12 – Numerical Issues #2: Acceleration methods for solving infinite-horizon DP problems

AGEC 642 - 2024

I. Introduction

The value function iteration method to solve infinite-horizon DP problems converges linearly at a rate proportional to $1/\beta$: the greater the discount rate (i.e., the smaller that β is) the faster your problem will converge. For relatively small problems this is not a problem. As the state space grows, however, it is not uncommon for the method to take hundreds or thousands of iterations before convergence is achieved, meaning hours or even days to find the solution. Hence, the value-function iteration method and the related Euler equation iteration method covered in the previous lecture become less applicable for large problems. Fortunately, there are alternative methods that can be used to solve large DP problems that can be applied when the curse of dimensionality begins to affect your ability to solve a problem.

Improved computational efficiency is an area of great growth in the literature in recent years, spurred on by the increasing use of dynamic programming in econometrics. As a result, these notes are becoming increasingly dated. Some papers that have not been incorporated here include Mrkaic (2002), Judd et al. (2009), and Cai et al. (2013). Nonetheless, to explore the state of the art, some understanding of more basic methods is required. Hence, these notes should be seen primarily as providing that background on more rudimentary methods, though the methods here can be fruitfully employed for relatively small DP problems.

II. Accelerated methods of successive approximation of the value function

Recall that in *finite* horizon dynamic programming problems, backward induction is used in which $V(x_t, t)$ is found by solving the Bellman's equation with $V(x_{t+1}, t+1)$ on the right-hand side. The process is similar for infinite-horizon problems, but each time you solve the Bellman's equation you should think of it as having the previous approximation of the value function on the right-hand side, which will be used to obtain the next approximation. In this section, we consider several methods to obtain better guesses at the infinite-horizon value function between iterations.

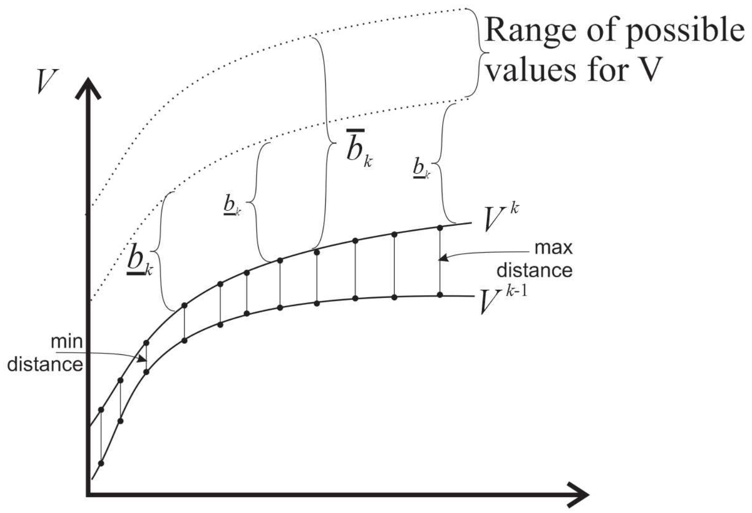
A. Successive approximation with error bounds

In some problems, the successive approximation algorithm can be accelerated by the McQueen-Porteus Error Bounds. This algorithm takes advantage of the contraction mapping feature of the successive approximation algorithm. (This can be applied directly for DD problems. The method can be adapted for CD or CC problems too.

Suppose that you have solved the Bellman's equation for all $x \in X$ with $V^{RHS}(x)$ on the right-hand side, yielding the vector $V^{LHS}(x)$. As you will recall, from the contraction mapping theorem referred to in Lecture 8, we know that we can obtain bounds for the true infinite-horizon value function $V(x)$:

$$V^{LHS}(x) + \underline{b}_k \leq V(x) \leq V^{LHS}(x) + \overline{b}_k$$

where $\underline{b}_k = \frac{\beta}{1-\beta}\min_x\left[V^{LHS}(x) - V^{RHS}(x)\right]$ and $\overline{b}_k = \frac{\beta}{1-\beta}\max_x\left[V^{LHS}(x) - V^{RHS}(x)\right]$.[1]



Hence, after each successive approximation loop, we know the bounds between which the true value function must lie. Furthermore, these bounds will get tighter as the number of iterations increases.

