state can have an effect on the policy at nearby states, and there is no guarantee that this will be an improvement. One would assume in this situation that the more complete the update, the less potential exists for such problems. Thus, instead of conducting frequent updates with only a few sample points in each, it would seem to make sense to conduct larger updates on a less frequent basis. Other factors influence this decision as well. Obviously, for the use of a function approximator to be warranted the ratio of points at which a given policy is used to the points used to update it must be high. This would push toward less frequent updates. On the other hand, continuing to use the same policy for too long can slow the learning process. All these factors must be balanced in choosing a policy updating algorithm.

One side benefit of calculating policies explicitly rather than implicitly should be mentioned. If an explicit policy exists, it can be used to provide an initial guess in the minimization process, potentially speeding it up. However, the increase in speed from using this initial guess may also come with an increase in the likelihood of getting stuck in a local minimum, so caution should be used when exercising this option.

The choices made for handling complex (continuous) action spaces in this thesis were as follows. An explicit policy update step was conducted after each trajectory. In this update step, a number of points equal to $\eta N$ were selected at random from the points in the most recent trajectory, where $N$ is the total number of points in the trajectory and $\eta$ is an algorithm parameter. For each of these points in turn, an action was selected according to either the first or the third greedy updates given in Table 2.2. Actual minimizations were carried out by testing each potential action at the resolution allowed by the function approximator.[5] To avoid duplication of effort, sample points were discarded if it was found that all the weights in their area of influence had already been modified in the current update step. The $\tilde{\pi}$ approximator was then trained using the accumulated $(\theta, \hat{\pi})$ pairs.

### 2.2.7 Initialization

Aside from setting the various algorithm parameters, the major task of this first step in the learning algorithm is to initialize the function approximators. This may seem like a simple operation, but depending on the circumstances may become a fairly involved procedure.

[5] A more efficient minimization method could have been used, but this one avoided any question of local minima.

As will be shown in Section 4.2, the choices made here can have a significant effect on the learning process.

The first step in initializing the function approximators is to choose the initial weight vectors $r$. The way this is done will depend on the particular architecture that has been chosen. For an MLP, the most common initialization method is to use random values that are small in magnitude. For a CMAC, the weights are typically all set to a single value.

At this point the user may want to do some training to get the function approximators to match desired initial functions. For $\tilde{Q}$, we have elected to approximate an initial function of $Q(\theta_k, a_k) = K$, where $K$ is a constant. If a CMAC is to be used, this is easily accomplished by setting all the weights to $K$. For most other approximation architectures, some degree of preliminary training will be necessary to approximate even this simple constant function. If a $\tilde{\pi}$ approximator is to be used, we train it to match any initial policy guess that may be available. If no initial policy guess is available, we default to training to match a constant value as with $\tilde{Q}$, and the initialization stops here.

If an initial policy guess $\pi_0$ is available, however, we add one more step to the initialization process for simple action spaces. In order to start the learning process off on a sound footing, we run a number of preliminary trajectories with the regular learning algorithm, but using $\pi_0$ for the policy and an exploration rate of $\epsilon = 0$. This step is necessary for simple action spaces but not for complex action spaces because with simple action spaces our algorithm chooses actions implicitly based on the current value of $\tilde{Q}$. If no preliminary trajectories were run, the actions chosen in the first few trajectories might bear little resemblance to those of the initial policy guess. With complex action spaces the actions are chosen explicitly using $\tilde{\pi}$, so this extra step is not necessary.

# Chapter 3

# Learning with G-functions

When treating learning problems a situation may sometimes be encountered where the Q-function has characteristics which make learning difficult. For instance, the cost variation in the action direction may be much smaller than the variation in the state direction, i.e.

$$|Q(x, a + \Delta a) - Q(x, a)| \ll |Q(x + \Delta x, a) - Q(x, a)|. \tag{3.1}$$

When this occurs, function approximators which are trained based on least squares criteria may be confounded, paying attention only to the (larger) state differentials and losing the action differential which is needed for learning to take place. Baird [7] has shown that this problem can easily occur when fine discretizations of continuous-time systems are treated.

In the course of our research we encountered a few problems which seemed to be affected by this type of issue. In our attempts to improve learning performance we had cause to investigate a class of transformations proposed by Baker [9] which can be used to transform the Q-function, producing a sort of generalized state-action value function. In this chapter we present new results relating to learning using these functions. Those with little interest in the subject of reinforcement learning for its own sake may safely skip this chapter and proceed to the examples beginning in Chapter 4.

## 3.1 Generalized State-Action Value Functions

Building on the ideas presented by Baird in his discussion of Advantage Updating, Baker [9] showed that it is possible to apply a transformation to the Bellman equation that gives rise to a whole new class of state-action value functions. These functions can be tailored to accentuate or deemphasize certain elements of the Q-function while retaining the essential relationship to the value function. We will refer to these transformed Q-functions as *G-functions*, as they can be thought of as *generalized* state-action value functions.

Baker defines the G-function implicitly using the equation[1]

$$\begin{aligned} \phi(G(x,a) - J(x), x) &= B(x,a,J) - J(x) \\ &= Q(x,a) - J(x) \end{aligned} \tag{3.2}$$

where $\phi(y,x)$ is real-valued, continuous, and strictly monotone increasing, and for each $x$ we have $\phi(0,x) = 0$. In the above equation $B$ is what Baker calls the Bellman operator and depends on the discount factor, system dynamics, cost function, etc. For the purposes of our discussion we will be using

$$B(x,a,J) = E[g(x,a) + \gamma J(y)] = Q(x,a) \tag{3.3}$$

The monotonicity restrictions on $\phi$ ensure that $\phi(\cdot,x)$ is invertible and allow us to solve Equation (3.2) for $G(x,a)$, giving

$$G(x,a) = \phi^{-1}(Q(x,a) - J(x), x) + J(x). \tag{3.4}$$

Note that $G$ retains the relationship of $Q$ to the value function $J(x) = \min_a G(x,a)$, and that for $\phi(y,x) = y$, $G(x,a) = Q(x,a)$. Another useful manipulation of Equation (3.2) gives

[1] Baker uses $\Phi$ instead of $G$.

it in terms of $Q(x,a)$:

$$Q(x,a) = \phi(G(x,a) - J(x), x) + J(x). \tag{3.5}$$

For our discussion of problems with large discount factors, the special case of Equation (3.2) where $\phi(y,x) = y/\kappa$ is of particular interest. For this special case Equation (3.4) becomes

$$G(x,a) = \kappa Q(x,a) + (1-\kappa) J(x). \tag{3.6}$$

When $\kappa > 1$ this transformation accentuates the effects of individual actions, which is precisely what we are looking for.

Operating in the context of a discretization of a continuous-time problem, if we replace our $\gamma$ with $\gamma^{\Delta t}$ and our $\kappa$ with $1/K\Delta t$, where $\Delta t$ is the discretization time-step, we have the Advantage function Baird defines in terms of his Advantage Learning algorithm[2] [8, 26]. However, $G(x,a)$ does not represent a relative "advantage" as with Baird's original definition of the Advantage function in terms of his Advantage Updating algorithm, and its use need not be limited to discretizations of continous-time problems.

## 3.2 Learning with the G-function

Though Baker is responsible for the original development of the G-function, he does not comment on the effects its use will have on the standard reinforcement learning algorithms. Baird, while he claims to have shown a proof of convergence for Advantage Updating, has no comments on the convergence properties of his Advantage Learning algorithm. In the paragraphs that follow, we will attempt to shed light on both these issues.

There are two major types of reinforcement learning algorithms; Sutton refers to them as on-policy and off-policy algorithms [56]. When a model is available and learning is conducted using the value function $J(x)$, on-policy algorithms are represented by the combination of

[2] Baird gives different definitions of $A(x,a)$ for Advantage Updating and Advantage Learning

TD($\lambda$) and generalized policy iteration, while off-policy algorithms are represented by dynamic programming's value iteration. When there is no model and the learning is conducted using the state-action value function $Q(x,a)$, the most commonly considered on-policy algorithm is SARSA($\lambda$) and the most common off-policy algorithm is Q-learning.

On-policy algorithms such as SARSA($\lambda$) attempt to learn the values for the current policy instead of the values for the optimal policy. After spending some amount of time learning the value function for the current policy, a policy improvement step is carried out to generate a new policy and the process is repeated. For SARSA(0) the updates of $Q^{\pi}(x,a)$ are carried out via

$$Q_n^{\pi}(x_k, a_k) = (1-\alpha_n) Q_{n-1}^{\pi}(x_k, a_k) + \alpha_n \left(g_k + \gamma Q_{n-1}^{\pi}(y_k, \pi(y_k))\right) \tag{3.7}$$

where the action $a_k$ is selected according to the current policy, and results in a new state $y_k$. The equivalent update for $G^{\pi}(x,a)$ would thus be

$$\begin{aligned} G_n^{\pi}(x_k, a_k) &= (1-\alpha_n) G_{n-1}^{\pi}(x_k, a_k) + \\ & \alpha_n \left[\phi^{-1}\left(g_k + \gamma G_{n-1}^{\pi}(y_k, \pi(y_k)) - J_{n-1}^{\pi}(x_k), x_k\right) + J_{n-1}^{\pi}(x_k)\right]. \end{aligned} \tag{3.8}$$

However, since $a_k$ is chosen according to the current policy, we have $G_{n-1}^{\pi}(x_k, a_k) = J_{n-1}^{\pi}(x_k)$ for all $k$ and so Equation (3.8) reduces to

$$G_n^{\pi}(x_k, a_k) = G_{n-1}^{\pi}(x_k, a_k) + \alpha_n \phi^{-1}\left(g_k + \gamma G_{n-1}^{\pi}(y_k, \pi(y_k)) - J_{n-1}^{\pi}(x_k), x_k\right). \tag{3.9}$$

In the special case where $\phi(y,x) = y/\kappa$ Equation (3.9) becomes

$$G_n^{\pi}(x_k, a_k) = (1-\alpha_n\kappa) G_{n-1}^{\pi}(x_k, a_k) + \alpha_n\kappa \left(g_k + \gamma G_{n-1}^{\pi}(y_k, \pi(y_k))\right). \tag{3.10}$$

If $\alpha_n\kappa < 1$ then this is exactly the same as the update in Equation (3.7) and the function learned is not $G^{\pi}(x,a)$ but $Q^{\pi}(x,a)$! If $\alpha_n\kappa > 1$ then this update has the potential for divergent behavior. In fact, we can see that as long as actions are chosen according to the current policy, for any choice of $\phi$ we have $G^{\pi}(x, \pi(x)) = Q^{\pi}(x, \pi(x))$. This is because the transformation we have performed only affects the values of actions that are *not* on the current policy. Thus $G^{\pi}(x, a)$ cannot be learned on-policy.

The optimal G-function can be learned off-policy, however. Remember that the Q-Learning algorithm updates the Q-function via

$$\begin{aligned} Q_n(x_k, a_k) &= (1 - \alpha_n) Q_{n-1}(x_k, a_k) + \alpha_n \hat{Q}_{n-1}(x_k, a_k) \\ &= (1 - \alpha_n) Q_{n-1}(x_k, a_k) + \alpha_n (g_k + \gamma J_{n-1}(y_k)) \end{aligned} \tag{3.11}$$

with

$$J_n(x_k) = \min_a Q_n(x_k, a). \tag{3.12}$$

The most obvious way of mimicking this with the G-function would be to use an update like:

$$\begin{aligned} G_n(x_k, a_k) &= (1 - \alpha_n) G_{n-1}(x_k, a_k) + \alpha_n \hat{G}_{n-1}(x_k, a_k) \\ &= (1 - \alpha_n) G_{n-1}(x_k, a_k) + \\ &\quad \alpha_n \left[\phi^{-1}(g_k + \gamma J_{n-1}(y_k) - J_{n-1}(x_k), x_k) + J_{n-1}(x_k)\right] \end{aligned} \tag{3.13}$$

with the value function calculated according to

$$J_n(x_k) = \min_a G_n(x_k, a). \tag{3.14}$$

When $\phi(y, x) = y/\kappa$ Equation (3.13) becomes

$$G_n(x_k, a_k) = (1 - \alpha_n) G_{n-1}(x_k, a_k) + \alpha_n [\kappa (g_k + \gamma J_{n-1}(y_k)) + (1 - \kappa) J_{n-1}(x_k)] \tag{3.15}$$

which corresponds to Baird's proposed Advantage Learning algorithm.

Since the exact conditions under which Equation (3.15) converges are an open research question, the potential for divergent behavior when $\alpha_n \kappa > 1$ observed in our analysis of

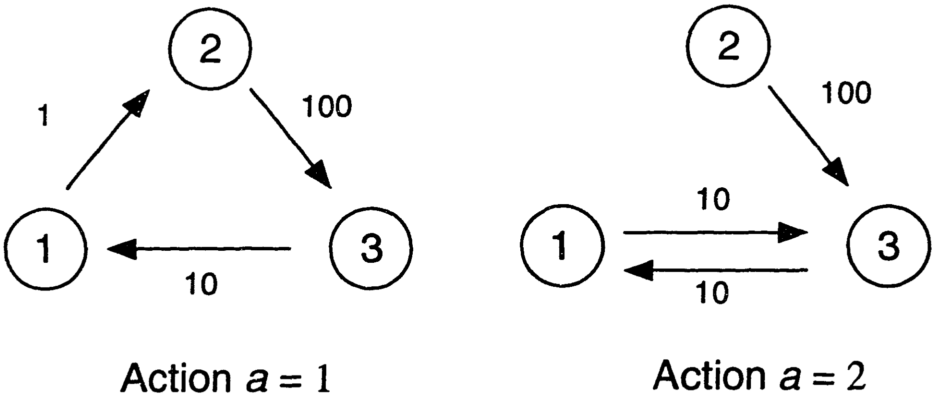


Figure 3-1: Sample Markov Decision Problem

possible on-policy learning algorithms suggests that a check for similar behavior with this off-policy algorithm is in order. In fact, in Chapter 5 we will see that when Equations (3.14) and (3.15) are applied to a measurement scheduling problem involving complex nonlinear estimation the algorithm diverges for $\alpha\kappa > 1$ (even when a decreasing learning rate was used) but performs well as long as care is taken to keep $\alpha\kappa < 1$ from the beginning.

A more concrete example of how this divergence can occur can be found in the simple example MDP from Figure 3-1. For this problem we find that with $\kappa = 10$, $\alpha$ constant at 0.5, and both $G_0$ and $J_0$ set to zero, the Advantage Learning algorithm diverges. Experimentation shows that overshoots occur when $\alpha\kappa > 1$ and divergence occurs for $\alpha\kappa > 1.93$. Of course, the standard Q-Learning algorithm calls for $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$, so we should not necessarily expect convergence. However, with a deterministic MDP Q-Learning will converge even with a constant learning rate less than one. One could argue that if the learning rate meets the stochastic approximation conditions it will eventually enter a regime where $\alpha_n\kappa < 1$ and so attain convergence. But in practice divergent behavior for even a finite length of time is unacceptable.

A close examination of the algorithm's behavior in states with only a single possible action casts some light on the reasons for this divergent behavior. For such states, we have

$$G_n(x,a) = \min_a G_n(x,a) = J_n(x). \tag{3.16}$$

When this is substituted into Equation (3.15) we get

$$\begin{aligned} J_n(x_k) &= (1-\alpha_n) J_{n-1}(x_k) + \alpha_n \left[\kappa\left(g_k + \gamma J_{n-1}(y_k)\right) + (1-\kappa) J_{n-1}(x_k)\right] \\ &= (1-\alpha_n\kappa) J_{n-1}(x_k) + \alpha_n\kappa\left(g_k + \gamma J_{n-1}(y_k)\right). \end{aligned} \tag{3.17}$$

Clearly we can run into trouble when $\alpha_n \kappa > 1$ and the same state is repeatedly updated.

An alternative off-policy algorithm takes an approach based more closely on the Q-learning algorithm. If we define

$$Q_n(x,a) = \phi\left(G_n(x,a) - J_n(x), x\right) + J_n(x) \tag{3.18}$$

and substitute Equation (3.18) into Equation (3.11) we get

$$\begin{aligned} \phi\left(G_n(x,a) - J_n(x), x\right) + J_n(x) = \quad & (1-\alpha_n)\left[\phi\left(G_{n-1}(x,a) - J_{n-1}(x), x\right) + J_{n-1}(x)\right] \\ & + \alpha\left(g_k + \gamma J_{n-1}(y_k)\right) \end{aligned} \tag{3.19}$$

which contains two different indices for both $G$ and $J$, where we only want an incremented index on the G-function itself. If we instead define

$$Q_n^-(x,a) = \phi\left(G_n(x,a) - J_{n-1}(x), x\right) + J_{n-1}(x) \tag{3.20}$$

and

$$Q_n^+(x,a) = \phi\left(G_n(x,a) - J_n(x), x\right) + J_n(x) \tag{3.21}$$

and substitute into a slightly modified update equation

$$Q_n^-(x_k,a_k) = (1-\alpha_n)\, Q_{n-1}^+(x_k,a_k) + \alpha\left(g_k + \gamma J_{n-1}(y_k)\right) \tag{3.22}$$

we can solve for $G_n(x_k, a_k)$ to arrive back at Equation (3.15) with one important difference: instead of updating our value function according to Equation (3.14) it makes more sense to

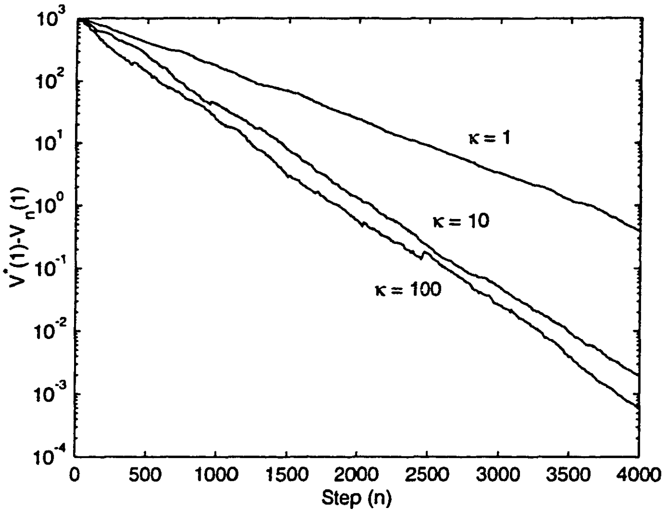


Figure 3-2: Convergence behavior of G-Learning on sample MDP for various values of $\kappa$.

update it according to

$$J_n(x_k) = \min_a Q_n^-(x_k, a_k) = \min_a \left[\phi\left(G_n(x_k, a_k) - J_{n-1}(x_k), x_k\right) + J_{n-1}(x_k)\right]. \tag{3.23}$$

Equations (3.15) and (3.23) make a new learning algorithm which we shall call *G-Learning.* If G-Learning is used instead of the combination of Equations (3.14) and (3.15) on our sample problem, we converge to the optimal value function regardless of whether $\alpha\kappa > 1$. With G-Learning, we can choose $\kappa$ to be as large as we like without having to decrease our learning rate correspondingly.

If Equation (3.14) is used, the rate of convergence for our sample MDP is roughly equivalent to that of Q-Learning using a learning rate of $\alpha\kappa$, while for G-Learning convergence speeds up with higher $\kappa$, as shown in Figure 3-2. So with G-Learning increasing $\kappa$ seems to effectively increase the learning rate as well, but in such a way that we approach but never reach unity.

Before we conclude, some qualification of these results is in order. Though G-Learning gives promising results for our sample MDP, convergence has not been shown for the general case. There are implementation issues as well. Minimization of Equation (3.23) can be carried out by minimizing $G$ and then plugging the resulting action into Equation (3.20). However, the presence of both $G_n$ and $J_{n-1}$ in Equation (3.23) means that G-Learning requires separate storage to be available for the G-function and value function. This could be a significant disadvantage and source of error when function approximation is to be used.

There may be a way around the requirement for storage of two functions, but it comes with its own problems. Let $\hat{Q}_n(x,a) = E[g(x,a)+J_n(y)]$. Then perhaps the most appropriate update would be

$$G_n(x,a) = (1-\alpha_n)\,G_{n-1}(x,a) + \alpha_n\left(\kappa\hat{Q}(x,a) + (1-\kappa)\min_a \hat{Q}(x,a)\right). \tag{3.24}$$

This would solve the divergence problem and only involves a single function, but it does have one major problem. While $\hat{Q}(x,a)$ can be approximated with a single sample point, $\min_a \hat{Q}(x,a)$ cannot. So a model would be necessary to complete this update, and even then the required minimization might be difficult.

## 3.3 Conclusion

In the preceding sections we have examined the issues involved with the use of G-functions in reinforcement learning algorithms. So far, the lessons learned can be summarized as:

- On-policy algorithms are incompatible with G-functions. Off-policy algorithms must be used instead.

- The goal of accentuating cost differentials can come into conflict with the need for rapid learning.

- In a sample MDP, Advantage Learning diverges when the accentuation is too large compared to the learning rate.

- A new algorithm, G-Learning, allows unlimited accentuation without restricting the learning rate.

We have described some of the characteristics of this new learning algorithm, including advantages such as rapid convergence compared to Q-Learning and disadvantages such as the need to keep track of two cost functions. We have not yet been able to show convergence for G-Learning, though we believe that such a proof may be possible.

# Chapter 4

# Linear Problems

For a variety of reasons, it makes sense to work with linear problems for a while before tackling the more difficult nonlinear problems which are our ultimate objective. To begin with, linear estimation problems are much better understood than nonlinear problems. For linear problems the optimal estimator is known, so we can choose our filter with confidence, knowing that we are making the best use of each measurement. The linear problem also has two characteristics which simplify the application of reinforcement learning: the optimal filter is Markovian, and when it is used the evolution of the error covariance is deterministic. Finally, since most of the literature on the measurement with cost problem deals with linear problems of one form or another, we have benchmarks which we can use to test the performance of our algorithms.

All of the problems that will be described in this chapter fit into the general formulation described in Chapter 2. Since the problems are all linear, they will hold certain aspects of their formulation in common, and we will begin the chapter with a discussion of those shared elements. Next, we will present in turn four variations of the linear measurement with cost problem:

- infinite horizon, continuous action space,
- infinite horizon, discrete action space,

- finite horizon, continuous action space, and
- finite horizon, discrete action space.

With the first two examples we will strive to give insight into both the nature of the problem and some of the issues involved with the chosen solution method. The optimal solutions to these infinite-horizon problems are not known, nor are algorithms for generating sub-optimal solutions available in the literature. The second two problems use formulations taken from the literature and thus can provide comparisons with known results. For each of these examples, we will show how it fits into the general framework we have developed, discuss our expectations with regard to its solution, and then present the results of numerical experiments using reinforcement learning.

## 4.1 Shared Characteristics

In the previous chapter we developed a framework that can handle a wide variety of problem formulations. In this chapter our objective is to use that framework to describe a series of linear problems, all of which involve estimation with costly measurements. Since the example problems that will be presented in the remaining sections are all linear, they share certain aspects of their formulation. To avoid unnecessary repetition, those common elements will be described here. The problem-specific formulation elements will be presented in the relevant sections.

The framework from Chapter 2 can be thought of in terms of a series of blocks, each of which must be filled in order to describe a problem completely. Of these blocks, all linear problems share the same state propagation equation ($f$ or $f_d$), measurement equation ($h$), and filter ($\zeta$,$\Phi$, and $\Psi$). The action space ($\mathcal{A}$), feature vector ($\Gamma$ and $\theta$), single-step cost ($g$), and cost function ($J$) will vary from problem to problem. The way in which the sensor noise $V_k$ depends on the action $a_k$ will also vary.

So, proceeding block by block, we have for all our linear examples

$$x_{k+1} = f_d\left(x_k, u_k, w_k\right) = Ax_k + Bu_k + w_k \tag{4.1}$$

where $x_k, w_k \in \mathcal{R}^n$, $u_k \in \mathcal{R}^m$, and $A \in \mathcal{R}^{n \times n}$ and $B \in \mathcal{R}^{n \times m}$ are constant, though time-varying matrices could be used. In our examples the process noise is zero mean $\mathrm{E}[w_k] = 0$, gaussian, and has constant second order statistics $\mathrm{E}[w_k w_k^T] = W$, though again none of these is required. The external input $u_k$ is assumed to be known. For the measurements we have

$$y_k = h\left(x_k, u_k, a_k, v_k\right) = Cx_k + Du_k + v_k \tag{4.2}$$

with $y, v \in \mathcal{R}^l$, and $C \in \mathcal{R}^{l \times n}$ constant. We assume the sensor noise is zero mean $\mathrm{E}[v_k] = 0$, gaussian, and uncorrelated with the process noise $\mathrm{E}[w_k v_k^T] = 0$, with second order statistics $\mathrm{E}[v_k v_k^T] = V_k$ determined by the action chosen at time $k$, $a_k$.

The addition of the gaussian assumption to our model allows us to choose a filter without hesitation, as we know that the Kalman filter is the optimal (minimum error variance) filter under these conditions. We could still use the Kalman filter without the gaussian assumption, but we would only be guaranteed of the minimum error variance estimate that is *linear* in the observations [25]. A nonlinear filter might do better.

Having chosen to use the Kalman filter, our filter state becomes $\zeta_k = (\hat{x}_k, P_k, t_k)$, where

$$P_k^- = \mathrm{E}\left[\left(x_k - \hat{x}_k^-\right)\left(x_k - \hat{x}_k^-\right)^T \middle| y_0 \dots y_{k-1}\right] \tag{4.3}$$

is the error covariance immediately before the measurement at time $t_k$,

$$P_k^+ = \mathrm{E}\left[\left(x_k - \hat{x}_k^+\right)\left(x_k - \hat{x}_k^+\right)^T \middle| y_0 \dots y_k\right] \tag{4.4}$$

is the error covariance immediately after, and

$$\hat{x}_k^- = \mathrm{E}\left[x_k \middle| y_0 \dots y_{k-1}\right] \tag{4.5}$$

and

$$\hat{x}_k^+ = \mathrm{E}\left[x_k \middle| y_0 \ldots y_k\right] \tag{4.6}$$

are the conditional expectations before and after the measurement update.

Our equations for updating and propagating the filter state are then

$$\Phi\left(\zeta_k^-, y_k, u_k, a_k\right) = \begin{bmatrix} \hat{x}_k^+ \\ P_k^+ \\ t_k^+ \end{bmatrix} = \begin{bmatrix} \hat{x}_k^- + K\left(y_k - C\hat{x}_k^- - Du_k\right) \\ (I - KC)\, P_k^- \\ t_k^- \end{bmatrix} \tag{4.7}$$

where the Kalman gain $K$ is

$$K_k = P_k^- C^T \left(C P_k^- C^T + V_k\right)^{-1} \tag{4.8}$$

and

$$\Psi\left(\zeta_k^+, u_k\right) = \begin{bmatrix} \hat{x}_{k+1}^- \\ P_{k+1}^- \\ t_{k+1}^- \end{bmatrix} = \begin{bmatrix} A\hat{x}_k^+ + Bu_k \\ AP_k^+ A^T + W \\ t_k^+ + \Delta t \end{bmatrix} \tag{4.9}$$

This information is summarized in Table 4.1. The remaining blocks necessary to fill out the framework will differ among the various linear formulations, and are described in the individual formulation sections for each sample problem. Table 4.2 provides a guide to locating this information in the text.

## 4.2 Continuous Action Space, Infinite Horizon

This first example is based on a surveillance radar tracking scenario. The continuous action space comes from the ability of phased array radar systems to distribute their total available power over a number of simultaneous pulses. For any given pulse, the accuracy of the measurement is proportional to the power put into it. The cost comes from the uncertainty in the estimate of the target's position and the quantity of resources diverted from other tasks, such as searching for new targets. In the infinite horizon case, the radar's objective is

| Block | Contents | Notes |
|---|---|---|
| $f_d\left(x_k, u_k, w_k\right)$ | $x_{k+1} = Ax_k + Bu_k + w_k$ | $\mathrm{E}[w_k] = 0 \qquad \mathrm{E}\left[w_k w_k^T\right] = W$ |
| $h(x_k, u_k, a_k, v_k, t_k)$ | $y_k = Cx_x + Du_k + v_k$ | $\mathrm{E}[v_k] = 0 \qquad \mathrm{E}\left[v_k v_k^T\right] = V_k$<br>$\mathrm{E}\left[w_k v_k^T\right] = 0$ |
| $\zeta_k$ | $(\hat{x}_k, P_k, t_k)$ | $\hat{x} \in \mathcal{R}^n \qquad P_k \in \mathcal{R}^{n \times n}$ |
| $\Phi(\zeta_k^-, y_k, u_k, a_k)$ | $\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^- - Du_k)$<br>$P_k^+ = (I - K_k C)P_k^-$ | $K_k = P_k^- C^T (C P_k^- C^T + V_k)^{-1}$ |
| $\Psi(\zeta_k^+, u_k)$ | $\hat{x}_{k+1}^- = A\hat{x}_k^+ + Bu_k$<br>$P_{k+1}^- = AP_k^+ A^T + W$ | |

Table 4.1: Shared elements of the linear formulations.

| | Continuous Action | Discrete Action |
|---|---|---|
| Infinite Horizon | Section 4.2<br>Table 4.3 | Section 4.3<br>Table 4.11 |
| Finite Horizon | Section 4.4<br>Table 4.15 | Section 4.5<br>Table 4.19 |

Table 4.2: Unique elements of the linear formulations.

to maintain an accurate track for the target over a long period of time while using as little power as possible.

The state propagation, measurement, and filter equations for this problem have been described in Section 4.1. The remaining portions of the formulation are described in Section 4.2.1 and summarized in Table 4.3. The remainder of Section 4.2 is concerned with the solution to the problem thus posed.

Since the optimal solution is unknown and no algorithms for finding suboptimal solutions to this problem are available in the literature, evaluation of the performance of our reinforcement learning algorithm becomes difficult. Section 4.2.2 attempts to address this issue by examining special cases of the problem to get an idea of the optimal solution's characteristics.

The remaining two sections use a simple scalar example problem as a platform for examination of a variety of reinforcement learning issues. The small scale of the problem allows a deep understanding of the processes involved that becomes impossible with the higher-dimensional problems that will be examined later in the thesis. In Section 4.2.3 we look at learning algorithms that sample states on a grid of evenly distributed points. In the process, we examine scaling issues, different types of function approximators, and alternative update methods. The algorithms in Section 4.2.4 sample states along trajectories generated using the learned policies, the standard method which will be used throughout the rest of the thesis. Applying trajectory-based methods to a problem for which grid-based methods can also be used allows us to compare the two approaches and understand better how they work.

The trajectory-based algorithm used in Section 4.2.4 is used throughout the thesis, and its implementation requires that a fairly large number of parameters be selected. The section ends with a description of what these parameters are and how the particular values used were chosen for this example.

### 4.2.1 Formulation

For our hypothetical radar system the set of possible actions at each timestep corresponds to the amount of energy put into the pulse. We can think of this as varying continuously from zero (no measurement) to one (maximum power), so we have as our action space the interval $\mathcal{A} = [0, 1]$. Knowing that for radar systems the measurement uncertainty is inversely proportional to the amount of energy used in a pulse, we derive a formula for $V_k$, the covariance from (4.2):

$$V_k = \begin{cases} \frac{1}{a_k} R & a_k \in (0, 1] \\ \infty & a_k = 0 \end{cases} \tag{4.10}$$

where $R$ is the sensor noise covariance when maximum power is used.

At each step, we measure our performance in terms of the track accuracy and the amount of energy used for the particular measurement taken. To acknowledge the fact that we may care more about accuracy in some states than in others, we introduce an $n \times n$ scaling matrix $L$, so we have

$$g\left(y_k, \zeta_k^-, a_k\right) = c_k = \mathrm{Tr}\left[\mathrm{LP}_\mathrm{k}^+\right] + \mathrm{Ma}_\mathrm{k} \tag{4.11}$$

where the scalar $M$ determines the relative cost of the measurement energy.

Since the Kalman filter propagates the state estimate and error covariance separately and we are measuring our performance solely in terms of the estimation error covariance and the action, all that we need to choose our action is the error covariance $P_k^-$. In fact, since $P_k^-$ is symmetric, all we need are its upper-triangular elements, so that our feature vector becomes

$$\theta_k = \left[P_k^-(1,1), \ldots P_k^-(1,n), P_k^-(2,2) \ldots P_k^-(2,n), \ldots P_k^-(n,n)\right]. \tag{4.12}$$

Finally, given all these things, we can define the infinite horizon cost as

$$J^\pi\left(\zeta_k^-\right) = \sum_{i=0}^{\infty} \gamma^i c_{k+i} \tag{4.13}$$

| Block | Contents | Notes |
|---|---|---|
| $\mathcal{A}$ | $[0,1]$ | $V_k = \frac{1}{a_k}R$ $\quad R \in \mathcal{R}^{n \times n}$ |
| $\Gamma\left(\zeta_k^-, \varsigma_k\right)$ | $\theta_k = \left[P_k^-(1,1), \ldots P_k^-(1,n), P_k^-(2,2) \ldots P_k^-(2,n), \ldots P_k^-(n,n)\right]$ | |
| $g\left(\zeta_k^-, a_k, y_k\right)$ | $c_k = \mathrm{Tr}\left[\mathrm{LP}_\mathrm{k}^+\right] + \mathrm{Ma_k}$ | $L \in \mathcal{R}^{n \times n}$ $\quad M \in \mathcal{R}$ |
| $J^\pi\left(\zeta_k^-\right)$ | $\sum_{i=0}^{\infty} \gamma^i c_{k+i}$ | $a_k = \pi(\theta_k)$ |

Table 4.3: Elements of the continuous action space, infinite horizon formulation.

where

$$a_k = \pi(\theta_k).$$

The formulation choices described above are summarized in Table 4.3.

### 4.2.2 Analysis

As we are working with a linear system and the action enters into the problem in a relatively straightforward way, one might think that the optimal solution could be found using relatively simple analytical techniques. In fact, the optimal solution can only be found analytically if drastic simplifications are made to the problem. However, even if we can't find the optimal solution analytically for the general case, examination of some special cases can tell us something about what to expect.

We begin by considering a scalar system. Here the error variance is a scalar quantity, and we can easily find for each $P_k^-$ the value of $a$ for which $P_{k+1}^- = P_k^-$

$$P = A(I - KC)PA + W = \frac{A^2RP}{C^2Pa + R} + W. \tag{4.14}$$

Solving for $a$, we get

$$a_{ss} = \frac{A^2R}{C^2(P - W)} - \frac{R}{C^2P} \tag{4.15}$$

This curve is plotted for a sample case in Figure 4-1. Note that the vector equivalent of (4.14)

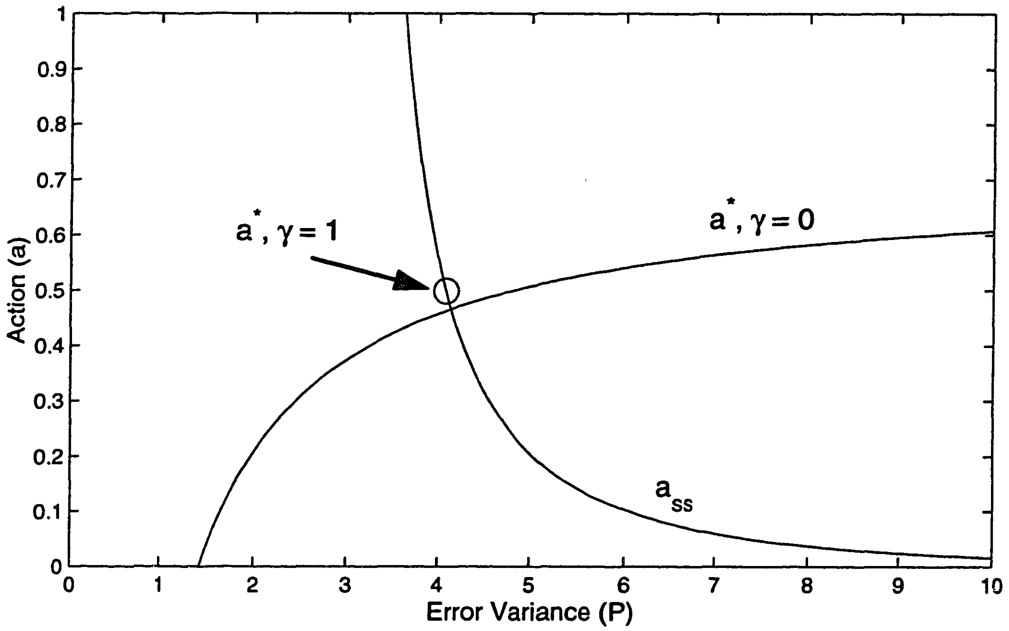


Figure 4-1: Solution space for a scalar linear problem with continuous action space and infinite horizon cost. Curves show the optimal policy when $\gamma = 0$ and the actions which will produce a steady state. The point on the steady state curve which has the minimum cost is marked with a circle.

is not generally soluble. The solution to (4.14) is stable in that $a_k > a_{ss} \Rightarrow P_{k+1}^- < P_k^-$ and $a_k < a_{ss} \Rightarrow P_{k+1}^- > P_k^-$: for any policy a trajectory that begins to the right of the steady state curve will move leftward, and a trajectory that begins to the left will move rightward. From this one might guess that the optimal policy would quickly converge to a steady state solution, though policies which produce limit cycles around the steady state are possible.

A further simplification of the problem allows us to solve for the optimal policy. If we take the special case where $\gamma = 0$ (i.e. no future costs are considered), our cost equation becomes

$$Q(P,a) = \frac{RP}{C^2Pa + R} + Ma \tag{4.16}$$

and a simple minimization yields

$$a^* = \sqrt{\frac{R}{MC^2}} - \frac{R}{C^2P}. \tag{4.17}$$

This is the other curve shown in Figure 4-1. A further simple calculation shows that the two curves intersect at

$$P = \frac{A^2}{C}\sqrt{RM} + W \tag{4.18}$$

which also happens to be the variance reached after following the policy (4.17) for one step from any point. Thus, the optimal policy for $\gamma = 0$ converges to the steady state in just a single step. From this, it seems likely that for other values of $\gamma$ convergence to the steady state is rapid, if not instantaneous.

Unfortunately, for $\gamma > 0$ analytical derivation of the optimal policy becomes intractable. We can, however, say something about what happens in the limit as $\gamma \rightarrow 1$. In this case, the transient portion of the trajectory has an infinitesmal contribution to the overall cost, and the optimal policy should thus intersect the steady state curve at the point on that curve of minimum cost. Figure 4-2 shows how the single-step cost varies along the steady state curve from Figure 4-1. By substituting (4.15) into (4.11) and minimizing with respect to $a$, we can see that this point of minimum steady state cost can be found as the solution of the

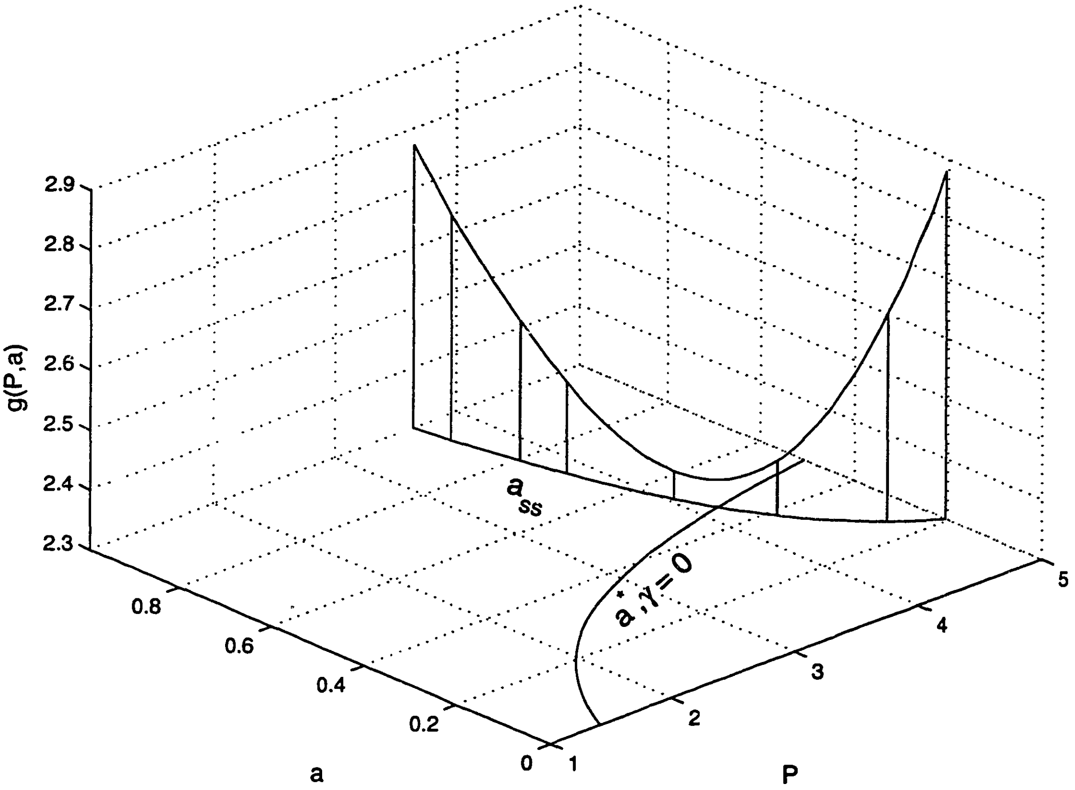


Figure 4-2: 3D plot showing the single-step cost along the steady state curve. The $a$-$P$ plane in this figure corresponds to the graph shown in Figure 4-1.

equation

$$\frac{1}{A^2} + \frac{MR}{C^2}\left(\frac{1}{P^2} - \frac{A^2}{(P-W)^2}\right) = 0. \tag{4.19}$$

This point is marked by a circle in Figure 4-1. Having located the points at which the optimal policy intersects the steady state curve for both $\gamma = 0$ and $\gamma \to 1$, one might reasonably expect that the corresponding intersections for other values of $\gamma$ lie somewhere in between.

Though the above analysis is limited to the scalar case and does not prove convergence to a steady state solution for general $\gamma$, it does provide us with some insight into the problem. For the scalar case, it provides us with expectations as to the nature of optimal trajectories and gives some reasonable bounds on where we might expect to find a steady state.

### 4.2.3 Learning with grid-based updates

Ideally, when employing a reinforcement learning algorithm one would like to conduct comprehensive updates, improving the quality of the estimate of the Q function at every point in the system's state space. The convergence proof for Q-learning, for instance, requires that each state be visited infinitely often. Unfortunately, there are a great many problems of interest with infinitely large state spaces, which make comprehensive updates impossible. However, if the dimension of the system is low and its states are bounded, a similar effect can be achieved by conducting updates at selected grid points and relying on a function approximator to fill in the gaps. As the dimension of the state space increases, the size of the grid needed to cover it increases exponentially and even this becomes impractical, necessitating a switch to trajectory-based updates.

The main algorithm presented in Chapter 2 and most examples that will be presented in this thesis rely on trajectory-based reinforcement learning. This is partly because most problems of practical interest in our chosen area involve high-dimensional, continuous state spaces. Nonetheless, in this section we will use a method which conducts updates on a grid instead of along a trajectory. This sort of grid-based update gives us a rare opportunity to visualize the way in which the learning process works, to actually see the differences between $\hat{Q}$ and $\tilde{Q}$ and the effects of different function approximators or scaling factors. Such visualization is only made possible by the combination of the special case of a scalar linear system with continuous action space with regular grid-based updates..

Since the main learning algorithm described in Chapter 2 is trajectory-based, a slightly different algorithm must be used. For this example we employ a Q-Learning based algorithm using MLP's for the function approximation. Our algorithm is then as follows:

1. Select a set of evenly distributed points $S_1 = \{(P, a)\}$ for training of $\tilde{Q}$.
2. Select a set of evenly distributed points $S_2 = \{P\}$ for training of $\tilde{\pi}$. At each point of $S_1$ evaluate $\hat{Q} = g(P, a) + \gamma\tilde{Q}\left(\theta', \tilde{\pi}(\theta', r)\right)$,[1] where $\theta' = \Gamma\left(\Psi\left(\Phi\left(P, u, a\right), u\right)\right)$.

[1] For convenience, the symbol $r$ is used for the weight vectors of both $\tilde{\pi}$ and $\tilde{Q}$, though these are separate quantities.

3. Train $\tilde{Q}$ with triples $(P, a, \hat{Q})$.

4. Minimize $\tilde{Q}$ at each point of $S_2$ to get samples $\hat{a}$.

5. Train $\tilde{\pi}$ with pairs $(P, \hat{a})$.

6. Repeat items 3-6 $N$ times.

In the above algorithm the choice of the sets $S_1$ and $S_2$ can have a strong influence on the results. Since the $a$ coordinates must come from $[0, 1]$, the selection of the distribution in that dimension is only a matter of choosing a level of resolution. Choosing a distribution for the $P$ coordinates is somewhat more problematic, as they can come from the range $[0, \infty]$. Ideally, one would like the distribution of $P$ coordinates in $S_1$ and $S_2$ to be wide enough to catch any limiting behaviors but dense enough to catch the details of the cost function near the steady state.

To get an idea of the proper range for a given system, one can find the steady-state error variances in the two limiting cases obtained if $\pi(\theta) = 1$ or $\pi(\theta) = 0$, which we shall call $P_{min}$ and $P_{max}$ respectively. Where the sets $S_1$ and $S_2$ should fall relative to these two quantities will depend on the measurement cost $M$. Two different grids were considered for a sample system: one with 20 divisions evenly distributed in the range $[P_{min}/2, 50P_{min}]$, and the other with the divisions distributed in the range $[P_{min}/2, 5P_{min}]$.

Figure 4-3 shows the situation after 60 iterations of the learning algorithm for the first of these grids. Details of the parameters can be found in Table 4.4. Looking at the upper left of the figure, we can see that the graph of $\hat{Q}$ is dominated by the region where $P$ is large and $a$ is small. Since the MLP approximating $\tilde{Q}$ is trained with the objective of minimizing the mean squared error over the sampled values, this high cost region exerts a very strong influence over the shape of $\tilde{Q}$ even though it is a region which any reasonable policy will never enter. In effect, the resources of the approximation architecture are being wasted on an area of little real concern. In the bottom graph we can see something of the difference between minimizing $\tilde{Q}$ and $\hat{Q}$. The fairly large difference between the two indicates that an algorithm which updates the policy based on minimization of $\hat{Q}$ might perform better.

