Lecture Notes 6: Basics of Projection

Ivan Rudik

ECON 509: Computational Methods

October 10, 2017

So far we have analyzed what dynamic programming problems are, and how a Bellman equation allows us to simplify a problem with many decision variables over many periods into a problem with just one decision and two periods: today and the future. 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. With this, we can start from any arbitrary initial state vector and simulate the optimal policy paths in deterministic or stochastic settings. We know that this problem has a fixed point (solution) from previous lectures, but *how* do we arrive at our fixed point, V^*? Generally this cannot be done analytically.[1] Numerically we have two broad classes of solution methods: projection methods and perturbation methods. In this class we will focus on projection methods.

1 Projection Methods

Projection methods are simple. They build some function $\hat{V}$ indexed by coefficients that, in this specific case, approximately solves the Bellman. What do I mean by approximately? The coefficients of $\hat{V}$ are selected to minimize some residual function that tells us how far away our solution is from solving the Bellman.

Before we continue let us rearrange our Bellman equation and define a new functional H that

[1] In the case of problems with a quadratic payoff and linear transitions we can ensure the value function is quadratic. This lends itself to easy analytic solutions.

will map the problem into a more general framework,

$$H(V) = V(S_t) - \max_{q_t} u(q_t) + \beta\, V(S_{t+1}(q_t)) \tag{1}$$

To solve our Bellman equation we can alternatively find some function V that solves $H(V) = 0$. We solve this by specifying some linear combination of basis functions $\Psi_i(\mathbf{S})$,

$$V^j(\mathbf{S}|\theta) = \sum_{i=0}^{j} \theta_i \Psi_i(\mathbf{S}),$$

with coefficients $\theta_0, ..., \theta_j$. We then define a residual,

$$R(\mathbf{S}|\theta) = H(V^j(\mathbf{S}|\theta)),$$

and select the coefficient values to minimize the residual, given some measure of distance. This step of selecting the coefficients is called *projecting* H against our basis (where it gets its name). We still have some choices to make? What basis do we select? How do we project (select the coefficients)?

Thinking about this may be confusing. For an example of projection methods, consider ordinary least squares linear regression. When performing linear regression we can think of the problem as searching for some unknown conditional expectation $E[Y|X]$, given outcome variable Y and regressors X. We don't know the true functional form, but we can approximate it using the first two monomials on X: 1 and X,[2]

$$E[Y|X] \approx \theta_0 + \theta_1 X.$$

These are the first two elements of the *monomial basis*. One residual function is then,

$$R(Y, X|\theta_0, \theta_1) = abs(Y - \theta_0 - \theta_1 X),$$

but in econometrics we often minimize the square of this residual, which is just OLS. So OLS is within the class of projection methods. In OLS we use observed data, but in theory we use the operator $H(V)$ imposed by theory.

Projection methods are separated into several broad classes by the type of residual we're trying to shrink to zero. More generally, we need to select some metric function ρ, that determines how we project. ρ tells us how close our residual function is to zero over the domain of our state space. For example, Figure 1 shows two different residuals on some capital domain of $[0, \bar{k}]$. The residual based on the coefficient vector θ_1 is large for small values of capital but near-zero everywhere else.

[2]This is equivalently a first-order taylor expansion of the conditional expectation.

Figure 2: Residual Functions

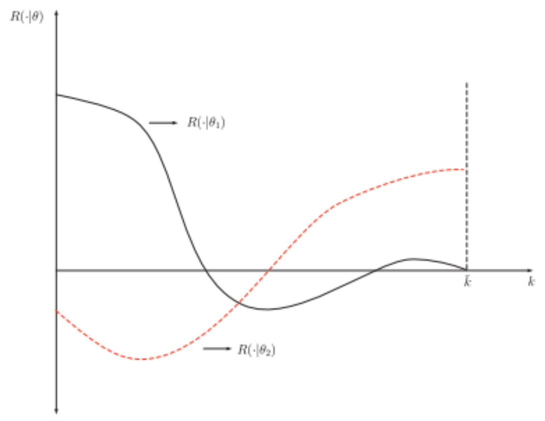

Figure 1: Two different residual functions (Fernandez-Villaverde et al., 2016).

The residual based on θ_2 has medium values just above everywhere. Which is closer to zero over the interval? It will depend on our selection of ρ.

