Adaptive Convex Enveloping for Multivariate Convex Dynamic Programming

by Sheng Yu

B.S. in Statistics, June 2007, Nankai University
M.A. in Applied Statistics, April 2009, University of Michigan

A Dissertation submitted to

The Faculty of
School of Engineering and Applied Science
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

August 31, 2012

Dissertation directed by

Johan René van Dorp
Professor of Engineering Management and Systems Engineering

The School of Engineering and Applied Science of The George Washington University certifies that Sheng Yu has passed the Final Examination for the degree of Doctor of Philosophy as of July 9, 2012. This is the final and approved form of the dissertation.

Adaptive Convex Enveloping for Multivariate Convex Dynamic Programming

Sheng Yu

Dissertation Research Committee:

Johan R. van Dorp, Professor of Engineering Management and Systems Engineering, Dissertation Director

Enrique Campos-Náñez, Senior Software Engineer of Alere, Inc., Committee Member

Hernan G. Abeledo, Associate Professor of Engineering Management and Systems Engineering, Committee Member

Miguel Lejeune, Assistant Professor of Decision Sciences, Committee Member

Julie J.C.H. Ryan, Associate Professor of Engineering Management and Systems Engineering, Committee Member

Gregory L. Shaw, Associate Professor of Engineering Management and Systems Engineering, Committee Member

Acknowledgment

I pursued the Ph.D. degree to understand and experience the academic world. Switching from the familiar field of statistics, I chose to get a degree in optimization, in order to double my skill reserves as well as the problems that I can solve. As an problem-solving enthusiast, I was lucky to find a dissertation topic with an origin of an important real-world application, and developed a method that could also potentially help many people solve their problems.

I am very thankful to my parents and grandparents. They work hard to provide me a life without pressure and a home where I can always find shelter. They also educate me so that I do not live like a spoiled son, but fight like a warrior. They pass to me philosophy, so that I can see clearly; and wisdom for living, so that I get less hurt on my road. I owe all of my current and future achievements to them.

I could not have asked for a better advisor than Dr. Enrique Campos-Náñez. He gave me complete freedom in choosing my research topics and approaches, and supported me all the way to my graduation, even after he had to leave GW. He is a great mentor and friend. I am grateful for all his help and dedication during these years of study.

I also owe my smooth graduation to Prof. Johan René van Dorp. When Dr. Campos-Náñez had to leave GW, Prof. van Dorp kindly offered to take the responsibilities as

my academic advisor and had been very supportive in helping me fulfill the degree requirements. I very much appreciate his time and effort to help me improve my dissertation.

I appreciate the support from the committee members. I thank Prof. Hernan Abeledo for helping me find the critical theoretical support to my method. I also thank Prof. Miguel Lejeune for the knowledge of stochastic optimization.

I thank Profs. Ji Zhu, Kerby Shedden and Divakar Viswanath from the University of Michigan for their teaching of statistics and mathematics, which are essential knowledge for the development of the new method in this dissertation. I also appreciate the discussion and help from my colleagues and friends Jian Guo, Tiago Filomena, Phillip Schrader, Yizao Wang, Yunpeng Zhao, Yan Shi, Xiyuan Li, Xing Hong and Yingzhuo Dai. I would also like to thank my friends Fanbo He and Liguo Li, from whom I learnt about the electric vehicles and the battery exchange stations.

I am grateful for the financial support that I received for my Ph.D. study from the department of Engineering Management and Systems Engineering.

Abstract

Adaptive Convex Enveloping for Multivariate Convex Dynamic Programming

We propose a new method, called Adaptive Convex Enveloping, for solving convex stochastic dynamic programs (DP) with multidimensional continuous state and decision variables. The new method approximates the value functions with supporting hyperplanes, which is convexity-preserving. As a result, the method is able to handle large numbers of decision variables and constraints with the speed and reliability of convex optimizations. To construct a supporting hyperplane approximation, we use a recursive partitioning algorithm, which "learns" the shape of the underlying function and adds supporting hyperplanes adaptively. The approximation so constructed uses supporting hyperplanes economically, and guarantees that the error is less than a tolerance for all points in a continuous state space.

Error bounds of the solution can be developed using convexity. Both the optimal value function and the value function of the obtained policy are bounded below by the solution of Adaptive Convex Enveloping, and bounded above by the solution plus the total tolerance. These bounds are directly available without having to do any simulation, and they are simple numbers, rather than analysis terms, thus they

provide confidence. The performance of the obtained policy can also be simulated and used as a tighter upper bound of the optimal value function.

An exciting feature of Adaptive Convex Enveloping is that it is very much standardized. It saves the user the time spent on trial and error for designing the approximation functions and tuning the parameters. For the optimization, our method calls existing mathematical optimization libraries. So to solve a DP, the user only needs to write models for the libraries to read, which is a similar experience as doing mathematical optimizations. In all, the method is convenient for modeling and solving real world problems.

We apply Adaptive Convex Enveloping to management of battery exchange stations to find the optimal policy for charging a large number of electric vehicle batteries. We solve two variants of the problem: one with known but changing electricity prices and stochastic customer demands, the other with an additional option for the station to discharge the batteries in the inventory and sell electricity back to the grid.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 The Motivating Problem

We propose a new method for solving dynamic programs (DP). The research was motivated by a problem from management of battery exchange stations, and we introduce it first to give the readers an idea of what type of problems we are trying to solve.

A battery exchange station is an infrastructure that serves electric vehicles. Since the power of electric vehicles comes from the batteries, they need to be charged after use. Charging a battery with capacity around 20 kWhs usually takes 2-3 hours[1]. As a result, instead of having cars occupying precious parking spaces for hours to charge their batteries, it would be better for densely populated metropolitan areas to use centralized charging facilities, e.g., the battery exchange stations. At a battery

[1]There are fast charging technologies, which could charge an empty battery to 80% in 20 minutes. However, fast charging reduces the total lifetime of the batteries, which are very expensive. Thus, we opt not to use fast charging, and consider only the regular charging speed.

exchange station, customers come to replace the batteries of their cars with full ones, which is done by a machine usually in less than 2 minutes. The customers can then drive away, and the battery exchange station needs to charge the used batteries to serve new customers.

The cost of charging the batteries is associated with the electricity price, which is not the same during a day for large buyers. In regulated markets, such as in China, there can be multiple prices per day; In deregulated markets, such as in the U.S., the price is random. So when the price is high, if the station charges too many batteries, their cost will be high; On the other hand, if they charge too few, they may not have enough full batteries when customers come. Therefore a question for battery exchange station management is at each time, how many batteries they should charge to maximize the daily profit, while maintaining a certain level of service rate.

This problem can be formulated as a dynamic program. Suppose batteries are calibrated so that a full battery has M bars (levels) of energy, and assume equal charging times per battery level[2]. Let the stage interval equal to the time for charging a battery's energy level up by 1 level, so if we charge a level m ($m < M$) battery, it will become a level $m+1$ battery at the next stage. Denote the number of level m batteries that the station has at stage t by $x_{t,m}$, $0 \leq m \leq M$, and the number of level m batteries that it charges at stage t by $u_{t,m}$, $0 \leq m \leq M-1$. Let $w_{t,m}$ be the number of customers who come during time t and time $t+1$ to replace a level m battery, $0 \leq m \leq M-1$, and assume that they pay R dollars for each bar of energy (so a customer who replaces an empty battery pays $M \cdot R$ dollars). Since the station may not have enough full batteries, some customers may be unserved. Denote the number of served customers with $\bar{w}_{t,m}$, and we assume that those unsatisfied demands are lost and are not backlogged. In order to maintain a level of service rate, we impose a $Pnlt$

[2]This assumption is reasonable since we could always rescale the levels

dollars penalty to the objective function for each customer lost. For now, assume non-random electricity price, and let c_t be the cost of charging a battery up by 1 bar between time t and time $t+1$. With these notations, we write the Bellman's equation of this battery charging problem as follows:

$$
\begin{aligned}
J_t(x_t) = & \max_{u_t, \bar{w}_t^k} \{\textstyle\sum_{k=1}^{K} \mathbb{P}(w_t^k)[Revenue_t^k - Cost_t - Penalty_t^k + J_{t+1}(x_{t+1}^k)]\} \\
\text{s.t. } & Revenue_t^k = R\textstyle\sum_{m=0}^{M-1}(M-m)\bar{w}_{t,m}^k, \text{ for all } k, \\
& Cost_t = c_t \textstyle\sum_{m=0}^{M-1} u_{t,m}, \\
& Penalty_t^k = Pnlt(\textstyle\sum_{m=1}^{M-1} w_{t,m}^k - \sum_{m=1}^{M-1} \bar{w}_{t,m}^k), \text{ for all } k, \\
& 0 \le \bar{w}_{t,m}^k \le w_{t,m}^k, \text{ for all } k, 0 \le m \le M-1, \\
& \textstyle\sum_{m=1}^{M-1} \bar{w}_{t,m}^k \le x_{t,M}, \text{ for all } k, \\
& 0 \le u_{t,m} \le x_{t,m}, \ 0 \le m \le M-1, \\
& x_{t+1,M}^k = x_{t,M} - \textstyle\sum_{m=1}^{M-1} \bar{w}_{t,M}^k + u_{t,M-1}, \text{ for all } k, \\
& x_{t+1,m}^k = x_{t,m} - u_{t,m} + u_{t,m-1} + \bar{w}_{t,m}^k, \text{ for all } k, 1 \le m \le M-1, \\
& x_{t+1,0}^k = x_{t,0} - u_{t,0} + \bar{w}_{t,0}^k, \text{ for all } k.
\end{aligned}
\tag{1.1}
$$

where the superscript k denotes scenario k, $1 \le k \le K$. The first 3 constraints of (1.1) are definitions of terms in the objective function; the next 3 constraints define the feasible set for the decisions variables; and the last 3 constraints are the system transitions. The value function $J_t(x_t)$ equals the expected total profit from time t to the end time under the optimal policy, given the battery inventory $x_t = (x_{t,0}, \ldots, x_{t,M})^T$ at time t. The reader may have noticed that the formulation of (1.1) allows the station to give service priority to more profitable customers, while in reality it is likely that the station runs on a first-come-first-served basis. The first-come-first-served basis would require a random number of random constraints in the formulation, which we omit to clarify exposition of the general algorithm. Finally, considering that the number of batteries is not small, we allow all variables to be

continuous, and in practice one could round the solution to integers.

Formulating the battery charging problem as a DP is easy, but solving it is hard. In fact, there is not an existing method that guarantees to solve it well. Problem (1.1) has multidimensional continuous state and decision variables, nontrivial state-dependent constraints and could not be decomposed. We are curious about how to solve this type of DP problems, which is the motivation of this work.

1.2 Contribution to the Literature

We developed a method to solve finite horizon stochastic convex dynamic programs with multidimensional continuous state and decision variables. In addition, we focus on problems with nontrivial constraints and strongly coupled state variables. For this type of problems, the solution of our method achieves the level of optimality that existing methods normally do not[3]. It also provides confidence in the solution with error bounds that are clear numbers, compared to bounds given by many existing methods that are analysis terms whose exact values are unknown. Furthermore, our method is standardized, so the user does not have to spend time on trying various approximation functions and tuning parameters, which typically takes large amount of time.

This work also serves as an early study of the optimal policy for charging a large number of batteries at battery exchange stations. Although the exact optimal policy changes with the settings, patterns of the optimal policy could be extracted to be used as a guidance for designing charging rules or for solving larger scale problems.

[3]Many existing methods theoretically could give optimal solutions — they do when they happen to be using the right approximation functions and the right setting of parameters, and assuming all the optimizations involved can be solved. However, in practice, such coincidence does not easily happen. See Section 2.2 for details.

In addition, solving the problem with various total numbers of batteries and comparing with the long term or average daily cost is a method to determine the optimal number of batteries for a station to prepare, which is another problem that the station managers care about.

1.3 Limitations

We limit our focus to finite horizon convex DPs with continuous decision variables. Infinite horizon DPs, or DPs with integer decision variables, are not explored. Although at the end of this dissertation we show that our method is possible to solve certain DPs with nonconvex value functions, we do not intend to solve general nonconvex DPs in this research, because the technique for optimizing nonconvex mathematical programs is not reliable yet.

While our method allows large numbers of decision variables, it does not allow many state variables. The approximation method that we use is a local method, and it suffers from the statistics version of the curse of dimensionality[4] as all local approximation methods do. Choosing between local methods and global methods is a trade-off. Here we give up scalability for precision and standardization.

The parameters in the numerical examples are set to make the effect of dynamic optimization easy to see, rather than to reflect the reality. In fact, there is no reality yet. This dissertation prepares a method to be used in the future. The first batch of battery exchange stations are being built at the time of the writing of this dissertation, which means that meaningful data of a mature market will only be available in a number of years in the future. Thus some of the parameters, e.g. the distribution of

[4]Explained in Section 2.1.

customer arrivals, are not necessarily true.

Finally, the goal of this work is to develop a general purpose algorithm for dynamic programming, while solving the optimal charging problem is just one illustrative application. Therefore we do not discuss or compare our method with any inventory or production control method from management science.

1.4 Organization of the Chapters

In Chapter 2 we review the literature of DP that is relevant to the type of problems that we are interested in. In Section 2.1 we review the basics of DP, and discuss the curse of dimensionality, especially the statistics version of the curse, which is not well addressed in the DP community. In Section 2.2 we review relevant existing algorithms. We put extra emphasis on simulation-based algorithms, as they are the only class of algorithms that have the possibility to solve our target problems, and we explain why they normally would not work very well.

In Chapter 3 we introduce our new method, the Adaptive Convex Enveloping (ACE). Section 3.1 discusses convexity and introduces how to efficiently add a single supporting hyperplane. Section 3.2 discusses error bounds of the supporting hyperplane approximation, and shows how to find a point where the potential approximation error is the largest. Section 3.3 introduces the recursive partitioning algorithm used by ACE, and illustrates with the simple inventory control problem. Section 3.4 discusses strengths, limitations and other interesting aspects of ACE.

Chapter 4 provides numerical demonstration of the algorithm, where we apply it to the optimal charging problem. We solve two variants of the problem. In Section

4.1 we solve the standard problem, and in Section 4.2 we solve the problem with an additional option to discharge the batteries and sell the electricity back to the grid.

Chapter 5 concludes the dissertation and discusses future work.

Data structures and efficient computation are discussed in the Appendix.

Chapter 2

Literature Review

2.1 Review of Dynamic Programming

Dynamic programming (Bellman [1957]) is a category of optimization problems of sequential decision making on a dynamic system that evolves over (discrete[1]) stages. At each stage t, $t = 1, \ldots, T$, a system state $x_t \in X_t$ is observed, and we need to make a corresponding decision $u_t \in U_t(x_t)$ that triggers a cost $c_t(x_t, u_t, w_t)$, where w_t is a random variable that is out of our control. The system will then enter the next stage with a new state:

$$x_{t+1}(x_t, u_t, w_t),\ t = 1, \ldots, T-1,$$

which is called the system transition function. Since w_t is a random variable, the system state of the next stage is also random.

The aim of DP is to find a policy $\pi = \{\mu_1, \mu_2, \ldots, \mu_{T-1}\}$, such that the expected total

[1]We do not consider continuous time DP in this review.

cost

$$E\left\{\sum_{t=1}^{T-1} c_t(x_t, \mu_t(x_t), w_t) + c_T(x_T, w_T)\right\} \tag{2.1}$$

is minimized. Recall that stochastic programming (SP) also solves sequential decision making problems that look similar to (2.1). The difference between SP and DP is that in SP we look for an optimal decision for a fixed system state, while in DP we look for a policy that tells what our best decision is for every possible state. In other words, we do not need to solve the DP again when the state changes, which is why DP is called "dynamic".

In the above formulation, we have a finite planning horizon T, and the corresponding problem is called finite horizon DP. We can also have infinite horizon DP, where $T = \infty$. In an infinite horizon problem, the stages are usually considered homogeneous[2], i.e., same states, decisions, cost functions and transitions for all the stages, and the problems are often considered as a Markov decision process (MDP) (see Puterman [1994]). The objective to optimize is different from the one in a finite horizon DP, since when $T = \infty$, (2.1) is likely to be unbounded. One common way to solve infinite horizon DPs is to minimize the total cost discounted in time value. Let $r \in (0, 1)$ be a discount factor, and we want to find a policy $\mu(x)$ that minimizes

$$E\left\{\sum_{t=1}^{\infty} r^{t-1} c(x_t, \mu(x_t), w_t)\right\}.$$

There are other alternative objectives when solving infinite horizon DPs, such as minimizing the expected average cost

$$E\left\{\lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} c(x_t, u(x_t), w_t)\right\}. \tag{2.2}$$

[2]Although theoretically non-homogeneous problems can be converted to homogeneous ones by augmenting the state space, it is generally impracticable to do so, due to the curse of dimensionality that is explained later.

Obtaining a dynamic policy instead of a point for a fixed situation is a good idea, but it is also very difficult, as the output of $\mu_t(x_t)$ is usually multidimensional and subject to constraints. Fortunately, in DP, we have the following basic observation.

Principle of Optimality

Consider the finite horizon problem. Let $\{\mu_1, \mu_2, \ldots, \mu_{T-1}\}$ be the optimal policy that minimizes the expected total cost

$$E\left\{\sum_{t=1}^{T-1} c_t(x_t, \mu_t(x_t), w_t) + c_T(x_T, w_T)\right\}.$$

If we observe at time $t_0 < T$ that the system is in state x, then the truncated policy $\{\mu_{t_0}, \mu_{t_0+1}, \ldots, \mu_{T-1}\}$ is still optimal for the remaining stages and minimizes

$$E\left\{\sum_{t=t_0}^{T-1} c_t(x_t, u_t(x_t), w_t) + c_T(x_T, w_T) \mid x_{t_0} = x\right\}.$$

Based on this fact, we define recursively a quantity $J_t(x_t)$, called the value function, for every state:

$$J_t(x_t) = \begin{cases} E\{c_T(x_T, w_T)\}, & t = T; \\ \min\limits_{u_t \in U_t(x_t)} E\{c_t(x_t, u_t, w_t) + J_{t+1}(x_{t+1}(x_t, u_t, w_t))\}, & t < T. \end{cases} \tag{2.3}$$

This recursive definition is called the Bellman's Equation. When c_t is a cost, the practical meaning of $J_t(x_t)$ is the expected total cost from stage t to the end using the optimal policy, given the system state at stage t is x_t. Therefore another name of $J_t(x_t)$ is the cost-to-go function.

The value functions contain all the information needed to determine the optimal

policy, as

$$\mu_t(x_t) = \arg \min_{u_t \in U_t(x_t)} E\{c_t(x_t, u_t, w_t) + J_{t+1}(x_{t+1}(x_t, u_t, w_t))\}.$$

Therefore, finding the value functions is sufficient to solve the dynamic program. Meanwhile, the output of the value functions are scalars, so they are much simpler than the policies, which have multidimensional outputs. As a result, finding the value functions is the central topic in most of the DP algorithms.

The Exact DP Algorithm

Here we introduce the classical DP algorithm, also called the exact DP algorithm, for solving dynamic programs with finite states, finite actions and a random variable with finite values. The most remarkable feature of the algorithm is that it solves backward in stage, which is the same as how the value functions are defined.

Algorithm 2.1 The Exact DP Algorithm

```
For t = T-1,...,1
For every x_t in X_t
    J_t(x_t) = ∞;
        For every u_t in U_t(x_t)
            J_t(x_t) =
            min (J_t(x_t), Σ_k P(w_t^k) (c_t(x_t, u_t, w_t^k) + J_{t+1}(x_{t+1}(x_t, u_t, w_t^k))));
        End
    End
End
```

The above computes the optimal value for every state, and stores them in a lookup table. The optimal decision for every state can also be stored in a lookup table for later use.