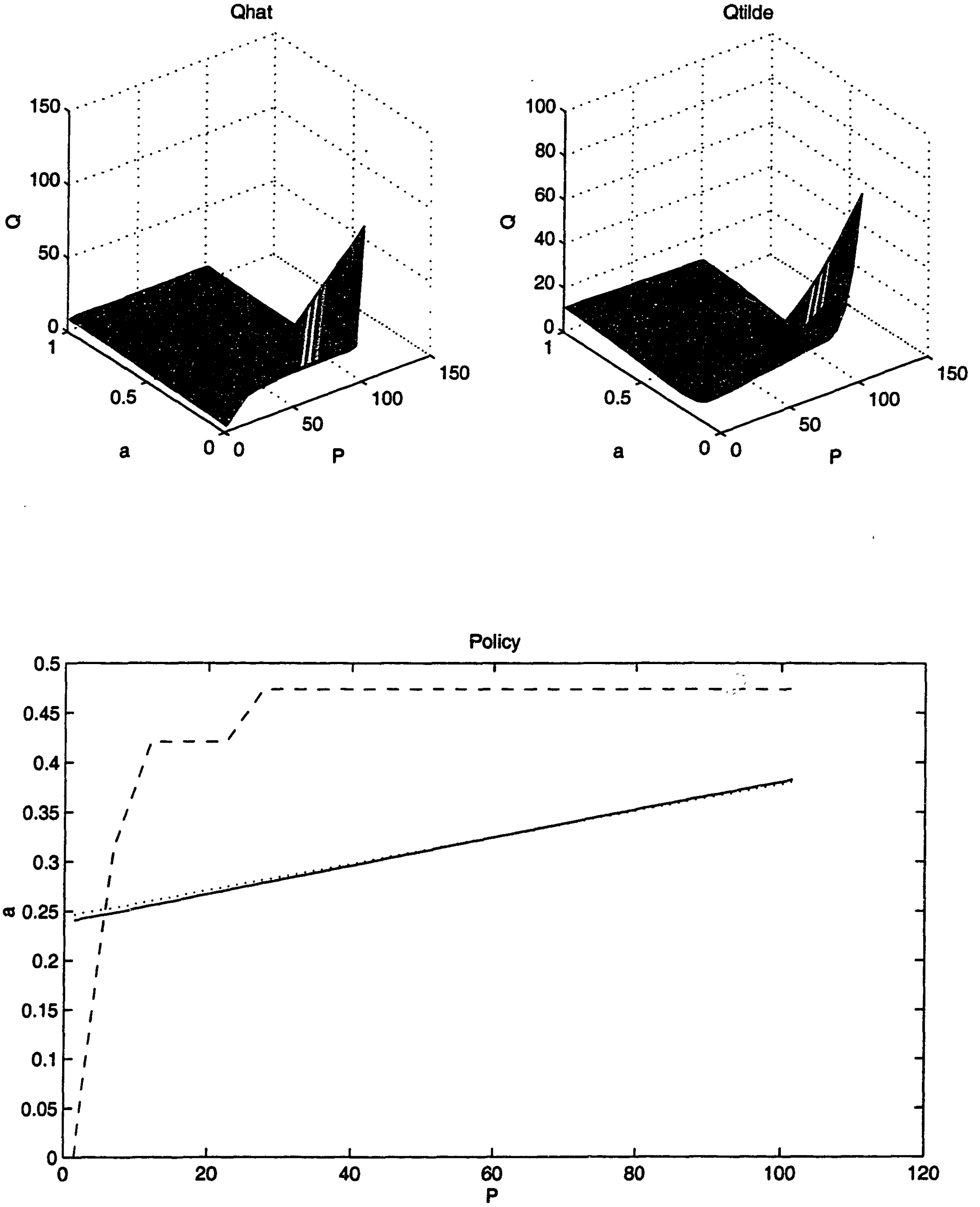


Figure 4-3: Visualization of the learning process: at upper left, $\hat{Q}(P, a) = g(P, a) + \gamma\tilde{Q}(\theta', \tilde{\pi}(\theta', r))$; at upper right, $\tilde{Q}(\theta, a, r)$. Bottom graph shows the policy approximation (solid), sample points (dotted), and minimum of $\hat{Q}$ on the grid (dashed).

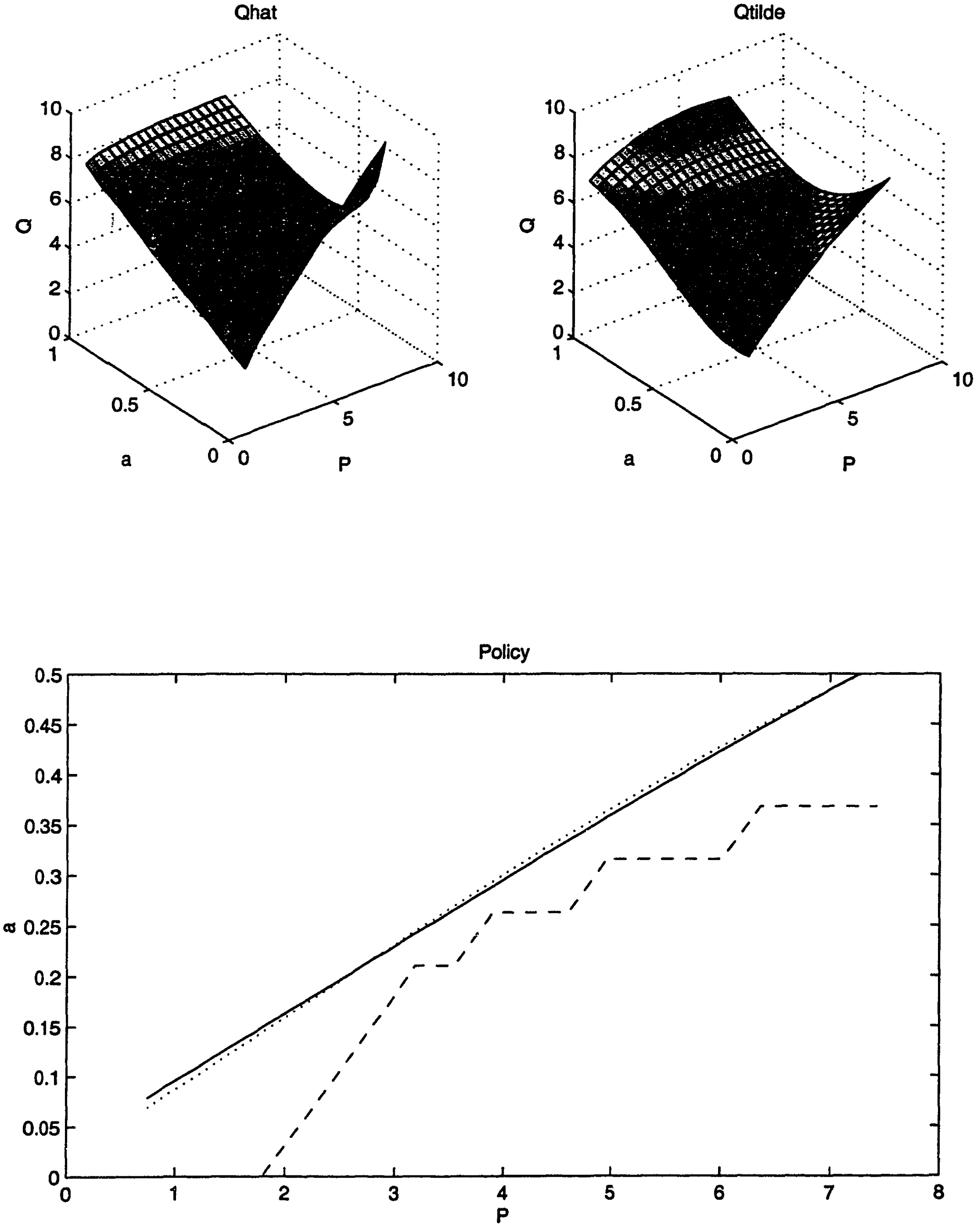


Figure 4-4: Visualization of the learning process: at upper left, $\hat{Q}(P,a) = g(P,a) + \gamma\tilde{Q}(\theta',\tilde{\pi}(\theta',r))$; at upper right, $\tilde{Q}(\theta,a,r)$. Bottom graph shows the policy approximation (solid), sample points (dotted), and minimum of $\hat{Q}$ on the grid (dashed).

| | | |
|---|---|---|
| Linear System | $A = 0.9$ | $B = 0$ |
| | $C = 1$ | $D = 0$ |
| Noise-related | $W = 1$ | $R = 1$ |
| Cost-related | $\gamma = 0.9$ | $M = 5$ |
| MLP Structure<br>(Inputs/Hidden/Outputs) | $\tilde{Q}$: (2/10/1) | $\tilde{\pi}$: (1/20/1) |
| Input Grid Size | $\tilde{Q}$: $20 \times 20$ | $\tilde{\pi}$: 20 |
| Number of Iterations | $N_{eg} = 60$ | |

Table 4.4: Parameters for tests conducted with grid-based algorithm.

Figure 4-4 shows the situation after 60 iterations for the second grid, which covers a smaller range of variances. Here the range of $\hat{Q}$ values is more balanced, resulting in an approximation which is qualitatively better in the region of minimum $a$. Looking at the bottom graph, we see closer agreement between the minima of $\hat{Q}$ and $\tilde{Q}$ than we did with the larger scale.

The policies generated by applying the learning algorithm on these two grids are compared in Figure 4-5. The solid lines are for the smaller grid. Note that when $\tilde{\pi}$ is applied outside the range it was trained on, it can produce values outside the allowable range of $[0, 1]$. These are truncated so that all actions fall within $\mathcal{A}$.

The large difference between the two policies, seen in the bottom graph, testifies to the sensitivity of the algorithm to function approximation issues. This sensitivity can also be seen in the top graph, which shows the relative costs. For small values of $P$ the policy learned on the smaller grid performs better, while for large values the policy learned on the larger grid would clearly have the lower cost were it not for the restriction which keeps all actions in the bounds set by the choice of $\mathcal{A}$. It is noteworthy that though the two policies look dramatically different, both generate trajectories that converge to steady-state values less than 4% apart.

The choice of appropriate grids for sampling the Q-function and updating the policy is

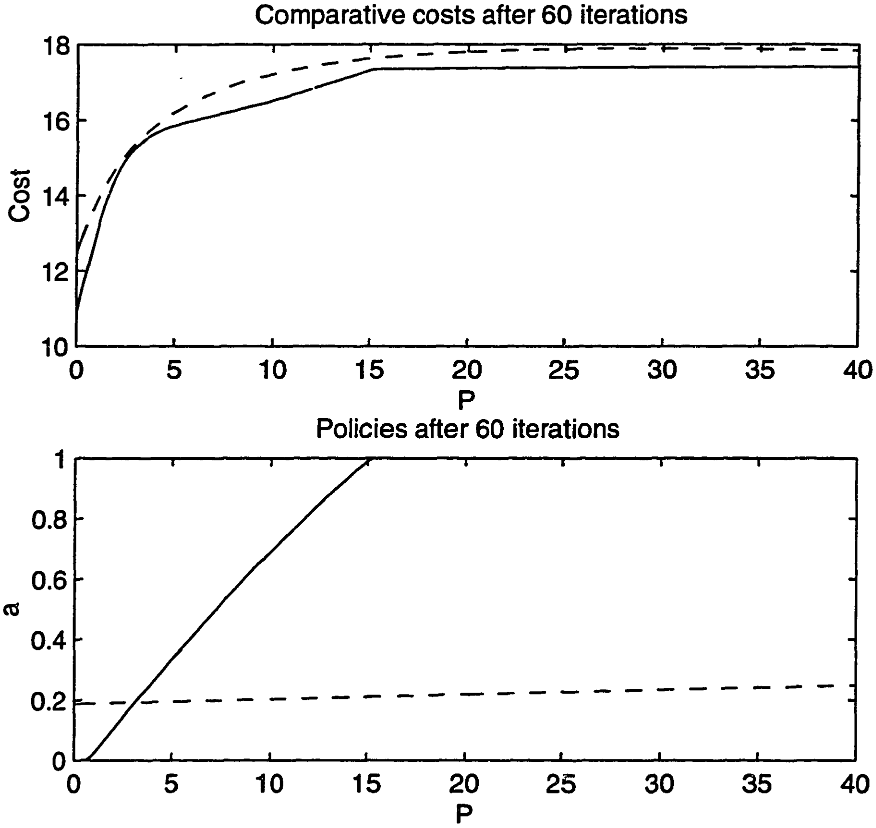


Figure 4-5: Comparison of the two grids: at top are the relative costs, and at bottom are the learned policies. The solid lines are for the finer grid.

just one of several decisions that need to be made. Another issue that must be resolved is the way in which updates are conducted. In the discussion of scaling above we pointed out the discrepancy between the policy obtained by minimizing $\tilde{Q}$ and the minimum of $\hat{Q}$. This suggests that we might get better results by generating our training data for $\tilde{\pi}$ based on the two-step update for complex action spaces described in Section 2.2.6. In other words, we would have

$$\hat{a} = \arg\min_{a\in A}\left[g\left(P,a\right)+\gamma\tilde{Q}\left(\theta',\tilde{\pi}\left(\theta',r\right),r\right)\right] \tag{4.20}$$

instead of

$$\hat{a} = \arg\min_{a\in A}\tilde{Q}\left(\theta,a,r\right). \tag{4.21}$$

When testing our learning algorithm with a two-step update, yet another issue that must be resolved becomes apparent.

Though our MLP did a fine job fitting the $\hat{a}$ data from the one-step updates, it does not do so well with the values from the two-step updates. This can be seen from the left-hand plot in Figure 4-6. In the figure, the dotted line marks the $\hat{a}$ values, and the solid line marks the MLP's approximation. There are a variety of possible explanations for this poor performance, and it is possible that by tweaking one parameter or another the approximation could be improved. As it is, though, it serves as a sort of warning about the kinds of problems that can develop in a reinforcement learning system. Even though we are using a multilayer perceptron – an architecture which has been proven to be capable of approximating an arbitrary function to an arbitrary level of accuracy – with twice as many neurons as were used to approximate the (higher dimensional) Q-function, we are not guaranteed a close approximation. In this particular case we can see this problem using a simple plot and evaluate its effects, but in a higher dimensional problem such visualization becomes impossible and issues such as this may go undetected. For this example, having detected the problem, it is relatively easy to fix. For policy approximation the 20-unit MLP was replaced with a 10th order polynomial fit, shown at right in Figure 4-6.

We now take a look at the relative performance of the learning algorithm using our various update procedures. Figure 4-7 compares three alternatives: a single-step update

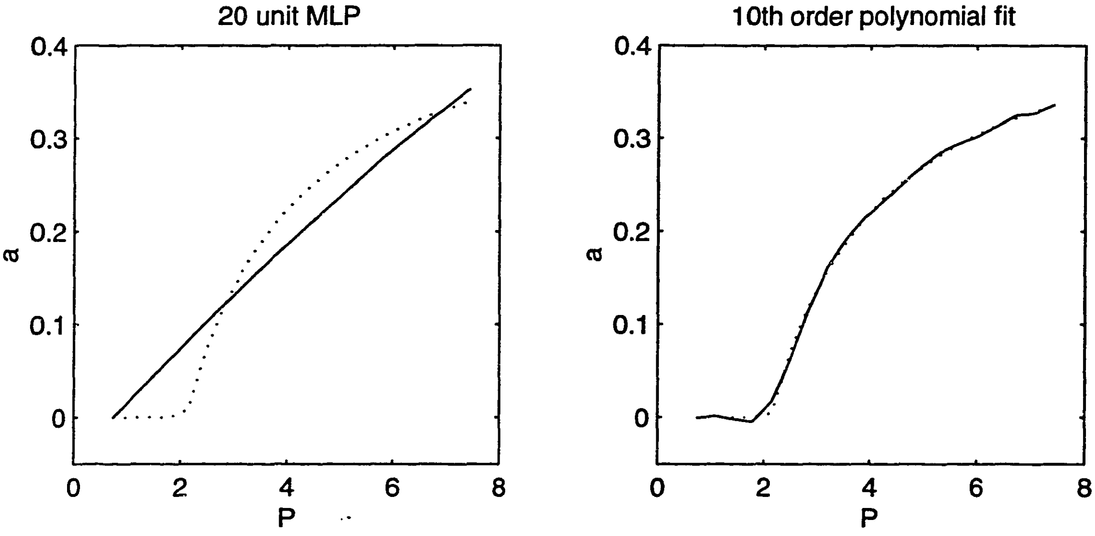


Figure 4-6: Comparison of the two policy approximation methods. Dotted lines show $\hat{Q}$ data, solid lines show $\tilde{Q}$ for the particular approximation type.

with MLP approximation, a two-step update with MLP approximation, and a two-step update with polynomial approximation. The simulations on which this data is based were conducted with a discount factor of $\gamma = 0.5$. This is because, based on the analysis conducted in Section 4.2.2, we expect the final trajectories to go to a steady-state relatively quickly. At high discount factors the steady-state portion of the trajectory is weighted more strongly in the cost, and differences in the alternative updates may be harder to see. Looking at the top of the figure, we can see that for low values of $P$ the two-step updates perform substantially better than the single-step update. The policy using a polynomial fit does better than the one using an MLP, but the difference is relatively minor.

A distinctive feature of the top graph in Figure 4-7 is that the costs of the policies using the different update methods only differ significantly at low values of $P$ even though the policies themselves exhibit significant differences over the entire range. This can be understood by examining the trajectories plotted in Figure 4-8. All of the trajectories plotted in this figure converge quickly to the same steady-state value, but the ones starting at $P_0 > P_{ss}$ reach the steady-state much more quickly than those starting at $P_0 < P_{ss}$. Essentially, the trajectories are much more sensitive to policy differences at low values of $P$

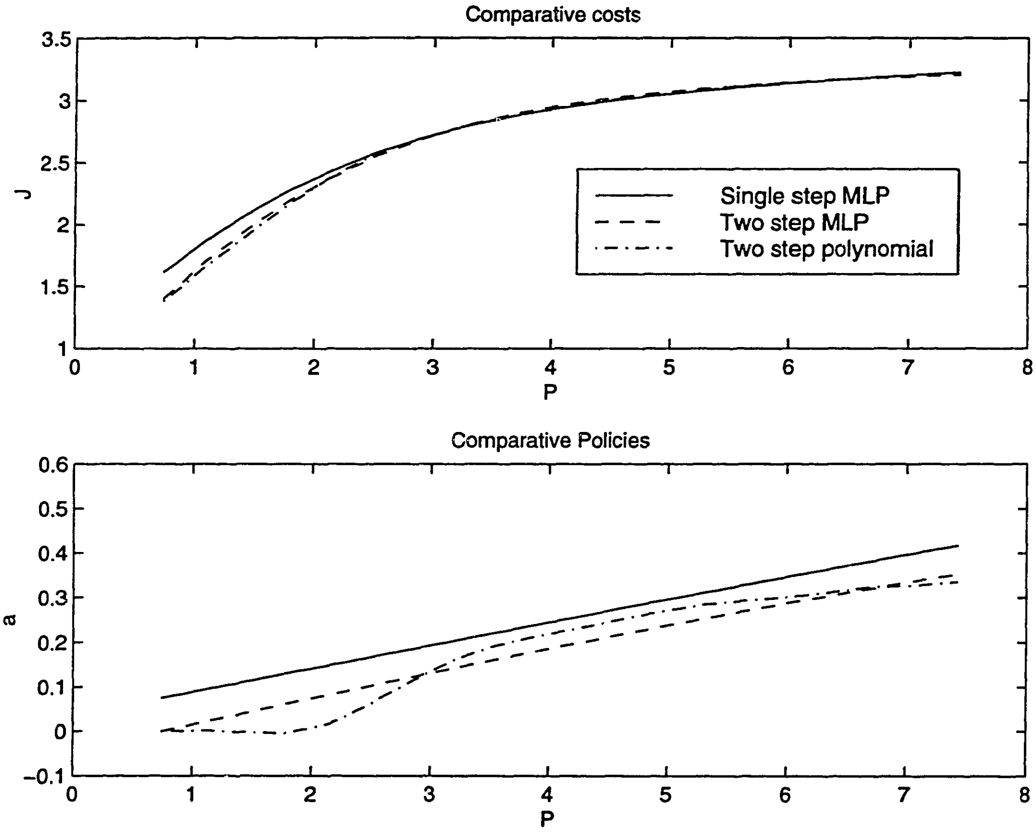


Figure 4-7: Comparison of three update methods. At top are the costs of the policies learned using a single-step update with an MLP for $\tilde{\pi}$, a two-step update with an MLP, and a two-step update with a 10th order polynomial fit. At bottom, the learned policies are shown. Simulation parameters were as in Table 4.4 except that $\gamma = 0.5$.

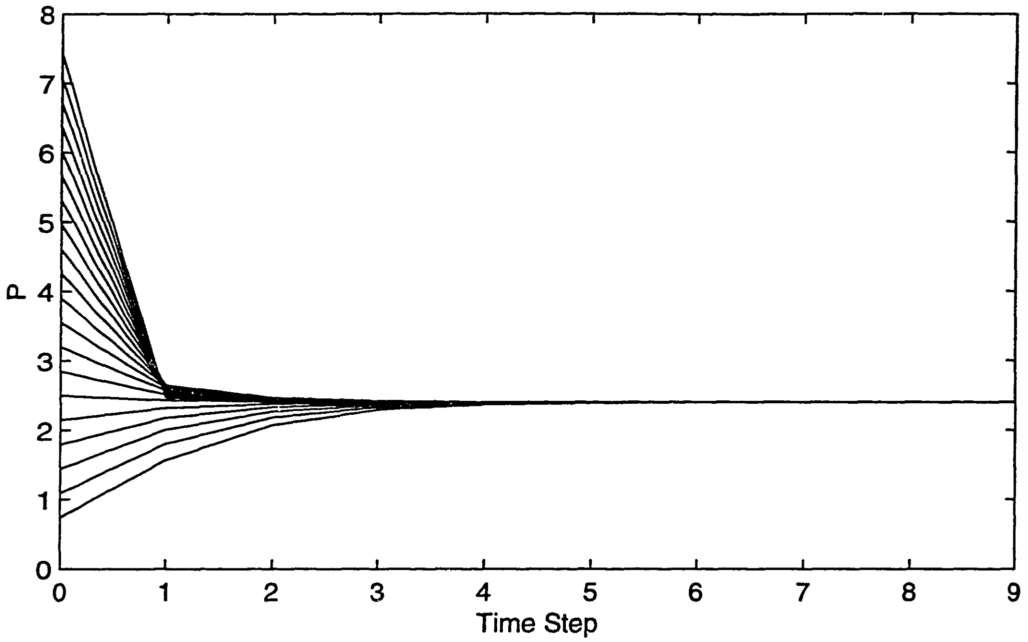


Figure 4-8: Sample trajectories using policy learned with single-step updates and MLP approximation.

than at high values. This explains the behavior seen in Figure 4-7. The major differences in cost come from the transients, which are much more significant at low variances.

A final important aspect of this comparison between update methods lies not in their differences, but in their similarities. At high variances, and hence at steady-state, all three update methods produce remarkably similar costs. The statistics for each policy at steady-state are shown in Table 4.5. Though the differences in steady-state variance are on the order of 10% and differences in steady-state actions are on the order of 50%, the steady-state costs of all three policies differ by less than 1%. Dramatically different policies can produce very similar costs.

Hopefully, the insights gleaned from these trials run using a grid-based learning algorithm on an extremely simple system will be of use when dealing with more complex systems and trajectory-based learning algorithms. The main points that should be remembered are as follows:

| Update Type | $P_{ss}$ | $a_{ss}$ | $c_{ss}$ |
|---|---|---|---|
| 1-step MLP | 2.40 | 0.16 | 3.21 |
| 2-step MLP | 2.66 | 0.11 | 3.22 |
| 2-step Polynomial | 2.74 | 0.10 | 3.24 |

Table 4.5: Comparison of steady-state values for different update methods.

- Too much exploration can degrade performance. Extreme values in areas of the state space that are unlikely to be encountered can use up approximation resources that would be better spent in more travelled areas.

- There can be a significant difference between single-step and two-step minimization for action selection.

- Just because an approximator is capable of doing a good job doesn't mean it will. Approximators that work well on one function may do poorly with another, and poor policy approximation may cause significant degradation of performance.

- Small changes in the problem formulation or learning algorithm can result in significantly different policies. If the cost function is relatively flat near the optimum, there may be a whole range of policies that have widely varying characteristics but similar costs.

### 4.2.4 Learning with trajectory-based updates

Though the small scale of our example problem allows us to use an algorithm with grid-based updates, this is the exception rather than the rule. Most of the problems that will be examined in this thesis will be complex enough to make trajectory-based updates a necessity. In order to make those later examples more transparent, we use this section to apply a trajectory-based algorithm to a problem that has already been examined using grid-based methods.

A key thing to watch for in this section is the different types of insights that can be gained from the two different approaches. Some of these differences result from the different types of function approximators used (MLP's and polynomial fits vs. CMAC's), but most are more closely related to the differences in the algorithms. Insights from both methodologies should prove to be generally useful, however. Lessons gleaned from analysis of grid-based examples may prove useful in dealing with trajectory-based examples even if the same effects are not directly visible.

With the grid-based method from the previous section, scaling was integrally combined with exploration in the selection of an update grid. The major point was that poor scaling can effectively result in excessive exploration and wasted approximator resources, ultimately hampering learning. With trajectory-based learning exploration and scaling are mostly separate. Appropriate scaling factors are determined based on statistical analysis of actual trajectories (which are in turn influenced by the chosen level of exploration); areas of importance in the state space are thus automatically identified. However, we still benefit from the lessons of the grid-based example, because we now know that overly agressive exploration diverts attention from the natural trajectories of the system and so hampers our function approximator's ability to do its job.

Another example of the different perspectives offered by the two methods comes from a look at the policies they generate. The grid-based method treats all states equally, resulting in a smooth policy over the selected input range. For this particular example, the grid-based method does not immediately reveal the central importance of the steady-state to the system. For the trajectory-based method, however, the dominance of the steady-state becomes the central fact of life. Since the vast majority of the learning time is spent in the immediate vicinity of the steady-state we obtain a policy that is extremely good in that area but relatively untrained elsewhere.

Understanding the lessons of these two methods is important. When we move to more complex examples, both grid-based methods and visualization will become impractical, so our results will be much harder to interpret. We will still face many of the same issues, however, so what we learn here should prove valuable.

The trajectory-based algorithm we use in this section (and the rest of the thesis) has been covered in detail in Section 2.2, so we will not spend time on it here. However, a quick look back before proceeding may be worthwhile.

This comparison will be conducted based on learning runs using the parameters shown in Table 4.6; we will call this set of parameters our baseline scenario. Once the results of applying our trajectory-based algorithm to the baseline scenario have been presented, we will discuss how the various parameters were chosen and show the effects of changing them.

#### 4.2.4.1 Baseline Experiments

For our baseline scenario the trajectory-based algorithm was run using SARSA($\lambda$) and the parameters laid out in Table 4.6 and the top part of Table 4.4. After each trajectory was run to completion and the corresponding policy update completed, a test trajectory of the same length was run using the updated policy with no $\epsilon$-based randomization. The discounted sum of the single-step costs was then calculated over the entire trajectory to get an approximation to $J^{\pi}(P_0)$. These costs are plotted in Figure 4-9. The cost of the final trajectory was found to be 26.67, as compared to 27.17 for the corresponding $J^{\pi}(P_0)$ from an analogous run using grid-based updating. Thus, the two algorithms give roughly the same performance. It is interesting to observe, however, that most of the actual learning seems to be completed in the first 2-3 trajectories. The remaining trajectories serve only to fine-tune the performance, with a gradual reduction in learning rates to assure convergence.

Though they seem to yield similar performance, the difference between the two methods can most clearly be seen by examining the final policy learned by the trajectory-based algorithm. This policy is plotted in Figure 4-10. Its shape reflects the nature of the problem, the learning algorithm, and the function approximator. The problem itself shapes the trajectories encountered during learning. Each starts at the initial variance of 7 and then moves quickly to the steady-state marked on the graph. As a result, most of the sample points seen in the learning process are either at the initial condition or the steady-state, as can be seen in Figure 4-11. The random exploration added to the trajectories through $\epsilon$

| | |
|---|---|
| Scaling | |
| State ($\theta$) | 10 |
| Action ($a$) | 100 |
| Cost ($Q$) | 1 |
| CMAC-related parameters (same for $\tilde{Q}$ and $\tilde{\pi}$) | |
| Memory size | 10000 |
| Number of layers | 16 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_1 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 20$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Estimation error variance | $P_0 = 7$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 100 |
| Weights for $\tilde{\pi}$ approximator | All weights initialized to 100 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 100$ |
| Number of steps/trajectory | $N = 40$ |
| Exploration rate | $\epsilon = 0.2$ |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.8$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |
| Fraction of samples updated after each trajectory | $\eta = 1$ |

Table 4.6: Parameter settings for scalar test case using trajectory-based learning. The contents of this table are duplicated and divided up into smaller sections to go along with the discussion of parameter selection in Section 4.2.4.2

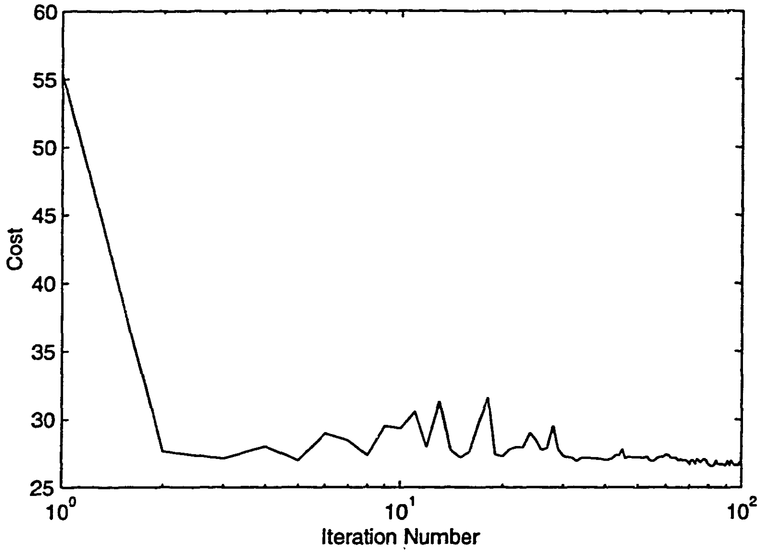


Figure 4-9: Performance of SARSA($\lambda$) on baseline scenario. Note that the cost gets to within 10% of the final value by the second iteration.

means that a few sample points fall in between these extremes. Since the CMAC is a local approximator, this means that the policy tends to default to its initial value where there is no training data. In this case, with 16 layers and a scaling factor of 10 on the state, the influence of a single point extends 1.6 variance units to either side. This explains the "V" shape in Figure 4-10 from $P = 5.4$ to $P = 8.6$, and the initial slope from $P = 0$ to $P = 1.6$.

The policy in this example was trained using trajectories that all had an initial covariance of 7. If the final learned policy was used for a trajectory with initial variance of 0 or 10, points which received no training in the learning process, the results could be significantly suboptimal. This is a fundamental problem with trajectory-based reinforcement learning, one that is not easily solved. For this example, the issue can be partially addressed by starting trajectories at a range of initial variances, though one can still only cover a finite range.

For higher dimensional systems, there will always be points in the state space that have not been encountered interspersed with those that have. The function approximator's ability to generalize may compensate for these holes to some extent, but there will still be problems. The interaction of these untrained states with those that have been trained creates

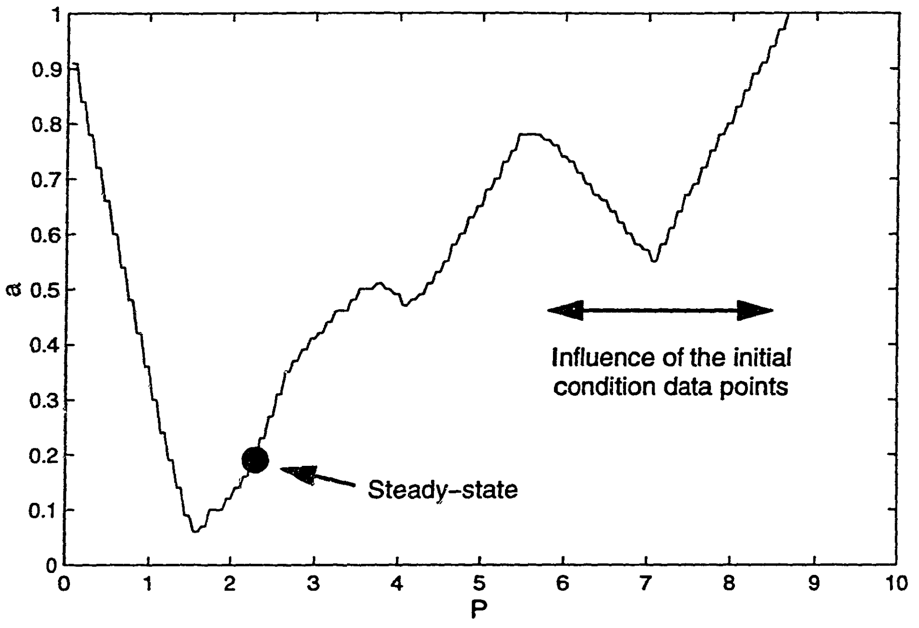


Figure 4-10: Policy found by use of SARSA($\lambda$) on baseline scenario.

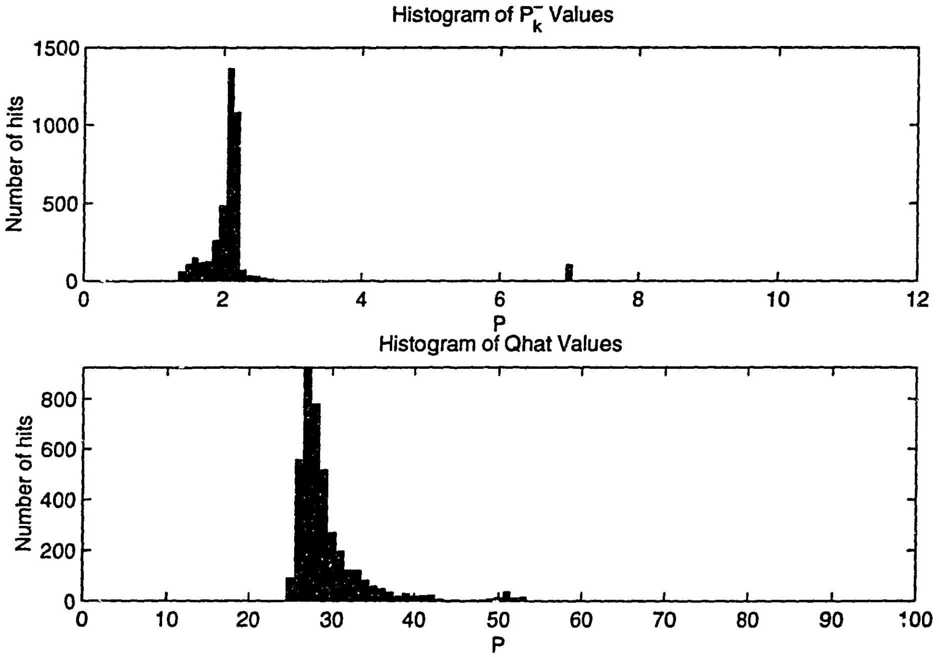


Figure 4-11: Histograms showing states and costs encountered during learning with SARSA($\lambda$) on baseline scenario. This type of histogram is used to determine appropriate scaling factors for the function approximator inputs. At top, note the spike at $P = 7$, the initial condition used in each trajectory.

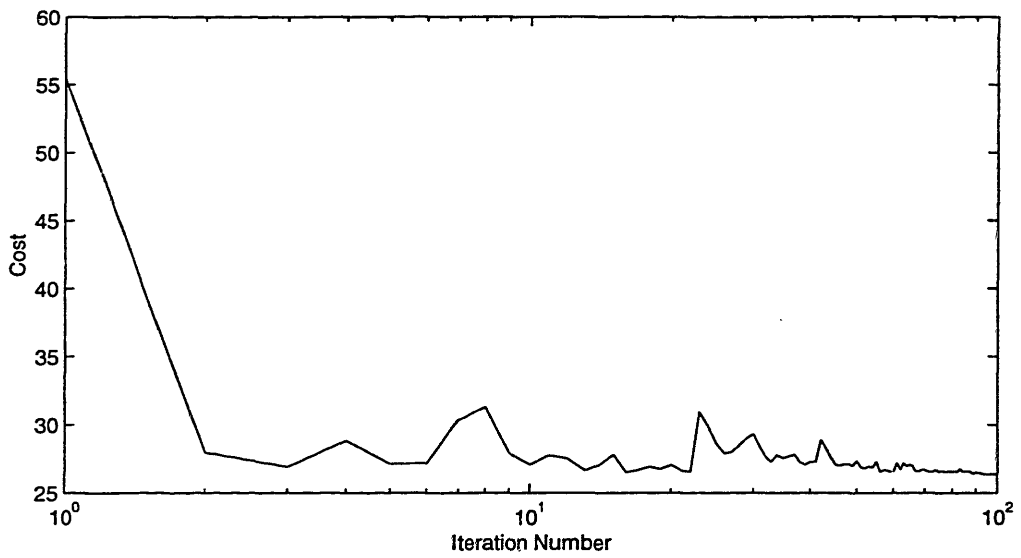


Figure 4-12: Performance of Q-Learning on baseline scenario.

a potential for unpredictable behavior which may make reinforcement learning a hard sell for many control problems.

The Q-Learning based algorithm described in Chapter 2 was also applied to our baseline scenario. The results are shown in Figure 4-12. Performance is very similar to the other two cases we have shown, with the cost of the final policy coming in at 26.39. It should be noted, though, that 100 trajectories of a Q-Learning algorithm involve substantially fewer calculations than the same number of trajectories using a SARSA($\lambda$) based algorithm.

#### 4.2.4.2 Parameter Selection

One of the chief obstacles to the wide acceptance of reinforcement learning algorithms is their complexity. Though based on relatively simple concepts, actual implementation of an RL algorithm on a real system requires the user to choose a large number of parameters. With so many degrees of freedom in the algorithm, it becomes difficult to make fair comparisons with other algorithms. Also, the experience of the user becomes a significant factor in determining performance levels achieved. Since performance can depend on good choice of the algorithm parameters and for most applications the optimal cost is unknown, one is often left wondering whether a different choice of parameters could have produced better results. We will use this example as well as the others in this thesis to try to give some insight into the logic behind parameter selection for RL problems. Of course, the exact trade-offs involved will be different for each problem, but it is hoped that describing the selection process for a few examples will provide insights that will be useful across a broader spectrum of problems.

This particular section provides a fairly comprehensive list of the parameters that must be chosen in order to run our reinforcement learning algorithm. It is intended in part as a reference should the reader wonder about the meaning and selection method for a particular parameter encountered in one of the sample problems. Though there are unique aspects to the parameter selection for each problem that is treated in this thesis, many parameters need only be described once, and that is done here. Not all readers will be interested in this level of detail; some may want to skip to Section 4.2.5.

**4.2.4.2.1 Scaling** All function approximators require that their inputs and outputs be properly scaled relative to each other. For the CMAC, which partitions the input and output spaces into discrete grids, scaling also determines resolution. Since we often don't know what state values and costs we will be encountering beforehand, choosing an appropriate scale can be somewhat problematic.

**$\theta$ scale** In this thesis, the general procedure used to select scaling factors for $\theta$ is as follows: first pick arbitrary scaling factors and run the algorithm. Next, plot a histogram with 100 divisions such as the one shown in Figure 4-11. If all the states encountered in the run are clumped into one or two bins, then the scaling is too small. If they are very spread out, then the scaling can probably be reduced, giving lower memory usage and quicker convergence without a large performance penalty.

When using CMAC's for function approximation, the exact level of scaling desired will be influenced by the number of layers in the approximator and the desired level of generalization. With a 16 layer CMAC, each state encountered affects the approximator's evaluation in the 16 adjacent bins to either side. Looking at the top graph in Figure 4-11 and considering that we are using a CMAC with 16 layers and that the total width of significant activity is only about 16 bins, our chosen scale of 10 gives a very high level of generalization.

**$\hat{Q}$ scale** The scale used for $\hat{Q}$ is chosen in a similar manner, but the number of layers in the CMAC is not important. Here the width of the active region in the histogram determines the algorithm's ability to distinguish between the costs of alternative actions. The larger the scale, the more sensitive the algorithm becomes to both real differences and noise-derived artifacts. A bit of experimentation has shown the algorithm to be relatively insensitive to changes in this parameter for this particular problem.

**Action scale** The continuous, bounded nature of this particular problem's action space means that the action scale becomes a sort of system resolution. A larger scale means a more nearly continous policy but also a more computationally expensive policy update. If a relatively small scale were chosen we would have a simple action space, and implicit policy updates would be more efficient. Since a larger scale should in theory provide more discrimination and hence a better policy as well as showcasing our ability to deal with complex action spaces, we have chosen a scale of 100.

| | |
|---|---|
| State ($\theta$) | 10 |
| Action ($a$) | 100 |
| Cost ($Q$) | 1 |

Table 4.7: Scaling Parameters.

**4.2.4.2.2 CMAC parameters** In our example we use two function approximators, one for $\tilde{Q}$ and one for $\tilde{\pi}$; for each there are a number of parameters which must be chosen. See Appendix A for a more thorough explanation of the quantities involved here.

**Memory** The amount of memory to allocate for the CMAC should be chosen in much the same way as the scaling was selected. Pick an arbitrary number, check the proportion of memory used in a sample run, and then adjust the memory size appropriately. Since inputs are mapped to memory locations using a hashing function, performance may start to degrade once full memory usage is reached.

**Layers** The number of layers in the CMAC determines the level of generalization that will be present. With only a single layer, the CMAC degrades to a table-based function approximator. The total physical memory used by the CMAC is roughly equal to the number of layers times the memory spaces allocated, so reducing the number of layers can significantly ease memory requirements. The number used in the example, 16, was recommended in the UNH CMAC documentation as a good starting point.[2]

**Learning Rates** The UNH CMAC code uses two learning rates which are expressed as bit-shifts (powers of 1/2) and are designated $\beta_1$ and $\beta_2$. $\beta_1$ is the base learning rate, and $\beta_2$ is a penalty imposed on the size of the weights to keep them from drifting toward large values. The UNH CMAC documentation recommends that $\beta_2$ be somewhat higher than $\beta_1$; hence the initial choice of $\beta_1 = 1$, $\beta_2 = 4$.

**Rate Reduction** In order to get the algorithm to converge to a single policy instead of jumping around in a limit cycle, we must gradually reduce the learning rates. In our algorithm the values of $\beta_1$ and $\beta_2$ are increased by one every $N_\beta$ iterations. The best value for $N_\beta$ will depend on the memory usage and number of trajectories to be run. High memory usage implies a large number of different states are being encountered. If one reduces the learning rates too quickly, the algorithm may not have a chance to visit some states enough times to reach an equilibrium. If this happens, the value function for some states can get stuck at an unrealistic level. This effect may become more severe as the number of layers in the CMAC is reduced and there is less generalization taking place. On the other hand, if the learning rates are reduced too slowly, convergence will be slow as well. The particular value of $N_\beta$ chosen for this example was found by trial and error.

[2] The UNH CMAC code also recommends that the number of layers be a power of 2.

| | |
|---|---|
| Memory size | 10000 |
| Number of layers | 16 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_1 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 20$ |

Table 4.8: CMAC Parameters. Same for both $\tilde{Q}$ and $\tilde{\pi}$.

**4.2.4.2.3 Controlling Exploration** There are four basic ways in which we can control how our algorithm explores the state space of our system looking for the optimal solution. We can use our insight into the system to choose an initial policy. Given a policy, we can choose the frequency with which to take experimental actions departing from that policy. We can influence the way in which policies are updated through our choice of initial weights for our Q-function approximator. Finally, we can control the starting point of each trajectory in the learning process. The parameters described below control these options.

**Number of policy initialization trajectories** ($N_0$) If an initial policy guess is available, we may want to run the learning algorithm through some number of initial trajectories in order to train $\tilde{\pi}$ to approximate the policy and initialize $\tilde{Q}$ with information about the policy's cost. During these initial trajectories the exploration rate $\epsilon$ is set to zero. In our sample problem no initial policy guess was available, so $N_0$ was set to zero.

**Initial weights for** $\tilde{\pi}$ The choice of the initial weights for the policy approximator can have a significant effect on the learning process. When a local approximator is being used, this initial choice sets up a default action to be taken when new states are encountered. Depending on how this default is chosen , it could have either a stabilizing or destabilizing effect on exploration.

Suppose, for instance, that at some point in the learning process most of the states which have been encountered are near the steady-state. From the steady-state, with probability $1 - \epsilon$ the steady-state action will be chosen and the next state will be the steady-state as well. However, with probability $\epsilon$ a random action will be chosen. Suppose that random action moves the state to a higher variance for which the policy has not yet been updated. If the default action is $a = 1$, then the next state and those that follow will be back down in the range of the steady-state. However, if the default action is $a = 0$, the succeeding states may get further and further from the steady-state. This may or may not be a desirable behavior. In our example, the initial policy weights were chosen to yield a default action $a = 1$. With an action space $\mathcal{A} = [0, 1]$ and an action scale of 100, this means that our initial policy weights were all set to 100.

**Exploration rate** ($\epsilon$) Once the learning process has begun, the exploration rate $\epsilon$ has the most direct effect on the exploration vs. exploitation trade-off. For our test case we chose a relatively high value of $\epsilon = 0.2$ because of the trajectories' rapid convergence to steady-state. Without a reasonably high $\epsilon$ virtually no exploration would occur.

$\epsilon$ **change rate** ($N_\epsilon$) If Q-Learning is to be used, $\epsilon$ can be maintained at a constant level throughout the learning process. But if we are using SARSA($\lambda$) we are continually learning the cost of $\pi_\epsilon$ instead of learning the cost of $\pi$, so we must gradually decrease $\epsilon$ to get close to the optimal policy, $\pi^*$. In our algorithm we do this by maintaining a vector of decreasing $\epsilon$ values and switching to the next every $N_\epsilon$ iterations. In this particular case, however, convergence is rapid enough so that a decreasing $\epsilon$ was deemed unnecessary. $N_\epsilon$ was set to 1000, substantially higher than the total number of iterations.

**Initial weights for** $\tilde{Q}$ The initial weights used in the Q-function approximator can have a great effect on the amount of exploration that takes place and the speed of convergence. Take, for instance, the case where the weights are set so that $\tilde{Q}$ is zero everywhere. When this happens, state-action pairs that have not been updated will always appear to have a lower cost than those that have been updated. Then, if a one-step policy update is being used, each new policy will attempt something new until all the possibilities have been updated. If a two-step policy update is being used, each new policy will favor actions which generate successor states that have not been encountered, as these will have lower Q-values than those that have. In contrast, if the weights are set so that $\tilde{Q}$ is everywhere higher than the optimal value the tendency will be to only take the exploratory actions dictated by the random effect of the exploration rate $\epsilon$. In this example, the latter strategy was used. Of course, without knowledge of the optimal cost it is difficult to pick a number that is everywhere higher than the optimal cost and yet not so high as to generate scaling problems. As with other scaling issues, this was dealt with using an iterative trial-and-error process.

**Initial filter states** ($P_0$) The choice of the initial conditions for each trajectory has an obvious effect on the set of states that are visited in the course of the learning process. In the example problem, the same value $P_0 = 7$ was used for every trajectory, resulting in the policy shown in Figure 4-10. If a range of values was used, a smoother policy could have been generated.

| | |
|---|---|
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Estimation error variance | $P_0 = 7$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 100 |
| Weights for $\tilde{\pi}$ approximator | All weights initialized to 100 |
| Exploration rate | $\epsilon = 0.2$ |
| $\epsilon$ change rate ($N_\epsilon$) | N/A |

Table 4.9: Parameters influencing exploration.

#### 4.2.4.2.4 Algorithm Parameters

**Eligibility decay rate** ($\lambda$) This is the $\lambda$ of SARSA($\lambda$). Experience of various authors has suggested that values of $\lambda$ between 0.6 and 0.9 work best, though there has been no clear explanation of why this is so. The choice for this example, $\lambda = 0.8$, was made rather arbitrarily.

**Eligibility cutoff** ($\kappa$) In the SARSA($\lambda$) algorithm each temporal difference $\delta$ is propagated backward to all the previous steps in the trajectory, with its effect reduced by a factor of $\gamma\lambda$ with each step. Eventually, this decay factor $(\gamma\lambda)^i$ becomes so small that it can be safely ignored and continued propagation becomes computationally inefficient. In our algorithm, temporal differences are not propagated beyond the point where $(\gamma\lambda)^i < \kappa$. The exact number used, $\kappa = 0.02$, was chosen arbitrarily.