We move from the plain residual to ρ because we want to set a *weighted residual* equal to zero, which gives us a bit more generality. Suppose we have some weight functions $\phi_i : \Omega \to \mathbb{R}$ that map from our state space to the real line, the metric is defined as,[3]

$$\rho(R \cdot |\theta, 0) = \begin{cases} 0 & \text{if } \int_\Omega \phi_i(\mathbf{S}) R(\cdot|\theta) d\mathbf{S} = 0, i = 1, ..., j+1 \\ 1 & \text{otherwise} \end{cases}$$

Where we want to solve for $\theta = \text{argmin}\, \rho(R(\cdot|\theta), 0)$. We can then change our problem to simply solving a system of integrals ensuring the metric is zero,

$$\int_\Omega \phi_i(\mathbf{S}) R(\cdot|\theta) d\mathbf{S} = 0, i = 1, ..., j+1.$$

We can solve this using standard rootfinding techniques. The question is, how do we choose our $j+1$ weight functions? First lets begin with a simple example before moving into the most commonly used weight function.

[3]Here I focus on a one-dimensional case for simplicity.

Least Squares Suppose we selected the weight function to be,

$$\phi_i(\mathbf{S}) = \frac{\partial R(\mathbf{S}|\theta)}{\partial \theta_{i-1}}.$$

Then we would be performing least squares! Why? Recall the objective of least squares is,

$$\min_{\theta} \int R^2(\cdot|\theta) d\mathbf{S}.$$

The first order condition for a minimum is,

$$\int \frac{\partial R(\mathbf{S}|\theta)}{\partial \theta_{i-1}} R(\cdot|\theta) d\mathbf{S} = 0, i = 1, ..., j+1,$$

so the first order condition sets the weighted average residual to zero where the weights are determined by the partial derivatives of the residual. The partial derivative weight function yields a metric function that solves the least squares problem!

2 Collocation

The most commonly used weight function gives us a methodology called *collocation.* Here our weight function is,

$$\phi_i(\mathbf{S}) = \delta(\mathbf{S} - \mathbf{S}_i).$$

Where δ is the Dirac delta function and $\mathbf{S}_i$ are the j+1 *collocation points* or *collocation nodes* selected by the researcher. The Dirac weight function is zero at all $\mathbf{S}$ except at $\mathbf{S} = \mathbf{S}_i$. What does this weight function mean? Before we even select our coefficients, this means that the residual can only be non-zero at a finite set of points $\mathbf{S_i}$. So the solution to our problem must set the residual to zero at these collocation points.[4] Since we have a finite set of points we do not need to solve difficult integrals but only a system of equations,

$$R(\mathbf{S}_i|\theta) = 0, i = 1, ..., j+1.$$

As with all projection methods, 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: $x, x^2, x^3, ...$ But how do we implement collocation? Recall we solve for our coefficients θ by setting the residual to be zero at all of our

[4]We will talk about selecting these points later.

collocation points. But the residual is a function of V, and thus will be a function of θ, a loop seemingly impossible to escape. However, this will not be an issue. We can solve this problem by *iterating* on the problem, and continually setting the residuals equal to zero, recovering new θs, and repeating. 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 our set of collocation points, 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 solving the 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 and there being no numerical error..)

This still leaves several questions unanswered:

- *Where* in the state space do we place our collocation nodes?
- 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.

3 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, technology shocks, 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 *collocation 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.

3.1 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,

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

Each $\psi_j(x)$ 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. In order to recover n coefficients, we need at least n equations that must be satisfied at a solution to the problem. If we have precisely n equations, we are just solving a simple system of linear equations: we have a perfectly identified system and are solving a collocation problem. This is what happens we select our number of grid points in the state space to be equal to the number of coefficients (which induces a Dirac delta weighting function). 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,

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

Where Ψ 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 Ψ which yields,

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

If we have more equations, 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 projection but we will work with collocation.

3.1.1 (Pseudo-)Spectral Methods

Spectral methods apply all of our basis functions to the entire domain of our grid: they are global. 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, $1, x, x^2, ...$ Even if this basis is not orthogonal, SW tells us that we can uniformly approximate any continuous function on a closed interval using them.

We never actually use this basis. Why? First because the matrix of polynomials, Φ, is often ill-conditioned, especially as the degree of the polynomials increases.[5] Second, they can vary dramatically in size, which leads to scaling issues of adding/substracting large and small numbers and getting truncation errors. For example, $x^1 1$ goes from $1e - 4$ to about 90 when moving x from 0.5 to 1.5.