The Curse of Dimensionality

The exact DP algorithm is very efficient, but it only works for very small scale problems. Let $|\cdot|$ be the cardinality of a set, then on each stage, the DP algorithm loops over approximately $|X_t| \times |U_t| \times |W_t|$ combinations, and takes $|X_t|$ space for storing the value functions. It is easy to see that when the number of state, action and random variables increase, making $|X_t|$, $|U_t|$ and $|W_t|$ grow exponentially, the algorithm becomes intractable very soon.

This problem is known as "the curse of dimensionality", a term coined by Bellman. Powell [2007] summarizes that there are three curses of dimensionality, coming from the number of state variables, the number of action variables and the number of random variables, respectively, referring to the $|X_t| \times |U_t| \times |W_t|$ combinations to loop at each stage. As a result, approximation becomes almost inevitable, and the approximate solution methods are often referred to as approximate dynamic programming (ADP).

An ADP method typically treats x_t as a continuous (multi-dimensional) variable, and approximates $J_t(x_t)$ using a continuous function that depends on a parameter vector θ. So, at least the storage problem is solved, because we only need to store the parameter θ. However, as the lookup table being replaced by a multi-dimensional function approximation, we are now facing the "geometrical" curse of dimensionality.

The curse of dimensionality, originally a term used in DP to refer to the number of $|X_t| \times |U_t| \times |W_t|$ discrete combinations to loop, is also borrowed by statisticians to describe a phenomenon of the high-dimensional geometry. In a high-dimensional space, points are "far" from each other. For example, if 500 points are uniformly distributed in a 10-dimensional unit ball, the mean distance from the origin to the nearest point

is approximately 0.52, which is more than halfway to the boundary. This means that when we want to estimate the value at a new point, all the known points are likely to be far away and provide little reference (see Hastie et al. [2009] for more examples and implications of this phenomenon). Therefore fitting high-dimensional functions are intrinsically difficult. Statisticians are very cautious when fitting multidimensional functions, and rely heavily on large sample size and techniques such as cross-validation to tune parameters and reduce overfitting[3]. Unfortunately this tough but important work is not enough addressed in the literature of ADP.

2.2 Review of Existing Methods

We need a method to solve the optimal charging problem (1.1). The problem has multidimensional continuous state and decision variables, so the exact DP algorithm is not directly applicable, due to the "curse of dimensionality". In this section, we look at other existing methods and see if they could solve this type of problems.

The Linear Programming Approach to Dynamic Programming

A DP could be rewritten as a linear program (LP), which was first proposed by Manne [1960] and de Ghellinck [1960], and improved by Denardo [1970], Derman [1970] and Hordijk and Kallenberg [1979]. Consider the infinite horizon problem with discount. Let the stages be homogeneous, so we remove the subscript t, and denote

[3]Overfitting means the approximation only fits the samples well but has poor performance when extended to unknown points. Flexible approximation functions and large noise in data are likely to cause overfitting.

the transition function with f. The Bellman's equation is

$$J(x) = \min_{u \in U(x)} E[c(x,u,w) + rJ(f(x,u,w))],$$

where $r \in (0,1)$ is the discount rate. Let the possible numbers of state x and decision u be finite. Also assume finite values for w. Since there are finite states, the value function $J(x)$ can be seen as a vector. It can be shown that $J(x)$ is the unique solution to the LP

$$\begin{aligned} &\max_{J(x)} \sum_x a(x)J(x) \\ &\text{s.t. } \sum_w P(w)[c(x,u,w) + rJ(f(x,u,w))] \geq J(x), \text{ for all } x \text{ and } u, \end{aligned} \tag{2.4}$$

where $\{a(x)\}_x$ are arbitrary positive numbers. One could see that the formulation of (2.4) suffers from the same curse of dimensionality as the exact DP algorithm does, as the number of decision variables for (2.4) is $|X|$ and number of constraints is $|X| \times |U|$. Approximation was introduced by Schweitzer and Seidmann [1985], called approximate linear programming (ALP), which approximates $J(x)$ by $\hat{J}(x) = \Phi(x)^T\beta$, where $\Phi(x) = (\phi_1(x), \ldots, \phi_m(x))^T$ is a collection of feature functions (or basis functions in some literature) chosen by the user, and β a vector of parameters to be determined. So (2.4) now becomes

$$\begin{aligned} &\max_{\beta} \sum_x a(x)\Phi(x)^T\beta \\ &\text{s.t. } \sum_w P(w)[c(x,u,w) + r\Phi(f(x,u,w))^T\beta] \geq \Phi(x)^T\beta, \text{ for all } x \text{ and } u. \end{aligned} \tag{2.5}$$

The benefit of the approximation is that (2.5) now has $|\beta|$ variables, which is usually far smaller than $|X|$. However, several additional problems are introduced as well with the approximation. First, the choice of the feature functions directly affects the

approximation quality. If $J(x)$ is far from the column space spanned by $\Phi(x)$, the approximation is going to be poor. However, $J(x)$ is unknown, so we do not know what features will cover $J(x)$. As a result, the selection of the feature functions is usually based on experience, and typically involves significant trial and error. Second, originally $\{a(x)\}$ could be any arbitrary positive values, but now the choice of $\{a(x)\}$ affects the approximation quality. Third, while (2.4) is guaranteed to be feasible, with $J(x)$ as the unique solution, the new LP (2.5) may not have a feasible solution. As an improvement, special feature functions are included in De Farias and Van Roy [2003] to prevent infeasibility. Finally, although (2.5) reduces the number of variables to $|\beta|$, the number of constraints is still $|X| \times |U|$, which could make (2.5) intractable. Techniques such as sampling, heuristics and constraint generation exist to alleviate this problem (see De Farias and Van Roy [2004], Trick and Zin [1993], Grötschel and Holland [1991]).

Overall, the LP or ALP approach to DP is for infinite horizon problems with homogeneous stages, and especially, they require discrete and finite (and in small amount) states and decisions. Therefore, this approach is not for the problems that we are solving, which have multidimensional continuous variables.

Temporal Difference Learning

Technically, temporal difference learning (TD) is not a dynamic programming method, but a technique from the field of reinforcement learning. The direct goal of TD is to learn the value (expected reward, cost, etc.) of each state through a sequence of observations under a fixed policy, and as one could see, there is no optimization involved. However, in practice, TD is usually combined with other methods to improve the policy, with successful early applications in artificial intelligence such as

playing checker (Samuel [1959]), bucket brigade (Holland [1986]) and backgammon (Tesauro [1995]). We find it necessary to introduce TD here as it is heavily used in simulation-based DP algorithms.

Since the policy is fixed, we omit u from the cost function. For now, assume for simplicity that $c(x, w) = 0$, and we only receive a final cost z when the process reaches an absorbing state. Let $\hat{J}(x) = \Phi(x)^T\beta$ be a model that predicts the final cost z using the current state x. If we observe the sequence $x_1, x_2, \ldots, x_T$, with the current prediction model, we would predict z as $\Phi(x_1)^T\beta, \ldots, \Phi(x_T)^T\beta$ at each stage. When the value of z is finally revealed, we would like to make adjustment to β using gradient adaptation:

$$\beta \leftarrow \beta + \sum_{t=1}^{T} \Delta\beta_t, \tag{2.6}$$

where

$$\begin{aligned}\Delta\beta_t &= \alpha(z - \Phi(x_t)^T\beta)\frac{\partial(\Phi(x_t)^T\beta)}{\partial\beta} \\ &= \alpha(z - \Phi(x_t)^T\beta)\Phi(x_t),\end{aligned}$$

and where α is a learning step. However, since we don't observe z until the process ends, we need to store the data of $\Phi(x_t)$ for all $t = 1, \ldots, T$, where T is random and can be very large. This could be a critical issue, especially during the time when even the large machines had tiny memories compared to today's cell phones. TD solves this problem by using a smart transformation. Observe that

$$z - \Phi(x_t)^T\beta = \sum_{k=t}^{T} \left[\Phi(x_{k+1})^T\beta - \Phi(x_k)^T\beta\right], \qquad \text{where } \Phi(x_{T+1})^T\beta \triangleq z.$$

Using this, we can rewrite 2.6 as

$$\begin{aligned}
\beta &\leftarrow \beta + \sum_{t=1}^{T} \alpha(z - \Phi(x_t)^T\beta)\Phi(x_t) \\
&= \beta + \sum_{t=1}^{T} \alpha \sum_{k=t}^{T} \left[\Phi(x_{k+1})^T\beta - \Phi(x_k)^T\beta\right] \Phi(x_t) \\
&= \beta + \sum_{k=1}^{T} \alpha \sum_{t=1}^{k} \left[\Phi(x_{k+1})^T\beta - \Phi(x_k)^T\beta\right] \Phi(x_t) \\
&= \beta + \sum_{t=1}^{T} \alpha \left[\Phi(x_{t+1})^T\beta - \Phi(x_t)^T\beta\right] \sum_{k=1}^{t} \Phi(x_k).
\end{aligned}$$

In other words, we can let

$$\Delta\beta_t = \alpha \left[\Phi(x_{t+1})^T\beta - \Phi(x_t)^T\beta\right] \sum_{k=1}^{t} \Phi(x_k),$$

and achieve the same update by only storing one vector equal to $\sum_{k=1}^{t} \Phi(x_k)$ at time t.

This version of TD method was later extended by Sutton [1988] to TD(λ). TD(λ) uses a more flexible form for the update by letting

$$\Delta\beta_t = \alpha \left[\Phi(x_{t+1})^T\beta - \Phi(x_t)^T\beta\right] \sum_{k=1}^{t} \lambda^{t-k}\Phi(x_k).$$

The parameter λ is between 0 and 1, and serves as a memory effect. For instance, the original TD corresponds to TD($\lambda = 1$). On the other extreme, if we let $\lambda = 0$, then

$$\Delta\beta_t = \alpha \left[\Phi(x_{t+1})^T\beta - \Phi(x_t)^T\beta\right] \Phi(x_t),$$

so the adjustment is made by only comparing to the estimate of the next stage, which is very similar to the idea behind the Bellman's equation. Of course, TD can also be

extended to problems with cumulative costs, and with discount, by letting

$$\Delta\beta_t = \alpha \left[c_{t+1} + r\Phi(x_{t+1})^T\beta - \Phi(x_t)^T\beta\right] \sum_{k=1}^{t} \lambda^{t-k}\Phi(x_k).$$

Sutton [1988] also shows convergence of TD(λ), but under rather strong assumptions. In practice, to help TD converge, the user usually let the learning step α shrink to zero at a moderate speed. For instance, if α_i is the learning step of the ith round of update, a commonly used criterion is

$$\sum_{i=1}^{\infty} \alpha_i = \infty, \text{ and } \sum_{i=1}^{\infty} \alpha_i^2 < \infty.$$

For example, $\alpha_i = \frac{m}{i}$, where m is a constant. A drawback of this practice is that the learning result depends on the order by which the training samples are presented to the model. In general, the convergence of TD is not guaranteed, and it often fails to converge in practice.

The standard TD updates β by gradient adaptation. Alternatively, β can be updated by statistical methods, such as the least squares. The method introduced by Bradtke and Barto [1996] updates β with least squares for TD(0), and Boyan [2002] extends the method to general TD(λ). These least squares based TD methods are generally referred to as LSTD. Like TD, the LSTD algorithm also does the computation in a cumulative fashion. If we observe $x_1, \ldots, x_T$, with costs $c_1, \ldots, c_T$, then LSTD computes a matrix A and a vector b cumulatively:

Algorithm 2.2 Least Squares Temporal Difference Learning

```
Set A = 0, b = 0;
Choose x_1 ∈ X;
z = Φ(x_1);
For t = 1 ... T

    A = A + z(Φ(x_t) − Φ(x_{t+1}))';
    b = b + z · c_t;
    z = λz + Φ(x_{t+1});

End
```

When the above routine finishes, the parameter β is updated as $\beta = A^{-1}b$ by solving $A\beta = b$. Solving the linear system typically takes $O(n^3)$ operations, where n is the length of θ. However, a recursive method is provided both in Bradtke and Barto [1996] and Boyan [2002] to reduce the computation to $O(n^2)$.

There is no consensus on whether TD or LSTD performs better. LSTD consumes more memory and computation than TD, but is not affected by the learning order. On the quality of the learning result, Boyan [2002] claims that LSTD uses samples more efficiently, while Sutton [1992] argues that with large amount of features and abundant training samples, TD may have better real time performance.

It is worth mentioning that although earlier applications use TD and LSTD in problems with finite states, they do have the capability to deal with multivariate continuous states. And as mentioned above, although by default the two algorithms are for learning the value of a fixed policy, they could be modified to learn and improve the policy at the same time. However, the convergence and consistency are not guaranteed.

Simulation-based Dynamic Programming

A popular class of the approximation algorithms are closely related to reinforcement learning: an algorithm of this type assumes a parametric form for the value function and/or for the optimal policy, and tries to improve the parameters based on the simulated performance. Some people use the term "approximate dynamic programming" or ADP exclusively for these algorithms, but since other people use the name for all non-exact DP algorithms, for clarity, we call them "simulation-based dynamic programming (SBDP)" in this dissertation.

The majority of SBDP algorithms, though through different forms, involve repeating the cycle of the following two steps:

$$\mu_t(x_t) := \arg\min_{u_t \in U_t(x_t)} E[c_t(x_t, u_t, w_t) + \hat{J}_{t+1}(x_{t+1}(x_t, u_t, w_t))], \\ t = T-1, \ldots, 1; \quad (2.7)$$

$$\hat{J}_t(x_t) := E[c_t(x_t, \mu_t(x_t), w_t) + \hat{J}_{t+1}(x_{t+1}(x_t, \mu_t(x_t), w_t))], \ t = T-1, \ldots, 1. \quad (2.8)$$

The step of (2.7) updates the policy $\{\mu_t\}_{t=1}^{T-1}$, using the current value function approximation $\{\hat{J}_t\}_{t=1}^{T-1}$. For problems with a finite and small number of states and decisions, the new policy can be computed as in (2.7), and stored exactly using look-up tables (see Bellman [1957]). For problems with continuous state variables and finite decisions, instead of approximating $J_t(x_t)$ directly, one could augment the input variable to a state-decision pair, and approximate

$$\hat{Q}_t(x_t, u_t) \approx E[c_t(x_t, u_t, w_t) + \hat{J}_{t+1}(x_{t+1}(x_t, u_t, w_t))],$$

so that

$$\mu_t(x_t) := \arg\min_{u_t} \hat{Q}_t(x_t, u_t)$$

can be chosen by enumeration, and $\hat{J}_t(x) := \hat{Q}_t(x_t, \mu_t(x_t))$. When both the state and the decision are continuous variables, giving a form to the policy becomes difficult: one could do a statistical regression on a sample of states to fit (2.7), but the output may not satisfy the feasibility constraint $\hat{\mu}_t(x_t) \in U_t(x_t)$, especially when the decision variable is multidimensional with constraints nontrivial; Alternatively, one could use a heuristic policy that guarantees $\mu_t(x_t) \in U_t(x_t)$, but it could be hard to argue why this heuristic is close to optimality. Whichever the approach, after the policy is updated, the algorithm then goes to step (2.8) and evaluates the value functions of the new policy by simulation. This could be done by regression or TD. Popular forms for the approximation include the linear form $\hat{J}_t(x_t) = \Phi_t(x_t)^T \beta_t$, neural-networks (see Bertsekas and Tsitsiklis [1996]) and kernel regressions (explained below). Once the value function evaluation is finished, the algorithm goes back to step (2.7) and repeats the cycle.

The above is the basic idea of SBDP, and several variants exist. We review representative SBDP algorithms and show how they work.

Value Iteration

Value Iteration algorithms do not have clear steps of (2.7) and (2.8). They merge the two steps as one (e.g., plug (2.7) into (2.8)), so they do not have an explicit policy, and appear to be only working on the value functions.

Q-learning, introduced by Watkins [1989], is a typical value iteration algorithm. It can be seen as the reverse of the exact DP algorithm, and can be used without knowing

the distribution of the random variable w. The Q-learning algorithm uses $Q(x, u)$ to record the value at state x for performing action u. Like the exact DP algorithm, it uses a lookup table, so Q-learning does not alleviate the curse of dimensionality; On the contrary, it is even more vulnerable to it, because now memory consumption for storing $Q(x, u)$ is $|X| \times |U|$. However, compared to TD that (by default) only learns the expected cost-to-go of a MDP with fixed policy, Q-learning tries to find the optimal policy as it approximates the optimal cost-to-go. The Q-learning algorithm is executed as follows:

Algorithm 2.3 Q-learning

Initialize $Q(x, u)$ for all possible pairs of x and u;
For $t = 1, 2, \ldots$

Choose an action u_t for the current state x_t;
observe the cost c_t and the next state x_{t+1};
$Q(x_t, u_t) = (1 - \alpha_t)Q(x_t, u_t) + \alpha_t(c_t + J(x_{t+1}))$,
where α_t is the learning step,
and $J(x_{t+1}) = \min_u Q(x_{t+1}, u)$;

End

Since Q-learning uses simulation-based stochastic approximation, the convergence and consistency could be an issue. A proof is given by Watkins and Dayan [1992], with the condition that the Q-value of every state and action pair is updated infinitely often, which is very strict, because we don't have control on the evolution of the Markov process.

Although Q-learning is less frequently used nowadays due to its vulnerability to the "curse of dimensionality", the way that it augments x to (x, u) is still popular among the later continuous approximations as a method to perform policy improvement.

There are many other ways to do value iteration. For example, Tsitsiklis and Van Roy [1996] introduce two feature-based value iteration algorithms, i.e., value function is

approximated as $\hat{J}(x) = \Phi(x)^T\beta$.

The first one partitions the state space $X = X_1 \cup \cdots \cup X_K$, $K = |\beta|$, where $X_i \cap X_j$ is empty if $i \neq j$, and the feature functions ϕ_k are indicator functions

$$\phi_k(x) = I_{X_k}(x) = \begin{cases} 1, & x \in X_k; \\ 0, & \text{otherwise.} \end{cases}$$

Thus feature functions are used as state aggregators, reducing the state space to a much smaller feature space, and the problem can be solved by using lookup tables.

The second algorithm uses generic feature functions, and assumes that we can find K points $x_1, \ldots, x_K$ that satisfy the condition:
(a) The vectors $\Phi(x_1), \ldots, \Phi(x_K)$ are linearly independent;
(b) There exists a value $r' \in [r, 1)$, where r is the discount factor, such that for any $x \in X$ there exist $\alpha_1(x), \ldots, \alpha_K(x) \in \mathbb{R}$ with

$$\sum_{k=1}^{K} |\alpha_k(x)| \leq 1,$$

and

$$\Phi(x) = \frac{r'}{r} \sum_{k=1}^{K} \alpha_k(x)\Phi(x_k).$$

The algorithm updates β by only looking at the fitness at the preselected representative states, and is shown to converge.

Policy Iteration

Unlike Value Iteration, the steps of (2.7) and (2.8) in Policy Iteration are clearly distinguishable. In some literature, the step of (2.7) that does the decision-making is

called the "actor", and the step of (2.8) that evaluates the performance is called the "critic". So another name of Policy Iteration is the Actor-critic algorithm.

One example of actor-critic policy iteration is given by Lagoudakis and Parr [2003], called least squares policy iteration (LSPI). Like Q-learning, LSPI augments the input of the feature functions to $\Phi(x, u)$ to facilitate policy update. It uses a TD process to approximate the Q-values of the current policy $Q(x, u) = \Phi(x, u)^T\beta$, and maintains and updates the policy using a lookup table (size of $|X|$). In the paper, the LSPI method is only applied to problems with finite states and finite decisions, but since $\Phi(x, u)^T\beta$ is a continuous approximation, LSPI does have the potential to be extended to multivariate continuous states and actions. The difficulty is the statistics version of the "curse of the dimensionality". The dimension of $\phi(x, u)$ can be much higher even than the dimension of (x, u), thus a reliable approximation of $Q(x, u)$ requires large of amount of sample and careful selection of the feature functions Φ.