**Number of iterations** ($N_{eg}$) The number of trajectories which must be run in order to achieve satisfactory convergence may vary widely from case to case. An appropriate number can only be arrived at through trial and error. Of course, a variety of other conditions can be used to terminate learning; termination after a fixed number of iterations was chosen merely for its simplicity.

**Number of steps/trajectory** ($N$) For finite horizon problems, choosing the number of steps per trajectory is a non-issue. For infinite horizon problems, however, a variety of factors need to be balanced. Given that we have chosen to conduct a policy update after each trajectory, the length of the trajectories determines the frequency of the policy updates. Given that we know most policies will reach steady-state in a few steps, the number of steps per trajectory determines the relative importance we are giving the steady-state versus the transient. Combined with our choice of $\epsilon$, it determines how much exploration around the steady-state will take place between policy updates. The trajectory length also determines the effectiveness with which the trajectory's initial conditions can be used for exploration.

**Policy update ratio** ($\eta$) As described in Section 2.2.6, we may not want to update our policy at every point in a trajectory. In each trajectory, only $\eta N$ of the $N$ samples, selected randomly, are used to update the policy. Generally, to make the use of a function approximator worthwhile we would want $\eta \ll 1$. In this particular case, however, we know that trajectories rapidly go to a steady-state. Since we have specified that repeated states in a single trajectory are not updated, we can leave $\eta = 1$ secure in the knowledge that only a few states will be updated from each trajectory. For example, if $N = 40$ and $\eta = 1$, 40 points will be selected for updating. However, the transient portion of the trajectory may only last 3 steps, with the remaining 37 ending up at the steady-state value. Thus, only 4 of the 40 points would actually get updated, and we could be sure of catching all the transient states. If we had chosen $\eta = 0.25$, we might have only caught one of the transient states. When we move to nonlinear examples, our choice of $\eta$ will be substantially different.

| | |
|---|---|
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.8$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 100$ |
| Number of steps/trajectory | $N = 40$ |
| Policy update ratio | $\eta = 1$ |

Table 4.10: Algorithm-related parameters.

### 4.2.5 Conclusion

As we noted at the beginning of the chapter, the linear measurement scheduling problem can take a variety of forms. The problem examined in this section was just one – and a relatively simple one at that – out of many possible variations on the same basic theme; three others will be presented in the sections to follow. Those sections will be considably more compact than this one has been. The extraordinary length of this section was due to its dual role as example and tutorial.

Because of the length of the section and the large amount of detail, the overall context of the problem may have faded into the background. The things that should be taken away from this example are not the details of which action is best in a particular state or what parameter setting should be used in a particular setting. What is important is an understanding of the issues involved and the processes used to resolve them. So before moving on to the next example, we will take a moment to look at the big picture.

For this particular problem, the central fact of importance was the dominance of the steady-state. Any reasonably good policy would generate a trajectory that moved to roughly the same steady-state value in just a few steps. This imparted a profound influence to the function approximation involved, to the learning process itself, and to the analysis of the algorithms' performance. We could have continued to fill page after page with parameter trade studies and comparisons of different algorithm variants, but these details would be of little use in addressing other problems.

What *is* transferable is the understanding of the differences between grid-based updating methods and trajectory-based methods. Insights into the particularities of different function approximators – MLP's, polynomial curve fits, or CMAC's – may also prove generally useful. And understanding the role that scaling plays in combining function approximation with learning algorithms is vitally important to anyone who intends to make practical use of reinforcement learning. The large variety of possible reinforcement learning algorithms and the cornucopia of problems to which they can be applied make it essential to understand the individual algorithm components and how to combine them to fit the problem at hand.

Hopefully, the material in this section and in the remaining portions of this thesis will help provide that knowledge.

## 4.3 Discrete Action Space, Infinite Horizon

In this second example, we look at the case where we are forced to choose from a small number of possible actions. In terms of the previous example, this could represent a siutation where the radar's control system only allows pulses at three or four discrete power levels. Alternatively, it might represent a situation where the sensors have no flexibility, and the choice at each time step becomes whether or not to take a measurement at all.

As a testbed, we have used a hypothetical microsatellite with two subsystems available for determining its orientation. One is a suite of onboard gyros with relatively low accuracy but low power consumption. These gyros would provide angular velocity measurements.

The other measurement subsystem is based on differential GPS. This system is extremely accurate, but has a high power consumption in terms of the satellite's limited power budget. Also, the satellite's electric propulsion system interferes with the GPS signal reception, so it must be turned off while GPS measurements are being taken. These two factors combine to put an effective cost on each GPS measurement that is taken. Possible actions include taking measurements with GPS, gyros, both, or neither.

Though the scenario as a whole may not be particularly realistic and the cost factors have been chosen only with an eye toward providing interesting results, the numbers used for the linearized satellite dynamics and assorted noise statistics are taken from actual work being done on a proposed satellite at Draper Laboratory. So one might call it a semi-realistic example. The main point, though, is to demonstrate the ability of the RL algorithm to solve relatively high dimensional linear problems where only a limited number of actions are possible.

### 4.3.1 Formulation

Most of the formulation for this example is the same as for the previous one. The big difference is that instead of $\mathcal{A} = [0,1]$, we have a discrete set $\mathcal{A} = [\alpha_1, \alpha_2, \ldots \alpha_N]$, where there are $N$ possible actions. As a consequence of the discrete nature of the domain, sensor noise covariances are determined using a discrete mapping

$$V_k = R\left(a_k\right) \quad \left(R : \mathcal{R} \rightarrow \mathcal{R}^{l \times l}\right) \tag{4.22}$$

rather than a continuous function.

Performance, as in the previous section, is measured by a combination of the post-measurement-update error covariance and the cost of the chosen action. The latter quantity is again determined by a discrete mapping instead of a smooth function, so that we have

$$g\left(y_k, \zeta_k^-, a_k\right) = \mathrm{Tr}\left(\mathrm{LP_k^+}\right) + \mathrm{M}\left(\mathrm{a_k}\right) \tag{4.23}$$

| Block | Contents | Notes |
|---|---|---|
| $\mathcal{A}$ | $\{\alpha_1, \alpha_2, \ldots \alpha_N\}$ | $V_k = R(a_k)$ $R : \mathcal{A} \rightarrow \mathcal{R}^{l \times l}$ |
| $\Gamma\left(\zeta_k^-, \varsigma_k\right)$ | $\theta_k = \left[P_k^-(1,1), \ldots P_k^-(1,n), P_k^-(2,2) \ldots P_k^-(2,n), \ldots P_k^-(n,n)\right]$ | |
| $g\left(\zeta_k^-, a_k, y_k\right)$ | $c_k = \mathrm{Tr}\left[\mathrm{LP}_\mathrm{k}^+\right] + \mathrm{M}(\mathrm{a}_\mathrm{k})$ | $L \in \mathcal{R}^{n \times n}$ $M : \mathcal{A} \rightarrow \mathcal{R}$ |
| $J^\pi\left(\zeta_k^-\right)$ | $\sum_{i=0}^{\infty} \gamma^i c_{k+i}$ | $a_k = \pi(\theta_k)$ |

Table 4.11: Elements of the discrete action space, infinite horizon formulation.

instead of (4.11). The details of this formulation are summarized in Table 4.11.

### 4.3.2 Experiments

With any of the formulations that are presented in this chapter there are a virtually limitless number of experiments that could be conducted to help illuminate characteristics of both the problem and the solution method. Due to the limited scope of this thesis and a desire to avoid boring the reader to death with an endless succession of graphs, only two problems will be discussed in this section. The treatment of these problems is intended to showcase the methodologies involved and describe some of the aspects of the solution, not to extract the best possible performance from the algorithm.

The first problem that will be treated is a simple on-off case. The only possible action choices will be to use the GPS system or not take any measurement at all. Based on the analysis from Section 4.2.2 we might expect to see a solution which tends to a limit-cycle about a steady-state value, and this is precisely what we get. The simplicity of this test case allows a rough test of optimality which helps to verify that our algorithm is actually working.

Having established the functionality of our algorithm with the above test case, we will proceed to a slightly more difficult problem. Instead of allowing only two options for measurement, we allow four. Measurements may be taken using GPS and gyros, GPS alone, gyros alone, or the measurement opportunity may be passed up entirely. A cost is associated with the use of either instrument, though the particular numbers used in the experiment were chosen without any physical basis. In this case, and with more complex action spaces, the form of the optimal solution is less obvious. Evaluation of the learning algorithm's performance relative to the optimal solution becomes more difficult.

In Section 4.3.2.1 we will describe the system parameters defining the two problems. Next, in Section 4.3.2.2 we will present the algorithm parameters and discuss the reasons for the choices that were made. Finally, in Section 4.3.2.3 we will describe the results of running the algorithm with the specified parameters on the two problems.

#### 4.3.2.1 System Parameters

The linear model in Table 4.12 describes the attitude dynamics of a small satellite. Of the model's six states, the first three are the roll, pitch, and yaw angles and are measured by the differential GPS system. The second three are the roll, pitch, and yaw angular rates, and are measured by the gyros. Angles are measured in radians, and rates in radians/second.

The measurement options for the two test cases are shown in Table 4.13. In the slots where no measurement is being taken the sensor noise should be effectively infinite; $10^6$ was chosen arbitrarily to fill this role.

#### 4.3.2.2 Algorithm Choices

The algorithm parameters used in the two test cases are shown in Table 4.14. Many of these require no explanation given the discussion in Section 4.2.4.2. However, some do deserve a bit of discussion.

First off, the choice of function approximation architecture should be explained. Though CMAC's are used for function approximation in almost every example presented in this thesis, the first experiments with this particular problem formulation were conducted using MLP's instead. Performance of the learning algorithm using MLP's was unreliable, however. Two consecutive runs using the same parameter settings could produce vastly different

| Linear system and noise parameters |
|---|
| $A = \begin{bmatrix} 1 & 0 & 0.0011 & 1 & 0 & 9.22e-05 \\ 5.54e-08 & 1 & -5.53e-08 & 2.77e-08 & 1 & -2.77e-08 \\ -0.0011 & 0 & 1 & -0.0006 & 0 & 1 \\ -3.06e-06 & 0 & -1.69e-09 & 1 & 0 & -0.0009 \\ -8.47e-14 & -3.06e-06 & 8.47e-14 & -2.82e-14 & 1 & 2.82e-14 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ |
| $C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad V = \begin{bmatrix} \sigma_1^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_4^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_5^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_6^2 \end{bmatrix}$ |
| $W = \begin{bmatrix} 2.78e-08 & 5.11e-16 & 5.12e-11 & 5.56e-08 & -3.91e-22 & 1.23e-10 \\ 5.11e-16 & 2.78e-08 & -1.84e-14 & 1.04e-15 & 5.56e-08 & -3.69e-14 \\ 5.12e-11 & -1.84e-14 & 1e-06 & -9.43e-10 & 1.41e-20 & 2e-06 \\ 5.56e-08 & 1.04e-15 & -9.43e-10 & 1.11e-07 & -7.98e-22 & -1.84e-09 \\ -3.91e-22 & 5.56e-08 & 1.41e-20 & -7.98e-22 & 1.11e-07 & 2.82e-20 \\ 1.23e-10 & -3.69e-14 & 2e-06 & -1.84e-09 & 2.82e-20 & 4e-06 \end{bmatrix}$ |

Table 4.12: System parameters for test case based on linearized satellite dynamics. No external input was used, so $B$ and $D$ are not considered. Values of $\sigma_1^2 \ldots \sigma_6^2$ determined by $a_k$.

| | Case 1 | | Case 2 | | | |
|---|---|---|---|---|---|---|
| Action | 1 | 2 | 1 | 2 | 3 | 4 |
| Cost | $M_{gps}$ | 0 | $M_{gps} + M_{gyros}$ | $M_{gps}$ | $M_{gyros}$ | 0 |
| $\sigma_1$ | 2.388e-5 | 1e6 | 2.388e-5 | 2.388e-5 | 1e6 | 1e6 |
| $\sigma_2$ | 2.388e-5 | 1e6 | 2.388e-5 | 2.388e-5 | 1e6 | 1e6 |
| $\sigma_3$ | 2.388e-5 | 1e6 | 2.388e-5 | 2.388e-5 | 1e6 | 1e6 |
| $\sigma_4$ | 1e6 | 1e6 | 5.29e-8 | 1e6 | 5.29e-8 | 1e6 |
| $\sigma_5$ | 1e6 | 1e6 | 2.50e-9 | 1e6 | 2.50e-9 | 1e6 |
| $\sigma_6$ | 1e6 | 1e6 | 1.00e-8 | 1e6 | 1.00e-8 | 1e6 |
| $M_{gps} = 0.001$ | | | | | $M_{gyros} = 0.0002$ | |

Table 4.13: Sensor noise values and costs for each action allowed in the two test cases.

results. It seems likely that this was the result of randomly selected initial weights generating encounters with various local minima during the training process. MLP's are trained by gradient descent on the squared error between the current function ($\tilde{Q}$) and the desired values ($\hat{Q}$); the local minima we are referring to are minima of this error surface, not of the Q-function itself. The decision to use CMAC's (which do not suffer from local minima during training) as the primary approximation architecture throughout the thesis was made in response to this issue.

The next issue that deserves comment is the selection of scaling factors. The principles involved are the same for this case as those set out in Section 4.2.4.2, but there are some complicating factors. Instead of a scalar $\theta$, we now have a feature vector with 21 elements representing the unique elements of the covariance matrix. Now, we could go through the standard iterative process for each element in the feature vector, but in this case that may not be the best thing to do.

This is because 15 of the 21 elements of the feature vector correspond to off-diagonal elements in a covariance matrix. The physical meaning of these elements comes from their

| Scaling | |
|---|---|
| State ($\theta$) | $10^6[$ 1 1 1 10 10 2.24 1 ...<br>1 10 10 2.24 1 10 10 ...<br>2.24 100 100 22.4 100 22.4 5 ] |
| Action ($a$) | 100 |
| Cost ($Q$) | 1 |
| CMAC-related parameters | |
| Memory size | 50000 |
| Number of layers | 8 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_1 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 2000$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Estimation error variance | $P_0 = 10^{-3} I_{6\times 6}$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 0 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 5000$ |
| Number of steps/trajectory | $N = 500$ |
| Exploration rate | $\epsilon = [0.1 \quad 0.02 \quad 0.01 \quad 0.005 \quad 0.001]$ |
| Rate of reduction for exploration rate | $N_\epsilon = 1000$ |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.9$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |

Table 4.14: Algorithm parameter settings for satellite test cases.

relationship with the diagonal elements. Scaling them independently would destroy that relationship. To preserve the relationship between the diagonal and off-diagonal elements, we choose scaling factors ($\rho$) for the diagonal elements using the iterative method described earlier and then the remaining factors are set relative to their corresponding diagonal factors:

$$\begin{bmatrix} \rho_1 & \rho_2 & \rho_3 & \rho_4 & \rho_5 & \rho_6 \\ & \rho_7 & \rho_8 & \rho_9 & \rho_{10} & \rho_{11} \\ & & \rho_{12} & \rho_{13} & \rho_{14} & \rho_{15} \\ & & & \rho_{16} & \rho_{17} & \rho_{18} \\ & & & & \rho_{19} & \rho_{20} \\ & & & & & \rho_{21} \end{bmatrix} = \begin{bmatrix} \rho_1 & \sqrt{\rho_1\rho_7} & \sqrt{\rho_1\rho_{12}} & \sqrt{\rho_1\rho_{16}} & \sqrt{\rho_1\rho_{19}} & \sqrt{\rho_1\rho_{21}} \\ & \rho_7 & \sqrt{\rho_7\rho_{12}} & \sqrt{\rho_7\rho_{16}} & \sqrt{\rho_7\rho_{19}} & \sqrt{\rho_7\rho_{21}} \\ & & \rho_{12} & \sqrt{\rho_{12}\rho_{16}} & \sqrt{\rho_{12}\rho_{19}} & \sqrt{\rho_{12}\rho_{21}} \\ & & & \rho_{16} & \sqrt{\rho_{16}\rho_{19}} & \sqrt{\rho_{16}\rho_{21}} \\ & & & & \rho_{19} & \sqrt{\rho_{19}\rho_{21}} \\ & & & & & \rho_{21} \end{bmatrix}.$$

A histogram shows the final distributions of the diagonal elements in Figure 4-13.

The scaling chosen for the action also deserves some comment. A large value (100) was chosen in order to *avoid* generalization between actions. Since the action values are simply arbitrarily assigned indices, adjacent actions don't necessarily have any relationship to each other, and so generalization along the action dimension is inappropriate.

Another parameter choice which may at first seem puzzling is the initialization of the $\tilde{Q}$ weight vector to zero. This is a dramatic change from the values used in Section 4.2.4. In that section's continuous-action example, the initial weights were set at a high level in order to avoid excessive generalization. When the initial weights of the approximator are set to zero, the tendency will be to try each possible action at least once before settling on any one as being the best. This is generally an undesirable behavior when there are many possible actions, but may well be desirable when there are only a few. In this section's cases, there are only 2-4 possible actions in each state, so it seemed reasonable to go the route of thorough exploration and set the initial weights to zero.

#### 4.3.2.3 Results

Having set all the relevant parameters, the algorithm was run on the two-action case. The algorithm's performance is shown in Figure 4-14, and exhibits a nice steady improvement

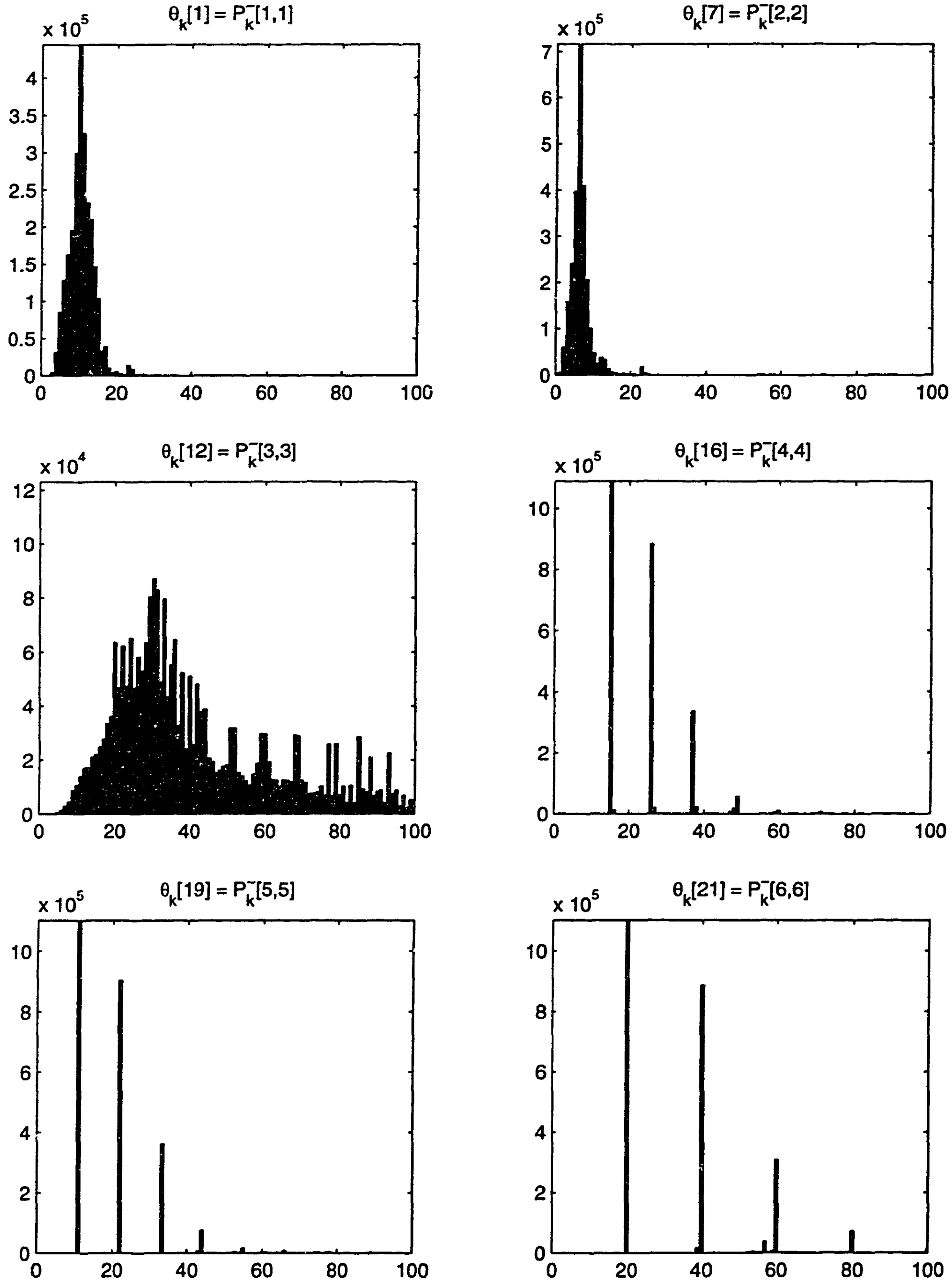


Figure 4-13: Histograms showing distribution of the scaled $\theta$ values matching the diagonal elements of $P$ for states encountered over the entire learning process. Data taken from the 4-action example.

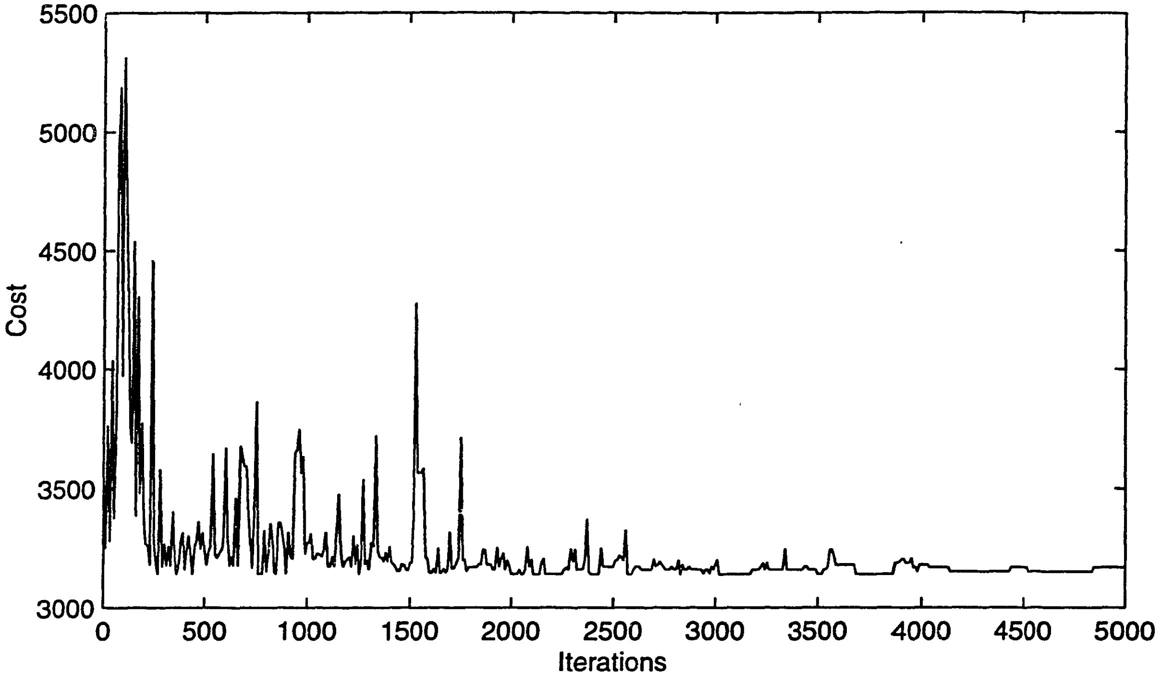


Figure 4-14: Performance of the learning process for the two-action case.

and convergence toward a single policy. The question, of course, is whether that ultimate policy compares favorably with the optimal.

Not knowing the true optimal policy, we settle for comparison with a series of heuristic policies. Figure 4-15 shows the last steps in a trajectory following the final learned policy. A clear pattern of one measurement every five steps is visible. It seems obvious to compare this policy with similar policies which measure at fixed intervals.

Of course, due to the initial transient the learned policy does substantially better than any such policy. A comparison at the steady-state is more revealing. Figure 4-16 shows the average single-step cost over 400 steps for open-loop policies which measure at a range of fixed intervals. The jaggedness of the plot comes from the discretization effects. From the plot, the best ratio seems to be one measurement in 5 steps, precisely the policy that the learning algorithm comes up with. Ratios of 4.6 and 4.8 also seem to do quite well, and in one run using slightly different algorithm parameters a policy was learned that corresponded to the former. This shows that the learning algorithm is capable of finding policies more

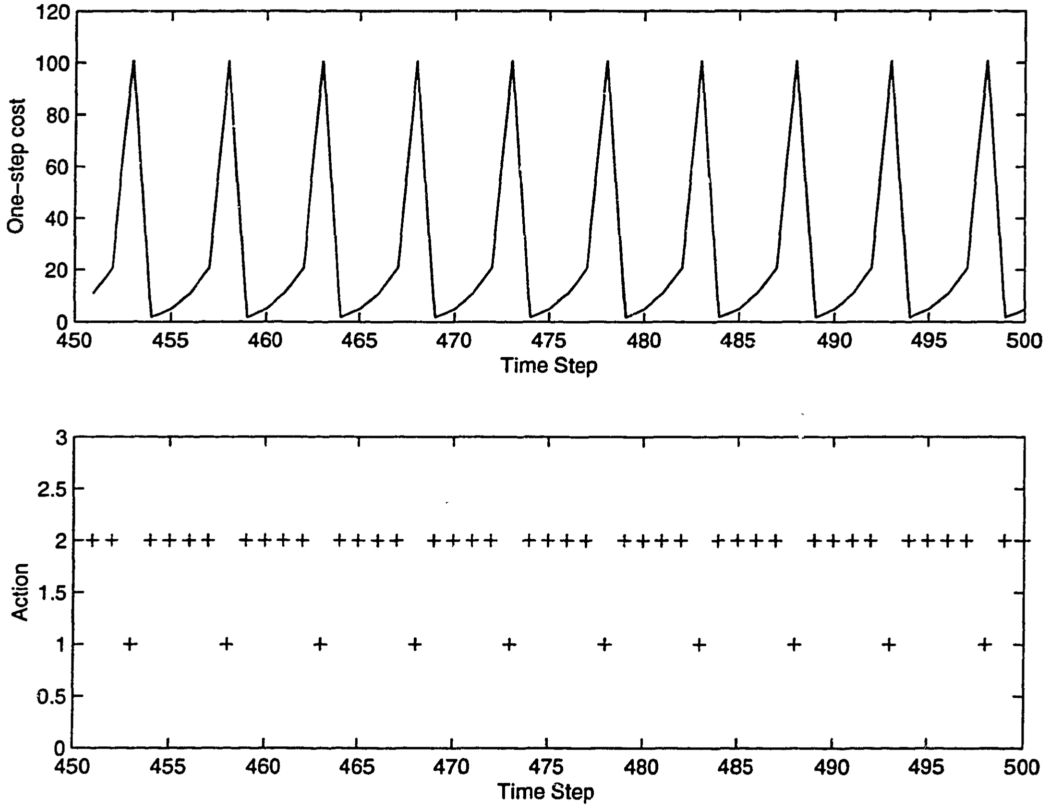


Figure 4-15: Final learned policy for the two-action case. Only the last 50 steps of a 500 step trajectory are plotted. The top graph shows the single-step costs, while the bottom graph shows the action chosen at each step. Action 1 is a GPS measurement; Action 2 is no measurement.

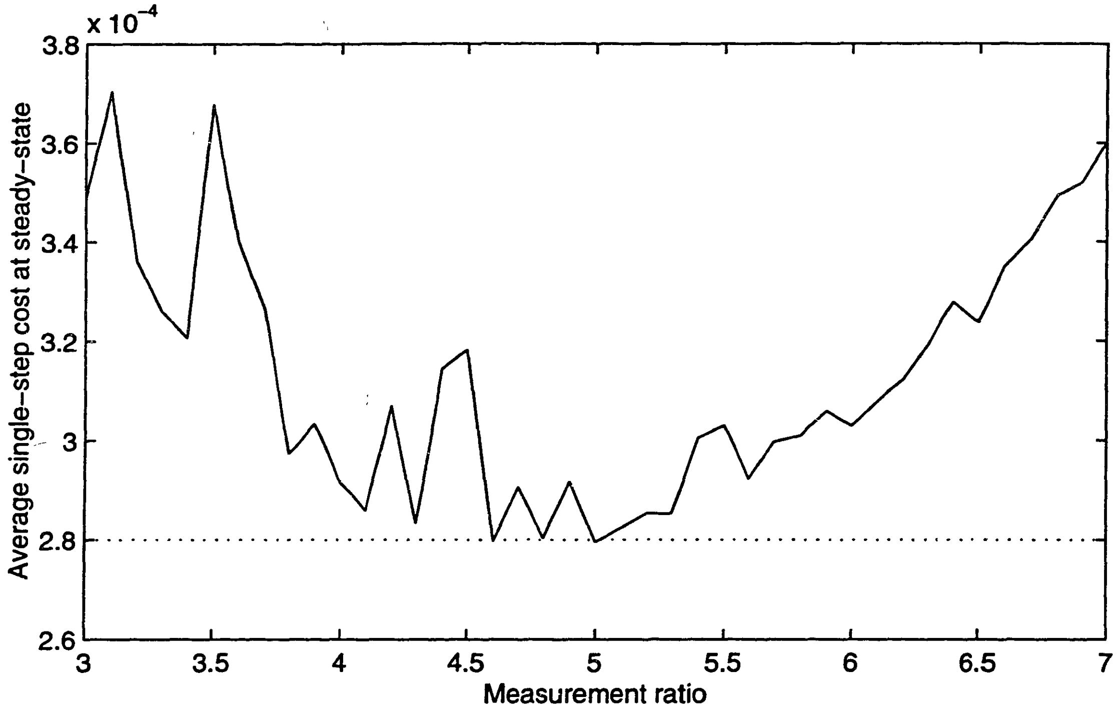


Figure 4-16: Comparison of the costs of various open-loop heuristic policies. Each point is the result of a test trajectory where actions 1 and 2 were taken in the specified ratio. The average single-step cost was then taken over the final 400 (out of 500) points. The dotted line shows the corresponding mean for the final learned policy.

complex than simple alternation in integer ratios.

The results for the two-action example, while good, are not overly impressive. After all, an open-loop policy that performs as well at steady-state can be found by a relatively simple search. The important thing to take from this example is that the algorithm *does* manage to find this policy. With this success, we can be more confident of finding a near-optimal solution when the action space is more complex.

With four possible actions, judging our algorithm's success may become more difficult. The exact form of the optimal solution will depend on the combination of system and cost parameters, but there are two main possibilities[3]. One is an alternation between two of the four possible actions much as in the previous example. Though it is hard to predict if this

[3] When we discuss the optimal solution in this section we will generally be referring to the steady-state behavior of the optimal solution. The initial transient is much harder to characterize.

will occur beforehand, if the learning algorithm produces such a policy it is relatively easy to check it against a range of similar policies.

The other possibility is a more complex pattern involving three or more of the available actions. The measurement costs specified in Table 4.13 were chosen with an eye toward generating this type of situation, and the resulting learned policy is shown in Figure 4-17. Though this policy is fairly complex, it is not too hard to see what it is doing. It takes the cheaper gyro measurements at regular intervals with a gradually degrading estimation accuracy. Every so often it becomes necessary to use one of the more expensive GPS measurements to correct for this gradual accumulation of error. In the few steps leading up to each GPS measurement a small cost savings can be achieved by reducing the frequency of the gyro measurements, secure in the knowledge that the additional errors will soon be corrected. The result is a cost about 5% lower than we were able to find using a heuristic of our own.

The policy shown in Figure 4-17 would be difficult to come up with heuristically. Looking at the top graph one might think that the policy could be characterized with switches based on the trace of the covariance. One could imagine something like:

$$\pi(\theta_k) = \begin{cases} 2 & \left(a \leq \mathrm{Tr}\left(\mathrm{P}_\mathrm{k}^-\right) < \mathrm{b}\right) \\ 3 & \left(b \leq \mathrm{Tr}\left(\mathrm{P}_\mathrm{k}^-\right) < \mathrm{c}\right) \\ 4 & \left(c \leq \mathrm{Tr}\left(\mathrm{P}_\mathrm{k}^-\right) \leq \mathrm{d}\right) \end{cases}$$

where the parameters $a$, $b$, $c$, and $d$ could be found with a simple search. A look at the bottom graph in the figure indicates that something more complex than this is going on, however. The learning algorithm must be making use of more information than just the trace of the covariance matrix.

Though the above two sample cases provide promising results, a few caveats are in order. First, it seems that the learning algorithm was not good at finding alternating policies with large ratios. For instance, if the best heuristic was to measure once out of every 100 timesteps, the learning algorithm would be unlikely to find it, at least with the parameter settings described above. To a degree, this makes physical sense. If the optimal solution is

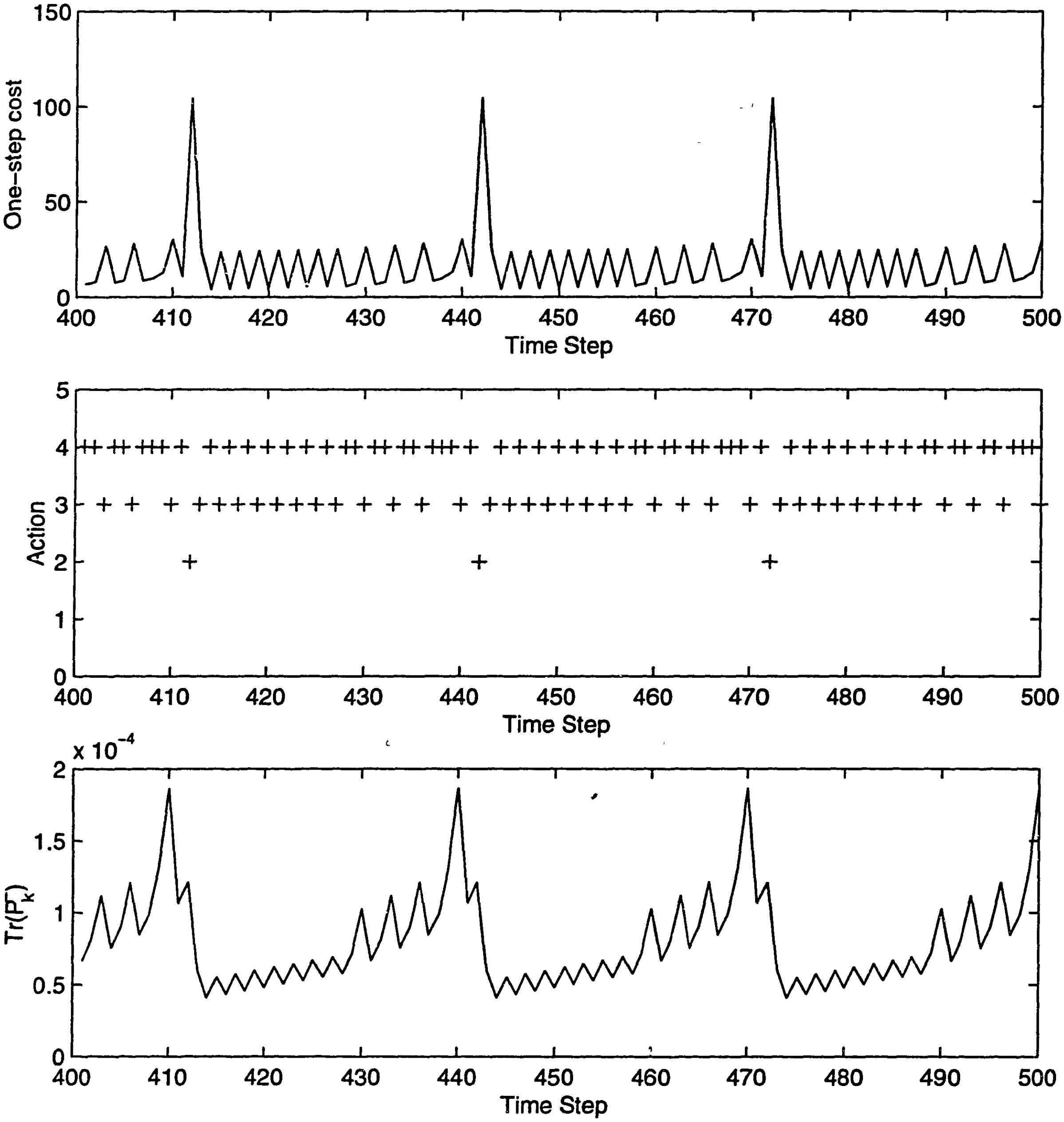


Figure 4-17: Final learned policy for the four-action case. The top graph shows the single-step costs, while the middle graph shows the action chosen at each step. Action 1 is a combined GPS and gyro measurement; Action 2 is GPS alone; Action 3 is gyros alone, and Action 4 is no measurement. The bottom graph shows the trace of the error covariance at each step.

to measure every 10 seconds, it doesn't make the decision as to whether or not to measure every microsecond. Changing the time scales involved should fix this problem, but one must remember to exercise care when formulating the problem.

Another thing to watch out for seems to be sensitivity to scaling issues. One may notice in Figure 4-13 that the first two histograms only show values varying roughly in the range 0-25. In order to provide a larger range for these values, the scaling on these two features was increased by a factor of 10, and the algorithm run again. Unexpectedly, the best policy found after making this change was 3-4% worse, and substantially different in character. An increase in the scaling of only a factor of 2 did better, but still worse than with the original scaling. It seems likely that these differences have to do with the amount of generalization in the $\tilde{Q}$ function approximator, but no reliable method is known for predicting what level of scaling will work best ahead of time. We are left with a rather unsatisfying reliance on trial and error for choosing scaling values.

## 4.4 Continuous Action Space, Finite Horizon

### 4.4.1 Formulation

The problem formulation presented in this section is adapted from the formulations appearing in Avitzour and Rogers [6] and Shakeri et. al. [51]. Put simply, the objective is to minimize the estimation error variance at a specific time given a fixed measurement budget.

As in Section 4.2, the decision variable $a_k$ represents the energy put into the measurement, so that

$$V_k = \frac{1}{a_k} R.$$

In this case, however, we have the added restriction that the sum of the actions over the specified horizon must be equal to the budget,

$$\sum_{k=0}^{N} a_k = M$$

. To keep track of how much of the budget has been used at any given point in time, we add a new variable $\mu$ to the filter state,

$$\mu_k = M - \sum_{i=0}^{k-1} a_i.$$

Our continuous, complex action space can then be expressed as $\mathcal{A}_k = [0, \mu_k]$. Note that in this case, the action space depends on the filter state through $\mu_k$, where in previous examples it was held constant.

Clearly, the cost of any filter state in this system will depend in part on the remaining budget, so $\mu_k$ must be added to the feature vector $\theta$. The switch to a finite horizon formulation also requires that time be taken into account, so our feature vector becomes

$$\theta_k = \left[P_k^-(1,1), \ldots P_k^-(1,n), P_k^-(2,2) \ldots P_k^-(2,n), \ldots P_k^-(n,n), \mu_k, t_k\right].$$

The single-step cost for this problem differs substantially from the formulations for the infinite-horizon problems. No explicit cost is put on the action chosen; the cost is instead specified indirectly through the restriction that the budget places on the action space. The explicitly stated cost depends only on the trace of the error covariance at the end of the specified horizon:

$$c_k = \begin{cases} 0 & k < N \\ \mathrm{Tr}\left[\mathrm{LP}_{\mathrm{k+1}}^-\right] & k = N \end{cases}.$$

Note that $P_{k+1}^-$ is used here, not $P_k^+$ as in the previous examples. This is done to match the formulations from the literature. As before, the $L$ matrix allows for the relative weighting of the states in the covariance matrix. The details of the formulation are summarized in Table 4.15.

### 4.4.2 Examples

Though Avitzour's problem fits easily enough into the general measurement with cost formulation developed in Chapter 2, it turns out that reinforcement learning is not well suited

| Block | Contents | Notes |
|---|---|---|
| $\mathcal{A}_k$ | $[0, \mu_k]$ | $V_k = \frac{1}{a_k} R \qquad R \in \mathcal{R}^{l \times l}$<br>$\mu_k = M - \sum_{i=0}^{k} a_i$ |
| $\Gamma\left(\zeta_k^-, \varsigma_k\right)$ | $\theta_k = \left[P_k^-(1,1), \ldots P_k^-(1,n), P_k^-(2,2) \ldots P_k^-(2,n), \ldots P_k^-(n,n), \mu_k, t_k\right]$ | |
| $g\left(\zeta_k^-, a_k, y_k\right)$ | $c_k = \begin{cases} 0 & k < N \\ \mathrm{Tr}\left[\mathrm{LP}_{\mathrm{k+1}}^-\right] & k = N \end{cases}$ | $L \in \mathcal{R}^{n \times n}$ |
| $J^\pi\left(\zeta_k^-\right)$ | $\sum_{i=0}^{N} c_i$ | $a_k = \pi(\theta_k)$ |

Table 4.15: Elements of the continuous action space, finite horizon formulation.

to the problem's solution.

It has been shown by Shakeri [51] that the problem is convex. With a convex problem, an algorithm based on guided random exploration such as reinforcement learning is unlikely to be able to compete with a more focused method such as the projected Newton's method which Shakeri uses. In general, reinforcement learning with function approximation will find near-optimal solutions, while convex problems can usually be solved exactly.

An effort was made to apply RL to Avitzour's sample problem anyway, and is outlined below. A degree of success was obtained only in the case where $N = 5$, compared to the more difficult $N = 15$ used by Avitzour and the $N = 50$ used by Shakeri on the same problem. These results will be explained in more detail in Section 4.4.2.3.

#### 4.4.2.1 System Parameters

The system parameters used were those from Avitzour's simple example of linear motion subject to a random acceleration with a scalar position measurement. They are shown in Table 4.16.

| | | |
|---|---|---|
| Linear System: | $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ | $C = \begin{bmatrix} 1 & 0 \end{bmatrix}$ |
| Noise related: | $W = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ | $R = 1$ |
| Budget: | 8.0 | |

Table 4.16: Parameters for Avitzour's example.

#### 4.4.2.2 Algorithm Choices

The algorithm parameters used are shown in Table 4.17. The scaling on the covariance-related features was kept small, as it was known that the solution was open loop (hence the time and budget related features would be more important). Larger scaling on these parameters was also tried, with little difference in performance. Scaling on the feature which keeps track of the time was kept large so that there would be no generalization between timesteps. The action scaling was determined by the desired action resolution (0.1). The cost scaling was chosen using an iterative process: the algorithm was run once, the cost of the best solution found examined, and then the scaling adjusted to provide a reasonable level of precision in comparing policies.

No initial policy guess was used by Avitzour or Shakeri, so none was used in the learning algorithm. The initial covariance was specified in Avitzour's paper. Initial weights for $\tilde{Q}$ were set to be slightly higher than optimal cost in an effort to balance exploration and exploitation. Considerable trial and error was used in setting this value. Initial weights for $\tilde{\pi}$ were set to zero, giving a default action of no measurement, based on the assumption that the optimal policy would use all of the budget in just a few steps.

| | |
|---|---|
| Scaling | |
| State ($\theta$) | [1 1 1 10 10] |
| Action ($a$) | 10 |
| Cost ($Q$) | 100 |
| CMAC-related parameters (same for $\tilde{Q}$ and $\tilde{\pi}$) | |
| Memory size | 50,000 |
| Number of layers | 8 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_1 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 20,000$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Estimation error variance | $P_0 = 1,000I_{2\times 2}$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 200 |
| Weights for $\tilde{\pi}$ approximator | All weights initialized to 0 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 200,000$ |
| Number of steps/trajectory | $N = 5$ |
| Exploration rate | $\epsilon =$ logspace(log10(.2),-4,20)[4] |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.99$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |
| Fraction of samples updated after each trajectory | $\eta = 0.2$ |

Table 4.17: Parameter settings for testing of Avitzour's sample problem.

|  | Optimal | RL (best) | RL (final) |
|---|---|---|---|
| $a_1$ | 0 | 0.1 | 3.6 |
| $a_2$ | 0 | 0 | 0.1 |
| $a_3$ | 2.0 | 2.0 | 0 |
| $a_4$ | 0 | 0 | 2.2 |
| $a_5$ | 6.0 | 5.9 | 2.1 |
| $P_6^-[1,1]$ | 1.75 | 1.7530 | 250.41 |

Table 4.18: Comparative policies for a budget of 8.0 with a 5-step horizon.

#### 4.4.2.3 Results

Even with a horizon of only 5 steps, the reinforcement learning algorithm was unable to find the optimal solution, though it did come fairly close. Table 4.18 shows the results from one run of the algorithm using the parameters described above. Note that when multiplied by the scaling factor of 100 and rounded to the nearest integer, the learning algorithm's solution is indistinguishable in cost from the optimal solution. This would seem to indicate that a larger scale should be used to provide increased resolution, but several runs made using a larger scale actually produced poorer results. Also note that though the best policy encountered during learning is quite close to the optimal, convergence to that policy was not obtained, and the final policy is substantially worse. The costs of both the best policy and the final policy varied significantly from run to run. It is possible that better convergence behavior could have been obtained with different exploration and learning rate schedules, but no superior schedules were found in the course of fairly extensive experimentation. With Avitzour's horizon of 15 steps, the algorithm was unable to find a policy with cost under 200 (175 optimal) in the rather lengthy training period alloted.

Even with a simple two-dimensional system, the internal workings of the learning algorithm are enormously complex, so we will never be able to say with certainty exactly why the algorithm fails with this type of problem. We can, however, make some guesses based on

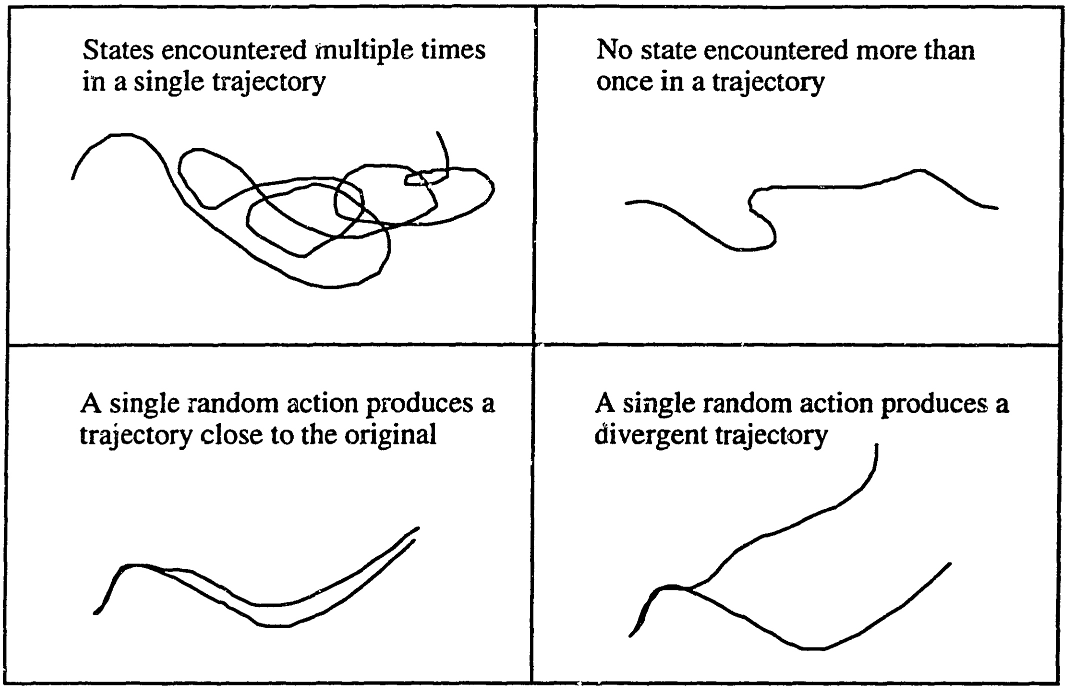

States encountered multiple times in a single trajectory
No state encountered more than once in a trajectory
A single random action produces a trajectory close to the original
A single random action produces a divergent trajectory

Figure 4-18: A comparison of different types of trajectories.

qualitative analysis which may help us recognize systems with similar potential problems.

