Lecture Notes 6: Basics of Collocation*

Ivan Rudik

ECON 509: Computational Methods

February 24, 2016

Thus far we have analyzed what dynamic programming problems are, and how a Bellman equation allows us to simplify a problem with many decision variables into a problem with just two. Recall that an arbitrary infinite horizon problem can be represented using a Bellman equation,

$$V(S_t) = \max_{q_t} u(q_t) + \beta V(S_{t+1}(q_t)).$$

Where V is our continuation value functional. With this functional, we can start from any arbitrary initial state vector and simulate the optimal policy paths in deterministic or stochastic settings. We know that this functional problem has a fixed point from previous lectures, but *how* do we arrive at our fixed point, V^*?

1 Collocation

One way to arrive at V^* is by using *collocation methods*. The key assumption for collocation methods is that we can bound the area of the state space that the policymaker will travel to given how the problem is parameterized. For example, even though capital is technically unbounded above, we know that in virtually all cases there are decreasing returns to holding additional capital so in practice there usually exists some capital level $\bar{K}$ such that no matter what our other states may be (within reason), our capital will never exceed that level conditional on current capital being weakly less than $\bar{K}$. Alternatively, we can often map unbounded intervals into a bounded interval, i.e. we can map $[0, \infty)$ into $[0, 1)$ by exploiting logarithms and exponentials.[1]

*These notes are based on those of Derek Lemoine's.

[1]This comes at a cost since exponentials contain substantial curvature and put a greater burden on whatever approximation scheme you use.

Collocation methods aim to approximate the value function, $V(S_t)$ with some linear combination of known, and usually orthogonal, basis functions. One example of a possible class of basis functions is the monomial basis: $x, x^2, x^3, ...$ In any given iteration of our collocation method, we begin with a vector of coefficients on the basis functions (or perhaps just an initial guess if we are in the first iteration of an infinite-horizon problem), and use a linear combination of the basis functions as an approximation to the value function on the right hand side of the Bellman. We then solve the Bellman with the approximated value function in it, at a set of points in our state space, and then recover a set maximized continuation values at these points in our state space conditional on the previous value function approximant we used. Finally, we use these new maximized values to obtain updated coefficients via regression or solving a system of linear equations, and repeat the process until we have "converged". By the contraction mapping theorem this is guaranteed to converge from any arbitrary initial set of coefficients! (conditional on satisfying the assumptions)

This still leaves several questions unanswered:

- *Where* in the state space do we solve the Bellman equation to update our coefficients?
- What basis functions do we use to approximate the value function?

Both of these choices are crucial, but others have done the heavy lifting already. Schemes exist to generate high quality approximations, and to obtain the approximation at low computational cost.

Once we have recovered an adequate approximation to the value function, we have effectively solved the entire problem! We know how the policymaker's expected discounted stream of future payoffs changes as we move through the state space. We can solve the Bellman at some initial state and know that solving the Bellman will yield us optimal policies. Therefore we can simulate anything we want and recover optimal policy functions given many different sets of initial conditions or realizations of random variables.

2 Interpolation

How many points in our state space do we select as points where we maximize the Bellman? We often have continuous states in economics (capital, temperature, oil reserves, etc.), so we must have some way to reduce the uncountable infinity of points in our state space into something more manageable. We do so by selecting a specific finite number of points in our state space and use them to construct a *grid* that spans the domain of our problem. Using our knowledge of how the value function behaves at the limited set of points on our grid, we can interpolate our value function approximate at all points off the grid points, but *within* the domain of our grid. It is very important to remember that our value function approximant is not valid outside the grid's domain

since that would mean extrapolating beyond whatever information we have gained from analyzing our value function on the grid. Most value function approximants explode once you leave the grid's domain.