The flexibility of feature-based approximation is determined by what features are added into Φ. Generally speaking, the more features we add , the greater potential the model has. However, adding features makes the dimension higher and the curse of dimensionality more severe. Due to this problem, the kernel trick has become popular for regression problems in statistics (see Bishop [2006]). Using dual representation, the feature-based approximation using least squares can be rewritten as

$$\Phi(x)'\beta = k(x)'K^{-1}y,$$

where $\{x_i\}_{i=1}^n$ are the training samples, $\{y_i\}_{i=1}^n$ the target values of the samples, $k(x_1, x_2) = \langle\Phi(x_1), \Phi(x_2)\rangle$ the inner product of two feature vectors, called the kernel function, $k(x) = (k(x, x_1), \ldots, k(x, x_n))'$, and K called the Gram matrix, where $K_{ij} = k(x_i, x_j)$. We see that the approximation can be expressed purely by the kernel

function. So the kernel trick is to design the kernel function directly, without having to worry about the underlying features. By the Mercer theorem (see Vapnik [1998]), if a kernel is positive definite, i.e., for any finite set of points $\{x_1, \ldots, x_n\}$, the Gram matrix K is positive definite, then there exists a Hilbert space H and a mapping $\Phi : X \to H$ such that

$$k(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle .$$

The underlying feature Φ of a kernel $k(\cdot, \cdot)$ can be infinite-dimensional, i.e., it has infinitely many feature functions. For instance, the feature Φ that correspond to the Gaussian radial basis kernel

$$k(x_i, x_j) = \exp\{-c \left\| x_i - x_j \right\|^2\},$$

where $c > 0$ is a tuning parameter, is infinite-dimensional. So the kernel trick allows us to achieve nonlinear regression in the lower-dimensional input space X, instead of a linear regression in the much higher-dimensional feature space $\Phi(X)$, increasing the flexibility of the model without increasing the dimension, and alleviating the "curse of dimensionality".

The kernel method surely has attracted attention in the DP community. Ormoneit and Sen [2002] discuss the use of kernel in SBDP and its convergence, but does not give a clear algorithm on the implementation. Xu et al. [2007] provide the KLSPI algorithm, which is an extension of LSPI with kernel. Xu et al. [2007] noticed that the evaluation of $Q(x, u)$ using the kernel form involves significant amount of computation (because we need to evaluate $k((x, u), (x_i, u_i))$ for every (x_i, u_i) in the sample, which is usually very large), thus they applied a sparsification method, called approximate linear dependency (ALD, Engel et al. [2002]). The idea of ALD is to select a representative subset of the sample that can approximate the whole sample set by

linear combinations. Suppose the original sample is $\{(x_1, u_1), \ldots, (x_n, u_n)\}$. Let ϵ be a threshold. The ALD subroutine is executed as follows:

Algorithm 2.4 Approximate Linear Dependency

```
Initialize dictionary D = {1};
For i = 2...n
    If min_c || Σ_{j∈D} c_j Φ(x_j, u_j) − Φ(x_i, u_i) ||^2 > ε
        D = D ∪ {i};
End
```

In this way the sample set is reduced to $\{(x_i, u_i)\}_{i \in D}$, which will be used for fitting $Q(x, u)$. Since

$$\begin{aligned}
&\left\| \sum_{j \in D} c_j \Phi(x_j, u_j) - \Phi(x_i, u_i) \right\|^2 \\
&= \sum_{k,j \in D} c_i c_j \left\langle \Phi(x_k, u_k), \Phi(x_j, u_j) \right\rangle - 2 \sum_{j \in D} c_j \left\langle \Phi(x_j, u_j), \Phi(x_i, u_i) \right\rangle + \left\langle \Phi(x_i, u_i), \Phi(x_i, u_i) \right\rangle \\
&= \sum_{k,j \in D} c_i c_j k((x_k, u_k), (x_j, u_j)) - 2 \sum_{j \in D} c_j k((x_j, u_j), (x_i, u_i)) + k((x_i, u_i), (x_i, u_i)),
\end{aligned}$$

the process can be carried out using the kernel method, too. This process imposes sparsity and greatly reduces the computational burden.

Both LSPI and KLSPI assume finite states and actions in the papers, and use enumeration to improve the policies. Konda and Tsitsiklis [2003] propose an algorithm that uses gradient descent for the policy improvement. They assume finite states and actions, and uses a stochastic policy that selects an action u randomly from a distribution that depends on the state x and a parameter θ, with $p(u \mid x, \theta) > 0$ for all u, x and θ. Their algorithm uses TD to estimate the Q-values $Q(x, u)$, based on which they estimate $\frac{\partial \lambda(\theta)}{\partial \theta}$, where $\lambda(\theta)$ is the average cost of the MDP using the decision generating distribution under the current θ.

Direct Gradient-based Policy Iteration

There are algorithms that do not explicitly approximate the value functions, but evaluate the gradient of the policy performance with regard to the parameters, so the policy can be improved by gradient descent.

In Marbach and Tsitsiklis [2001], an algorithm is given for finite state MDP with perfect information. It doesn't explicitly discuss the action, but assumes that the policy is determined by θ, and the analytical forms of the transition probability $p_{ij}(\theta)$ and the deterministic cost $c_i(\theta)$, $i, j \in X$, are known for all states, where either $p_{ij}(\theta) \neq 0$ or $\nabla p_{ij}(\theta) \neq 0$ for all i, j and θ. It needs to estimate the gradient of performance with regard to θ

$$\nabla\lambda(\theta) = \sum_i \pi_i(\theta) \left\{ \nabla c_i(\theta) + \sum_j \nabla p_{ij}(\theta) v_j(\theta) \right\},$$

where

$$v_i(\theta) = E\left[\sum_{k=0}^{T-1} (g_{i_k}(\theta) - \lambda(\theta)) \mid i_0 = i \right],$$

$\{\pi_i(\theta)\}$ is the steady-state probability such that $\sum_i \pi_i(\theta) = 1$ and

$$\sum_i \pi_i(\theta) p_{ij}(\theta) = \pi_j(\theta), \text{ for all } j,$$

and T is the time when the system visits a recurrent state i^*. To do so, it looks at a recurrent state i^* and a simulated sequence $i^* \to i_1 \to \cdots \to i_T \to i^*$, and approximates $\nabla\lambda(\theta)$ with

$$\sum_{t=1}^{T} \left(\tilde{v}_{i_t}(\theta, \tilde{\lambda}) L_{i_{t-1} i_t}(\theta) + \nabla c_{i_t}(\theta) \right),$$

where

$$L_{ij}(\theta) = \frac{\nabla p_{ij}(\theta)}{p_{ij}(\theta)},$$

and

$$\tilde{v}_{i_t}(\theta, \tilde{\lambda}) = \sum_{k=t}^{T} (c_{i_k}(\theta) - \tilde{\lambda}),$$

and $\tilde{\lambda} \approx \lambda(\theta)$ is assumed to be known. We see that there are many layers of approximations. Also, to generate one sample for the estimation, we must wait the system to start from i^* and to come back to it, which is very costly even for moderately big Markov chains. Moreover, this single sample is used for the estimation, which means that the variance must be very large. The authors recognize the large variance problem, and propose modifications to the algorithm in Marbach and Tsitsiklis [2000, 2003]. Campos-Náñez [2010] also addresses other downsides of above algorithm, namely the reliance on regenerative cycle observation, the need for parametric certainty and the need to observe system-wide information.

The problem with the same setting is solved by Baxter and Bartlett [1999] with less assumptions, using the following much more effective algorithm:

Algorithm 2.5 Direct Gradient-based Reinforcement Learning

Set $z_0 = 0$ and $\Delta_0 = 0$;
For $t = 0, 1, \ldots$

$z_{t+1} = r z_t + \frac{\nabla p_{x_t x_{t+1}}(\theta)}{p_{x_t x_{t+1}}(\theta)}$;
$\Delta_{t+1} = \Delta_t + \frac{1}{t+1}(c(x_{t+1}) z_{t+1} - \Delta_t)$;

End

where r is the discount factor. It is shown that $\Delta_t \to \pi' \nabla P(\theta) J_\theta$ with probability 1, and that $\pi' \nabla P(\theta) J_\theta$ converges to $\nabla \lambda(\theta)$ as $r \to 1$. Δ_t is therefore used as a biased estimator of $\nabla \lambda(\theta)$.

Baxter and Bartlett [1999] propose another algorithm for MDP with imperfect information. The decision is randomly drawn from two layers of distributions: first a parametric distribution generates a second parameter that determines the decision generating distribution, and then the decision is drawn from this second distribution.

None of the above gradient-based algorithms were given analysis on the convergence to optimality. In fact, the optimality is hard to guarantee, since even we were able to compute $\nabla\lambda(\theta)$ accurately, $\lambda(\theta)$ itself may not be convex, making it hard to find the optimal θ.

Limitations of SBDP

The SBDP algorithms have generated many successful applications, such as inventory control (Van Roy et al. [1997]), network pricing (Campos-Náñez [2003]), radiation therapy planning (Deng and Ferris [2008]) and fleet management (Simão et al. [2009]). The range of these applications shows the advantage of simulation — it can be used on almost everything. However, it also comes with disadvantages. Equations (2.7) and (2.8) describe the goals of the steps, but achieving them is not easy. Consider Problem (1.1). The value functions are multidimensional, and as we discussed in the "curse of dimensionality" section, these function are not easy to fit. For example, if we use feature functions for the approximation, it would normally project the problem into a much higher-dimensional feature space (and even much higher if the state-decision pair augmentation is used). Considering the large noise from the simulated samples, fitting this type of high-dimensional functions is especially vulnerable to overfitting. The overfitting issue is well studied in statistics, but not enough addressed in SBDP, and the common ways to control overfitting do not directly fit into these DP algorithms, either. Next, common ways for approximations, such as feature functions,

kernels or neural-networks architectures, generally do not give a function form suitable for optimization, which is needed in (2.7). Without reliable optimizations, the solution could be substantially suboptimal. And this brings the next concern — it is usually hard to measure the optimality of the solutions of these algorithms. Simulation could estimate the performance of the obtained policy, and could show that maybe it outperforms another one, but we still do not know how it compares to the optimal policy. Under rather strict mathematical assumptions, some algorithms could be given error bounds, which are normally analysis terms, such as the big-O notation, and relative to an unknown quantity, e.g., the smallest possible error that the chosen features could achieve for this particular approximation. Finally, convergence and consistency[4] are well-known issues of SBDP algorithms. In some cases, as we have seen above, convergence could be proved under strict assumptions, while in most of the applications, we do not know if an algorithm will converge, or where it will converge to. The consistency issue is also related to the exploration vs. exploitation dilemma. In SBDP algorithms, we make decisions under the estimate of the value functions of the current policy, which are not optimal, meaning that we could be better off if we guided the system to another state that is now wrongly estimated to have a high cost. The wrong estimate for the state value can only be corrected by visiting the state (or the area around the state, if it is continuous approximation) multiple times, which we don't, however, because the algorithm says we should go to the not-so-good state that the current estimate suggests as good. This behavior is called exploitation. SBDP algorithms do this to avoid exploring the vast state space, but will have the above bias. To counter the bias, an algorithm could try to explore the states that appear suboptimal, but the more exploration it does, the higher the computational burden, and the more severe the curse of dimensionality.

[4]We say the approximation $\hat{J}$ is consistent if we have $\left\|\hat{J} - J\right\| \to 0$, where $\|\cdot\|$ is a norm and J is the true function.

Relaxation of Weakly Coupled Dynamic Programs

When applying SBDP algorithms, the value function is sometimes assumed to be separable, i.e.,

$$J_t(x) = J_t^{(1)}(x_1) + J_t^{(2)}(x_2) + \cdots + J_t^{(I)}(x_I),$$

where the state variable $x = (x_1, x_2, \ldots, x_I)^T$ and $\{J_t^{(i)}\}$ are 1-dimensional parametric functions. This assumption is often made without justification, and is purely for simplicity, or tractability, especially when I is large. However, the value function could be shown to be separable in some cases under relaxation.

Hawkins [2003] and Adelman and Mersereau [2008] introduce a type of dual relaxation for DPs with weakly coupled constraints, and was applied by Topaloglu [2009] to solve the conventionally difficult flight leg booking problem. Consider the infinite horizon DP with the following assumptions on the state components:

• Decision component $u_i \in U_i(x_i)$, i.e., the constraints depend only on x_i.

• The state component of the next stage $x_i' = x_i'(x_i, u_i, w_i)$, and $\{w_i\}_i$ are independent.

• There is a partial cost $c_i(x_i, u_i, w_i)$, and the total cost

$$c(x, u, w) = \textstyle\sum_i c_i(x_i, u_i, w_i).$$

• There are N linking constraints of the form $\sum_i \mathbf{D}_i(x_i, u_i) \leq \mathbf{b}$, where $\mathbf{b} \in \mathbb{R}^N$ and $\mathbf{D}_i : \{(x_i, u_i) \mid u_i \in U_i(x_i)\} \to \mathbb{R}^N$. So the overall constraint on the decision is $u \in U(x)$ where

$$U(x) = \{(u_1, \ldots, u_I) \mid u_i \in U_i(x_i), \text{ for all } i; \textstyle\sum_{i=1}^I \mathbf{D}_i(x_i, u_i) \leq \mathbf{b}\}.$$

Note that x_i, u_i and w_i can all be multidimensional. The Bellman's equation is

$$\begin{aligned} J(x) = &\min_u E\left[\textstyle\sum_i c_i(x_i, u_i, w_i) + \beta J(x'(x, u, w))\right] \\ &\text{s.t. } u_i \in U_i(x_i), \text{ for all } i, \\ &\textstyle\sum_{i=1}^I \mathbf{D}_i(x_i, u_i) \le \mathbf{b}. \end{aligned}$$

Consider the Lagrangian relaxation. We remove the linking constraints, and add Lagrangian terms to the objective function:

$$\begin{aligned} J^\lambda(x) = &\min_u E\left[\textstyle\sum_i c_i(x_i, u_i, w_i) + \lambda^T \left[\mathbf{b} - \sum_{i=1}^I \mathbf{D}_i(x_i, u_i)\right] + \beta J^\lambda(x'(x, u, w))\right] \\ &\text{s.t. } u_i \in U_i(x_i), \text{ for all } i, \end{aligned}$$

where $\lambda \in \mathbb{R}_+^N$. It can be shown that

$$J^\lambda(x) = \frac{1}{1-\beta}\lambda^T \mathbf{b} + \sum_{i=1}^I J_i^\lambda(x_i),$$

where $J_i^\lambda(x_i)$ solves

$$\begin{aligned} J_i^\lambda(x_i) = &\min_{u_i} E\left[c_i(x_i, u_i, w_i) + \lambda^T \mathbf{D}_i(x_i, u_i) + \beta J_i^\lambda(x_i'(x, u, w))\right] \\ &\text{s.t. } u_i \in U_i(x_i). \end{aligned}$$

In this way, a high-dimensional DP is decomposed to lower-dimensional subproblems, making the master problem much easier to solve. However, as one could see, the system transitions in (1.1) show that our state variables are strongly coupled, so this decomposition technique does not apply to our problem.

2.3 Conclusion

The problems that we are interested in, such as (1.1), have multidimensional continuous state and decision variables, nontrivial state-dependent constraints and could not be decomposed. Among all the existing methods that are reviewed above, only SBDP algorithms could potentially solve this type of problems. However, as discussed above, SBDP algorithms have the overfitting problem that does not have an easy cure, and the approximation forms they use are not good for optimizations with multidimensional continuous decision variables. These drawbacks, together with the convergence and the consistency issues, make us unsure about the optimality of the solutions obtained by SBDP. To solve multidimensional continuous DPs with more confidence in the solution quality, we developed the Adaptive Convex Enveloping, which is introduced in the next chapter.

Chapter 3

Adaptive Convex Enveloping

In this chapter, we develop a new algorithm to solve convex dynamic programs. Godfrey and Powell [2001] introduce a method called CAVE for obtaining a convex approximation of the value function, but is restricted to 1-dimensional simple problems. Our algorithm is designed for multidimensional problems using a completely different approach, with also a more advanced and easier way to obtain the gradient.

3.1 The Supporting Hyperplane Approximation

Consider a finite horizon dynamic program. Let x_t, u_t and w_t be vectors of state variables, decision variables and random information variables at stage t, $t = 1, \ldots, T$, respectively. Let $J_t(x_t)$, $x_t \in X_t$, be the value function defined by the Bellman's equation:

$$\begin{aligned} J_t(x_t) = &\min_{u_t} E[c_t(x_t, u_t, w_t) + J_{t+1}(x_{t+1}(x_t, u_t, w_t))] \\ &\text{s.t.} \quad g_t(x_t, u_t) \leq 0, \end{aligned} \tag{3.1}$$

where $c_t(x_t, u_t, w_t)$ is the cost function, $g_t = (g_{1,t}, \ldots, g_{I_t,t})^T$ is a set of constraints on the action u_t that depend on the state x_t, and $x_{t+1}(x_t, u_t, w_t)$ is the system transition function.

The supporting hyperplane approximation requires that the value functions are convex. The following set of assumptions is one possible way to guarantee that.

Assumptions

- $J_T(x_T)$ is convex on X_T;
- The state domain X_t, $t = 1, \ldots, T$, is compact and convex;
- The cost function c_t with w_t fixed, and the constraints $g_{i,t}$, $i = 1, \ldots, I_t$, are jointly convex on $X_t \times \mathbb{R}^{dim(u_t)}$;
- If the constraints are not all linear, the set $\{(x_t, u_t) \mid g_t(x_t, u_t) \leq 0\}$ must have a relative interior point;
- The transition $x_{t+1}(x_t, u_t, w_t)$ is linear, i.e., $x_{t+1} = Ax_t + Bu_t + w_t$, where A and B are known matrices;
- The distribution of w_t is independent of x_t and u_t.

For the convenience of the reader, we reiterate some definitions here.

Definition. Let $S \subseteq \mathbb{R}^n$. The relative interior of a set S (denoted $ri(S)$) is defined as

$$ri(S) = \{x \in S \mid \exists \epsilon > 0, N_\epsilon(x) \cap aff(S) \subseteq S\},$$

where

$$N_\epsilon(x) = \{y \in \mathbb{R}^n \mid \|y - x\| < \epsilon\}$$

is a ball of radius ϵ centered at x, and $aff(S)$ is the affine hull of S, defined as

$$aff(S) = \{\sum_{i=1}^{k} \alpha_i x_i \mid x_i \in S, \alpha_i \in \mathbb{R}, i = 1, \ldots, k; \sum_{i=1}^{k} \alpha_i = 1; k \in \mathbb{N}\}.$$

Informally, the relative interior of a set is the interior relative to a lower-dimensional space. For instance, consider the set $S = \{(x, y) \mid y = 0; 0 \leq x \leq 1\}$. The interior of S is empty, while the relative interior of S is $ri(S) = \{(x, y) \mid y = 0; 0 < x < 1\}$.

Now we prove that $\{J_t(x_t)\}_{t=1}^{T}$ are convex. First we introduce two lemmas that have been proved by Fiacco and Kyparisis [1986].

Definition. A point-to-set map $R : \mathbb{R}^k \to \mathbb{R}^n$ is one that maps a point of $\mathbb{R}^k$ to a subset of $\mathbb{R}^n$. It is convex on a set $S \subseteq \mathbb{R}^k$ if $G(R) \cap (S \times \mathbb{R}^n)$ is convex, where $G(R) = \{(\epsilon, x) \mid x \in R(\epsilon)\}$.

