Introduction

Now that we have covered how to solve simple dynamic programming problems by value function iteration, we consider the convergence of the algorithm. We demonstrate two other methods known as policy function iteration, and modified policy function iteration. These methods are also sometimes known as Howard's Improvement for Ronald A. Howard, a Stanford Professor who pioneered their development.

Policy Function Iteration

For infinite horizon dynamic programming problems, it can be shown that value function iteration converges at the rate β, where β is the discount factor. In practice, β is usually close to one which means this algorithm often converges slowly.

In order to examine the value function iteration algorithm, it is helpful to see which functions take the most runtime.

Problem 1 In Ipython, enter

```
%run -p -s cum Value_Function_Iteration.py
```

where Value_Function_Iteration.py is the name of your script that solves the infinite horizon problem by value function iteration. This will list the function calls made by your code, sorted by the time it spends within each function (including time spent in subfunctions).

Run the same command, this time changing the number of grid points N to be 1000.

Run the command once more, this time setting $N = 1000$ and $\beta = .95$.

In problem 1 you should have noticed that runtime was significantly longer to run for larger N or β closer to 1. The profiler gives more detailed information than just the overall runtime, however. The results of problem 1 should look something like the following.

```
%run -p -s cum Value_Function_Iteration.py
         622 function calls in 2.542 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.065    3.065 <string>:1(<module>)
        1    0.001    0.001    3.065    3.065 {execfile}
        1    0.955    0.955    3.064    3.064 Value_Function_Iteration.py:5(<
            module>)
       59    0.000    0.000    1.418    0.024 fromnumeric.py:683(argmax)
       59    1.417    0.024    1.417    0.024 {method 'argmax' of 'numpy.ndarray'
           objects}
       59    0.001    0.000    0.613    0.010 fromnumeric.py:1774(amax)
       59    0.612    0.010    0.612    0.010 {method 'max' of 'numpy.ndarray'
           objects}
```

We notice that the most time was spent in the maximization step. Remember, the value function iteration method maximizes V (and determines the corresponding policy function ψ) at every step. Because we find both a new V and a new ψ at every step, we only apply the new policy function for one iteration. This gives us a crude approximation to the value function that corresponds to the new policy function, resulting in slow convergence of the value function. Instead, we might consider finding the exact value function associated with each new policy function.

This is the idea behind the policy function iteration algorithm. In this way we iterate on the policy functions rather than the value functions. The algorithm for the policy function iteration can be summarized as follows:

1. Set an initial policy rule $W' = \psi_0(W)$ and a tolerance δ.
2. Compute the value function assuming this rule is used forever:

$$V_k(W) = \sum_{t=0}^{\infty} \beta^t u(W - \psi_k(W)) \tag{1}$$

   We will discuss how to compute this below.

3. Determine a new policy ψ_{k+1} so that

$$\psi_{k+1}(W) = \text{argmax}_{W'} \{u(W - W') + \beta V_k(W - W')\} \tag{2}$$

4. If $\delta_k = ||\psi_{k+1} - \psi_k|| < \delta$, stop, otherwise go back to step b with subscript $k + 1$.

In order to compute the value function, V_k corresponding to a given policy ψ_k, we must solve

$$V_k(W) = u(W - W') + \beta V_k(W') \tag{3}$$
$$= u(W - \psi(W)) + \beta V_k(\psi(W)) \tag{4}$$

for V_k.

Once we have discretized W into $W_1, \ldots, W_N$, equation (3) is a linear system which we can rewrite as

$$V_k(W) = u(W - \psi(W)) + \beta Q V_k(W) \tag{5}$$

where if W is a vector of length N, then Q is the $N \times N$ matrix

$$Q_{ij} = \begin{cases} 1 & \text{if} \quad W_j = \psi(W_i) \\ 0 & \text{otherwise} \end{cases}. \tag{6}$$

That is, $Q_{ij} = 1$ if $W_i = \psi(W_j)$ and is zero otherwise. We see that Q has the property that $QW = \psi(W)$ (thinking of W as a vector). Similarly $QV(W) = V(\psi(W))$. Thus we have manipulated 3 so that we are plugging the same variable, W, into both instances of V_k. By doing so, we can now solve for V_k as

$$V_k = (I - \beta Q)^{-1} u(W - W'). \tag{7}$$

Although Q may be large, we can take advantage of the fact that it is sparse, containing only N nonzero entries out of N^2 total entries.

Problem 2 Solve the infinite horizon cake eating problem from the Value Function Iteration lab again, this time using policy function iteration. In order to take advantage of the sparse matrices I and Q, use the following import line

```
from scipy.sparse.linalg import spsolve
```

and the following code to initialize I (outside the loop)

```
I = sp.sparse.identity(N)
```

and Q (inside the loop since it depends on the current policy function)

```
rows = sp.arange(0,N)
columns = psi_ind
data = sp.ones(N)
Q = sp.sparse.coo_matrix((data,(rows,columns)),shape = (N,N))
Q = Q.tocsr()
```

where N is the size of the W grid and psi_ind is the vector of indices of W' for a given W according to the current policy (psi_ind is obtained from the argmax step). Rather than compute $(I - \beta Q)^{-1}$ directly, use scipy's sparse solver

```
V = spsolve(I-beta*Q,u(W-W[psi_ind])).
```

Take $N = 1000$ and $\beta = .95$.

Plot the policy function and compare with your policy function from the Value Function Iteration Lab.

Modified Policy Function Iteration

While policy function iteration converges in fewer iterations, solving the linear system can be slow, especially for problems with a large state space. There is an alternative to this called modified policy function iteration.

In modified policy function iteration, we don't compute the exact value function corresponding to a policy. Instead, at step (2) of the policy iteration algorithm we iterate m times on the value function equation (5) to get an approximation of the new value function. By iterating on (5) we mean evaluate the right side of (5) to get a new value function. Then plug this new function into the right hand side to get yet another value function. Repeat this m times. This is faster than solving for the exact value function for large state spaces. There is no strict rule on the value of m, the number of value function iterations. In practice values such as $m = 10$ or $m = 15$ often work well.

Note that our methods for solving dynamic programs boil down to some combination of two things: iterating on the value function and iterating on the policy function. Modified policy function does a combination of the two, taking advantage of the strengths of both methods. Because modified policy iteration takes only slightly more work to code than value function iteration, it is often preferred in practice. Whether policy or modified policy iteration will perform better may depend on the problem.

Problem 3 Solve the same problem as in problem 2, this time using the modified policy function iteration method with $m = 15$. In this case let convergence be determined in the same way (computing δ_k) in the same way as in the value function iteration problem.

Problem 4 Solve the cake eating problem with each of the three methods: Value Function Iteration, Policy Function Iteration, and Modified Policy Function Iteration. Report how many iterations each method takes. Also determine which methods have the fastest and slowest runtimes. Use $N = 1000$ as the number of grid points for W and $\beta = 0.95$. It is important that you

use the same initial guess in each case in order to make the results comparable. The accuracy of the initial guess greatly effects the number of iterations to convergence. Take your initial guess as $V = 0$ which corresponds to an initial guess of the policy function with indices $[0, 1, 2, \ldots, N-1]$ (meaning $\psi = 0$).

In general we should see that value function iteration takes more iterations than modified policy function iteration which in turn takes more iterations than policy function iteration. It is important to note that this does not directly say anything about runtime. Each iteration of policy iteration may take longer than an iteration of value function iteration.