Spline Approximations to Value Functions: A Linear Programming Approach

Michael A. Trick* Stanley E. Zin†

May 1995
Revised: January 1997

Abstract

We review the properties of algorithms that characterize the solution the Bellman equation of a stochastic dynamic program, as the solution to a linear program. The variables in this problem are the ordinates of the value function, hence, the number of variables grows with the state space. For situations when this size becomes computationally burdensome, we suggest the use of low-dimensional cubic-spline approximations to the value function. We show that fitting this approximation through linear programming provides upper and lower bounds on the solution to the original "large" problem. The information contained in these bounds leads to inexpensive improvements in the accuracy of approximate solutions.

1 Introduction

For a large (and for economists, an interesting) class of nonlinear stochastic dynamic programming problems, the Bellman equation can be characterized by

*GSIA, Carnegie Mellon University. Supported in part by Office of Naval Research Grant, Grant N00014–92–J–1387.

†GSIA, Carnegie Mellon University and NBER.

a set of linear restrictions on the value function. Trick and Zin (1993) propose linear programming algorithms that exploit this feature to find the fixed point of Bellman's equation, *i.e.*, the optimal value function. For a finite and discrete problem, or for a finite discretization of a continuous problem, they show how so-called *constraint-generation* algorithms can (1) provide order-of-magnitude speed gains over more traditional value-function iteration algorithms, and (2) provide increased accuracy (without increased complexity) through the *adaptive grid generation* that the linear programming approach affords. The problem remains, however, that for large problems, benefits of constraint generation are still dwarfed by the "curse of dimensionality." This paper investigates new algorithms that allow the linear programming approach to be applied to much larger problems. Moreover, these algorithms also allow for a form of adaptation that provides increases in accuracy at a very small cost.

We reduce the size of the problem by assuming that an approximation to the value function lies in a space of flexible functional forms, namely, cubic splines. Cubic splines have a number of attractive qualities. For example, they can greatly reduce the dimensionality of the problem: a good approximation can be obtained from a spline with a much smaller number of parameters than the number of unrestricted value-function ordinates needed to obtain a comparable approximation. However, the main feature for our purposes is that, even though these functions are highly nonlinear in the state variables, they are linear functions of their parameters. Written in terms of splines, the *approximation* to the discrete Bellman equation is linear in the parameters. Therefore, these parameters, and hence the approximate value functions, can be found using *constraint-generation* linear programming techniques. The benefits of solving this problem with constraint generation rather than, say, *least squares* are (1) computational speed, (2) ease of imposing additional restrictions such as monotonicity of the value function or its derivatives, and most importantly, (3) the linear programming spline approximation cannot lie below the exact solution to the discrete value function.

This last feature motivates the accuracy enhancing adaptations we propose. Increased accuracy obtains as follows. Take an arbitrary (and perhaps coarse) partition of the finely discretized state space. Use the constraint generation algorithm to obtain a spline approximation of the value function. This approximate value function must lie on or above the discrete value function (*i.e.*, the exact value function calculated directly on the discrete grid). Calculate the optimal actions implied by this approximation. Calculate the unrestricted value-function ordinates implied by these approximate actions. Note that since these actions are feasible but not necessarily optimal (given the approximate nature of the value function they were constructed from), these values must lie on or *below* the discrete value function. We now have an upper and a lower bound on this function. Check where these bounds "differ the most" and adapt the spline accordingly, *e.g.*, make the partition finer at that location, move the partition, or increase the order of the polynomial over that partition. Use constraint generation to find a new approximation over this new partition of the state space, and iterate. Stop when the approximation achieves the desired degree of accuracy. The complexity of the problem does not change dramatically as this adaptation proceeds so that accuracy can be improved at a very low cost.

Many of the benefits we note for spline approximations apply more generally to other functional approximations. The essential property is the linearity in the unknown parameters (but not the state variables). For example, Mrkaic, Trick and Zin (1997) employ neural-network approximations in a very similar fashion. Schweitzer and Seidmann (1985) also use linear programming (among other techniques) to compute generalized polynomial approximations to the value function.

We develop our algorithms in the context of the neoclassical stochastic growth model detailed in the next section. We compare the speed of linear programming algorithms to conventional value-function iterations in Section 3. In Section 4 we introduce our spline approximations. The value-function bounds generated by these approximations are derived in Section 5 and are then used to enhance the accuracy of the solution in Section 6.

2 The Linear Programming Approach

We begin by outlining the basic linear programming approach to solving stochastic dynamic programming problems in the context of the stochastic growth model. For time periods $t = 1, 2, \ldots$, the production technology is given by

$$y_t = z_t f(k_t) \ ,$$

where y_t is output produced in period t, k_t is the stock of capital available at the beginning of period t, f is a well-behaved production function and $\{z_t\}$ is a stationary stochastic process representing the technology shock. The social planner ranks random consumption sequences, $\{c_t\}$ according to the expected utility index

$$U_0 = E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \quad ,$$

where $0 < \beta < 1$ is the discount factor, u is a well-behaved within-period utility function, and E_0 denotes the period-0 conditional expectations operator. The planner chooses a sequence of state-contingent consumption and capital pairs $\{c_t,\ k_{t+1}\}_{t=1}^{\infty}$, to maximize utility subject to the constraint

$$c_t + k_{t+1} - (1-\delta)k_t = z_t f(k_t) \quad ,$$

where $0 < \delta \leq 1$ is the rate of depreciation of capital. Implicit in this constraint is a timing assumption that allows the planner to observe the realization of z_t before making the period-t consumption/investment decision.

The dynamic programming approach to solving this problem uses the Bellman equation

$$v(k,\ z) = \max_{k' \in \mathcal{A}(k,\ z)} \left\{ u\left(zf(k) + (1-\delta)k - k'\right) + \beta E\left[v(k',\ z') \mid k,\ z\right]\right\} \quad , \qquad (1)$$

where $v(k,\ z)$ is the value of the optimal plan given a capital stock k and technology shock z, and $\mathcal{A}(k,\ z)$ is the set of feasible actions satisfying $0 \leq k' \leq zf(k) + (1-\delta)k$. Given v, optimal policies obtain from the maximization on the right-hand side of (1). Closed-form solutions for optimal policies and values are generally unavailable. This motivates the interest in solutions to numerical examples of these economies.

We restrict our attention to a finite discrete-state version of this economy. That is, capital and the technology shock are assumed to line in finite sets defined respectively as

$$\mathcal{K} = \left\{k^{(1)},\ k^{(2)},\ \ldots,\ k^{(n_k)}\right\} \quad ,$$

and

$$\mathcal{Z} = \left\{z^{(1)},\ z^{(2)},\ \ldots,\ z^{(n_z)}\right\} \quad .$$

The stochastic process for the technology shock is a first-order Markov chain with transition probabilities given by

$$\pi_{ij} = \text{Prob}\left(z_t = z^{(j)} \;\middle|\; z_{t-1} = z^{(i)}\right) \quad .$$

With this additional notation, we can write equation (1) as

$$v_{ij} = \max_{a \in \mathcal{A}_{ij}} \left\{ u_{ija} + \beta \sum_{l=1}^{n_z} \pi_{jl} v_{al} \right\} \quad , \tag{2}$$

where

$$v_{ij} = v(k^{(i)},\ z^{(j)}) \quad ,$$

$$u_{ija} = u\left(z^{(j)} f(k^{(i)}) + (1-\delta)k^{(i)} - k^{(a)}\right) \quad ,$$

and

$$\mathcal{A}_{ij} = \left\{ a \,\middle|\, 1 \leq a \leq n_k, \ \text{ and } \ z^{(j)} f(k^{(i)}) + (1-\delta)k^{(i)} - k^{(a)} > 0 \right\} \quad .$$

Let n_{ij} denote the number of elements in the set $\mathcal{A}_{ij}$.