The McQueen-Porteus Error Bounds approach is to make use of these bounds to get a better update of the value function. Specifically, instead of simply setting $V^{RHS}(x)=V^{LHS}(x)$ as in a standard value-function iteration, use the midpoint of your bounds, i.e., for the next iteration use

$$V^{RHS'}(x) = V^{LHS}(x) + \left(\frac{\underline{b}_k + \overline{b}_k}{2}\right).$$

Bertsekas (1987) has shown that this approach can significantly speed up the conversion.[2] Further, this approach can be coded quite easily, requiring only a couple of extra lines of programming beyond the standard successive approximation approach, so it is often worth using.

B. Pre-Gauss-Seidel method (not recommended, but of pedagogical interest)

For DD problems or when using rounding for CD problems, the standard successive approximation method can also be accelerated by making use of the new information about the value function immediately as you are looping through the points in your state space. This technique is known as the Pre-Gauss-Seidel method (Judd 1998).

In the *k*th iteration of the standard successive-approximation algorithm, the stage loop consists of solving the following problem for each point in your state space:

[1] Note that $\beta/(1-\beta) = r^{-1}$.

[2] Rust (1996) notes that Bertsekas also showed that if the optimal Markov transition matrix has multiple ergodic sets there is little improvement in the speed of convergence.

$$V^k(x_t) = \max_{z_t} E\left[u(z_t, x_t, \varepsilon_t) + \beta V^{k-1}(x_{t+1})\right].$$

The purpose of the kth iteration of the successive approximation algorithm is to obtain an improved estimate of V^*, using $V^{k-1}(\cdot)$ on the right-hand side of the Bellman's equation. So, for example, if attempting to find $V^k(x_i)$, the value function at the ith point in the state space, the algorithm has already found $V^k(x_j)$, $j=1,\ldots, i-1$. When the $V^k(x_i)$ has been found, we have i new estimates of $V^*(\cdot)$ that are better than $V^{k-1}(\cdot)$. The pre-Gauss-Seidel method takes advantage of this by using on the RHS of the Bellman's equation $V^k(x_j)$ for $j<i$ and $V^{k-1}(x_j)$ for $j\geq i$.

This method is also easy to implement. Recall that in the successive approximation algorithm, you need to retain two value function arrays, $V^k(x)$ and $V^{k-1}(x)$ for all $x\in X$. In this method, we only need to hold onto one value function array, replacing the old estimate of $V(x_i)$ with the new estimate as we loop through the points. The difference between the value function approximations from one iteration to the next that is used to test for convergence needs to be calculated incrementally at each point in the state space prior to replacing the old estimate with the new estimate.

C. Policy Iteration & Modified Policy Iteration (review, covered in Lecture 9)

An alternative method for solving infinite-horizon DP problems is a technique known as policy iteration. This is the approach that is used by Burt and Allison (1963) that we saw in Lecture 9. Like successively approximating the value function, this technique has strong intuitive appeal.

The basic idea of policy iteration involves three steps:

1. Pick some arbitrary value function, $V^0(x)$, and find a policy rule, $z^0(x)$, that solves the associated Bellman's equation at each point in the state space.
2. Using $z^0(x)$ and the associated candidate Markov transition matrix, find $V^1(x)$ using the matrix operations we saw in Lecture 9. This yields the value that would be obtained if the rule $z^0(x)$ were followed indefinitely.
3. Put $V^1(x)$ on the right-hand side and solve Bellman's equation for all x again to obtain $z^1(x)$. Repeat steps 2&3 until $z^k(x) \approx z^{k-1}(x)$.