2.1 Basis Functions

Let V be the value function we wish to approximate with some $\hat{V}$. $\hat{V}$ is constructed as a linear combination of n linearly independent (i.e. orthogonal) basis function,[2]

$$\hat{V}(x) = \sum_{j=1}^{n} c_j \phi_j(x).$$

Each $\phi_j(x)$ is a basis function, and the coefficients c_j determine how they are combined at some point $\bar{x}$ to yield our approximation $\hat{V}(\bar{x})$ to $V(\bar{x})$. The number of basis functions we select, n, is the degree of interpolation. Collocations methods effectively estimate the coefficients c_j. In order to estimate n coefficients, we need at least n facts about the value function, whether they are the value of the value function, or the derivative of the value function. If we have precisely n facts, we are just solving a simple system of linear equations: we have a perfectly identified system. This is what happens we select our number of grid points in the state space to be equal to the number of coefficients. In this case, we can solve the Bellman at the n grid points, recover the maximized values, and then solve a system of equations, *linear in* c_j that equates the value function approximant at the grid points to the recovered maximized values,

$$\Phi \mathbf{c} = \mathbf{y}.$$

Where Φ is the matrix of polynomials, c is a vector of coefficients, and y is a vector of the maximized values of the Bellman. We can recover c by simply left dividing by Φ which yields,

$$\mathbf{c} = \Phi^{-1}\mathbf{y}.$$

If we have more facts, or grid points, than coefficients, then we can just use OLS to solve for the coefficients by minimizing the sum of squared errors. There are many ways to do collocation. We will work through two of them.

[2] These are called projection methods.

2.1.1 Spectral Methods

Spectral methods apply all of our basis functions to the entire domain of our grid. When using spectral methods we virtually always use polynomials. Polynomials are used because of the Stone-Weierstrass Theorem which states (for one dimension),

Theorem 1. *Suppose f is a continuous real-valued function defined on the interval $[a, b]$. For every $\epsilon > 0$, $\exists$ a polynomial $p(x)$ such that for all $x \in [a, b]$ we have $||f(x) - p(x)||_{sup} \leq \epsilon$.*

What does the SW theorem say in words? For any continuous function $f(x)$, we can approximate it arbitrarily well with some polynomial $p(x)$, as long as $f(x)$ is continuous. This means the function can even have kinks. Unfortunately we do not have infinite computational power so solving kinked problems with spectral methods is not advised. Note that the SW theorem *does not* say what kind of polynomial can approximate f arbitrarily well, just that some polynomial exists. This is critical.

The most simple basis is just the monomial basis, $1, x, x^2, ...$ We never actually use this basis because the matrix of polynomials, Φ, is often ill-conditioned, especially as the degree of the polynomials increases. More frequently used is the Chebyshev basis which has better approximation properties. Chebyshev polynomials are often selected because they minimize the oscillations that occur when approximating functions like Runge's function. Indeed, the Chebyshev polynomial closely approximates the *minimax polynomial* which is the polynomial, given some degree d, that minimizes any approximation error to the true function. Chebyshev polynomials are defined by a recurrence relation,

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1} &= 2xT_n(x) - T_{n-1}(x) \end{aligned}$$

and are defined on the domain $[-1, 1]$.[3] Chebyshev polynomials look similar to monomials, but are slightly more complex and are displayed in Figure 1. Compare the Chebyshev basis to the monomial basis in Figure 2. Clearly Chebyshev polynomials do a better job spanning the space than monomials which tend to clump together. Chebyshev polynomials are very nice for approximation because they are *orthogonal*, i.e. they are linearly independent, and they form a basis.[4] This means that you can form any polynomial of degree equal to less than the Chebyshev polynomial you are using. Moreover, it guarantees that Φ has full rank and is invertible.

Thus far we have displayed the Chebyshev basis in only one dimension. We approximate func-

[3] In practice you can expand this to a domain with any bounds you want with a simple 1 to 1 mapping.
[4] They span the polynomial vector space.

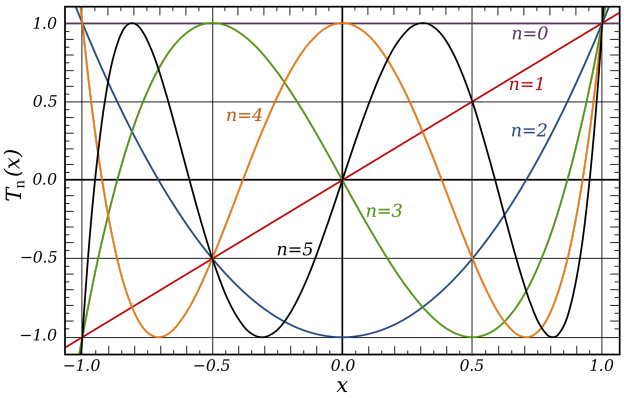

Figure 1: Chebyshev basis.

