A Linear Progamming Approach to Solving Stochastic Dynamic Programs*

Michael A. Trick† Stanley E. Zin‡

August 1993

Abstract

Recent advances in algorithms for solving *large* linear programs, specifically constraint generation, motivate new algorithms for solving discrete stochastic dynamic programs. We use a standard optimal growth problem to demonstrate the performance benefits of these new algorithms for solving discrete problems and for accurately approximating solutions to continuous problems through discretization. Computational speed over value iteration is substantial. Furthermore, computational speed does not depend on the parameter settings (in particular the degree of discounting). An added benefit of a linear programming solution is the byproduct of *shadow prices* which we use to generate a discrete grid *adaptively*. That is, for a fixed number of grid points, our algorithm determines how they should be distributed over the state space to obtain greater accuracy without increasing dimensionality.

*This work was inspired by conversations with Victor Rios. We thank Chris Telmer and Tony Smith for helpful comments.

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

‡GSIA, Carnegie Mellon University and NBER.

1 Introduction

Computational economics has enabled researchers to push out the frontiers of the discipline far beyond what purely analytical methods will allow. No area of economics seems to be untouched by this computational revolution. It has had profound effects on applied microeconomics, labor economics, business cycles, finance, game theory, social choice, and econometrics. Unfortunately, researchers adopting the computational approach have quite naturally reached another barrier: existing numerical algorithms cannot always deal with the increasing complexities of economic models. Economic problems that lack sufficient smoothness (such as models with fixed costs or borrowing constraints) do not easily fit into smooth approximation methods (such as linear-quadratic approximations). Dynamic problems with high dimensional state spaces rule out most algorithms by virtue of simple memory requirements. Econometric estimators based on repeated simulations of solutions to dynamic economies are typically limited by the slowness of most computational algorithms. For these and countless other reasons the benefits from improving our ability to solve interesting/difficult economic models are immeasurable.

This paper bring recent advances in Operations Research to bear on some of these computational issues in economics. Recent advances in algorithms for solving *very large* linear programs, specifically constraint generation, motivate new algorithms for solving discrete stochastic dynamic programs. We use a standard optimal growth problem to demonstrate the performance benefits of these new algorithms for solving discrete problems and for accurately approximating solutions to continuous problems through discretization. Our results, reported below, suggest that computational speed over value iteration is substantial. Furthermore, computational speed does not depend on the parameter settings (in particular the degree of discounting).

Perhaps most important, though, is our use of shadow prices to automatically generate a discrete grid. A major difficulty in discretization is the choice of the grid. Too coarse a grid may lead to inaccurate solutions while too fine a grid may be computationally intractable. Our approach can automatically generate a suitable grid without prior knowledge of properties of the solution. Moreover, this is accomplished at no extra computational cost. Our algorithms, therefore, obtain greater accuracy without increasing dimensionality or computational cost.

The simplest algorithm for solving finite discrete numerical dynamic programs and for approximating continuous problems through discretization is value–function iteration. It requires no specialized computer software and is based on the same contraction–mapping principle that is typically used to establish the existence of a solution. The problems associated with standard value–function iteration are well known and have often led researchers to abandon this algorithm in favor of other methods (see Judd (1991), for a thorough discussion of numerical dynamic programming and solution methods). Among these problems are the rapid increase in the size of the problem as the state space expands, the sensitivity of the algorithm to properties of the problem (in particular the degree of discounting), and when approximating continuous problems, knowing how to discretize the state space and when a particular discretization has provided a sufficiently accurate approximation. However, there are also benefits from solving directly for the value function. For example, the value function is defined for all problems so it can be obtained in problems where corner solutions or non-differentiabilities make solving for optimal policies using alternative methods inapplicable. Moreover, the ability to obtain arbitrarily accurate discrete approximations to continuous problems by using arbitrarily fine discretizations has lead to the use of value-function-based solutions as a benchmark for accuracy checks (see, for example, Judd (1991), chapter 13, and Christiano (1990)).

In this paper we adopt a linear programming approach for solving directly for value functions in stochastic dynamic programming problems. The use of linear programming, *per se*, is neither new nor, as we show below, is it necessarily *better* than value iteration. What is new in this context is the use of constraint–generation algorithms for solving these linear programs, which can provide orders of magnitude computational savings, and the use of dual values (*i.e.*, shadow prices) for determining an efficient location of points on the discretization of the state space. As we shall see, the combination of constraint generation and *adaptive* grid generation provides an extremely attractive algorithm for solving discretized stochastic dynamic programs.

The class of dynamic programming problems that economists would like to solve numerically is extremely large. Rather than present the most general case, we develop our computational methods in terms of the standard optimizing growth model. Since this problem is the starting point for most dynamic economic theories, other problems inherit much of the structure of this problem and generalizations are fairly obvious. Moreover, this model has become an informal benchmark for comparisons of competing algorithms (*e.g.*, Taylor and Uhlig (1990)). We begin by laying out the structure of a standard optimal growth model (*e.g.*, Stokey and Lucas (1989), chapter 2) in Section 2. This model serves both as a benchmark for comparisons with other solution methods and as a *canonical* stochastic dynamic program. In Section 3 we solve a variety of numerical examples of this growth model using three different algorithms. We compare speed and robustness for value function iteration, straight linear programming and constraint generation and we discuss grid generation as a way of speeding up both value iteration and linear programming. Finally in Section 4 we present the adaptive grid generation algorithm and discuss issues of accuracy. Section 5 present extensions and other applications of this approach that we will pursue in the future.