Policy iteration has many similarities to the value-function iteration method. The key difference lies in step 2 where the update of the value function is obtained. Let us look at this step in detail. Let P^k be the candidate Markov transition matrix that follows from the decision rule $z^k(x)$. The i, jth cell of P^k indicates the probability of being in state j next period given that you're in state i this period and you follow the policy $z^k(x)$. The present-discounted value of following this policy can be written

$$V^k(x) = Eu\left(z^k(x), x\right) + \beta \cdot P^k V^k(x). \tag{1}$$

Except for β, everything in this equation is either a matrix or a vector. If the state variable takes on n values, then $V^k(x)$ and $u\left(z^k(x), x\right)$ are $n\times 1$ vectors and P^k is an $n\times n$ matrix. Finding the values of the $V^k(x)$ can, therefore, be found by solving this (possibly very large) system of linear equations. Using matrix algebra, we see that

$$V^k(x) = \left[I - \beta \cdot P^k\right]^{-1} Eu\left(z^k(x), x\right). \tag{2}$$

When the problem being considered is relatively small ($|X|<500$)[3] with a discount factor relatively close to 1 ($\beta>0.95$) the policy iteration method is regarded as one of the fastest methods for solving infinite-horizon DP problems (Rust, 1996). Policy iteration typically requires many fewer iterations than successive approximations of the value function. However, because of the need to invert a matrix, each iteration is much more computationally intensive. For small problems, therefore, it may be slower than value function iteration.

If there are *n* total combinations of state variable values, then solving the system of equations in (2) involves inverting an $n \times n$ matrix or solving a system of *n* equations for *n* unknowns – computationally intensive tasks. If you have access to a library of subroutines, you will certainly be able to invert large matrices or solve (1) directly.[4]

Regardless of the approach that you use to solve the problem, it is recommended that you use step 2 of the policy iteration algorithm to obtain your final estimate of the value function. This will ensure that the values you present are associated with an infinite horizon.

As noted in the discussion of the Burt and Allison paper, an alternative to inverting the matrix in (2) is to approximate the value function by applying the decision rule $z^k(x)$ numerous times. Rust (1996) points out that for large problems approximating the value function in this manner can be done at substantially less computational expense an approach known as the *Modified Policy Iteration Method.* That is, we could find V^k by repeatedly applying the system of equations

$$V^{k,l}(x) = Eu\left(z^k(x), x\right) + \beta \cdot P^k V^{k,l-1}(x) \tag{3}$$

for $l=0,1,2,\ldots,m$. Once $V^{k,l}(x)$ is quite close to $V^{k,l-1}$, then we stop and apply step 3 of the policy iteration method once again. Rust also suggests that the modified policy iteration method can be sped up by using McQueen-Porteus error bounds in the updating step, (3).[5]

D. Summary of solution methods discussed so far.

The four methods discussed above fall into essentially the same pattern, presented in the pseudo-code below.

[3] The notation $|X|$ indicates the cardinality of X, i.e., the total number of states that you are considering.

[4] A matrix inversion subroutine is included in the VB Matrix Operations subroutines available from class web page.

[5] Mrkaic (2002) shows how *Krylov methods* can be used to greatly accelerate policy iteration methods, resulting in convergence that can be up to an order of magnitude faster than policy iteration and value function iteration.

```
Stage Loop
Set $V_{RHS}(x)=0$ (or some other starting value) for every state $x \in X$.
For iteration 1,2,…, max iter
    State Loop
    **State loop.**
    For every point in your state space, $x_t \in X$
        **Solve Bellman's Equation:**
        Find the $z$ that maximizes
            $E\{u(\cdot)+\beta V_{RHS}(\cdot)\}$
        This can be done using a control loop.
        This will yield $z^*$ and $V_{LHS}(x_t)$
                {if using Pre-Gauss Seidel, set $V_{RHS}(x_t)= V_{LHS}(x_t)$ immediately}
    **Save (candidate) optimal policy**
            $R = u(z^*)$ and
            $P$ = the Markov transition matrix associated with $z^*$
    **Check for Convergence** of the value function. i.e.
            find $diff = \max_x \left| V_{RHS}(x) - V_{LHS}(x) \right|$.[6]
            If $diff$<convergence criterion, exit stage loop
    **Update the Value Function (choose 1 of these 4 methods)**
            Option 1: Value Function iteration:  $V_{RHS} = V_{LHS}$
            Option 2: Value Function iteration with error bounds:
                $V_{RHS} = V_{LHS} + \left( \frac{\underline{b}_k + \overline{b}_k}{2} \right)$
            Option 3: Policy Iteration: $V_{RHS}=(I-\beta P)^{-1}\cdot R$
            Option 4: Modified Policy Iteration:
                iterate $V^k = R+\beta P\cdot V^{k-1}$
                until $V^k \approx V^{k-1}$
                $V_{RHS}=V^k$
    **End of Stage loop**: If convergence is not satisfied, start next stage
```