Lemma 1. *Consider the parametric nonlinear optimization problem*

$$\begin{aligned} &\min_{x \in M} f(x, \epsilon) \\ &s.t.\ g_i(x, \epsilon) \geq 0, i = 1, \ldots, m, \\ &\quad h_j(x, \epsilon) = 0, j = 1, \ldots, p. \end{aligned}$$

where $\epsilon \in S$. If $\{g_i\}$ are jointly quasi-concave on $M \times S$, $\{h_j\}$ are jointly quasi-monotonic on $M \times S$, and M and S are convex sets, then R, given by

$$R(\epsilon) = \{x \in M \mid g_i(x, \epsilon) \geq 0, i = 1, \ldots, m, h_j(x, \epsilon) = 0, j = 1, \ldots, p\},$$

is convex on S.

Proof. See proposition 2.3 of Fiacco and Kyparisis [1986]. Note that their x corresponds to our decision variable u, and their ϵ correspond to our state variable x. □

Lemma 2. *Consider the optimization problem*

$$f^*(\epsilon) = \min f(x, \epsilon)$$
$$s.t.\ x \in R(\epsilon).$$

If f is jointly convex on the set $\{(x, \epsilon) \mid x \in R(\epsilon), \epsilon \in S\}$, R is convex on S, and S is a convex set, then the optimal value function $f^*(\epsilon)$ is convex on S.

Proof. See proposition 2.1 of Fiacco and Kyparisis [1986]. Note that R being convex is a stronger condition than R being essentially convex. □

Proposition 3. *Assume that **Assumptions** hold, then $J_t(x_t)$ is convex on X_t, $t = 1, \ldots, T$.*

Proof. The convexity can be shown by induction. If $J_{t+1}(x_{t+1})$ is convex, then for any realization of w_t, $c_t(x_t, u_t, w_t) + J_{t+1}(x_{t+1}(x_t, u_t, w_t))$ is jointly convex on $X_t \times \mathbb{R}^{dim(u_t)}$ by our assumptions on c_t and the transition, and so is $E[c_t(x_t, u_t, w_t) + J_{t+1}(x_{t+1}(x_t, u_t, w_t))]$. Since X_t is a convex set, and the constraints $g_{i,t}$, $i = 1, \ldots, I_t$, are jointly convex on $X_t \times \mathbb{R}^{dim(u_t)}$, the point-to-set map $R(x_t) = \{u_t \mid g_t(x_t, u_t) \leq 0\}$ is convex on X_t by Lemma 1. Then by Lemma 2, $J_t(x_t)$ is convex on X_t. □

Recall that a supporting hyperplane of a set S is a hyperplane that has at least one common point (intersection) with S, and has S entirely contained on one side of it. When talking about supporting hyperplanes of a function $f(x)$, S is the set $\{(x, y) \mid y = f(x)\}$. Since $J_t(x_t)$, $t = 1, \ldots, T$, are convex, they can be approximated (enveloped) by a set of supporting hyperplanes. Assume that $J_{t+1}(x_{t+1})$ has been investigated at $\{x_{t+1}^j\}_{j=1}^N$, where we know the function value $J_{t+1}(x_{t+1}^j)$, as well as a subgradient $\nabla J_{t+1}(x_{t+1}^j)$ (with abuse of the notation). Thus for any point x_{t+1}, we

know from convexity that

$$J_{t+1}(x_{t+1}) \geq J_{t+1}(x_{t+1}^j) + \nabla J_{t+1}(x_{t+1}^j)^T(x_{t+1} - x_{t+1}^j),\ j = 1, \ldots, N.$$

and $J_{t+1}(x_{t+1})$ is approximated as

$$\hat{J}_{t+1}(x_{t+1}) = \max_{j \in \{1,\ldots,N\}} \left\{ J_{t+1}(x_{t+1}^j) + \nabla J_{t+1}(x_{t+1}^j)^T(x_{t+1} - x_{t+1}^j) \right\}.$$

We now show how to efficiently obtain the function value and a subgradient of $J_t(\cdot)$ at x_t to construct a supporting hyperplane.

Rewrite (3.1) in sample form as

$$\begin{aligned} J_t(x_t) = & \min_{u_t} \textstyle\sum_{k=1}^K P(w_t^k)[c(x_t, u_t, w_t^k) + J_{t+1}(x_{t+1}(x_t, u_t, w_t^k))] \\ & \text{s.t.} \quad g_t(x_t, u_t) \leq 0. \end{aligned} \tag{3.2}$$

If w_t is discrete and the set of its possible values is small enough, we can select all of its possible values as $\{w_t^k\}_{k=1}^K$; Otherwise $\{w_t^k\}_{k=1}^K$ will have to be a sample from the distribution. The error introduced by this sampling will not be considered in this dissertation, and (3.2) replaces (3.1) as the definition of $J_t(x_t)$.

To add a supporting hyperplane at point x_t, we need to solve an optimization problem. Substitute J_{t+1} with its approximation $\hat{J}_{t+1}$. To compute the value of $J_t(x_t)$, we solve the following program:

$$\min_{u_t,\{J_{t+1}^k\}} \sum_{k=1}^{K} P(w_t^k)\{c(x_t,u_t,w_t^k) + \max_{j\in\{1,\ldots,N\}}[J_{t+1}(x_{t+1}^j) + \nabla J_{t+1}(x_{t+1}^j)^T(x_{t+1}(x_t,u_t,w_t^k) - x_{t+1}^j)]\}$$
$$\text{s.t.} \quad g_t(x_t,u_t) \le 0,$$

However, the max operation in the objective function is not convenient. To simplify, we replace it with a decision variable J_{t+1}^k, and add the supporting hyperplane constraints (linear):

$$\begin{aligned} &\min_{u_t,\{J_{t+1}^k\}} \sum_{k=1}^{K} P(w_t^k)[c(x_t,u_t,w_t^k) + J_{t+1}^k] \\ &\text{s.t.} \quad g_t(x_t,u_t) \le 0, \\ &\qquad J_{t+1}^k \ge J_{t+1}(x_{t+1}^j) + \nabla J_{t+1}(x_{t+1}^j)^T(x_{t+1}(x_t,u_t,w_t^k) - x_{t+1}^j), \text{ for all } j,k. \end{aligned} \tag{3.3}$$

The above formulation allows us to compute $J_t(x_t)$, but is not enough to obtain $\nabla J_t(x_t)$. Conventionally, obtaining the gradient is not easy, which is why the supporting hyperplane approximation has not been popular. If we were using simulation-based methods, we could estimate the gradient at a point by repeated sampling at its neighboring points. However, such an estimate is not only expensive, but also inaccurate. Fortunately, with convex programming, we can use sensitivity analysis. We replace x_t with a dummy decision variable s_t that is locked to be equal to x_t:

$$\begin{aligned} &\min_{s_t,u_t,\{J_{t+1}^k\}} \sum_{k=1}^{K} P(w_t^k)[c(s_t,u_t,w_t^k) + J_{t+1}^k] \\ &\quad \text{s.t.} \quad g_t(s_t,u_t) \le 0, \\ &\qquad J_{t+1}^k \ge J_{t+1}(x_{t+1}^j) + \nabla J_{t+1}(x_{t+1}^j)^T(x_{t+1}(s_t,u_t,w_t^k) - x_{t+1}^j), \text{ for all } j,k, \\ &\qquad s_t = x_t. \end{aligned} \tag{3.4}$$

Solving (3.4) not only gives us $J_t(x_t)$, which is the optimal objective value, but our assumptions also guarantee that there exists a Lagrange multiplier vector λ associated to the constraint $s_t = x_t$, and that $-\lambda$ (or λ, depending on how you write the Lagrangian function) is a subgradient of $J_t()$ at x_t. Thus we can obtain $\nabla J_t(x_t)$ accurately for free by looking at the Lagrange multiplier of the constraint $s_t = x_t$.

Here we prove that the Lagrange multiplier associated to the constraint $s_t = x_t$ exists and that it is a subgradient of J_t at x_t. We prove it in a more general, and the proof is based on Proposition 5.3.2 of Bertsekas [1999].

Consider the problem

$$\begin{aligned} &\min\ f(x) \\ &\text{s.t. } x \in X \\ &\quad Ex - d = 0 \end{aligned}$$

where X is a closed convex set. The objective function f is convex on X. Matrix E and vector d has appropriate dimensions (E has less rows than columns) and the rows of E are linearly independent. Let the optimal value of this problem be f^*. Assume that there exists $\bar{x} \in ri(X)$ and $E\bar{x} - d = 0$. Let

$$p(y) = \inf \{f(x) \mid x \in X, Ex - d = y\}.$$

So $p(0) = f^*$. And since X is closed, when $p(y)$ is finite, we can find the the point x, such that $x \in X$, $Ex - d = y$ and $p(y) = f(x)$.

Proposition 4. *There exists a Lagrange multiplier for the constraint $Ex - d = 0$, and the multiplier is a subgradient of $p(y)$ at $y = 0$.*

Proof. First assume that X has an nonempty interior. Consider the set

$$A = \{(y, w) \mid \text{there exists } x \in X \text{ such that } Ex - d = y, f(x) \leq w\}.$$

We show that A is a convex set. If $(y, w) \in A$ and $(\tilde{y}, \tilde{w}) \in A$, there exists $x \in X$ and $\tilde{x} \in X$, such that $Ex - d = y$, $f(x) \leq w$ and $E\tilde{x} - d = \tilde{y}$ and $f(\tilde{x}) \leq \tilde{w}$. For any $\alpha \in [0, 1]$, $\alpha x + (1 - \alpha)\tilde{x} \in X$, because X is convex. By using the convexity of f, we have

$$f(\alpha x + (1 - \alpha)\tilde{x}) \leq \alpha f(x) + (1 - \alpha) f(\tilde{x}) \leq \alpha w + (1 - \alpha)\tilde{w}.$$

We also have

$$E(\alpha x + (1 - \alpha)\tilde{x}) - d = \alpha y + (1 - \alpha)\tilde{y}.$$

So $(\alpha y + (1 - \alpha)\tilde{y}, \alpha w + (1 - \alpha)\tilde{w}) \in A$. This proves the convexity of A.

We next observe that $(0, f^*)$ is not an interior point of A, because otherwise, for some $\epsilon > 0$, the point $(0, f^* - \epsilon)$ would belong to A, which contradicts with the assumption that f^* is the optimal value when $y = 0$. Therefore, there is a supporting hyperplane passing through $(0, f^*)$. In particular, there exists a vector $(\lambda, \beta) \neq (0, 0)$ such that

$$\lambda^T (y - 0) + \beta(w - f^*) \geq 0, \ \forall (y, w) \in A. \tag{3.5}$$

This implies $\beta \geq 0$, because for any $(y, w) \in A$, we also have $(y, w + \gamma) \in A$ for all $\gamma > 0$.

We now show that $\beta > 0$. Indeed, if it was not so, we would have $\lambda^T y \geq 0$ for any $(y, w) \in A$. Note that for all $x \in X$, $(Ex - d, f(x)) \in A$. Thus $\lambda^T (Ex - d) \geq 0$, $\forall x \in X$. Since $E\bar{x} - d = 0$, we obtain

$$\lambda^T E(x - \bar{x}) \geq 0, \ \forall x \in X. \tag{3.6}$$

Recall that $\bar{x}$ is an interior point of X, thus (3.6) implies that $E^T\lambda = 0$. By assumption, the rows of E are linearly independent, implying $\lambda = 0$, which contradicts with $(\lambda, \beta) \neq (0, 0)$. Therefore we must have $\beta > 0$.

Divide both sides of (3.5) by β and let $\lambda^* = \frac{\lambda}{\beta}$, we have

$$f^* \leq \lambda^{*T} y + w, \ \forall (y, w) \in A. \tag{3.7}$$

Again, for all $x \in X$, $(Ex - d, f(x)) \in A$. Thus $f^* \leq f(x) + \lambda^{*T}(Ex - d)$, for all $x \in X$. Taking the infimum of $x \in X$, we have

$$f^* \leq \inf_{x \in X} \{f(x) + \lambda^{*T}(Ex - d)\} \triangleq q(\lambda^*).$$

On the other hand, by the weak duality theorem $q(\lambda^*) \leq f^*$, therefore $q(\lambda^*) = f^*$, λ^* is a Lagrange multiplier, and there is no duality gap.

Rearrange (3.7) we get $w \geq f^* - \lambda^{*T} y, \ \forall (y, w) \in A$. If $p(y)$ is finite, then $(y, p(y)) \in A$. Thus we have $p(y) \geq p(0) - \lambda^{*T} y$, which means $-\lambda^*$ is a subgradient of $p(y)$ at $y = 0$.

In the case where X has an empty interior, it means that some of the variables can be expressed by the others, thus we can reformulate the problem so that it is defined over the subspace which is parallel to the affine hull of X. □

3.2 Error Control

We just showed how to efficiently obtain J_t and ∇J_t to build supporting hyperplanes of $J_t(x_t)$, but we haven't discussed where to build these hyperplanes. From the geometric point of view, the flatter the function in a region, the fewer supporting

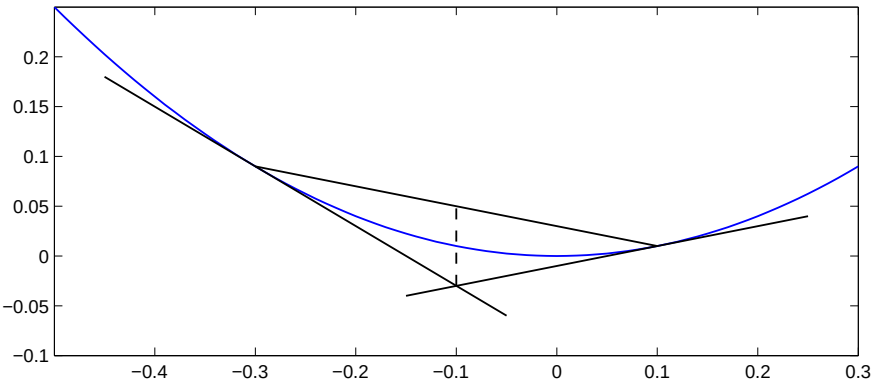

Figure 3.1: Upper, lower bounds and the max error

hyperplanes needed, vice versa. In principle, we want to be as economical as possible, because each supporting hyperplane will add a number of linear constraints to (3.4) in stage $t-1$. However, without prior knowledge of $J_t(x_t)$, presetting the investigation points $\{x_t^j\}$ is difficult — we will likely use too many supporting hyperplanes where the function is flat, and too few where the function is curved.

The approach used by ACE — the reason why it got the name — is to add supporting hyperplanes gradually and learn the shape of $J_t(x_t)$ on the way. It checks how well the current approximation does, and if at some place the current approximation is not good enough, it will add a new supporting hyperplane there.

To do this, we first need to measure the approximation error. Although the target function $J_t(x_t)$ is not known, recall that the supporting hyperplanes form a lower bound of it. In addition, the convexity of $J_t(x_t)$ gives an upper bound of it. With the lower bound and the upper bound, we can develop an estimate for the error.

Figure 3.1 provides a 1-dimensional example. We have two supporting hyperplanes, or tangents, added at x_t^1 = -0.3 and x_t^2 = 0.1, respectively. For $x_t \in [x_t^1, x_t^2]$, by the

definition of supporting hyperplanes,

$$J_t(x_t) \geq J_t(x_t^j) + \nabla J_t(x_t^j)^T(x_t - x_t^j),\ j = 1, 2,$$

and by the definition of convexity

$$J_t(x_t) \leq \alpha_1 J_t(x_t^1) + \alpha_2 J_2(x_t^2),$$

where

$$\alpha_1 = \frac{x_t^2 - x_t}{x_t^2 - x_t^1} \text{ and } \alpha_2 = \frac{x_t - x_t^1}{x_t^2 - x_t^1}.$$

Thus if we use the max of the tangents as the approximation of $J_t(x_t)$, the maximum potential error at x_t is the vertical distance from the approximation

$$\max_{j \in \{1,2\}} \left[J_t(x_t^j) + \nabla J_t(x_t^j)^T(x_t - x_t^j) \right]$$

to the segment connecting the two points of tangency,

$$y(x_t) = \frac{x_t^2 - x_t}{x_t^2 - x_t^1} J_t(x_t^1) + \frac{x_t - x_t^1}{x_t^2 - x_t^1} J_2(x_t^2).$$

Therefore, the maximum potential approximation error of the region $[x_1, x_2]$ is

$$\max_{x_t \in [x_1, x_2]} \left\{ \frac{x_t^2 - x_t}{x_t^2 - x_t^1} J_t(x_t^1) + \frac{x_t - x_t^1}{x_t^2 - x_t^1} J_2(x_t^2) - \max_{j \in \{1,2\}} \left[J_t(x_t^j) + \nabla J_t(x_t^j)^T(x_t - x_t^j) \right] \right\},$$

and in the 1-dimensional case, it is the distance between the upper and lower bounds at the point where the two tangent intersects, as Figure 3.1 shows.

More generally, suppose that the domain X_t is a p-dimensional body. Let $\{x_t^j\}_{j=1}^{p+1}$ be $p+1$ points whose convex hull $H_{conv}(\{x_t^j\}_{j=1}^{p+1})$ is p-dimensional. Recall that the

convex hull of a set S is the unique minimal convex set that contains S. In our case,

$$H_{conv}(\{x_t^j\}_{j=1}^{p+1}) = \{\sum_{j=1}^{p+1} \alpha_j x_t^j \mid \alpha_j \geq 0, j = 1, \ldots, p+1; \sum_{j=1}^{p+1} \alpha_j = 1\}$$

is the convex polytope whose vertices are $\{x_t^j\}_{j=1}^{p+1}$. If we have supporting hyperplanes at $\{x_t^j\}$, $j = 1, \ldots, N$ $(N \geq p+1)$, then for any $x_t \in X_t \cap H_{conv}(\{x_t^j\}_{j=1}^{p+1})$,

$$J_t(x_t) \geq J_t(x_t^j) + \nabla J_t(x_t^j)^T (x_t - x_t^j),\ j = 1, \ldots, N,$$

and

$$J_t(x_t) \leq \sum_{j=1}^{p+1} \alpha_j J_t(x_t^j),$$

where

$$\alpha_j \geq 0, j = 1, \ldots, p+1, \text{ and } \sum_{j=1}^{p+1} \alpha_j = 1.$$

Therefore, if we approximate $J_t(x_t)$ with

$$\hat{J}_t(x_t) = \max_{j \in \{1,\ldots,N\}} J_t(x_t^j) + \nabla J_t(x_t^j)^T (x_t - x_t^j),$$

the maximum potential error for the region $X_t \cap H_{conv}(\{x_t^j\}_{j=1}^{p+1})$ can be found by expressing the point x_t as a convex combination of $\{x_t^j\}_{j=1}^{p+1}$ and solving the following maximization problem:

$$\begin{aligned}
&\max_{x_t, \alpha, J} \left[\textstyle\sum_{j=1}^{p+1} \alpha_j J_t(x_t^j) - \max_{j \in \{1,\ldots,N\}} J_t(x_t^j) + \nabla J_t(x_t^j)^T (x_t - x_t^j) \right] \\
&\text{s.t. } x_t = \textstyle\sum_{j=1}^{p+1} \alpha_j x_t^j, \\
&\qquad \alpha_j \geq 0, \text{ for } j = 1, \ldots, p+1, \\
&\qquad \textstyle\sum_{j=1}^{p+1} \alpha_j = 1, \\
&\qquad x_t \in X_t,
\end{aligned} \tag{3.8}$$

where the objective function is the distance between the upper bound $\sum_{j=1}^{p+1}\alpha_j J_t(x_t^j)$ and the lower bound $\max_{j\in\{1,\ldots,N\}} J_t(x_t^j)+\nabla J_t(x_t^j)^T(x_t-x_t^j)$, the first three constraints defines x_t as the convex combination of $\{x_t^j\}_{j=1}^{p+1}$, and the last constraint is for feasibility. Since the max operation in the objective function of (3.8) is not convenient, we rewrite (3.8) as

$$\begin{aligned}
&\max_{x_t,\alpha,J} \textstyle\sum_{j=1}^{p+1}\alpha_j J_t(x_t^j) - J \\
&\quad \text{s.t. } x_t = \textstyle\sum_{j=1}^{p+1}\alpha_j x_t^j, \\
&\qquad \alpha_j \geq 0, \text{ for } j = 1,\ldots,p+1, \\
&\qquad \textstyle\sum_{j=1}^{p+1}\alpha_j = 1, \\
&\qquad x_t \in X_t, \\
&\qquad J \geq J_t(x_t^j) + \nabla J_t(x_t^j)^T(x - x_t^j), \text{ for } j = 1,\ldots,N.
\end{aligned} \tag{3.9}$$

There are two facts that may counter one's geometric intuition. First, one may wonder, to find the maximum potential error, why don't we look at the intersection point of the supporting hyperplanes at $\{x_t^j\}_{j=1}^{p+1}$, as solving linear equations is much easier and faster than solving a mathematical program. The reason is that when $p > 1$, the intersection may not be in $H_{conv}(\{x_t^j\}_{j=1}^{p+1})$; In fact it may not be in X_t, either, even if $H_{conv}(\{x_t^j\}_{j=1}^{p+1}) \subset X_t$. For instance, consider $p = 2$ and $J(x_1, x_2) = x_1^3 + x_2^3$. Let the feasible region be

$$X = \{(x_1, x_2) \mid x_1, x_2 \geq 0, x_1 + x_2 = 1\},$$

which is also the convex hull of $(0,0)$, $(0,1)$ and $(1,0)$. If we add supporting hyper-

planes at $(0,0)$, $(0,1)$ and $(1,0)$, the three supporting hyperplanes are

$$\begin{aligned} y &= 0, \\ y &= 3x_1 - 2, \\ y &= 3x_2 - 2, \end{aligned}$$

and they intersect at $x = (\frac{2}{3}, \frac{2}{3})$, which is neither in the convex hull, nor in the feasible region. Therefore solving (3.9) is necessary.