The maximization in (2) implies a set of inequalities that must be satisfied by the value function:

$$v_{ij} \geq u_{ija} + \beta \sum_{l=1}^{n_z} \pi_{jl} v_{al} \quad , \tag{3}$$

for all i, j, and $a \in \mathcal{A}_{ij}$. It is well-known (*e.g.*, Bertsekas (1976) and Ross (1983)) that finding the smallest set of v_{ij}'s that satisfy these constraints amounts to solving a linear program of the form

$$\min \sum_{ij} v_{ij} \quad , \tag{4}$$

subject to (3).

Linear programming techniques are well know for stochastic dynamic programs, but are generally dismissed as inefficient. For instance, Puterman (1994, p. 230), relying on work of Koehler (1976) states that "modified policy iteration is considerably more attractive computationally than the simplex–method–based linear programming codes," citing times worse by a factor of 10. He goes on to state that despite the difficulties inherent in linear programming, it might be considered due to "the facility of sensitivity analysis" and "the ability to include additional constraints". Our results show that these two possibilities, together with much faster current linear programming codes, are sufficient to reverse the speed comparison in many cases.

Constraint generation is a technique for solving linear programs with a large number of constraints. Rather than have a computer code attempt to solve a large linear program, the solution procedure begins with a small number of constraints. The linear program over this subset of constraints is solved. If the result is feasible to all of the other constraints, then the incumbent solution is optimal. Otherwise, some of the constraints violated by the solution are added to the linear program and the linear program is resolved. This process iterates until all constraints are satisfied.

We adapt the constraint generation technique to the problem of solving the discrete stochastic dynamic programs. We will begin with a small number of constraints, which correspond to feasible actions, and add constraints only when the current solution violates them. In the examples below we solve linear programs in more than $8,000$ variables (value-function ordinates) subject to more than 18.3 million constraints (restrictions implied by Bellman's equations). Moreover, we are able to accomplish this is a little more than an hour and a quarter of workstation time. In addition to solving large problems, constraint generation provides speed gains over solving the full linear program for a number of reasons: (1) By knowing that the optimal solution needs only one binding constraint (action) for each state, we can add only the most violated constraint for each state, rather than possibly a large number of unneeded constraints; (2) We can precalculate common terms used in multiple constraints; and (3) We can ignore entire states, and only add them when we have a good estimate of where their optimal actions occur. As we shall see, these reasons are sufficient for orders of magnitude speedup over the full linear program.

3 Speed Comparisons

In this section, we compare the speed of constraint-generation algorithms to standard value-function iteration. We do not make direct comparisons with a full range of competing algorithms. However, since value-function iteration is frequently used as a benchmark in other studies, these comparisons are implicit.

The numerical example we use for speed comparisons is as follows. The exogenous technology shock is a two-state Markov chain, with a high state of $z_2 = 1.377$ and a low state of $z_1 = 0.726$. The transition matrix is

$$\Pi = \begin{bmatrix} 0.975 & 0.025 \\ 0.025 & 0.975 \end{bmatrix} .$$

This is the high variance model in Christiano (1990) and corresponds to the log of the shock having a mean of zero, a variance of 0.1, a high degree of persistence, and a symmetric ergodic distribution. We choose simple power functions for the production function, $f(k) = k^{\alpha}$, and the utility function, $u(c) = c^{\rho}/\rho$. The share parameter, α is set at 0.33 and the depreciation rate, δ, is set at zero. The risk aversion parameter, ρ, is 0.5. The discount factor, β, is set to 0.98 for most cases, however, to evaluate the sensitivity of our results to the value of β, we also conduct experiments where β varies over the values $\{0.75, 0.85, 0.9, 0.95, 0.98, 0.99, 0.999\}$.

The discrete grid over the capital stock is equally spaced with end points chosen so that roughly 10% of the points lie below $k^*(z_1)$ and roughly 10% of the points lie above $k^*(z_2)$ defined by

$$k^*(z_1) = \left[\frac{\beta\alpha z_1}{(1-(1-\delta)\beta)}\right]^{\frac{1}{1-\alpha}} \qquad k^*(z_2) = \left[\frac{\beta\alpha z_2}{(1-(1-\delta)\beta)}\right]^{\frac{1}{1-\alpha}} .$$

The quantities $k^*(z_1)$ and $k^*(z_2)$ are the deterministic steady–state values for equilibrium capital when z_1 and z_2, respectively, are permanent features of the fixed technology. This somewhat arbitrary choice of endpoints for the capital grid provides an automatic way of ensuring that the solution has a well dispersed ergodic set when we vary the parameter β. If we were more interested in the exact solutions to this problem rather than the properties of computational algorithms for solving this problem, then we would want to be more careful in choosing these points and perhaps tailor these choices to each numerical version of the model being solved.

Starting values for value iteration are chosen as follows. For each point in the state space, we calculate the steady-state utility as if the smallest feasible capital stock was the deterministic steady state. This value, $u(k)/(1-\beta)$, forms the initial value from which we iterate until convergence. Starting values for value iteration are an extremely important determinant of the speed of the algorithm: the better the starting values, the faster the algorithm. The method we adopt for choosing starting values is a simple automatic method that does not require a lot of *ex ante* information about the solution, hence, it allows for reasonably fair comparisons with other methods. In particular, we take comparable steps when starting up the constraint generation linear programming algorithm described below. Later we will discuss the possibility for grid generation to provide more accurate starting values and a commensurate increase in speed. The convergence criterion is $\max_{(i,j)} |v_{ij}^{m+1} - v_{ij}^{m}| < 0.000001$.

The following experiments were performed on an HP 720 workstation with

32MB memory running HP–UX 8.0. All of the computer codes were written in "C" and compiled with the operating system's "cc" compiler. The linear programs were solved with "CPLEX", a commercial code widely available for a number of computer systems. One idiosyncracy of CPLEX is the relative ease of adding variables to a linear program, rather than constraints. As a result, in anticipation of our use of constraint-generation algorithms, we find it more convenient to always solve the dual of 4, and implement our algorithms with an equivalent *variable-generation*.