2 The Stochastic Growth Model

We now lay out the basic structure of the stochastic growth model. For time periods $t = 1, 2, \ldots$, the production technology is given by

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

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

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

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

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

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

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

and

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

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

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

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

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

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

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

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:

$$\text{s.t.}\ \ v_{ij} \ge u_{ija} + \beta \sum_{l=1}^{n_z} \pi_{jl} v_{al} \quad , \tag{3}$$

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

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

subject to (3). We can put this problem into more standard linear programming notation. Define

$$\begin{aligned} x \ &= \ [v_{11}, v_{12}, \ldots, v_{1n_z}, v_{21}, v_{22}, \ldots, v_{2n_z}, \\ &\qquad \ldots, v_{n_k 1}, v_{n_k 2}, \ldots, v_{n_k n_z}]' \quad , \end{aligned}$$

$$\begin{aligned} b \ &= \ [u_{111}, u_{111}, \ldots, u_{11n_{11}}, u_{121}, u_{122}, \ldots, u_{12n_{12}}, \\ &\qquad \ldots, u_{n_k n_z 1}, u_{n_k n_z 2}, \ldots, u_{n_k n_z n_{n_k n_z}}]' \quad , \end{aligned}$$

and $\mathbf{1}_n = [1, 1, \ldots, 1]'$ is an $n = n_k n_z$ dimensional column vector of ones. The value function ordinates are given by the solution to

$$\begin{aligned} &\min_x \mathbf{1}_n' x \\ &\text{s.t.}\ \ Ax \ge b \quad , \end{aligned} \tag{4}$$

where A is an $(\sum_{ij} n_{ij}) \times n$ matrix given by the constraints in (3). This matrix has a great deal of structure which can be seen from a simple example. Consider the case where $n_k = 3$, $n_z = 2$, $n_{ij} = n_k = 3$ for all i and j, and the shocks are independent, $\pi_{ij} = \pi_i$. The matrix A in this case has the form:

$$A = \begin{bmatrix}
1-\beta\pi_1 & -\beta\pi_2 & 0 & 0 & 0 & 0 \\
1 & 0 & -\beta\pi_1 & -\beta\pi_2 & 0 & 0 \\
1 & 0 & 0 & 0 & -\beta\pi_1 & -\beta\pi_2 \\
-\beta\pi_1 & 1-\beta\pi_2 & 0 & 0 & 0 & 0 \\
0 & 1 & -\beta\pi_1 & -\beta\pi_2 & 0 & 0 \\
0 & 1 & 0 & 0 & -\beta\pi_1 & -\beta\pi_2 \\
-\beta\pi_1 & -\beta\pi_2 & 1 & 0 & 0 & 0 \\
0 & 0 & 1-\beta\pi_1 & -\beta\pi_2 & 0 & 0 \\
0 & 0 & 1 & 0 & -\beta\pi_1 & -\beta\pi_2 \\
-\beta\pi_1 & -\beta\pi_2 & 0 & 1 & 0 & 0 \\
0 & 0 & -\beta\pi_1 & 1-\beta\pi_2 & 0 & 0 \\
0 & 0 & 0 & 1 & -\beta\pi_1 & -\beta\pi_2 \\
-\beta\pi_1 & -\beta\pi_2 & 0 & 0 & 1 & 0 \\
0 & 0 & -\beta\pi_1 & -\beta\pi_2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1-\beta\pi_1 & -\beta\pi_2 \\
-\beta\pi_1 & -\beta\pi_2 & 0 & 0 & 0 & 1 \\
0 & 0 & -\beta\pi_1 & -\beta\pi_2 & 0 & 1 \\
0 & 0 & 0 & 0 & -\beta\pi_1 & 1-\beta\pi_2
\end{bmatrix}.$$

The pattern in this matrix is fairly obvious and is easily extended to the general case.

The size of this linear program will clearly present a problem. The value function at each point $(i,\ j)$ in the state space must satisfy n_{ij} restrictions. The number of constraints for the $n_k n_z$ variables could be as large as $n_k^2 n_z$. In particular, when this discrete problem is approximating a continuous one, accurate solutions would seem to require n_k to be quite large, hence making standard numerical methods impractical. For this reason, we explore constraint generation algorithms in this context.

Constraint generation is a technique for solving linear programs with a large number of constraints. Rather than have a computer code attempt to solve such 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.

Constraint generation has been spectacularly successful recently in solving tremendous linear programs. For instance, in the work of Grötschel and Holland (1991), a formulation for the traveling salesman problem that is estimated to have 2^{60} constraints is solved in a matter of hours on a workstation. Work such as this requires the ability to solve some optimization problem in order to identify violated constraints.

We adapt the constraint generation technique to solving the linear programming formulation for discrete stochastic dynamic programs. We will begin with a small number of constraints and add constraints only when the current solution violates them. Unlike the work listed above, we need to check each constraint in turn to see if it is violated. (In this paper we solve linear programs in more than $8,000$ variables subject to more than 18.3 million constraints. 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:

- 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.
- We can precalculate common terms used in multiple constraints.
- 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 Solving for Value Functions

We specialize the growth model further by specifying explicit functional forms for f and u, choosing numerical values for all of the models parameters, and choosing a discretization of the state space. For these numerical models we compare the performance of value iteration, linear programming and constraint generation algorithms.

3.1 Parameter settings

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. In the "base case" we set the discount factor, β, to 0.98 and the risk aversion parameter, ρ, to 0.5. For this base case, we evaluate the performance of each algorithm as the n_k increases which increases both the number of choice variables and the number of constraints in the linear program. To evaluate the robustness of these results we also conduct experiments where n_k is fixed and 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. 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.

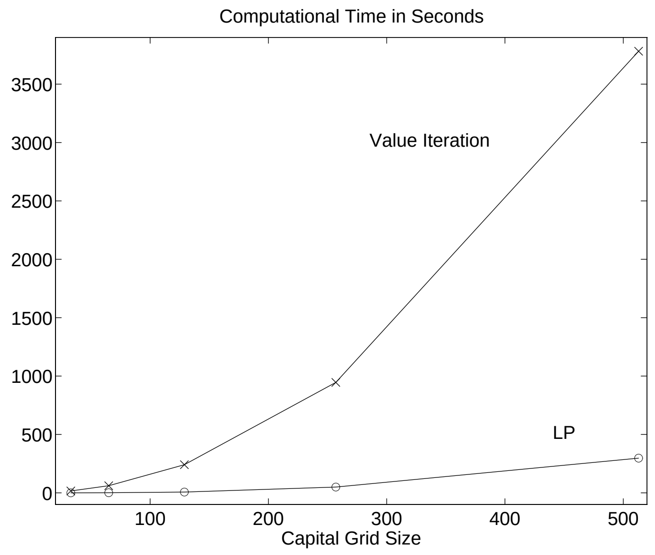
Starting values for value iteration are chosen as follows. For each point in the state space, we calculate the steady-state utility as if the smallest feasible capital stock was the deterministic steady state. This value, $u(k)/(1-\beta)$, forms the initial value from which we iterate until convergence. 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, we believe, as 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 $\max_{(i,j)} |v_{ij}^{m+1} - v_{ij}^{m}| < 0.000001$.