Second, a supporting hyperplane at x_t^j, $j > p+1$, can be an active lower bound, even when $x_t^j \notin H_{conv}(\{x_t^j\}_{j=1}^{p+1})$. For example, consider the $x = (0.6, 0.6)$, which is out of the convex hull of the above three points. The supporting hyperplane at $(0.6, 0.6)$ is

$$y = -0.864 + 1.08(x_1 + x_2).$$

If we look at the point $(0.5, 0.5)$, the supporting hyperplane at $(0.6, 0.6)$ has a larger value than the first three, so it is going to be active in (3.9). This is the reason why in the last constraint of (3.9) we have $j = 1, \ldots, N$.

3.3 Recursive Partitioning

Let the optimal solution of (3.9) be $(\bar{x}_t, \bar{\alpha}, \bar{J})$ and the optimal value be E^*. The approximation error in the region $X_t \cap H_{conv}(\{x_t^j\}_{j=1}^{p+1})$ is bounded above by E^*. Let *tol* be a user-defined tolerance. If $E^* \leq tol$, it means the approximation for the whole region of $X_t \cap H_{conv}(\{x_t^j\}_{j=1}^{p+1})$ is good enough. If $E^* > tol$, then we can add a supporting hyperplane to reduce the error, and it is natural to add it at $\bar{x}_t$, which is where the error is potentially the largest.

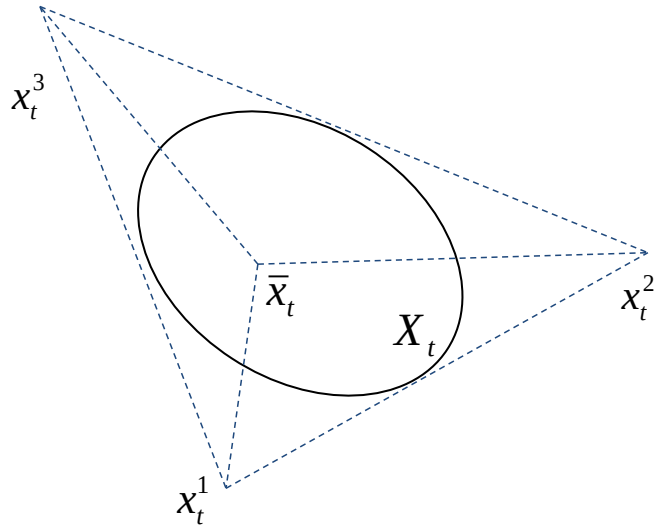

Figure 3.2: Partitioning at the potentially worst point

After adding the new supporting hyperplane at $\bar{x}_t$, we partition $H_{conv}(\{x_t^j\}_{j=1}^{p+1})$ to $p+1$ smaller convex hulls, each with p points from $\{x_t^j\}_{j=1}^{p+1}$ and $\bar{x}_t$ as vertices (see Figure 3.2). Now we are facing $p+1$ sub-regions whose maximum potential errors can be found by solving (3.9) again, and each can be further partitioned if the approximation in that sub-region is not good enough. By doing this partitioning recursively, the maximum potential error in all sub-regions will eventually be less than or equal to *tol*, which means the approximation error is now less than or equal to *tol* for all $x_t \in X_t \cap H_{conv}(\{x_t^j\}_{j=1}^{p+1})$.

Note that if $\bar{x}_t$ is on a facet or an edge of $H_{conv}(\{x_t^j\}_{j=1}^{p+1})$, some of the sub-regions will be less than p-dimensional. We ignore these sub-regions, as they are covered by the other p-dimensional sub-regions. We can tell the dimension of a sub-region by looking at $\bar{\alpha}$: if a convex hull is formed by $\bar{x}_t$ and $\{x_t^j\}_{j=1}^{p+1} \backslash \{x_t^i\}$, then it is p-dimensional if and only if $\bar{\alpha}_i > 0$.

To initiate the recursive partitioning process, we need to have $p+1$ initial vertices $\{x_t^j\}_{j=1}^{p+1}$ that satisfy two conditions: 1, their convex hull $H_{conv}(\{x_t^j\}_{j=1}^{p+1})$ contains X_t; 2,

the feasible decision sets $\{u_t \mid g_t(x_t^j, u_t) \leq 0\}$, $j = 1, \ldots, p+1$, are not empty. The first condition is for controlling error on the entire state space, while the second condition is needed so that we can build supporting hyperplanes at these vertices. These points are usually not hard to find. For example, if the states have constraints $0 \leq x_{t,i} \leq 1$, $i = 1, \ldots, p$, one may want to check the intersections of the hyperplanes $x_i = 0$, $i = 1, \ldots, p$, and $\sum_{i=1}^{p} x_i = p$ to see if they satisfy the second condition. However, we are not sure yet whether these points always exist, and we cannot guarantee that they are always easy to find. If one really has difficulties in finding these points, then one may want to make some compromise and choose $\{x_t^j\}_{j=1}^{p+1}$ that satisfy the second condition and their convex hull covers as much of X_t as possible.

Pseudo-code of the process is given in Algorithm 3.1. In the algorithm, we call a convex hull a "section", and we maintain a list of sections. The beginning of a list is its first item, while the end of a list is the position after the last item. An iterator of a list is like a pointer that points to an item of the list. The "`++`" operator moves the iterator to the next item, but if the iterator is now pointing to the last item of the list, "`iterator++`" will point it to the end of the list, i.e., the position after the last item.

As Algorithm 3.1 describes, at one time we look at only one convex hull, which is the one that the iterator points to. When we partition a section to subsections, the iterator ignores the new subsections and continues with the next item of the original list. When the iterator reaches the end of the list, it goes back to the beginning and starts the next round of refining.

It is also possible to do the recursive partitioning with a single round of refining. Algorithm 3.2 shows the pseudo-code of the process. Different from what we do in Algorithm 3.1, now when we do a partitioning, the iterator points to the first

Algorithm 3.1 Recursive Partitioning 1

```
Loop t from T-1 to 1

    Read supporting hyperplane information of J_{t+1} from file;
    Add supporting hyperplanes at initial
                              points {x_t^j}_{j=1}^{p+1} by solving (3.4);
    Form the first item of the section list with {x_t^j}_{j=1}^{p+1};
    currentMaxError = tolerance + 1;
    While (currentMaxError > tolerance AND list size <= Budget)
        currentMaxError = 0;
        iterator = beginning of the list;
        While (iterator != end of the list)
            Err = optimal value of (3.9);
            currentMaxError = max(currentMaxError, Err);
            If (Err <= tolerance)
              iterator++;
            Else
              Solve (3.4);
              Add a new supporting hyperplane at x̄_t;
              Separate the current section at x̄_t to subsections;
              iterator++;
              Replace the previous section with the subsections;
            End
        End
    End
    Write supporting hyperplane information of J_t to file;
    Release memory;

End
```

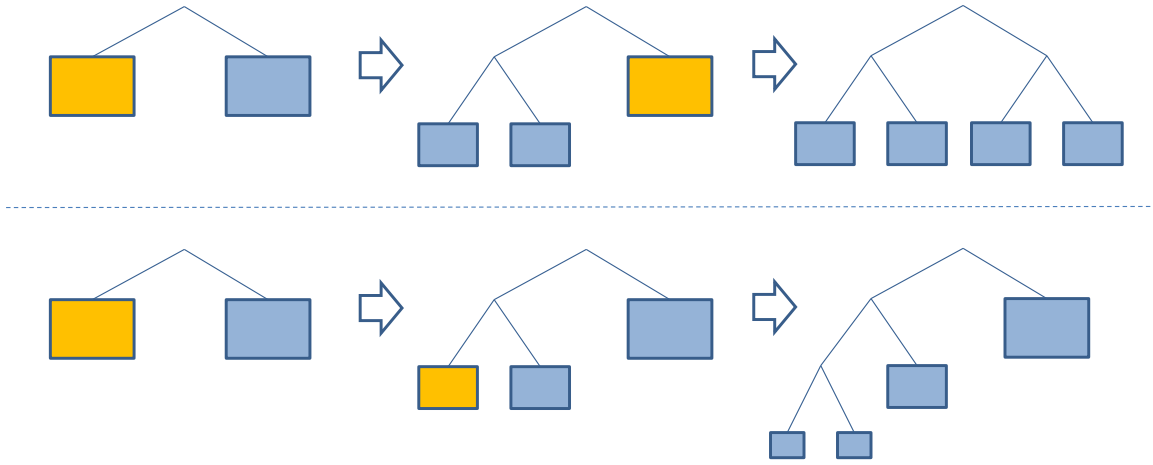

Figure 3.3: Ways for partitioning

new subsection on the list (see Figure 3.3), and the effect is that we do not leave a section until the approximation error of this section is less than or equal to the tolerance everywhere, which further means that when we reach the end of the list, the approximation is good enough for the entire state space.

Although both algorithms work for the recursive partitioning, our numerical experience shows that the breadth-first version (Algorithm 3.1) usually uses less supporting hyperplanes than the depth-first version (Algorithm 3.2) does. As we mentioned above, for $p \geq 2$, supporting hyperplanes outside a convex hull (a section) could improve the approximation quality inside it. Thus we may find that a subsection that would need partitioning if we went depth-first becomes well approximated after a round of refining in other sections. As a result, Algorithm 3.1 uses supporting hyperplanes more uniformly and more efficiently.

Note that since $\hat{J}_{t+1}$ also has approximation error, the error that we are talking about here is the error of $\hat{J}_t$ relative to $\hat{J}_{t+1}$. Bounds for the absolute error are developed in

Algorithm 3.2 Recursive Partitioning 2

```
Loop t from T-1 to 1
Read supporting hyperplane information of J_{t+1} from file;

    Add supporting hyperplanes at initial
                              points {x_t^j}_{j=1}^{p+1} by solving (3.4);
    Form the first item of the section list with {x_t^j}_{j=1}^{p+1};
    currentMaxError = 0;
    iterator = beginning of the list;
    While (iterator != end of the list)

        Err = optimal value of (3.9);
        currentMaxError = max(currentMaxError, Err);
        If (Err <= tolerance)

            iterator++;

        Else

            Solve (3.4);
            Add a new supporting hyperplane at x̄_t;
            Separate the current section at x̄_t to subsections;
            Replace the previous section with the subsections;
            iterator = the first new subsection;

        End

    End
    Write supporting hyperplane information of J_t to file;
    Release memory;

End
```

Section 3.4.

The Simple Inventory Control Example

Here we use the simple inventory control problem to demonstrate how recursive partitioning works. The simple inventory control problem is a well-known textbook example. At each stage, we order new items to meet a stochastic demand. Let x_t be the initial inventory at the beginning of stage t, $u_t \geq 0$ be the amount of new items we order, and w_t be the demand. The cost function per stage is

$$c_t(x_t, u_t, w_t) = cu_t + p\max(0, w_t - x_t - u_t) + h\max(0, x_t + u_t - w_t),$$

where c is the purchasing cost for an item, p is the shortage cost per item if inventory does not meet the demand, and h is the holding cost per item that is left at the end of each period. The excess demand is backlogged and is filled when additional inventory is available, so the system transition is

$$x_{t+1} = x_t + u_t - w_t,\ t = 1, \ldots, T-1.$$

The goal is to find the policy $\{\mu_t\}$ that minimizes the total expected cost

$$E\left\{\sum_{t=1}^{T-1}\left[cu_t + p\max(0, w_t - x_t - \mu_t(x_t)) + h\max(0, x_t + \mu_t(x_t) - w_t)\right]\right\},$$

which does not include salvage value, so $J_T(x_T) = 0$.

Let c, p and h be 2, 4 and 0.2, respectively. Let the demand w_t be uniform on $[0, 10]$, and let the samples be 0.0, 0.1,. . . ,9.9 with equal weights. The assumptions listed in Section 3.1 are satisfied, thus the value functions are convex. The optimal policy is a

well-known S-type policy:

$$\mu_t(x_t) = \begin{cases} S_t - x_t, & x_t < S_t; \\ 0, & x_t \geq S_t; \end{cases}$$

that is, to order and increase the inventory up to a level S_t. However, the value of S_t is not known, and need to be computed numerically. Conventionally this is done by simulation, that is, by trying various values of S_t to find the one that minimizes the cost. Here we find S_t as well as $J_t(x_t)$ with ACE.

We solved the DP for $x \in [0, 15]$, t from $T-1$ to $T-10$, at tolerance $tol = 0.05$. We used Algorithm 3.2 for the recursive partitioning, which was for no special reason, as Algorithms 3.1 and 3.2 have no difference in supporting hyperplane efficiency for 1-dimensional problems.

The approximation of J_{T-1} is shown by Figure 3.4 (top). The supporting hyperplanes information of J_{T-1} is given in Table 3.1. By analysis, treating w_{T-1} as continuous, we know that the true J_{T-1} is

$$J_{T-1}(x_{T-1}) = \begin{cases} 15.238 - 2x_{T-1}, & x_{T-1} < S; \\ 0.21x_{T-1}^2 - 4x_{T-1} + 20, & S \leq x_{T-1} \leq 10; \\ -1 + 0.2x_{T-1}, & x_{T-1} > 10, \end{cases}$$

where $S = 4.762$. Replacing the expectation with the sample average brings sampling error, and the policy that we obtained is to order up to $S = 4.75$, which is the intersection of the first two tangents from the left. For the region $x_{T-1} < S$ and $x_{T-1} > 10$, $J_{T-1}(x_{T-1})$ is linear, and we see that ACE does not waste supporting hyperplanes (tangents) there. For the region $S \leq x_{T-1} \leq 10$, $J_{T-1}(x_{T-1})$ is quadratic under a demand from a uniform distribution, and the approximation reflects the shape

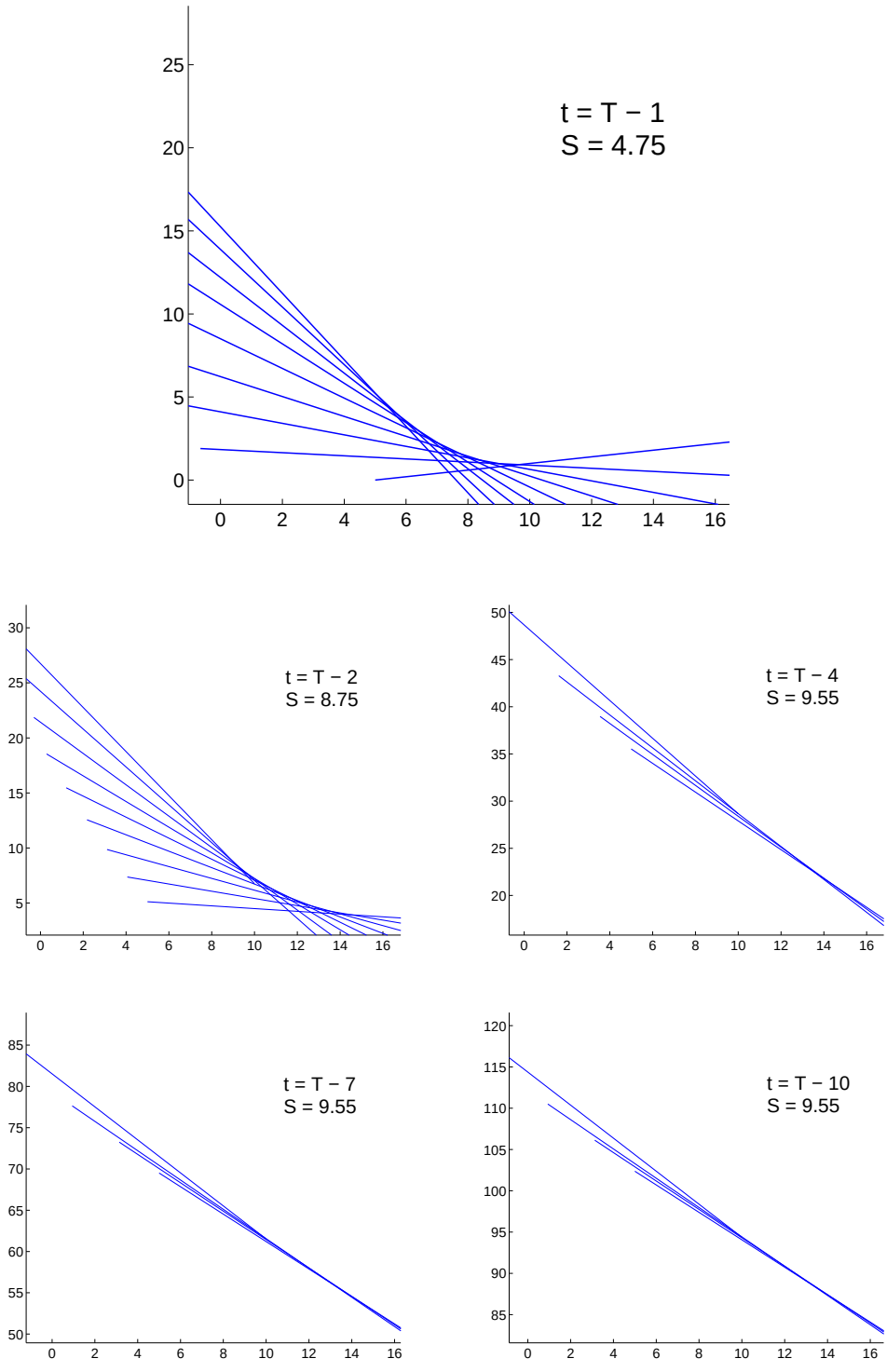

Figure 3.4: Approximations for the simple inventory control problem

x	$\nabla J_{T-1}(x)$	$J_{T-1}(x)$
0	-2	15.2376
15	0.2	2
7.38073	-0.892	1.91679
8.7	-0.346	1.0949
9.35	-0.094	0.9582
8.05	-0.598	1.408
6.08051	-1.438	3.44213
6.75	-1.186	2.5676
5.43007	-1.732	4.47152

Table 3.1: Supporting hyperplane information of J_{T-1}

closely. Since the next stage value function $J_T(x_T)$ is known, if we do not count the sampling error, the function values and gradients in Table 3.1 are exact. Overall, the approximation has an error less than 0.05 for any point $x_{T-1} \in [0, 15]$.