Our intention in these tests was to generate conclusions applicable to more than just the simple growth model. To this end, we tried to exploit only those features of the model that have wide applicability. Therefore, all of these codes precalculated terms when possible, provided the space required was no more than $n_k n_z$. This meant that codes could not precalculate all of the u_{ija} but they could precalculate the (expensive) term that depends only on i and j ($(1-\delta)k_i + k_i^\alpha z_j$ in this case). Similarly, to update after each iteration of value iteration or to generate constraints in constraint generation, the term $\beta \sum_{j'} \pi_{jj'} v_{aj'}$ needs to be calculated only once for each (a, j). Other aspects specific to the growth model, such as the curvature of the utility function and the near–linearity of the value function for certain parameter values are not explicitly exploited. In particular, we enumerate all feasible actions when determining the optimal action.

Figure 1 plots the computational speed in seconds against the size of the grid for the capital stock, for value iteration and linear programming solutions to the base-case growth model. It is clear from this figure that linear programming provides dramatic increases in speed. Moreover, computational time appears to be growing much more slowly for linear programming than for value iteration. For the smallest problem in this figure, $n_k = 33$, linear programming is almost 80 times faster than value iteration (0.2 seconds compared to 15.86 seconds). For the largest problem in this figure, $n_k = 513$, linear programming is approximately 13 times faster than value iteration (297.11 seconds compared to 3781.7 seconds). These results indicate that standard linear programming can provide at least an order-of-magnitude improvement over standard value-function iteration for problems of this size. The primary drawback of standard linear programming is the large amount of memory needed to solve large problems. However, as discussed above, constraint generation algorithms alleviate much of this memory burden. Having established the benefits of the linear programming approach over value iteration, we now turn to refinements on the linear programming algorithm, namely, constraint generation.

Figure 2 compares the relative performance of standard linear programming

Figure 1: Value Iteration and Linear Programming Comparisons

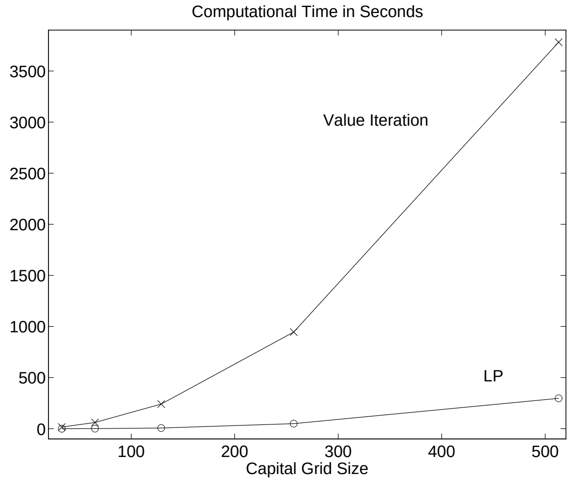
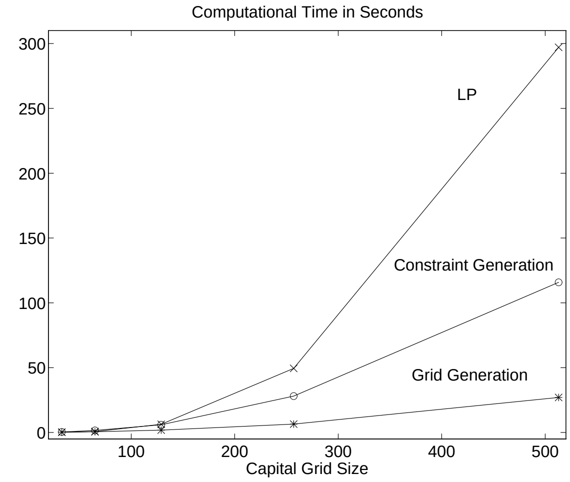


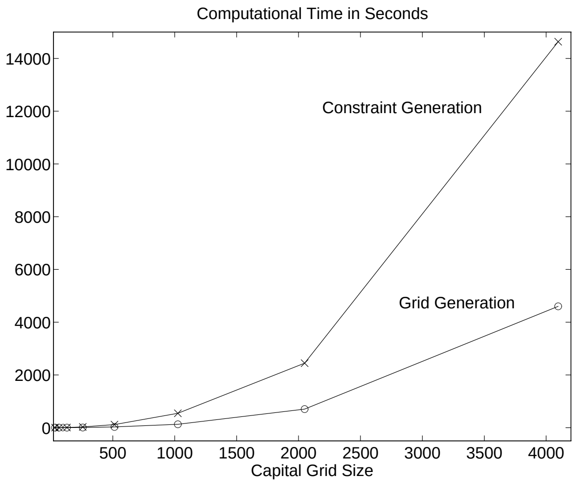
Figure 2: LP, Constraint and Grid Generation Comparisons



to the constraint–generation algorithm for solving linear programs. As described above, constraint generation begins by solving the linear program subject to a subset of the constraints, then repeatedly adding in violated constraints and resolving, until all constraints are satisfied. For the problem at hand, we implement this algorithm by beginning with the linear program that includes only the constraints defined by the smallest feasible action for each point in the state space. At each iteration, for each state (i, j) we add the constraint corresponding to the action, a, that has the largest value of $u(i, j, a) + \sum_{j'} \beta \pi_{jj'} v_{k-1}(a, j')$, unless this constraint is already in the linear program. When each constraint is satisfied to within 0.000001, we conclude that the algorithm has converged. For the smallest problem in the figure, $n_k = 33$, constraint generation is actually slower than straight linear programming (0.4 compared to 0.1), however, for the largest problem in this figure, $n_k = 513$, constraint generation is more than two and a half times faster than straight linear programming (115.82 seconds compared to 297.11). Speed is not the only motivation for constraint generation. Of even greater benefit is the ability to solve very large problems (as in Figure 3).

Along with standard linear programming and constraint generation, Figure 2 contains results for an algorithm that we term grid generation. The basic idea behind this algorithm is as follows. We begin by solving the problem using only a subset of states. We use the solution to the subset to generate good starting solutions to a larger set of states. We continue until we have solved for all the states. In this case, we begin by solving the problem corresponding to $n_k = 16$, choosing these 16 points equally spaced over the entire large grid. When we have found the solution to this small problem, we then add new points to the capital grid halfway between each of the current points (note that these new points are also on the large grid), doubling the grid size in the process. For each point that we add, we include three new constraints: the constraint corresponding to a guess for the optimal action for the new point (computed as the average of the optimal actions of its neighbors) and the points on the $n_k = 32$ grid adjacent to this guess. We also include new constraints corresponding to the actions on this finer grid that are adjacent to the optimal actions from the $n_k = 16$ problem, since these are the newly introduced actions that are most likely to be close substitutes for the original actions. We then optimize this larger problem completely over the set of capital points (using constraint generation) before adding new points. New points are added in exactly the same way, doubling the grid size each time, until the full problem is completely solved. Since, with constraint generation, we are already solving a sequence of larger and larger linear programs, increasing the grid size in this way is a natural extension. This

Figure 3: Larger Problem

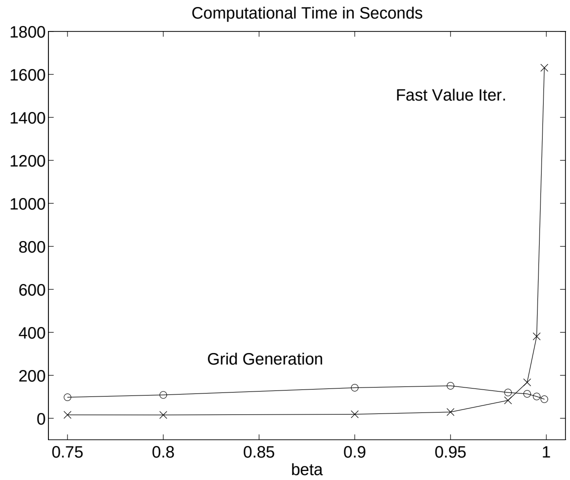


approach is similar in spirit to Whitt (1978, 1979) who examined the theoretical aspects of successively finer grids and showed it is possible to get upper and lower bounds on the value of a finer grid based on the solution to a coarser grid.