III. Comparison of the computational time of various methods

Rust (1996) compares the computational time of some of the methods outlined above. He uses a variety of numerical approaches to solve a problem with a known analytical result, which allows him to compare how closely the numerical results get to the true value function, V^*. The results presented in Table 1 and Figure 1 are quite revealing of the relative numerical efficiencies of the various methods. When β=0.95 in his problem, the successive approximation algorithm took 48 seconds of CPU. Applying error bounds cut the time to 29.7 seconds and the modified policy iteration method cut it even further

[6] Convergence can also be checked by looking at the difference in the optimal policy function, i.e., $diff_z = \max_x \left| z^{*,k}(x) - z^{*,k-1}(x) \right|$. When policy iteration is used, this criterion should be used and continued until $diff_z = 0$.

to 21.8 seconds. However, when β=0.9999, as when the time-step is daily rather than annual, the successive approximation algorithm took over 4,600 seconds to converge compared with 75.7 and 50.0 seconds for the error-bounds and modified policy iteration approaches respectively. From these results, we see that substantial gains, indeed most of the computational gains possible, can be achieved at a rather small cost, simply by applying the McQueen-Porteus error bounds, one of the easiest of the above-mentioned methods.

	β=0.95			β=0.9999		
	Iterations	CPU Seconds	$\|V-V^*\|$	Iterations	CPU Seconds	$\|V-V^*\|$
Successive Approximations	114	48.9	571.7	>10,000	>4600	>30,000
Error Bounds	65	29.7	503.4	166	75.7	4.4E^{-1}
Policy Iteration	6	46.1	1.9E^{-9}	8	71	2.9E^{-7}
Modified Policy Iteration	5	21.8	52.3	11	50	174.9

Table 1: Comparison of run-times for various methods
(Source: Tables 5.1 and 5.2 in a draft copy of Rust, 1996, page 69)

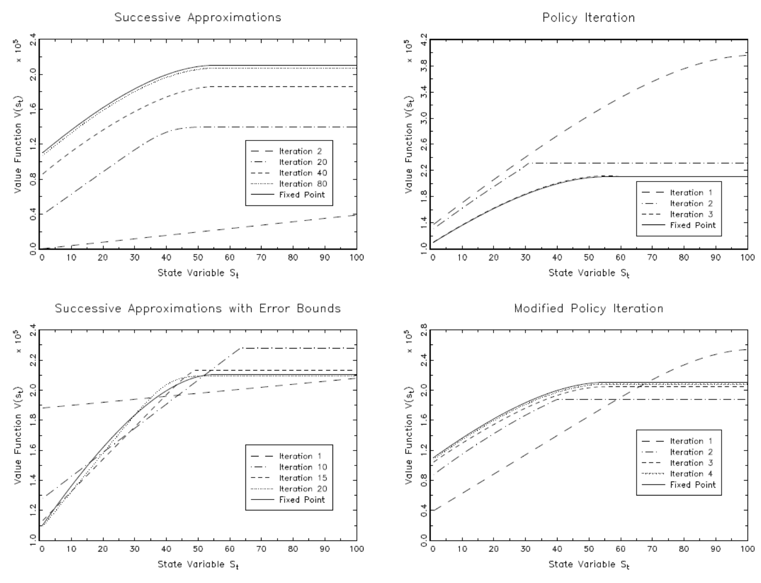


Figure 5.2: Convergence of V_t to V when $\beta = .95$

Figure 1: Comparison of the value function over various iterations
(Source: Figure 5.2 in Draft version of Rust, 1996, p. 69.)

