Numerical Methods
Lecture 4: Projection Methods

Zachary R. Stangebye

University of Notre Dame

Fall 2017

Constrained Max

- (Most) projection methods will rely heavily on large-scale optimization algorithms
- A brief discussion is in order
  1. Which algorithms are out there?
  2. What do they do?
  3. How do they work?
  4. When are they appropriate?
- Cover 4 different ones
  1. L-BFGS (local max)
  2. Pattern Search (local max, non-gradient-based)
  3. Genetic Algorithm (global max)
  4. Simulated Annealing (global max)
- Useful for other reasons e.g. moment-matching, estimation

Simple Example

- Consider following large-scale problem

$$\max_{\{c_i\}_{i=1}^N} \sum_{i=1}^{N} \beta_i \log(c_i)$$

$$\text{subject to } \sum_{i=1}^{N} p_i c_i \leq I$$

- Want to solve this model for large N e.g. 20 or 40
- Suppose that $\beta_i = i$ and $p_i = e^{\beta_i/4}$ i.e. higher priced goods deliver more utility
  - But not in a way that can be scaled by hand

Simple Example: Dual

- Dual problem features non-linear constraints
- Suppose utility maximizing bundle $\implies \bar{u}$

$$\min_{\{c_i\}_{i=1}^N} \sum_{i=1}^N p_i c_i$$

$$\text{subject to} \quad \sum_{i=1}^N \beta_i \log(c_i) \geq \bar{u}$$

Approach 1: L-BFGS

- 'fmincon' in Matlab
- Variant of Multidimensional Newton's Method
- Recall in one-dimension

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

- In higher dimensions

$$\mathbf{x}_{n+1} = \mathbf{x}_n - H_f^{-1}(\mathbf{x}_n) \triangledown_f(\mathbf{x}_n)$$

- Very fast, but three big issues
  1. Local maxima (mimina) only
  2. Pure Newton-Raphson does not handle constraints well
  3. Numerical Hessians notoriously inaccurate

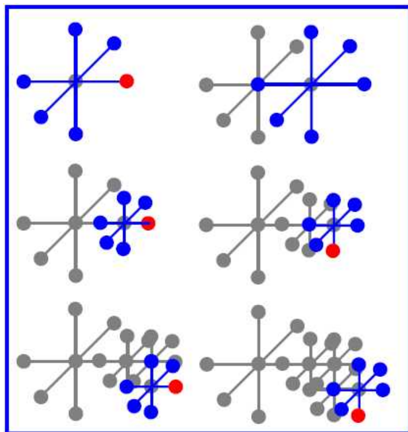
Approach 1:L-BFGS

- Issue (1) cannot be fixed...
- Issue (2) handled in a 'traditional' way ('interior-point' method)
- Issue (3) is the cool part!
  - Algorithm carries around an approximation to the Hessian
  - Necessarily symmetric
  - Uses gradient to update it with the gradient every iteration
  - By the time it converges, get a pretty good estimate!
- Hessian is useful for *lots* of reasons
  - Really cool to have a robust estimate of it!
  - Even if you use a different maximizer, often helpful to feed that answer into this algorithm if only to get the Hessian

Approach 2: Pattern Search

- Non-gradient-based method for local optimization
- Builds ‘mesh’ of new points around current point
- Takes a ‘poll’ of those points to find the best
- Shift to best; increase or decrease the mesh as we progress
- Very fast; handles constraints pretty well

Approach 2: Pattern Search



Approach 2: Pattern Search

- Rough procedure. Set $MeshSize = 1$
  1. Build mesh around current point, x_i

     $$x_{i+1,poll_j} = x_i + (-)MeshSize \times e_j$$

     For $j = 1, \ldots, N$ i.e. positive and negative in all directions
  2. If best poll point beats x_i, set that to x_{i+1} and double *MeshSize* for next iteration
  3. If no poll point beats x_i, halve *MeshSize* and try again
  4. Stop when *MeshSize* gets small e.g. $1e-6$

Approach 3: Genetic Algorithm

- What if we know there are many local maxima?
- Gradient methods will lead us astray
- Genetic algorithm viable alternative
- Designed to imitate evolutionary behavior
  1. Create a random initial ‘population’ i.e. collection of points
  2. Randomly create a sequence of new populations using current ones as ‘parents’
  3. Best populations ‘survive’ iterations
  4. Stops when average change in generation ‘fitness’ is small
  5. Best point from last generation is solution
- Might also stop after certain number of generations/time (controllable options)