Figure 3.4 also shows the approximations of J_{T-2}, J_{T-4}, J_{T-7} and J_{T-10}, whose closed form solutions are hard to derive, which is the reason why we need numerical methods. The policy at stage $T-2$ is to order up to $S = 8.75$. For stage $T-3$, $S = 9.45$. Starting (backwards) from stage $T-4$, the policy becomes fixed, and we always order up to $S = 9.55$. We see that less supporting hyperplanes are used as the stage proceeds backwards and as the function becomes closer to linear. The maximum potential error accumulates. We will show in Section 3.4 that at stage $T-10$, a conservative error bound for a point $x \in [0, 15]$ is 0.5. On the other hand, the cost-to-go accumulates, too; and compared to the scale of the target, the error is very small.

3.4 Features of ACE

ACE is a deterministic algorithm with optimization-oriented designs. It doesn't have the convergence issue, nor the exploration vs. exploitation dilemma as the simulation-

based algorithms do. For the approximation, ACE does not use popular forms such as feature functions or kernels, but uses supporting hyperplanes. The supporting hyperplane approximation is convexity-preserving, so we can use convex programming, which makes optimization of large numbers of decision variables under nontrivial, state-dependent constraints fast and reliable. On the other hand, the sensitivity information that convex programming algorithms provide for free is essential to the fast and accurate construction of supporting hyperplanes. Therefore, as we can see, it is a perfect symbiotic relationship between the supporting hyperplane approximation and convex optimization. The supporting hyperplane approximation also makes ACE easier to standardize than existing methods, because the user does not need to try different feature functions, kernels or neural network designs, tune the function parameters and the learning speed, which are crucial to the performance of the obtained policy and are extremely time consuming. With ACE, the user only needs to give the model of the problem and provide points to initiate the algorithm, which are very similar as what people do with the mature mathematical programming algorithms today.

Unlike SBDP that uses forward simulations repeatedly, ACE steadily proceeds backwards in stage, same as the exact DP algorithm. One of the biggest advantages provided by proceeding backwards is the superb data quality. Approximating multidimensional functions from noisy observations is difficult. The larger the noise, the more difficult the approximation. Unfortunately, the noise in the simulated samples could be very large as it is accumulated through the stages. However, when proceeding backwards, we don't have this problem. If the approximation of the next stage value function J_{t+1} were exact, with a sample of the random variable large enough, we can obtain the value of J_t at the investigated state with little error. So the the data is "clean" relative to the estimate of the next stage, which is critical to the quality of the approximation.

Of course the data quality alone does not guarantee the quality of the approximation. The aggressive nature of many approximation functions are another source of overfitting. Thanks to the recursive partitioning error control mechanism and the conservative nature of the supporting hyperplane approximation, ACE is able to get rid of overfitting and achieve high quality approximation on the entire state domain.

More specifically, let $J_t(x_t)$ and $\hat{J}_t(x_t)$ be the optimal value function and the value function estimated by ACE, respectively. Let $J_t^{\pi}(x_t)$ be the actual expected cost-to-go of the obtained policy, i.e.,

$$J_t^{\pi}(x_t) = E\left[\sum_{i=t}^{T-1} c_i(x_i, \hat{\mu}_i(x_i), w_i) + c_T(x_T, w_T) \mid x_t\right], \ \forall x_t \in X_t.$$

Also define $\tilde{J}_t(x_t) \triangleq \min_{u_t} E[c_t(x_t, u_t, w_t) + \hat{J}_{t+1}(x_{t+1})]$, $\forall x_t \in X_t$. Then we have the following relationships:

$$\hat{J}_t(x_t) \leq \tilde{J}_t(x_t) \leq J_t(x_t) \leq J_t^{\pi}(x_t) \leq \hat{J}_t(x_t) + (T-t)tol, \ \forall x_t \in X_t. \tag{3.10}$$

The first inequality of (3.10) is by convexity. The second inequality can be shown by induction: if $\hat{J}_{t+1}(x_{t+1}) \leq J_{t+1}(x_{t+1})$ for all $x_{t+1} \in X_{t+1}$, since

$$\tilde{J}_t(x_t) = \min_{u_t} E[c_t(x_t, u_t, w_t) + \hat{J}_{t+1}(x_{t+1})]$$

and

$$J_t(x_t) = \min_{u_t} E[c_t(x_t, u_t, w_t) + J_{t+1}(x_{t+1})],$$

we have $\tilde{J}_t(x_t) \leq J_t(x_t)$, $\forall x_t \in X_t$, which also gives $\hat{J}_t(x_t) \leq J_t(x_t)$. The third inequality is apparent: by definition, the optimal policy performs at least as good as any other policy. The last inequality assumes that we do not need to approximate $J_T(x_T)$ (otherwise it would be $J_t^{\pi}(x_t) \leq \hat{J}_t(x_t) + (T-t+1)tol$), and can be shown by

induction, too: if

$$J^{\pi}_{t+1}(x_{t+1}) \leq \hat{J}_{t+1}(x_{t+1}) + (T-t-1)tol, \text{ for all } x_{t+1} \in X_{t+1},$$

then for any $x_t \in X_t$,

$$\begin{aligned} J^{\pi}_t(x_t) &= E[c_t(x_t, \hat{\mu}_t(x_t), w_t) + J^{\pi}_{t+1}(x_{t+1})] \\ &\leq E[c_t(x_t, \hat{\mu}_t(x_t), w_t) + \hat{J}_{t+1}(x_{t+1}) + (T-t-1)tol] \\ &= \tilde{J}_t(x_t) + (T-t-1)tol \\ &\leq \hat{J}_t(x_t) + tol + (T-t-1)tol \\ &= \hat{J}_t(x_t) + (T-t)tol. \end{aligned} \tag{3.11}$$

Note that the second inequality uses the fact that $\tilde{J}_t(x_t) \leq \hat{J}_t(x_t) + tol$, which is by the stopping criterion of the algorithm.

From (3.10), we see that the ACE estimate $\hat{J}_t(x_t)$ is a lower bound of the optimal value function $J_t(x_t)$ and that we can either use $J^{\pi}_t(x_t)$ or $\hat{J}_t(x_t) + (T-t)tol$ as an upper bound. The actual performance $J^{\pi}_t(x_t)$ usually needs to be simulated, but it is often a much tighter bound than $\hat{J}_t(x_t) + (T-t)tol$. A good thing about these bounds is that they are clear numbers, rather than analysis terms such as the big-O notation, so they give us confidence in knowing how the policy performs and how it compares to the optimal policy.

There are many techniques to speed up ACE. In addition to the standard recursive partitioning algorithm that significantly saves supporting hyperplanes, there also exist problem-dependent tricks. For instance, if the cost functions and constraints are piecewise linear, such as in the above simple inventory control example, then only the first supporting hyperplane needs to be computed from scratch — the later ones can

be computed by dual simplex updates. In fact, the above inventory control example was solved just in this fashion, and it took 0.087 second per stage (including setting up the optimization environment and models, reading and writing files, etc.) on average on a personal computer. We would like to mention once again that in addition to this computation time, we only had to plug in the model. No time was spent on designing or parameter tuning, and we know the performance of the solution and the maximum possible error of the estimated value functions right after the computation finishes.

The handling of $J_{t+1}(x_{t+1})$ in (3.3) and (3.4) may remind the reader of 2-stage stochastic programming, especially, the Benders' decomposition (Benders [1962], Van Slyke and Wets [1969]). The similarity lies in the use of supporting hyperplanes (cutting planes, in the literature of SP); the difference is that in Benders' decomposition, cutting planes are added after and in response to a first stage decision u_t, while in ACE supporting hyperplanes are constructed before choosing u_t and for the entire X_{t+1}. Of course, we are not solving 2-stage problems, but multistage ones. The nested Benders' decomposition (Birge [1985]) solves multistage stochastic programs, but the CPU time and memory consumption grow exponentially with the number of stages, so it will soon be intractable even with a very small number of scenarios (realizations of w_t). In comparison, the CPU time of ACE grows linearly with the number of stages, and the memory consumption remains roughly the same, because J_{t+1} contains all the relevant information of the future stages for the optimization, and we are always performing 2-stage stochastic optimization with the Bellman's equation. As a result, ACE allows a much longer planning horizon and a larger number of scenarios at each stage. And needless to say, ACE is for dynamic programming — once we have obtained $\{\hat{J}_t\}_{t=1}^{T}$, we can find the optimal decision at any time and any state by solving (3.3), without having to run the entire computation again.

ACE is not without its limitations, with the major one being the limited number

of state variables. Solving hundreds of decision variables is not a threat to ACE, because convex optimization algorithms are not sensitive to that. However, as the dimension of the state space grows, the number of supporting hyperplanes needed grows exponentially. The supporting hyperplanes are global constraints, but usually they are active only locally, so ACE bears more traits of local approximation methods than those of global ones, and has this kind of dimensionality problems as a result. Generally speaking, if the value function is very curved, 10 state variables would make the problem hard to solve for personal computers. For problems of this scale, the user may want to coarsely solve the problem with a relatively large tolerance first. Since the optimal policy may try to guide the system to a small portion of the state space, obtaining a precise approximation of the entire value function may not be necessary. Thus the user can improve the approximation by importance with a lower new tolerance using Algorithm 3.3.

Algorithm 3.3 Approximation by Importance

```
Observe or simulate state path {x_t}_{t=1}^T under the current policy;
Loop t from T-1 to 1
    Find the section that contains x_t;
    If (maximum potential error > tolerance)
        Add a supporting hyperplane at x_t;
        Separate the current section at x_t;
    End
End
```

Finally, let us not forget that there are many problems that can be decomposed. For example, certain types of multistage portfolio management problems are decomposable. Let $x_{t,i}$ be the investment (measured in money value) in asset i at time t, and let $buy_{t,i}$ and $sell_{t,i}$ be the amount of asset i we purchase and sell, respectively. Assume that the transaction cost is proportional to $\sum_i (buy_{t,i} + sell_{t,i})$, the amount of the total

transaction. Then the problem could be formulated as

$$
\begin{aligned}
J_t(x_t) = \max\nolimits_{u_t} E[-c\textstyle\sum_i(buy_{t,i} + sell_{t,i}) + J_{t+1}(x_{t+1})] \\
\text{s.t. } x_{t,i} + buy_{t,i} - sell_{t,i} \geq 0, \text{ for all } i, \\
buy_{t,i} \geq 0, sell_{t,i} \geq 0, \text{ for all } i, \\
\textstyle\sum_i(buy_{t,i} + sell_{t,i}) = 0, \\
x_{t+1,i} = w_i(x_{t,i} + buy_{t,i} - sell_{t,i}), \text{ for all } i,
\end{aligned}
$$

where c is the rate of the transaction cost, and $w_i \geq 0$ reflects the random change of the asset value. The value in the last stage $J_T(x_T)$ equals the total asset value $\sum_i x_{T,i}$. We see that $\sum_i(buy_{t,i} + sell_{t,i}) = 0$ is the only linking constraint across the assets, so this DP could be decomposed to a set of 1-dimensional problems. Since ACE solves 1-dimensional problems very fast, it is capable to handle a large number of assets.

Chapter 4

The Optimal Charging Problem

We began the dissertation with the optimal battery charging problem from battery exchange station management, then we introduced Adaptive Convex Enveloping for general convex dynamic programming. Now let us come back to the battery problem and solve it with ACE.

The first batch of battery exchange stations are being built right this year (2012) in China. As for now, there is no real market data that we could use. On the other hand, the purpose of the numerical examples in this chapter are for demonstrating the capability of the new method for dynamic programming, rather than for actual operational deployments, so we can take the liberty and set the parameters with values that make the effect of the optimal policy easy to see. The optimization library used in the following examples is GUROBI 4.5.2 (Bixby et al. [2010]).

4.1 Optimal Policy for Battery Charging

In this example, we assume that the station has 100 batteries. Each full battery has $M = 3$ bars of energy, so the energy level m could be 0, 1, 2 or 3. Assume that an empty battery takes 3 hours to charge to full, which means charging a battery's energy level up by 1 takes 1 hour, and for convenience we set the time interval between stages to be 1 hour. Let the planning horizon be 6:00 am to 3:00 am of the next day, so there are 21 stages in total. From 3:00 am to 6:00 am, the electricity price is usually the lowest during a day, so the station can use this period of time to charge all the batteries and start a new day with all batteries full. Also for simplicity assume that no customer comes during 3:00 am - 6:00 am, so we can set $J_T(x_T) = 0$, $\forall x_T \in X_T$.

Suppose that a customer pays $R = 1.5$ dollar for every bar of energy charged, and let the penalty be $Pnlt = 5$ for losing a customer. During the day, we assume a stable distribution for customer arrival per hour: the number of customers who come with a level 0/1/2 battery is Poisson with mean 4/9/2, respectively. We use a stable distribution for the customer arrival to make the optimal policy's response to the electricity price easier to see. The electricity prices at each hour are given in the 2nd column in Table 4.1. Finally, for each stage, we use 50 randomly generated sample for the scenarios.

Unlike the standard form in our algorithm development, the value functions are concave, and we do maximization here, so the original lower and upper bounds are reversed. We solved the problem at $tol = 1$. Since $\sum_{m=0}^{3} x_{t,m} = 100$, we chose the intersections of this hyperplane with each axis as the initial points for the ACE algorithm. Table 4.1 shows the decisions and $\tilde{J}_t$ at each hour at 3 inventory levels: (10, 20, 30, 40) (i.e. 10 empty, 20 LV1, 30 LV2 and 40 full), (25, 25, 25, 25) and (40, 30, 20, 10).

		State (10,20,30,40)				State (25,25,25,25)				State (40,30,20,10)			
Time	Price	LV0	LV1	LV2	Value	LV0	LV1	LV2	Value	LV0	LV1	LV2	Value
6	0.05	10	20	30	590.1	25	25	25	560.6	40	30	20	496.6
7	0.08	10	20	30	545.4	25	25	25	505.2	40	30	20	430.0
8	0.15	0	0	6.2	507.3	0	7.0	20.9	461.5	0	14.8	20	381.4
9	0.15	0	0	0	486.8	0	0.1	11.7	446.0	2.7	16.0	19.0	369.7
10	0.1	10	20	30	461.8	25	25	25	426.8	40	30	20	361.7
11	0.1	10	20	30	437.2	25	25	25	400.5	40	30	20	330.9
12	0.15	0	0	0	413.2	0	1.5	12.5	371.9	0	15.3	20	294.8
13	0.1	10	20	30	386.7	25	25	25	351.7	40	30	20	279.0
14	0.1	10	20	30	359.2	25	25	25	324.2	40	30	20	260.6
15	0.1	10	20	30	330.3	25	25	25	284.8	40	30	20	208.0
16	0.1	0	20	30	294.6	11.6	25	25	226.9	24.8	30	20	131.4
17	0.2	0	20	30	249.3	0	25	25	173.0	7.2	30	20	49.4
18	0.4	0	0	0	227.9	0	0	10.0	144.6	0	11.0	16.8	20.0
19	0.3	0	0	0	234.8	0	3.0	13.2	166.5	0	16.6	20	65.1
20	0.25	0	0	7.6	233.9	0	7.4	21.7	174.6	0	14.9	20	78.9
21	0.25	0	0	0	225.4	0	0	11.8	175.4	0	11.0	18.3	88.1
22	0.2	0	0	0	205.9	0	0	10.9	171.2	0	13.1	17.8	102.2
23	0.1	0	0	7.5	175.1	0	6.4	19.1	154.7	0	13.9	20	94.8
24	0.1	0	0	0	136.3	0	0	11.8	124.9	0	15.4	19.6	74.7
1	0.04	0	0	0	94.7	0	0	15.0	90.5	0	0	20	50.7
2	0.02	0	0	0	47.0	0	0	0	47.0	0	0	0	10.5

Table 4.1: Decisions at certain states

We can see from Table 4.1 that the policy in general favors charging fuller batteries first, which makes sense as they would be ready to serve the customers earlier than emptier ones. In fact this was our initial hypothesis for the battery charging problem when we first looked at it, and if the hypothesis was true, it could reduce the number of decision variables to 1 (i.e., the total number of batteries to charge), which would greatly simplify the optimization and would make SBDP a good alternative method. However, we found that the hypothesis was not always true. For a counter example, assume that we have 2 batteries in total, one at LV $M-1$, the other at LV $M-2$. Let the charging cost be 10 per battery per level for the current stage, 0 for the next stage, and let the penalty for losing a customer be ∞. Also assume that no customer will arrive for the current stage and the next stage, but 2 will arrive in the third stage. We see that for this example, the optimal policy is to charge the emptier battery at the current stage, and charge both at the second stage, then we will have 2 full batteries ready at the third stage with a total cost of 10. In comparison, if we charge both batteries at the current stage, and the one not full the second stage, the cost will be 20; And if we charge only the fuller battery at the current stage, we will have a full and a LV $M-2$ battery in the next stage, and at best a full and a LV $M-1$ battery in the third stage, thus we will not have enough batteries to serve customers, triggering the penalty, which is ∞. This counter example shows that the heuristic that charges fuller batteries first is not always optimal, which is the reason why we decided to use mathematical programming and developed ACE. The decisions in Table 4.1 also confirm that while the policy favors charging fuller batteries first, it also needs to balance the numbers of batteries at each energy level so that we can charge more batteries when a good opportunity comes.

One might be curious about the optimality of the policy that we obtained. We assumed that the station starts a day with all 100 batteries full. The expected daily profit of the policy estimated by ACE is $\tilde{J}_1(x_1^0) = 634.90$, where $x_1^0 = (0, 0, 0, 100)^T$.

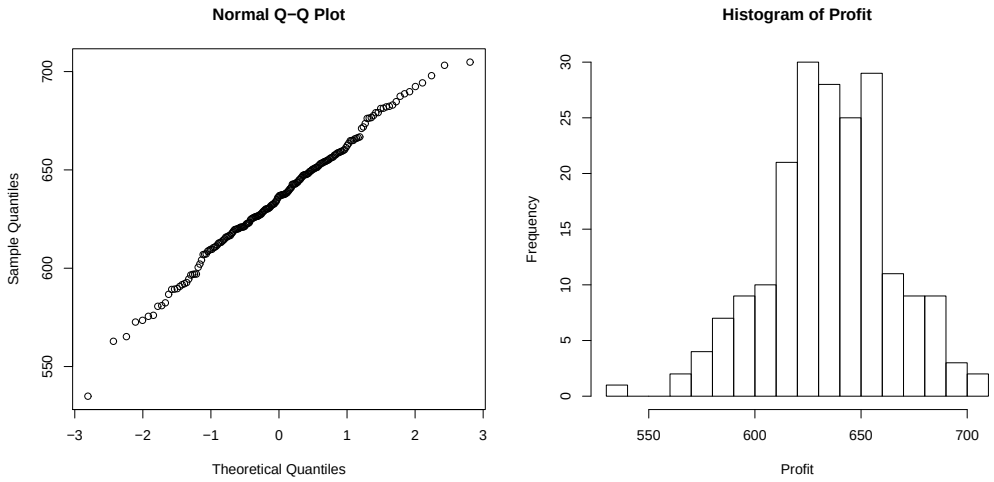

Figure 4.1: Simulated daily profits (3 bars of energy)

By (3.10) and the second equality of (3.11), the performance of the optimal policy and the actual performance our obtained policy are bounded by

$$634.90 = \tilde{J}_1(x_1^0) \geq J_1(x_1^0) \geq J_1^\pi(x_1^0) \geq \tilde{J}_1(x_1^0) - (T - t - 1)tol = 614.90.$$

However, the lower bound 614.90 is actually very conservative. We ran simulation 200 times to estimate the actual expected daily profit using the sample mean, which turned out to be 635.33, even greater than $\tilde{J}_1(x_1^0) = 634.90$ and seems to be contradicting with (3.10). But this does not mean that something went wrong. The sample mean is a random variable. The standard deviation of the sample is 29.12, and for the sample mean it is 2.06, so there is a chance that the estimate of $J_1^\pi(x_1^0)$ exceeds the upper bound $\tilde{J}_1(x_1^0)$ when they are close enough. Indeed, $J_1^\pi(x_1^0)$ and $\tilde{J}_1(x_1^0)$ are so close that we could not even reject $J_1^\pi(x_1^0) = \tilde{J}_1(x_1^0)$ with hypothesis test (we assumed normal distribution as Figure 4.1 indicates), which would further imply that the error is not different from 0. Therefore, we conclude that the policy we obtained with ACE is "statistically optimal".