3.2 Solutions

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.

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

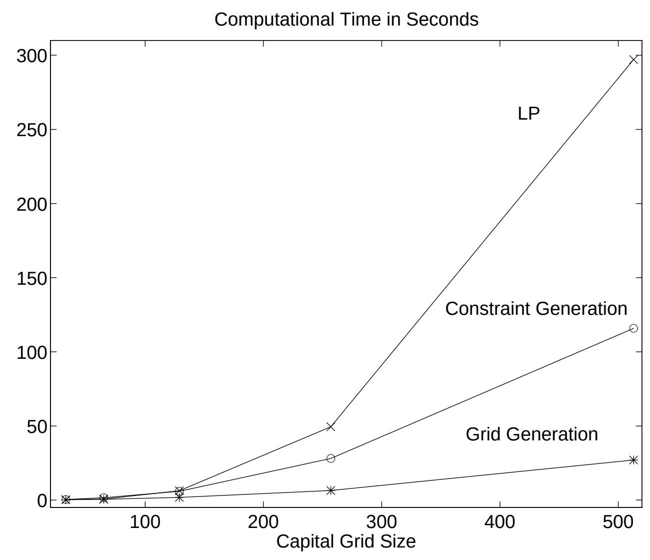
Figure 1: Value Iteration and Linear Programming Comparisons



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 ($(1-\delta)k_i + \gamma k_i^\alpha z_j$ 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.

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 com-

Figure 2: LP, Constraint and Grid Generation Comparisons

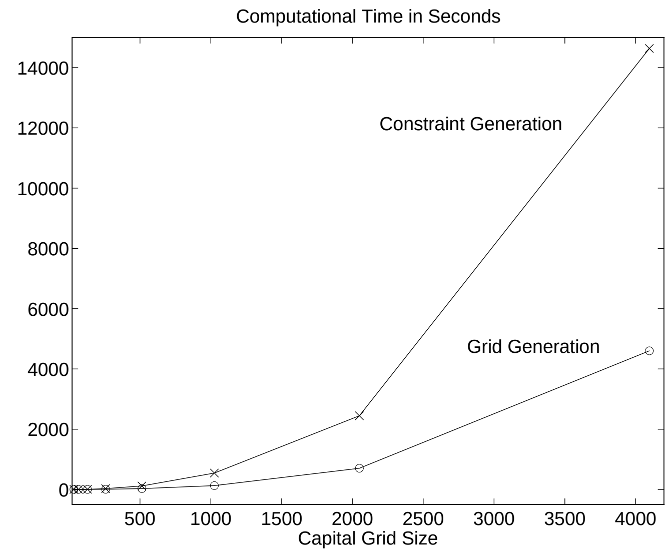


pared to 15.86 seconds). For the largest problem in this figure, $n_k = 513$, linear programming is approximately 13 times faster than value iteration (297.11 seconds compared to 3781.7 seconds). These results indicate that standard linear programming can provide at least an order-of-magnitude improvement over standard value–function iteration for problems of this size. The drawback of standard linear programming is the large amount of memory needed to solve large problems. However, as discussed in the introduction, recent algorithmic advances help 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 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

Figure 3: Larger Problem



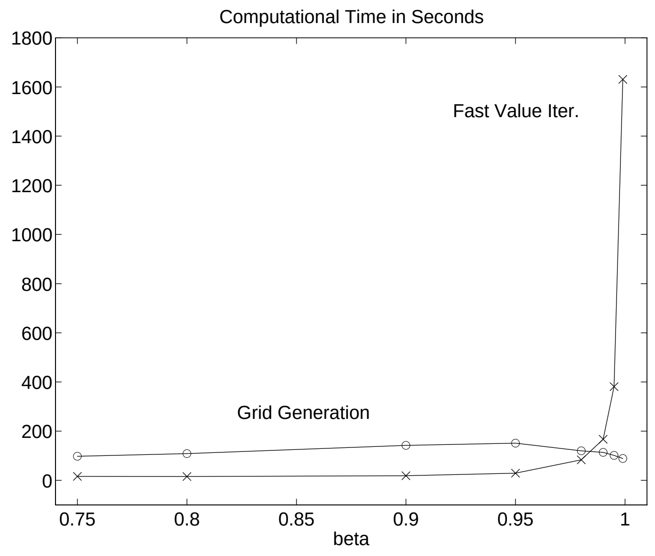
extension.

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 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 column 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

Figure 4: Sensitivity to Discounting

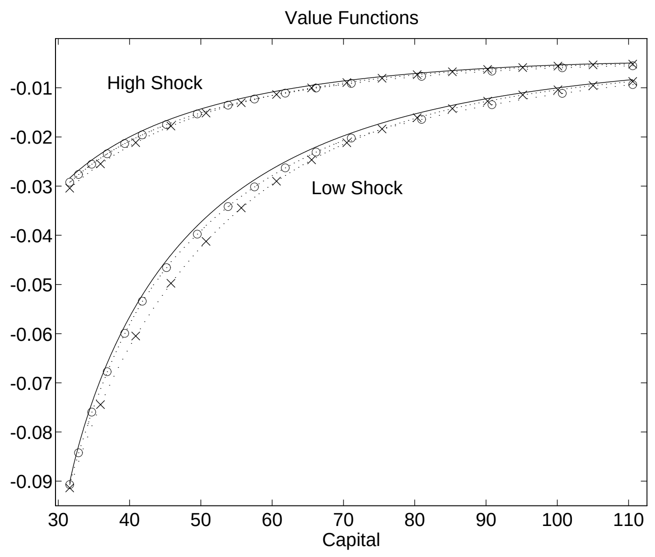


(2,205 seconds for value iteration compared to 88.69 seconds for grid generation at $\beta = 0.999$). Straight column generation, though slower than grid generation, is also insensitive to the value of β. With the increasing popularity of simulation estimators in econometrics, the stability of an algorithm as one searches over a parameter space is extremely important. Our linear programming algorithms seem to satisfy this need.