Ideally we want an orthogonal basis: when we add another element of the basis, it has sufficiently different behavior than the elements before it so it can capture features of the unknown function that the previous elements couldn't. Most frequently used is the Chebyshev basis which has nice approximation properties: they are easy to compute, they are orthogonal, and they are bounded between $[-1, 1]$. 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,

$$T_0(x) = 1$$
$$T_1(x) = x$$
$$T_{n+1} = 2xT_n(x) - T_{n-1}(x)$$

and are defined on the domain $[-1, 1]$.[6] Chebyshev polynomials look similar to monomials, but are actually linear combinations of monomials and are displayed in Figure 2. Compare the Chebyshev basis to the monomial basis in Figure 3. Clearly Chebyshev polynomials do a better job spanning

[5]Indeed the first 6 monomials can induce a condition number of 10^{10}, a substantial loss of precision.

[6]In practice you can expand this to a domain with any bounds you want with a simple 1 to 1 mapping.

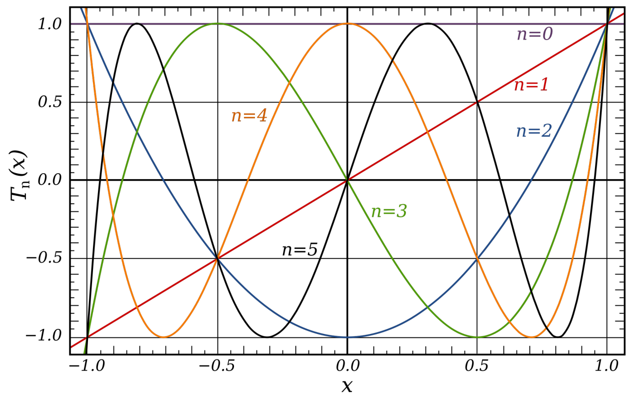

Figure 2: Chebyshev basis.

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 with respect to the weight function $\phi(x) = 1/\sqrt{1-x^2}$.[7] 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. Also note that the Chebyshev polynomial of order n has n zeros given by,

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

which tend to cluster quadratically towards the edges of the domain.[8] This will be important later. An alternative representation of Chebyshev polynomials is,

$$T_n(x) = cos(narccos(x)).$$

They are trigonometric functions disguised as polynomials.

There are two important theorems to know about Chebyshev polynomials.

Theorem 2. *Chebyshev interpolation theorem: If $f(x) \in \mathbb{C}[a,b]$, if $\{\psi_i(x), i = 0, ...\}$ is a system of polynomials (where $\psi_i(x)$ is of exact degree i) orthogonal with respect to $\phi(x)$ on $[a,b]$ and if $p_j = \sum_{i=0}^{j} \theta_i \psi_i(x)$ interpolates $f(x)$ in the zeros of $\psi_{n+1}(x)$, then:*

$$\lim_{j\to\infty} (||f-p_j||_2)^2 = \lim_{n\to\infty} \int_a^b \phi(x)\left(f(x)-p_j\right)^2 dx = 0.$$

[7] They span the polynomial vector space.

[8] You can think about this as projecting sequentially finer uniform grids onto a hemicircle.

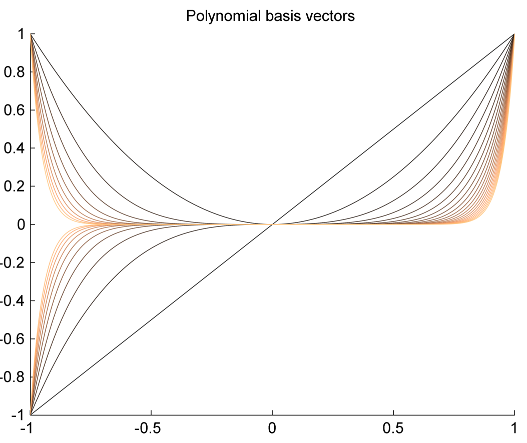

Figure 3: Monomial basis.

This says that if we have an approximation set of basis functions that are exact at the roots of the n^{th} order polynomials, then as n goes to infinity the approximation error becomes arbitrarily small and converges at a quadratic rate. This holds for any type of polynomial but if they are Chebyshev then convergence is uniform. Unfortunately we cant store an infinite number of polynomials in our computer. We would like to know how big our error is after truncating our sequence of polynomials.