As we see in Figure 2, grid generation provides a speed gain over simple constraint generation comparable to that of constraint generation over standard linear programming. For the largest sized problem in this figure, grid generation is more than 4 times faster than constraint generation (27 seconds compared to 115.82). Grid generation is, therefore, more than 10 times faster than standard linear programming. Since memory demands are not as great for these two algorithms (relative to standard linear programming), we can solve larger problems. Figure 3 continues the results in the left panel out to $n_k = 4097$. We can see that the speed gains from grid generation continue as the size of the problem increases. It is worth noting the size of the linear programs that we are solving. With a capital grid of $4,097$ points, we solve for $8,194$ variables subject to $18,507,872$ constraints. Grid generation solves this large linear program in a little over an hour and a quarter.

We also experimented with a grid generation algorithm for standard value–function iteration. We began by solving on an initial grid of 16 points using the value–iteration algorithm described above. Given the solution to this problem, we add points on the capital grid halfway between each of the current points, doubling the size of the grid. We then take as the starting value for the next

Figure 4: Sensitivity to Discounting



round of value iteration, the average of the values at the two neighboring points (given by the solution to the $n_k = 16$ problem). This process is continued until the full problem has been solved. Although this grid generation improves the performance of the value–iteration algorithm, the gains are typically on the order of 30% (with a maximum of 90% for the $n_k = 513$ problem), it is not enough to make value iteration competitive with either constraint generation or grid generation.

One of the known drawbacks of value iteration is its sensitivity to the degree of persistence and the degree of discounting in the problem being solved. Our base case already has a high degree of persistence in the technology shock and has no depreciation in the capital stock. To examine the relative performance of our algorithms we solve the base–case model with $n_k = 1025$ for a grid of values for the discount factor: $\beta \in \{0.75, 0.8, 0.9, 0.95, 0.98, 0.99, 0.995, 0.999\}$. Figure 4 plots the computation time for grid generation and for value–function iteration against these values of the discount factor. In fact, standard value iteration takes a prohibitively long time to converge for large values of β. We, therefore, exploit a very specific feature of the problem at hand to speed up the algorithm. This goes against our objective of providing results that are likely to be true beyond this simple model, but it does make the comparisons we have in mind feasible. Specifically, when searching for the optimal action for each point in the state space, we begin at the current action and search by increasing the value of the action until the maximand decreases. This allows us to terminate the search before conducting a full enumeration of the action space. The monotonicity that this procedure exploits is a property that can be shown to hold at the optimum. It typically also holds at earlier iterations provided the initial conditions are increasing in the capital stock. We increase the speed of this algorithm further by exploiting the grid–generation method of obtaining accurate starting values, as described above. With this problem-specific speed up, value iteration can be faster than grid generation for small values for β. The important point to note, however, is that computational speed for grid generating is almost unaffected by increasing the values of β. In contrast, note the extremely rapid increase in computational time for value iteration ($2,205$ seconds for value iteration compared to 88.69 seconds for grid generation at $\beta = 0.999$). Constraint generation, though slower than grid generation, is also insensitive to the value of β.

4 Dimension Reduction through Spline Approximations

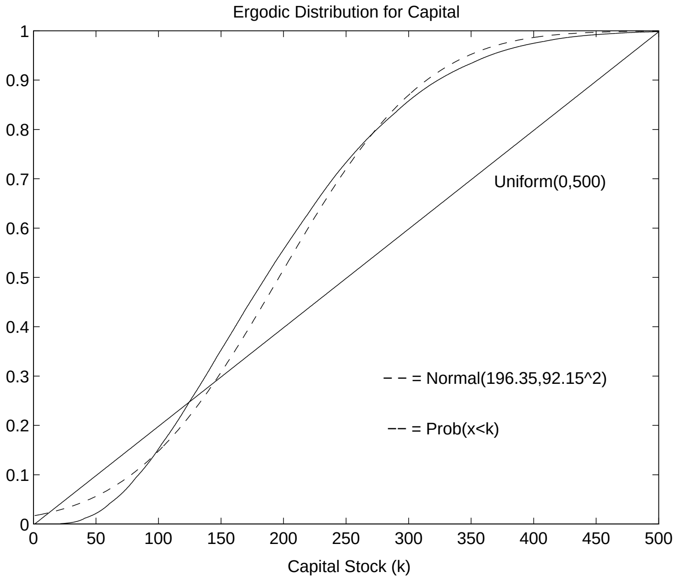
In this section we explore the possibility of reducing the size of the linear program we solve through the use of cubic spline approximations. Throughout this section we retain the same parameterizations as above with two exceptions. We increase the curvature of the utility function by setting the parameter ρ equal to -5.0. Since this greatly increases the precautionary savings motive on the part of the representative consumer, we expand the state space $\mathcal{K}$ to include the interval $[5, 800]$.

Note that these numerical values imply strong persistence in the dynamics of the capital stock since the exogenous shock is very persistent, the capital stock doesn't depreciate, and the agent has a very low rate of time preference (*i.e.*, the agent is very patient). Moreover, the large value of the risk aversion parameter will impute a high degree of curvature to the value function. Combining these features makes this model a challenge to solve.

Figure 5 plots the ergodic distribution function for the capital stock for a solution on a fairly fine grid, $n_k = 4097$. That is, the economy described above is solved (exactly) on this grid. Ergodic probabilities are computed by solving the equation

$$\Pi p = p,$$

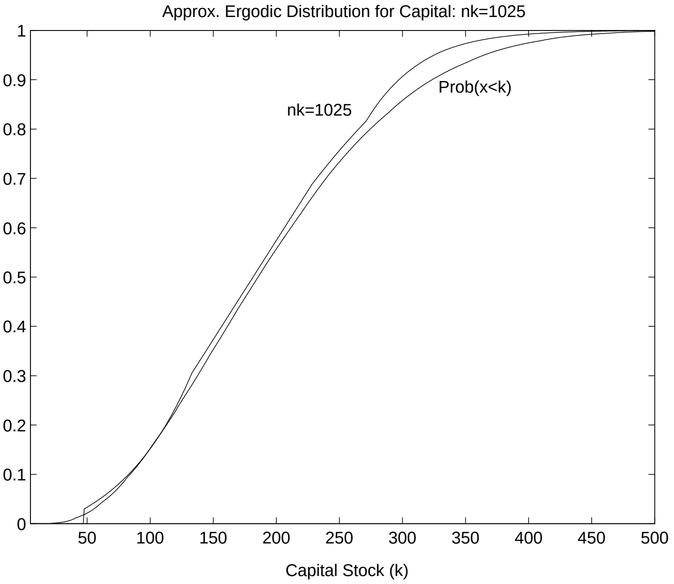
Figure 5:



where Π is the 8194 by 8194 matrix of transition probabilities for the optimal capital actions, and p is the 8194 by 1 vector of ergodic probabilities. For comparison sake a uniform distribution function on $(0, 500)$ and a normal distribution with mean 196.35 and standard deviation 92.15 are plotted on the same graph. Note that the ergodic distribution for the capital stock is more like a normal distribution than the uniform, though it is not symmetric (capital is bounded below by zero). We will use ergodic probabilities to compare features of various solutions.