Approach 3: Genetic Algorithm

- Important part: How to produce 'children'
- Three ways
  1. 'Elite' parents (best value) automatically survive as children in the next generation (*elite* child)
  2. Two random parents 'mate' i.e. some vector elements chosen from parent 1 and others from parent 2 (*crossover* child)
  3. Introduce small, random changes to a random parent (*mutation* child)
- Very slow, inaccurate if solution time is restricted (as is default in Matlab)
- But if you have time to let it run for a long time...
  - One of most reliable ways to find difficult global max
  - Handles all kinds of constraints really well

Approach 4: Simulated Annealing

- Alternative global search
- Meant to mimic physical process of heating a material and then slowly lowering the temperate to decrease defects
- Procedure: Starting from an initial point
  1. Randomly generate a nearby new point in a radius that shrinks with iterations ('temperature')
     - If new point is better, make it the current point
     - If new point is worse, make it the new point with probability =

$$\frac{1}{1+\exp\left(\frac{f(x_n)-f(x_{n+1})}{T}\right)}$$

     where T is the current 'temperature' (shrinks over time)
  2. Systematically lower the temperate with each iteration
  3. Check gradient: If sufficiently small, 'reheat' (not all the way) the system to get it searching more broadly again

Approach 4: Simulated Annealing

- Benefits
  - Largely (but not totally) gradient-free
  - If temperature cools sufficiently slowly, convergence to global max is guaranteed!
  - Errors tend to be smaller than genetic algorithm
- Cons
  - Really slow
  - Cannot handle constraints well (only 'box' constraints)

Example: Simulated Annealing

- Since simulated annealing cannot handle general constraints, we consider a different problem
  - A firm sells its good at a normalized price $p = 1$
  - It uses N intermediate inputs and has a production function $f(\mathbf{x}) = \prod_{i=1}^{N} x_i^{\alpha_i}$, where $\sum_{i=1}^{N} \alpha_i = 1$
  - The prices of intermediate goods are as in demand problem

$$\max_{\{x_i\}_{i=1}^N} \prod_{i=1}^{N} x_i^{\alpha_i} - \sum_{i=1}^{N} p_i x_i$$

- Notice this problem is unconstrained; we could put large 'box' bounds on it if we wanted

Aside

- Optimization routines often very similar to root-finding routines
    - Often the former is the latter on a FOC
- Computation time similar across two for related algorithms
- Big difference: Local vs. global distinction
    - Matters a lot for optimization methods
    - Irrelevant for rootfinding
        - May be many roots, but generally each is equally useful

Rootfinding

- Solve n-dimensional, nonlinear system of equations $F(\mathbf{x}) = \mathbf{0}$
  - In Matlab, use *fsolve*
- Under the hood, though, Matlab uses a variation of least-squares minimization to solve this problem i.e.

$$\min_{\mathbf{x}} [F(\mathbf{x})]' W [F(\mathbf{x})] = 0$$

  for some positive definite $n \times n$ weighting matrix, W
- Typically employ Newton-Raphson or similar technique
- Often pretty fast, but does not handle constraints well

Rootfinding: Example

- The FOC of our producer's problem yields (for $i = 1, \ldots, N$)

$$\alpha_i x_i^{\alpha_i - 1} \prod_{j \neq i} x_j^{\alpha_j} - p_i = 0$$

Rootfinding: Julia

- Two good ways
  1. Use 'NLsolve' package
     - $nlsolve(f!, initialGuess)$, where $f : \mathcal{R}^N \rightarrow \mathcal{R}^N$
     - Comparable to Matlab's fsolve
  2. Use 'Optim' package (minimization)
     - Define objective to be

$$\min_x \sum_{i=1}^{N} f_i(x)^2$$

     - A zero of f will be a local minimum of this problem
     - Has options for BFGS, PatternSearch (NelderMead), and Simulated Annealing
- Work pretty well (much faster than Matlab, though perhaps not as robust)

Projecting Functions

- Same goal: Approximate the function $d(\mathbf{x})$ such that

$$\mathcal{H}(d) = \mathbf{0}$$

- Basic idea: Build up approximation to $d(\mathbf{x})$ with linear combination of known functions

$$d^j(\mathbf{x}|\theta) = \sum_{i=0}^{j} \theta_i \Psi_i(\mathbf{x})$$

- The collection of functions $\{\Psi_i(\mathbf{x})\}_{i=0}^{\infty}$ are called *basis functions*
    - j is degree of approximation (accuracy)
- All we need to do is find the right vector $\theta \in \mathcal{R}^{j+1}$