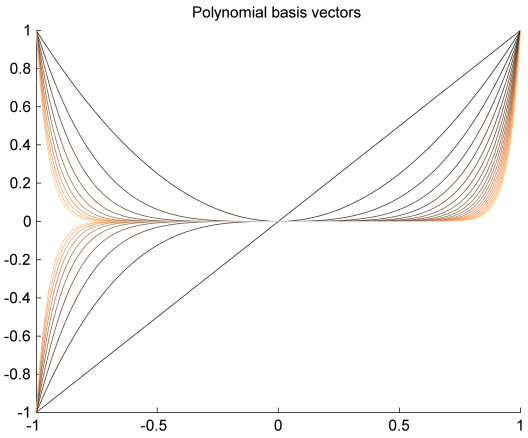

Figure 2: Monomial basis.

tions of some arbitrary dimension N by taking the tensor product of vectors of the one-dimensional Chebyshev polynomials. For example if we wanted to approximate a two dimensional function with a degree 3 polynomial, implying 3 nodes for each dimension, we could do the following. We have a vector of polynomials $[\phi_{1,1}, \phi_{1,2}, \phi_{1,3}]$ for dimensions 1 and $[\phi_{2,1}, \phi_{2,2}, \phi_{2,3}]$ for dimension 2. The tensor product is just the product of every possibly polynomial pair which results in $[\phi_{1,1}\phi_{2,1}, \phi_{1,1}\phi_{2,2}, \phi_{1,1}\phi_{2,3}, \phi_{1,2}\phi_{2,1}, \phi_{1,2}\phi_{2,2}, \phi_{1,2}\phi_{2,3}, \phi_{1,3}\phi_{2,1}, \phi_{1,3}\phi_{2,2}, \phi_{1,3}\phi_{2,3}]$. When then solve for the 9 coefficients on these two dimensional polynomials.

If the value function has large amounts of curvature then higher degree polynomials will be required to get an accurate approximation. If the polynomial can't capture this curvature well, this often manifests as "waviness" in the value function approximant where the underlying Chebyshev polynomial is showing through instead of the value function we are trying to approximate.

2.1.2 Finite Element Methods

Finite element methods use basis functions that are non-zero over subintervals of the domain of our grid. For example, we can use *splines*, which are piecewise polynomials, over segments of our domains where they are spliced together at prespecified breakpoints, which are called knots. The higher the order the polynomial we use, the higher the order of derivatives that we can preserve continuity at the knots. For example, a linear spline yields an approximate that is continuous, but its first derivatives are discontinuous step functions unless the underlying value function happened to be precisely linear. If we have a quadratic spline, we can also preserve the first derivative's continuity at the knots, but the second derivative will be a discontinuous step function. This is because as we increase the order of the spline polynomial, we have increasing numbers of coefficients we need to determine. To determine these additional coefficients using the same number of points, we require additional conditions that must be satisfied. These are what ensure continuity of higher order derivatives at the knots as the degree of the spline grows.

With linear splines, each segment of our value function approximant is defined by a linear function. For each of these linear components, we only need to solve for one coefficient and the intercept term. Each end of the linear segment must be pinned at a certain value we recovered from maximizing the previous spline value function approximant on our grid of spline knots. We have two conditions and two unknowns for each segment. This is a simple set of linear equations that we can solve. In numerical models we typically don't use linear splines because we often care about the quality of approximation of higher order derivatives. Suppose we wish to approximate using a cubic spline on $N + 1$ knots. We need N cubic polynomials when entails $4N$ coefficients to determine. We can obtain $3(N - 1)$ equations by ensuring that the approximant is continuous, and its first and second derivatives are continuous at all interior knots $[3 \times (N + 1 - 1 - 1)]$. This

means that the value of the left cubic polynomial equals the value of the right cubic polynomial at each interior knot. Ensuring the the approximant equals the function's value at all of the nodes adds another $N+1$ equations. We therefore have a total of $4N-2$ equations for $4N$ coefficients. We need two more conditions to solve the problem. What is often used is that the approximant's first or second derivative matches that of the function at the end points.

