Note on Neoclassical Growth Model: Value Function Iteration + Discretization

Makoto Nakajima, UIUC
January 2007

1 Introduction

We study the solution algorithm using value function iteration, and discretization of the state space.

2 Bellman Equation and Value Function Iteration

It is known that a solution to the following recursive problem is identical to a solution to the original sequential formulation (Problem 1). It is called Bellman's Principle of Optimality. For more formal proof, see Stokey et al. (1989).

Problem 1 (Neoclassical Growth Model: Recursive Formulation)

$$V(K) = \max_{C,K'} \{u(C) + \beta V(K')\}$$

subject to

$$C + K' = zF(K, 1) + (1 - \delta)K$$
$$C \geq 0$$
$$K' \geq 0$$

where, as usual, a prime means the variable in the next period. If we define the Bellman operator B(V) which updates a value function V using the Bellman equation above, we can show the following properties (again, see Stokey et al. (1989) for a formal proof),

1. V^* such that $B(V^*) = V^*$ exists and is unique.
2. $V^* = \lim_{t\to\infty} B^t(V^0)$ for any continuous function V^0. In addition, $B^t(V^0)$ converges to V^* monotonically.

In other words, if we supply an initial guess V^0 and keep applying the Bellman operator, we can asymptotically get to the solution V^* of the Bellman equation. **Value function iteration** is the solution method which uses the properties.

3 Discretization

However, there is a problem. The value function is defined over a continuous state space (space of K), but computers cannot deal with that, unless the function can be represented by finite number

of parameters. Therefore, we need to approximate the value function so that a computer can store the approximated value function and thus we can implement value function iteration algorithm on a computer.

The simplest way to approximate a function over a continuous domain is to represent the original function as a finite set of points. Suppose $V(K)$ is defined over $[\underline{K}, \overline{K}]$. We can set an integer n_k and put n_k discrete points $\{K_0, K_1, K_2, ..., K_{n_k}\}$ over $[\underline{K}, \overline{K}]$. With this discrete domain, the value function can be represented at a set of n_k points $\{V_i\}_{i=1}^{n_k}$, where $V_i = V(K_i)$. This is definitely what a computer can handle.

Remember that the choice of the agent in each period is the capital stock in the next period K' and thus chosen from the same set as K. Therefore, we can restrict the set of the choice to the set of discrete points on K that we created.

Now we can construct an algorithm to solve the neoclassical growth model, using value function iteration and discretization of the state space.

Algorithm 1 (Neoclassical Growth Model: Value Function Iteration and Discretization)

1. *Set n_k (number of grid points), $\underline{K}$ (lower bound of the state space), $\overline{K}$ (upper bound of the state space), and ϵ (tolerance of error). n_k is determined weighting the tradeoff between speed and precision. $\underline{K}$ can be slightly higher than 0, as $K = 0$ is a steady state with 0 consumption forever. $\overline{K}$ can be set slightly above the steady state level (which can be computed analytically), assuming all we are interested in is the dynamics below the steady state level.*

2. *Set grid points $\{K_1, K_2, ..., K_{n_k}\}$. Default is to set equidistance grid points. The value function can be stores as a set of n_k points $\{V_i\}_1^{n_k}$*

3. *Set an initial value of $V^0 = \{V_i^0\}_{i=1}^{n_k}$. A trivial initial condition is $V^0 = 0$. More sophisticated one is to compute the value when the agent is saving K_i capital stock each period and assign it as the value associated with K_i. Be careful in using rather sophisticated guess, because in some models, you might end up assigning infeasible decision for some states.*

4. *Update the value function and obtain $V^1 = \{V_i^1\}_{i=1}^{n_k}$. More specifically, do the following steps for each of $i = 1, ..., n_k$.*

   (a) *Compute the value conditional on the choice K_j. Call it $V_{i,j}^1$. It can be computed using the Bellman Equation, as follows:*

   $$V_{i,j}^1 = u(zF(K_i, 1) + (1 - \delta)K_i - K_j) + \beta V_j^0$$

   If the consumption $C = zF(K_i, 1) + (1 - \delta)K_i - K_j$ turns out to be negative, assign a very large negative number to $V_{i,j}^1$ so that K_j will never be an optimal choice.

   (b) *Choose j which gives the highest value among $\{V_{i,j}^1\} j = 1^{n_k}$. Call it V_i^1. Store the optimal decision as $j = g_i \in \{1, 2, ..., n_k\}$.*

   After implementing the procedure above for $i = 1, ..., n_k$, We can construct a new (discretized) value function as $V^1 = \{V_i^1\}_{i=1}^{n_k}$.