Theorem 3. *Chebyshev truncation theorem: The error in approximating f is bounded by the sum of all the neglected coefficients.*

Since Chebyshev polynomials satisfy SW an infinite sequence of them can in theory perfectly approximate any continuous function. And since Chebyshev polynomials are bound between $[-1, 1]$, the sum of the omitted terms is bound by the sum of the magnitude of the coefficients, so the error in the approximation is as well. We often also have that Chebyshev approximations geometrically converge which give us the following convenient property:

$$|f(x) - f^j(x|theta)| \sim O(\theta_j).$$

The truncation error by stopping at polynomial j is of the same order as the magnitude of the coefficient θ_j on the last polynomial. Thus in many situations we can simply check the size of the last polynomial to gauge how accurate our approximation is.

Boyd's Moral Principle Chebyshev polynomials are the most widely used basis. This is not purely theoretical but also from practical experience. John Boyd summarizes decades of experience with function approximation with his moral principle:

1. When in doubt, use Chebyshev polynomials unless the solution is spatially periodic, in which case an ordinary fourier series is better
2. Unless you are sure another set of basis functions is better, use Chebyshev polynomials
3. Unless you are really, really sure another set of basis functions is better use Chebyshev polynomials.

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

3.2 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.

```
# Plot the function
using PyPlot
using Polynomials

# Number of collocation nodes
N = 10

# Runge's function
f(x) = 1./(1.+25.*x.^2)

# Generate evenly spaced x values
x = linspace(-1,1,500)

# Recover true y-values
y_true = f(x)

# Plot
plot(x, y_true, color="red", linewidth=2.0)

# Generated evenly spaced grid
xdata = linspace(-1,1,N+1)

# Get y-values of the true function on the grid
ydata = f(xdata)

# Fit a polynomial to the values on the grid
```

```
p1 = polyfit(xdata,ydata)

# Plot
plot(x, polyval(p1,x), color="blue", linewidth=2.0)
plot(x, y_true, color="red", linewidth=2.0)
scatter(xdata,ydata)
title("Runge's function (red), collocation nodes (gray points), function approximar

# Compute mean and maximum relative error
mean_error = mean(abs.((polyval(p1,x) - y_true)./y_true))
max_error = maximum(abs.((polyval(p1,x) - y_true)./y_true))

# Plot relative errors
plot(x, abs.((polyval(p1,x) - y_true)./y_true), color="blue", linewidth=2.0)

println("The mean relative error in our approximation on an evenly spaced grid of 5
```

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.

```
# Function for generating Chebyshev nodes
cheb_nodes(n) = [cos((2k-1)/2n*pi) for k = 1:n]

# Generate Chebyshev nodes
xdata = cheb_nodes(N)

# Get true function value at Chebyshev nodes
ydata = f(xdata)

# Fit polynomial
p2 = polyfit(xdata,ydata)
```

```
# Plot
plot(x, polyval(p2,x), color="blue", linewidth=2.0)
plot(x, y_true, color="red", linewidth=2.0)
scatter(xdata,ydata)
title("Runge's function (red), collocation nodes (gray points), function approximan

# Compute mean and maximum relative error
mean_error = mean(abs.((polyval(p2,x) - y_true)./y_true))
max_error = maximum(abs.((polyval(p2,x) - y_true)./y_true))

# Plot relative errors
plot(x, abs.((polyval(p2,x) - y_true)./y_true), color="blue", linewidth=2.0)

println("The mean relative error in our approximation on an evenly spaced grid of 5
```

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.

3.3 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.

4 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 Julia. 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.

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

We use an example of a stochastic growth model following Judd (1998),

$$\max_{\{c_t\}_{t=0}^{\infty}} \sum_{t=1}^{\infty} \beta^t u(c_t)$$
$$\text{subject to:} \quad k_{t+1} = f(k_t) - c_t$$

where both consumption and time $t+1$ capital are positive, an initial condition is given for capital and $\alpha > 0$, $\beta \in (0,1)$.