If the derivative is of interest for optimization, or to recover some variable of economic meaning, then we may need to have these derivatives preserved well at the knots. One large benefit of splines is that they can handle kinks or areas of high curvature by having the modeler place many breakpoints in a concentrated region. If the knots are stacked on top of one another, this actually allows for a kink to be explicitly represented in the value function approximant. However the economist must know precisely where the kink is.

2.2 Interpolation Nodes

We construct the approximant by evaluating the function (or higher order derivatives) on our predefined grid. These grid points are called *interpolation nodes*. These are specific values in the domain of V. If we want an n degree interpolation to match the value of the function, we need at least n nodes. If we have precisely n nodes, x_i, we then have,

$$\sum_{j=1}^{n} c_j \phi_j(x_i) = V(x_i) \ \forall i = 1, 2, ..., n \qquad \text{(interpolation conditions)}$$

We can write this problem more compactly as,

$$\Phi c = y \qquad \text{(interpolation equation)}$$

where y is the column vector of $V(x_i)$, c is the column vector of coefficients c_j, and Φ is an nxn matrix of the n basis functions evaluated at the n nodes. If we solve the Bellman and recover a set of optimized values at our interpolation nodes, $V*(x_i)$, we can then simply invert Φ and right multiply it by y to recover our coefficients for our next iteration.

The next question is, how do we select our set of nodes x_i? An intuitive selection of nodes would be evenly spaced nodes over our domain. If we don't know where there are areas of high curvature where we would like to place many nodes, this seems like a good first guess. However, evenly spaced nodes often do poorly. We can show this by looking at Runge's function,

$$f(x) = \frac{1}{1+25x^2}.$$

If we attempt to approximate it with evenly spaced polynomials we get extreme oscillations near the end points.

```
% Create anonymous function of Runge's function
f = @(x) 1./(1 + 25*x.^2);
x = linspace(-1,1,500);
y_true = f(x);
plot(x,y_true,'r','linewidth',2);
hold on;
% Equally spaced points
N = 10; % Degree of the polynomial we try to fit
xdata = linspace(-1,1,N+1)'; % Need N+1 nodes
ydata = f(xdata); % recover function value at the nodes
p = polyfit(xdata,ydata,N); % fit polynomial
y_fit = polyval(p,x); % evaluate our polynomial
% approximant at the 500 points
plot(x,y_fit,'b','linewidth',2); % plot function
plot(xdata,ydata,'k.','markersize',30); % plot nodes
```

A better selection of nodes are called *Chebyshev nodes.* These are simply the roots of the Chebyshev polynomials above on the domain $[-1, 1]$. They are given by,

$$x_k = cos\left(\frac{2k-1}{2n}\pi\right),\ k = 1, ..., n,$$

for some Chebyshev polynomial of degree n. Mathematically, these also help reduce error in our approximation. We can gain intuition by looking at a graph of where Chebyshev nodes are located.

```
% Create a structure defining the approximation function we will be using
fspace = fundef({'cheb',15,-1,1});
% Recover the nodes
nodes = funnode(fspace);
% Plot the nodes
scatter(nodes,zeros(size(nodes)), '.')
```

This plots Chebyshev nodes corresponding to a degree 15 Chebyshev polynomial approximant. Clearly the nodes are heavily focused near the end points. Why is that? Imagine areas of our approximant near the center of our domain but not at a node. These areas benefit from having multiple nodes on both the left and right. This in a sense provides more information for these off-

node areas and allows them to be better approximated because we know whats happening nearby in several different directions. If we moved to an area closer to the edge of the domain, for example near the exterior node, there is only one node to the left or right of it providing information on what the value of our approximant should be. The approximation won't be as good, all else equal. Therefore, it's best to put more nodes in these areas to shore up this informational deficit and get good approximation quality near the edges of our domain.