Figure 6 demonstrates how a solution deteriorates when a a coarser grid, $n_k = 1025$ (roughly one quarter the size of that in Figure 5), is used as an approximation. Moments of this approximation are given in Table 1. Constraint generation has proven useful for solving dynamic programs when $n_k n_z$ is large. However, should these dimension get "too large" then the number of variables, *i.e.*, value function ordinates, in the linear program may make the solution algorithm infeasible. We now show how approximating the value function with a flexible functional form can greatly reduce the number of variables and still permit solution by linear programming.

Figure 6:



The cubic spline approximate value function, $\tilde{v}(k^{(i)}, z^{(j)})$, is defined by

$$\sum_{l=1}^{n_p} \mathbf{1}(k^{(i)} \in \mathcal{K}_l) \left[\gamma_0^{(l,j)} + \gamma_1^{(l,j)} k^{(i)} + \gamma_2^{(l,j)} (k^{(i)})^2 + \gamma_3^{(l,j)} (k^{(i)})^3\right]$$

where $\{\mathcal{K}_l\}$, $l = 1, 2, \ldots, n_p$, is a partition of $\mathcal{K}$, $\mathbf{1}(k^{(i)} \in \mathcal{K}_l) = 1$ if $k^{(i)} \in \mathcal{K}_l$ and 0 otherwise, and $\gamma = \{\gamma_0^{(l,j)}, \gamma_1^{(l,j)}, \gamma_2^{(l,j)}, \gamma_3^{(l,j)};\ j = 1, 2, \ldots, n_z,\ l = 1, 2, \ldots, n_p\}$, are the constant parameters of the spline approximation.

Continuity of this function in k requires the restrictions

$$\begin{aligned} \gamma_0^{(l,j)} &+ \gamma_1^{(l,j)} \tilde{k}(l, l+1) + \gamma_2^{(l,j)} \tilde{k}(l, l+1)^2 + \gamma_3^{(l,j)} \tilde{k}(l, l+1)^3 \qquad (5) \\ &= \gamma_0^{(l+1,j)} + \gamma_1^{(l+1,j)} \tilde{k}(l, l+1) + \gamma_2^{(l+1,j)} \tilde{k}(l, l+1)^2 + \gamma_3^{(l+1,j)} \tilde{k}(l, l+1)^3 \end{aligned}$$

for all j and $l = 1, 2, \ldots, n_p - 1$, where $\tilde{k}(l, l+1)$ is the point that "joins" $\mathcal{K}_l$ and $\mathcal{K}_{l+1}$.

Continuity of the first derivative of the function in k requires the restrictions

$$\begin{aligned}\gamma_1^{(l,j)} &+ 2\gamma_2^{(l,j)}\tilde{k}(l,l+1) + 3\gamma_3^{(l,j)}\tilde{k}(l,l+1)^2 \\ &= \gamma_1^{(l+1,j)} + 2\gamma_2^{(l+1,j)}\tilde{k}(l,l+1) + 3\gamma_3^{(l+1,j)}\tilde{k}(l,l+1)^2 \; .\end{aligned} \tag{6}$$

Similarly for the second derivative

$$\gamma_2^{(l,j)} + 3\gamma_3^{(l,j)}\tilde{k}(l,l+1) = \gamma_2^{(l+1,j)} + 3\gamma_3^{(l+1,j)}\tilde{k}(l,l+1) \; , \tag{7}$$

and the third derivative

$$\gamma_3^{(l,j)} = \gamma_3^{(l+1,j)} \; . \tag{8}$$

The approximate value function is nonlinear in the state variable k. Note, however, that it is still linear in γ, *i.e.*, the cubic spline parameters. Therefore, the solution to this approximate problem is the solution to the linear program

$$\min_{\gamma} \sum_{i,j} \sum_{l=1}^{n_p} \mathbf{1}(k^{(i)} \in \mathcal{K}_l) F_3^{(l,j)}(k^{(i)}) \tag{9}$$

subject to the restrictions

$$\sum_{l=1}^{n_p} \mathbf{1}(k^{(i)} \in \mathcal{K}_l) F_3^{(l,j)}(k^{(i)}) \geq u_{ija} + \beta \sum_{m=1}^{n_z} \pi_{jm} \sum_{l=1}^{n_p} \mathbf{1}(k^{(a)} \in \mathcal{K}_l) F_3^{(l,m)}(k^{(a)}), \tag{10}$$

where the cubic polynomial is given by

$$F_3^{(l,j)}(x) = \gamma_0^{(l,j)} + \gamma_1^{(l,j)}x + \gamma_2^{(l,j)}x^2 + \gamma_3^{(l,j)}x^3 \; .$$

Continuity of the function and its derivatives requires the additional restrictions given in (5)–(8).

Note the reduction in the number of variables in (9) compared to (4). For example, when $n_p = 20$, $n_k = 4097$, $n_z = 2$, solving for the value function ordinates involves 8194 "variables" in the LP, whereas solving for the cubic spline approximation involves only 160 "variables."

Figure 7 plots the ergodic distribution for an approximate solution that uses a cubic spline over $\mathcal{K}$ with 20 equal sized partitions. Continuity in k is imposed on the level and the first derivative of the value function. This distribution lies to the left of the exact distribution (*i.e.*, the distribution of the exact solution for $n_k = 4097$), which is further reflected in the moments in Table 1.