Figure 4.2 shows one of the simulations. The background displays the number of

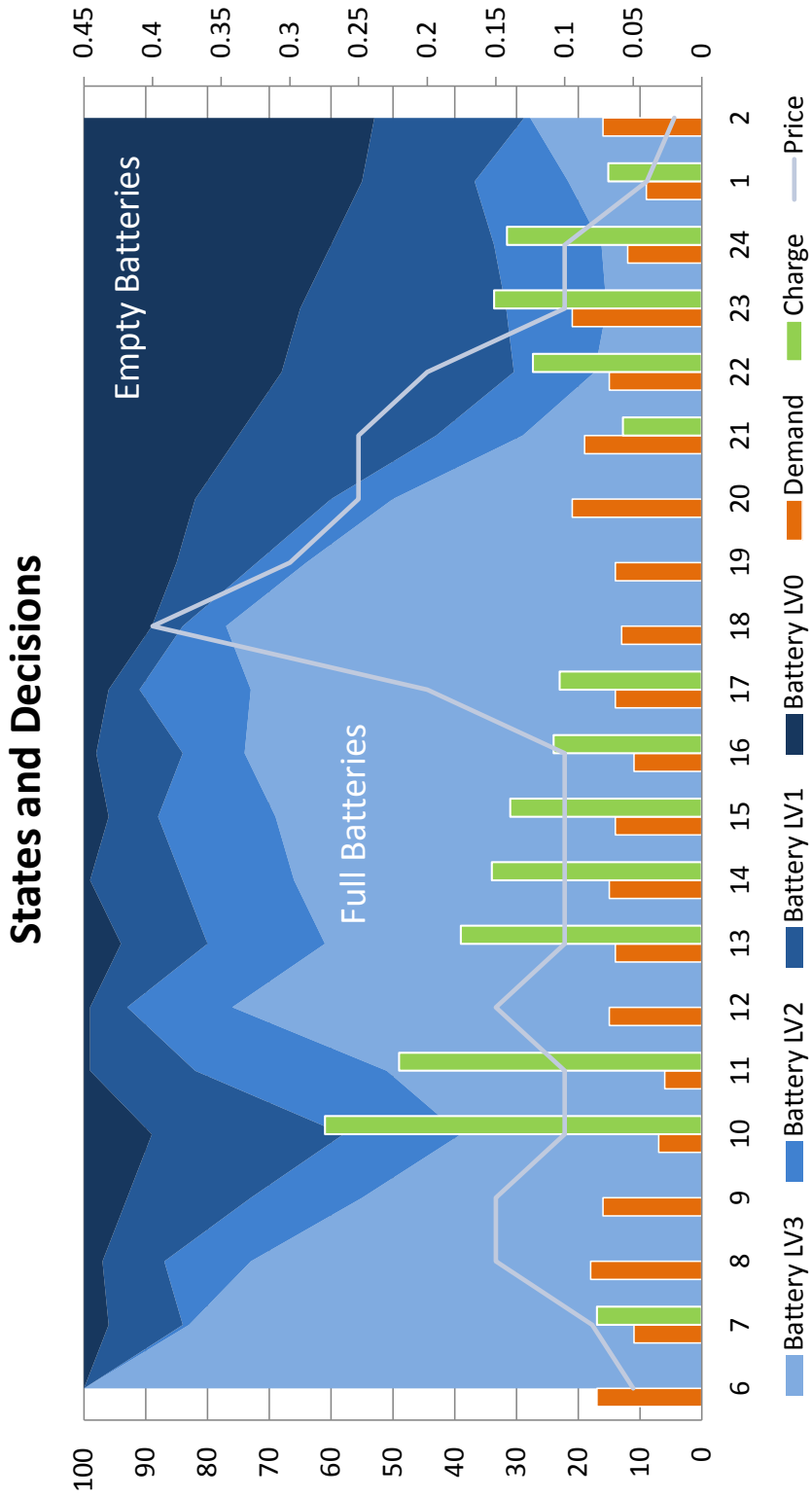

Figure 4.2: Simulation of a day (3 bars of energy)

batteries at each energy level that the stations has at each stage. The quantities are stacked upon each other, where at the bottom is the number of full batteries, and at the top is the number of empty batteries. The amount of the demand and the number of batteries to charge are aggregate values, i.e., $\sum_{m=0}^{M-1} w_{t,m}$ and $\sum_{m=0}^{M-1} u_{t,m}$. We see that there are 3 price peaks during the day, and the policy tends not to charge during those periods. On the other hand, the policy is not a simple response to the inventory and the price. For instance, the electricity price at 5pm is higher than that at 9am, and the inventory of full batteries at 5pm is higher, too. However, we choose to charge batteries at 5pm but not so at 9am. The explanation is that the third price peak is much longer than the first two, which means more customer demand, so we need to charge at some time; also since the price will be even higher in succeeding hours, it is best that we charge at 5pm. Of course, this type of long-term plannings is what we expect from DP and what we should get if we solve the Bellman's equation accurately.

4.2 Optimal Charging Policy with Reverse Dispatch to the Grid

Here we demonstrate the convenience of ACE for modeling by slightly modifying the problem.

Since that a battery exchange station is also an electricity storage facility, and that it is technically possible to send/sell electricity back to the grid. it could be profitable to buy and store electricity at low price and sell it back when the price is high. On the other hand, from the point of view of the power system, the electricity is cheap when the demand is low (relative to the supply), and expensive when the demand

is high, so this buying-storing-selling operation could smooth the electricity demand and reduce the cost spent on the peak-load power generation, which is the reason why it is advocated by the "smart grid" technology. So here we consider the optimal charging problem with the selling option.

For many methods, this modification would mean a complete redesign of policies and large amount of time spent on choosing feature functions and tuning parameters — maybe it would turn out that the old parameters and features are still good, but one would not be sure until a large amount of choices have been tried. In comparison, ACE only needs the mathematical model, so we only need to add the new variables for selling and modify some constraints, which is very convenient and time saving.

We assume that 5% of the energy is lost during the charge/discharge process (so if we charge an empty battery with 10 kWhs of electricity, we can get 9.5 kWhs out of it). For convenience, let the charging and discharging speed be the same. In addition, we now assume that a battery is calibrated to have 4 bars of energy, and a customer pays 1 dollar for charging each bar. The interval between stages is now 45 minutes, and we have 28 stages in total. The distributions for customer arrivals are now Poisson with mean 4/9/2/1 for the numbers of customers who come with batteries at LV 0/1/2/3, respectively. The penalty for losing a customer remains the same. Finally, we modified the electricity prices during the day.

With the new setting, the dimension of the state space increased, which means the number of supporting hyperplanes needed would increase significantly. However, as the previous example shows, it is possible that we get a high quality solution without using very strict tolerance. So in this example, we relaxed the tolerance to $tol = 2.5$, which makes the number of supporting hyperplanes needed per stage remain roughly at the same level as in the previous example.

The Bellman's equation of the new problem is as follows, where we use $v_{t,m}$ for the number of level m batteries to discharge at time t:

$$
\begin{aligned}
J_t(x_t) = & \max_{u_t, \bar{w}_t^k} \{\textstyle\sum_{k=1}^{K} \mathbb{P}(w_t^k)[Revenue_t^k - Cost_t - Penalty_t^k + J_{t+1}(x_{t+1}^k)]\} \\
\text{s.t. } & Revenue_t^k = R\textstyle\sum_{m=0}^{M-1}(M-m)\bar{w}_{t,m}^k + 0.95c_t\sum_{m=0}^{M-1} v_{t,m}, \text{ for all } k, \\
& Cost_t = c_t\textstyle\sum_{m=0}^{M-1} u_{t,m}, \\
& Penalty_t^k = Pnlt(\textstyle\sum_{m=1}^{M-1} w_{t,m}^k - \sum_{m=1}^{M-1}\bar{w}_{t,m}^k), \text{ for all } k, \\
& 0 \le \bar{w}_{t,m}^k \le w_{t,m}^k, \text{ for all } k, 0 \le m \le M-1, \\
& \textstyle\sum_{m=1}^{M-1}\bar{w}_{t,m}^k + v_{t,M} \le x_{t,M}, \text{ for all } k, \\
& 0 \le u_{t,m} + v_{t,m} \le x_{t,m}, \ 1 \le m \le M-1, \\
& 0 \le u_{t,0} \le x_{t,0}, \\
& x_{t+1,M}^k = x_{t,M} - \textstyle\sum_{m=1}^{M-1}\bar{w}_{t,M}^k + u_{t,M-1} - v_{t,M}, \text{ for all } k, \\
& x_{t+1,m}^k = x_{t,m} - u_{t,m} + u_{t,m-1} \\
& \qquad\qquad - v_{t,m} + v_{t,m+1} + \bar{w}_{t,m}^k, \text{ for all } k, 1 \le m \le M-1, \\
& x_{t+1,0}^k = x_{t,0} - u_{t,0} + v_{t,1} + \bar{w}_{t,0}^k, \text{ for all } k.
\end{aligned}
$$

Again, we start a day with 100 full batteries. The ACE-estimated average daily profit is $\tilde{J}_1(x_1^0) = 754.35$, where $x_1^0 = (0, 0, 0, 0, 100)^T$, and the average of 200 simulation is 745.39, with standard deviation 2.20. The performance of the optimal policy is bounded by $745.39 \le J_1(x_1^0) \le 754.35$. As we can see, although we cannot claim optimality this time, the maximum possible error is very small compared to the scale, and the largest possible improvement is less than or equal to $(754.35 - 745.39)/745.39 = 1.20\%$.

Tables 4.2, 4.3 and 4.4 are the decisions at each stage for states $(10, 15, 20, 25, 30)$, $(20, 20, 20, 20, 20)$ and $(30, 25, 20, 15, 10)$, respectively, where CLVm is the number

Time	Price	CLV0	CLV1	CLV2	CLV3	DLV1	DLV2	DLV3	DLV4	Value
6:00	0.02	10	15	20	25	0	0	0	0	725.60
6:45	0.03	10	15	20	25	0	0	0	0	678.02
7:30	0.08	10	15	20	25	0	0	0	0	629
8:15	0.1	10	15	20	25	0	0	0	0	595.34
9:00	0.13	0	0	1.80	10.59	6.23	3.06	0	0	567.93
9:45	0.1	10	15	20	14.93	0	0	0	0	547.82
10:30	0.08	10	15	20	25	0	0	0	0	523.52
11:15	0.1	10	15	20	25	0	0	0	0	489.60
12:00	0.13	10	6.76	8.50	21.23	0	0	0	0	461.05
12:45	0.13	10	0.47	3.55	11.45	0	0	0	0	443.76
13:30	0.1	10	15	20	25	0	0	0	0	423.59
14:15	0.1	10	15	20	25	0	0	0	0	397.36
15:00	0.12	10	15	20	25	0	0	0	0	368.63
15:45	0.13	10	15	20	25	0	0	0	0	344.15
16:30	0.17	10	15	20	25	0	0	0	0	319.92
17:15	0.2	10	15	20	18.40	0	0	0	0	307.86
18:00	0.2	10	15	20	25	0	0	0	0	304.57
18:45	0.25	0	0	5.49	15.60	0	0	0	0	297.91
19:30	0.25	0	0	0	6.79	15	7.93	2.53	0	307.48
20:15	0.15	0	0	4.39	13.26	12.76	2.39	0	0	291.57
21:00	0.13	0	0	5.20	14.50	10.70	0	0	0	276.51
21:45	0.12	0	3.94	8.88	16.68	0	0	0	0	257.55
22:30	0.12	0	0	1.97	11.96	15	2.35	0	0	237.19
23:15	0.1	0	0	4.22	18.31	15	0	0	0	214.61
0:00	0.1	0	0	2.11	11.70	15	8.14	0	0	188.39
0:45	0.07	0	0	0	13.16	15	20	0	0	156.20
1:30	0.05	0	0	0	9.68	15	20	15.32	0	108.58
2:15	0.02	0	0	0	0	15	20	25	6	51.41

Table 4.2: Decisions at State (10,15,20,25,30)

Time	Price	CLV0	CLV1	CLV2	CLV3	DLV1	DLV2	DLV3	DLV4	Value
6:00	0.02	20	20	20	20	0	0	0	0	708.50
6:45	0.03	20	20	20	20	0	0	0	0	654.51
7:30	0.08	20	20	20	20	0	0	0	0	597.25
8:15	0.1	20	20	20	20	0	0	0	0	561.87
9:00	0.13	1.96	11.61	16.59	19.80	0	0	0	0	534.77
9:45	0.1	20	20	20	20	0	0	0	0	519.03
10:30	0.08	20	20	20	20	0	0	0	0	491.91
11:15	0.1	20	20	20	20	0	0	0	0	454.08
12:00	0.13	20	17.07	20	20	0	0	0	0	425.97
12:45	0.13	20	14.41	18.13	20	0	0	0	0	409.54
13:30	0.1	20	20	20	20	0	0	0	0	393.19
14:15	0.1	20	20	20	20	0	0	0	0	362.38
15:00	0.12	20	20	20	20	0	0	0	0	326.83
15:45	0.13	20	20	20	20	0	0	0	0	298.22
16:30	0.17	20	20	20	20	0	0	0	0	268.15
17:15	0.2	20	20	20	20	0	0	0	0	250.44
18:00	0.2	20	20	20	20	0	0	0	0	242.89
18:45	0.25	0	7.31	15.49	20	0	0	0	0	234.26
19:30	0.25	0	5.14	11.12	15.68	10.53	0	0	0	247.07
20:15	0.15	0	11.28	17.52	20	3.32	0	0	0	252.98
21:00	0.13	0	14.32	19.58	20	0.51	0	0	0	241.94
21:45	0.12	0	20	19.15	20	0	0	0	0	224.78
22:30	0.12	0	11.64	15.82	20	6.31	0	0	0	206.53
23:15	0.1	0	13.87	19.49	20	6.13	0	0	0	188.03
0:00	0.1	0	5	15.30	19.36	15	0	0	0	163.95
0:45	0.07	0	0	13.20	20	20	6.80	0	0	138.79
1:30	0.05	0	0	0	17.68	20	20	2.32	0	99.66
2:15	0.02	0	0	0	0	20	20	20	0	50.22

Table 4.3: Decisions at State (20,20,20,20,20)

Time	Price	CLV0	CLV1	CLV2	CLV3	DLV1	DLV2	DLV3	DLV4	Value
6:00	0.02	30	25	20	15	0	0	0	0	649.42
6:45	0.03	30	25	20	15	0	0	0	0	584.33
7:30	0.08	30	25	20	15	0	0	0	0	519.93
8:15	0.1	30	25	20	15	0	0	0	0	486.54
9:00	0.13	11.39	20.68	20	15	0	0	0	0	463.03
9:45	0.1	30	25	20	15	0	0	0	0	445.56
10:30	0.08	30	25	20	15	0	0	0	0	412.65
11:15	0.1	30	25	20	15	0	0	0	0	376.04
12:00	0.13	30	20.79	20	15	0	0	0	0	345.62
12:45	0.13	30	18.32	20	15	0	0	0	0	331.47
13:30	0.1	30	25	20	15	0	0	0	0	321.08
14:15	0.1	30	25	20	15	0	0	0	0	286.70
15:00	0.12	30	25	20	15	0	0	0	0	246
15:45	0.13	30	25	20	15	0	0	0	0	217.55
16:30	0.17	30	25	20	15	0	0	0	0	182.07
17:15	0.2	30	25	20	15	0	0	0	0	161.97
18:00	0.2	30	25	20	15	0	0	0	0	150.78
18:45	0.25	0	12.85	18.47	15	0	0	0	0	141.55
19:30	0.25	0	12.46	17.50	15	5.20	0	0	0	161.24
20:15	0.15	2.36	16.49	20	15	0	0	0	0	175.88
21:00	0.13	1.96	15.47	20	15	0	0	0	0	160.22
21:45	0.12	4.23	18.43	20	15	0	0	0	0	150.34
22:30	0.12	0	18.41	20	15	0	0	0	0	135.71
23:15	0.1	0	21.04	20	15	0	0	0	0	116.51
0:00	0.1	0	13.33	20	15	11.67	0	0	0	98.97
0:45	0.07	0	0	16.68	15	25	3.32	0	0	72.07
1:30	0.05	0	0	0	15	25	20	0	0	44.72
2:15	0.02	0	0	0	0	25	20	15	0	12.80

Table 4.4: Decisions at State (30,25,20,15,10)

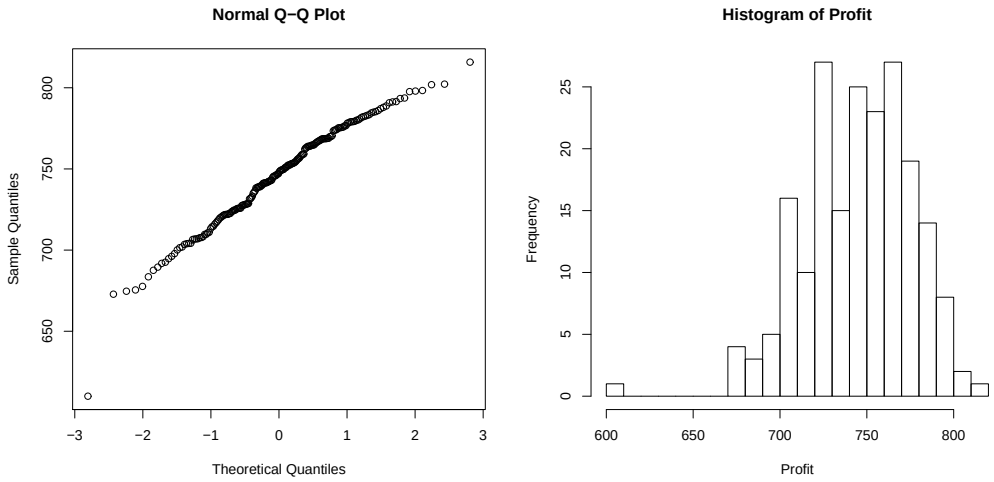

Figure 4.3: Simulated daily profits (4 bars of energy, with sell-back)

of level m batteries to charge, and DLVm is the number of batteries to discharge. Figure 4.4 shows one of the 200 simulations. From the tables, we see that when selling electricity back to the grid, the policy favors discharging the emptier batteries first, which is for the same reason as why we give priority to charging fuller batteries. However, the batteries are only discharged occasionally, probably because the penalty for losing a customer is too higher compared to the profit from selling electricity. We see that batteries are discharged at the last few stages, but that is because we set $J_T(x_T) = 0$, and the decisions would not be so if we included salvage value. Therefore, the idea of selling electricity back to the grid is in general not profitable for the battery exchange stations with the above setting. The true profitability of selling electricity, however, needs to be verified with real market data, and it is beyond the scope of this dissertation.