One aspect of the formulation that may make things difficult is the presence of two monotonic features, the time and remaining budget. This monotonicity means that no state is ever visited twice in the same trajectory, as illustrated in Figure 4-18. The nature of the chosen features also means that the random actions chosen for exploration will generate trajectories that differ dramatically from the trajectory that would be generated by the base policy. Reinforcement learning algorithms learn from experience, however, so these are not desirable features.

More preferable would be the sorts of trajectories shown at the left of the figure. If individual states or even small regions of the state space are visited with a relatively high frequency, we can learn about the costs in those areas and act accordingly. In the previous two infinite horizon formulations trajectories tended toward a steady-state behavior of some sort, so learning focused around a small area in the state space. Excursions out of that area due to random exploratory actions would quickly return. This type of trajectory is well suited to learning. When trajectories are extremely spread out as in this problem, we may not visit any given area in the enormous state space often enough to be able to evaluate its cost realistically. Also, with trajectories that are widely spread through the state space the sheer number of points encountered can rapidly overload the function approximator's ability to cope.

## 4.5 Discrete Action Space, Finite Horizon

To round out our discussion of measurement scheduling for linear systems, we present one final class of problems. Like the problem presented in the previous section, this was taken from the literature in order to show the generality of the formulation we are using and evaluate the utility of the solution approach we have chosen. Also like the problem from the last section, we have here a finite-horizon cost function where time must be considered as one of the decision variables. There is one crucial difference, however. Whereas the previous section dealt with a complex, continuous action space this section examines a simple, discrete action space. This simplification appears to make a great difference, and the performance obtained using our reinforcement learning algorithm is much better.

### 4.5.1 Formulation

The formulation presented in this section is a discrete-time version of the problem examined by Athans [5]. The idea is that various measurement subsystems are available, each potentially coming with an associated cost, but only one measurement can be taken at a time. The measurements taken within a specific interval ($0 < k \leq N_\epsilon$) are used to predict the state at some later timestep $N_T > N_\epsilon$.

So, just as in Section 4.3 we have a discrete action space $\mathcal{A} = [\alpha_1, \alpha_2, \ldots \alpha_N]$, with each $\alpha_i$ mapping to a covariance matrix $R(\alpha_i)$ and a cost $M(\alpha_i)$. As in Section 4.4 we must augment our basic feature vector to include the current time (though in this case we need

| Block | Contents | Notes |
|---|---|---|
| $\mathcal{A}$ | $[\alpha_1, \alpha_2, \dots \alpha_N]$ | $V_k = R(a_k) \quad R : \mathcal{A} \rightarrow \mathcal{R}^{nxn}$ |
| $\Gamma\left(\zeta_k^-, \varsigma_k\right)$ | $\theta_k = \left[P_k^-(1,1), \dots P_k^-(1,n), P_k^-(2,2) \dots P_k^-(2,n), \dots P_k^-(n,n), t_k\right]$ | |
| $g\left(\zeta_k^-, a_k, y_k\right)$ | $c_k = \begin{cases} M\left(a_k\right) & k < N_\varepsilon \\ \mathrm{Tr}\left[\mathrm{LP}^-_{\mathrm{k+N_T}}\right] + \mathrm{M}\left(\mathrm{a_k}\right) & k = N_\varepsilon \end{cases}$ | $L \in \mathcal{R}^{nxn} \qquad M : \mathcal{A} \rightarrow \mathcal{R}$ |
| $J^\pi\left(\zeta_k^-\right)$ | $\sum_{i=0}^{N_\epsilon} c_{k+i}$ | $a_k = \pi(\theta_k)$ |

Table 4.19: Elements of the discrete action space, finite horizon formulation.

not add a state to keep track of a budget), giving

$$\theta_k = \left[P_k^-(1,1), \dots P_k^-(1,n), P_k^-(2,2) \dots P_k^-(2,n), \dots P_k^-(n,n), t_k\right]. \tag{4.24}$$

At each step in the initial interval we incur costs according to the measurement selected, and at the end a further cost is added depending on the covariance of the final state estimate, giving

$$c_k = \begin{cases} M\left(a_k\right) & k < N_\varepsilon \\ \mathrm{Tr}\left[\mathrm{LP}^-_{\mathrm{k+N_T}}\right] + \mathrm{M}\left(\mathrm{a_k}\right) & k = N_\varepsilon \end{cases}. \tag{4.25}$$

As before, our cost function is simply

$$J^\pi\left(\zeta_k^-\right) = \sum_{i=0}^{N_\epsilon} c_{k+i}. \tag{4.26}$$

The details of the formulation are summarized in Table 4.19.

### 4.5.2 Examples

In Section 4.2 we examined our algorithm's performance in light of insights gained from algebraic analysis of a simple scalar problem. In Section 4.3 we examined performance relative to simple heuristic policies that may come close to optimal in the steady-state. In Section 4.4 we were able to compare performance to the optimal solution, found by another method and presented in the literature. In this section we compare our results with those for another suboptimal algorithm from the literature, specifically Athans' [5].

### 4.5.2.1 System Parameters

The sample system Athans uses to illustrate his algorithm is summarized in Table 4.20. It describes the rectilinear motion of a mass subject to a gaussian white noise acceleration and jerk. The three states are the position, velocity, and acceleration of the mass, and the two possible measurements are a noisy position measurement and a noisy velocity measurement. This might be thought of as a simplified version of a radar tracking problem, where one type of pulse produces a good position measurement while another type of pulse produces a good velocity measurement [36].

Note that the model described in Table 4.20 is a continuous-time model. Though Athans does everything in terms of continuous time, we prefer to use a discrete time formulation. The system is therefore discretized before application of the learning algorithm using a time step specified in the next subsection.

### 4.5.2.2 Algorithm Choices

The rationale for various parameter choices has been discussed extensively in the earlier sections of this chapter. The choices themselves are presented in Table 4.14; only a few of them require comment beyond what has gone before.

The relatively large scale used for the cost deserves some comment. A scaling factor of 100 was chosen after noting that in Athans' paper, several of the policies listed differed in cost by as little as 0.02. With a smaller scaling on the cost, these different policies would become indistinguishable.

In Athans' paper, an initial policy guess of

$$\begin{aligned} a_k &= 1 \quad (0 \leq t_k \leq 3) \\ a_k &= 2 \quad (3 < t_k \leq 4) \end{aligned}$$

| | | |
|---|---|---|
| Linear System | $A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ | $C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ |
| Process Noise | $W = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 9 \end{bmatrix}$ | |
| Sensor Noise | $V(1) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ | $V(2) = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ |
| Action Costs | M(1) = 0 | M(2) = 0 |
| Horizons | $T = 4$ sec | $T_p = 7$ sec |

Table 4.20: Parameters for tests conducted with grid-based algorithm.

was used; measure position for the first three seconds, then velocity. In order to provide a fair comparison, the same initial guess was used for the learning algorithm. It should be noted that this is the only example so far to have made use of an initial policy guess. The initial covariance $P_0$ shown in the table was also specified in Athans' example.

#### 4.5.2.3 Results

Considering the failure of the learning algorithm for our other finite-horizon example, the results for this problem were surprisingly good. Figure 4-19 shows the best policy found by Athans' algorithm compared to the best policy found by the reinforcement learning algorithm. The latter's cost is lower by about 3% , an impressive figure when one considers that Athans was choosing among policies that differed by less than 1/100 of a percent in cost. Unfortunately, we can be no more sure of the proximity of our solution to the true optimal than Athans was of his.

One might wonder why reinforcement learning was able to produce a good result in this case, but not in the finite horizon problem treated in the previous section. The difference is especially striking when one considers that the example in the previous section dealt with a two-dimensional system, while the system in this example is three-dimensional. It is hard to say exactly what the main reason is for the difference in performance, but here are some considerations that may help to explain it:

- Given a budget of 10.0 and a discretization level of 0.1 for action choice, over a horizon of 80 steps there are $100^{80}$ possible policies for our continuous-action, finite horizon problem. In comparison, given a time horizon of 4 seconds with timestep of 0.05 yielding a total of 80 discrete steps for our discrete-action example, we have only $2^{80}$ possible policies, roughly 135 orders of magnitude fewer.

- With those same givens, each error covariance can potentially correspond to 10,000 different states in the continuous-action formulation with its additional features keeping track of the timestep and remaining budget. In the discrete-action case, which need not keep track of a budget, this number is reduced by a factor of 100 to a mere 100

| | |
|---|---|
| Scaling | |
| State ($\theta$) | $[20 \quad 20 \quad 20 \quad 10 \quad 20 \quad 5 \quad 1]$ |
| Action ($a$) | 10 |
| Cost ($Q$) | 100 |
| Time Step for Discretization | $\Delta t = 0.05$ |
| CMAC-related parameters | |
| Memory size | 50000 |
| Number of layers | 8 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_1 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 10000$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 2$ |
| Estimation error variance | $P_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 30000 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 10000$ |
| Number of steps/trajectory | $N = 80$ |
| Exploration rate | $\epsilon = [0.2 \quad 0.1 \quad 0.04 \quad 0.02]$ |
| Rate of reduction for exploration rate | $N_\epsilon = 2000$ |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 1$ |
| Cutoff level for eligibility traces[5] | $N/A$ |

Table 4.21: Algorithm parameter settings for Athans' example problem.

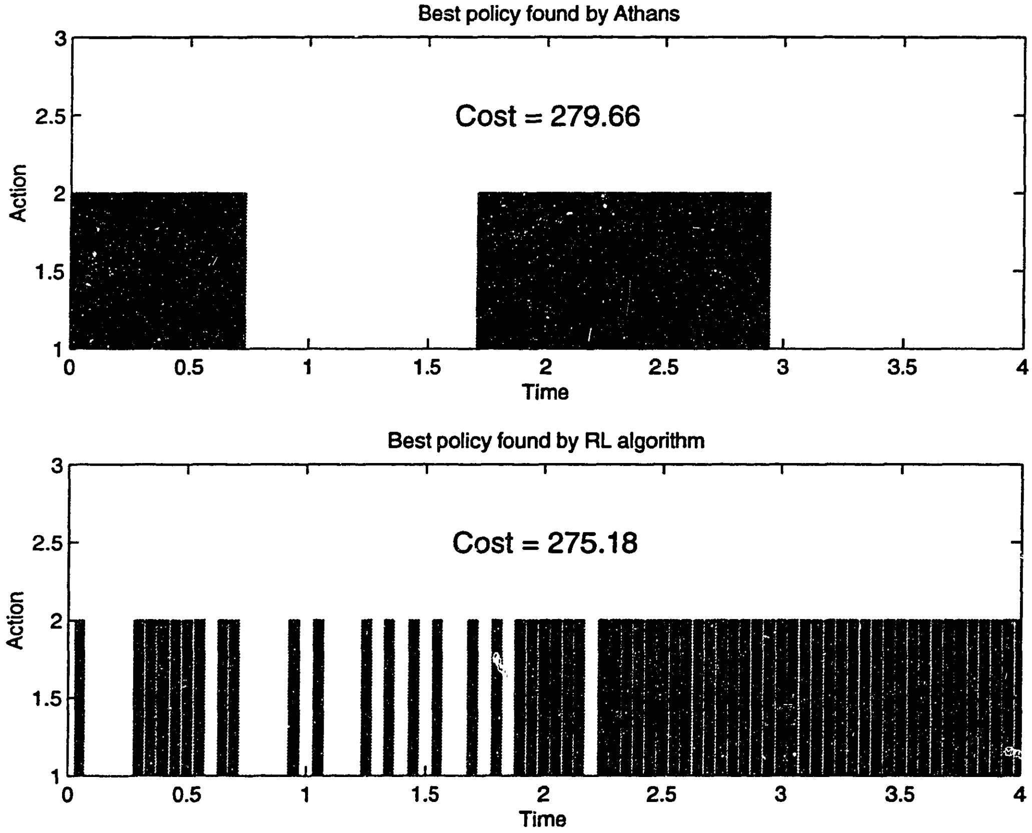


Figure 4-19: Comparison of learned policy (bottom) with best suboptimal policy found by Athans' algorithm (top).

different states. This is not so far from the 25 different states present in the 5-step horizon continuous-action case for which some degree of success was observed.

- In the discrete-action case we are dealing with a simple action space, and no function approximation is necessary for the policy.
- The discrete-action problem is non-convex. Thus, instead of competing against algorithms that can take advantage of the problem structure to quickly find the optimal solution, our algorithm is competing against another suboptimal algorithm. We have little real idea how well our algorithm's solution compares to the optimal solution.

## 4.6 Summary

Overall, the results for this chapter are promising. The unique characteristics of the linear problem and the Kalman Filter provided an excellent platform for testing the applicability of reinforcement learning algorithms to the measurement with cost problem.

In Section 4.2 we examined the infinite-horizon, continuous action space problem. Since we only dealt with a scalar system in this section, analysis of some special cases allowed us to get an idea of the nature of the optimal solution, which was dominated by the steady-state behavior. Near optimal results were obtained using a variety of different reinforcement learning algorithms. Mostly, though, this section was about understanding the various pieces of the algorithms and how they fit together.

Section 4.3 moved on to more difficult problems in dealing with the infinite-horizon, discrete action space problem. Relatively high-dimensional problems were treated with as many as 21 inputs considered in choosing any given action. The limited number of possible available actions gave rise to limit-cycling steady-state policies. Solutions generated by our main RL algorithm consistently matched or beat the best heuristics that could be found. In the case where four different actions were allowed the RL algorithm found some highly complex, efficient policies that could not easily be modeled using a heuristic.

Section 4.4 treated a finite-horizon, continuous action space problem from the literature as a test case. Interestingly enough, this was both the one case for which the optimal solution was known and the one case where reinforcement learning performed poorly. The convexity of the problem made it suitable for more standard solution techniques, while its structure made generalization and learning from experience more difficult.

Finally, in Section 4.5 a finite-horizon, discrete action space problem was treated. This problem, like the one treated in Section 4.4, was taken from the literature for comparison purposes. However, results were much more positive in this case, perhaps due to a much more limited set of policies. The optimal solution for this problem was unknown, but for the test case the reinforcement learning algorithm used produced a policy superior to the best suboptimal policy found in the literature.

The problems treated in this chapter do not just show that reinforcement learning is a strong tool able to deal with a variety of different formulations. They also provide insight into the ways in which this tool can be applied, for with its flexibility comes complexity. One thing that is clear is that the application of these techniques to any given problem requires a fairly high level of expertise.

# Chapter 5

# Nonlinear Examples

In dealing with linear systems, we were able to make a number of assumptions which simplified our problem and gave us confidence that our reinforcement learning algorithm should work. When we move to the domain of nonlinear systems, our problem becomes more complex not just in one way, but in several different ways at once. In this chapter we will treat a simple nonlinear problem in order to develop an understanding of the new issues involved. Addressing these issues on a small scale should help prepare us for the more complex, real-world application to follow.

## 5.1 Differences between the linear and nonlinear cases

The most immediate difference we encounter in our move to the nonlinear domain is the lack of an optimal filter. With extremely few exceptions, optimal filters for nonlinear systems are infinite-dimensional and hence unrealizable. With a linear system we could always use the Kalman filter to process our measurements and be confident that we were making the best use of the information provided. With a nonlinear system we must choose from a suite of known suboptimal filters. Since the performance of these filters generally depends on the specific types of nonlinearities present in the system, no one filter will be best for all situations. See Poor [45] for detailed derivations of suboptimal nonlinear filters.

One of the simplifications obtained by using the Kalman filter for linear systems was the deterministic propagation of the estimate's error covariance independent of the actual stochastic trajectory. Since we were measuring performance in terms of that error covariance, the stochastic portion of the filter (i.e. the state estimate) became irrelevant and we were assured of deterministic state transitions. Learning becomes much easier when taking the same action in the same state always results in the same cost and same state transition. With nonlinear systems, however, the propagation of the estimates of the state and error variance cannot be separated. This results in stochastic transitions for the filter state, making learning harder.

Also, with a nonlinear filter we do not have access to the actual error covariance, only an estimate of that quantity. The quality of that estimate will depend not just on the current state, but on the filter's entire past history. This means that when we deal with nonlinear problems our filter state transitions are no longer Markov. Since the Markov property is the foundation on which all reinforcement learning and dynamic programming theory rests, we can no longer make an *a priori* assumption that our algorithms will work. We can hold out some hope, however. Under certain conditions having to do with the levels of process and sensor noise and the type of nonlinearities present in the system we may be able to say that our system is *almost* Markov.[1] If we make sure that our estimates do not become too poor, we should be all right.

There are other obstacles as well. An essential difficulty with applying reinforcement learning to estimation problems is that the performance metric cannot be directly measured. In the linear case, this is not so important, as the Kalman filter can propagate the estimation error covariance from step to step without the need of direct measurements. However, as mentioned above filters must *estimate* the error covariance in the nonlinear case. The quality of that estimate will depend on the noise statistics and the frequency of measurements, and it may well be biased. It may be possible to get unbiased estimates of the error covariance

[1] Note that though the problem we are dealing with here can be classified as a Partially Observable Markov Decision Problem (POMDP), it differs from most POMDP's treated in the literature in an important respect. In most treatments of POMDP's the formulation of the belief state is one of the primary degrees of freedom; in our formulation the form of the belief state is specified ahead of time.

by comparing measured values to predicted values each time a measurement is taken, but since the problem we are trying to solve is *when* measurements should be taken, this method of generating a reinforcement is somewhat problematic.

One final area of difference between linear and nonlinear problems should be mentioned here. With linear problems the actual trajectory followed was irrelevant, since the Kalman filter allows separate propagation of the estimate and the error covariance. With nonlinear problems these two facets of the filter cannot be separated. There are two major consequences of this fact. The first is that the feature vector for nonlinear problems will generally be larger than for linear problems. This will be explained in more detail in the discussion of learning using the Extended Kalman Filter in Section 5.2. The second consequence is that the presence of a known external input exciting the system's dynamics can become an important factor. Knowledge of such an input and its future behavior can affect decisions on how measurements should be taken.

## 5.2 The Extended Kalman Filter

Though the formulation developed in Chapter 2 is general enough to embrace a wide variety of nonlinear systems, in this chapter we will restrict our analysis to a subclass of nonlinear systems and a specific filter, the Extended Kalman Filter (EKF). The EKF is a filter which can be applied to systems where the state propagates according to

$$\dot{x}(t) = f(x(t), u(t), t) + w(t) \tag{5.1}$$

where $w(t)$ is a zero-mean Gaussian white noise process with spectral density matrix $W(t)$, and measurements are taken at times $\{t_k\}$ according to

$$y_k = h(x_k, u_k, a_k, t_k) + v_k \tag{5.2}$$

where $v_k \sim N(0, V_k)$, $E[v_k v_{k+1}^T] = 0$, and $E[v_k w_k^T] = 0$. As in previous chapters, $V_k$ is a function of $a_k$. Note that we use $f$ and $h$ here though the arguments of the functions are different than those used in equations (2.1) and (2.3). The EKF operates essentially by linearizing equations (5.1) and (5.2) about the current state estimate and applying the Kalman filter. Derivations of the filter's equations can be found in Gelb [25], Poor [45], and many other estimation texts.

In terms of the formulation of Chapter 2 we have for the EKF a filter state $\zeta = [\hat{x} \quad \hat{P} \quad t]^T$. The filter is updated according to

$$\Phi\left(\zeta_k^-, y_k, u_k, a_k\right) = \begin{bmatrix} \hat{x}_k^+ \\ \hat{P}_k^+ \\ t_k^+ \end{bmatrix} = \begin{bmatrix} \hat{x}_k^- + K_k\left(y_k - h\left(\hat{x}_k^-, u_k, a_k, t_k\right)\right) \\ \left(I - K_k H_k\right) \hat{P}_k^- \\ t_k^- \end{bmatrix} \tag{5.3}$$

where the filter gain $K_k$ is calculated according to

$$K_k = \hat{P}_k^- H_k^T \left(H_k \hat{P}_k^- H_k^T + V_k\right)^{-1} \tag{5.4}$$

with

$$H_k = \left.\frac{\partial h\left(x\left(t_k\right), t_k\right)}{\partial x\left(t_k\right)}\right|_{x(t_k)=\hat{x}_k^-}. \tag{5.5}$$

The state estimate is then propagated to the next timestep by

$$\begin{aligned} \Psi\left(\zeta_k^+\right) \;=\; & \begin{bmatrix} \hat{x}_{k+1}^- \\ \hat{P}_{k+1}^- \\ t_{k+1}^- \end{bmatrix} = \begin{bmatrix} \hat{x}_k^+ \\ \hat{P}_k^+ \\ t_k^+ \end{bmatrix} + \\ & \begin{bmatrix} \int_{t_k}^{t_{k+1}} f\left(\hat{x}\left(\tau\right), u\left(\tau\right), \tau\right) d\tau \\ \int_{t_k}^{t_{k+1}} \left(F\left(\hat{x}\left(\tau\right), u\left(\tau\right), \tau\right) \hat{P}\left(t\right) + \hat{P}\left(t\right) F^T\left(\hat{x}\left(\tau\right), u\left(\tau\right), \tau\right) + W\left(t\right)\right) d\tau \\ \Delta t \end{bmatrix} \end{aligned} \tag{5.6}$$

where

$$F\left(\hat{x}\left(t\right), t\right) = \left.\frac{\partial f\left(x\left(t\right), t\right)}{\partial x\left(t\right)}\right|_{x(t)=\hat{x}(t)}. \tag{5.7}$$

## 5.3 The Cubic Plant Problem

In our examination of learning algorithms operating on nonlinear systems one of our principal difficulties will be with the evaluation of the algorithms' performance. Having no idea what the actual optimal policy looks like, we can only compare the policies that we learn with policies arrived at heuristically. This problem is exacerbated by the inherent difficulty of nonlinear estimation problems: for many problems of interest a good heuristic may be hard to find. Even having found a heuristic which seems to offer good performance, there will be a tendency to worry that some other heuristic could have done better if one had only thought of it. If we do not have faith in our ability to choose a good heuristic, we can not have confidence in our analysis of the performance of our RL algorithms. For these reasons we have chosen to begin with simple problems that are amenable to basic analysis for selection of a good heuristic.

One such problem is the cubic plant. With a state propagation equation

$$f\left(x\left(t\right),u\left(t\right),t\right)=-x^{3}\left(t\right)+u\left(t\right) \tag{5.8}$$

and linear measurement equation

$$h\left(x_k,u_k,a_k,t_k\right)=x_k \tag{5.9}$$

this scalar system is simple enough to lend itself to some basic analysis. Specifically, when we propagate the estimated error variance according to the EKF's equations we get

$$\dot{\hat{P}}\left(t\right)=-6\hat{x}^{2}\left(t\right)\hat{P}\left(t\right)+W\left(t\right). \tag{5.10}$$

From this we can see that in the absence of measurements, when $\hat{x}^2$ is large the error variance will tend to decrease and when $\hat{x}^2$ is small the error variance will tend to increase. This in turn suggests that when measurements are costly, more effort should be put into measurement when $\hat{x}$ is small in magnitude than when $\hat{x}$ is large in magnitude.

Keeping this in mind, we will look at two example problems using the cubic plant. First we will treat a simple action space with only two choices: take a measurement or don't. Then we will examine the problem using a continuous (complex) action space similar to the one treated in Section 4.2.

## 5.4 Discrete Action Space, Infinite Horizon

### 5.4.1 Formulation

The basic system equations for the cubic plant problem have already been given in Equations (5.8) and (5.9). One further element must be specified, however: we need to know the way in which the known control $u(t)$ is generated. When we were dealing with linear problems in the previous chapter, the propagation of the state estimate and error covariance were decoupled, and so $u(t)$ could safely be ignored. With nonlinear problems this is no longer true. The known control signal may exert a profound influence on the state trajectories, and through them on the optimal measurement policy.

In order to excite the system's dynamics at a suitable frequency range we will be making use of a known control $u(t)$ which evolves as a first-order Gauss-Markov process with time constant $T$; i.e.

$$\dot{u}(t) = -\frac{1}{T}u(t) + w_2(t) \tag{5.11}$$

where $w_2(t)$ is a zero-mean gaussian white-noise process of intensity $W_2(t)$, uncorrelated with either the system's process noise $w(t)$ or the sensor noise $v_k$. This sort of process is frequently used by engineers to model a variety of semi-random physical phenomena, from wind gusts to pilot actions.

In our discussion of the overall problem framework in Chapter 2 we mentioned that any information about the past, present, or future of the known control signal may prove useful in choosing a measurement strategy. For that reason we introduced the variable $\varsigma$, which contains all the available information about $u(t)$. This new variable was then used as one of the arguments of the function $\Gamma$ in order to generate the decision variable: $\theta_k = \Gamma(\zeta_k, \varsigma_k)$. In this particular case we have no access to future values of $u(t)$. The past and present values are available, however, so we have we have

$$\varsigma_k = [u_0 \quad u_1 \quad \dots \quad u_k]. \tag{5.12}$$

The past values of the input have only an extremely indirect influence on current estimation decisions and no influence at all on future values of $u_k$, so the only part of $\varsigma_k$ that we care about is $u_k$ itself.

Since neither the plant dynamics nor the measurement equation is time-varying, we can leave $t_k$ out of the filter state, which gives us

$$\zeta_k = [\hat{x}_k \quad \hat{P}_k]^T.$$

We thus have three potential components for our decision variable:

$$\theta_k = \Gamma\left(\hat{x}_k, \hat{P}_k^-, u_k\right). \tag{5.13}$$

This is where a bit of engineering judgement comes into our problem formulation. There is no doubt that $u_k$ provides information that can help us decide how to measure. There is some question as to its *relative* importance, however. Since our cost is measured in terms of the error covariance, $u_k$ must go through two levels of indirection in order to make a difference. The known control exerts an influence over expected future value of $\hat{x}$, which in turn exerts an influence over future values of $\hat{P}$. In this particular case, where we are allowing only two possible actions (measure or don't measure), the influence of $u_k$ is further diminished. The relative importance of this variable must then be weighed against the expense of adding another input to the $\tilde{Q}$ function approximator. If the dependence of $Q$ on $u_k$ is weak, then including $u_k$ in $\theta_k$ may just serve to dilute experience and add error into the learning process. Both intuition and some basic tests indicate that this is the case for this problem, and so $u_k$

has been left out of the decision variable in the experiments described below. This gives us

$$\theta_k = [\,\hat{x}_k \quad \hat{P}_k^-\,]. \tag{5.14}$$

Most of the remaining formulation elements should be familiar from either the explanation of the EKF in Section 5.2 or the discrete action space, infinite horizon cost formulation of Section 4.3. The complete formulation is summarized in Table 5.1.

The nonlinear nature of our problem complicates our choice of cost functions substantially. What we would like to use for our performance cost is the estimation error covariance. Unfortunately, we have no way of either directly measuring that quantity or propagating it accurately from one timestep to another. All we have is the EKF's Ricatti matrix, $\hat{P}$.

Depending on the particular circumstances in which we are using the EKF, $\hat{P}$ may be a good estimate of the error covariance or it may be an extremely poor estimate. If we are in the former situation we may be justified in using $\hat{P}$ in our cost formulation and can be optimistic about our chances of generating a good policy through reinforcement learning. If we are in the latter situation and we use $\hat{P}$ in our cost formulation, we run the risk of the learning process failing completely or generating a policy that produces low $\hat{P}$'s but large estimation errors.

The quality of $\hat{P}$ as an estimate of the error covariance depends on a variety of factors, including the initial estimation error, the magnitudes of the process and sensor noise relative to the nonlinearities, and the frequency of measurements relative to the system timescales. To some extent these factors can be dealt with using some simple tests. By running the filter with a policy of measuring at every possible instant and comparing the $\hat{P}$'s with actual squared errors we can get a best case estimate of their quality. Proceeding with the learning algorithm only makes sense if this best case scenario does well.

If our system passes this test we are still faced with a difficult problem. The quality of $\hat{P}$ may still vary widely with the measurement frequency, and that frequency will be determined by the policies we employ. Since we do not know ahead of time what policies we will be using during the learning process, we do not know how often measurements will be taken. This, of course, means that we cannot be sure that the $\hat{P}$ we are using at any given time is representative of the actual estimation error covariance. What we would like is some sort of on-line reality check that could tell us whether our estimate of the error covariance is any good.

One possibility for conducting such a reality check would be to compare the squared output error with $\hat{P}$ each time a measurement is taken. For our current example system, we have

$$\mathrm{E}\left[\left(y_k - \hat{y}_k^-\right)^2\right] = \mathrm{E}\left[\left(x_k - \hat{x}_k^-\right)^2\right] + V_k. \tag{5.15}$$

If we then assume that our estimate is unbiased and our error is Gaussian, we could calculate the probability of getting the observed measurement error if we believe $\hat{P}_k^-$:

$$\Omega = \mathrm{P}\left[Z \leq (y_k - \hat{y}_k) \middle| \sigma_Z^2 = \hat{P}_k^- + V_k\right] \tag{5.16}$$

where $Z = Y_k - \hat{y}_k$ and $y_k$ is a sample value of the random variable $Y_k$. If calculation of $\Omega$ indicated that our current $\hat{P}$ is unlikely to be correct, we could then take action.

Of course, there are several major problems with this approach. First of all, we can only perform the above calculation if we have a linear measurement equation. If it is nonlinear, then there would probably be no equivalent to Equation (5.15). For instance, a sinusoidal output equation would yield

$$\mathrm{E}\left[\left(y_k - \hat{y}_k^-\right)^2\right] = \mathrm{E}\left[\left(\sin\left(x_k\right) - \sin\left(\hat{x}_k^-\right)\right)^2\right] + V_k \neq f\left(\left(x_k - \hat{x}_k^-\right)^2\right) + V_k. \tag{5.17}$$

Secondly, the assumptions of no bias and a Gaussian distribution are unlikely to be valid. Another major problem is that this check could only be conducted when measurements were taken. It thus provides no verification of $\hat{P}$ during the (potentially long) intervals between measurements. Finally, even if all these other problems could be overcome, it is unclear what sort of action would be appropriate if calculation of $\Omega$ indicated a poor $\hat{P}$. The most obvious alternatives all have a heuristic, ad-hoc nature that would tend to negate the logical basis that we are trying to establish by using reinforcement learning.

| Description | Symbol | Contents | Notes |
|---|---|---|---|
| State Propagation | $f(x(t),u(t),t)$ | $-x^3(t)+u(t)$ | $\dot{x}(t)=f(x(t),u(t),t)+w(t)$ |
| Measurement | $h(x_k,u_k,a_k,t_k)$ | $x_k$ | $y_k=h(x_k,u_k,a_k,t_k)+v_k$ |
| Filter State | $\zeta_k$ | $[\hat{x}_k \quad \hat{P}_k]^T$ | No time dependence, so $t$ unnecessary. |
| Known Control History | $\varsigma_k$ | $[u_0 \quad u_1 \quad \dots \quad u_k]$ | $\dot{u}(t)=-\frac{1}{T}u(t)+w_2(t)$ |
| Estimate Propagation | $\Psi\left(\zeta_k^+,u_k\right)$ | $\begin{bmatrix}\hat{x}_k^+\\\hat{P}_k^+\end{bmatrix}+\begin{bmatrix}\int_{t_k}^{t_{k+1}}\left(-x^3\left(\tau\right)+u\left(\tau\right)\right)d\tau\\\int_{t_k}^{t_{k+1}}\left(-6x^2\left(\tau\right)\hat{P}\left(\tau\right)+W\left(\tau\right)\right)d\tau\end{bmatrix}$ | |
| Estimate Update | $\Phi(\zeta_k^-,y_k,u_k,a_k)$ | $\begin{bmatrix}\hat{x}_k^-+K_k\left(y_k-\hat{x}_k^-\right)\\\left(I-K_k\right)\hat{P}_k^-\end{bmatrix}$ | $K_k=\frac{\hat{P}_k^-}{(\hat{P}_k^-+V_k)}$ |
| Action Space | $\mathcal{A}_k$ | $\{\alpha_1,\alpha_2,\dots\alpha_N\}$ | $V_k=R(a_k) \qquad R:\mathcal{A}\mapsto\mathcal{R}$ |
| Feature Vector | $\Gamma\left(\zeta_k^-,\varsigma_k\right)$ | $\theta_k=[\hat{x}_k^- \quad \hat{P}_k^-]$ | |
| Single Step Cost | $g\left(y_k,\zeta_k^-,a_k\right)$ | $c_k=\hat{P}_k^++M(a_k)$ | $M:\mathcal{A}\mapsto\mathcal{R}$ |
| Cost Function | $J^\pi\left(\zeta_k^-\right)$ | $\sum_{i=0}^{\infty}\gamma^i c_{k+i}$ | $a_k=\pi(\theta_k)$ |

Table 5.1: Elements of the cubic plant formulation with discrete action space.

Lacking any better performance metric than $\hat{P}$ and any good method of checking $\hat{P}$ on-line, we must accept that, at least for the EKF, the scope of problems to which we can confidently apply reinforcement learning is limited. We use $\hat{P}$ in our cost function, but conduct checks both before and after learning to make sure that $\hat{P}$ is reliable over the range of policies we are likely to encounter.

| | |
|---|---|
| Process Noise ($W$) | 1 |
| Control Noise ($W_2$) | 1 |
| Control Time Constant ($T$) | 1 |
| Discount Factor ($\gamma$) | 0.99 |
| Timestep ($\Delta t$) | 0.01 |

| Action ($a$) | 1 | 2 |
|---|---|---|
| Sensor Noise ($V$) | 1 | $10^6$ |
| Cost ($M$) | 1 | 0 |

Table 5.2: System parameters for discrete action cubic plant test case.

### 5.4.2 Parameters

The first parameters that we must choose for our system are those that determine the shape of its trajectories. These include both the process noise, $W(t)$, and the parameters controlling the known input, $T$ and $W_2(t)$. We have two conflicting objectives for our trajectory: we would like large excitations so that we can see the effects of the nonlinearity, but we need small errors so that the EKF's linearizations hold up. We can produce large excitations by increasing $W(t)$, but doing so also increases estimation error and can cause problems with the EKF. For this reason we use the known external input to achieve our excitation, and this is why $W_2$ is so much larger than $W$ (see Table 5.2). The control input's time constant $T$ must also be chosen carefully if we are to see the nonlinearity's effects. If it is too small relative to the measurement interval we will be unable to separate out the effects of different excitation levels, but if it is too large we won't be able to observe a significant range of excitations in a reasonable number of timesteps. Figure 5-1 shows a sample trajectory generated using the parameters from Table 5.2.

Once the general characteristics of the trajectories to be estimated have been established, we can say something about the timescales to be used in simulating the system and

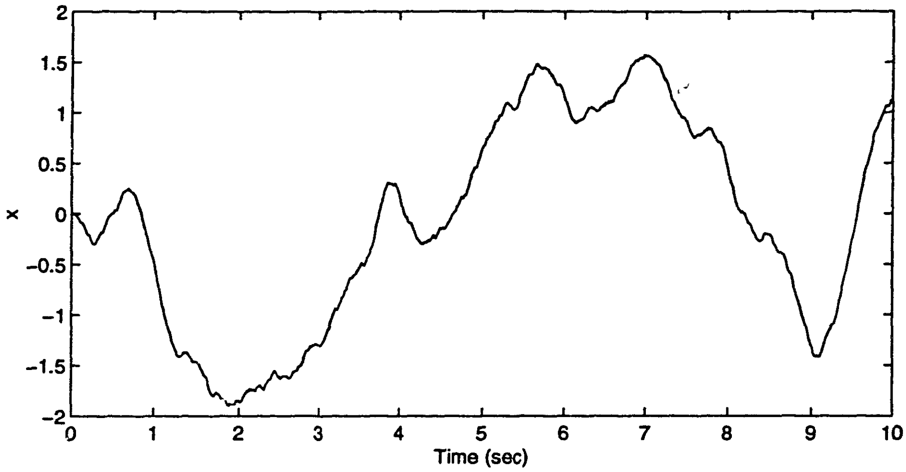


Figure 5-1: A sample trajectory generated using the system parameters specified in Table 5.2.

running the filter. There are three separate timesteps which we must consider. First, we have the timestep used to integrate the state equations for both plant and filter. The necessary characteristics for this stepsize depend on the type of nonlinearities present and the method of integration which is used. For the nonlinear examples presented in this thesis, the `odeint` function from *Numerical Recipes in C* [47] was used; this code contains provisions for automatically adjusting the stepsize to ensure a desired level of accuracy.

Next, we have the timestep used to simulate the additive white noise in the state propagation equations. Recall Equations (5.1) and (5.11), both of which are driven by white noise. Due to the characteristics of white noise, we cannot simulate these equations exactly. Instead we approximate: for the known input $u(t)$ which propagates linearly we have

$$u_k = e^{-\Delta t/T} u_{k-1} + \sqrt{W_2 \Delta t} d_2 \tag{5.18}$$

and for the state propagation equation we have

$$x_{k+1} = \int_{t_k}^{t_{k+1}} f\left(x\left(t\right), u_k, t\right) dt + \sqrt{W \Delta t} d_1 \tag{5.19}$$

where $d_1$ and $d_2$ are normally distributed random variables. For this approximation to be valid, the timestep $\Delta t$ used in these two equations must be small enough. Just how small it needs to be will depend on $T$ and the characteristics of the nonlinearity. For this case, a timestep of $\Delta t = 0.01$ was used.

The third timestep we must consider is the interval at which we make decisions as to whether or not to measure. This interval may be determined by external constraints, but in this example problem we are free to choose whatever value we like. To keep things simple, we have elected to use a single timestep (the $\Delta t$ specified above) for both this decision interval and the noise approximation interval described in the previous paragraph.

Since there are so many different approximations going on simultaneously, one should always check to make sure that the combined system is performing in a reasonable manner. For this particular example, such a test was performed by running the EKF over a period of 400 seconds (40,000 time steps), measuring at every 10th step. The resulting $\hat{P}$'s were then sorted and grouped into 100 bins of 400 samples each. The means of both $\hat{P}$ and $(x-\hat{x})^2$ were then taken over the samples in each of these bins and plotted against each other in Figure 5-2. If everything is going well, the data points should lie approximately on the line $y = x$. The graph on the left shows that our $\hat{P}$ values are reasonably accurate for the parameters we have chosen. On the right we see what would occur if a larger stepsize ($\Delta t = 0.1$) were used. Though measurements are taken at the same interval[2] $\hat{P}$ underestimates the actual error variance at low values. This occurs because those low values of $\hat{P}$ tend to be associated with high values of $x^2$. In these regions the 0.1 second timestep is not small enough for Equation (5.19) to accurately approximate the effects of the white noise, and so the simulated trajectory is no longer representative of the model that the filter is based upon.

Once we have settled the various time scales involved in our system we can select a discount factor. Here as well we have some extra freedom because we have no externally imposed constraints. Since we are using this example as a test problem, we would like to choose a formulation that lends itself to easy performance analysis. With this in mind, we choose a discount factor close to 1 in order to approximate an average cost formulation. With a discount factor near unity we can compare two policies that operate on the same

[2] In seconds, not timesteps.

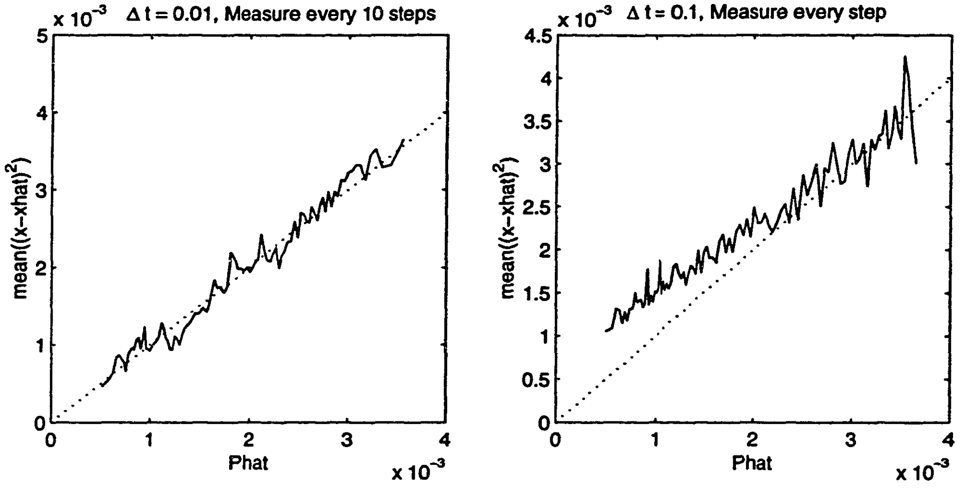


Figure 5-2: Comparison of the quality of $\hat{P}$ for filters running at different time scales. At left, the filter was propagated in 0.01 second intervals, while at right the timestep used was 0.1 seconds. In both cases measurements were taken at 0.1 second intervals.

data simply by summing the individual costs. With a smaller discount factor we would have to compare policy performance on a state-by-state basis. A discount factor of $\gamma = 0.99$ was used for this example problem.

### 5.4.3 A Priori Heuristic

One of the reasons for using reinforcement learning is the hope that analysis of the RL-based policy will lead to improved heuristic policies. If we have a simple heuristic that seems to work well but are worried that there may be another simple heuristic that would do better, RL can help there as well. But to get an idea of the benefits that can be attributed to learning we must create comparison heuristics both before and after the application of the learning algorithm.

For this particular example basic analysis of the problem suggests that we should measure more when $\hat{x}$ is small in magnitude than when it is large. Translated into a simple

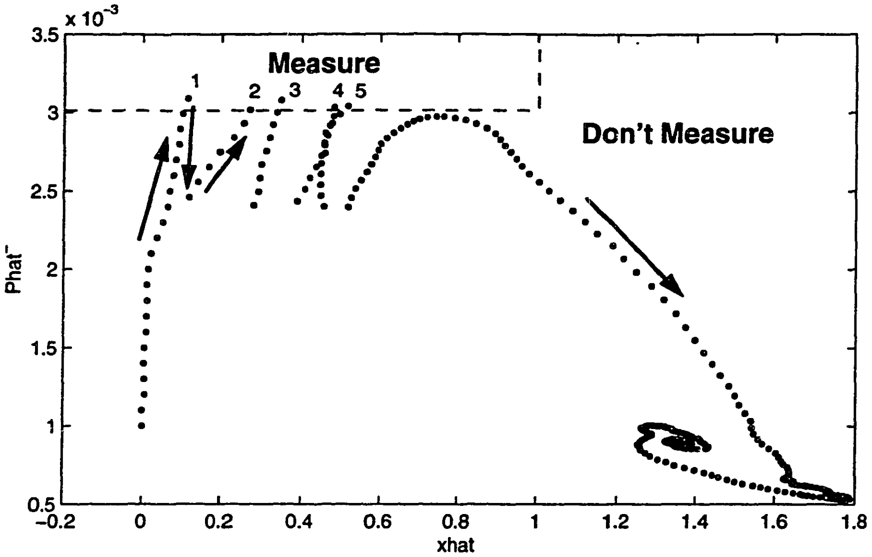


Figure 5-3: Sample trajectory for *a priori* policy. Numbers indicate places where measurements are taken.

heuristic policy, we get

$$a = \begin{cases} 1 & \left(\hat{P}_k^- > P_m\right) \wedge \left(\left|\hat{x}_k^-\right| < x_m\right) \\ 2 & elsewhere \end{cases} \tag{5.20}$$

where $P_m$ and $x_m$ are parameters to be determined by a simple search. Figure 5-3 shows a sample trajectory generated using such a heuristic. The values used for $P_m$ and $x_m$ are 0.003 and 1.0 respectively. A longer trajectory using the same policy is shown in Figure 5-4.

Before proceeding with the learning algorithm, it also seems wise to check the accuracy of the $\hat{P}$'s under the *a priori* heuristic. Figure 5-5 shows a statistical comparison of predicted variances versus actual observed squared errors. The bias in the graph is −0.00016 and the standard deviation is 0.0002, so our filter is slightly underestimating the error variance. These numbers seem good enough to enable learning to occur, but the uncertainties are large enough that we should expect the boundary between states where we should measure and states where we should not to be a bit fuzzy.

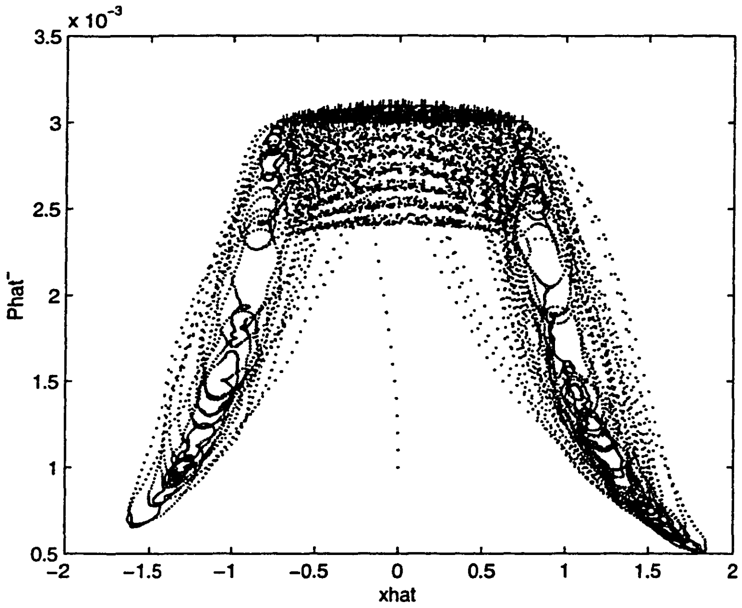


Figure 5-4: A longer trajectory for the *a priori* policy. Places where measurements are taken are indicated by + symbols.

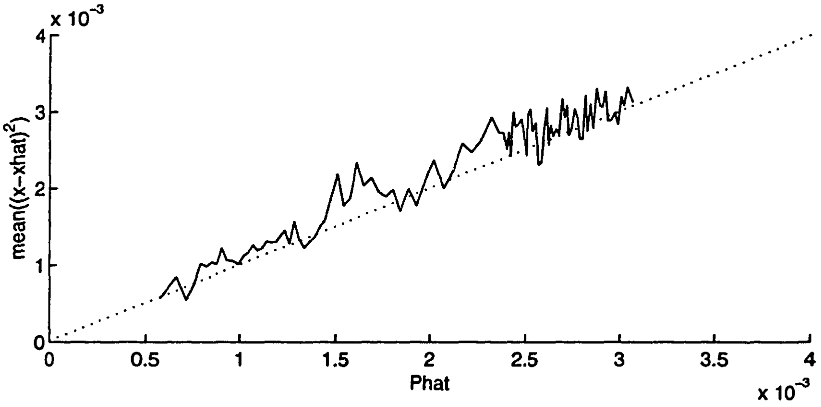


Figure 5-5: Check on quality of $\hat{P}$ for *a priori* heuristic. Dotted line is $y = x$.