Projecting Functions

- Want to find the best θ by minimizing errors (residuals)
- Residual function very intuitive

$$R(\mathbf{x}|\theta) = \mathcal{H}(d^j(\mathbf{x}|\theta))$$

- Want to minimize some function of the residuals

$$\min_{\theta} \ \rho(R(\mathbf{x}|\theta), \mathbf{0})$$

  where $\rho(\cdot,\cdot)$ is some distance metric
- Three big degrees of freedom
  1. Precision/degree (j)
  2. Choice of basis functions
  3. Choice of metric

Relation to Econometrics

- 'Projecting' in this fashion smacks of OLS econometrics
- This is no coincidence
- Old-school regression can be interpreted as this
  - Want to approximate unknown function $E[Y|X]$
  - Use monomials as basis function $(1, x, x^2, \ldots)$
  - Use Euclidean distance metric

Basis Functions: One-Dimensional

- Important to get basis functions right
- Start with *Spectral Bases*
  - Smooth over entire region
  - Tightly parameterized
- Unidimensional Basis Functions
  1. Monomials
  2. Spline
  3. Jacobi Polynomials
  4. Chebyshev Polynomials
- Notation: One vs. Multi-dimensional
  - $\psi(x) : \mathcal{R} \rightarrow \mathcal{R}$
  - $\Psi(\mathbf{x}) : \mathcal{R}^n \rightarrow \mathcal{R}$

Monomials

- Most intuitive set of basis functions

$$\psi_i(x) = x^i$$

$$\implies \mathcal{B} = \{1, x, x^2, x^3, \dots\}$$

- And they work!

Theorem (Stone-Weierstrass Corollary)

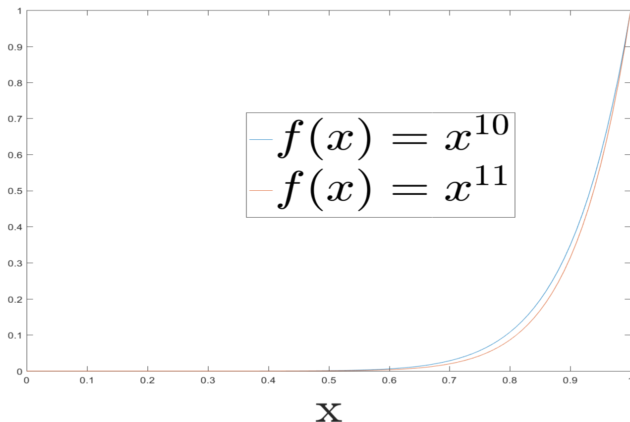
If $f(x)$ is a real-valued, continuous function on the compact set K, then for any $\epsilon > 0$, $\exists h(x) \in span(\mathcal{B})$ such that

$$\sup_{x \in K} |f(x) - h(x)| < \epsilon$$

Monomials

- We could use some linear combination of monomials to get arbitrarily close to any continuous function over any compact set
- BUT...not always very efficient
- Two big problems
  1. Nearly multicollinear in function space, especially for high j
  2. Considerable variation in size
     - e.g. $\psi_{11}(.5) = 4.8e - 4$, $\psi_{11}(1.5) = 86.5$
     - Scaling problems
     - Accumulation of numerical errors

Monomial Problems



Splines

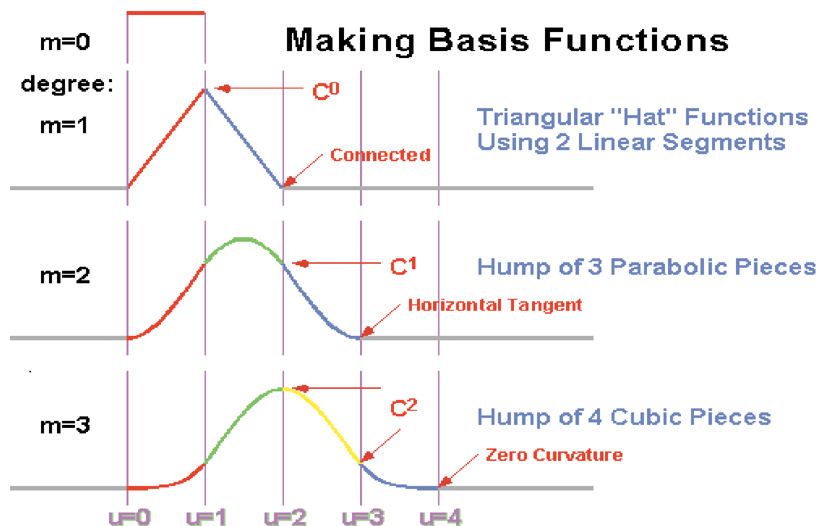
- Already thought of splines as an interpolation method
- Another sort can be used as basis functions: *B-splines*
  - Piecewise polynomials of order p
  - Can be defined recursively
  - Define $n+1$ equidistant *knots* over domain: $\{u_0, u_1, \ldots, u_n\}$