5. *Compare V^0 and V^1 and compute the distance d. One way to define the error is to use the sup norm, as follows:*

   $$d = \max_{i \in \{1,2,...,n_k\}} |V_i^0 - V_i^1|$$

6. *If $d > \epsilon$, the error is not small enough. Update the value function using:*

$$V^0 = V^1$$

and go back to step 4.

7. *If $d \leq \epsilon$, then we find our approximated optimal value function. The value function is $V^1 = \{V_i^1\}_{i=1}^{n_k}$. The optimal decision rule is $g = \{g_i\}_{i=1}^{n_k}$.*

8. *Check if the bounds of the state space is not binding. In particular, make sure $g_i \in \{2, 3, ..., n_k - 1\}$. If not ($g_i = 1$ or $g_i = n_k$ for some i), the bounds of the state space is too tight. Relax the bounds and restart.*

9. *Make sure that ϵ is small enough. Reduce ϵ and redo all the process. If the resulting optimal decision rule is substantially different from the originally obtained one, the initial ϵ might be too large. Keep reducing ϵ until the results are insensitive to a reduction in ϵ.*

10. *Make sure that n_k is large enough. Increase n_k and redo all the process. If the resulting value function or the optimal decision rule is substantially different from the original one, the initial n_k might be too small. Keep increasing n_k until the results are insensitive to an increase in n_k.*

Below are some remarks:

1. If you have some information about the initial guess, use the information to give a good initial guess. The quality of the initial guess is crucial in computational time. One thing you can do is to run at first with small n_k and use the obtained optimal decision to construct a good guess for the next run with larger n_k.

2. All the languages must have a built-in function to implement step 4(b). In fortran, the function which picks up the largest value out of a vector is called `maxloc`.

4 Speed-up the Algorithm

There are three ways to speed-up the algorithm that we have studied. The most robust method is called **Howard policy iteration** algorithm. This method works regardless of the properties of the problem we are dealing with. The second option is to **exploit properties** of the value function or the optimal decision rule. Of course, we have to know some properties of the value function or the optimal decision rule to exploit. We will see two methods in this category, associated with two different properties. The last option is to implement **local search**. It is an atheoretical method. There is no guarantee that this method works, for some problem, but it turns out that for many problems it turns out to be useful. But you have to be really careful in using the method. We will see the four methods (in three categories above) one by one.

4.1 Howard's Policy Iteration

The most time consuming part of Algorithm 1 above is to find an optimal choice for each state, in each iteration. If we have an decision rule which is not far from the optimal one, we can apply

the already obtained decision rule many times to update the value function many times, without solving the optimal decision rule. This procedure lets us approach to the optimal value function faster by updating the value function much more times than finding the optimal decision rules. This is the idea of Howard's policy function iteration.

In using the Howard's algorithm, it is needed to set n_h, which determines how many times we update the value function using the already obtained decision rule. If the decision rule is close to the optimal decision rule, higher n_h implies faster convergence (because we can skip a lot of optimization steps), but a decision rule which is obtained at the beginning or in the middle of the iteration process might not be close to the optimal one. In this case, too high n_h might hurt the speed of convergence. This is because applying a wrong decision rule many times might put the value function far away from the optimal one. So the bottom line is, you might want to do try and error. Change n_h to different values and see how the speed of convergence changes. That's how we can get a good sense of how to pick n_h.

Suppose we picked n_h. Howard's policy iteration algorithm modifies the original algorithm by inserting the steps specified below between step 4 and step 5. Let's call the additional step as step 4.5. Step 4.5 is the following:

Algorithm 2 (Howard's Policy Iteration Algorithm)

4.5 Set $V^{1,1} = V^1$. Then update the value function $V^{1,t}$ by applying the following steps nh times and obtain V^{1,n_h}. Replace V^1 by V^{1,n_h} and go to step 5.

(a) For each of $i = 1, 2, ..., n_k$, we have the optimal decision $g_i \in \{1, 2, ..., n_k\}$ associated with each of i.