### 5.4.4 Learning Setup

Table 5.3 shows the settings that were used for the learning algorithm. Parameter choices were made using the same techniques as were used for the linear problems in Chapter 4. Some tuning of parameters was done, but not to a very large extent. It is quite possible that better results could be achieved by tuning these parameters more carefully. Note that learning was attempted both with and without the external input in the feature vector. No significant difference in performance was observed, so the results presented below reflect the runs made *without* the external input in the feature vector.

### 5.4.5 Learning Results

None of the learning algorithms used converged to a single policy despite the use of a steadily decreasing learning rate. This is probably a reflection of the chattering effect described by Bertsekas, [13] and may be unavoidable. However, sampling of the policies generated by the learning algorithms did show that improvement was taking place. The policies described below are the best policies found out of those that were tested.

Before presenting those policies, however, some explanation of the criteria used in their selection is in order. Policies were tested at even intervals during the learning process. Each policy was tested using identical external input and process noise values. Since measurements occurred at different points in each policy, the same sensor noise values could not be used. Test trajectories were substantially longer in length than the learning trajectories in order to ensure roughly equal amounts of time were spent in the positive and negative $x$ regions. It was found that when shorter test trajectories were used, selection for policies that performed well in one region of the state space but poorly in others occurred.

Figure 5-6 shows a sample trajectory using the best policy that was found using SARSA($\lambda$). The regions delineating the learned policy are also shown on the graph; measurements are taken when the shaded region is entered.

In the process of experimentation with this problem, it was observed that substantial

| | |
|---|---|
| Scaling | |
| State ($\theta$) | $[20 \quad 10000]$ |
| Action ($a$) | 40 |
| Cost ($Q$) | 1000 |
| CMAC-related parameters | |
| Memory size | 50000 |
| Number of layers | 16 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_2 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 3000$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Estimation error variance | $P_0 = 0.001$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 300 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 10000$ |
| Number of steps/learning trajectory | $N = 400$ |
| Number of steps/test trajectory | $N = 3200$ |
| Exploration rate | $\epsilon = [0.1 \quad 0.02 \quad 0.01 \quad 0.005 \quad 0.001]$ |
| Rate of reduction for exploration rate | $N_\epsilon = 2000$ |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.9$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |

Table 5.3: Algorithm parameter settings for cubic plant tests with discrete action space.

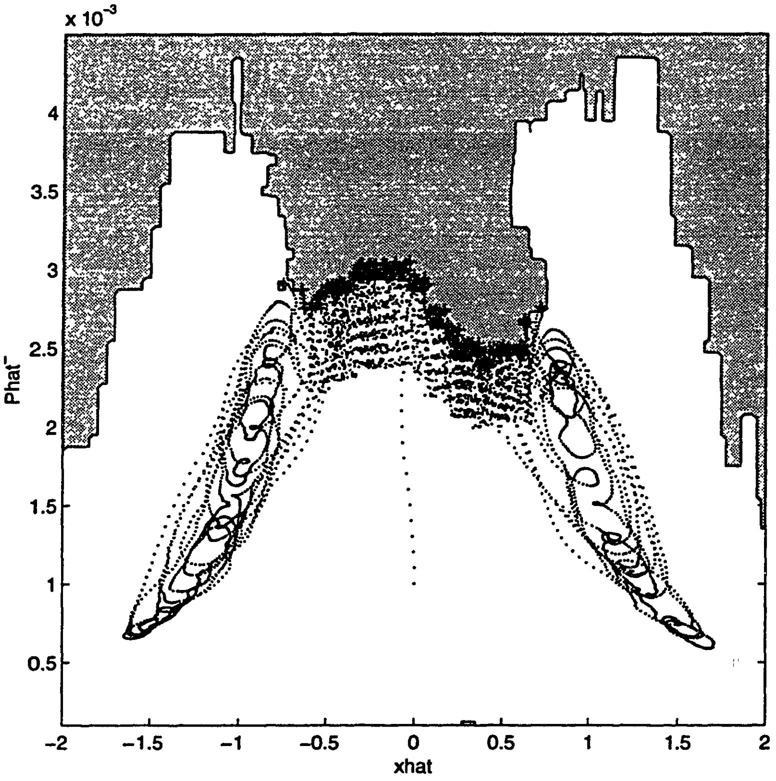


Figure 5-6: Sample trajectory using best policy found with SARSA($\lambda$).

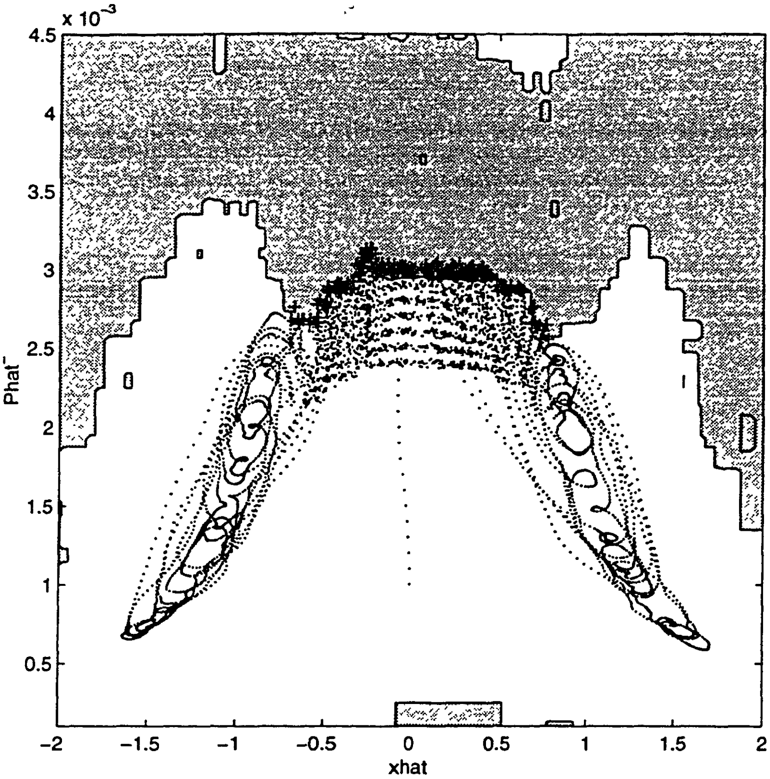


Figure 5-7: Sample trajectory using best policy found with Advantage Learning.

variations in policy in the critical central region yielded surprisingly small changes in total cost. In other words, the differential in the action direction, $\tilde{Q}(\theta, 1) - \tilde{Q}(\theta, 2)$ was very small. For best learning performance it seemed that some sort of accentuation of this differential would be useful. To achieve this, the Advantage Learning algorithm described in Chapter 3 was used. As predicted, when the initial learning rate was greater than $1/\kappa$ the algorithm diverged, but when care was taken to avoid this condition the algorithm performed quite well.[3] With $1/\kappa = 0.3$ and $\beta_1 = 2$ (yielding a learning rate of 0.25) Advantage Learning was able to generate a better policy than was found with any combination of parameters when either SARSA($\lambda$) or Q-Learning were used. This policy is shown in Figure 5-7 along with a sample trajectory.

[3] The $\kappa$ used here refers to the accentuation factor used in Advantage Learning and G-Learning, described in Chapter 3. It should not be confused with the use of the symbol $\kappa$ for the eligibility trace cutoff in the SARSA($\lambda$) algorithm.

| Policy | Cost |
|---|---|
| *A Priori* Heuristic | 97609 |
| SARSA($\lambda$) | 97942 |
| Advantage | 97152 |
| *A Posteriori* Heuristic | 97259 |

Table 5.4: Policy cost comparison.

Though systematic differences were observed in the costs of the various policies tested, the magnitude of these differences was not large. These costs are summarized in Table 5.4. The costs shown in the table represent the sum of the single-step costs over 40,000 steps. In order to make a fair comparison, the same process noise and external input values were used for each. The policy arrived at by the worst of these methods only differs from the best by about one percent. The table entry for the *a posteriori* heuristic refers to the policy described in the following section.

### 5.4.6 A Posteriori Heuristic

Looking at the policy found using Advantage Learning, it seems that our initial assumptions regarding the shape of the optimal policy may have been in error. Our *a priori* policy assumed that measurements are unnecessary when the magnitude of $\hat{x}$ is large. This resulted in the box-like boundary shown in Figure 5-3. After looking at the learning results, however, it seems that a boundary closer to the natural contours of the system is more appropriate. Based on the Advantage Learning results the heuristic shown in Figure 5-8 was developed. Here the boundary separating the "measure" region from the "don't measure" region makes a sort of trapezoidal shape. Some experimentation was done to find the best locations for the corner points. The result gives a cost consistently lower than could be found using the simple *a priori* heuristic, though still not consistently better than produced by the learning

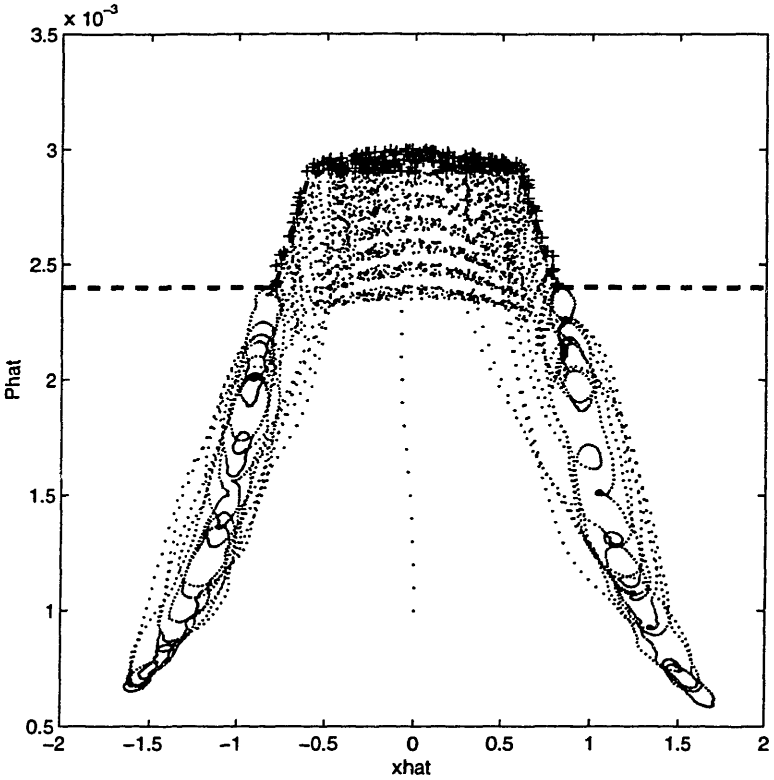


Figure 5-8: Trapezoidal heuristic policy. Measurements are taken at states above the dashed line.

algorithm. Further experimentation with the exact policy parameters might be able to provide a superior policy, but with the tiny differences in cost shown in Table 5.4 this hardly seems important.

The key observation here is that by using reinforcement learning we were able to (a) confirm that our initial policy was a good one and (b) make improvements to that policy which were non-obvious. Though we cannot say we have found the *optimal* policy, we can be fairly confident that we have come close to it and that we know its general characteristics.

## 5.5 Continuous Action Space, Infinite Horizon

This same cubic problem becomes substantially more complex when we allow our actions to be chosen freely from the unit interval. Visualization of the Q-function goes from being difficult (two 3-dimensional surfaces) to near-impossible (a single 4-dimensional surface). Formulation of policies goes from specification of a boundary in a 2-dimensional plane to specification of a full 3-dimensional surface. When the external input is used as a feature, a whole extra dimension is added, making the problem even harder. Nonetheless, our reinforcement learning algorithm is able to come up with a policy which performs quite well.

### 5.5.1 Formulation

To a large extent, the continuous action case shares its formulation with the discrete action case. Details of the latter can be found in Section 5.4.1. Here we will only describe the differences.

For this example our actions will come from the unit interval: $\mathcal{A}_k \in [0,1]$, where $a = 1$ corresponds to the decision to use the maximum available power and $a = 0$ corresponds to the decision not to measure. Accordingly, the variance of our sensor noise becomes

$$V_k = \begin{cases} R/a_k & a_k > 0 \\ V_\infty & a = 0 \end{cases} \tag{5.21}$$

where $R$ is the minimum possible noise variance and $V_\infty$ is some large constant chosen so as to avoid the discontinuity at $a_k = 0$.

With regard to the feature vector, we are again faced with the question of whether or not to use the known external input. Given the increased flexibility in action selection relative to the discrete case, it was thought that the external input might be a significant feature. Extensive testing was conducted both including $u_k$ in $\theta_k$ and excluding it. None of the tests showed any improvement due to the inclusion of $u_k$ in the feature vector, nor was any significant performance degradation observed.

| Description | Symbol | Contents | Notes |
|---|---|---|---|
| State Propagation | $f(x(t),u(t),t)$ | $x^3(t)+u(t)$ | $\dot{x}(t)=f(x(t),u(t),t)+w(t)$ |
| Measurement | $h(x_k,u_k,a_k,t_k)$ | $x_k$ | $y_k=h(x_k,u_k,a_k,t_k)+v_k$ |
| Filter State | $\zeta_k$ | $[\hat{x}_k \quad \hat{P}_k]^T$ | No time dependence, so $t$ unnecessary. |
| Known Control History | $\varsigma_k$ | $[u_0 \quad u_1 \quad \ldots \quad u_k]$ | $\dot{u}(t)=-\frac{1}{T}u(t)+w_2(t)$ |
| Estimate Propagation | $\Psi\left(\zeta_k^+,u_k\right)$ | $\begin{bmatrix} \hat{x}_k^+ \\ \hat{P}_k^+ \end{bmatrix}+\begin{bmatrix} \int_{t_k}^{t_{k+1}}\left(-x^3(\tau)+u(\tau)\right)d\tau \\ \int_{t_k}^{t_{k+1}}\left(-6x^2(\tau)\hat{P}(\tau)+W(\tau)\right)d\tau \end{bmatrix}$ | |
| Estimate Update | $\Phi(\zeta_k^-,y_k,u_k,a_k)$ | $\begin{bmatrix} \hat{x}_k^-+K_k(y_k-\hat{x}_k) \\ (I-K_k)\hat{P}_k^- \end{bmatrix}$ | $K_k=\frac{\hat{P}_k^-}{(\hat{P}_k^-+V_k)}$ |
| Action Space | $\mathcal{A}_k$ | $[0,1]$ | $V_k=\begin{cases} R/a_k & a_k>0 \\ V_\infty & a=0 \end{cases} \qquad R\in\mathcal{R}$ |
| Feature Vector | $\Gamma\left(\zeta_k^-,\varsigma_k\right)$ | $\theta_k=[\hat{x}_k^- \quad \hat{P}_k^-]$ | |
| Single Step Cost | $g\left(y_k,\zeta_k^-,a_k\right)$ | $c_k=\hat{P}_k^+ + Ma_k$ | $M\in\mathcal{R}$ |
| Cost Function | $J^\pi\left(\zeta_k^-\right)$ | $\sum_{i=0}^\infty\gamma^i c_{k+i}$ | $a_k=\pi(\theta_k)$ |

Table 5.5: Elements of the cubic plant formulation with continuous action space.

For the single step cost, we simply have

$$g\left(y_k,\zeta_k^-,a_k\right)=\hat{P}_k^+ + Ma_k. \tag{5.22}$$

The complete formulation is summarized in Table 5.5.

|  |  |
|---|---|
| Sensor noise at max power ($R$) | 0.0001 |
| Action cost factor ($M$) | 0.005 |

Table 5.6: System parameters for discrete action cubic plant test case.

### 5.5.2 Parameters

Again, parameter selection for the cubic problem has been discussed in Section 5.4.2. Here we need only specify those parameters relating to the action choice, $R$ and $M$. These are given in Table 5.6.

### 5.5.3 A Priori Heuristic

Development of a good *a priori* heuristic for the continuous case is substantially more challenging than for the discrete case. It is fairly clear that the optimal policy should be symmetric about the $\hat{P}$ axis and that it should increase with increasing $\hat{P}$. Before the learning algorithm was actually tested it was unclear exactly what functional form the policy should take, however. Preliminary learning tests seemed to indicate that a policy of the form

$$\pi(\theta_k) = \max(0, \min(1, \bar{\pi}(\theta_k))) \tag{5.23}$$

where

$$\bar{\pi}(\theta_k) = k_1 \left(\hat{x}_k^-\right)^2 + k_2 \hat{P}_k^- + k_3 \tag{5.24}$$

with $k_1 < 0$ worked well. The approximate values of the constants were found by fitting the policy surface to points generated by the learned policy. An automated search was then conducted to find the best parameters in that neighborhood. Some testing was done with constants in other neighborhoods, but no superior policies were found.

In retrospect, it seems that this policy form should have been an obvious candidate, and a straightforward search of the three-dimensional parameter space would have yielded the proper values for the constants. Still, before extensive learning tests have been conducted it is unclear whether a policy of some other form will produce superior results.

### 5.5.4 Learning Setup

Table 5.7 shows the settings that were used for the learning algorithm. Parameter choices were made using the same techniques as were used for the linear problems in Chapter 4. Some tuning of parameters was done, but not to a very large extent. It is quite possible that better results could be achieved by tuning these parameters more carefully. Note that learning was attempted both with and without the external input in the feature vector. No significant difference in performance was observed, so the results presented below reflect the runs made *without* the external input in the feature vector.

### 5.5.5 Learning Results and A Posteriori Heuristic

As with the discrete case, convergence to a single policy did not occur. In contrast to the discrete case, however, best results were obtained using SARSA($\lambda$), not Q-Learning or Advantage Learning.

Figure 5-9 shows a scatter plot of the actions chosen by the best learned policy in a test trajectory. Also shown in the figure is a surface which matches the data from the learned policy fairly closely and represents the best heuristic policy which was found. Two different views are shown, and the surface in the right-hand plot is transparent so that learned actions falling below the level of the heuristic policy can be seen. Total costs for the learned and heuristic policies over a 40,000 step trajectory differ by less than 0.5%. The heuristic policy shown in Figure 5-9 is *not* of the form described in Section 5.5.3. A quartic dependence on $\hat{x}_k^-$ was found to be a better match to the learned policy, giving us

$$\bar{\pi}(\theta_k) = k_1 \left(\hat{x}_k^-\right)^4 + k_2 \hat{P}_k^- \tag{5.25}$$

instead of Equation (5.24). With values $k_1 = 0.003$ and $k_2 = 90$ this policy produces costs roughly 5% lower than the best quadratic policy using Equation (5.24).

| | |
|---|---|
| Scaling | |
| State ($\theta$) | $[40 \quad 40000]$ |
| Action ($a$) | 200 |
| Cost ($Q$) | 10000 |
| CMAC-related parameters (same for $\tilde{Q}$ and $\tilde{\pi}$) | |
| Memory size | 50000 |
| Number of layers | 16 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_2 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 10000$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Estimation error variance | $P_0 = 0.001$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 700 |
| Weights for $\tilde{\pi}$ approximator | All weights initialized to 0 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 50000$ |
| Number of steps/learning trajectory | $N = 400$ |
| Number of steps/test trajectory | $N = 3200$ |
| Exploration rate | $\epsilon = [0.1 \quad 0.02 \quad 0.01 \quad 0.005 \quad 0.001]$ |
| Rate of reduction for exploration rate | $N_\epsilon = 8000$ |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.9$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |
| Fraction of samples updated after each trajectory | $\eta = 0.1$ |

Table 5.7: Algorithm parameter settings for cubic plant tests with continuous action space.

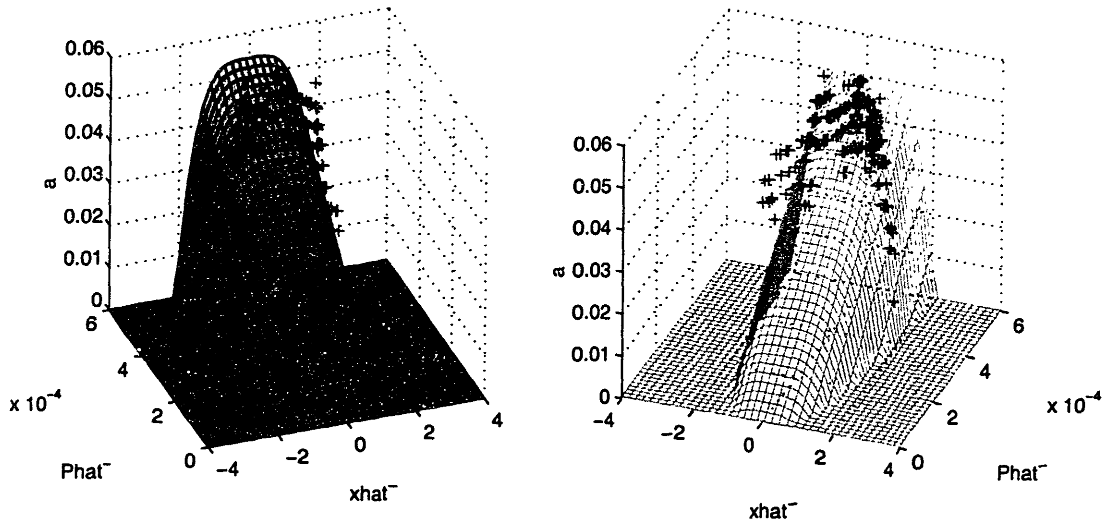


Figure 5-9: Two views of a policy for the continuous action case. The plus symbols mark actions taken by the best learned policy. The surface represents the best *a posteriori* heuristic policy that was found. The one on the right uses a transparent mesh so that learned actions falling below the level of the heuristic policy can be seen.

Various statistics from a sample trajectory are shown in Figure 5-10. At upper left the state estimates from both heuristic and learned policies are shown alongside the actual state. The bottom two graphs show the covariance estimates and actions chosen for both the heuristic and learned policies.

## 5.6 Summary

In this chapter we have demonstrated that reinforcement learning can be applied successfully to nonlinear versions of the measurement scheduling problem. The use of a cubic plant as a simple test case allowed formulation of reasonable *a priori* heuristics and made it possible to check the learned policies for reasonability. Good results were obtained for both discrete and continuous action spaces, paving the way for the use of RL in the more complex application of Chapter 6. While studying this problem, a number of interesting observations were made.

First of all, none of the reinforcement learning algorithms tested converged to a final policy despite steady reductions in learning and exploration rates. This chattering behavior

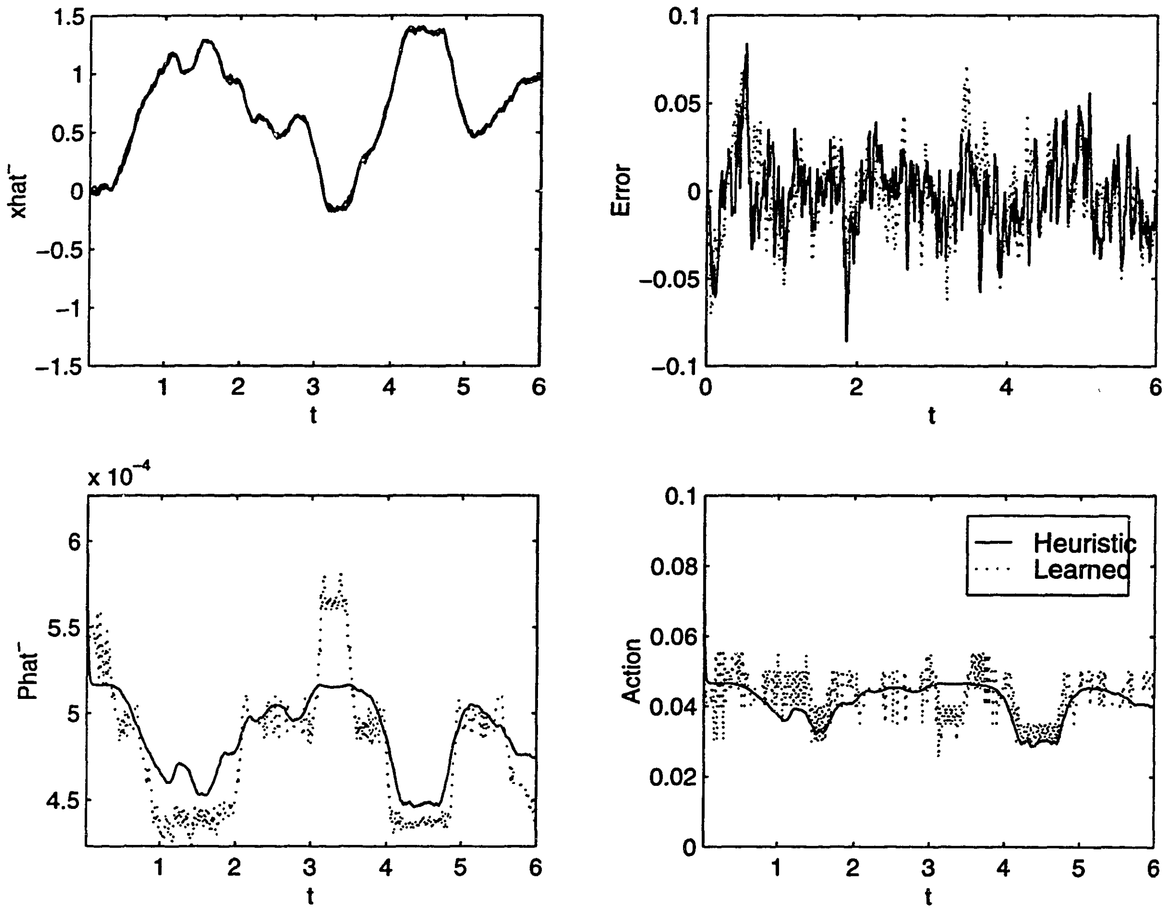


Figure 5-10: Statistics from the best learned policy and *a posteriori* heuristic policy.

has been observed by Bertsekas [13] in other RL problems using function approximation. Despite a lack of convergence, a general downward trend in costs was observed, and testing of policies encountered during learning yielded excellent policies. Invariably, the best policies were found near the end of the learning run – a good indication that the algorithm was working as intended.

The discrete action case provided an excellent opportunity for testing of some of the observations made in Chapter 3. The divergence of Advantage Learning when high accentuation levels are used without correspondingly low learning rates. Advantage Learning was successfully applied when the limits described in Chapter 3 were observed, providing better results than either SARSA($\lambda$) or Q-Learning.

In addition, a sort of synergy between the use of heuristic and learned policies was observed. Since a large number of parameters need to be chosen to successfully apply an RL algorithm, it is not always clear whether the results being obtained reflect the best that the algorithm can do. When a heuristic policy is available we have a basis for comparison, and parameters can be adjusted until the learning algorithm meets or beats the heuristic's performance. Often, it is then possible to look at the learned policy and come up with a superior heuristic. If that new heuristic performs better than the learned policy, the process can be repeated. With a problem where we have no objective way of finding the optimal solution, this comparison of heuristics with learned policies can give us a confidence in a policy's performance that would be lacking if either was used alone.

# Chapter 6

# Application to Radar Tracking

## 6.1 Description of Problem

In order to demonstrate the effectiveness of our approach to the measurement scheduling problem in real-world situations, we will treat a problem in theater missile defense. The scenario we envision is displayed graphically in Figure 6-1. The overall objective is to defend strategic sites in a limited geographic area from missile attacks originating in a neighboring hostile country. To this end, a single phased-array search radar is deployed in a central location with smaller targeting radars positioned around it. The search radar's mission is to detect incoming missiles and track them until they come in range of a targeting radar (Figure 6-2). When the missile enters the targeting radar's range its position must be known well enough so that the hand-off can occur with little or no search required on the part of the targeting radar. That radar then uses its measurements to guide an intercepting missile which destroys the incoming BRV.

The search radar must be prepared for the possibility of multiple BRV's incoming at the same time. So while it is tracking one BRV it must track every other BRV it has detected and continue its search for new threats. We thus have a resource allocation problem, and desire a measurement strategy that ensures the necessary track accuracy to perform a successful hand-off to a targeting radar using as few measurements as possible.

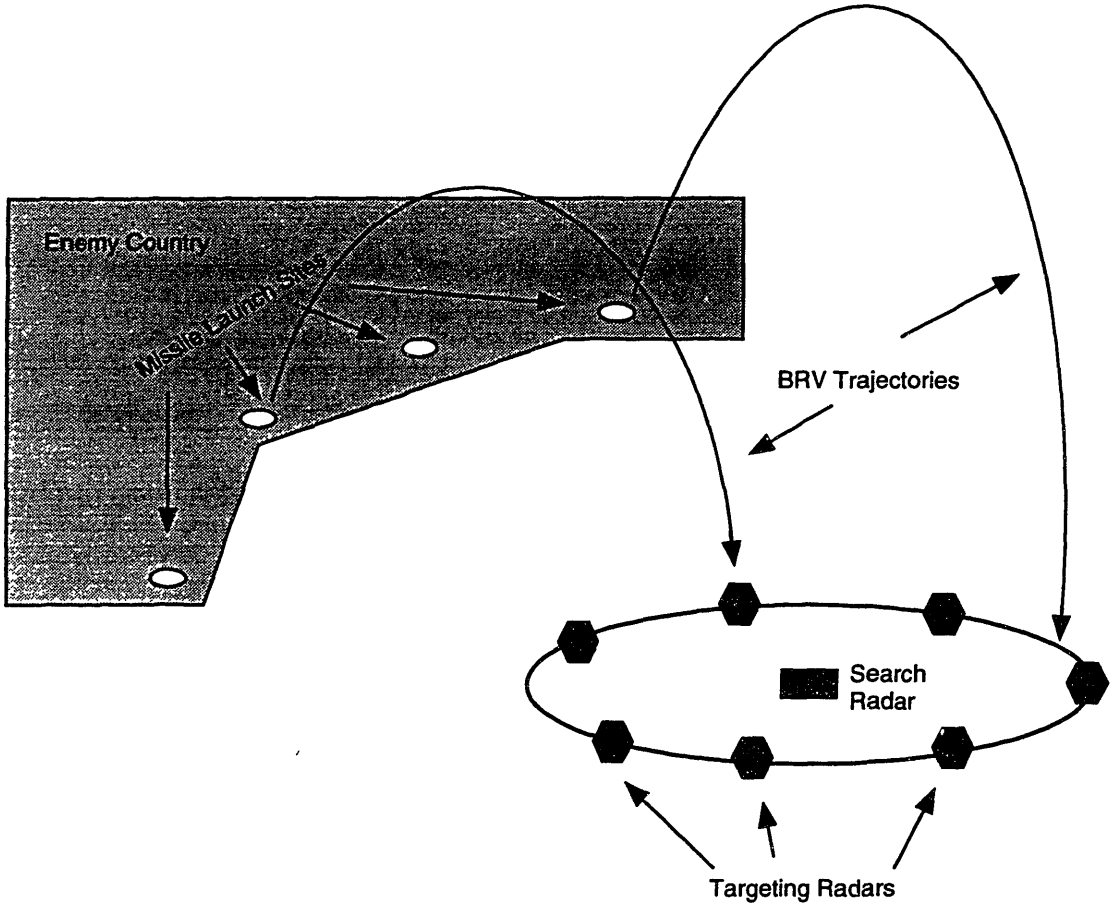


Figure 6-1: Overall scenario for this application. Intermediate-range missiles are used to attack a small region in a friendly country. A centrally located search radar detects the incoming missiles and follows them before handing off tracking to a targeting radar close to the missile's target.

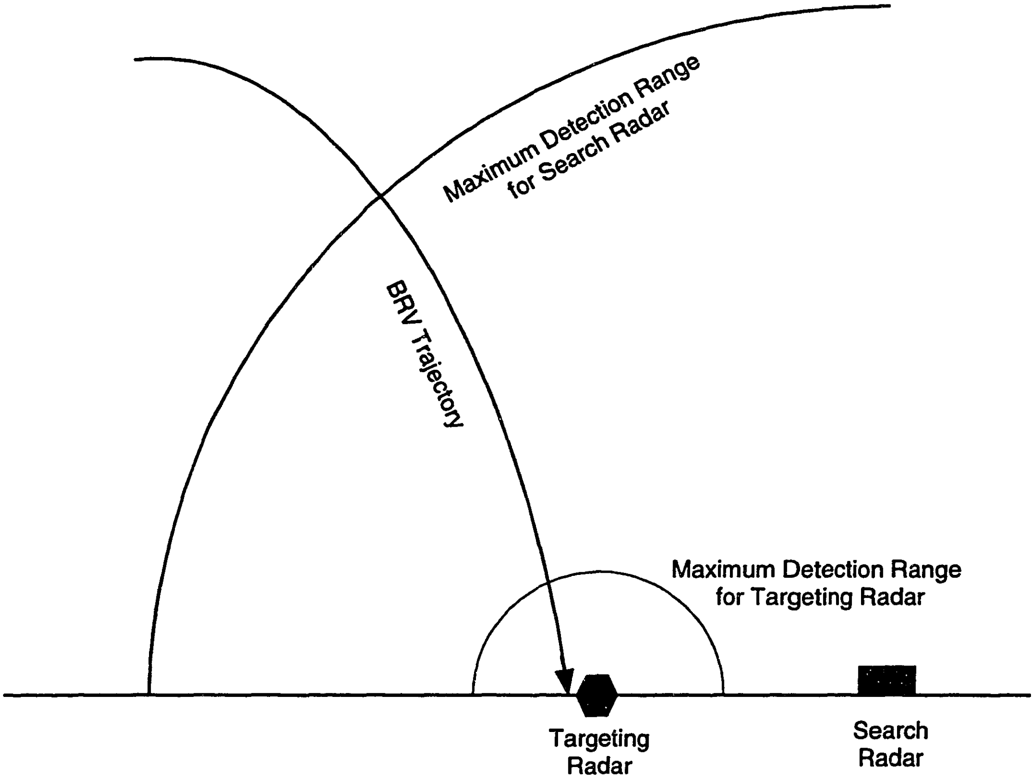

Maximum Detection Range for Search Radar
BRV Trajectory
Maximum Detection Range for Targeting Radar
Targeting Radar
Search Radar

Figure 6-2: A diagram showing the various stages of tracking. Tracking is begun by a long range search radar and is handed off to a less powerful targeting radar when the missile comes within range.

As we shall see in the succeeding sections, this is not a simple problem. Both the state propagation and measurement equations are highly nonlinear, and the dimensionality of the system is high (7 states with 3 measurements). If we base our measurement strategy on the current state estimate and its estimated covariance matrix we have 35 separate inputs to our policy.

## 6.2 Formulation of Problem

Before getting into the details of the problem formulation a few notational ground-rules need to be established. The relative complexity of this application has necessitated the reuse of a variety of symbols. There are two major contexts in this chapter; one relating to the learning algorithm and overall problem formulation, the other to the details of the system's equations of motion. We have avoided reuse of symbols within these contexts, but there may still be some confusion in situations where the two contexts interact. In situations where confusion seems likely we will try to explicitly point out what the symbols refer to.

### 6.2.1 State and measurement vectors

The first of these awkward situations occurs right as we begin to describe our problem formulation. For the formulation, we need to refer both to $x$, $y$, and $z$ coordinates, for which we shall use italics, and the state vector $\mathbf{x}$ and output vector $\mathbf{y}$ for which we shall use boldface type. As usual, dotted quantities refer to the derivative with respect to time. Our BRV's state vector is then:

$$\mathbf{x}(t) = [x'(t) \quad y'(t) \quad z'(t) \quad \dot{x}'(t) \quad \dot{y}'(t) \quad \dot{z}'(t) \quad \alpha(t)]^T. \tag{6.1}$$

Here $\alpha$ is a state relating to the drag force on the reentry vehicle and the primes on the coordinates refer to a radar-based coordinate system. In the interest of brevity, detailed explanations of the meaning of many of the quantities appearing in this section will be omitted. Interested readers are encouraged to pursue these details by reading Section 6.3, which provides more complete explanations.

For our measurement vector we use the standard radar returns of range, elevation, and azimuth:

$$\mathbf{y}_k = \begin{bmatrix} R_k \\ \theta_k \\ \psi_k \end{bmatrix} \tag{6.2}$$

The $\theta_k$ used here is an angle and should not be confused with the use of $\theta$ for the feature vector. Nor should the range measurement $R_k$ be confused with the use of $R$ in contexts relating to the sensor noise covariance.

### 6.2.2 State propagation and measurement equations

As usual, the state $\mathbf{x}(t)$ is propagated according to

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t)) + w(t). \tag{6.3}$$

In this application there was no need for consideration of an external input $u(t)$. We then have

$$f(\mathbf{x}(t)) = [f_1(\mathbf{x}(t)) \quad \cdots \quad f_7(\mathbf{x}(t))]^T \tag{6.4}$$

and

$$w(t) = [0 \quad 0 \quad 0 \quad w_4(t) \quad w_5(t) \quad w_6(t) \quad w_7(t)], \tag{6.5}$$

where

$$\begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ f_3(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \dot{x}' \\ \dot{y}' \\ \dot{z}' \end{bmatrix}, \tag{6.6}$$

$$\begin{aligned} \begin{bmatrix} \ddot{x}' \\ \ddot{y}' \\ \ddot{z}' \end{bmatrix} &= \begin{bmatrix} f_4(\mathbf{x}) \\ f_5(\mathbf{x}) \\ f_6(\mathbf{x}) \end{bmatrix} + \begin{bmatrix} w_4 \\ w_5 \\ w_6 \end{bmatrix} \\ &= -\frac{V\alpha}{2} \begin{bmatrix} \dot{x}' \\ \dot{y}' \\ \dot{z}' \end{bmatrix} - g_c \begin{bmatrix} x' + \gamma_1 \\ y' + \gamma_2 \\ z' + \gamma_3 \end{bmatrix} + \omega^2 S \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} + 2\omega U \begin{bmatrix} \dot{x}' \\ \dot{y}' \\ \dot{z}' \end{bmatrix} + \omega^2 \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \begin{bmatrix} w_4 \\ w_5 \\ w_6 \end{bmatrix} \end{aligned} \tag{6.7}$$

and the equation for $\dot{\alpha}$ is given in Section 6.3. For brevity of notation, the time dependencies have been left out of the equation above. The derivation for Equation (6.7) takes into account centrifugal and coriolis forces, the variation of gravity with altitude, Earth oblateness effects, and changes in atmospheric characteristics with altitude.

Alongside this we have the measurement equation,

$$\mathbf{y}_k = h(\mathbf{x}(t_k)) + v_k \tag{6.8}$$

where

$$h(\mathbf{x}) = \begin{bmatrix} \sqrt{x'^2 + y'^2 + z'^2} \\ \sin^{-1}(z'/R) \\ \tan^{-1}(x'/y') \end{bmatrix} \tag{6.9}$$

and the statistics of $v_k$ depend on the action taken at time $t_k$.

### 6.2.3 Filter Equations

Taking a cue from the results presented by Mehra [34], we have elected to use an Iterated Extended Kalman Filter (IEKF) [25] for this application. The multiple iterations of the IEKF help compensate for the nonlinearities in the measurement equations. Our filter state, as before, consists solely of the state estimate and the estimated error covariance

$$\zeta_k = [\,\hat{\mathbf{x}}_k \quad \hat{P}_k\,]. \tag{6.10}$$

Our filter update blocks then become

$$\zeta_k^+ = \Phi\left(\zeta_k^-, \mathbf{y}_k, a_k\right) = \begin{bmatrix} \hat{\mathbf{x}}_{k,n}^+ \\ \hat{P}_{k,n}^+ \end{bmatrix} \tag{6.11}$$

and

$$\zeta_{k+1}^- = \Psi\left(\zeta_k^+\right) = \begin{bmatrix} \hat{\mathbf{x}}_k^+ + f\left(\hat{\mathbf{x}}_k^+\right)\Delta t + F\left(\hat{\mathbf{x}}_k^+\right) f\left(\hat{\mathbf{x}}_k^+\right)\frac{(\Delta t)^2}{2} \\ \left[I + F\left(\hat{\mathbf{x}}_k^+\right)\Delta t\right]\hat{P}_k^+\left[I + F\left(\hat{\mathbf{x}}_k^+\right)\Delta t\right]^T + W \end{bmatrix} \tag{6.12}$$

where

$$\hat{\mathbf{x}}_{k,i+1}^+ = \hat{\mathbf{x}}_k^- + K_{k,i}\left[\mathbf{y}_k - h_k\left(\hat{\mathbf{x}}_{k,i}^+\right) - H\left(\hat{\mathbf{x}}_{k,i}^+\right)\left(\hat{\mathbf{x}}_k^- - \hat{\mathbf{x}}_{k,i}^+\right)\right], \tag{6.13}$$

$$\hat{P}_{k,i+1}^+ = \left[I - K_{k,i}H\left(\hat{\mathbf{x}}_{k,i}^+\right)\right]\hat{P}_k^-, \tag{6.14}$$

$$K_{k,i} = \hat{P}_k^- H^T\left(\hat{\mathbf{x}}_{k,i}^+\right)\left[H\left(\hat{\mathbf{x}}_{k,i}^+\right)\hat{P}_k^- H^T\left(\hat{\mathbf{x}}_{k,i}^+\right) + V_k\right]^{-1}, \tag{6.15}$$

and $n$ is the number of iterations used in the IEKF.

### 6.2.4 Cost functions

Our cost function for this application will be somewhat more complex than those used in our previous examples. Its shape is determined by our overall requirement (enough accuracy to hand tracking over to a targeting radar) and the characteristics of our chosen measurement system.

The total single step cost consists of three components.

$$c_k = g\left(\zeta_k^-, a_k, y_k\right) = g_1\left(\zeta_k^+\right) + g_2\left(\zeta_k^+\right) + g_a\left(a_k\right) \tag{6.16}$$

The first of these is meant to ensure that the quality of the estimate does not degrade to the point where the radar is unlikely to successfully paint the target on its first try. The requirement is thus expressed by comparing the error in the direction perpendicular to the BRV's position vector with the radar's beam width. A slight modification is added in to take into account distortion of the phased-array radar's beam as it moves off the $z'$ axis. This portion of the cost can be expressed as

$$g_1\left(\zeta_k^+\right) = \begin{cases} C_1 & \left(\sigma^*/\hat{R} > \rho_1 \hat{R}/z'\right) \\ 0 & \mathit{elsewhere} \end{cases} \tag{6.17}$$

where the various quantities involved will be explained in more detail in Section 6.5.

Our terminal requirement is that when the estimated height reaches a certain level (we use 20,000 feet) the position estimate is good enough so that when tracking is handed off to a targeting radar, that radar will be able to find the BRV with a minimum of effort. Specifically, we require that the longest axis of the error ellipsoid defined by the covariance matrix be no greater than a specific value $\rho_2$, i.e.:

$$g_2\left(\zeta_k^+\right) = \begin{cases} C_2 & (\sigma_{\max} > \rho_2 \quad \& \quad \hat{h} < h^*) \\ 0 & \mathit{elsewhere} \end{cases} \tag{6.18}$$

Again, these quantities will be explained in more detail in Section 6.5.

The final portion of Equation (6.16) is $g_a(a_k)$, the cost relating specifically to the action.

The exact form that this takes will depend on the particular action space being treated; in this chapter we will be looking at both discrete action spaces (Section 6.8) and continuous-discrete action spaces (Section 6.9).

The total cost at any given state is then the expected sum of the $c_k$'s from the current step to the point at which $\hat{h} < 20{,}000$ and the terminal condition is evaluated, which we designate as step $N$.

$$J^{\pi}(\zeta) = \mathrm{E}\left[\sum_{k=0}^{N} g\left(\zeta_k^-, \pi\left(\zeta_k^-\right) y_k,\right) \mid \zeta_0^- = \zeta\right] \tag{6.19}$$

Our problem, as usual, is to get as close as possible to an optimal policy

$$\pi^*\left(\zeta_k^-\right) = \min_a Q^*\left(\zeta_k^-, a\right) \tag{6.20}$$

where

$$Q^*\left(\zeta_k^-, a\right) = E\left[g\left(\zeta_k^-, a, y_k\right) + J^*\left(\zeta_{k+1}^-\right)\right]. \tag{6.21}$$

Also as usual, the realities of the situation will force us to use features $\theta_k = \Gamma\left(\zeta_k^-\right)$ and function approximation so that the problem we actually solve looks more like

$$\pi_n(\theta_k) = \arg\min_a \tilde{Q}_n(\theta_k, a) \tag{6.22}$$

where $n$ is the iteration number. The feature extraction function $\Gamma$ used in this chapter is somewhat different from the ones used in earlier examples in this thesis; it is explained in more detail in Section 6.6.

## 6.3 Equations of Motion

Parts of the derivation to follow were taken from Mehra [34] and Larson et. al [31], but considerable detail has been added to the drag and atmospheric models. The equations are expressed in the coordinate system described by Mehra, which is shown in Figure 6-3.

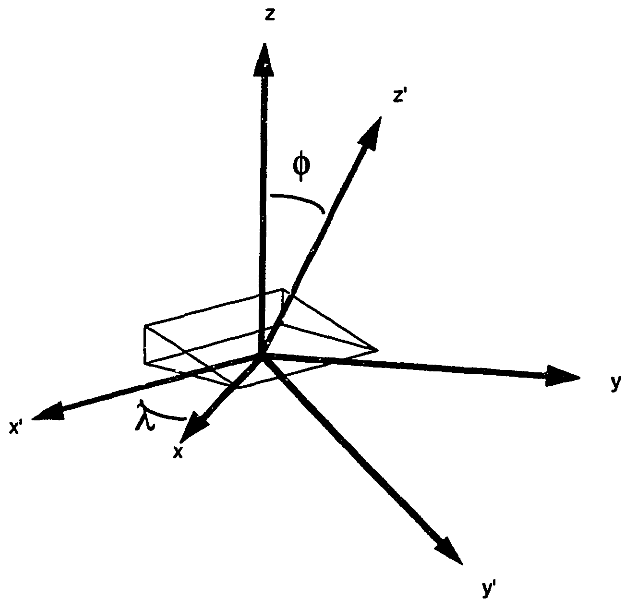


Figure 6-3: Transformation from LVLH ($x$, $y$, and $z$) to Radar-based ($x'$,$y'$,$z'$) coordinates. The $x$ axis points to the east, the $y$ axis points north, and the $z$ axis points up.

The unprimed coordinates refer to a local-vertical, local-horizontal (LVLH) system with the $z$ axis pointing up and the $y$ axis pointing north. The primed coordinates, in which the equations of motions are expressed, are transformed by a rotation in elevation of angle $\phi$ and a rotation in azimuth of angle $\lambda$. This leaves $z'$ pointing along the normal of the radar's face and $y'$ directed along the radar's face in the direction of the radar's horizontal facing. The transformation matrix from $(x, y, z)$ coordinates to $(x', y', z')$ coordinates is given by

$$T = \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{bmatrix} = \begin{bmatrix} \cos\lambda & -\sin\lambda & 0 \\ \cos\phi\sin\lambda & \cos\phi\cos\lambda & -\sin\phi \\ \sin\phi\sin\lambda & \sin\phi\cos\lambda & \cos\phi \end{bmatrix} \tag{6.23}$$

and appears in several places in the equations to follow.

The seventh state in our state vector, $\alpha$, is an amalgamation of drag-related quantities defined as

$$\alpha = \eta_0 \rho\,(h)\, C_D\,(V, a) \tag{6.24}$$

where $\rho$ is the atmospheric density, $h$ is the height of the BRV, $V$ is its velocity, $a$ is the speed of sound, and

$$\eta_0 = Ag/W. \tag{6.25}$$

Here $A$ is the effective drag area of the BRV, $g$ is the acceleration due to gravity at sea level, and $W$ is the vehicle's weight. The large number of symbols which are used in this derivation may make it hard to keep track of them all; for this reason we have created a comprehensive list (Table 6.10) of symbols used in this application that can be found at the end of the chapter.