4.1 Value Function Iteration

The first collocation algorithm we will be using, called *value function iteration.* With value function iteration we represent the growth model as a Bellman equation

$$V(k_t) = \max_{c_t} u(c_t) + \beta V(k_{t+1})$$
$$\text{subject to:} \quad k_{t+1} = f(k_t) - c_t$$

How do we solve this? We approximate the value function with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients. The Bellman equation will satisfy the conditions for a contraction mapping so continually iterating on the Bellman will result in us converging towards the true value function. The algorithm is as follows,

1. Select the number of collocation nodes in each dimension and the domain of the approximation space
2. Guess an initial vector of coefficients b_0 with the same number of elements as the collocation grid. Select coefficient values so that implied consumption policies are non-zero, but not so large that next period capital is negative. Select initial guesses for consumption if necessary for the solver.
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 $\Gamma(k_{t+1}; b_0)$ in place of $V(k_{t+1})$
   (b) Recover the maximized values, conditional on the approximant
   (c) Fit the polynomial to the values and recover a new vector of coefficients $\hat{b}_1$.
   (d) Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b_{p+1} = (1-\gamma)b_p + \gamma\hat{b}_{p+1}$ where $\gamma \in (0,1)$.
   (e) Use the optimal controls for this iteration as our initial guess for next iteration
5. Error check your approximation.[9]

As an example, 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 NLopt/ipopt/etc
2. Define the domain of approximation space, which is just a hypercube, as two vectors. One vector is for the upper bound of the interval for each dimension, and one for the lower bound
3. Call `fundefn` to define a function space for approximation
4. Call `funnode` to create a cell array of collocation nodes
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, evaluate the continuation value inside the Bellman using `funeval`
      ii. Return maximized value and optimal controls
   (b) Use `funfitxy` to solve the system of linear equations and obtain the new vector of coefficients

4.2 Fixed Point Iteration

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 does not bear a terrible computational cost and is derivative-free. However it will not always converge and is generally unstable. 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,

$$u'(c_t) = \beta u'(c_{t+1})f'(k_{t+1}). \tag{2}$$

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

If we invert the marginal utility function on the left hand side we have consumption expressed as a fixed point,

$$c_t = u'^{(-1)}\left(\beta u'(c_{t+1})f'(k_{t+1})\right). \tag{3}$$

How do we solve this? We approximate the consumption policy function $c_t = C(k_t)$ with some flexible functional form $\Psi(k_t; b)$ where b is a vector of coefficients. The functional form can be Chebyshev polynomials if you wish, or any other kind. We have defined c_t in two ways, once as an outcome of the policy function, and once as an equilibrium condition in equation (3). Now we can form our consumption policy *function* as a fixed point by substituting $C(k_t)$ into the right hand side of equation 3 as follows

$$C(k_t) = u'^{(-1)}\left(\beta u'(C(k_{t+1}))f'(k_{t+1}(C(k_t), k_t))\right) \tag{4}$$

We perform the algorithm as follows

1. Build a collocation grid.
2. Guess an initial vector of coefficients b_0 with the same number of elements as the collocation grid. Select coefficient values so that implied consumption policies are non-zero, but not so large that next period capital is negative.
3. Substitute $C(k_{t+1}; b_0)$ into the right hand side of equation 3 and recover values of $C(k_t; b_0)$. Repeat until convergence.
4. Fit the polynomial to the values and recover a new vector of coefficients $\hat{b}_1$.
5. Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b_{p+1} = (1-\gamma)b_p + \gamma\hat{b}_{p+1}$ where $\gamma \in (0,1)$.
6. Iterate 3–5 until convergence.

4.3 Time Iteration

An alternative iteration procedure with the Euler equation is time iteration. In time iteration we solve the same system as with fixed point iteration, however we do not iterate on the Euler equation as a fixed point. Instead we solve it using root-finding techniques on our n node collocation grid where we search for the $C_{i+1}(k_t)$ that solves,

$$u'(C_{i+1}(k_t^j)) = \beta u'(C_i(f(k_t^j) - C_{i+1}(k_t^j)))f'(f(k_t^i) - C_{i+1}(k_t^j)) \quad \text{for j } = 1, ..., n \tag{5}$$

$C_i(\cdot)$ is our current approximation to the policy function, and we are searching for a scalar $C_{i+1}(k_t^j)$, given our collocation node k_t^j, that solves the Euler equation root-finding problem. In the Euler equation C_{i+1} corresponds to today's policy function while C_i corresponds to tomorrow's policy function. We are searching for today's policy that satisfies the Euler equation. As we iterate and i increases, $C_i(k)$ should converge because of a *monotonicity property.* If $C_i'(k) > 0$, and $C_i(k) < C_{i-1}(k)$, then $C_{i+1}(k) < C_i(k)$ and $C_{i+1}'(k) > 0$. It preserves the shape of the policy function so it is reliable and convergent.