3 Collocation in Practice

Now we know that we can approximate any function by solving a simple set of linear equations, regardless if we use spectral methods or finite element methods. How do we do this in practice? We will proceed by going through how to code up a Chebyshev polynomial approximant in MATLAB. First, you will need the CompEcon toolbox. This toolbox has functions ready to go that do all of the heavy lifting in terms of creating the matrix of polynomials, Φ, and finding where are nodes are. Assume we have a well-defined Bellman, and that we have found a bounded area of the state space on which to solve the Bellman.

3.1 Value Function Iteration

The collocation algorithm we will be using, called *value function iteration*, is as follows,

1. Select the number of collocation nodes in each dimension and the domain of the approximation space
2. Select an initial guess for the value function coefficients (typically zeros to start), and for the controls for the optimization routine
3. Select a rule and a corresponding tolerance for convergence (max value function change, average value function change, max coefficient change, etc)
4. While convergence criterion $>$ tolerance
   (a) Solve the right hand side of the Bellman equation using the value function approximant in place of $V(x)$
   (b) Recover the maximized values, conditional on the approximant
   (c) Given these values and the Chebyshev polynomial matrix, solve for a new set of coefficients
   (d) Update our value function approximant using these coefficients
   (e) Store old coefficients or old values to determine convergence
   (f) Use the optimal controls for this iteration as our initial guess for next iteration
5. Error check your approximation.[5]

[5] We will discuss ways to do this later.

Using the CompEcon toolbox, this algorithm looks like,

1. Code a function that is the right hand side of the Bellman.
   (a) Code a function that is the non-maximized part of the right hand side
   (b) Code a container function that does the maximization with KNITRO/fmincon/NLopt/ipopt
2. Define the domain of approximation space
3. Call `fundefn` to define a function space for approximation
4. Call `funnode` to create a cell array of collocation nodes. Use `gridmake` to convert to a matrix.
5. Define an initial guess for the coefficients.
6. While convergence criterion > tolerance
   (a) For (loop) each node in the grid
      i. Maximize the right hand side of the Bellman with your optimization routine of choice
      ii. Return maximized value and optimal controls
   (b) Use `funfitxy` to solve the system of linear equations and obtain the new vector of coefficients

4 Tips

There are many formal ways to speed up the approximation process. Here are several others,

- Solve a coarser problem with fewer nodes and use the final solution as the initial guess for a problem with more nodes. This often decreases the computing time.
- Keep track if, when you maximize the right hand side of the Bellman in the inner loop above, whether you will transition to a state outside the domain. This can cause serious convergence problems because the maximization routine will be evaluating your Bellman outside the domain where the value function approximant does not work.
- Determine which dimensions contain most of the curvature (plot the value function from a coarse solution, look at gradients). Increase the degree of approximation along these dimensions.
- Keep the function being maximized as light as possible. It will be called thousands or millions of times. Each line of code counts. If you use Julia be aware of type stability and global variables.

- Get your domain as tight as possible. This is one of the easiest yet biggest payoff things you can do. This will reduce the degree of approximation you require and can substantially reduce computing time, e.g. Cai et al. (2015) solves a 6-stated problem in minutes with a 4th degree approximation on a tight and time-variant grid while Lemoine and Traeger (2014) takes a day on a wide and time-invariant grid.

- If you have exogenously evolving processes on an infinite horizon you can include time as a state variable by transforming time t to artificial time τ as $\tau(t) = 1 - exp(-z\,t)$ for some $z > 0$. This maps time into $[0, 1)$, and z determines how far the nodes are stretched towards the infinite horizon (but still at some finite time). This way you do not need a state variable for each process.

5 Other Solution Methods

Collocation is one of many ways to solve dynamic problems. Here we describe other ways to compute discrete time dynamic problems. See Fernandez-Villaverde et al. (2016) for a high level description of a few methods, and Judd (1998) for a bit more detail.

For the next two methods we use an example of a stochastic growth model following Judd et al. (2014b),

$$\max_{\{c_t, k_{t+1}\}_{t=0}^{\infty}} E_0 \left[\sum_{t=1}^{\infty} \beta^t u(c_t) \right]$$
$$c_t + k_{t+1} = (1-\delta)k_t + \theta_t f(k_t)$$
$$log(\theta_t) = \rho log(\theta_{t-1}) + \sigma\epsilon_t, \epsilon_t \sim \mathcal{N}(0,1)$$

where both consumption and time $t+1$ capital are positive and initial conditions are given for capital and the stochastic production level. $\rho \in (-1, 1)$, $\beta \in (0, 1)$, and $\sigma > 0$.