IV. DP as LP

For DD problems, it is possible to redefine our optimization problem as a linear programming (LP) minimization problem. To see this, we need to explicitly use the Markov transition matrix P. Let $P=[q_{ij}(z)]_{\text{for all } i,j}$. That is, q_{ij} is the probability that we will be in state *j* next period given that we're in state *i* this period and we follow the decision rule *z*. Hence, the Bellman's equation can be written

$$V(x_i) = \max_z Eu(z, x_i, \varepsilon_t) + \beta \sum_j q_{ij}(z) V(x_j).$$

When the solution is found, the $V(\cdot)$ function on the left, must be the same as the one on the right. This problem can be equivalently written using a *dual* representation:

$$\min_{z, V(x_i) \forall x_i} \sum_{x \in X} V(x_i)$$

$$V(x_i) \geq Eu(z, x_i, \varepsilon_t) + \beta \sum_j q_{ij}(z) V(x_j(x_i, z, \varepsilon)).$$

This problem is linear in the $V(x_i)$ and can, therefore, be solved using a linear programming algorithm. Trick and Zin (1997) found that it can be solved quite efficiently using constraint generation techniques (Rust 1996).[7]

V. Collocation methods[8]

The approaches that are outlined above are directly applicable to DD problems and most can be adapted to CD or CC problems. Collocation methods, on the other hand, are specifically designed for problems with a continuous state space.

Recall from Lecture 11 that one way to approximate the value function in problems with a continuous state space is to define a continuous function to approximate the true value function, $\tilde{V}(x;c)$, where *c* is a vector of coefficients. If we followed a successive approximation approach, convergence would be reached when $\tilde{V}(x, c^k) \approx \tilde{V}(x, c^{k-1})$ where c^k is the *k*th set of coefficients of the function $\tilde{V}$.

If $\tilde{V}$ is a polynomial (e.g., a standard polynomial, or a Chebyshev polynomial – see Lecture 11 on CD problems), then it can be written

$$\tilde{V}(x, c) = \sum_{i=1}^{n} c_i \phi_i(x),$$

where $\phi_i(x)$ is the *i*th order polynomial of the variable *x* (e.g., x^i for standard polynomials) and *n* is the order of the polynomial being used. At the infinite-horizon optimum, the following Bellman's equation must be satisfied at each of the *m* points in *X*:

$$V(x_i; c) = \sum_{j=1}^{n} c_j \phi_j(x_i) = \max_z \left[u(x_i, z) + \beta \sum_{l=1}^{m} q_{il}(z) \sum_{j=1}^{n} c_j \phi_j(x_l) \right]$$

or,

$$\sum_{j=1}^{n} c_j \phi_j(x_i) - \max_z \left[u(x_i, z) + \beta \sum_{l=1}^{m} q_{il}(z) \sum_{j=1}^{n} c_j \phi_j(x_l) \right] = 0. \qquad (4)$$

[7] The methods proposed by Cai et al. (2013) are described a nonlinear programming approach, analogous to the LP approach described here, which allows the authors to handle continuous state and control variables.

[8] These methods were originally developed by Judd 1992. The discussion here is based on Miranda and Fackler section 6.8. These sources provide substantially more detail on their implementation.

The problem of finding the infinite-horizon value function, therefore, becomes one of choosing the coefficients, *c*, to set each of the equations in (4) to zero. In principle, a nonlinear root-finding algorithm could be used to solve this problem. Newton's method or function iteration methods (see Judd, 1998 or Miranda and Fackler) could be used to choose the coefficients *c* that solve these equations.

Of course, one of the challenges of solving (4) is that there is a maximization problem embedded in the equation. While in practice this creates substantial difficulties, in principle you can imagine having a subroutine that solves the Bellman's equation for any point in *x* and any set of coefficients, *c*. In other situations, since $V(\cdot)$ is approximated by an analytical function, if *z* is continuous, it may be possible to write z_t^* as a closed form expression of x_t and the vector of coefficients, *c*; i.e., the solution to the problem

$$\partial\left(u\left(x_i,z\right)+\beta\sum_{l=1}^{m}q_{il}\left(z\right)\sum_{j=1}^{n}c_j\phi_j\left(x_l\right)\right)\Big/\partial z=0\,.$$

Collocation methods have been found to be among the most efficient means of solving continuous DP problems. In a comparison of solution methods conducted by Christiano and Fisher, it was found that collocation methods were orders of magnitude faster than some other techniques.