Recall Equation (6.7) from Section 6.2.2. We repeat it here for easier reference.

$$\begin{bmatrix} \ddot{x}' \\ \ddot{y}' \\ \ddot{z}' \end{bmatrix} = -\frac{V\alpha}{2}\begin{bmatrix} \dot{x}' \\ \dot{y}' \\ \dot{z}' \end{bmatrix} - g_c \begin{bmatrix} x'+\gamma_1 \\ y'+\gamma_2 \\ z'+\gamma_3 \end{bmatrix} + \omega^2 S \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} + 2\omega U \begin{bmatrix} \dot{x}' \\ \dot{y}' \\ \dot{z}' \end{bmatrix} + \omega^2 \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \begin{bmatrix} w_4 \\ w_5 \\ w_6 \end{bmatrix} \tag{6.26}$$

The first term in this equation relates to the drag force on the reentry vehicle. As we have already noted, $V$ is the BRV's velocity and $\alpha$ is our drag state. The next term in the equation models the effect of gravity on the vehicle; $g_c$ is the local acceleration due to gravity with an additional directional normalization factor

$$g_c = GM_e \big/ r^3. \tag{6.27}$$

Here $G$ is the universal gravitational constant, $M_e$ is the mass of the Earth, and $r$ is the distance from the center of the Earth to the BRV, given by

$$r = \left[a_0^2 + \left(x'^2 + y'^2 + z'^2\right) - 2c_2\left(T_{12}x' + T_{22}y' + T_{32}z'\right) + 2c_3\left(T_{13}x' + T_{23}y' + T_{33}z'\right)\right]^{1/2} \tag{6.28}$$

and shown graphically in Figure 6-4. In Equation (6.28) $a_0$ is the distance from the center of the Earth to the radar station, given by

$$a_0 = \sqrt{a_1^2 \cos^2 \mu + a_2^2 \sin^2 \mu} \tag{6.29}$$

where $\mu$ is the geodetic latitude of the radar station, $a_1$ and $a_2$ are

$$a_1 = \frac{a_e}{\sqrt{1 - e^2 \sin^2 \mu}} + h_s \tag{6.30}$$

and

$$a_2 = \frac{a_e (1 - e^2)}{\sqrt{1 - e^2 \sin^2 \mu}} + h_s, \tag{6.31}$$

$a_e$ is the equatorial radius of the Earth, $e$ is the Earth's eccentricity, and $h_s$ is the altitude of the radar station. A detailed explanation of this geometry can be found in Bate [11]. The quantities $c_2$ and $c_3$ appearing in Equation (6.28) are given by

$$c_2 = (a_1 - a_2) \sin \mu \cos \mu \tag{6.32}$$

and

$$c_3 = a_1 \cos^2 \mu + a_2 \sin^2 \mu. \tag{6.33}$$

Finally, returning to the gravity-related term of Equation (6.26), we have

$$\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{bmatrix} = \begin{bmatrix} -T_{12}c_2 + T_{13}c_3 \\ -T_{22}c_2 + T_{23}c_3 \\ -T_{32}c_2 + T_{33}c_3 \end{bmatrix}. \tag{6.34}$$

The third and final terms on the right-hand side of Equation (6.26) deal with the centripetal acceleration of the BRV. In these terms, $\omega$ represents the Earth's rotational rate, $S$ is a matrix given by

$$S = T \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sin^2 \mu & -\sin \mu \cos \mu \\ 0 & -\sin \mu \cos \mu & \cos^2 \mu \end{bmatrix} T^T, \tag{6.35}$$

and the $\beta$'s are

$$\begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = T \begin{bmatrix} 0 \\ -a_1 \cos \mu \sin \mu \\ a_1 \cos^2 \mu \end{bmatrix}. \tag{6.36}$$

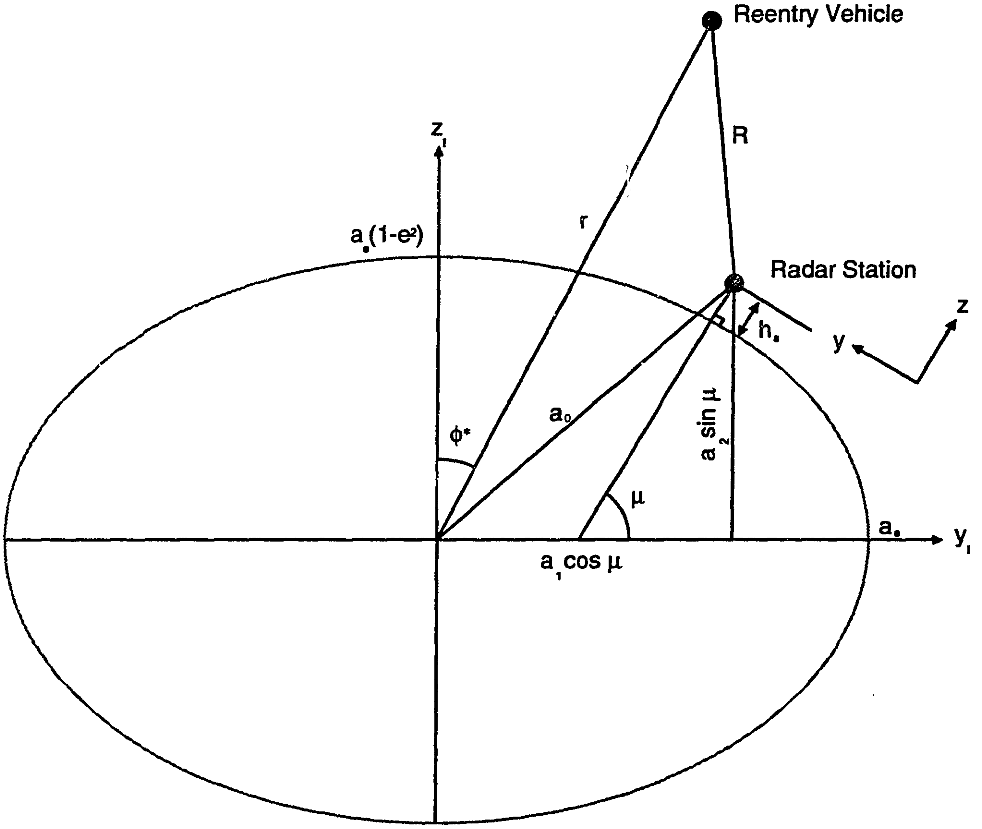


Figure 6-4: Geometry of the problem relative to the Earth. Shows various quantities used in equations of motion below.

The remaining term in Equation (6.26) models the coriolis acceleration on the BRV. In it, $U$ is a matrix given by

$$U = T \begin{bmatrix} 0 & \sin\mu & -\cos\mu \\ -\sin\mu & 0 & 0 \\ \cos\mu & 0 & 0 \end{bmatrix} T^T. \tag{6.37}$$

### 6.3.1 Drag Model

To capture the variation of the BRV's drag coefficient with Mach number we use the model proposed by Barbera [10]. Based on a combination of experimental data and computational fluid dynamic analysis Barbera puts forth an exponential model

$$C_D = \eta_3 \left(\frac{V}{a}\right)^{\eta_4} \tag{6.38}$$

where the coefficients $\eta_3$ and $\eta_4$ depend on the geometry of the vehicle and whether the vehicle is operating at supersonic or hypersonic Mach numbers. For our application we have assumed a nose-to-base radius ratio of 0.2 and a cone half-angle of 7° for the vehicle, which result in coefficient values

$$\begin{array}{lll} \eta_3 = 0.4720 & \eta_4 = -0.8683 & (V/a \leq 10) \\ \eta_3 = 0.0812 & \eta_4 = -0.1040 & (V/a > 10) \end{array}.$$

## 6.3.2 Atmospheric Models

The atmospheric models used here were generated using tables for the standard atmosphere found in McCormick [33]. The models for both density and speed of sound are divided into two regions, with the transition occurring at the edge of the troposphere.

### 6.3.2.1 Density

Data for the density of the standard atmosphere as a function of height above sea level is plotted in Figure 6-5. Two separate regions are clearly visible in the graph, with the transition at an altitude of roughly 37,000 feet. Applying two linear curve fits on our log-log scale, we arrive at a model

$$\rho(h) = \eta_1 e^{\eta_2 h} \tag{6.39}$$

where

$$\begin{array}{lll} \eta_1 = 2.444 \cdot 10^{-3} & \eta_2 = -3.364 \cdot 10^{-5} & (h \leq 37,000\ ft) \\ \eta_1 = 3.974 \cdot 10^{-3} & \eta_2 = -4.781 \cdot 10^{-5} & (h > 37,000\ ft) \end{array}.$$

The BRV's height can be calculated from our state vector using the relations

$$h = r - a_e \sqrt{\frac{1 - e^2}{1 - e^2 \sin^2 \phi^*}} \tag{6.40}$$

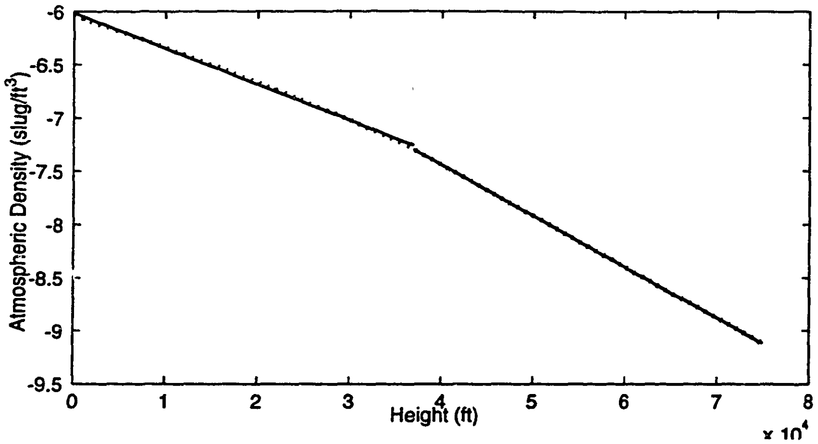


Figure 6-5: The density of the standard atmosphere as a function of height, with bilinear exponential curve fit.

and

$$\cos\phi^* = \tfrac{1}{r}\left[a_2 \sin\mu + (T_{12}x' + T_{22}y' + T_{32}z')\cos\mu + (T_{13}x' + T_{23}y' + T_{33}z')\sin\mu\right]. \quad (6.41)$$

The angle $\phi^*$ is marked in Figure 6-4.

#### 6.3.2.2 Speed of Sound

Data for the speed of sound in the standard atmosphere as a function of height above sea level is plotted in Figure 6-6. Again, a clear change in behavior is evident at 37,000 feet, though this time the relationship is linear, not exponential. A simple curve fit yields a model

$$a = \eta_5 + \eta_6 h \quad (6.42)$$

where the two coefficients are given by

$$\begin{array}{lll} \eta_5 = 1118 & \eta_6 = -4.09e - 3 & (h \leq 37{,}000\ ft) \\ \eta_5 = 968.08 & \eta_6 = 0 & (h > 37{,}000\ ft) \end{array}$$

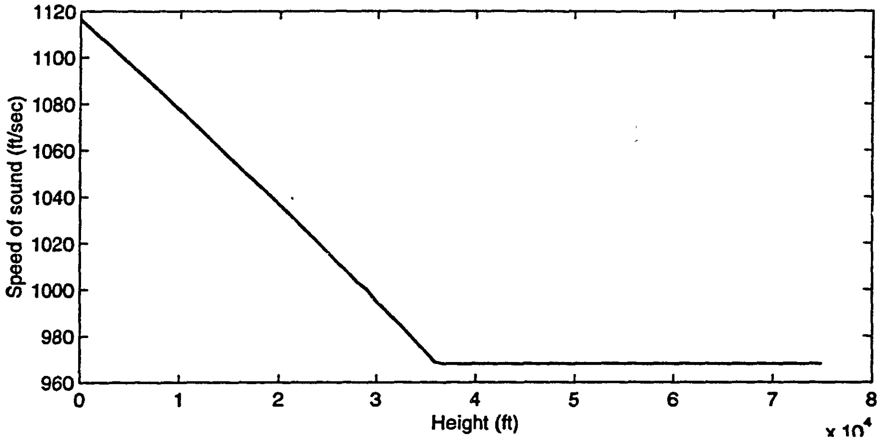


Figure 6-6: The speed of sound for the standard atmosphere as a function of height.

### 6.3.3 Derivatives

Now that we have described the various models which go into the equation for our drag state $\alpha$, we can calculate its time-derivative (with additive noise):

$$\dot{\alpha} = f_7(x) + w_7 = \frac{d\alpha}{dr}\dot{r} + \frac{\alpha\eta_4}{V}\dot{V} + w_7 \tag{6.43}$$

where

$$V = \sqrt{(\dot{x}')^2 + (\dot{y}')^2 + (\dot{z}')^2} \tag{6.44}$$

and

$$\dot{V} = \frac{1}{V}(\dot{x}'\ddot{x}' + \dot{y}'\ddot{y}' + \dot{z}'\ddot{z}') = b_1/V. \tag{6.45}$$

In Equation (6.43) and throughout this section, we make an approximation by equating changes in $h$ with changes in $r$.

In addition to the equation for $\dot{\alpha}$, we also need to calculate the elements of the Jacobian matrices $F$ and $H$. For the former we can show with some (not so simple) algebra that

$$F(x) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ F_{41} & F_{42} & F_{43} & F_{44} & F_{45} & F_{46} & F_{47} \\ F_{51} & F_{52} & F_{53} & F_{54} & F_{55} & F_{56} & F_{57} \\ F_{61} & F_{62} & F_{63} & F_{64} & F_{65} & F_{66} & F_{67} \\ F_{71} & F_{72} & F_{73} & F_{74} & F_{75} & F_{76} & F_{77} \end{bmatrix} \tag{6.46}$$

where

$$F_{41} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial x'}\dot{x}' - \left[\frac{\partial g_c}{\partial x'}(x' + \gamma_1) + g_c\right] + \omega^2 S_{11} \tag{6.47}$$

$$F_{42} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial y'}\dot{x}' - \frac{\partial g_c}{\partial y'}(x' + \gamma_1) + \omega^2 S_{12} \tag{6.48}$$

$$F_{43} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial z'}\dot{x}' - \frac{\partial g_c}{\partial z'}(x' + \gamma_1) + \omega^2 S_{13}, \tag{6.49}$$

$$F_{44} = -\frac{\alpha V}{2}\left[1 + \left(\frac{\dot{x}'}{V}\right)^2 (\eta_4 + 1)\right] + 2\omega U_{11} \tag{6.50}$$

$$F_{45} = -\frac{\alpha \dot{x}'\dot{y}'}{2V}(1 + \eta_4) + 2\omega U_{12} \tag{6.51}$$

$$F_{46} = -\frac{\alpha \dot{x}'\dot{z}'}{2V}(1 + \eta_4) + 2\omega U_{13}, \tag{6.52}$$

$$F_{47} = -\frac{V\dot{x}'}{2}, \tag{6.53}$$

$$F_{51} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial x'}\dot{y}' - \frac{\partial g_c}{\partial x'}(y' + \gamma_2) + \omega^2 S_{21} \tag{6.54}$$

$$F_{52} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial y'}\dot{y}' - \left[\frac{\partial g_c}{\partial y'}(y' + \gamma_2) + g_c\right] + \omega^2 S_{22} \tag{6.55}$$

$$F_{53} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial z'}\dot{y}' - \frac{\partial g_c}{\partial z'}(y' + \gamma_2) + \omega^2 S_{23}, \tag{6.56}$$

$$F_{54} = -\frac{\alpha \dot{x}'\dot{y}'}{2V}(1 + \eta_4) + 2\omega U_{21} \tag{6.57}$$

$$F_{55} = -\frac{\alpha V}{2}\left[1 + \left(\frac{\dot{y}'}{V}\right)^2 (\eta_4 + 1)\right] + 2\omega U_{22} \tag{6.58}$$

$$F_{56} = -\frac{\alpha \dot{z}'\dot{y}'}{2V}(1 + \eta_4) + 2\omega U_{23}, \tag{6.59}$$

$$F_{57} = -\frac{V\dot{y}'}{2}, \tag{6.60}$$

$$F_{61} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial x'}\dot{z}' - \frac{\partial g_c}{\partial x'}(z' + \gamma_3) + \omega^2 S_{31} \tag{6.61}$$

$$F_{62} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial y'}\dot{z}' - \frac{\partial g_c}{\partial y'}(z' + \gamma_3) + \omega^2 S_{32} \tag{6.62}$$

$$F_{63} = -\frac{V}{2}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial z'}\dot{z}' - \left[\frac{\partial g_c}{\partial z'}(z' + \gamma_3) + g_c\right] + \omega^2 S_{33}, \tag{6.63}$$

$$F_{64} = -\frac{\alpha \dot{x}'\dot{z}'}{2V}(1 + \eta_4) + 2\omega U_{31} \tag{6.64}$$

$$F_{65} = -\frac{\alpha \dot{z}'\dot{y}'}{2V}(1 + \eta_4) + 2\omega U_{32} \tag{6.65}$$

$$F_{66} = -\frac{\alpha V}{2}\left[1 + \left(\frac{\dot{z}'}{V}\right)^2 (\eta_4 + 1)\right] + 2\omega U_{33}, \tag{6.66}$$

$$F_{67} = -\frac{V\dot{z}'}{2}, \tag{6.67}$$

$$
\begin{aligned}
F_{71} &= \frac{\dot{\alpha}}{\alpha}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial x'} + \frac{\partial \alpha}{\partial r}\frac{\partial \dot{r}}{\partial x'} + \frac{\alpha \eta_4 \eta_6^2 \dot{r}}{a^2}\frac{\partial r}{\partial x'} && (6.68)\\
F_{72} &= \frac{\dot{\alpha}}{\alpha}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial y'} + \frac{\partial \alpha}{\partial r}\frac{\partial \dot{r}}{\partial y'} + \frac{\alpha \eta_4 \eta_6^2 \dot{r}}{a^2}\frac{\partial r}{\partial y'} && (6.69)\\
F_{73} &= \frac{\dot{\alpha}}{\alpha}\frac{\partial \alpha}{\partial r}\frac{\partial r}{\partial z'} + \frac{\partial \alpha}{\partial r}\frac{\partial \dot{r}}{\partial z'} + \frac{\alpha \eta_4 \eta_6^2 \dot{r}}{a^2}\frac{\partial r}{\partial z'}, && (6.70)
\end{aligned}
$$

$$
\begin{aligned}
F_{74} &= \frac{\partial \alpha}{\partial r}\frac{\partial \dot{r}}{\partial \dot{x}'} + \frac{\eta_4}{V^2}\left(\dot{\alpha}\dot{x}' + \alpha\ddot{x}' - \frac{2\alpha b_1 \dot{x}'}{V^2}\right) && (6.71)\\
F_{75} &= \frac{\partial \alpha}{\partial r}\frac{\partial \dot{r}}{\partial \dot{y}'} + \frac{\eta_4}{V^2}\left(\dot{\alpha}\dot{y}' + \alpha\ddot{y}' - \frac{2\alpha b_1 \dot{y}'}{V^2}\right) && (6.72)\\
F_{76} &= \frac{\partial \alpha}{\partial r}\frac{\partial \dot{r}}{\partial \dot{z}'} + \frac{\eta_4}{V^2}\left(\dot{\alpha}\dot{z}' + \alpha\ddot{z}' - \frac{2\alpha b_1 \dot{z}'}{V^2}\right), && (6.73)
\end{aligned}
$$

and

$$F_{77} = \frac{\dot{\alpha}}{\alpha}. \qquad (6.74)$$

In the above equations

$$\frac{d\alpha}{dr} = \alpha\left(\eta_2 - \frac{\eta_4\eta_6}{a}\right), \qquad (6.75)$$

$$\dot{r} = \frac{1}{r}\left[\left(x'\dot{x}' + y'\dot{y}' + z'\dot{z}'\right) - c_2\left(T_{12}\dot{x}' + T_{22}\dot{y}' + T_{32}\dot{z}'\right) + c_3\left(T_{13}\dot{x}' + T_{23}\dot{y}' + T_{33}\dot{z}'\right)\right], \qquad (6.76)$$

$$
\begin{aligned}
\frac{\partial r}{\partial x'} &= \frac{1}{r}\left(x' - c_2 T_{12} + c_3 T_{13}\right) && (6.77)\\
\frac{\partial r}{\partial y'} &= \frac{1}{r}\left(y' - c_2 T_{22} + c_3 T_{23}\right) && (6.78)\\
\frac{\partial r}{\partial z'} &= \frac{1}{r}\left(z' - c_2 T_{32} + c_3 T_{33}\right), && (6.79)
\end{aligned}
$$

$$\begin{aligned} \frac{\partial \dot{r}}{\partial x'} &= \frac{1}{r}\left(\dot{x}' - \frac{\partial r}{\partial t}\frac{\partial r}{\partial x'}\right) && (6.80) \\ \frac{\partial \dot{r}}{\partial y'} &= \frac{1}{r}\left(\dot{y}' - \frac{\partial r}{\partial t}\frac{\partial r}{\partial y'}\right) && (6.81) \\ \frac{\partial \dot{r}}{\partial z'} &= \frac{1}{r}\left(\dot{z}' - \frac{\partial r}{\partial t}\frac{\partial r}{\partial z'}\right), && (6.82) \end{aligned}$$

and

$$\begin{aligned} \frac{\partial \dot{r}}{\partial \dot{x}'} &= \frac{1}{r}(x' - c_2 T_{12} + c_3 T_{13}) && (6.83) \\ \frac{\partial \dot{r}}{\partial \dot{y}'} &= \frac{1}{r}(y' - c_2 T_{22} + c_3 T_{23}) && (6.84) \\ \frac{\partial \dot{r}}{\partial \dot{z}'} &= \frac{1}{r}(z' - c_2 T_{32} + c_3 T_{33}). && (6.85) \end{aligned}$$

For the output Jacobian $H$ some (relatively simple, this time) algebra gives

$$\begin{bmatrix} \frac{x'}{R} & \frac{y'}{R} & \frac{z'}{R} & 0 & 0 & 0 & 0 \\ \frac{-z'x'}{R^3\sqrt{1-(z'/R)^2}} & \frac{-z'y'}{R^3\sqrt{1-(z'/R)^2}} & \frac{\sqrt{1-(z'/R)^2}}{R} & 0 & 0 & 0 & 0 \\ \frac{y'}{(y'^2+x'^2)} & \frac{-x'}{(y'^2+x'^2)} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \qquad (6.86)$$

## 6.4 Radar Measurement Considerations

Since we are taking the trouble to produce a reasonably realistic model of the BRV's reentry dynamics, to some extent we are obligated to model our measurement subsystem with a similar level of fidelity. While we can't take into account all the processing details that go into radar measurements in actual systems, we can model some of the higher-level aspects of a radar system.

### 6.4.1 Painting the target

One of these aspects involves the need for a radar beam to "paint" a target in order to measure its position. By this we mean that the radar's beam must be directed so that the

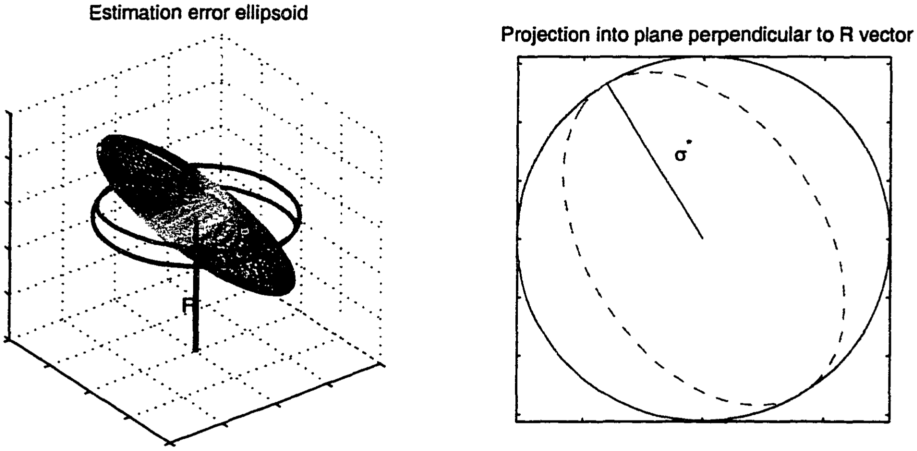

Estimation error ellipsoid
R
Projection into plane perpendicular to R vector
σ*

Figure 6-7: The estimation error ellipsoid and its projection onto the plane perpendicular to $R$, the vector from the radar to the estimated location of the BRV. The semimajor axis of this projected ellipse is $\sigma^*$.

target is inside of the cone formed by the radar beam in order to get the reflected signal necessary for a measurement. Because measurements are taken at discrete intervals, there will always be a degree of uncertainty about the target's position. This in turn leaves open the possibility that the radar will not successfully paint the target on its first try, necessitating a search to re-aquire the target.

Actual radars use a system of "gates" both to aid in reaquiring target tracks on successive measurements and to distinguish bogus returns and multiple targets. Without getting into the details of any particular gating system, we can use the same overall concept with the covariance estimate we get from our filter. The $\hat{P}$ matrix we get from our IEKF defines an error ellipsoid centered on the current estimate as shown in Figure 6-7. To determine the likelihood of successfully painting our target with a pulse centered on the current $\hat{x}$ we need to find the projection of this ellipsoid onto the plane perpendicular to the BRV's position vector (in radar coordinates) which passes through $\hat{x}$. This is also shown in the figure.

This projection forms an ellipse, and is characterized by the matrix $\hat{P}'_\perp$ formed by eliminating the third row and column from $\hat{P}'$:

$$\hat{P}'_{\perp} = \begin{bmatrix} \hat{P}'_{11} & \hat{P}'_{12} \\ \hat{P}'_{21} & \hat{P}'_{22} \end{bmatrix} \tag{6.87}$$

where $\hat{P}'$ is the transformed covariance matrix

$$\hat{P}' = T_R \hat{P} T_R^T \tag{6.88}$$

and the transformation matrix $T_R$ is

$$T_R = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\theta\sin\psi & \sin\theta\cos\psi & -\cos\theta \\ \cos\theta\sin\psi & \cos\theta\cos\psi & \sin\theta \end{bmatrix} \tag{6.89}$$

where the angles $\theta$ and $\psi$ in the above equation are simply the elevation and azimuth angles from Equation (6.2).

Once we have $\hat{P}'_{\perp}$ we can find the semimajor axis of the ellipse simply by taking the square root of the matrix's larger eigenvalue

$$\sigma^* = \mathrm{sqrt}\left(\max\left(\mathrm{eig}\left(\hat{\mathrm{P}}'_{\perp}\right)\right)\right). \tag{6.90}$$

Having established a measure for the uncertainty of our estimate in the beam pointing directions, we will need to establish a constraint to ensure that we successfully paint the target with our radar beam on the first try. Our constraint will probably look something like

$$\sigma^*/R < \kappa\theta_b/2 \tag{6.91}$$

where $\kappa$ is some fraction of the half-beamwidth $\theta_b/2$.

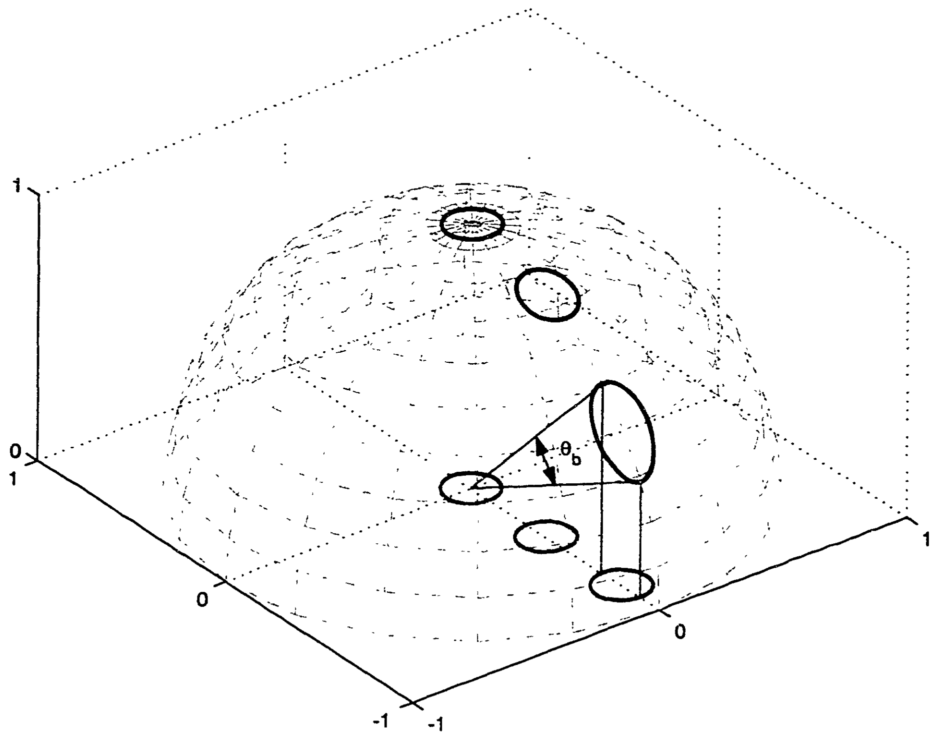


Figure 6-8: As a phased-array radar points further from its boresight the beam stretches in one direction.

### 6.4.2 Phased-array beam spreading

Our assumption that our search radar is a phased-array system adds an additional complexity to the considerations discussed in Section 6.4.1. When using a phased-array radar, the beamwidth $\theta_b$ in Equation (6.91) varies as the beam moves away from the $z'$ axis [54]. This behavior is shown in Figure 6-8. The beam distortion only occurs in the elevation ($\theta$) direction, not the azimuth ($\psi$) direction. The actual beamwidth is given by

$$(\theta_b)_{\text{actual}} = (\theta_b)_{\text{base}}/\cos\theta = (\theta_b)_{\text{base}}(R/z') \tag{6.92}$$

which means that our pointing constraint becomes

$$\sigma^*/R < \frac{\kappa R\theta_b}{2z'}) \tag{6.93}$$

instead of the one shown in Equation (6.91).

### 6.4.3 Sensor Noise Model

Determining the error statistics for a single radar measurement is no easy task. A seemingly endless series of error sources exist, many of which are highly dependent on the particular circumstances in which the measurement is taken. For the purposes of this thesis it suffices to model some of the higher-level effects in order to get the flavor of the measurement system without getting into the fine details.

When a radar system takes a measurement, what it is actually doing is sending out a pulse of energy and listening for the reflected echo. The accuracy of the resulting measurement depends on three basic things:

- whether or not the reflected signal is detected at the receiver.
- the strength of the reflected signal relative to the receiver noise.
- noise sources of fixed magnitude inherent in the system.

Of these, the first two are related in that the probability of detection $P_D$ is a function of the strength of the received signal. The received energy $E_r$ is in turn related to the transmitted energy $E_t$ according to

$$E_r \sim E_t \big/ R^4. \tag{6.94}$$

The probability of detection was calculated based on a graph found in Skolnik's text [54]. A probability of false alarm of $P_{FA} = 10^{-6}$ was used, together with the assumption that at the radar's maximum range ($R_{\max}$) when the maximum possible power is transmitted ($E_t = E_{max}$), $P_D = 0.8$. The resulting curve is shown in Figure 6-9.

Assuming the target is detected, the accuracy of the resulting measurement depends on the amount of energy transmitted and the range. We can keep the terms of the calculation general by specifying the normalized energy $E(a_k)$ as the fraction of the maximum energy

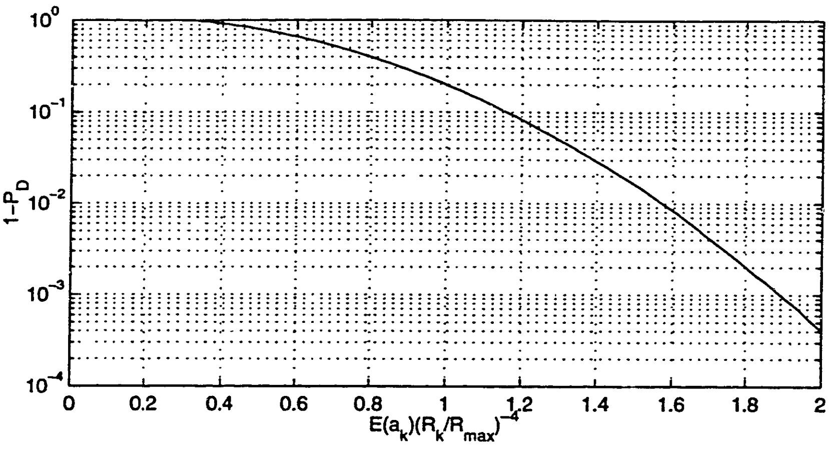


Figure 6-9: Probability of a missed detection as a function of the received energy.

which the radar can transmit

$$E(a_k) = E_t(a_k)/E_{t_{\max}} \tag{6.95}$$

and calculating the noise covariance as a fraction of the noise level achieved when the maximum energy is transmitted at the maximum range, $V_{\max}$:

$$V_k' = \frac{(R_k/R_{\max})^4}{E(a_k)} V_{\max}. \tag{6.96}$$

Remember that we are dealing with the filter here, so $a$'s are actions (not the speed of sound) and $V$'s are covariance matrices (not velocities). After calculating the base noise level $V_k'$ we need to adjust for the beam widening discussed in Section 6.4.2:

$$V_k'' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & R_k/z_k' & 0 \\ 0 & 0 & 1 \end{bmatrix} V_k'. \tag{6.97}$$

At short ranges, the resulting noise covariance may be low enough that it is overwhelmed by fixed noise sources. To model this, we bound our sensor noise below by $V_{\min}$,

$$V_k''' = \max(V_k'', V_{\min}). \tag{6.98}$$

| Symbol | Description | Value |
|---|---|---|
| $V_{\min}$ | Minimum level for noise variance | $\operatorname{diag}\left(\begin{bmatrix} 400 \\ 10^{-6} \\ 10^{-6} \end{bmatrix}\right)$ |
| $V_{\max}$ | Noise variance when maximum power is used at maximum range | $\operatorname{diag}\left(\begin{bmatrix} 4 \cdot 10^{4} \\ 4.9 \cdot 10^{-5} \\ 4.9 \cdot 10^{-5} \end{bmatrix}\right)$ |
| $R_{\max}$ | Approximate range at which tracking begins | $400,000$ ft |
| $P_{FA}$ | Probability of false alarm | $10^{-6}$ |
| $P_D$ | Probability of detection when maximum power is used at maximum range | 0.8 |

Table 6.1: Statistics for modeling of sensor noise from radar measurements.

Of course, if the target is not detected our measurement noise is effectively infinite, so our final expression becomes

$$V_k = \begin{cases} V_k''' & \text{with probability } P_D\left(E\left(a_k\right), R_k\right) \\ V_\infty & \text{with probability } 1 - P_D\left(E\left(a_k\right), R_k\right) \end{cases} \tag{6.99}$$

where $V_\infty$ is some arbitrarily chosen large positive number. The parameters used in this section and their values are summarized in Table 6.1.

## 6.5 Cost Model

As described in Sections 6.1 and 6.2.4, our objective is to use our search radar to localize the BRV's position well enough to allow a smooth hand-off to a targeting radar using as few measurements as possible. At first it might seem that a cost function descibing this objective would be relatively straightforward. Two parts should suffice: the total number of measurements taken plus a large fixed penalty if the desired "smooth hand-off" is not achieved. The cost model we use in this application is somewhat more complex, however. A third term is added, imposing an additional fixed cost if certain accuracy limits are exceeded during the period before the hand-off to the targeting radar.

The addition of this third term becomes necessary because of the nature of the radar measurement system. With many conventional measurement systems, the optimal solution to our problem would be to wait until just before the hand-off before taking any measurements. With a radar system, however, when the target's position uncertainty gets too large the possibility of track loss occurs. Once the track is lost, a new search requiring a number of measurements must be conducted to reaquire it. Ideally, a learning system should be able to take this behavior into account automatically by recording the actual number of measurements needed to reaquire the target when a track is lost. However, simulation of this track loss and reaquisition procedure would require a level of detail that is beyond the scope of this thesis. Instead, we constrain our solutions by picking an error level at which we are confident that track loss will not occur and penalizing any policy which allows the error to grow beyond that level.

The quantities that must be calculated to measure this constraint have been described in Sections 6.4.1 and 6.4.2. For our tests we set a fairly conservative constraint: $\kappa = 0.3$ in Equation (6.93). This means that the position error estimate would have to exceed $3\sigma^*$ for track loss to occur due to mispointing. Taking the search radar's beam width as $\theta_b = 1°$ and referring back to our problem formulation from Section 6.2.4 this gives $\rho_1 = 2.62 \times 10^{-3}$ in Equation(6.17).

Our final hand-off requirement is actually quite similar to the intermediate pointing requirement described above. The major difference is that we don't know ahead of time where the targeting radar receiving the BRV's track will be located. This leaves us unable to compute a projection of the error ellipsoid onto the plain perpendicular to the vector from the new radar. Instead, we base our requirement on the ellipsoid's semimajor axis,

$$\sigma_{\mathrm{max}} = \mathrm{sqrt}\left(\max\left(\mathrm{eig}\left(\hat{\mathrm{P}}'\right)\right)\right) \tag{6.100}$$

| Symbol | Description | Value |
|---|---|---|
| Search $\theta_b$ | Beam width for search radar | 1° |
| Target $\theta_b$ | Beam width for targeting radar | 0.5° |
| $\kappa$ | Error ratio for track loss | 0.3 |
| $h_c$ | Height at which hand-off occurs | $20,000$ ft |
| $R_c$ | Maximum range to targeting radar at hand-off | $30,000$ ft |
| $\rho_1$ | Constraint for $g_1$ | $2.62 \times 10^{-3}$ |
| $\rho_2$ | Constraint for $g_2$ | 39.27 ft |
| $C_1$ | Penalty for potential track loss | 100 |
| $C_2$ | Penalty for potential failed hand-off | $1,000$ |

Table 6.2: Parameters for cost model.

and an arbitrary expected range $R_c$ to the targeting radar based on our cutoff altitude $h_c$. Thus $\rho_2$ in Equation(6.17) becomes

$$\rho_2 = \frac{\kappa \theta_b R_c}{2} \tag{6.101}$$

which, with our assumed values of $\theta_b = 0.5°$ (for the targeting radar), $R_c = 30,000$ ft ($h_c = 20,000$ ft), and $\kappa = 0.3$ gives an allowable error of about 40 feet.

The actual penalties $C_1$ and $C_2$ were chosen somewhat arbitrarily within a fairly simple set of bounds. Given a sampling frequency of 20 Hz, a time to impact of roughly 30 seconds, and a cost of 1 for each measurement taken, a policy which measured at every opportunity would incur a cost on the order of 600. The penalty for failing to get the desired accuracy at hand-off was taken to be substantially greater than this at $C_2 = 1000$. An intermediate constraint penalty $C_1$ smaller than $C_2$ yet substantially greater than one was desired, so $C_1 = 100$ was used. The parameters determining the cost model are summarized in Table 6.2.

## 6.6 Feature Selection

In our previous examples from Chapters 4 and 5, feature extraction was not a big issue. In the nonlinear examples from Chapter 5, there were only two elements in the filter state vector ($\hat{x}$ and $\hat{P}$, three if the external input $u$ was included). This made simplification of the state space through the use of features unnecessary. In Chapter 4 there was one problem with a 6 × 6 covariance matrix containing 21 independent elements. Though one might be justifiably skeptical of the ability of a function approximator to capture the relevant dependencies without some sort of reduction in the size of the state vector, learning performed quite well without any simplification. This was likely due to the deterministic nature of the state propagation in the Kalman filter. The portion of this 21-dimensional space explored by the learning algorithm was actually quite small. A single trajectory might return to the same vicinity in the state space a number of times.

Our current application, however, has a raw state vector of 35 elements and is not deterministic. The enormous size of this state vector combined with the large potential variation in individual trajectories make this a much harder problem. Since the estimated position and velocity vectors are included in the filter state for the IEKF, no single state will be encountered more than once in a single trajectory. Indeed, a number of trajectories might go by before the same region in this 35-dimensional state space is encountered a second time. This sparse sampling makes learning directly with all 35 states impractical. Some sort of transformation must be applied to the state vector in order to extract the common elements which allow learning to take place and intelligent decisions to be made. In making this transformation we will be discarding some information and biasing our decision process, but this is unavoidable.

The particular choice of features we will use is heavily influenced by the cost function we have selected. From the analysis of our measurement system in Section 6.4 and our formulation of the cost function in Section 6.5 there are five quantities which seem particularly relevant to measurement decisions. Each will be described below.

First we have the two quantities associated with our intermediate pointing requirement,

$\sigma^*/R$ and $\hat{z}'/R$. The former is strictly a performance measure, and enters into the decision process only through the cost function element $g_1$. The latter is a measure of the beam distortion that will take place and enters into the decision process both through $g_1$ and through the equations for calculating the sensor noise covariance. We can combine these two features into a single quantity $\sigma^*\hat{z}'/R^2$, reducing the dimension of the state space by one, but if we do so we lose directional information that might influence the optimal policy. One might want to change one's measurement strategy based on the angle of the BRV's approach to take advantage of changes in measurement accuracy as the radar beam narrows and widens. In testing of the learning algorithm these features were used both together and separately.

Next we have the two quantities related to our final pointing requirement, $\hat{h}$ and $\sigma_{\max}$. These are different from the first two in that they only exert a significant influence on the measurement policy in the final stages of the trajectory.[1] In the early stages of the tracking problem these quantities are largely irrelevant and serve only to dilute the important information with effectively random noise. For this reason a cutoff height $h_\theta = 1.5h_f$ was defined beyond which neither $\hat{h}$ nor $\sigma_{\max}$ is considered important. Our features thus become:

$$\bar{h} = \begin{cases} 600,000 & \left(\hat{h} > h_\theta\right) \\ \hat{h} & \left(\hat{h} < h_\theta\right) \end{cases} \tag{6.102}$$

and

$$\bar{\sigma}_{\max} = \begin{cases} 50,000 & \left(\hat{h} > h_\theta\right) \\ \sigma_{\max} & \left(\hat{h} < h_\theta\right) \end{cases} \tag{6.103}$$

where the values for $\bar{h}$ and $\bar{\sigma}_{\max}$ $\hat{h} > h_\theta$ were chosen arbitrarily so as to be out of the range of values that might be encountered for $\hat{h} < h_\theta$.

Finally, we have as a potential feature the estimated range from the radar to the BRV, $\hat{R}$. This affects the sensor noise covariance and hence may influence the optimal policy. When we use it as a feature, we normalize by the maximum detection range and raise it to

[1] Since the BRV enters with a relatively constant flight path angle the height could influence the policy in the earlier stages of the trajectory through its relationship to the range. However, this relationship may not be consistent from trajectory to trajectory, and so could be misleading.

| Feature | Description |
|---|---|
| $\sigma^*/\hat{R}$ | Beam-width error |
| $\hat{z}'/\hat{R}$ | Beam angle |
| $\sigma^*\hat{z}'/\hat{R}^2$ | Combined beam-width error and beam angle |
| $\bar{h}$ | Height with state aggregation |
| $\bar{\sigma}_{\max}$ | Maximum error dimension with state aggregation |
| $(\hat{R}/R_{\max})^4$ | Range-related attenuation |

Table 6.3: List of potential features.

the fourth power to describe the effect on the sensor noise, to get $(\hat{R}/R_{\max})^4$. This was found to be more significant to the continuous action problem than the discrete action problem.

The features we consider for use in this application are summarized in Table 6.3. The specific features used for the discrete action and continuous action cases are described in Sections 6.8.2 and 6.9.2 respectively.

## 6.7 Scenario Details

In this section we present the details of our BRV tracking scenario. The search radar is located at sea level ($h_s = 0$) at a geodetic latitude $\mu = 24°$. The radar face is directed at angles $\lambda = 45°$ and $\phi = 51.5°$ as per Figure 6-3. Sixteen different trajectories are considered; one each coming from one of four different directions and heading toward one of four different targets in the radar's field of view as shown in Figure 6-10. The trajectories each start at an altitude of $150,000$ feet at a horizontal distance of 300,000 feet from their targets. MATLAB code for generating these initial conditions is given in Figure 6-11. The initial flight path angle is about 25° and the total initial speed is roughly Mach 15. These quantities are

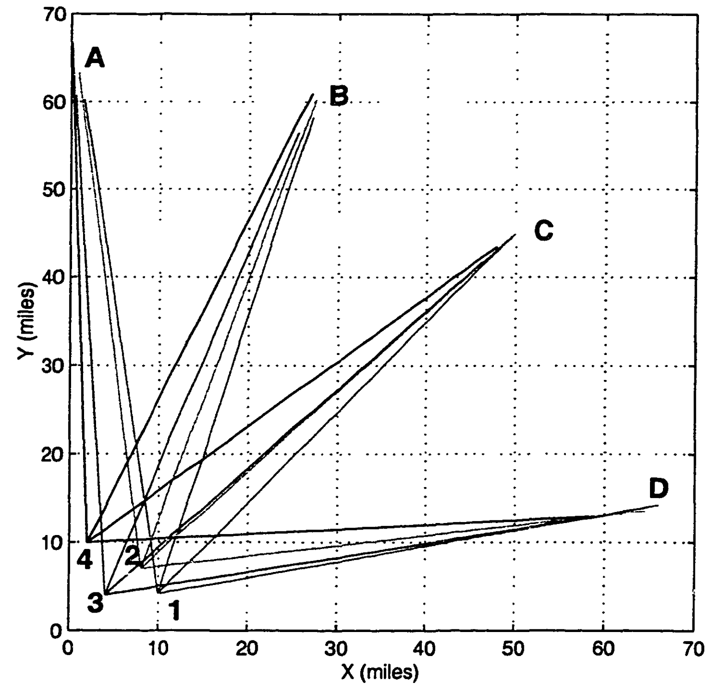


Figure 6-10: BRV trajectories used for learning. Source locations are marked A-D; target locations are marked 1-4.

summarized in Table 6.4.