4 Adaptive Grid Generation

The algorithms that we have discussed thus far all require the solution of larger problems to obtain greater accuracy. With the linear programming approach, however, the necessity of this size–accuracy tradeoff is less clear. Whenever we solve the problem for a given grid, the linear programming solution provides us with information about where to locate a new grid point to obtain the greatest impact on the accuracy of the ultimate solution. We for-

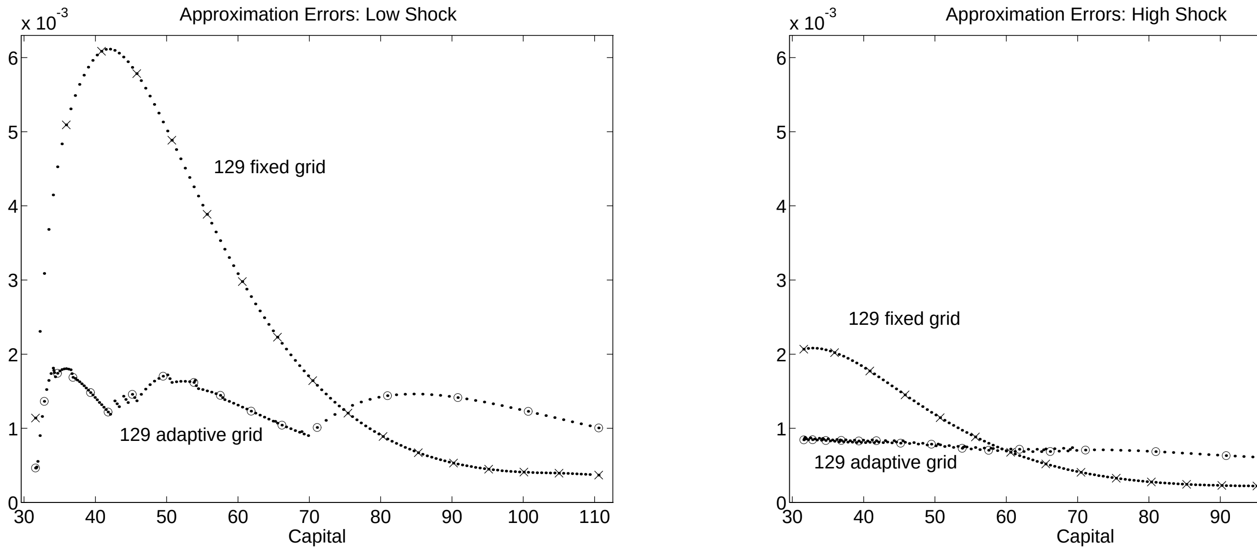
Figure 5: $\rho = -5$: (–) $n_k = 2049$; (x) $n_k = 129$ fixed; (o) $n_k = 129$ adaptive



malize this as follows. We begin by optimizing over a coarse evenly spaced grid using column generation. Each state (i, j) on this coarse grid has a optimal action $a(i, j)$ and a shadow price $w(i, j)$ generated by the linear programming solution. This shadow price measures the impact on the sum of the value function ordinates (the objective function in the linear program), of a small change in the constraint, specifically $u(i, j, a(i, j))$. We next calculate the *slack* of each constraint adjacent to the constraint for $a(i, j)$. This amounts to finding the change in u that would result in the constraint holding with equality. We then multiply these slacks by the shadow price to obtain a measure of the impact on the objective function of placing a new grid point adjacent to $a(i, j)$. We do this for all (i, j). We then add the actions and the states corresponding to the highest values of this product of shadow price and slack. The new points are always the midpoint between two existing points. We call this method adaptive grid generation.

Figure 5 plots the optimal value functions (with a solid line) for our basic model with $\rho = -5$. This solution is still approximate since it was obtained

Figure 6: $\rho = -5$: (x) $n_k = 129$ fixed; (o) $n_k = 129$ adaptive



with a grid size of $n_k = 2,049$. This grid is, however, sufficiently fine that we treat this as the *true* solution. Note that the amount of curvature in the value function is greater conditional on low value of the technology shock. We also plot the approximate solution for a fixed grid of 1/16th the size ($n_k = 129$), using an "x" to mark every eighth grid point. Note that this solution lies everywhere below the true solution and that the approximation error is greatest for small values of the capital stock when the technology shock is in the low state. This is also where the value function has its greatest curvature. Finally in Figure 5 we plot the solution using the adaptive grid generation method described above. This solution is plotted using an "o" to make every eighth grid point. Note that with the same number of grid points, the adaptive grid method generally does a better job approximating the true value function. This is especially true in the region where the value function has its greatest curvature.

Figure 6 plots the difference between the true solution and these approximations. A number of features of the adaptive grid generation method are apparent from these figures. Note that the adaptive method concentrates far more points in the region of the state space where the value function has its greatest curvature. As a result it places relatively few points in the region where the value function is more flat. This results in an approximation error that is roughly the same size for the entire state space. In contrast note the dramatic difference in the size of the approximation errors over the capital grid for the fixed grid approximation. Our experience is that with adaptive grid generation, adaptively increasing the grid size lowers the approximation errors for the entire range of capital. For the fixed grid, an extremely large grid must be solved to get the approximation error down for points where the value function has its greatest curvature. Much of the computational burden involved in this is, of course, unnecessary since for large values of the capital stock the error is already quite small. The results for adaptive grid generation are, therefore, extremely encouraging.