5.1 Fixed Point Iteration (GSSA)

Fixed point iteration re-casts equilibrium conditions of the model as a fixed point. We then perform multi-dimensional function iteration to solve for the fixed point. This ends up being very simple and it works on any dimension function. It is also not bear a terrible computational cost and is derivative-free. However it will not always converge. Damping tends to solve this where our true updated guess is a convex combination of our previous guess and what our guess would be without

damping. This keeps the change in our guesses small and limits unrecoverable errors. To employ fixed point iteration, we recover the Euler equation and use primes to indicate next period,

$$u'(c) = \beta E\left[u'(c')\left(1 - \delta + \theta' f'(k')\right)\right]. \tag{1}$$

If we divide both sides by the left hand side of the equation, we have defined,

$$1 = \beta E\left[\frac{u'(c')}{u'(c)}\left(1 - \delta + \theta' f'(k')\right)\right].$$

Therefore if we then multiply both sides by k', we have a fixed point expression,

$$k' = \beta E\left[\frac{u'(c')}{u'(c)}\left(1 - \delta + \theta' f'(k')\right)k'\right]. \tag{2}$$

How do we solve this? We approximate the capital policy function $k_{t+1} = K(k_t, \theta_t)$ with some flexible functional form $\Psi(k_t, \theta_t; b)$ where b is a vector of coefficients. The functional form can be Chebyshev polynomials if you wish, or any other kind. We have defined k_{t+1} in two ways, once as an outcome of the policy function, and once as a conditional expectation of a time $t+1$ random variable in equation 2. Now we can form our capital policy *function* as a fixed point by substituting $K(k_t, \theta_t)$ into the right hand side of equation 2. We perform the algorithm as follows

1. Guess an initial vector of coefficients b_0.
2. Choose an initial state k_0, θ_0
3. Draw a sequence of shocks $\{\epsilon_t\}_{t=0}^{T}$ for a simulation of horizon T and compute $\theta_t \ \forall t = 0, .., T$
4. Simulate the model forward using our approximating policy function, $k' = \Psi(k, \theta; b)$ and the capital/consumption transition equation.
5. Conditional on b, compute k', k'', c (via transition equation and $K(k, \theta, b)$ approximation), and c'' (via transition equation and $K(k', \theta', b)$.
6. Substitute into the right hand side of equation 2 and denote this vector of data y_t.
7. Minimize the errors in the regression by selecting a vector $\hat{b}$, $y_t = \Psi(k_t, \theta_t; b) + \epsilon_t$ according to some norm (e.g. SSE).
8. Compute the next vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}$ where $\gamma \in [0, 1)$.
9. Iterate until convergence.

5.2 Time Iteration

An alternative iteration procedure with the Euler equation is time iteration.[6] In time iteration we solve a 3 equation system, our Euler equation (equation 1), the current transition equation, and a one-step ahead transition equation using an approximation of capital in time $t+2$. The system of equations is,

$$u'(c) = \beta E\left[u'(c')\left(1-\delta+\theta' f'(\hat{k}')\right)\right]$$
$$c = (1-\delta)k + \theta f(k) - \hat{k}'$$
$$c'' = (1-\delta)\hat{k}' + \theta' f(\hat{k}') - K(\hat{k}', \theta'; b)$$

where $\hat{k}' = K(k, \theta; b)$ is our approximated capital policy function. Similar to value function iteration, we are solving for our current capital decision $\hat{k}'$ given our approximating function over our future capital decision $k'' = K(\hat{k}', \theta; b)$. The algorithm works as follows,

1. Create a grid on our capital state.
2. Guess an initial vector of coefficients b.
3. Solve the system of equilibrium and transition equations on our grid by selecting k_{t+1} with a numerical solver.
4. Solve for a new vector $\hat{b}$ by interpolation.
5. Iterate until convergence.

Unfortunately time iteration tends to be slow, especially as the number of dimensions grows. It is the policy-equivalent of value function iteration.