For the filter, the timestep is $\Delta t = 0.05$ sec. This is also the interval at which actions may be taken. Four iterations are used in the IEKF. The process noise intensity matrix $W$ is assumed to be diagonal with magnitudes of 5000 for each velocity component and $10^{-18}$ for the drag parameter $\alpha$. No noise is assumed in the position components of the state vector. The initial error covariance $\hat{P}_0^-$ is taken to be diagonal with values of $10^6$ ft$^2$ in the position components, $10^4$ ft$^2$/sec$^2$ in the velocity components, and $10^{-14}$ in the $\alpha$ component. For each trajectory, initial position and velocity estimates are calculated randomly using a gaussian distribution with covariance $\hat{P}_0^-$ centered on the actual position. These quantities are summarized in Table 6.5.

```
x0 = zeros(16,7);
xhat0 = zeros(16,7);

xt = 5280*[10 8 4 2];        % Target X coordinate (Miles to feet)
yt = 5280*[4 7 4 10];        % Target Y coordinate

xs = 5280*[0 30 50 60];      % Source X coordinate
ys = 5280*[70 67 45 13];     % Source Y coordinate

Xt = [xt xt xt xt];          % Each target is used for four trajectories
Yt = [yt yt yt yt];

% Each source is used for four trajectories
Xs = [xs(1)*ones(1,4) xs(2)*ones(1,4) xs(3)*ones(1,4) xs(4)*ones(1,4)];
Ys = [ys(1)*ones(1,4) ys(2)*ones(1,4) ys(3)*ones(1,4) ys(4)*ones(1,4)];

% With starting flight path angle of 25 degrees, BRV impacts after
% traveling roughly 300,000 feet.  Thus to get our initial conditions
% we project backwards 300,000 feet from the target along the line
% between the target and source locations.

theta = atan2(Ys-Yt,Xs-Xt);
r = 300000;
X = Xt+r*cos(theta);
Y = Yt+r*sin(theta);

v = -13000;  % Initial speed in the XY plane; initial vertical speed is -6100 ft/sec

for(i=1:16)
   x0(i,:) = [T*[X(i); Y(i); 150000];                         % Initial position
              T*[v*cos(theta(i)); v*sin(theta(i)); -6100];    % Initial velocity
              4.65e-8]';                                      % Initial alpha
   xhat0(i,:) = x0(i,:)+[1e3*randn(1,3) 1e2*randn(1,3) 0];    % Add noise to match P0
end
```

Figure 6-11: MATLAB code for generating initial positions. The variable T contains the transformation matrix $T$ described in Equation (6.23).

| Symbol | Description | Value |
|---|---|---|
| $h_s$ | Altitude of radar station | 0 |
| $\mu$ | Geodetic latitude of radar station | 24° |
| $\lambda$ | Azimuthal facing of search radar | 45° |
| $\phi$ | Elevation facing of search radar | 51.5° |
| $z_0$ | Height of BRV at $t = 0$ | 150,000 ft |
|  | Horizontal distance from target to initial position | 300,000 ft |
|  | Initial flight path angle (approximate) | 25° |
|  | Initial Mach Number (approximate) | 15 |

Table 6.4: Scenario initial conditions.

| Symbol | Description | Value |
|---|---|---|
| $\Delta t$ | Time step | 0.05 sec |
| $n$ | IEKF iterations | 4 |
| $W$ | Process noise intensity | $\text{diag}([0 \quad 0 \quad 0 \quad 5000 \quad 5000 \quad 5000 \quad 10^{-18}])$ |
| $\hat{P}_0^-$ | Initial error covariance | $\text{diag}([10^6 \quad 10^6 \quad 10^6 \quad 10^4 \quad 10^4 \quad 10^4 \quad 10^{-14}])$ |

Table 6.5: Filter initial conditions.

## 6.8 Learning with Discrete Actions

Though many real radar systems have the capability of generating pulses of varying strength and multiple simultaneous beams [54], few take advantage of it [14]. Simple heuristic algorithms exist for determining the measurement frequency, but these rely on the assumptions that only a single pulse is generated and that the maximum power level is used. In this section we will treat this same problem both with a heuristic and with reinforcement learning. In the next section these assumptions will be relaxed and we will see what can be done if we allow multiple simultaneous pulses of varying strength.

For our experiments we allow only two possible actions: either a measurement at full power ($a_k = 1$) or no measurement at all ($a_k = 2$). If a measurement is taken a unit cost is applied; otherwise no cost is incurred. Thus our action related cost is

$$g_a\left(a_k\right) = \begin{cases} 1 & (a_k = 1) \\ 0 & (a_k = 2) \end{cases} . \tag{6.104}$$

### 6.8.1 Heuristic Policy

To determine the efficacy of our learning algorithms we must have a basis for comparison. For this purpose we have developed a heuristic policy which, though simple, performs quite well. Its excellent performance is due largely to the fact that it is based closely on the cost formulation. The policy used was:

$$\begin{array}{lll} a = 1 & \left( \sigma^{*}\hat{z}' \big/ \hat{R}^2 > 0.8\rho_1 \quad or \right. & \left. \left( h < 1.2h^{*} \quad and \quad \sigma_{\max} > 0.8\rho_2 \right) \right) \\ a = 2 & & elsewhere \end{array} \tag{6.105}$$

where the various cutoff levels were determined experimentally. These levels are somewhat conservative, but when more agressive levels were used it was found that the intermediate and final constraints were occasionally violated. The hope is that higher performance policies can be found using reinforcement learning which use fewer actions but still manage to consistently meet the constraints.

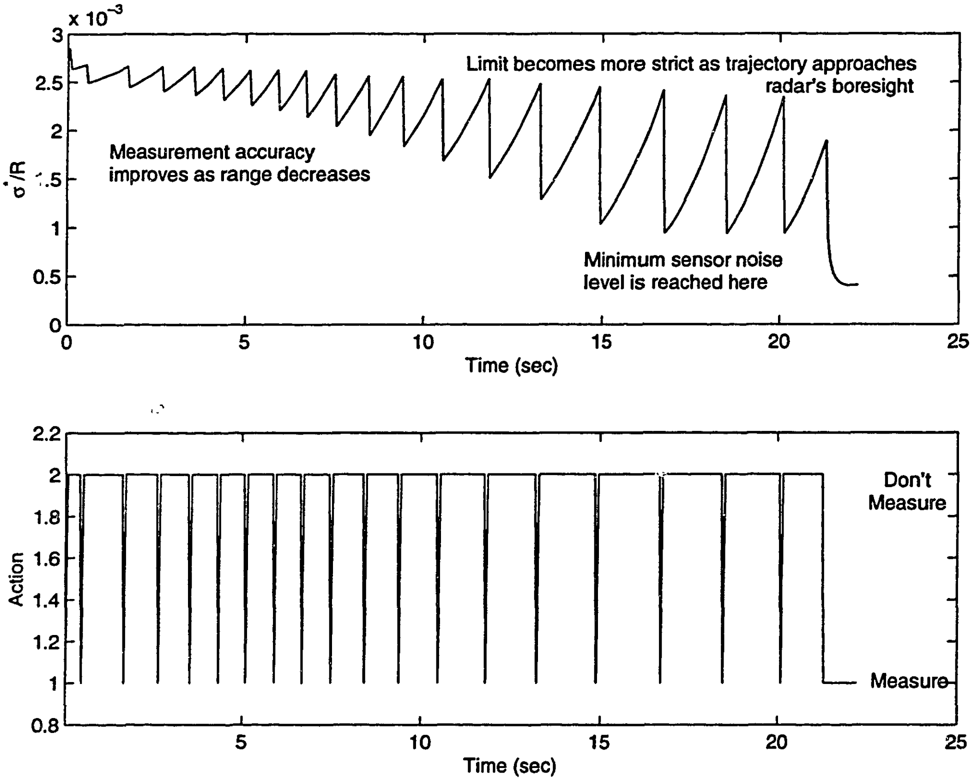


Figure 6-12: Time histories for trajectory A1 (from Figure 6-10) using heuristic policy.

The behavior of the filter under the above policy is shown graphically in Figure 6-12. The top plot shows the angular uncertainty used in the intermediate pointing constraint; the bottom shows the actions chosen. In these plots the influence of several of the radar system's characteristics can be observed. The increase in measurement accuracy as range decreases can be seen as each successive measurement produces a greater decrease in error. We can also see the point at which the minimum sensor noise level is reached, and the gradual decrease in allowable error as the trajectory brings the BRV closer to the boresight and the beam becomes narrower. The point at which the heuristic reaches $h_c$ and switches to the more strict error limit is clearly visible: before it we have isolated single measurements, and after it the heuristic demands measurements at each step until the hand-off occurs. Finally, we can see that early in the trajectory $\sigma^*/R$ increases less rapidly between measurements than it does in later stages. This is because $\sigma^*/R$ is an angular quantity and the process noise $W$ is expressed in terms of distance.

Each of the sixteen trajectories shown in Figure 6-10 was simulated 100 times using this heuristic. The resulting average costs are shown in Table 6.6. From the table, we can see that the costs for Target #2 are lower than for the others. This is most likely due to the fact that Target #2 is closer to the search radar. The other trend that is visible is an increase in cost as the sources approach the radar boresight. This is a somewhat counterintuitive result. A number of possible explanations exist, but from the limited information available it is difficult to tell which, if any, is correct. It is precisely this type of result which motivates the use of a learning algorithm. Heuristics are basically limited by the designer's understanding of the system, while learning systems may be capable of picking up on subtle effects and handling them appropriately.

Before proceeding with the reinforcement learning it behooves us to verify, as we did in the previous chapter, that our covariance estimate is providing a realistic measure of the filter's performance. Figure 6-13 displays the results of running 1600 trajectories and comparing the sample means of the squared errors to the estimated covariances. The resulting curve holds fairly close to the line $y = x$ and so indicates a reliable covariance estimate.

| | Source | | | |
|---|---|---|---|---|
| Target | A | B | C | D |
| 1 | 39.52 | 43.90 | 47.84 | 43.50 |
| 2 | 38.84 | 45.44 | 49.20 | 43.68 |
| 3 | 35.04 | 40.79 | 42.64 | 38.91 |
| 4 | 38.33 | 45.02 | 45.65 | 41.00 |

Table 6.6: Average costs over 100 trials for trajectories using the best heuristic policy.

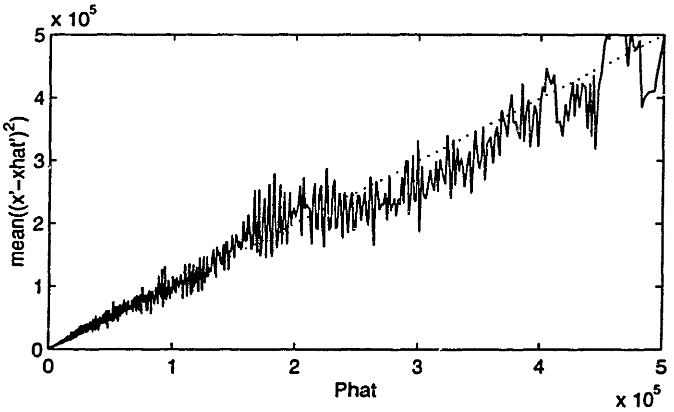


Figure 6-13: Verification that covariance estimate is reasonably close to actual covariance.

### 6.8.2 Learning Setup

Learning experiments were conducted using both SARSA($\lambda$) and Q-Learning with a variety of algorithm settings. SARSA($\lambda$) was found to be more effective than Q-Learning for this problem. Initial experiments used the full 35 independent elements of the filter state as features, but were largely unsuccessful. Operating on the assumption that this failure was due to the large input space, a smaller feature set was used as described in Section 6.6. Experiments were conducted both with and without the state aggregation of the height and $\sigma_{max}$ features at high altitudes ($\bar{h}$ and $\sigma^{\bar{\mathrm{m}}\mathrm{ax}}$ instead of $\hat{h}$ and $\sigma_{\mathrm{max}}$), and with and without combining the $\sigma^*/\hat{R}$ and beam angle features.

The parameters which produced the best results are shown in Table 6.7. State aggregation was used at high altitudes. The features used were, in order, $\bar{h}$, $\sigma_{\bar{\mathrm{m}}\mathrm{ax}}$, $\sigma^*/\hat{R}$, and $\hat{z}'/\hat{R}$. Each of the sixteen test trajectories was run $12,000$ times for a total of $192,000$ iterations.

Testing for the best policy was problematic for two reasons. First, a policy which performed well with one of the sixteen trajectories might perform poorly with others. In addition, a very agressive policy might perform well most of the time but occasionally fail to meet the constraints and incur a large cost penalty. Ideally, we would like to test each policy on all of the sixteen trajectories and average results over a large number of trials, but this is somewhat impractical. If such a methodology were used, time spent testing would quickly exceed the time spent learning. Ultimately, in the interests of speed, testing during the learning process was conducted using only a single trial of the first trajectory (A1). These tests were conducted a total of one thousand times at even intervals through the learning process. The resulting "best" policy is described in detail in the following section.

### 6.8.3 Learning Results

Once a candidate policy had been selected, its overall performance was tested by running each of the sixteen trajectories 100 times and averaging the resulting costs. These averages are shown in Table 6.8, and should be compared to the corresponding results from the heuristic policy shown in Table 6.6. The learned policy has a lower average cost in fifteen

| | |
|---|---|
| Scaling | |
| State ($\theta$) | $[0.004 \quad 0.004 \quad 10000 \quad 50]$ |
| Action ($a$) | 40 |
| Cost ($Q$) | 1 |
| CMAC-related parameters | |
| Memory size | 50000 |
| Number of layers | 16 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_2 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 6000$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 60 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 192000$ |
| Number of steps/trajectory | Variable (until $\hat{h} < h^*$) |
| Exploration rate | $\epsilon = [0.1 \quad 0.02 \quad 0.01 \quad 0.005 \quad 0.001]$ |
| Rate of reduction for exploration rate | $N_\epsilon = 12000$ |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.7$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |

Table 6.7: Algorithm parameter settings for discrete BRV tests.

|  | Source |  |  |  |
|---|---|---|---|---|
| Target | A | B | C | D |
| 1 | 26.83 | 30.48 | 33.12 | 35.62 |
| 2 | 30.29 | 32.63 | 33.01 | 34.00 |
| 3 | 31.93 | 26.01 | 26.96 | 25.23 |
| 4 | 38.80 | 34.31 | 31.63 | 27.00 |

Table 6.8: Average costs over 100 trials for trajectories using the best learned policy.

of the sixteen cases, with a maximum improvement of over 35%. In the one case where the heuristic performs better (trajectory A4) the difference is very small: just over 1%.

Interestingly enough, the costs from Table 6.8 make more intuitive sense than those from Table 6.6. In both tables we can see lower costs for trajectories heading toward target three, but with the learning results we can see several other trends which make intuitive sense. For target #1, we see the lowest cost from source A and the highest from source D. This makes sense as the first of these trajectories makes its final approach at lower ranges while the last suffers the most from beam spreading. These relative differences are shown graphically in Figure 6-14. Similarly, when we look at the four trajectories coming from source A we see that the trajectories heading toward target #1 and #2 suffer less from beam spreading than those heading toward #3 and #4, and again the ordering makes sense. The learning algorithm appears to "understand" the trajectory differences in a way that could not be easily captured in a heuristic.

To gain more insight into the differences between the learned policy and the heuristic we must look at some actual trial runs. Figure 6-15 shows how both policies perform on trajectory A1, and Figure 6-16 shows how they perform on trajectory A4. Trajectory A4 is the most difficult of the 16 trajectories. Not only does it come in at an angle so that it suffers from beam spreading, but it also suffers from the fact that it is almost directly approaching

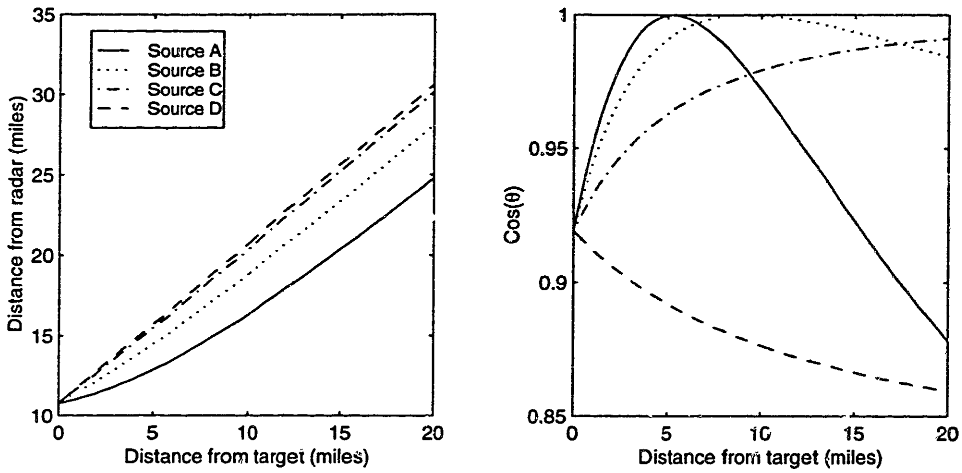


Figure 6-14: Statistics showing the relative difficulties of estimation for trajectories approaching target #1 from each source. At left, lower numbers correspond to better measurements, and at right higher numbers correspond to better measurements.

the search radar. This means that as it approaches the hand-off altitude it will be further from the search radar than just about any of the other trajectories. Its situation is similar to that of trajectory D1 (as seen in Figure 6-14) only more so.

Thus it is trajectory A4 which sets the bar for the heuristic policy. If we make our heuristic's cutoff levels more agressive we risk failed hand-offs or lost tracks on this trajectory. On the other hand, this conservatism is not necessary for more forgiving trajectories such as A1 or D4. The learning algorithm is able to take advantage of these differences. It is conservative where necessary (note the similarity of the two policies in Figure 6-15) and agressive where it can safely be so. It may be possible to come up with a heuristic that can compete with the learned policy, but it would not be an easy task.

## 6.9 Learning with Continuous Actions

The discrete action space formulation with its measure or no measure choices is actually much more representative of the manner in which real-world radar tracking systems operate, but the ability to generate multiple beams of varying strength does exist in virtually all

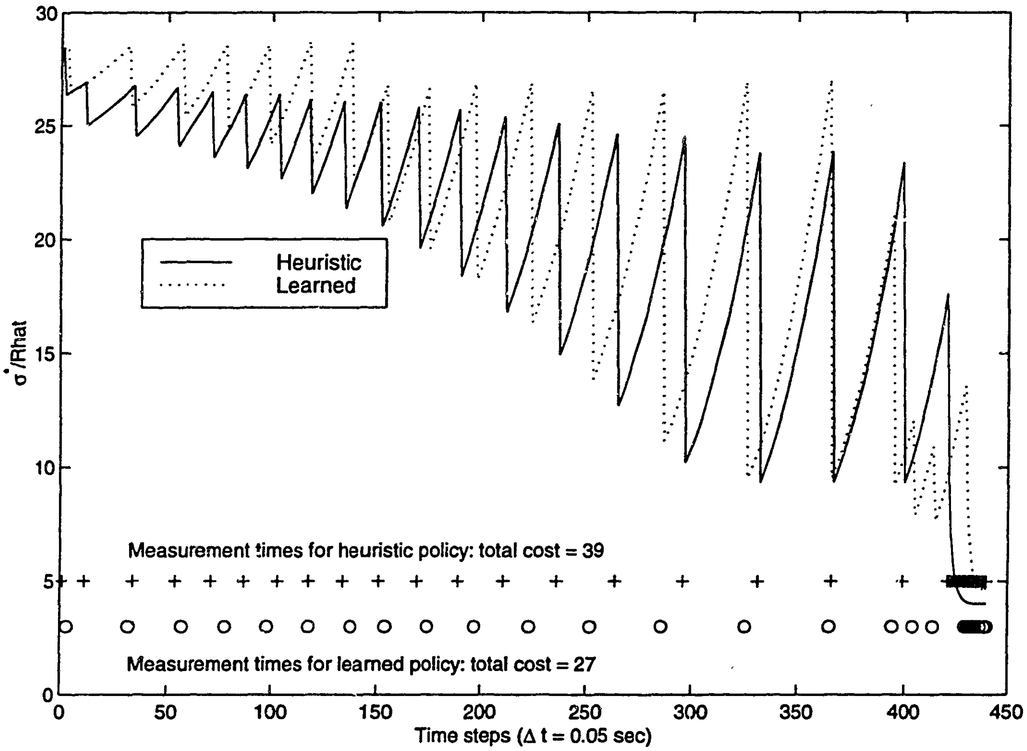


Figure 6-15: Time history for trajectory A1 comparing learned and heuristic policies.

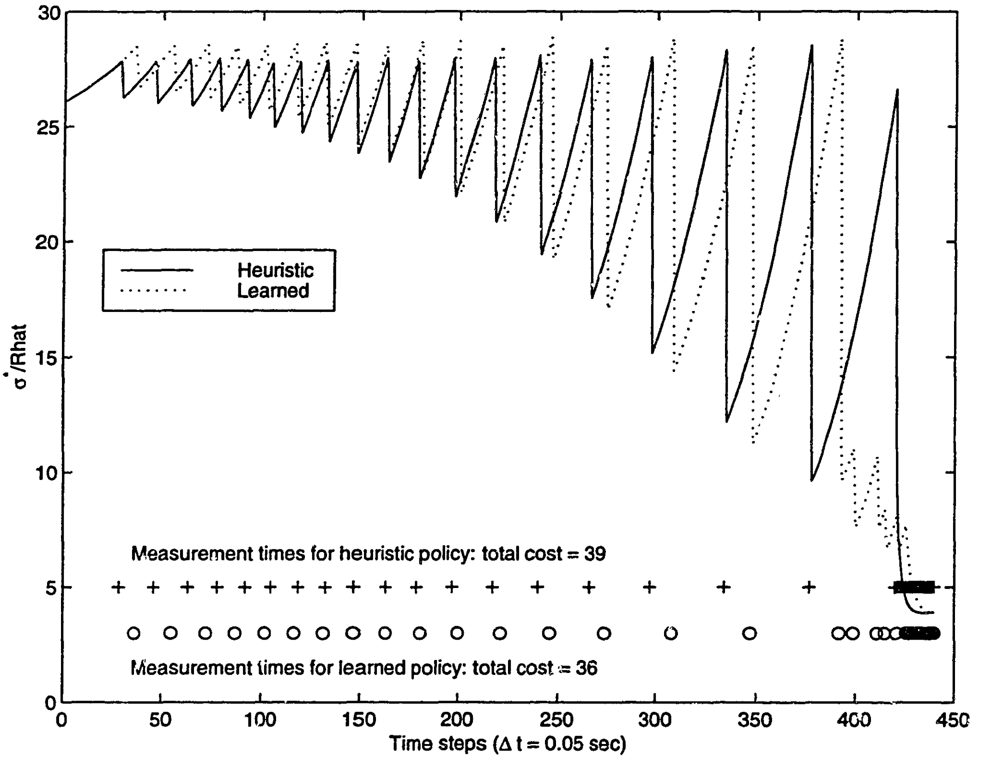


Figure 6-16: Time history for trajectory A4 comparing learned and heuristic policies.

phased-array radar systems. It is hard to say for sure why this capability is not taken advantage of, but four possible explanations occur:

- The calculations necessary to control the array to generate multiple beams in arbitrary directions are complex.
- Pulse rates are high enough that continuous power variations can be approximated by varying dwell times.
- No clear algorithms are available to dictate how this capability could best be used.
- At low signal-to-noise ratios dividing power into multiple beams is cost ineffective due to decreased probability of detection.

The first of these reasons will become less and less relevant as computing power increases. The logic of the remaining reasons can be tested to some extent by applying learning and looking at the results. It seems likely that significant savings can be derived while tracking targets at low ranges. As the power coming back to the radar varies inversely with the fourth power of the range, it stands to reason that less power will be needed for close-in targets.

### 6.9.1 Heuristic Policy

The heuristic policy that was developed for the continuous action case is the product of understanding of the radar measurement model, experience with the discrete action problem, and insight gained from looking at learned policies. Of these, the value of the last two is certainly open to debate. Though the policy is fairly complex in formulation, the ideas behind it are straightforward. One could arguably come up with the same policy without having worked with the discrete case or having seen any learning results. Nonetheless, there are enough things going on with the overall system that it would be hard to have confidence that this provided the best solution without the learning results to back it up. Nor was this the first policy that was tried; other policies were formed and then discarded when they were beaten by either the discrete action policy or a learned policy.

Under the best heuristic policy that was found, actions are chosen as follows. First, the intermediate and final action values ($a_i$ and $a_f$ are calculated. The former is designed to ensure that the intermediate angle error limit is not exceeded, and is given by

$$a_i = \begin{cases} \eta_d \left(\hat{R}\middle/R_{\max}\right)^4 & \sigma^* \hat{z}'\middle/\hat{R}^2 > 0.8\rho_1 \\ 0 & \sigma^* \hat{z}'\middle/\hat{R}^2 \leq 0.8\rho_1 \end{cases}. \tag{6.106}$$

The power allocated is lowered as the range decreases to reduce costs while keeping a roughly constant probability of detection, determined by $\eta_d$. Testing showed $\eta_d = 1.33$ to work best. The final action value $a_f$ is designed to ensure the final hand-off requirement is met and is given by

$$a_f = \begin{cases} \frac{\sigma_{\max}}{0.8\rho_2} & \left(\hat{h} < 1.2h^*\right) \\ 0 & \left(\hat{h} > 1.2h^*\right) \end{cases}. \tag{6.107}$$

Thus it is only relevant once the BRV has approached the hand-off height. The larger of these values, truncated to go no higher than unity, becomes the base action, $\bar{a}$:

$$\bar{a} = \min\left(1, \max\left(a_i, a_f\right)\right). \tag{6.108}$$

A further modification is then made to the action to ensure that power is not used above the level necessary to get the maximum accuracy. At lower ranges, the base action may result in a sensor variance limited by the radar system's minimum sensor noise, $V_{\min}$. If this occurs, power is being wasted, as a lower power level could produce a measurement of equal accuracy. To avoid this situation, the two quantities

$$\eta_R = \frac{V_{\max}[1,1]\hat{R}^4}{V_{\min}[1,1]R_{\max}^4\bar{a}} \quad \eta_\theta = \frac{V_{\max}[2,2]\hat{R}^5}{V_{\min}[2,2]R_{\max}^4\hat{z}'\bar{a}} \tag{6.109}$$

are calculated[2]. The additional factor of $\hat{R}/\hat{z}'$ in $\eta_\theta$ compensates for off-axis beam distortions. If both $\eta_R$ and $\eta_{theta}$ are below one, then power is being wasted, and the action must be

[2] $V_{\max}[1,1]$ refers to the entry in the first row and first column of the $V_{\max}$ matrix.

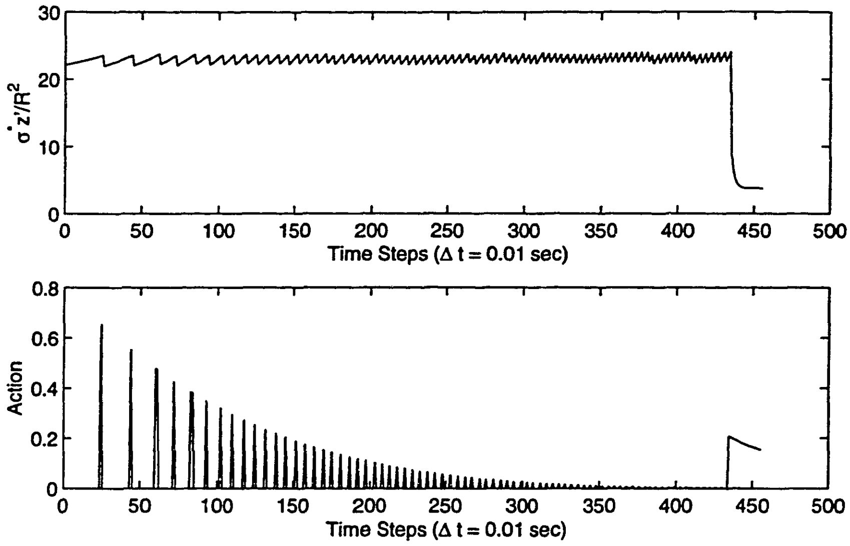


Figure 6-17: Time histories for trajectory A1 using the heuristic policy. The top plot shows the quantity tested for the intermediate pointing requirement. The corresponding actions are shown in the bottom plot.

adjusted. This gives a final action of

$$a = \begin{cases} \bar{a} & (\max(\eta_R, \eta_\theta) \geq 1) \\ \bar{a} \max(\eta_R, \eta_\theta) & (\max(\eta_R, \eta_\theta) < 1) \end{cases} . \tag{6.110}$$

When the policy described above is used, the total expense is less than half of what would be required were a discrete action policy (either learned or heuristic) used. A sample trajectory is shown in Figure 6-17. Notice the relatively flat curve in the top plot, in contrast to the increasingly deep dips of Figure 6-12 marking wasted energy. Also notice the repeated action near the 80th time step, probably due to a missed detection.

### 6.9.2 Learning Setup

Learning runs were conducted using a variety of configurations with mixed results. For the learning algorithm attention was focused on SARSA($\lambda$) due to the poor results with Q-Learning in the discrete action case. The base feature set was the same as in the discrete action case with the addition of a range-related feature, $(\hat{R}/R_{\max})^4$. Tests were conducted both with and without the range feature and with and without condensing $\sigma^*/\hat{R}$ and $\hat{z}'/\hat{R}$ into a single feature.

The parameters which produced the best results are shown in Table 6.9. The features used were, in order, $\bar{h}$, $\sigma_{\bar{\max}}$, $\sigma^*\hat{z}'/\hat{R}^2$ and $(\hat{R}/R_{\max})^4$. Learning data was taken exclusively from tests of trajectory A1 from Figure 6-10. Multiple runs of the algorithm were conducted using the best policies and the corresponding $\tilde{Q}$ weights from previous runs as starting points and reducing exploration rates.

### 6.9.3 Learning Results

The continuous action space formulation for this application represents the most difficult problem presented in this thesis, and results are mixed. Figures 6-18 and 6-19 show the time histories and cumulative costs for a test of trajectory A1 using both learned and heuristic policies. The two policies exhibit significant differences in action selection, but have almost identical performance. Which policy generates the lower cost varies from run to run. Both are substantially better than the policies found from the discrete action case.

Unfortunately, we were not able to obtain good results when learning was done on all 16 test trajectories simultaneously. In tests where the range was included as a feature, learned policies did not even perform as well as those from the discrete action case. A variety of different parameters were tested, and runs long enough to require multiple days on a reasonably fast computer were made. This is where we see that reinforcement learning, while it is sometimes able to achieve remarkable things, is still dependent on the user's understanding of the problem. At some point the interaction between the various parameters which must be chosen reaches a level of complexity which cannot be fathomed by even a reasonably knowledgable user, and appropriate parameter choices become difficult. This appears to be that point for this problem. It may be that RL can do well for some set of parameter choices, but no such set was found.

Lest the wrong impression be given here, it should be pointed out that even the successful applications of RL presented in this chapter and the ones before required significant

| | |
|---|---|
| Scaling | |
| State ($\theta$) | $[0.004 \quad 0.004 \quad 10000 \quad 100]$ |
| Action ($a$) | 100 |
| Cost ($Q$) | 1 |
| CMAC-related parameters (same for $\tilde{Q}$ and $\tilde{\pi}$) | |
| Memory size | 50000 |
| Number of layers | 16 |
| Main learning rate | $\beta_1 = 1$ (in powers of 1/2) |
| Secondary learning rate | $\beta_2 = 4$ (in powers of 1/2) |
| Rate of reduction for learning rates | $N_\beta = 8000$ |
| Initial Conditions | |
| Number of trajectories run using initial guess | $N_0 = 0$ |
| Weights for $\tilde{Q}$ approximator | All weights initialized to 30 |
| Weights for $\tilde{\pi}$ approximator | All weights initialized to 100 |
| Algorithm Parameters | |
| Number of $\epsilon$-greedy trajectories (iterations) | $N_{eg} = 96000$ |
| Number of steps/trajectory | Variable (until $\hat{h} < h^*$) |
| Exploration rate | $\epsilon = [0.1 \quad 0.02 \quad 0.01 \quad 0.005 \quad 0.001]$ |
| Rate of reduction for exploration rate | $N_\epsilon = 10000$ |
| Decay rate for SARSA($\lambda$) eligibility traces | $\lambda = 0.7$ |
| Cutoff level for eligibility traces | $\kappa = 0.02$ |
| Fraction of samples updated after each trajectory | $\eta = 0.5$ |

Table 6.9: Algorithm parameter settings for continuous action BRV tests.

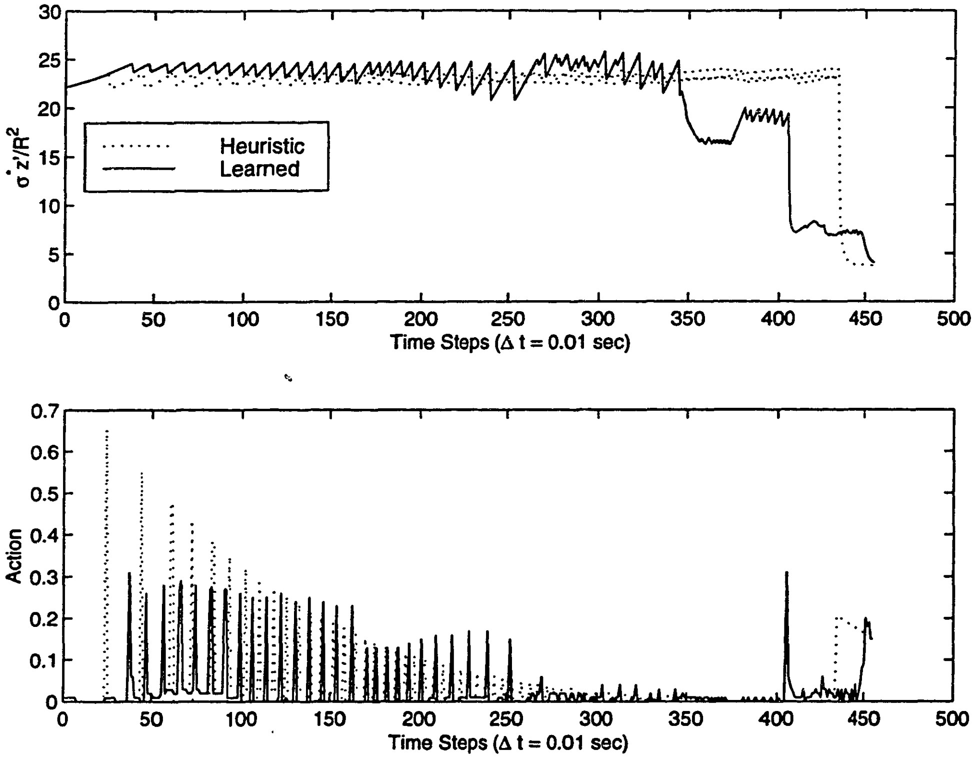


Figure 6-18: Time histories for trajectory A1 using both the learned and heuristic policies. The top plot shows the quantity tested for the intermediate pointing requirement. The corresponding actions are shown in the bottom plot.

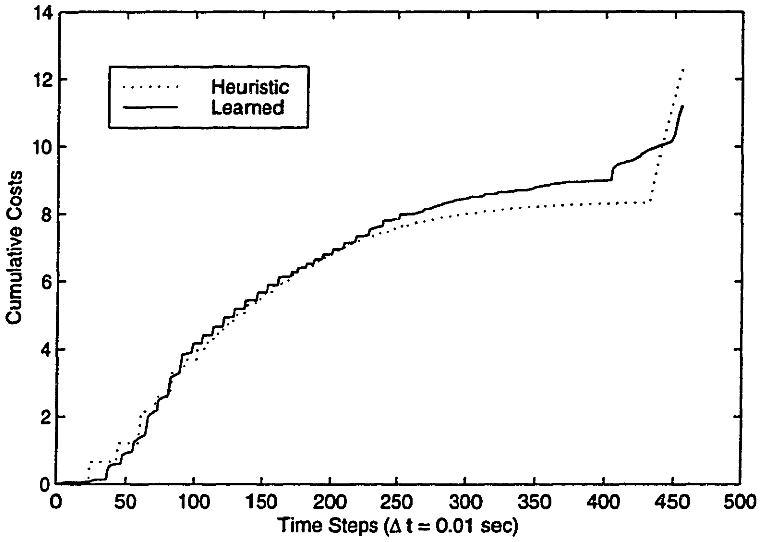


Figure 6-19: Cumulative costs for learned and heuristic policies for run of trajectory A1 shown in Figure 6-18.

experimentation in order to find parameter settings conducive to learning. Poor scaling, bad choices for initial weights, learning or exploration rates set too high or too low: all of these were able to throw off RL algorithms to a greater or lesser extent. This particular problem is complex enough that it is hard to say if any one of these, or some combination is causing learning to fail.

## 6.10 Summary

In this chapter we have provided an extremely challenging test for our reinforcement learning algorithms. We have developed a realistic model of a practical measurement scheduling application with both nonlinear plant and output equations. Though our algorithms were unable to handle the 35 separate inputs of the raw filter state, we were able to reduce the number of inputs using features and obtain excellent results for the discrete action case.

The learned policy was able to distinguish between trajectories coming from different directions and react appropriately, varying its level of aggressiveness according to the difficulty of the problem. In contrast, the best heuristic policy which could be developed was forced to adopt a level of conservativeness appropriate to the most difficult trajectories, sacrificing performance on the easier ones. In these cases the learned policy outperformed the heuristic by as much as 35%.

For the more difficult continuous action case results were mixed. When learning was performed based on trajectories coming from a single direction we were able to learn policies that performed on a par with the best heuristics. When inputs from multiple directions were used, however, RL was unable to find competitive policies.

| Symbol | Explanation | Symbol | Explanation |
|---|---|---|---|
| $(x, y, z)$ | LVLH coordinates of BRV | $(x', y', z')$ | Radar-based coordinates of BRV |
| $\alpha$ | Drag state; function of height, velocity | $\rho$ | Atmospheric Density (slugs/ft$^3$) |
| $C_D$ | Drag Coefficient; function of Mach No. | $h$ | height of BRV (feet) |
| $A$ | Drag area (ft$^2$) | W | BRV Weight (lb) |
| $g$ | Acceleration due to gravity at sea level (ft/sec$^2$) | $g_c$ | Local gravity measure (sec$^{-2}$) |
| $V$ | Speed of BRV (ft/sec) | $a$ | Local speed of sound (ft/sec) |
| $\eta_0 \ldots \eta_6$ | Constants in drag model | $G$ | Univeral gravitational constant (lb-ft$^2$/slug$^2$) |
| $M_e$ | Mass of the Earth (slugs) | $w$ | Earth's rotational speed (rad/sec) |
| $\mu$ | Geodetic latitude of radar station. See Figure 6-4 | $h_s$ | Altitude of radar station (feet). See Figure 6-4 |
| $a_0$, $a_e$ | See Figure 6-4 | $a_1$,$a_2$ | See Figure 6-4 |
| $r$, $R$ | See Figure 6-4 | $\phi^*$ | See Figure 6-4 |
| $e$ | Eccentricity of the Earth | $c_2$,$c_3$ | Fixed geometrical quantities relative to Earth (feet) |
| $\phi$ | Tilt angle of the radar face normal in elevation from the vertical | $\lambda$ | Tilt angle of the radar face normal in azimuth from the North |

Table 6.10: List of symbols for BRV equations of motion, radar measurements, and cost function

| Symbol | Explanation | Symbol | Explanation |
|---|---|---|---|
| $T$ | Transformation matrix between LVLH and Radar-based coordinates | $S$ | Centrifugal cross-product matrix |
| $U$ | Coriolis cross-product matrix | $\gamma_1$,$\gamma_2$,$\gamma_3$ | Distances related to gravity term |
| $\beta_1$,$\beta_2$,$\beta_3$ | Distances related to centrifugal term | $\sigma^*$ | See Figure 6-7 |
| $\theta$ | Radar measurement elevation angle | $\psi$ | Radar measurement azimuth angle |
| $\theta_b$ | Radar beamwidth angle | $C_1$,$C_2$,$C_3$ | Cost function penalties |
| $\rho_1$ | Intermediate angle limit for cost function | $h^*$ | Terminal height for cost function |

Table 6.11: List of symbols, continued

# Chapter 7

# Summary and Conclusions

## 7.1 Summary

In this thesis we have made contributions to two different fields. The first of these fields, that of measurement scheduling, has until now suffered from a gap between theory and reality. Previous work with real-world problems has relied mainly on heuristics, while work grounded in optimization theory has dealt only with very simple problems. We have advanced this field by treating complex problems with real-world characteristics using algorithms which are well grounded in optimization theory. Those algorithms come from our second field, reinforcement learning (RL). To date, RL has seen application mainly to heavily constrained environments. We have made some small additions to the theory of reinforcement learning (the treatment of G-functions in Chapter 3), but our main contribution here comes from the testing of RL's concepts in extremely challenging environments. Measurement scheduling problems are challenging because they exhibit many characteristics which are not normally seen in reinforcement learning applications: continuous, multi-dimensional state spaces; continuous action spaces; and non-Markovian transitions, among others.

In order to advance these two fields simultaneously, we needed to lay a certain amount of groundwork on each side. On the measurement scheduling side, we began by developing a formulation general enough to accommodate the wide variety of problems found in the literature. In addition, this formulation had to be able to handle the more complex problems we would use to move the field closer to the real world. On the reinforcement learning side, we needed to describe a modular algorithm able to accommodate a variety of different RL methods in a single framework. This would allow us to experiment freely with different techniques to find which ones worked best. As part of this framework, we needed to address some of the issues which arise when problems with complex action spaces are to be treated. This groundwork was laid in Chapter 2.

In the course of our research we came across problems where the Q-function seemed quite insensitive to the effects of individual actions. Investigation of ways of improving performance for these problems led to some extensions to the theory of RL. These were presented in Chapter 3. The results from this chapter included a new algorithm which we call G-Learning and a counterexample showing conditions under which an existing algorithm, Advantage Learning, may diverge.

Having established a general problem formulation with an accompanying algorithmic framework, we proceeded to apply them to problems of gradually increasing complexity. We began this process in Chapter 4 with a series of linear problems. Working with problems that have linear state and measurement equations was easier because we could use the Kalman Filter. With the Kalman filter we could track the filter's error variance using deterministic, Markovian transitions. We also had the potential for steady-state solutions, which made it easier to evaluate the effectiveness of our learning algorithms.

Four linear problems were treated, two of which were taken from the literature. The first was a scalar problem with continuous action space and infinite horizon cost formulation. The low dimension of this problem allowed us to perform a limited exploration of the solution space using analytical methods. We were also able to use this problem as a platform for exploring algorithmic implementation issues and visualize the results in an intuitive manner.

The second linear problem treated also used an infinite horizon cost formulation, but this time with a high dimensional state space and simple, discrete action space. Our reinforcement learning algorithms were able to juggle the 21 independent elements of the problem's $6 \times 6$ covariance matrix to generate policies which were non-intuitive and superior to the best heuristics we were able to come up with. The adroit handling of such a large input space was particularly significant, and was probably made possible by the deterministic transitions of the Kalman Filter, which substantially reduced volume of the state space that was encountered.

One of the linear problems taken from the literature gave us our only real failure. Given a finite horizon with a fixed total measurement budget and continuous action space, none of our reinforcement learning algorithms were able to find the known optimal solution. The reasons for this lack of success are explained in detail in Section 4.4, but have to do with the convexity of the problem[1] and a low repeatability of trajectories in the state space. This example served to show that reinforcement learning is not the best tool for every problem. Care should be taken to check a problem's suitability before expending a great deal of effort in looking for a learning-based solution.

Much better results were obtained with the other linear problem taken from the literature. This problem, addressed by Athans [5], involved a state space of moderate dimension and a discrete action space with a finite horizon cost. Using reinforcement learning we were able to find a policy with lower cost than the best policy Athans was able to find using an alternative method.

Having effectively treated a variety of linear problems, we moved on to the domain of nonlinear estimation. With this switch in domain came a host of new issues. Since optimal nonlinear filters are infinite-dimensional and hence unrealizable, we were forced to work with suboptimal filters. The use of these suboptimal filters meant that our underlying process was not only no longer deterministic, but also no longer Markov. In addition, where we had been able to safely work with a propagated covariance matrix as our cost feedback, we now were left with only an estimated covariance with no guarantees that it bore any relation to reality. We decided to treat a simple nonlinear problem before proceeding to the complex, high-dimensional, nonlinear application which was our ultimate objective.

We began our foray into the nonlinear world in Chapter 5 with a scalar cubic plant.

[1] Better methods than exist for dealing with convex problems, so it is hard for RL to compete.

This particular problem was chosen because it was simple enough to yield to some basic analysis which could give us an idea of what a good heuristic policy might look like. Both discrete and continuous action spaces were treated. Reinforcement learning was able to find policies that performed better than the initially chosen heuristics. In turn, analysis of the learned policies allowed the development of better heuristics which performed roughly as well as the learned policies but were much simpler. More will be said on the interplay of learning and heuristic policies later. With this sample problem we were also able to verify some of the results from Chapter 3 and explore issues relating to the use of known external inputs for system excitation.

Our ultimate objective, of course, was the application of reinforcement learning to a complex, real-world problem in measurement scheduling. This was finally accomplished in Chapter 6. For our application we chose to look at the tracking of ballistic reentry vehicles using phased-array radar. The scenario involved defense of a city-sized area from intermediate-range ballistic missiles. As the missiles began their descent they would be acquired by a search radar, which would then track their approach until the missiles entered the range of a targeting radar near the ultimate target. The search radar would be responsible for localizing the missile's position to the level of accuracy required for a smooth hand-off to occur. The targeting radar would then guide an intercepting weapon to destroy the incoming missile. The problem here was to use up as little of the search radar's time as possible in tracking known missiles so that more time could be devoted to searching for new ones. Two separate cases were considered: one where the only choices at each time step were to measure or not, and another which took advantage of the phased-array radar system's ability to generate pulses of varying strength.

A realistic treatment of this problem was quite involved. Both the plant and the measurement system were complex and highly nonlinear. The system's state vector had seven dimensions (three position, three velocity, and one drag-related), and derivation of the equations of motion required detailed models of the Earth's geometry, its atmosphere, and the drag characteristics of the reentry vehicle. On the measurement side we had to take into account a variety of radar specific effects. These included the possibility of missed detection, variation of measurement accuracy with range, and changes in the radar beam shape with its pointing angle.

Applying reinforcement learning to this problem presented additional challenges. The seven-dimensional state estimate combined with the 28 independent elements of the covariance estimate made for a total of 35 continuously varying inputs to the learning system at each time step. These were reduced to 3-5 features based on the problem's cost structure. Sixteen separate base trajectories were used to simulate the possible approach of missiles from different directions and provide learning experience; individual trajectories varied from the base due to the gaussian process noise in the system model.

For the discrete action case generation of a good heuristic policy was a simple matter dictated by the problem's cost structure. However, random effects such as the potential for missed detections and variations in the difficulty of the problem among the different test trajectories necessitated a certain amount of conservatism. The best learned policies, in contrast, seemed to take into account both random effects and directional differences to adjust their level of aggressiveness according to circumstances. The learned policies were able to save as much as 40% over the heuristics on the easier trajectories while maintaining the needed level of conservatism on the more difficult ones. This was an excellent result that demonstrated conclusively the flexibility and robustness of reinforcement learning techniques.

Development of a good heuristic for the continuous action case was a much more difficult proposition. Saving power by reducing measurement strengths resulted in an increased likelihood of missed detections. Policies which seemed to perform well most of the time would occasionally fail to meet the terminal accuracy requirement for hand-off to the targeting radar. It was difficult to determine how best to introduce the needed level of conservatism without sacrificing too much performance. Learning was rendered more difficult by the increased importance of the range as a feature relative to the discrete action case. Emulation of the learned policies with heuristics was difficult, as it was not always clear what rules the learned policies were following. As a result, the final learned policies and the final heuristics were quite different in nature, though very similar in performance. Both represented a significant cost savings over the discrete action case. Notably, with the continuous action case we were only able to get good results when learning based on individual trajectories. When all sixteen trajectories were used in the learning process, the best policies generated were not even as good as those from the discrete action case.

## 7.2 Conclusions

We began this research looking at the gap between theory and practice in the measurement scheduling literature, and hypothesizing that reinforcement learning was the tool which could bridge this gap. We hoped that we could use RL to help in real-world problems where heuristics were difficult to formulate and in need of some sort of performance verification. In the end we found that RL, though far from perfect, can be successfully applied to the problems we were interested in.