5 Extensions

There are many ways the algorithms that we propose can be improved and extended. For the type of application detailed above, the next step will be to investigate the role of the particular objective function we choose for the linear program. We currently minimize the sum across all points in the state space of the value function ordinates. This choice does not affect the solution for a fixed grid, however, it is very important for adaptive grid generation. What we propose is to calculate the ergodic distribution of the state space for each subproblem that we solve. These probabilities can then be used to weight the value–function ordinates before forming the objective function. With this weighting, adaptive grid generation will generate more points not just where they have the biggest effect on values, but rather where they will have the biggest effect on the most-likely-to-occur values. For problems in which the main focus is on properties of probability distributions of the solutions, we think that this will provide much more accurate results. The primary computational issue is whether we must solve for ergodic probabilities in addition to solving the linear program (and if so, what is the relative cost involved), or whether these probabilities can be determined from information that is already available from the linear programming algorithm.

More generally, we plan to explore the issue of accuracy for our algorithm as well as other algorithms. The adaptive grid generation methodology yields bounds on how much the value function ordinates can be affected by increasing the size of the problem. This provides us with a measure of accuracy. However, since the values themselves are rarely of fundamental interest, we plan to extend this concept of accuracy to other properties of the solution such as actions and ergodic distributions. If this proves successful, we envision using our algorithm to evaluate the accuracy of solutions provided by other algorithms as well.

Much of our future research in this area will involve developing computational methods related to the ones described above for other classes of economic models. The problems of immediate interest to us include:

- Solutions to other optimum problems such as the growth model with fixed costs. Fixed costs, in particular, confound many methods that rely on approximations by *smooth* functions, since optimal actions may be subject to large jumps. Value–function iteration is always available for such problems but, as described above, is not very useful in practice. Our linear programming methods inherit all of the good properties of value–function iteration for such problems but eliminate most of the deficiencies. In particular, adaptive grid generation should prove very effective for these types of problems since it has the ability to find the location of discontinuities quickly and accurately.

- Solutions to *non-optimum* equilibrium problems such as heterogeneous agent–incomplete markets models. This class of problems has attracted a lot of attention in both finance and macroeconomics. The problems that people have been able to address with confidence in accuracy, however, have been quite limited in terms of complexity. Our goal here is to be able to solve for equilibria recursively. If our dynamic programming solutions can be obtained rapidly and accurately, then we can solve individual–level planning problems conditional on set of market prices. Given these solutions, we can then calculate new market clearing prices. We then iterate between individual and the market until we find a fixed point. The advantage of this approach is that it is very easy to impose restrictions such as borrowing constraints when solving for value functions. The equilibrium naturally satisfies all of the constraints automatically.

- Solutions to partial differential equations. A common approach to solving these problems is to discretize the state space and then solve a system of linear equations. Solutions, however, are very sensitive to the form of the discretization. As a consequence of this a number of problem-specific transformations of the state space have been proposed to ensure that there is a relatively fine discretization over *important* regions of the state space and a coarse grid over less important regions. For example, Duffie (1992) details a common change of variables that has proven successful for interest rate problems. Since interest rates are are in the interval zero to infinity, the state space is large. However, experience and intuition dictate that very large interest rates are highly unlikely. Therefore, a uniformly spaced discrete grid is placed over $1/(1+\gamma x)$ rather than the original variable, x. The tuning parameter γ dictates how many grid points are used for relatively small values of x. This is a very sensible approach. Its primary drawback is that it requires *ex ante* knowledge of the solution. Our adaptive grid generation approach can accomplish exactly the same goal without any knowledge of the ultimate solution.

References

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

Duffie, Darrell (1992), *Dynamic Asset Pricing Theory.* Princeton: Princeton University Press.

Grötschel, Martin and Olaf Holland (1991), “Solution of Large-Scale Symmetric Travelling Salesman Problems,” *Mathematical Programming* **51**, 141-202.

Judd, Kenneth L. (1991), *Numerical Methods in Economics.* manuscript, Stanford University.

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

Stokey, Nancy L. and Robert E. Lucas, Jr. (with Edward C. Prescott) (1989), *Recursive Methods in Economic Dynamics.* Cambridge: Harvard University Press.

Taylor, John B. and Harald Uhlig (1990), “Solving Nonlinear Stochastic Growth Models: A Comparison of Alternative Solution Methods,” *Journal of Business and Economic Statistics* **8**, 1-18.