(b) Update the value function from $V^{1,t}$ to $V^{1,t+1}$ using the following modified Bellman Equation for each of $i = 1, 2, ..., n_k$:

$$V_i^{1,t+1} = u(zF(K_i, 1) + (1 - \delta)K_i - K_{g_i}) + \beta V_{g_i}^{1,t}$$

An interesting case is $n_h = \infty$. In other words, you are finding a fixed point to the following equation, given $\{g_i\}_{i=1}^{n_k}$:

$$V_i^{1,\infty} = u(zF(K_i, 1) + (1 - \delta)K_i - K_{g_i}) + \beta V_{g_i}^{1,\infty}$$

There are two ways to solve for $V_i^{1,\infty}$. You could nest another iteration algorithm to find $V_i^{1,\infty}$ (keep updating $V_i^{1,t}$ until the error between $V_i^{1,t}$ and $V_i^{1,t+1}$ gets sufficiently small), or you could solve for $V_i^{1,\infty}$ by constructing a matrix representation of the Bellman operator and solving for $V_i^{1,\infty}$ (you need matrix inversion). If you are using Matlab, the second method might be a better one, as Matlab is relatively fast in matrix operation.

4.2 Exploiting Monotonicity of Optimal Decision Rule

The basic idea of this algorithm is to reduce the number of grids that are searched when looking for an optimal choice g_i. It is possible when we know (correctly, for current problem) that the optimal decision rule is an increasing function in K. In other words, take K_i and K_j where $K_i < K_j$. If the associated optimal decisions are g_i and g_j, then $g_i \leq g_j$. Notice that it could be that $g_i = g_j$. This happens when the grids are not fine enough.

Therefore, if we have obtained the optimal choice associated K_i (g_i), and suppose we want to find g_j which is associated with $K_j > K_i$, we do not need to search grids $\{1, 2, ..., g_{i-1} - 1\}$ because of the monotonicity property of the decision rule.

Specifically, step 4 of algorithm 1 is going be replaced by the following:

Algorithm 3 (Exploiting Monotonicity of Optimal Decision Rule)

4 Update the value function and obtain $V^1 = \{V_i^1\}_1^{n_k}$. More specifically, do the following steps for each of $i = 1, ..., n_k$.

(a) Find the lower bound of the optimal choice $\underline{j}$. For $i = 1$, $\underline{j} = 1$. For $i > 1$, $\underline{j} = g_{i-1}$.

(b) Compute the value conditional on the choice K_j for $j = \underline{j}, \underline{j}+1, ..., n_k$. Call it $V_{i,j}^1$. It can be computed using the Bellman Equation, as follows:

$$V_{i,j}^1 = u(F(zK_i, 1) + (1 - \delta)K_i - K_j) + \beta V_j^0$$

If the consumption $C = zF(K_i, 1) + (1 - \delta)K_i - K_j$ turns out to be negative, assign a very large negative number to $V_{i,j}^1$.

(c) Choose j which gives the highest value among $\{V_{i,j}^1\} j = \underline{j}^{n_k}$. Call it V_i^1. Also store the optimal decision j as $g_i = j$.

4.3 Exploiting Concavity of the Value Function

The approach is similar to the previous trick. In our current model, the maximand in the Bellman Equation is strictly concave in the choice K'. Suppose we search for the optimal K' by looking at the conditional value $V_{i,j}^1$ from $j = 1$ to $j = n_k$. We can start from $j = 1$ and keep increasing j until it happens that $V_{i,j}^1 > V_{i,j+1}^1$. If this happens, definitely the optimal value is $V_{i,j}^1$, because the maximand is strictly concave and $V_{i,j}^1$ will keep decreasing as j increases.

Specifically, step 4 of algorithm 1 is going be replaced by the following:

Algorithm 4 (Exploiting Concavity of the Value Function)

4 Update the value function and obtain V^1. More specifically, do the following steps for each of $i = 1, ..., n_k$.

(a) Set $j = 1$.

(b) Compute $V_{i,j}^1$ using the following:

$$V_{i,j}^1 = u(zF(K_i, 1) + (1 - \delta)K_i - K_j) + \beta V_j^0$$

(c) Compute $V_{i,j+1}^1$ using the following:

$$V_{i,j+1}^1 = u(zF(K_i, 1) + (1 - \delta)K_i - K_{j+1}) + \beta V_{j+1}^0$$

(d) Compare $V_{i,j}^1$ and $V_{i,j+1}^1$. If $V_{i,j+1}^1$ is larger, do $j = j + 1$ and go back to step (b).

(e) Otherwise, $V_{i,j}^1$ is the optimal value. $g_i = j$

5 Local Search

The idea of local search is that the optimal decision rule is going to be continuous. For our current example, we know that the optimal decision rule is going to be a continuous function. Therefore, if we know that $g_i = j$ for some i, and if we have reasonably fine grids, we are sure that g_{i+1} is located in the neighborhood of j. So we might only need to search a small neighborhood of j in search for g_{i+1}.