Figure 7:

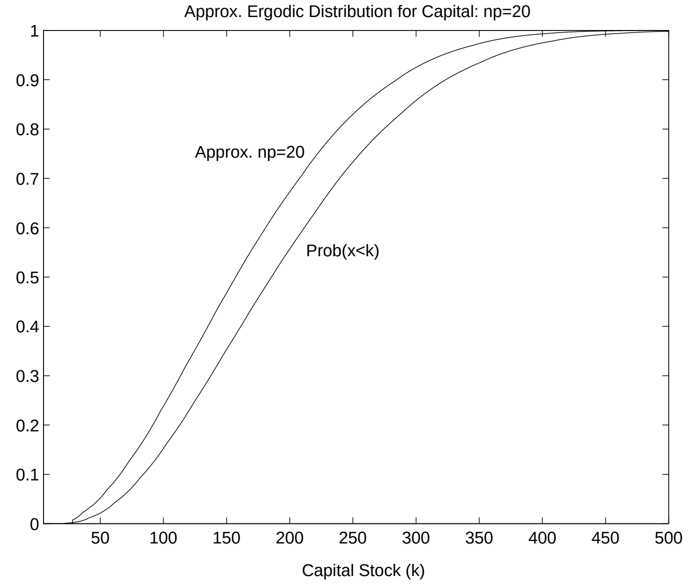
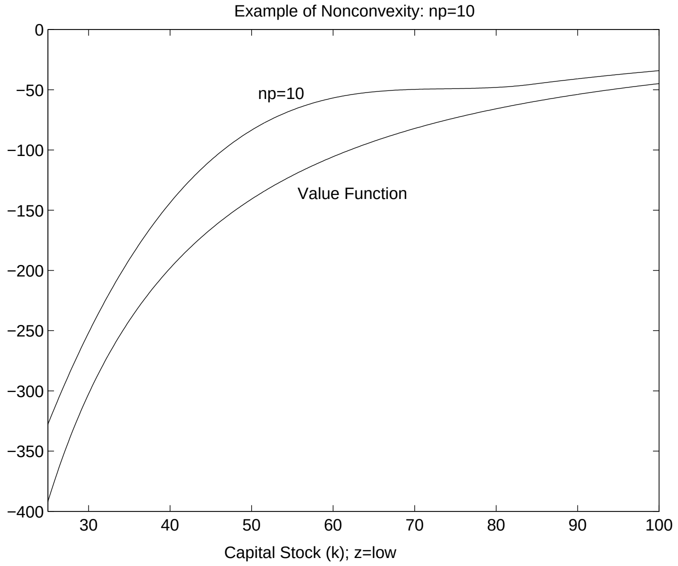


Figure 8 highlights a hazard inherent in the use of spline approximations. It is not difficult to generate convex approximate value functions in a situation where the true value function is globally concave. In this case, the spline approximation with a partition of 10 admits a "hump" just to the left of the first "join point". The consequence of this is evident in Table 1: the ergodic distribution is truncated at a very low value of the capital stock. Judd and Solnick (1994) provide methods for "shape preservation" that enable one to impose monotonicity and concavity on spline approximations. They show that incorporating these constraints can significantly improve the accuracy of these approximations without increasing the complexity of the spline (*e.g.*, increasing the number of partitions over the statespace). We would anticipate a comparable increase in accuracy from incorporating Judd-Solnick-type restrictions in our algorithms, however, since we found that the nonconcavities in our problem do not arise for splines with 20 or more partitions, we do not impose their restrictions on the solutions we report below.

Finally, we compute a solution with 40 partitions (320 variables). This solution is too close to the exact solution to demonstrate graphically. The moments are given in Table 1 and are extremely close to those of the exact solution.

Figure 8:



Example of Nonconvexity: np=10
0
−50
−100
−150
−200
−250
−300
−350
−400
np=10
Value Function
30
40
50
60
70
80
90
100
Capital Stock (k); z=low

Table 1: Moment Comparisons

Algorithm	$\mu = E(k)$	$[E(k-\mu)^2]^{1/2}$	$[E(k-\mu)^3]^{1/3}$	$[E(k-\mu)^4]^{1/4}$
$n_k = 4097$	196.3489	92.1525	76.3402	120.8191
Spline $n_p = 40$	195.6602	92.1934	76.1663	120.8491
Spline $n_p = 20$	167.6257	83.5304	69.4390	108.9115
Spline $n_p = 10$	5.5678	5.5736	12.0165	17.7886
$n_k = 2049$	178.9011	82.0057	66.5746	107.3697
$n_k = 1025$	187.8145	82.4057	58.4772	104.7454
Adaptive Spline I	159.4942	78.2545	68.3564	103.6734
Adaptive Spline II	174.2334	83.4206	68.9132	108.6981

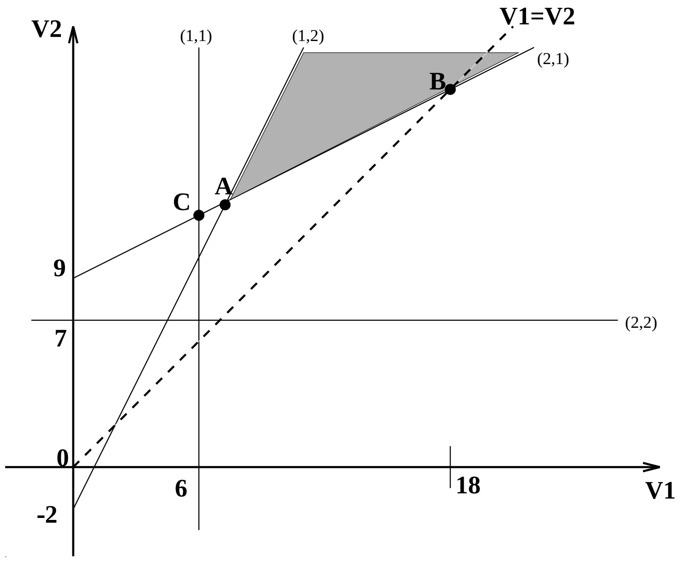
5 Value Function Bounds

The use of linear programming as a method for solving for the cubic-spline approximation to the value function has an additional benefit. The approximation must always lie above the exact discrete value function. This property is easily seen through a simple two-dimensional example.

Assume that the growth model described above is deterministic ($n_z = 1$). Further assume that the capital stock can take on only two values ($n_k = 2$). This implies that there are two variables and (at most) 4 restrictions in the linear program in (4) and (3). Assign the following numerical values to the instantaneous utilities:

$$u_{11} = 3.0 \qquad u_{12} = 1.0$$

Figure 9: Two-Dimensional Example



$$u_{21} = 9.0 \qquad\qquad u_{22} = 3.5$$

and set $\beta = 0.5$. The resulting restriction on the value function are

$$\begin{array}{lcll}
v_1 \geq u_{11} + \beta v_1 & \Rightarrow & v_1 \geq 3 + .5v_1 & (1,1) \\
v_1 \geq u_{12} + \beta v_2 & \Rightarrow & v_1 \geq 1 + .5v_2 & (1,2) \\
v_2 \geq u_{21} + \beta v_1 & \Rightarrow & v_2 \geq 9 + .5v_1 & (2,1) \\
v_2 \geq u_{22} + \beta v_2 & \Rightarrow & v_2 \geq 3.5 + .5v_2 & (2,2)
\end{array}$$

These restrictions are depicted in Figure 9.

The solution to this dynamic program is depicted by the point A in Figure 9. The optimal policy is to take action 2 in state 1 and action 1 in state 2. The optimal value in state 1 is 7.33 and the optimal value in state 2 is 12.67. If we were to reduce the dimensionality of this problem by imposing, say, the linear restriction $v_1 = v_2$, the linear program with this restriction would have a solution at point B in the figure. The value function ordinates are equal to each other at a level of 18. This implies action 1 be taken in state 1 and action 1 in state 2. If we calculate the unrestricted value of these actions the result is point C with the value in the first state equal to 6 and the value in the second state equal to 12.