Appendix: C Programs

1. Value Iteration (with Grid Generation)

```
/*******************************************************************
* Gridval.c:  Program to do value iteration using grid generation
*             Last Modified: July 15, 1993 MT
*******************************************************************/

#include <math.h>
#define TRUE 1
#define FALSE 0
#define ABS(a) (((a) > 0) ? (a) : -(a))
#define MAXI    20001
#define MAXJ    2
#define MAXA    20001

/******************************************************************
*   Utility function and feasibility definitions
******************************************************************/

#define  u(i,j,a)  (pow((-k[a]+temp[i][j]),ro)/ro)
#define valid(i,j,a) ((temp[i][j] - k[a]) > 0 ? TRUE : FALSE)

/******************************************************************
*   Other definitions
******************************************************************/

double max_err;
int max_i,max_j,max_a;
double beta;
double temp[MAXI][MAXJ];
double v1[MAXI][MAXJ];
double v2[MAXI][MAXJ];
double disc_sum[MAXA][MAXJ];
double max,sum,oldval;
int iter, done;
double delta;
int skip;
int place;
int num,num_val;
```

```
int
main ()
{
int           i, j;

double trans[MAXJ][MAXJ],gamma,ro,alpha,k[MAXI],z[MAXJ];
int a,j1;
double low,high,diff;

/* Read in problem parameters */

   ro = .5;
   alpha = 0.33;
   printf("Range of i : ");
   scanf("%d",&max_i);
   printf("Range of j : ");
   scanf("%d",&max_j);
   printf("Range of a : ");
   scanf("%d",&max_a);
   printf("Value for beta : ");
   scanf("%lf",&beta);
   printf("Value for delta : ");
   scanf("%lf",&delta);
   gamma = 1;

/* Define values of high and low states */

   for (j=0;j<max_j;j++)
      z[j] = exp(-.32)+(exp(.32)-exp(-.32))*j/(max_j-1);

/* Find reasonable range for problem */

   low = pow(beta*alpha*z[0]/(1-(1-delta)*beta),1.0/(1.0-alpha));
   high = pow(beta*alpha*z[1]/(1-(1-delta)*beta),1.0/(1.0-alpha));
   diff = (high-low)/(0.8*max_i);
   low -= .1*max_i*diff;
   high += .1*max_i*diff;
   for (a=0;a<max_a;a++)
     k[a] = low + ((high-low)*a)/(max_a-1);

/* Define transition matrix */

   for (j=0;j<max_j;j++) {
     for (j1=0;j1<max_j;j1++) {
        if (j==j1) trans[j][j1] = (.975)*beta;
        else trans[j][j1] = (0.025)/(max_j-1)*beta;
      }
    }

/* Precalculate expensive part of utility function */

    for (i=0;i<max_i;i++) {
      for (j=0;j<max_j;j++) {
        temp[i][j] = (1-delta)*k[i]+gamma*pow( (double) k[i],alpha)*z[j];
      }
    }

/* Define initial value for each state */

    for(i=0;i<max_i;i++) {
      for (j=0;j<max_j;j++) {
        for (a=0;a<max_a;a++) {
          if (valid(i,j,a)) {
            v[i][j] = u(i,j,a)/(1-beta);
            break;
        }
        if (a==max_a) printf("No valid action for %d %d\n",i,j);
      }
    }

/* Define initial grid */

    skip = (max_i-1)/2;
    done = FALSE;
    iter = 0;
/* Outer loop to refine grid */

    while (!done) {
      iter++;
      done = TRUE;

/* Precalculate expensive operation each iteration */

      for(a=0;a<max_a;a+=skip) {
        for (j=0;j<max_j;j++) {
          disc_sum[a][j] = 0;
          for (j1=0;j1<max_j;j1++) {
              disc_sum[a][j] += trans[j][j1]*v1[a][j1];
          }
        }
      }

/* Loop through each state looking for improved value */

      for (i=0;i<max_i;i+=skip) {
        for (j=0;j<max_j;j++) {
          max = -10000000.0;
          for (a=0;a<max_a;a+=skip) {
            if (!(valid(i,j,a))) continue;
            sum = u(i,j,a)+disc_sum[a][j];
            if (sum > max) max = sum;
          }
          v2[i][j] = max;
          if (ABS(max-v1[i][j]) > .000001) done = FALSE;

        }
      }

/* Update values */

      for(i=0;i<max_i;i+=skip) {
        for(j=0;j<max_j;j++) {
          v1[i][j]=v2[i][j];
        }
      }

/* Check if finished? */

      if (done) {
        if (skip > 1) {

/* Refine grid */

          printf("At iteration %d with skip %d\n",iter,skip);
          done = FALSE;
          for (i=skip/2;i<max_i;i+=skip) {
            for (j=0;j<max_j;j++) {
              v1[i][j] = .5*(v1[(i-skip/2)][j]+v1[i+skip/2][j]);
            }
          }
          skip = skip/2;
        }
```

```
    }
  }
  printf("Done Value Iteration (with grids) after %d iterations.\n",iter);

} /* END MAIN */
```