Even if we don't have the continuity result, we can guess that the optimal decision rule for g_i looks like, by implementing some experiments and see the obtained decision rule. If we are reasonably sure that the optimal decision rule is close to continuous, we can limit our search to a small neighborhood of $g_i = j$ when searching for g_{i+1}. We can make sure that our guess is correct by solving the global optimization (not limited to the neighborhood) after solving the value function using the local search, or checking that the bounds we put are not binding in each iteration.

The algorithm requires a modification of Algorithm 1 as follows:

Algorithm 5 (Local Search)
Step 3.5 below must be added between step3 and 4 in Algorithm 1. Step 4 must be replaced by step 4 below. Finally, step 11 must be added at the end of Algorithm 1.

3.5 Fix s^- and s^+. These characterize the local search in the following way. The meaning is the following. Suppose we have the optimal decision for K_i as g_i, for $i+1$, we will search for the region between $\underline{j} = \max\{1, g_i - s^-\}$ and $\overline{j} = \min\{n_k, g_i + s^+\}$ for $j = g_{i+1}$. It is easy to see that small s^- and s^+ are aggressive, and time-saving but more risky.

4 Update the value function and obtain $V^1 = \{V_i^1\}_1^{n_k}$. More specifically, do the following steps for each of $i = 1, ..., n_k$.

(a) Find the lower and upper bounds of the optimal choice. For $i = 1$, use $\underline{j} = 1$, and $\overline{j} = n_k$. For $i > 1$, use the following formula:

$$\underline{j} = \max\{1, g_{i-1} - s^-\}$$

$$\overline{j} = \min\{n_k, g_{i-1} + s^+\}$$

(b) Compute the value conditional on the choice K_j for $j = \underline{j}, \underline{j}+1, ..., \overline{j}$. Call it $V_{i,j}^1$. It can be computed using the Bellman Equation, as follows:

$$V_{i,j}^1 = u(F(zK_i, 1) + (1-\delta)K_i - K_j) + \beta V_j^0$$

If the consumption $C = zF(K_i, 1) + (1-\delta)K_i - K_j$ turns out to be negative, assign a very large negative number to $V_{i,j}^1$.

(c) Choose j which gives the highest value among $\{V_{i,j}^1\} j = \underline{j}^{n_k}$. Call it V_i^1. Also store the optimal decision j as $g_i = j$.

(d) Check the validity. If $g_i = \overline{j}$ or $g_i = \underline{j}$, most likely the local search is binding. The neighborhood constricted by $\underline{j}$ and $\overline{j}$ is too restrictive. In this case, go to the global search. Re-set $\underline{j} = 1$, and $\overline{j} = n_k$ and solve for V_i^1 and g_i again.

10 After you finish the iteration, make sure that using the local search did not change the solution of the original problem. You can check it by updating the value function without the local search (global search) and make sure that the updated value is still within the tolerance level of the value function that is obtained from the local search.

6 Final Remarks

Discretization method is considerably slower compared with other methods that you will learn, but it's the most robust method. Robustness is one of the most important (but sometimes under-appreciated) property. As long as the computational time allows, I suggest you always use discretization as the starting point. As your model gets more complicated, you can move on to more efficient method, but it's always nice to be able to compare your results from efficient methods with slower but more robust method like discretization to find bugs in your code or the problem in the model easily.

In addition, the class of interesting problems which cannot be solved other than discretization is large. Especially, the models where you don't have the continuity of the value function or the monotonicity of the optimal decision rule, one of the small number of available methods is discretization.

Even though discretization is a very very slow method, there are ways to speed up the process, like we learned, and all of them can be combined. An important lesson is, except for Howard's algorithm, you need to have properties of the solution. It's always a nice thing to know as many properties of the solution as possible before starting computation. The more properties you know about the solution, you can use more speed-up tricks, you can use more sophisticated approximation algorithm to exploit the properties, and it becomes easier to find problems in your results and thus find bugs in your code.

As I discussed in the local search method, it is possible to make a wild guess and use one of the speed-up tricks even if we are not sure about the properties of the value function or the optimal decision rule. In this case, make sure to check the validity of the solution using the most robust method (without any trick).

References

Stokey, Nancy, Robert E. Lucas, and Edward C. Prescott, *Recursive Methods in Economic Dynamics*, Cambridge, MA: Harvard University Press, 1989.