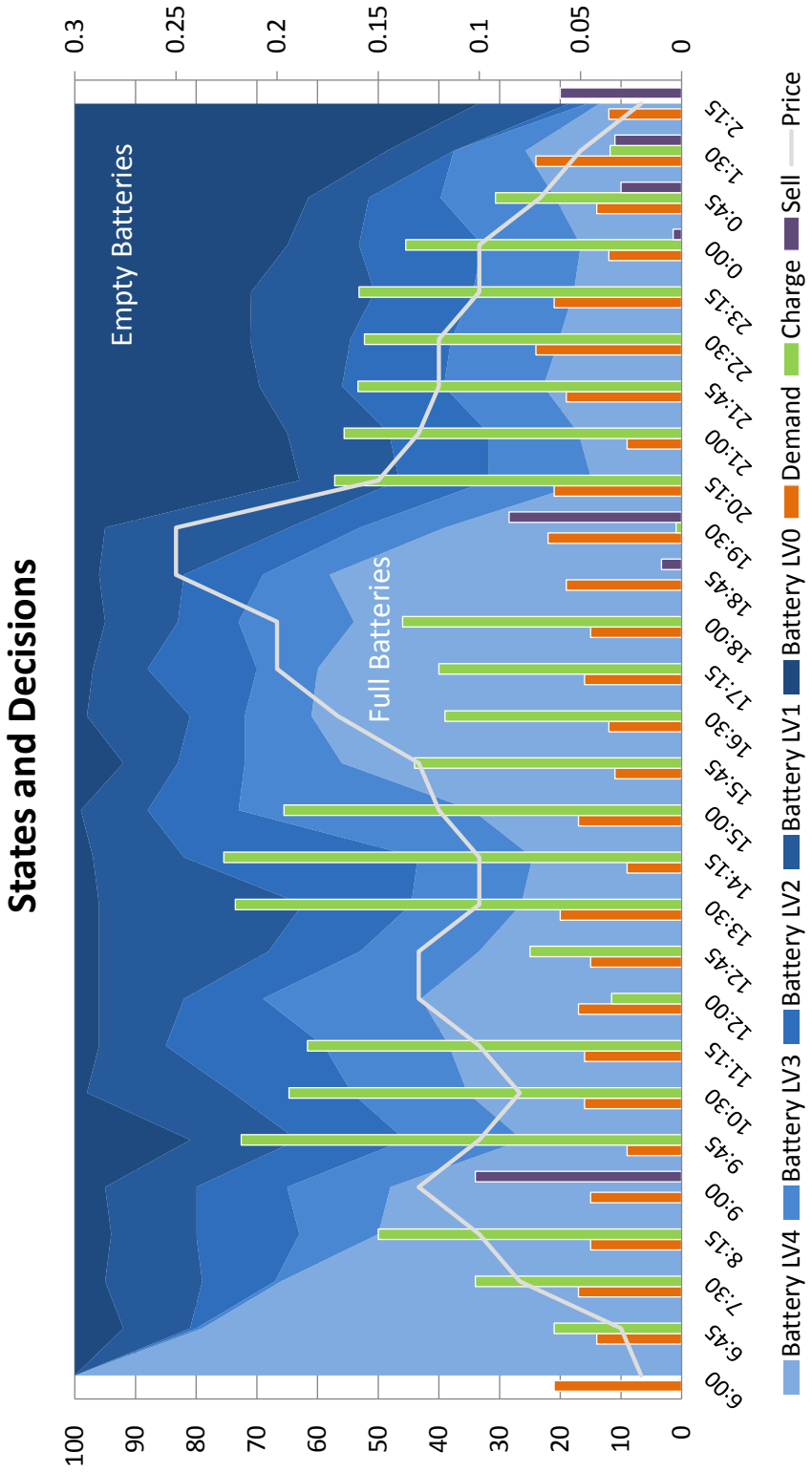

Figure 4.4: Simulation of a day (4 bars of energy, with sell-back)

Chapter 5

Conclusion and Future Research

Motivated by the optimal charging problem from battery exchange station management, we developed the Adaptive Convex Enveloping method for solving convex stochastic dynamic programs with multidimensional continuous state and decision variables, and especially for those with nontrivial constraints and strongly coupled state variables.

In Chapter 2, we reviewed existing dynamic programming algorithms, among which, only the class of simulation-based algorithms could potentially solve the problems that we are interested in. However, in practice the simulation-based algorithms will most likely fail, because the two core steps of the algorithms — value function approximation and policy optimization — cannot be done reliably. The value functions are fitted with aggressive approximation functions using simulated data that has large noise, which almost surely brings overfitting; the policy is difficult to optimize for problems with multidimensional decision variables: if the policy is implicitly given by the Bellman's equation, we will find that the approximation of the value function usually does not have a form that can be well handled by existing mathematical pro-

gramming algorithms; otherwise, if we explicitly give the policy a heuristic and/or parametric form, it may reduce the difficulty of the optimization, but it raises the question of why such form is (roughly) optimal.

In Chapter 3, we introduced the Adaptive Convex Enveloping method for solving multidimensional DPs. We believe that the success to solving multidimensional DPs lies in achieving reliable approximation and optimization simultaneously. Therefore we avoided nonconvex optimization, which is still an open question in mathematical programming, and focused on DPs with convex value functions. We chose to use supporting hyperplanes for the approximation, which is convexity-preserving, and works well with the optimization libraries. On the other hand, the construction of supporting hyperplanes also relies on the accurate gradient information that convex optimization algorithms give for free. Therefore the supporting hyperplane approximation and convex programming form a terrific combination.

The algorithm of Adaptive Convex Enveloping proceeds backwards in stage and enjoys superior data quality, which is an advantage over simulation-based algorithms. To guarantee approximation quality, and to use supporting hyperplanes efficiently, we used a recursive partitioning algorithm, which adds a new supporting hyperplane only where the approximation is not good enough. When the algorithm terminates, we have bounds for the approximation error. Furthermore, the error bounds are clear numbers that either are directly known, or can be simulated. Thus the user can have confidence in the performance of the solution.

In Chapter 4 we demonstrated the capability of Adaptive Convex Enveloping with the optimal charging problem. We solved two versions of the problem: one in standard setting as introduced at the beginning of this dissertation, with 4 state variables constrained in a 3-dimensional subspace; and the other one with 5 state variables in a

4-dimensional subspace, with the additional option to discharge batteries and sell the electricity back to the grid, which is an operation that the "smart grid" technology would like to see. For the first example, we achieved "statistical optimality", as the simulated data does not support the hypothesis that the error is not zero. This is a very strong result, as the other approximation algorithms only claim sub-optimality without knowing how sub-optimal in most cases. For the second example, we demonstrated that it does not require the tolerance to be very strict to achieve a high quality solution. We relaxed the tolerance to a degree that would give an error range that was about 10% of the scale of the target if we only used the conservative error bounds. The better bound given by simulation showed that the actual error arrange was barely over 1% of the scale of the target, which is a very satisfying result for multidimensional DP. Since using lenient tolerances greatly reduces the computational burden, it is highly recommended to do so when using ACE.

The numerical examples also give insights to optimal charging policies. On the one hand, the optimal policy favors charging fuller batteries first; one the other hand, the policy also balances the numbers of batteries at each level, so that we can have enough batteries to charge when a good time comes. These rules of thumb can be used as guidance when designing heuristic policies for large scale problems that we cannot afford to solve accurately with ACE. In addition, if we discharge batteries to sell electricity, the policy suggests that we start from the batteries with less energy left. However, as the solution shows, selling the electricity may not be profitable, especially when the total inventory of batteries at the station is tight compared to the demand.

So far, we have been solving the optimal charging problem with known electricity prices. In deregularized markets, one may want to incorporate random prices into the model. Naturally, price could be added as a component of the state variable, but

doing so makes the value functions non-concave. ACE is designed for convex (and concave) DPs, so it does not directly solve the non-concave optimal charging problem with random prices. However, if we look at a "slice" of the value function by fixing the price, the "slice" $J_t((x_t, p_t) \mid p_t)$ is still concave. Therefore, we can discretize the price to a set of values $\{p_t^i\}_i$, and approximate each $J_t^i(x_t) \triangleq J_t((x_t, p_t) \mid p_t^i)$ with supporting hyperplanes. Assuming that the variation of the electricity price is Markovian, we could modify (3.4) as

$$
\begin{aligned}
J_t^i(x_t) = & \max_{s_t, u_t, \{J_{t+1}^k\}} \sum_{k=1}^{K} P(w_t^k)[-c((s_t, p_t^i), u_t, w_t^k) + J_{t+1}^k] \\
& \text{s.t.} \quad g_t(s_t, u_t) \leq 0, \\
& \qquad J_{t+1}^k \leq J_{t+1}^{i'}(x_{t+1}^j) + \nabla J_{t+1}^{i'}(x_{t+1}^j)^T (x_{t+1}^k - x_{t+1}^j), \text{ for all } j, k \text{ and } p_{t+1}^{i'} = p_{t+1}^k, \\
& \qquad s_t = x_t,
\end{aligned}
$$

where the upper bound constraints are generated according to the sample. We plan to explore this problem in the future work.

References

D. Adelman and A.J. Mersereau. Relaxations of weakly coupled stochastic dynamic programs. *Operations Research*, 56(3):712–727, 2008.

J. Baxter and P.L. Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. Technical report, Citeseer, 1999.

R.E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

J.F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(1):238–252, 1962.

D.P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

D.P. Bertsekas and J.N. Tsitsiklis. Neuro-dynamic programming. 1996.

J.R. Birge. Decomposition and partitioning methods for multistage stochastic linear programs. *Operations Research*, pages 989–1007, 1985.

C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer New York, 2006.

R. Bixby, Z. Gu, and E. Rothberg. Gurobi optimization, 2010.

J.A. Boyan. Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2):233–246, 2002.

S.J. Bradtke and A.G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1):33–57, 1996.

E. Campos-Náñez. *Adaptive on-line optimization of Markov reward processes with application to pricing of multiclass loss network services*. PhD thesis, University of Virginia, Charlottesville, VA, 2003.

E. Campos-Náñez. Decentralized algorithms for adaptive pricing in multiclass loss networks. *IEEE/ACM Transactions on Networking*, 18(3):830–843, 2010.

D.P. De Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Operations Research*, pages 850–865, 2003.

D.P. De Farias and B. Van Roy. On constraint sampling in the linear programming approach to approximate dynamic programming. *Mathematics of Operations Research*, pages 462–478, 2004.

G. de Ghellinck. Les problemes de decisions sequentielles. *Cahiers du Centre d¡ ¯Etudes de Recherche Opérationnelle*, 2(2):161–179, 1960.

E.V. Denardo. On linear programming in a markov decision problem. *Management Science*, pages 281–288, 1970.

G. Deng and M.C. Ferris. Neuro-dynamic programming for fractionated radiotherapy planning. *Optimization in Medicine*, pages 47–70, 2008.

C. Derman. *Finite State Markovian Decision Processes*. Academic Press, Inc., 1970.

Y. Engel, S. Mannor, and R. Meir. Sparse online greedy support vector regression. *Machine Learning: ECML 2002*, pages 1–3, 2002.

A.V. Fiacco and J. Kyparisis. Convexity and concavity properties of the optimal value function in parametric nonlinear programming. *Journal of optimization theory and applications*, 48(1):95–126, 1986.

G.A. Godfrey and W.B. Powell. An adaptive, distribution-free algorithm for the newsvendor problem with censored demands, with applications to inventory and distribution. *Management Science*, pages 1101–1112, 2001.

M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51(1):141–202, 1991.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. New York: Springer, 2009.

J.T. Hawkins. *A Langrangian decomposition approach to weakly coupled dynamic optimization problems and its applications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.

J.H. Holland. Escaping brittleness: the possibilities of general purpose learning algorithms applied to parallel rule-based system. *Machine learning*, pages 593–623, 1986.

A. Hordijk and LCM Kallenberg. Linear programming and markov decision chains. *Management Science*, pages 352–362, 1979.

V.R. Konda and J.N. Tsitsiklis. On actor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166, 2003.

M.G. Lagoudakis and R. Parr. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149, 2003.

A.S. Manne. Linear programming and sequential decisions. *Management Science*, pages 259–267, 1960.

P. Marbach and J.N. Tsitsiklis. Gradient-based optimization of markov reward processes: Practical variants. Technical report, 2000.

P. Marbach and J.N. Tsitsiklis. Simulation-based optimization of markov reward processes. *Automatic Control, IEEE Transactions on*, 46(2):191–209, 2001.

P. Marbach and J.N. Tsitsiklis. Approximate gradient methods in policy-space optimization of markov reward processes. *Discrete Event Dynamic Systems*, 13(1): 111–148, 2003.

D. Ormoneit and S. Sen. Kernel-based reinforcement learning. *Machine Learning*, 49 (2-3):161–178, 2002.

W.B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Wiley-Blackwell, 2007.

M. Puterman. *Markov Decision Process*. New York: John Wiley & Sons, 1994.

A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

P.J. Schweitzer and A. Seidmann. Generalized polynomial approximations in markovian decision processes. *Journal of mathematical analysis and applications*, 110(2): 568–582, 1985.

H.P. Simão, J. Day, A.P. George, T. Gifford, J. Nienow, and W.B. Powell. An approximate dynamic programming algorithm for large-scale fleet management: A case application. *Transportation Science*, 43(2):178–197, 2009.

R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

R.S. Sutton. Gain adaptation beats least squares. In *Proceedings of the 7th Yale Workshop on Adaptive and Learning Systems*, pages 161–166, 1992.

G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

H. Topaloglu. Using lagrangian relaxation to compute capacity-dependent bid prices in network revenue management. *Operations Research*, 57(3):637–649, 2009.

M.A. Trick and S.E. Zin. A linear programming approach to solving stochastic dynamic programs. *Unpublished manuscript*, 1993.

J.N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1):59–94, 1996.

B. Van Roy, D.P. Bertsekas, Y. Lee, and J.N. Tsitsiklis. A neuro-dynamic programming approach to retailer inventory management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, volume 4, pages 4052–4057. IEEE, 1997.

R.M. Van Slyke and R.J.B. Wets. L-shaped linear programs with applications to optimal control and stochastic programming. *SIAM Journal on Applied Mathematics*, pages 638–663, 1969.

V. Vapnik. *Statistical Learning Theory*. New York: Wiley, 1998.

C.J.C.H. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

X. Xu, D. Hu, and X. Lu. Kernel-based least squares policy iteration for reinforcement learning. *Neural Networks, IEEE Transactions on*, 18(4):973–992, 2007.

Appendix - Discussion on the Data Structures

An algorithm needs to be fast to be useful, so here we introduce the data structures that we used for ACE. These data structures may not be most efficient, but could serve as a reference for new work.

We have four classes (C++ data structures): `Plane`, `Plane_vector`, `Section` and `Section_list`. The data structures and designs below are based on the examples in this dissertation, where every optimization problem is linear. The `GRB` variables are for the linear optimization library GUROBI, and should be modified when using other libraries.

Class Plane

```
class Plane
{ public:

    Plane(const vector<double>& X,const  double& Value,
          const  vector<double>& Gradient); // constructor
    ~Plane(void); // destructor
```

```
    vector<double> x; // location of the point of tangency
    double value; // function value at x
    vector<double> gradient; // function gradient at x
    double gradientTx; // inner product of gradient and x
    static int stateVarDim; // dimension, # of variables
}
```

The `Plane` class is the structure that stores supporting hyperplanes. It stores x, $J(x)$ and $\nabla J(x)$, which must be provided when constructing a supporting hyperplane. It also stores $x^T \nabla J(x)$ with `gradientTx`, as the inner product is frequently used. The `stateVarDim` is the dimension of the state variable.

Class Plane_vector

```
class Plane_vector :
      public std::vector<Plane>
{ public:
    Plane_vector(void);
    ~Plane_vector(void);
    void read(string filename, int t); // t for stage
    void write(string filename, int t) const; // t for stage
}
```

The `Plane_vector` class inherits from the vector class from Standard Template Library (STL) of `C++`, and is the container of `Planes`. We maintain one `Plane_vector` per stage to store all the supporting hyperplanes that have been constructed. We

only added two functions for reading and writing supporting hyperplane information from/to files.

Class Section

```
class Section
{ public:

    Section(const Plane_vector & SHPs,
           const vector<int> & ids); // constructor
    ~Section(void); // destructor
    vector<int> associatedVertices;
    vector<double> worstPoint;
    double maxError;
    vector<double> convexCoeff;
    static int stateDomainDim; // dimension of the state domain
    static GRBEnv* env;
    static GRBModel* model;// for finding the worst point
    static vector<GRBVar> X;
    static vector<GRBVar> alpha;
    static GRBVar J;

}
```

A `Section` is the convex hull of $p+1$ vertices in a p-dimensional space, and is constructed with $p+1$ supporting hyperplanes from the `Plane_vector SHPs` with `ids` as their positions in the vector, which are then stored in `associatedVertices`. The constructor solves (3.9) to find the maximum potential error `maxError` and the point `worstPoint` where the error would occur. It also stores $\{\alpha_j\}_{j=1}^{p+1}$ in `convexCoeff` for

determining whether `worstPoint` is on the boundary, which is needed when partitioning the section.

Since setting up an optimization model is very time consuming, to speed up the computation, one would not want to set up the same optimization model again and again. We set the model of (3.9) as `static`, so that it is shared by all the `Sections`. In the `main()` function, we set up everything of (3.9), except those related to the vertices. When we construct a `Section`, we only needed to fill in the coefficients of $\{\alpha_j\}_{j=1}^{p+1}$ in the objective function, and link x_t and $\{\alpha_j\}_{j=1}^{p+1}$ with $x_t = \sum_{j=1}^{p+1} \alpha_j x_t^j$. After we solved (3.9), we remove the link between x_t and $\{\alpha_j\}_{j=1}^{p+1}$ to make the model general again by deleting the constraint $x_t = \sum_{j=1}^{p+1} \alpha_j x_t^j$. In this way, we reuse the optimization model in every `Section`, making the algorithm efficient.

Class Section_list

```
class Section_list :
       public std::list<Section>
{
public:
    Section_list(const Plane_vector & SHPs, int stage);
          // constructor
    ~Section_list(void); // destructor
    void initiate(Plane_vector & SHPs);
          // initiate the first Section
    void refine(double tol, int Budget, Plane_vector& SHPs);
private:
    GRBEnv* env;
```

```
        GRBModel* model; // the Bellman's equation
        vector<GRBConstr> states;
        vector<GRBVar> batteryCharge; // the decision variable
        int stage;
    }
```

The `Section_list` class inherits from the list class from Standard Template Library (STL) of `C++`, and is the container of `Sections`. We maintain one `Section_list` per stage to store all the partitions. The constructor of `Section_list` sets up the optimization model, which is also the Bellman's equation (3.4). Then `initiate()` is called in the `main()` function to add the initial supporting hyperplanes and form the first `Section`. After initiation is done, `refine()` is called to add new supporting hyperplanes and partition the `Sections` until the maximum potential error is less than the tolerance everywhere.

Again, we minimize the time spent on setting up the model for (3.4) by reusing it. Observe that the Bellman's equations for the `Sections` are the same, with the exception of different values of the state variable. Thus, when we add a new supporting hyperplane, we only need to modify some parameters of the model without having to rebuild the main structure, and the same optimization model is used from the construction of the `Section_list` to the destruction of it, i.e. the entire stage. Also observe that in the optimization model, the state variable is the "right hand side" of some constraints. Since the optimizations are linear in our example, and by our design the state point tends to be close to the one where we just added the last supporting hyperplane, we could continue from the previous solution and solve (3.4) easily for the new point by using dual simplex update, which is much faster than solving from scratch.