2. Constraint Generation (with Grid Generation)

```
/*******************************************************************
* Gridgen.c:  Program to do constraint generation using grids
*             Last Modified: July 15, 1993 MT
*******************************************************************/


#include "cpxdefs.inc"
#include <math.h>
#define TRUE 1
#define FALSE 0

/******************************************************************
*   Utility function and feasibility definitions
******************************************************************/

#define  u(i,j,a)  (pow((-k[a]+temp[i][j]),ro)/ro)
#define valid(i,j,a) ((temp[i][j] - k[a]) > 0 ? TRUE : FALSE)

/******************************************************************
*   Other definitions
******************************************************************/

#define ABS(a) (((a) > 0) ? (a) : -(a))
#define MAXI    17000
#define MAXJ    2
#define MAXA    17000
#define MACSZ   MAXI*MAXJ*6
#define MARSZ   MAXI*MAXJ
#define MATSZ   MACSZ*(MAXJ+1)
#define COLADD  MAXI*MAXJ*6
#define ENTRYADD COLADD*(MAXJ+1)
#define CSTORSZ 0
#define RSTORSZ 0
```

```
/* Linear program variables */

char     *probname = "dynamic";
int      objsen = -1;
double   objx[MACSZ],objxadd[COLADD];
double   rhsx[MARSZ];
char     senx[MARSZ];
int      matbeg[MACSZ],matbegadd[COLADD];
int      matcnt[MACSZ],matcntadd[COLADD];
int      matind[MATSZ],matindadd[ENTRYADD];
double   matval[MATSZ],matvaladd[ENTRYADD];
double   bdl[MACSZ],bdladd[COLADD];
double   bdu[MACSZ],bduadd[COLADD];
char     *dataname = NULL;
char     *objname = NULL;
char     *rhsname = NULL;
char     *rngname = NULL;
char     *bndname = NULL;
char     *cname = NULL;
char     *rname = NULL;
int      macsz = MACSZ;
int      marsz = MARSZ;
int      matsz = MATSZ;
unsigned cstorsz = CSTORSZ;
unsigned rstorsz = RSTORSZ;
int      lpstat;
double   obj;
double   x[MACSZ];
double   pi[MARSZ];
double   slack[MARSZ];
double   dj[MACSZ];

/* Other variables */

int low,high;
int max_i,max_j,max_a;
double beta;
double delta;

int skip;
int place,place1;
double max;
int done;
int new_col,new_entry;
int max_col,max_entry;
double sum;

double temp[MAXI][MAXJ];
double disc_sum[MAXA][MAXJ];
int tot_var;

int action[MAXI][MAXJ];

int icol[MACSZ];
int jcol[MACSZ];
int acol[MACSZ];

int
main ()
{
struct cpxlp  *lp = NULL;
FILE          *logfile = NULL, *changefile = NULL;
int           status;
int           i, j;
int           toosmall, toobig;

double trans[MAXJ][MAXJ],gamma,ro,alpha,k[MAXI],z[MAXJ];
int a,j1;
double low,high,diff;

/* Set up log file for CPLEX */

   logfile = fopen ("dyn.log", "w");
   if ( !logfile                  ||
        setlogfile (logfile) != 0 ||
        setscr_ind (1)       != 0   ){
      printf ("Failure to connect logging channel.\n");
      goto TERMINATE;
   }


/* Prepare to solve the dual of the stochastic DP */

/* Read in the problem size. */

   matsz = 0;
   macsz = 0;
   marsz = 0;
   ro = .5;
   alpha = 0.33;
   beta = .98;
   printf("Range of i : ");
   scanf("%d",&max_i);
   printf("Range of j : ");
   scanf("%d",&max_j);
   printf("Range of a : ");
   scanf("%d",&max_a);
   printf("Value for beta : ");
   scanf("%lf",&beta);
   printf("Value for delta : ");
   scanf("%lf",&delta);
   gamma = 1.0;
   max_col = MACSZ;
   max_entry = MATSZ;



/* Generate the problem */

   max_a = max_i;
   for (j=0;j<max_j;j++)
      z[j] = exp(-.32)+(exp(.32)-exp(-.32))*j/(max_j-1);
   low = pow(beta*alpha*z[0]/(1-(1-delta)*beta),1.0/(1.0-alpha));
   high = pow(beta*alpha*z[1]/(1-(1-delta)*beta),1.0/(1.0-alpha));
   diff = (high-low)/(0.8*max_i);
   low -= .1*max_i*diff;
   high += .1*max_i*diff;
   printf("Low is %lf high is %lf\n",low,high);
   for (a=0;a<max_a;a++)
     k[a] = low + ((high-low)*a)/(max_a-1);

   for (j=0;j<max_j;j++) {
     for (j1=0;j1<max_j;j1++) {
       if (j==j1) trans[j][j1] = (.975)*beta;
       else trans[j][j1] = (0.025)/(max_j-1)*beta;
     }
   }

/* Generate expensive calculation outside of loop */

   for (i=0;i<max_i;i++) {
     for (j=0;j<max_j;j++) {
        temp[i][j] = (1-delta)*k[i]+gamma*pow(k[i],alpha)*z[j];
      }
    }

/* Put in artificial action */

    matbeg[0] = 0;
    for(i=0;i<max_i;i++) {
      for(j=0;j<max_j;j++) {

/* Use lowest feasible action in each state */

        for (place=0;place<max_a;place++)
          if (valid(i,j,place)) break;
        if (place ==max_a) {
          printf("No valid action for %d %d\n",i,j);
          goto TERMINATE;
        }
        objx[macsz] =  u(i,j,place);
          bdl[macsz] = 0.0;
          bdu[macsz] = INFBOUND;
          matcnt[macsz] = 0;
          if (i!= place) {
            matval[matsz] = 1;
            matind[matsz] = i*max_j+j;
            matsz++;
            matcnt[macsz]++;
          }
          else {
            matval[matsz] = 1-trans[j][j];
            matind[matsz] = i*max_j+j;
            matsz++;
            matcnt[macsz]++;
          }
          for (j1=0;j1<max_j;j1++) {
            if ((i==place)&&(j1==j)) continue;
            matval[matsz] = -trans[j][j1];
            matind[matsz] = place*max_j+j1;
            matsz++;
            matcnt[macsz]++;
          }
          icol[macsz] = i;
          jcol[macsz] = j;
          acol[macsz] = place;
        macsz++;
        matbeg[macsz] = matbeg[macsz-1] + matcnt[macsz-1];
      }
    }


/* RHS */

  for (i=0;i<max_i;i++){
    for(j=0;j<max_j;j++){
      rhsx[marsz] = 1;
      senx[marsz] = 'E';
      marsz++;
    }
  }
  tot_var = macsz;

/* Load in the linear program */

  lp = loadprob (probname, macsz, marsz, 0, objsen, objx, rhsx,
                 senx, matbeg, matcnt, matind, matval,
                 bdl,bdu, NULL, NULL,
                 NULL, NULL, NULL, NULL, NULL,
                 dataname, objname, rhsname, rngname, bndname,
                 NULL, NULL, NULL, NULL, NULL, NULL,
                 max_col, marsz, max_entry, 0, 0, cstorsz,
                 rstorsz, 0);
  if ( lp == NULL )  goto TERMINATE;

/* Optimize the linear program and get solution */

  status = optimize (lp);
  status = solution (lp, &lpstat, &obj, x, pi, slack, dj);
  if ( status )  goto TERMINATE;

/* Set skip to determine active grid */

  skip = (max_i-1)/16;

  done = FALSE;

/* Outer loop to determine level of grid refinement */

  while (!done) {
     status = solution (lp, &lpstat, &obj, x, pi, slack, dj);
     if ( status )  goto TERMINATE;
     done = TRUE;
     new_col = 0;
     new_entry = 0;
     matbegadd[0] = 0;

/* Calculate expensive calculation outside of i loop */

     for(a=0;a<max_a;a+=skip) {
       for (j=0;j<max_j;j++) {
         disc_sum[a][j] = 0;
         for (j1=0;j1<max_j;j1++) {
             disc_sum[a][j] += trans[j][j1]*pi[a*max_j+j1];
           }
       }
     }

/* Determine if there is a better action */

     for (i=0;i<max_i;i+=skip) {