This simple example demonstrates a general property of a cubic-spline approximate value function solved with linear programming. The approximation must always lie above the exact discrete value function. Moreover, the actions implied by this approximation generate unrestricted values that must lie below the exact discrete value function (since these actions are feasible but suboptimal). Through these simple upper and lower bounds on the exact discrete value function, we get valuable information about the accuracy of the approximation. That is, if the bounds are close to each other, the approximate solution is accurate.

The proof that the cubic-spline approximate value function is an upper bound on the exact discrete value follows directly from the following Lemma.

Lemma: For any x and y in $\mathbf{R^n}$ define the vector $z \in \mathbf{R^n}$ as $z(x, y) = \min(x, y)$ where the min is taken componentwise. Let P be the polyhedron defined by (3). If $x \in P$ and $y \in P$ then $z(x, y) \in P$.

Proof: Consider two feasible solutions to (4), v^1 and v^2 with typical components v^1_{ij} and v^2_{ij}. Let z_{ij} be a typical component of $z(v^1, v^2)$. We need to show that:

$$z_{ij} \geq u_{ija} + \beta \sum_{l=1}^{n_z} \pi_{jl} z_{al} \ ,$$

for all (i, j).

Consider an instance where $z_{ij} = v^1_{ij}$. Then we know that

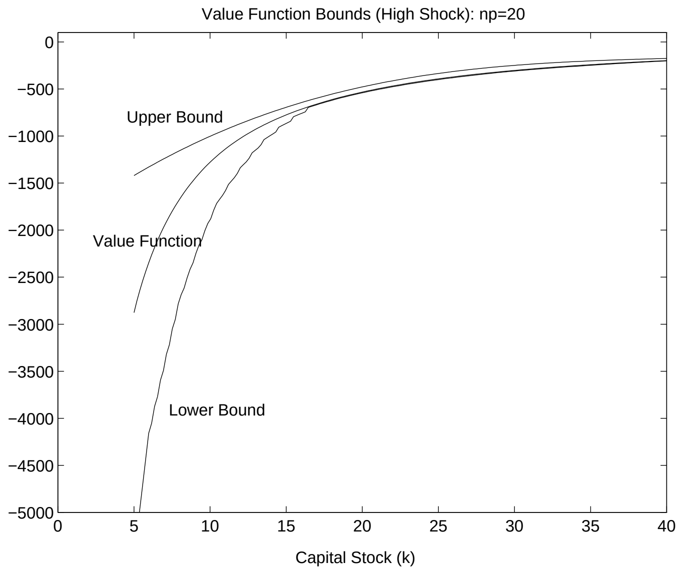
$$z_{ij} \geq u_{ija} + \beta \sum_{l=1}^{n_z} \pi_{jl} v^1_{al} \ .$$

But since $z_{al} \leq v^1_{al}$ by definition, it follows that

$$z_{ij} \geq u_{ija} + \beta \sum_{l=1}^{n_z} \pi_{jl} z_{al} \ .$$

The same would be true if we began with an instance where $z_{ij} = v^2_{ij}$. Therefore $z(v^1, v^2)$ is a feasible solution to (4).

Figure 10:



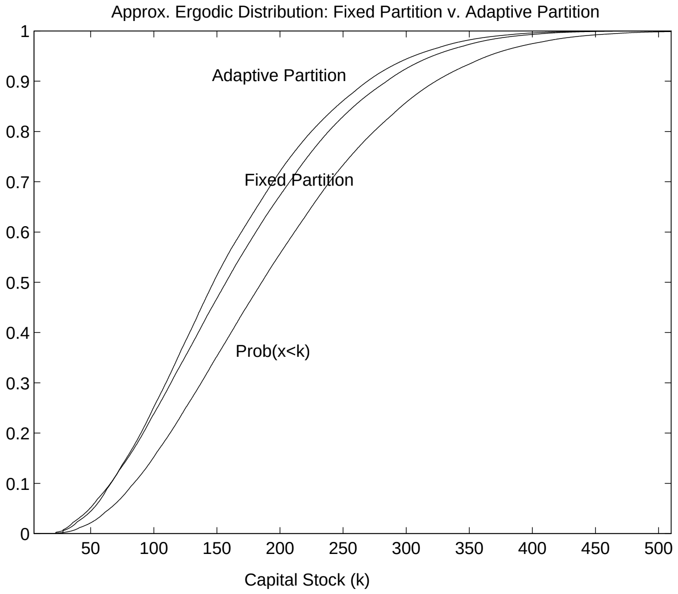
Theorem: The cubic-spline approximate value function is an upper bound on the exact discrete value function.

Proof: Let v^* be the solution to (4) and let v^S be the cubic-spline approximate solution to (4). For v^S to be an upper bound on v^* it must be that $z(v^*, v^S) = v^*$. If this is not the case, then $z(v^*, v^S)$ is a feasible solution to (4) by the Lemma and it has a smaller value for the objective function than v^*, which contradicts the definition of v^*.

As discussed above, the lower bound to the exact discrete value function is found by finding the unrestricted values implied by the spline-approximate actions. Note that the theorem could be stated in more general terms since nothing specific about the structure of splines *per se* was used in the proof. That is, any approximation method that is amenable to fitting with linear programming (specifically, those that are linear in unknown parameters) will bound the exact discrete value function.

Figure 10 demonstrates these bounds (with partition size of 20) for a particularly difficult-to-approximate region of the parameter space.

Figure 11:

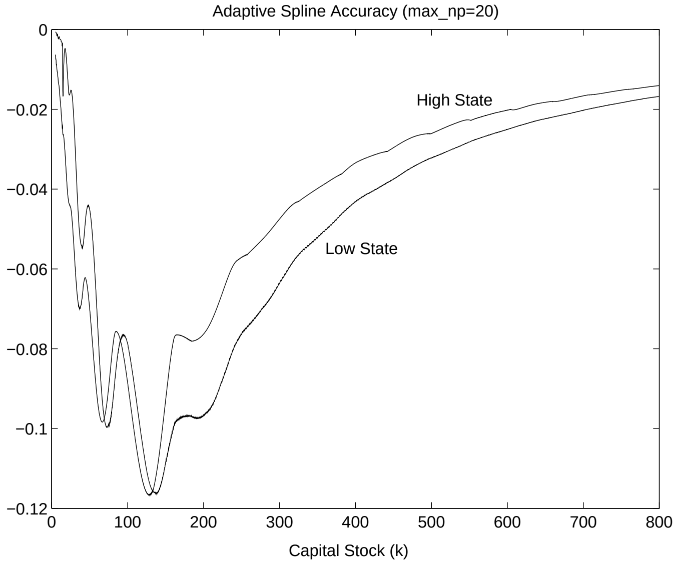


6 Adaptive Spline Generation

The bounds discussed above can be used to refine the accuracy of a given solution. When the bounds are far apart for a particular region of the state space, the spline approximation can be adapted accordingly. That is, either a finer partition or a richer polynomial can be given to that region of the state space. We opt for the former of these two options and maintain a cubic spline throughout. The model can be resolved with the new parameterization, the new accuracy can be checked, and so on. The process stops when a desired level of accuracy is achieved.