VI. A quick introduction to some simulation-based approaches

Finally, methods that use Monte Carlo sampling as part of the solution method are also an option. I do not attempt to explore any of these methods in detail here, but instead, point you in the direction of the next generation of methods for solving dynamic programming problems and demonstrate that there is still work to be done in this area.

First, Rust (1997) developed an approach that essentially involved using Monte Carlo sampling to estimate the value function at each iteration. Like the use of simulation methods in econometrics, Rust's approach takes advantage of the fact that randomization can be used to find approximate solutions to intractably large integration problems.

Second, Powell (2007) advocates the use of "Approximate Dynamic Programming" (ADP). This is a process in which the value function is approximated using forward simulations of optimal paths and successive updates of the value and policy functions. While Powell's approach has the disadvantage of lacking formal proofs of convergence, it has been applied to very large dynamic programming problems that would be impossible to solve using more standard methods. (This will be explored later in this course).

Finally, Judd, Maliar, and Maliar (2009) offer an approach that appears to have much in common with Powell's ADP algorithm. A key advantage of their approach is that the analyst does not have to evaluate substantial portions of the state space that would never be reached under an optimal policy. They show that this can substantially reduce the dimensionality of the problem.

VII. Final thoughts and recommendations

So, what is the applied economist to do? Various methods can be used to solve infinite-horizon DP problems. Which should one choose? The answer to this question depends upon the needs of the analyst. First, if your algorithm takes 10 seconds as opposed to 0.01 seconds, it probably does not matter much what approach you take if you only have to solve the problem once. Keep it simple. For example, Anderson, Kellogg, and Salant (2018) use the successive approximation algorithm to solve a simple dynamic programming problem for their mostly conceptual analysis of the oil market. On the other hand, in some applications of DP (see econometrics methods discussed in Lecture 13), the DP problem must be solved many times – so speed is critical. As your state space grows either in dimension or in the number of points in each dimension, then computational efficiency becomes more of an issue. Some of the basic methods discussed here can help greatly and should be applied. Further, as noted above, using the policy iteration final step to calculate the final value function is a good practice. When the problem becomes larger and larger and/or precision becomes more and more important, analysts should look to the cutting edge for the most efficient available approaches.

VIII. References

Anderson, Soren T., Ryan Kellogg, Stephen W. Salant. 2018. "Hotelling Under Pressure" *Journal of Political Economy.* DOI: 10.1086/697203

Bertsekas, Dimitri P. 1987. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice Hall.

Cai, Yongyang, Kenneth L. Judd, Thomas S. Lontzek, Valentina Michelangeli, and Che-Lin Su. 2013. Nonlinear Programming Method for Dynamic Programming. *NBER Working Paper* 19034.

Christiano, L.J., and J.D.M Fisher. 1997. "Algorithms for solving dynamic models with occasionally binding constraints." NBER Working Paper No. t0218. Summary results from this paper are presented in a pre-publication version of Rust (1996).

Judd, Kenneth L. 1996. *Numerical Methods in Economics*. Cambridge, Mass.: The MIT Press.

Judd, Kenneth, Lilia Maliar, and Serguei Maliar. 2009. Numerically Stable Stochastic Simulation Approaches for Solving Dynamic Economic. *NBER Working Papers* 15296.

Miranda, Mario J., and Paul L. Fackler. 2002. *Applied Computational Economics and Finance.* Cambridge, Mass.: MIT Press.

Mrkaic, Mico. 2002. Policy Iteration Accelerated with Krylov Methods. *Journal of Economic Dynamics and Control* 26(4):517-45.

Powell, Warren B. 2007. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Hoboken, New Jersey: John Wiley & Sons, Inc.

Rust, John. 1996. Numerical Dynamic Programming in Economics. In H. Amman, D. Kendrick and J. Rust (eds.), Handbook *of Computational Economics*. New York: North Holland.

Rust, John. 1997. Using Randomization to Break the Curse of Dimensionality. *Econometrica* 65(3):487-516.

Trick, M.A., and S.E. Zin (1997). Spline approximations to value functions — linear programming approach. *Macroeconomic Dynamics* 1:255–277.

IX. Readings for next class

Kamien and Schwartz, pp. 202-217