       for(j=0;j<max_j;j++) {
         max = 0.000001;
         place1 = -1;
         for (a=0;a<max_a;a+=skip) {
           if (!(valid(i,j,a))) continue;
           sum = u(i,j,a)-pi[i*max_j+j] +disc_sum[a][j];
           if (sum > max) {
             max = sum;
             place1 = a;
           }
         }

         if (place1 >=0) {

/* Add in better action */

           place = place1;
           done = FALSE;
           objxadd[new_col] =  u(i,j,place);
           bdladd[new_col] = 0.0;
           bduadd[new_col] = INFBOUND;
           matcntadd[new_col] = 0;
           if (i!= place) {
             matvaladd[new_entry] = 1;
             matindadd[new_entry] = i*max_j+j;
             new_entry++;
             matcntadd[new_col]++;
           }
           else {
             matvaladd[new_entry] = 1-trans[j][j];
             matindadd[new_entry] = i*max_j+j;
             new_entry++;
             matcntadd[new_col]++;
           }
           for (j1=0;j1<max_j;j1++) {
             if ((i==place)&&(j1==j)) continue;
             matvaladd[new_entry] = -trans[j][j1];
             matindadd[new_entry] = place*max_j+j1;
             new_entry++;
             matcntadd[new_col]++;
           }
           icol[macsz] = i;
           jcol[macsz] = j;
           acol[macsz] = place;
           macsz++;
           new_col++;
           matbegadd[new_col] = matbegadd[new_col-1] +
             matcntadd[new_col-1];
         }
       }
     }

     if (!done) {

/* If some better actions found, then add in new columns */

       printf("Adding %d columns (size %d) ... ",
              new_col,new_entry);
       tot_var += new_col;
       if (tot_var > max_col) {
         printf("OUT OF SPACE!  Max cols is %d.\n",max_col);
         goto TERMINATE;
       }
       status = addcols (lp, new_col, new_entry,objxadd,
                         matbegadd, matindadd,
                         matvaladd, bdladd,
                         bduadd, NULL);
       printf("done.\n");

   /* Optimize linear program */

       status = optimize (lp);

       if ( status )  goto TERMINATE;
     }

   /* Otherwise, refine grid if possible */

     if (done) {
       if (skip != 1) {
         done=FALSE;
         for(place=0;place < macsz;place++) {
           if (x[place] > .000001) {
             action[icol[place]][jcol[place]]=acol[place];
           }
         }

         new_col = 0;

   /* Add in three actions centered on midpoint of actions for
   adjacent states for each new point */

         for (i=skip/2;i<max_i;i+=skip) {
           for(j=0;j<max_j;j++) {
             low = (action[i-skip/2][j]+action[i+skip/2][j])/2
               -skip/2;
             high = low+skip;
             for (place=low;place<=high;place+=skip/2) {
               if ((place < 0) || (place >= max_a)) continue;
               if (!(valid(i,j,place))) continue;
               objxadd[new_col] =  u(i,j,place);
               bdladd[new_col] = 0.0;
               bduadd[new_col] = INFBOUND;
               matcntadd[new_col] = 0;
               if (i!= place) {
                 matvaladd[new_entry] = 1;
                 matindadd[new_entry] = i*max_j+j;
                 new_entry++;
                 matcntadd[new_col]++;
               }
               else {
```

```
              matvaladd[new_entry] = 1-trans[j][j];
              matindadd[new_entry] = i*max_j+j;
              new_entry++;
              matcntadd[new_col]++;
            }
            for (j1=0;j1<max_j;j1++) {
              if ((i==place)&&(j1==j)) continue;
              matvaladd[new_entry] = -trans[j][j1];
              matindadd[new_entry] = place*max_j+j1;
              new_entry++;
              matcntadd[new_col]++;
            }
            icol[macsz] = i;
            jcol[macsz] = j;
            acol[macsz] = place;
            macsz++;
            new_col++;
            matbegadd[new_col] = matbegadd[new_col-1] +
              matcntadd[new_col-1];
          }
        }
      }

/* Add in adjacent actions for old states */

      for (i=0;i<max_i;i+=skip) {
        for(j=0;j<max_j;j++) {
          low =action[i][j]-skip/2;
          high = low+skip;
          for (place=low;place<=high;place+=skip) {
            if ((place < 0) || (place >= max_a)) continue;
            if (!(valid(i,j,place))) continue;
            objxadd[new_col] =  u(i,j,place);
            bdladd[new_col] = 0.0;
            bduadd[new_col] = INFBOUND;
            matcntadd[new_col] = 0;
            if (i!= place) {
              matvaladd[new_entry] = 1;
              matindadd[new_entry] = i*max_j+j;
              new_entry++;
              matcntadd[new_col]++;
            }
            else {
              matvaladd[new_entry] = 1-trans[j][j];
              matindadd[new_entry] = i*max_j+j;
              new_entry++;
              matcntadd[new_col]++;
            }
            for (j1=0;j1<max_j;j1++) {
              if ((i==place)&&(j1==j)) continue;
              matvaladd[new_entry] = -trans[j][j1];
              matindadd[new_entry] = place*max_j+j1;
              new_entry++;
              matcntadd[new_col]++;
            }
            icol[macsz] = i;
            jcol[macsz] = j;
            acol[macsz] = place;
            macsz++;
            new_col++;
            matbegadd[new_col] = matbegadd[new_col-1] +
              matcntadd[new_col-1];
          }
        }
      }
      skip = skip/2;
      printf("NEW PHASE (skip %3d) Adding %d columns (size %d) ... ",
             skip,new_col,new_entry);
      tot_var += new_col;
      if (tot_var > max_col) {
        printf("OUT OF SPACE!  Max cols is %d.\n",max_col);
        goto TERMINATE;
      }
      status = addcols (lp, new_col, new_entry,objxadd,
                        matbegadd, matindadd,
                        matvaladd, bdladd,
                        bduadd, NULL);
      printf("done.\n");
      status = optimize (lp);

      if ( status )  goto TERMINATE;
    }
  }

 } /* While !done */

/* Finished: print out results. */
```

```
   printf("Completed successfully!\n");
   cpxmsg (cpxresults, "\nSolution status = %d\n", lpstat);
   cpxmsg (cpxresults, "Solution value  = %f\n", obj);
   cpxmsg(cpxresults, "Number of pivots: %d\n",getitc(lp));
   cpxmsg(cpxresults,"Number of variables = %d\n",tot_var);
   printf("Final number of columns : %d\n",macsz);


TERMINATE:

   printf("Terminating\n");
   /* Flush all channels before deleting files as destinations. */

   flushchannel (cpxresults);
   flushchannel (cpxlog);
   flushchannel (cpxerror);
   flushchannel (cpxwarning);

   /* Delete log file from all four channels and close. */

   setlogfile (NULL);
   if (logfile) fclose (logfile);

   /* Delete changefile from results channel, then close. */

   delfpdest (cpxresults, changefile);
   if (changefile) fclose (changefile);


} /* END MAIN */
```