Figure 11 plots the ergodic distribution of an approximation that follows the following adaptation rule. Start with 10 equal partitions, solve the cubic spline approximation using constraint generation, and locate the partition that has the greatest distance between upper and lower bounds on the value function. Split this partition in half and iterate until the approximation has 20 partitions. For comparison, the distributions of the "exact" solution and the fixed partition of size 20 are also plotted in Figure 11. Note that this particular form of adaptation does not generate much additional accuracy on average. The adaptation gets closer to the exact solution at very small values of capital but is

Figure 12:



further away for high values. The reason for this can be seen in Figure 12 where *percentage* deviations of the bounds are plotted for the final adapted solution. This particular rule overemphasizes the dramatic absolute distance at the low end of the state space. There is little accuracy to be gained over this region, therefore, this adaptation sacrifices accuracy over the rest of the distribution where the relative distance between the bounds is actually much greater.

To address this issue, we adopt an alternative form of adaptation. Start with 10 equal partitions, solve the cubic spline approximation using constraint generation, and locate the partition that has the greatest *relative* distance between upper and lower bounds on the value function. Split this partition in half and iterate until the approximation has 20 partitions. The results for this rule are given in Figure 13. From this figure it is clear that this is a much more reasonable adaptation rule which can also be seen from the moments in Table 1. Figure 14 reveals that relative accuracy in this example is within one percent.

Figure 13:

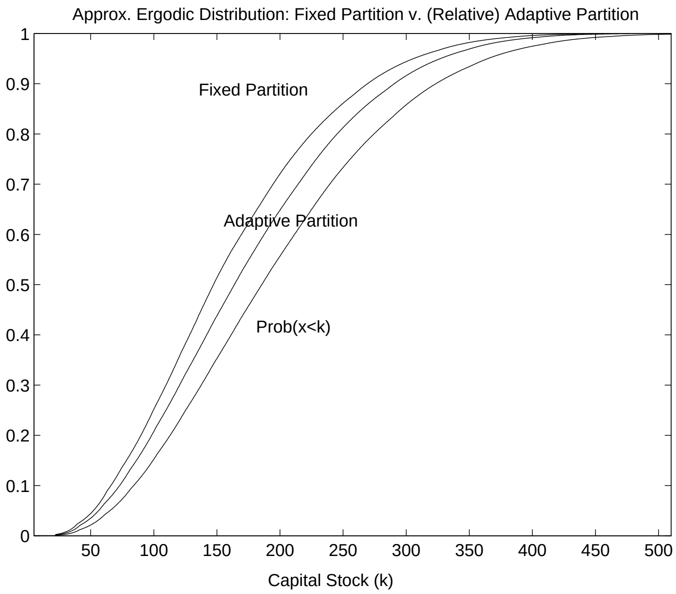
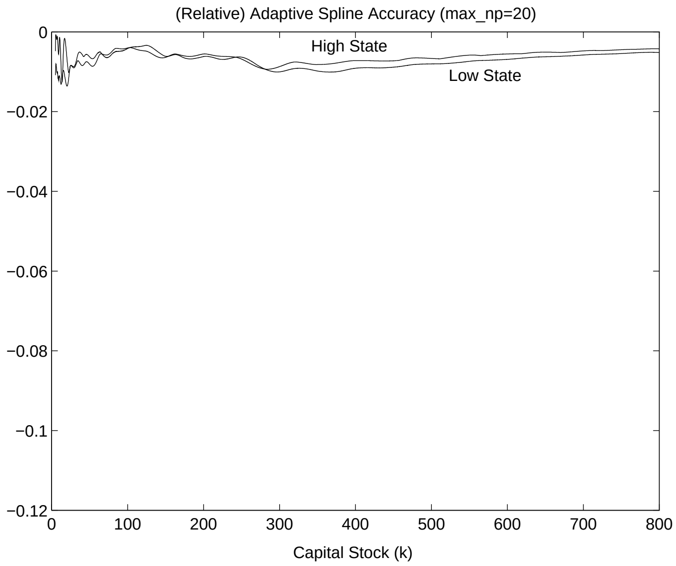


Figure 14:



7 Final Remarks

The preliminary performance of our linear-programming-based adaptive spline generation demonstrates that this method has a number of advantages over competing approaches. Future work will focus on developing this algorithm for other economic models, in particular, those with higher-dimensional state spaces. For example, Johnson et al. (1993) used conventional spline approximations for a series of stochastic water supply reservoir problems with multidimensional state spaces. They found that higher dimensional splines offered significant improvements over standard discretization methods (although for dimensions higher than three, their algorithms are still very time consuming). In addition, exploring alternative functional approximations other than splines, *e.g.* neural networks as in Mrkaic, Trick, and Zin (1997), also holds promise for expanding the scope of linear programming algorithms and the accuracy adaptations they provide, to a broader class of economic models.

References

Bertsekas, D. (1976), *Dynamic Programming and Stochastic Control.* New York: Academic Press.

Christiano, Lawrence J. (1990), "Linear-Quadratic Approximation and Value-Function Iteration: A Comparison," *Journal of Business and Economic Statistics* 8, 99-114.

Johnson, Sharon A., Jery R. Stedinger, Christine A. Shoemaker, Ying Li, Jos'e Alberto Tejada-Guibert (1993), "Numerical Solution of Continuous–State Dynamic Programs Using Linear and Spline Interpolation," *Operations Research* 41, 484-500.

Judd, K. and A. Solnick (1994), "Numerical Dynamic Programming with Shape-Preserving Splines," Manuscript, Hoover Institution.

Koehler, G.J. (1976), "A Case for Relaxation Methods in Large Scale Linear Programming," in G. Guardabassi and A. Locatelli, (eds.), *Large Scale Systems Theory and Applications.* Pittsburgh: IFAC, 293-302.

Mrkaic, Mico, Michael Trick, and Stanley E. Zin (1997), "Neural Networks as Polyhedral Approximations to Markov Decision Problems," Working Paper, Carnegie Mellon University.

Puterman, Martin L. (1994), *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* New York: John Wiley & Sons, Inc..

Ross, Sheldon M. (1983), *Introduction to Stochastic Dynamic Programming.* Orlando: Academic Press.

Schweitzer, Paul J., and Abraham Seidmann (1985), "Generalized Polynomial Approximations in Markovian Decision Processes," *Journal of Mathematical Analysis and Applications* 110, 568-582.

Trick, M.A. and Stanley E. Zin (1993), "A Linear Programming Approach to Solving Stochastic Dynamic Programs," Working Paper, Carnegie Mellon University.

Whitt, Ward (1978), "Approximations of Dynamic Programs, I," *Mathematics of Operations Research* 3, 231-243.

Whitt, Ward (1979), "Approximations of Dynamic Programs, II," *Mathematics of Operations Research* 4, 179-185.