The algorithm works as follows,

1. Build a collocation grid.
2. Guess an initial vector of coefficients b_0 with the same number of elements as the collocation grid. Select coefficient values so that implied consumption policies are non-zero, but not so large that next period capital is negative.
3. Substitute $C(k_{t+1}^j; b_0)$ into both sides of equation 3 and search for $C(k_{t+1}^j; b_1) \in \mathbb{R}$ that solves the Euler equation on all n collocation nodes.
4. Fit the polynomial to the values and recover a new vector of coefficients $\hat{b}_1$.
5. Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b_{p+1} = (1-\gamma)b_p + \gamma\hat{b}_{p+1}$ where $\gamma \in (0, 1)$.
6. Iterate 3–5 until convergence.

Unfortunately time iteration tends to be slow, especially as the number of dimensions grows.

4.4 Stability in General

shape preservation etc, initial guesses

4.5 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.

5 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.

We will now move into one practical matter regarding solving a Bellman using collocation methods: domain selection. The key assumption for collocation methods (and other projection methods) is that we can bound the area of the state space that the policymaker will travel to given how the problem is parameterized: we need to select a domain for the problem. 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.[10]

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. (2014) 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 terms,

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

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 as 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

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

policy rule is the same order of magnitude of the Euler error, and the change in welfare is the square of the Euler error. The required assumptions are,

1. Compactness and convexity of the feasible set
2. Smoothness and strong concavity of the payoff function
3. Interior solution

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.

6.2 Finding a Lower Bound on Errors

The assumptions of Santos (2000) are not always satisfied, but we still need to calculate errors in some way. Judd et al. (2014) provide a way to put a lower bound on errors. In other words, there's a way for us to test the null hypothesis that our model is accurate in terms of the model's variables instead of errors or residuals in equilibrium conditions. This approach is very general and does not depend on the solution method used to compute the model.

Consider a system of n equations and n unknowns,

$$G_i(x_1, ..., x_n) = 0, i = 1, ..., n \tag{6}$$

Let x^* be the exact solution vector and $\hat{x}$ be the approximate solution vector to equation (6). We can define an approximation error as a compensation $\delta^* \in \mathbb{R}^n$ that is required to make the approximate solution satisfy the system of equations exactly,

$$G(\hat{x}(\mathbf{1} + \delta^*)) = 0,$$

where G is the system of equations in vector notation and $\mathbf{1}$ is a vector of ones.[11] Finding δ^* is difficult, but we can find a lower bound on its size. First, remove $n-m$ equations where $1 \geq m < n$,

[11] $\hat{x}$ is not a function of $(\mathbf{1} + \delta)$, it is multiplication.

typically the most complex to solve equations. Suppose these are just the first m equations,

$$g_i(x_1, ..., x_n) = 0, i = 1, ..., m \tag{7}$$

which is underdetermined and has multiple solutions. A consequence of this is that there are multiple δs that allow some approximation $\hat{x}$ to satisfy equation (7), we can denote this set of δs as $\Omega \equiv \{\delta \in \mathbb{R}^n : g(\hat{x}(\mathbf{1} + \delta)) = 0\}$. We then define,

$$\hat{\delta} \equiv \text{argmin}_{||\delta|| \in \Omega} \text{ subject to } g(\hat{x}(\mathbf{1} + \delta)) = 0$$

. Then (Judd et al., 2014) show that for a given $\hat{x}$ and some given norm $||\cdot||$, $||\hat{\delta}|| \leq ||\delta^*||$ where δ^* is the compensation of the full system in equation (6) and $\hat{\delta}$ is the compensation of the reduced system in equation (7). In other words, the error we calculate for the reduced system is a lower bound on the true error.

So which norm should we choose? The L_2 norm is convenient since we can use first-order conditions of the minimization problem (now over $\delta^T\delta$) to solve for $\hat{\delta}$ which yields,

$$2\hat{\delta} + \lambda \nabla g\left(\hat{x}\left(\mathbf{1} + \hat{\delta}\right)\right)\hat{x} = 0.$$

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 (2014) "Lower Bounds on Approximation Errors: Testing the Hypothesis That a Numerical Solution Is Accurate," *SSRN Electronic Journal.*

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.