Success, however, can be a somewhat fuzzy concept, and we should clarify what we mean when we use the term. When evaluating the efficacy of an algorithm, the tendency is always to look for concrete measures. Did the algorithm find the optimal solution? Did it beat the solution generated by some other algorithm? If so, by how much? When the optimal solution is unknown, we have to settle for comparison to policies generated by other optimization methods or by heuristics. If a heuristic can be found that beats the policy found by the algorithm, the algorithm is deemed to have failed. We were able to show some cases where our learned algorithms were better than any heuristic we were able to find, but in others we ended up with heuristics which were simpler than the learned policies and performed about as well. By some standards, these latter cases would not be counted as a success for learning.

This way of looking at things sometimes falls short. One of the things that we observed in the examples from this thesis is a sort of synergy between the generation of heuristic policies and the generation of policies using reinforcement learning. Without a heuristic policy for comparison, we are unable to tell whether the learning algorithm is doing as well as it can. If we have a heuristic, however, we know something is wrong if the learning algorithm is unable to beat it. In turn, when the learning algorithm generates a policy that is better than our current heuristic we can often look at that policy, see what it is doing, and improve our heuristic. If the new heuristic performs significantly better than the learned policy, we know that there is room for improvement and we can repeat the process. On the other hand, if we are unable to generate a heuristic that performs on a par with the learned policy, we know that our understanding of the problem is lacking. When both the learned policy and the heuristic give similar performance it is like having independent verification that we are doing the right thing. We can have more confidence in our solutions when we use learning and heuristics together than with either of them alone.

Of course, the back-and-forth process described above points to one of the major shortcomings of practical reinforcement learning. There are simply too many parameters which must be chosen and too little knowledge of how this should be done. Function approximation architectures must be selected, features chosen, scaling factors decided. Both exploration and learning rates must be set, and then some scheme for their gradual reduction implemented. Virtually nothing is known about the effect of the $\lambda$ of TD($\lambda$) and SARSA($\lambda$). All we have is anecdotal reports that such and such a value worked well on such and such a problem. The end result of the learning algorithm is always a function of the interplay between these various factors. Often, the quality of that result depends on the degree to which the person choosing these factors understands both the problem and the algorithm. Thus an inductive bias enters the system which runs counter to the implied promise of a learning system: to derive that information automatically through interaction with the environment. Until a learning system is developed which does not require an expert to implement it, these algorithms will always seem somewhat quirky and unreliable.

Two particular shortcomings of reinforcement learning algorithms stood out in the investigations that we conducted. First was the lack of convergence for the more difficult problems, described by Bertsekas and Tsitsiklis [13] as "chattering". Though it was possible to extract individual excellent policies from the learning process, and a general trend toward improved performance was clearly visible, the large variation in the quality of individual policies was quite unsettling. Better theoretical understanding of this phenomenon is needed.

The second area where a lack was felt was in function approximation. The behavior of function approximators with small numbers of inputs seems to be fairly well understood, but when we move to higher dimensional spaces visualization of the internal workings of an approximator becomes impossible. There is a complex interaction going on between the approximator architecture, the distribution of the training samples, and the level of generalization being used, not to mention the learning rates. Since direct visualization is impossible in these higher dimensions, a set of statistical tools needs to be developed that can encapsulate the relevant interactions and allow the algorithm designer to adjust the approximator's parameters accordingly.

Reinforcement learning as it stands today is useful tool. It can be applied with some success to real-world problems, but has not yet realized its full potential. As work in the field continues, RL should become more reliable, easier to use, and able to handle higher dimensional problems.

## 7.3 Suggestions for future work

We have already mentioned a couple of possible research directions in the previous section. In this final section we will suggest a few more specific research topics suggested by our work.

The problems covered in this thesis involved measurement scheduling for pure state estimation problems. The most obvious extension to the work we have done would be the use of reinforcement learning in an examination of the measurement scheduling problem in the context of integrated control and estimation problems. A significant amount of work has been done in this area already, but it revolves mainly around discrete state spaces and POMDP's. A successful treatment of this problem for multi-dimensional continuous state spaces with nonlinear transitions would be a real achievement.

Also on the measurement scheduling side, a real issue was the dependence of our formulations on cost measures that were largely fictitious. With nonlinear estimators such as the EKF, we really have no way of knowing for sure if our covariance estimate reflects the reality of the situation. If it does not, the effects on the learning process could be disastrous. We bypassed this issue by working with relatively high-accuracy systems and doing statistical checks for verification. If some way could be found of integrating actual measurements into the cost structure it might be possible to tackle more difficult problems.

In a related issue more on the reinforcement learning side of things, further study would be useful on the issue of robustness to non-Markovianness. When we treated nonlinear estimation problems with RL we used suboptimal estimators and hence departed from the realm of strict Markov transitions. With this departure all bets were off on the effectiveness of our algorithms. However, since we were dealing with relatively low-noise systems and fairly accurate estimates, our transitions were *almost* Markov and we were able to achieve a fair level of success. If the noise levels were to be increased or the measurement frequency significantly decreased, at some point the non-Markovianness of the system would presumably overwhelm the algorithm's robustness, and learning would fail. A detailed study of the conditions under which learning begins to fail would be of substantial interest.

A final area for potential future work would involve the G-functions described in Chapter 3. So far only the special case of linear tranformations has been treated in any detail. The transformations described by Baker [9] are potentially much more general, however, and it would be interesting to see which of the possibilities could prove useful. Also, an alternative to Advantage Learning which we called G-Learning was proposed. Though we showed an example where G-Learning corrected for flaws in the Advantage Learning algorithm, we did not provide a convergence proof. If such a proof could be found, it would be a significant contribution.

# Appendix A

# Function Approximation using CMAC's

The mathematical subject of function approximation has been studied for centuries. It encompasses a variety of subfields, but at a fundamental level its concern is the recognition of patterns, or learning. Interpolation, extrapolation, filtering, and compact representation of data sets can all be thought of in terms of the same fundamental concepts.

For precisely this reason, the research into the workings of the human brain and attempts to develop artificial intelligence over the past few decades have prompted an explosion in the field of function approximation. The focus on the brain and cognition has led to the widespread use of the term *neural networks* to refer to many of the newer methods of function approximation, but the principles which drive them are little different than those that have long been used for simple polynomial curve fits.

One of the first function approximation architectures to come out of this new research was the multilayer perceptron, or MLP. MLP's have received a large amount of attention from researchers in a variety of disciplines, and despite the development of a wide array of alternative architectures are still the most popular "neural network" in use today. Indeed, MLP's and neural networks have become so linked that many people think they are synonymous.

The primary reason for the popularity of multilayer perceptrons is probably the existence of a theorem that shows that MLP's are capable of approximating an arbitrary finite-dimensional function to an arbitrary degree of accuracy. Though this theorem at first seems to be an extremely promising result, there are two problems which make MLP's less than ideal in practice. The first is that the arbitrary level of accuracy guaranteed by the theorem applies only at the data points used for training and requires an indefinite number of units. No way has yet been found to determine the appropriate number of units, and as the number of units used increases the danger of overfitting and poor generalization increases. The second problem is that even when an appropriate network structure has been selected, finding the set of weights which minimizes the mean squared approximation error can be an extremely difficult task. Since the mean squared approximation error is generally non-convex in the weights, training via gradient descent is slow and subject to difficulties with local minima.

## A.1 Linear Function Approximators

The problem with local minima occurs with virtually all nonlinear approximation architectures, and for this reason many researchers have begun to favor linear approximation architectures. Linear approximation architectures form their output from a linear combination of $C$ basis functions $\phi_i(x)$:

$$\tilde{f}(x,r) = \sum_{i=1}^{C} w_i^T \phi_i(x) \tag{A.1}$$

where

$$r = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_C \end{bmatrix}. \tag{A.2}$$

The basis functions themselves need not be linear; a common choice for a two dimensional input $x = [x_1 \quad x_2]$ might be $\phi_1 = x_1^2$, $\phi_2 = x_1 x_2$, $\phi_3 = x_2^2$. Because they are combined in a linear fashion, however, the least squares approximation error

$$J = \left(f(x) - \tilde{f}(x,r)\right)^2 \tag{A.3}$$

is convex in $r$. The problem of finding the weights which minimize $J$ is then a simple one, though the accuracy of the approximation will ultimately depend on the choice of basis functions. Unlike MLP's, linear function approximators offer no guarantee that the particular basis functions chosen will be able to achieve any desired level of accuracy.

## A.2 Local Function Approximators

Along with the trend toward the use of linear function approximators there has been a trend toward the use of *local* function approximators. With a local function approximator, the influence of each data point is limited to its immediate neighborhood in the state space. In addition to providing computational savings vs. global architectures such as MLP's, local architectures may do better at approximating functions which are themselves spatially localized.

A commonly used local architecture is the radial basis function network (RBFN). With an RBFN, the user distributes a series of nonlinear (usually gaussian) basis functions $\phi_i(x)$ through the state space and specifies the extent of their influence. The approximator output is then calculated according to Equation (A.1). As long as the locations of the basis functions are fixed, this can be considered a linear architecture and training is a simple matter. If the locations of the basis functions are adjusted automatically to minimize the approximation error, however, we have a nonlinear architecture that is subject to the dangers of local minima.

## A.3 CMAC's

The cerebellar model articulation controller (CMAC) developed by Albus in 1971 [1] is a local, linear function approximation architecture designed in a way that lends itself to extremely efficient training and evaluation. CMAC's have been used extensively at the University of New Hampshire [39], and researchers there have compiled a standard library of C code for implementation of the architecture as well as an extensive bibliography of CMAC related papers. As neither the term "CMAC" nor it's expansion of "cerebellar model articulation controller" is particularly descriptive, Sutton refers to the architecture as "tile coding" in his book on reinforcement learning [56].

A CMAC generates its output by linear combination of $C$ basis functions according to Equation (A.1). Each of the basis functions, often referred to as *layers*, maps the entire input space to an $M$ dimensional vector, where $M$ is referred to as the *memory* of the CMAC. Each of these vectors contains a single element equal to 1, with the rest equal to 0. Since each basis function contains only a single nonzero element, the multiplication in Equation (A.1) can be reduced to a simple summation,

$$\tilde{f}(x, r) = \sum_{i=1}^{C} w_i^T \phi_i(x) = \sum_{i=1}^{C} w_i[A_i] \tag{A.4}$$

where $A_i$ is the index of the nonzero element of the $i$th basis vector.

That index is specified by dividing each layer into a uniform grid of *receptive fields.* The particular receptive field in which an input point is located determines the location of the 1 in the vector $\phi_i(x)$. This is illustrated for a two dimensional space in Figure A-1. The input space is divided up into a grid with four integer-numbered receptive fields. Since $x$ is located in the third field, $\phi(x)$ has a 1 in the third position.

### A.3.1 Hashing

There are two problems with this type of coding. First of all, it can't handle unbounded state spaces with a finite grid size. To do so would require that $\phi(x)$ map to infinite-

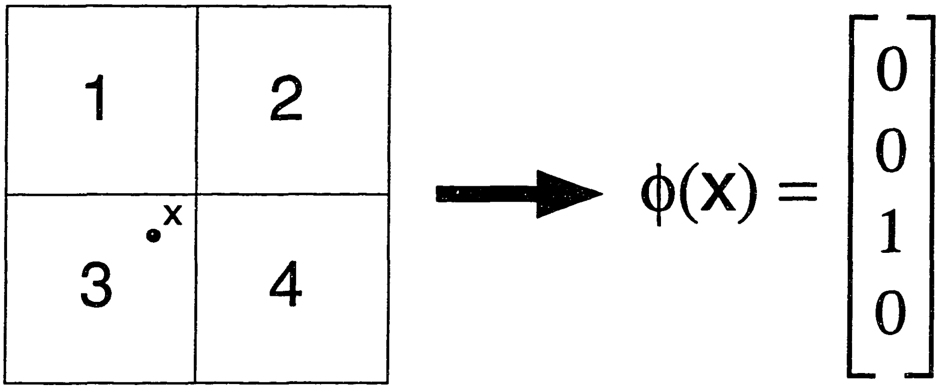
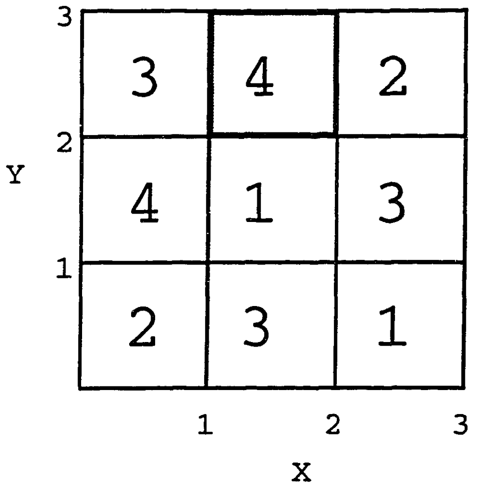


Figure A-1: Mapping of $\phi(x)$ for a simple 2D case.

dimensional vectors, which would thus require infinite memory. Even with bounded state spaces, the amount of memory required increases exponentially with the number of inputs to the approximator. These problems are both solved by adding an intermediate step to the process described above. Each input point $x \in \mathcal{R}^n$ is mapped to a point in an integer address space $x' \in \mathcal{Z}^n$ and then into the finite memory of size $M$ using a pseudorandom mapping known as a *hashing function.* This process is illustrated in Figure A-2.

The use of a hashing function introduces a source of error to the function approximator. This is because modification of a weight associated with one grid location affects the evaluation of the approximator at every other point which hashes to the same memory location. For instance, in Figure A-2 training of the approximator to match a data point at $(1.2, 3.4)$ would also affect the output at $(0.6, 1.5)$ in a completely different grid location. Though this would at first seem to be a very damaging behavior, it is rarely a serious problem. Most function approximation problems deal only with a small portion of the possible input space, and as long as the memory is made large enough compared to the area of the input space actually used, collisions of the sort described above are fairly rare. The noise introduced by use of hashing functions is further mitigated by the use of multiple layers in the CMAC.



$$(1.2\,,2.4) \rightarrow (1\,,2) \rightarrow 4$$

$$\phi(x) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Figure A-2: Hashing of 9 grid locations into a memory of size 4 using the transcendental numbers $e$ and $\pi$. The hashing function maps a point $(i, j)$ to the number $1 + (a + b)\%4$ where $a$ is the $i$th digit of $e$ and $b$ is the $j$th digit of $\pi$.

### A.3.2 Layering

In the traditional Albus CMAC, the receptive fields in each layer are $C$ units wide in each dimension, and each layer is offset by one unit in each dimension relative to the previous layer in a sort of hyperdiagonal arrangement. This sort of arrangement ensures that a change of 1 unit in any input parameter causes a change in exactly one basis function, which makes for uniform quantization throughout the input space. It also determines an area of generalization for each input point, equal to the total extent of all the receptive fields which it activates. This is shown graphically in Figure A-3. Each time a data point is presented to the CMAC for training, the approximator output is affected throughout that point's area of generalization.[1]

Research at the University of New Hampshire has shown that the Albus CMAC's hyperdiagonal arrangement does not provide the most uniform generalization [2] [42]. In an effort to provide a better alternative arrangement, the concept of the *displacement vector* was developed. The displacement vector $D$ for a CMAC specifies the offset of each layer rel-

[1] Of course, the use of hashing as explained above means that the approximator output will be affected at other random locations in the state space as well.

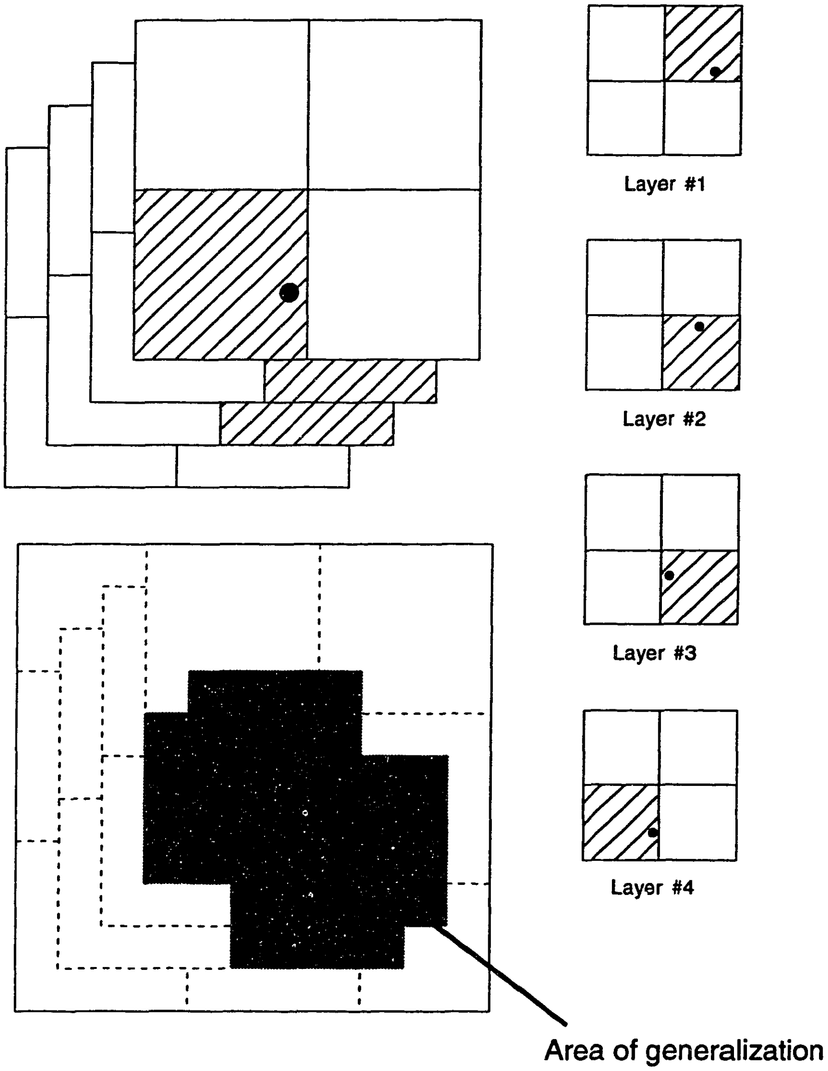


Figure A-3: Layering of a CMAC. The receptive fields activated by the marked input point are shaded. At bottom left the input point's area of generalization is shown. This is the total extent of all the active receptive fields.

|  | Displacement Vector |  |  |
|---|---|---|---|
| $C$ | 1 input | 2 inputs | 3 inputs |
| 4 | [1] | $[1 \quad 1]$ | $[1 \quad 1 \quad 1]$ |
| 8 | [1] | $[1 \quad 3]$ | $[1 \quad 3 \quad 3]$ |
| 16 | [1] | $[1 \quad 3]$ | $[1 \quad 3 \quad 5]$ |

Table A.1: Displacement vectors suggested by Glanz and Miller's heuristic rule for various numbers of layers and inputs.

ative to the one before. For instance, the hyperdiagonal arrangement of the Albus CMAC is specified by $D = [1 \quad 1 \quad \ldots \quad 1]$. Parks and Militzer [42] have done a comprehensive study to find the displacement vectors which provide the most uniform generalization for a variety of different circumstances, summarizing their results in a series of tables. Based on those results, Miller and Glanz [39] have developed a heuristic method for choosing displacement vectors which produces roughly equivalent levels of generalization.

Glanz and Miller's method works as follows. For a CMAC with $N$ inputs and $C$ layers, candidate offsets for $D$ include the integers from 1 to $C/2$ which are not factors of $C$ or integer multiples of factors of $C$, with 1 included. From these candidates $N$ values are selected. If more than $N$ candidates are available, then any $N$ can be selected. If less than $N$ are available, then some candidates will have to be selected more than once. However, Glanz and Miller suggest that in this case it is better to increase the number of layers until $N$ unique candidates are available. Table A.3.2 summarizes the choices suggested by this heuristic rule for a few combinations of layers and inputs. Figure A-4 shows how the choice of displacement vector can affect the area of generalization for a two input example. At left the Albus displacement vector of $D = [1 \quad 1]$ is used, and at right the vector suggested by Glanz and Miller $D = [1 \quad 3]$ is used.

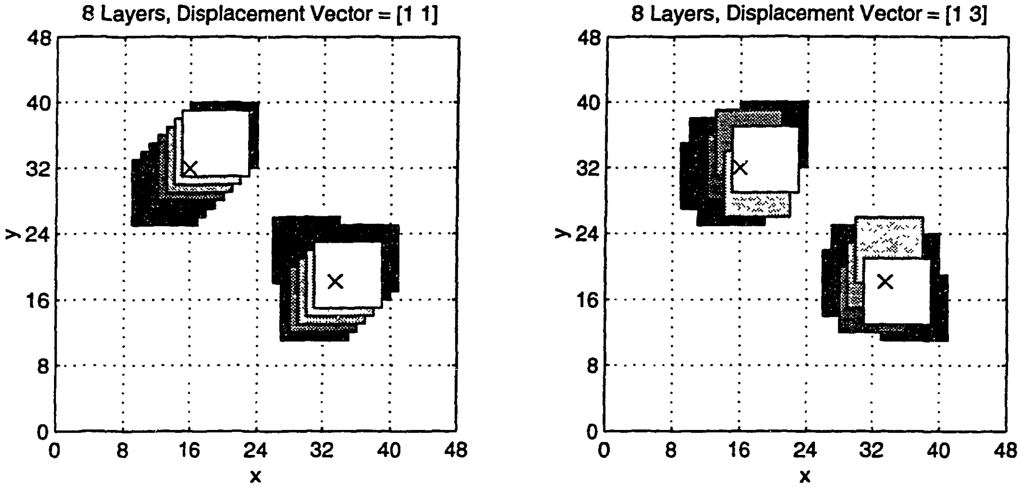


Figure A-4: Plots showing the effect of the displacement vector on the generalization area. At left the generalization areas for two different points are plotted using the standard Albus CMAC displacement vector of [1 1]. At right, the generalization areas for the same two points are shown when a displacement vector of [1 3] is used.

### A.3.3 Training

Since the CMAC's output at any given point is just the sum of the $C$ weights activated by the input point, training is an extremely simple matter. Given an input point $x$ and a desired output $y$, we simply add

$$\Delta w = \beta/C\left(y(x) - \tilde{f}(x,r)\right) \tag{A.5}$$

to each of the $C$ relevant weights, where $\beta \in [0,1]$ is the learning rate. When $\beta = 1$ the approximator is trained to reproduce the data point exactly; when $\beta = 0.5$ it is trained to produce an output halfway between the old output and the new data point; and when $\beta = 0$ no change is made to the approximator's output. In the UNH CMAC code, which makes extensive use of integer and bitwise operations in the interest of computational efficiency, $\beta$ is specified in integer powers of 1/2. Thus $\beta = 1$ would be a learning rate of 0.5, $\beta = 2$ would be a learning rate of 0.25, and so on.

There is one further complication involved in the training of CMAC's, however. Small fluctuations in training data over a long period of time can result in individual weights drifting toward large positive values while other weights drift toward large negative values even though the average values remain in the appropriate range. Since the averages are unaffected, this creates no direct difficulties. However, indirect problems may arise. When combined with hashing collisions, for instance, individual large weights may exert an influence in the absence of their moderating counterparts. Depending on the physical method of implementation of the approximator there may also be weight saturation or machine precision issues.

In the UNH CMAC implementation this is addressed by the addition of a penalty on large weight magnitudes to Equation (A.5). The new training equation becomes

$$\Delta w_i\left[A_i\right] = \frac{1}{C}\left[\beta_1\left(y\left(x\right) - \tilde{f}\left(x,r\right)\right) + \beta_2\left(y\left(x\right) - w_i\left[A_i\right]\right)\right] \tag{A.6}$$

where again, $A_i$ is the index of the nonzero element of the $i$th basis vector. In this equation $\beta_2$ controls the relative importance of weight magnitude normalization; generally this is taken to be substantially smaller than $\beta_1$.

The training process is shown graphically in Figure A-5. Random points from the interval $[0,1]$ were chosen one at a time to train the function $y = \sin(2\pi x)$.

### A.3.4 Other CMAC Variants

The concept of using displacement vectors for layering to produce more uniform generalization was not a part of the original CMAC concept. Other extensions to the architecture originally put forth by Albus have been proposed as well. At the cost of increased computation, nonuniform activation functions can be used to generate continuous and/or smooth approximator outputs [38]. Also, nonuniform scaling of the approximator inputs can be used to give greater weight to important areas of the state space [39]. This concept was tested in the context of reinforcement learning by Santamaría [49]. A variety of different options for dealing with memory hashing exist, and are discussed in the UNH CMAC guide [39].

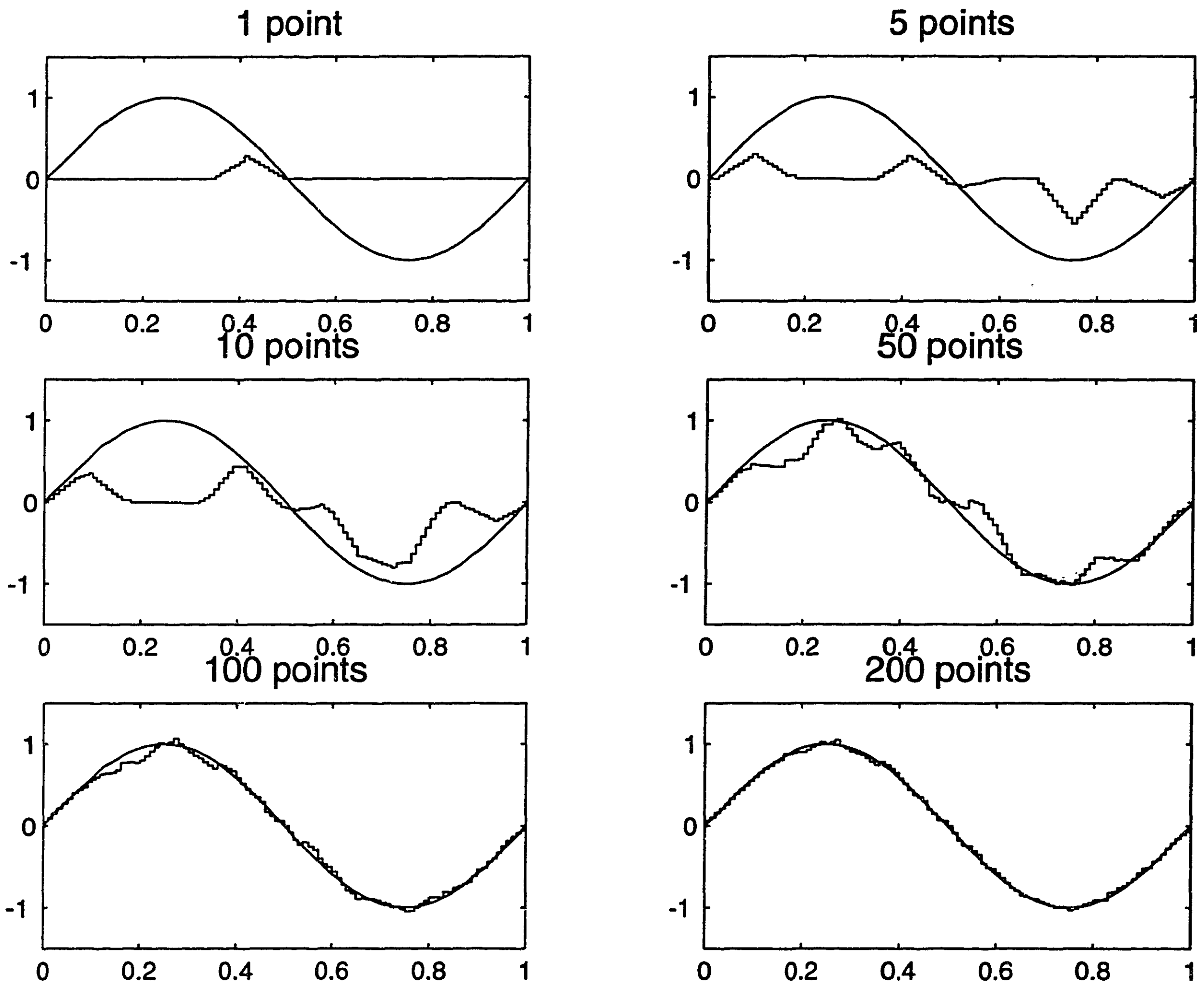


Figure A-5: Plots showing the CMAC training process for approximation of a simple sine curve.

### A.3.5 Pros and Cons

To conclude our discussion of CMAC function approximators, an evaluation of the architecture relative to other approximator types seems in order. CMAC's have three major advantages. As function approximators go, both training and evaluation using CMAC's are extraordinarily fast. This is a result of the discretization and reliance on integer arithmetic. As mentioned above, the CMAC architecture is linear, so there is no potential for problems with local minima in training. Finally, CMAC's gain the benefits of local approximation architectures without having to deal with explicit placement of basis functions. These are all very attractive features.

On the down side, the discretization which gives CMAC's so much speed takes away from the architecture in other areas. For one thing, the discrete nature of its outputs may be undesirable for some applications. CMAC's can be adapted to generate continuous outputs, but only at a considerable cost in speed. Discretization also necessitates a significantly greater concern with scaling of inputs than may be the case with other approximation architectures.

Localization, which we listed among the advantages of CMAC's, may be a disadvantage in some situations. It can make CMAC's more vulnerable to holes in the training data than nonlocal architectures such as MLP's. Another disadvantage that CMAC's have relative to MLP's is in the area of input pruning. If an MLP is presented with extra inputs which bear no functional relationship to the desired output, the weights associated with those inputs will eventually go to zero and automatic pruning can be performed. CMAC's do not handle this sort of extraneous information as well, and pruning of inputs, while not impossible, would be much more involved than with MLP's.

Of course, no single function approximator architecture will be the best for all applications. Indeed, it is often hard to make objective judgements as to which architecture is best for any specific problem, much less the general case. CMAC's make an excellent addition to the family of function approximators – one that is probably under-appreciated in the learning community, which is still mostly dominated by the MLP and its variants.

# Appendix B

# Function Approximation and Convergence with RL

## B.1 Prediction

In this section we will discuss convergence results for approximating the value function of a fixed policy using a function approximator.

### B.1.1 Monte-Carlo methods

Monte-Carlo methods generate unbiased samples of the value function of a fixed stationary policy, so stochastic approximation (SA) methods are guaranteed to find at least a local minimum in the limit as t goes to infinity, providing the SA conditions on the stepsize $\alpha$ are met:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \quad \sum_{t=0}^{\infty} \alpha_t^2 < \infty \tag{B.1}$$

If the function approximation architecture is linear, then convergence to a global minimum is guaranteed. Simulated annealing offers the guarantee that the global minimum can be obtained if the function being approximated and the architecture chosen are appropriately conditioned and the temperature of the system is lowered slowly enough. However simulated annealing methods may take much longer to converge than gradient descent methods.

### B.1.2 Linear TD($\lambda$)

Bootstrapping methods are in general much more practical to implement than Monte-Carlo methods, but suffer from the problem that the samples used for updating are normally biased, so that many of the normal convergence guarantees no longer apply. A substantial amount of effort as been expended over the past decade in attempting to prove convergence of bootstrapping algorithms for policy evaluation using function approximation. The most general result showing convergence of TD($\lambda$) using function approximation to date is the one by Tsitsiklis and Van Roy, described in their 1997 paper [63]. It makes the following assumptions:

- The state space is finite or countably infinite.
- The underlying Markov chain is irreducible and aperiodic with unique steady state probability vector.
- The single step transition costs have finite second moment.
- The basis functions upon which the linear function approximator is constructed are linearly independent and have finite second moments at each state.
- The step sizes of the updates meet the standard stochastic approximation requirements.
- One further assumption which is hard to describe without introducing a lot of notation, but which basically says that certain expectations involving the basis functions and the single step costs are finite. See [63] for details (this is Assumption 3)

Under the above assumptions TD($\lambda$) with linear function approximators (as defined by Tsitsiklis) converges with probability 1. Note that Tsitsiklis' algorithm generates samples according to the state transition probability matrix for the policy being evaluated. Also, it does not guarantee convergence to the actual cost function, only to within a certain distance of the projection of the cost function onto the space spanned by the basis functions being used. So we only get as close to the actual cost function as the basis functions we have chosen allow.

Earlier papers have proved various things about TD($\lambda$) with function approximation. These results can all be regarded as special cases of the conditions required by Tsitsiklis' proof.

Tsitsiklis and Van Roy [64] have also extended their results with TD($\lambda$) to average-cost formulations. The results are similar in nature to those described above.

### B.1.3 RLS TD

Bradtke has described a version of the basic temporal difference algorithm which uses recursive least squares (RLS) for its updates rather than stochastic approximation. He has shown convergence with probability 1 using linear function approximators and generating trajectories according to the Markov chain's transition matrix as in Tsitsiklis' method. The switch from stochastic approximation to RLS removes the stochastic approximation learning rate and its associated conditions, but adds a new condition requiring that a quantity associated with the RLS algorithm be non-singular (see Condition A.1, p. 45, [16]. It is important to note that the two main conditions, a linear architecture with linearly independent basis functions and trajectories generated according to the underlying Markov model are the same as in Tsitsiklis' result.

## B.2 Control

### B.2.1 Approximate Policy Iteration

Bertsekas and Tsitsiklis show in their book [13] that approximate policy iteration converges under the following conditions:

- The maximum error in the approximate value function after a policy evaluation step is less than $\varepsilon$. This does not specify the prediction method used for policy evaluation.
- The maximum error incurred during policy update is less than $\delta$ (the exact quantity that $\delta$ measures is described on pp 275–276 of [13]). This allows for approximation of policies as well as approximations of value functions.

Then in the limit the maximum error (approximate cost vs. optimal cost) will be less than

$$\frac{\delta + 2\gamma\varepsilon}{(1-\gamma)^2} \tag{B.2}$$

where $\gamma$ is the discount factor.

This puts no explicit limitations on the form of the function approximator, though for nonlinear approximators we may be subject to local minima which could make it hard to get a tight value on $\varepsilon$.

The use of an approximate policy iteration algorithm implies all of the restrictions which come with policy iteration: finite or countably infinite state space, finite number of possible control choices, markovianness of the underlying process, etc.

With this result we get convergence of approximate policy iteration with any of the prediction methods shown to converge in the section on prediction. Unfortunately, this does not guarantee the convergence of generalized approximate policy iteration, where policies are incompletely evaluated, as we have no results telling us that we can reliably get within any particular e in a finite amount of time. In practice, though, generalized policy iteration using a prediction algorithm guaranteed to converge usually works just fine.

### B.2.2 Approximate Value Iteration

Bertsekas and Tsitiklis also show convergence for an algorithm which they call approximate value iteration. This algorithm assumes the use of a least-squares approximation architecture for the cost-to-go function. Implementation of the approximate value iteration algorithm requires a model. It also requires that the function approximation architecture be rich enough so that at each stage $k$ of the algorithm

$$\|J_{k+1} - TJ_k\|_\infty \leq \varepsilon \tag{B.3}$$

for some $\varepsilon > 0$. Here $J_k$ is the approximate cost function at stage $k$, and $T$ is the dynamic programming operator. If no such $\varepsilon$ exists, the possibility of divergence exists (as shown in the example on p. 334 of [13]).

### B.2.3 Value Iteration with State Aggregation

As an alternative to approximate value iteration, Bertsekas and Tsitsiklis present an algorithm which uses a function approximator derived from state-aggregation instead of least-squares minimization. With state aggregation, groups of states with similar costs in a finite state space are lumped together and used in a table-based lookup. Bertsekas and Tsitsiklis show convergence by demonstrating that the approximate algorithm can also be viewed as an exact value iteration algorithm for the aggregated problem.

In addition to the need for a finite state space and the requirement that a model be available, the state aggregation algorithm requires that either all policies in the base problem be proper or all of the following conditions are true for convergence with probability 1:

- At least one proper policy exists.
- All improper policies have infinite cost for some initial state.
- All single-step costs are nonnegative.
- The algorithm is initialized with a nonnegative cost vector.

### B.2.4 Bradtke's LQR Algorithm

Bradtke has also proven convergence with probability one for an algorithm which solves the linear quadratic regulator (LQR) problem with unknown model [17]. His convergence result relies on exact knowledge of the form of the model and the particular qualities of linear systems which enable him to choose a linear approximation architecture which is capable of exactly approximating the cost of a policy and provides a simple method of calculating the policy update step. It requires perfect state knowledge and an initially stable controller. It is remarkable, however, in that it uses continuous state and action spaces, whereas the above results are all for at most countably infinite state and action spaces.

### B.2.5 Residual Gradient Methods

Off-policy methods such as Q-learning have the potential for divergence if used directly with function approximation. Baird [8] claims to have shown that if gradient descent is performed on the Bellman error residual instead of just the Q-function, the resulting algorithm is guaranteed to converge to the optimal Q-function given an architecture with sufficient power to approximate and a deterministic system. The globally optimal solution is due to the fact that the Bellman error residual has no local minima. In the case of a stochastic system two independent samples of each state are needed to get an unbiased estimate of the residual, which in most practical situations means that a model will be needed.

## B.3 Non-converging methods

Though much of the reinforcement learning literature is devoted to the quest for convergence proofs, convergence is not essential for an algorithm to have practical value. There are certainly roles for methods that are not guaranteed to converge. Most practical applications of reinforcement learning fall in this category, with Tesauro's TD-Gammon [60] being an outstanding example.

Negative convergence results can in some ways be more useful than positive ones, as they tell us which algorithms *not* to pursue. They can also give us an idea of safe parameter ranges. An example of this is given in Chapter 3, where we show that Advantage Learning is likely to diverge when the accentuation of cost differentials is too large compared to the learning rate.

It is worth noting that one could easily find methods that converge, but not to an optimal (or even near optimal) solution, whether due to an overly quick reduction in learning rate or some other reason. These methods are also unlikely to be of practical use.

Convergence often depends on the gradual reduction of a learning rate. For non-stationary systems, one might deliberately keep a high learning rate and avoid convergence in order to maintain flexibility in dealing with changes in the environment. In these situations, non-converging methods could thus be of great utility. There may be other situations where non-converging methods have practical use, since the space of available methods and possible uses is large.

There are also cases, as witnessed in this thesis, where learning is definitely taking place but convergence to a single policy does not occur despite gradual reductions in learning and exploration rates. In these situations useful results can often be extracted by testing policies as they are generated by the learning algorithm.

# Bibliography

[1] J. S. Albus. A theory of cerebellar functions. *Mathematical Biosciences*, 10:25–61, 1971.

[2] P.-C. E. An. *An improved multi-dimensional CMAC neural network: receptive field function and placement.* PhD thesis, University of New Hampshire, 1991.

[3] M. S. Andersland. On the optimality of open-loop lqg measurement scheduling. *IEEE Transactions on Automatic Control*, 40(5):1796–1799, October 1995.

[4] M. Aoki and M. T. Li. Optimal discrete-time control system with cost for observation. *IEEE Transactions on Automatic Control*, 41(5):689–701, May 1996.

[5] M. Athans. On the determination of optimal costly measurement strategies for linear stochastic systems. *Automatica*, 8:397–412, 1972.

[6] D. Avitzour and S. R. Rogers. Optimal measurement scheduling for prediction and estimation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(10):1733–1739, October 1990.

[7] L. C. Baird, III. Advantage updating. Technical Report WL-TR-93-1146, Wright Patterson Air Force Base, 1993. Available from the Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145.

[8] L. C. Baird, III. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning: Proceedings of the Twelfth International Conference*, 1995.

[9] W. L. Baker. *Learning via Stochastic Approximation in Function Space.* PhD thesis, Harvard University, 1997.

[10] F. J. Barbera. Closed-form solution for ballistic vehicle motion. *Journal of Spacecraft*, 18:52–57, Jan-Feb 1981.

[11] R. R. Bate, D. D. Mueller, and J. E. White. *Fundamentals of Astrodynamics.* Dover, 1971.

[12] D. Bertsekas. *Dynamic Programming and Optimal Control*, volume I and II. Athena Scientific, 1995.

[13] D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic Programming.* Athena Scientific, 1996.

[14] S. S. Blackman. *Multiple-Target Tracking with Radar Applications.* Artech House, 1986.

[15] S. S. Blackman. Multitarget tracking with an agile beam radar. In Y. Bar-Shalom, editor, *Multitarget-Multisensor Tracking: Applications and Advances*, volume 2. Artech House, 1992.

[16] S. J. Bradtke and A. G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57, 1996.

[17] S. J. Bradtke, B. E. Ydstie, and A. G. Barto. Adaptive linear quadratic control using policy iteration. In *Proceedings of the American Control Conference*, pages 3475–9, 1994.

[18] D. Castañon, A. Chao, S. Gully, R. Washburn, and J. Wissinger. Phase I final report: Stochastic dynamic programming for farsighted sensor management. Technical report, Alphatech, Inc., 1995.

[19] D. E. Cerrato. Modeling unknown dynamical systems using adaptive structure networks. Master's thesis, Massachusetts Institute of Technology, 1993.

[20] R. C. Chen and W. D. Blair. Optimal measurement scheduling for track accuracy control for cued target acquisition. *Proceedings of the SPIE- The International Society for Optical Engineering*, 2759:406–417, 1996.

[21] C. A. Cooper and N. E. Nahi. An optimal stochastic control problem with observation cost. *IEEE Transactions on Automatic Control*, 16:185–189, April 1971.

[22] F. Daum. A system approach to mulitple target tracking. In Y. Bar-Shalom, editor, *Multitarget-Multisensor Tracking: Applications and Advances*, volume 2. Artech House, 1992.

[23] P. Dayan. The convergence of TD($\lambda$) for general $\lambda$. *Machine Learning*, 8:341–362, 1992.

[24] E. Feron and C. Olivier. Targets, sensors, and infinite-horizon tracking optimality. In *Proceedings of the 29th Conference on Decision and Control*, pages 2291–2292, 1990.

[25] A. Gelb, editor. *Applied Optimal Estimation.* M.I.T. Press, 1974.

[26] M. Harmon and L. C. Baird. Residual advantage learning applied to a differential game. In *Proceedings of the International Conference on Neural Networks*, 1995.

[27] S. Haykin. *Neural Networks: A Comprehensive Foundation.* MacMillan, 1994.

[28] H. Heffes and S. Horing. Optimal allocation of tracking pulses for an array radar. *IEEE Transactions on Automatic Control*, 15(1):81–87, February 1970.

[29] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[30] H. J. Kushner. On the optimum timing of observations for linear control systems with unknown initial state. *IEEE Transactions on Automatic Control*, AC-9:144–150, April 1964.

[31] R. E. Larson, R. M. Dressler, and R. S. Ratner. Application of the extended kalman filter to ballistic trajectory estimation. Technical report, Stanford Research Institute, 1967.

[32] M. Littman, A. Cassandra, and L. P. Kaelbling. Efficient dynamic-programming updates in partially observable markov decision processes. Technical Report CS-95-19, Brown University, 1995.

[33] B. W. McCormick. *Aerodynamics, Aeronautics, and Flight Mechanics.* Wiley, 1979.

[34] R. K. Mehra. A comparison of several nonlinear filters for reentry vehicle tracking. *IEEE Transactions on Automatic Control*, AC-16(4):307–319, August 1971.

[35] R. K. Mehra. Optimization of measurement schedules and sensor designs for linear dynamic systems. *IEEE Transactions on Automatic Control*, AC-21(1):55–64, February 1976.

[36] L. Meier, III, J. Peschon, and R. M. Dressler. Optimal control of measurement subsystems. *IEEE Transactions on Automatic Control*, AC-12(4):528–536, August 1971.

[37] D. J. Mellefont and R. W. H. Sargent. Optimal measurement policies for control purposes. *International Journal of Control*, 26:595–602, 1977.

[38] W. T. Miller, E. An, F. H. Glanz, and M. J. Carter. The design of cmac neural networks for control. In *Adaptive and Learning Systems*, volume 1, pages 140–145, 1990.

[39] W. T. Miller and F. H. Glanz. *UNH_ CMAC Version 2.1: The University of New Hampshire Implementation of the Cerebellar Model Arithmetic Computer - CMAC*, 1996.

[40] N. M. Olgac, C. A. Cooper, and R. W. Longman. The impact of costly observations and observation delay in stochastic optimal control problems. *International Journal of Control*, 41:769–785, 1985.

[41] Y. Oshman. Optimal sensor selection strategy for discrete-time state estimators. *IEEE Transactions on Aerospace and Electronic Systems*, 30(2):307–314, April 1994.

[42] P. C. Parks and J. Militzer. Improved allocation of weights for associative memory storage in learning control systems. In *IFAC Design Methods of Control Systems*, 1991.

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

[44] J. Peng and R. J. Williams. Incremental multi-step Q-learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 226–232, 1994.

[45] H. V. Poor. *An Introduction to Signal Detection and Estimation.* Springer-Verlag, 1988.

[46] R. Popoli. The sensor management imperative. In Y. Bar-Shalom, editor, *Multitarget-Multisensor Tracking: Applications and Advances*, volume 2. Artech House, 1992.

[47] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[48] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Dept., 1994.

[49] J. C. Santamaría, R. S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2), 1998.

[50] Y. Sawaragi, T. Soeda, Y. Tomita, and I. Imai. Optimal timing of the observation for the state estimation and control of the stochastic discrete linear system. *International Journal of Control*, 27:621–637, 1978.

[51] M. Shakeri, K. R. Pattipati, and D. L. Keinman. Optimal measurement scheduling for state estimation. *IEEE Transactions on Aerospace and Electronic Systems*, 31(2):716–728, April 1995.

[52] S. Singh. Learning without state-estimation in partially observable markovian decision problems. In *Proceedings fo the Eleventh International Machine Learning Conference*, 1994.

[53] S. P. Singh and R. C. Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16:227–233, 1994.

[54] M. I. Skolnik. *Introduction to Radar Systems.* McGraw Hill, 1980.

[55] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[56] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[57] S. Tanaka and T. Okita. On suboptimal selection of observation times in a linear discrete dynamical system. *International Journal of Control*, 34:143–152, 1981.

[58] S. Tanaka and T. Okita. Optimal timing of observations for state estimation in a one-dimensional linear continuous system. *Automatica*, 21(3):329–331, 1985.

[59] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[60] G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[61] G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–67, 1995.

[62] S. Thrun and A. Schwartz. Issues in function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School.* Lawrence Erlbaum Publishers, Hillsdale, NJ, 1993.

[63] J. Tsitsiklis and B. V. Roy. An analysis of temporal-difference learning with function approximation. *IEEE Tranactions on Automatic Control*, 42:674–690, May 1997.

[64] J. Tsitsiklis and B. V. Roy. Average cost temporal-difference learning. Technical Report LIDS-P 2390, Laboratory for Information and Decision Systems, 1997.

[65] J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16:185–202, 1994.

[66] G. Van Keuk and S. S. Blackman. On phased-array radar tracking and parameter control. *IEEE Transactions on Aerospace and Electronic Systems*, 29(1):186–194, January 1993.

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

[68] R. J. Williams and L. C. Baird, III. Tight performance bounds on greedy policies based on imperfect value functions. Technical Report NU-CCS-93-14, Northeastern University, 1993.

# THESIS PROCESSING SLIP

FIXED FIELD: ill. ______________ name ______________

index ______________ biblio ______________

COPIES: [✓ Archives] [✓ Aero] Dewey Eng Hum

Lindgren Music Rotch Science

TITLE VARIES: ► □ ______________

NAME VARIES: ► □ ______________

IMPRINT: (COPYRIGHT) ______________

COLLATION: ~~264p~~ 263P

ADD: DEGREE: ________ ► DEPT.: ______________

SUPERVISORS: ______________

NOTES:

cat'r: date:

page: S 41

► DEPT: Aero

► YEAR: 1999 ► DEGREE: Ph.D.

► NAME: ROGERS, [illegible] Eric