5.3 Perturbation Methods

Perturbation methods approximate solutions by starting from the exact solution, e.g. a deterministic steady state. The conventional way to do this is by using Taylor approximations. Why do we want to use these methods? First, they are extremely accurate locally. If we care about approximating a model near a steady state, we will be able to characterize the steady state well. Second, it's a simple and intuitive way to solve a problem. Third, simple linearization of steady states is just a special case of perturbation methods (a first-order one). Fourth, software has been developed (Dynare) that automates a lot of the process.

[6] This is alternatively called policy function iteration. However to avoid confusion with modified policy iteration it has been renamed time iteration.

6 Error Checking

We have gone through several different ways to compute dynamic economic models. However, we still need to *verify* that our solutions are accurate. This seems like a tall task: we do not actually know for certain what the true solution is, so how can we possibly figure out how close we are? There are several ways and some are better than others. See Judd et al. (2014a) for a nice list of the different approaches, and Fernandez-Villaverde et al. (2016) for a detailed description of two approaches.

6.1 Euler Errors

The most common way to check model accuracy is by determining the error in the Euler equation. We know along any equilibrium trajectory the Euler equation must be satisfied exactly. Any errors in the Euler equation of a simulated set of trajectories must be due to numerical error. How do we compute the Euler error? Let us return to our cake eating example which had an Euler equation,

$$u'(c_t) = \beta u'(c_{t+1}).$$

Notice that if we invert the marginal utility function on the left hand side, divide both sides by c_t, and then subtract the remaining 1, we arrive at an expression of the Euler equation that equals zero analytically, and is expressed in consumption units,

$$EEE = u'(\beta\, u'(c_{t+1}))^{-1}/c_t - 1 = 0.$$

However in practice, this expression will not equal zero due to numerical error. We call the left hand side of the transformed Euler equation the Euler equation error (EEE). Since it is in consumption units it has an economic interpretation has the relative optimization error due to following the approximated policy rule instead of the optimal policy rule. E.g. if EEE $= 10^{-3}$, then we are making a \$1 mistake for every \$1000 spent along our approximated policy trajectory. What is nice about the Euler equation error, is that under certain conditions (Santos, 2000), the error in the policy rule is the same order of magnitude of the Euler error, and the change in welfare is the square of the Euler error.

6.1.1 Bellman Errors

Unfortunately many problems do not permit an Euler equation. In this case we can select a different equilibrium condition: the Bellman equation itself. Computing Bellman errors follows the same procedure as the Euler errors, but to obtain the same interpretation we must find a way to express

the error in consumption units. Note that the Bellman error analysis does not permit the same bounds on policy rule and welfare errors as the Euler error analysis, but it may be the best we can do in some circumstances.

References

Cai, Yongyang, Kenneth L Judd, and Thomas S Lontzek (2015) "The Social Cost of Carbon with Economic and Climate Risks," *arXiv preprint arXiv:1504.06909*.

Fernandez-Villaverde, Jesus, Juan Rubio-Ramirez, and Frank Schorfheide (2016) "Solution and Estimation Methods for DSGE Models," *NBER Working Paper 21862*.

Judd, Kenneth L. (1998) *Numerical Methods in Economics*, Cambridge, MA: MIT Press.

Judd, Kenneth L, Lilia Maliar, and Serguei Maliar (2014a) "Lower Bounds on Approximation Errors: Testing the Hypothesis That a Numerical Solution Is Accurate," *SSRN Electronic Journal*.

Judd, Kenneth L., Lilia Maliar, Serguei Maliar, and Rafael Valero (2014b) "Smolyak method for solving dynamic economic models: Lagrange interpolation, anisotropic grid and adaptive domain," *Journal of Economic Dynamics and Control*, Vol. 44, pp. 92–123.

Lemoine, Derek and Christian Traeger (2014) "Watch Your Step: Optimal policy in a tipping climate," *American Economic Journal: Economic Policy*, Vol. 6, No. 1.

Santos, Manuel (2000) "Accuracy of Numerical Solutions Using the Euler Equations Residual," *Econometrica*, Vol. 68, No. 6, pp. 1377–1402.