$$N_{i,0}(u) = \begin{cases} 1, & u_i \leq u < u_{i+1} \\ 0, & o/w \end{cases}$$

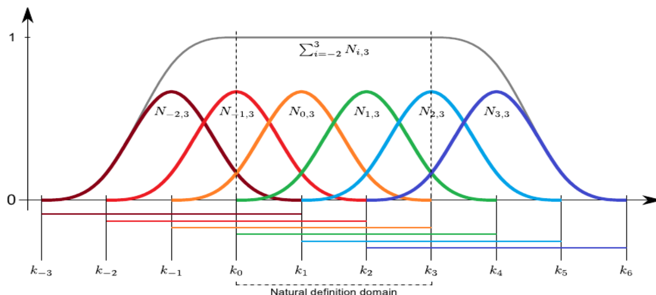
$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

- Basic idea: 'Hills' between each of the knots

B-Splines



B-Splines



- Pieces much more distinct than monomials
- Relation to finite-elements (later)
- Residual choice function not always so natural...

Resolving Multicollinearity

- Monomials seemed to lack a certain 'spanning' property
- How to define it properly? Use a 'dot-product' type definition
  - Two vectors, v_1, v_2 are orthogonal iff their dot-product is zero i.e. $v_1' v_2 = 0$

Definition

A set of basis function $\{\psi_i(x)\}_{i=0}^{\infty}$ will be **orthogonal** on $[a, b]$ iff there exists a nonnegative weighting function, $w(x)$, such that for any $m \neq n$,

$$\int_a^b w(x)\psi_n(x)\psi_m(x) = 0$$

- Different basis functions could be orthogonal for different weights

Orthogonal Polynomials Class I: Gauss-Legendre

- In simplest case, $w(x) = 1$ on $[-1, 1]$
- We get back the Gauss-Legendre Polynomials
- We saw (used) these in integration!
- Relatively simple to use, but not always most efficient
- Since we're really interested in efficiency, we'll move on

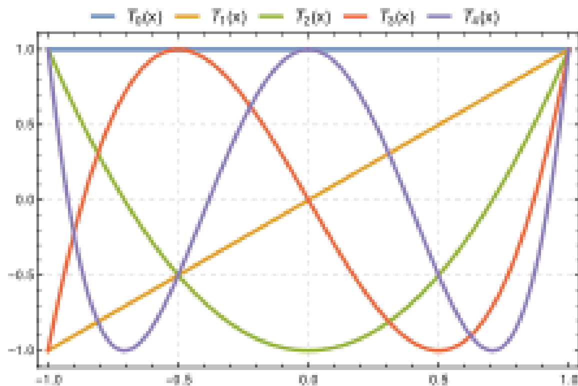
Orthogonal Polynomials Class II: Jacobi

- Weighting function: $w(x) = (1-x)^{\alpha}(1+x)^{\beta}$
- Parameterized by (α, β)
- General recursive forms for any given (α, β)
- Focus on most convenient/efficient form: Chebyshev $\iff \alpha = \beta = -1/2$

$$T_0(x) = 1, \; T_1(x) = x$$

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

Chebyshev Polynomials



Chebyshev Polynomials

- Many nice properties
- First: If we fit exactly the zeros (and nothing else), the approximation error becomes arbitrarily small
- Motivates ideal choice of residual function (later)

Theorem (Chebyshev Interpolation)

If $d(x) \in C[-1,1]$ and $p_j = \sum_{i=0}^{j} \theta_i \phi_i(x)$ and p_j interpolates $d(x)$ at the zeros of ϕ_{n+1}, then

$$\lim_{n\to\infty} (||d - p_j||_2)^2 = \lim_{n\to\infty} \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} (d(x) - p_j)^2 dx = 0$$

Chebyshev Polynomials

- Can also quantify the *size of the error* for a discrete approximation

Theorem (Chebyshev Truncation)

Under certain technical conditions, the error in approximating a function $d(x)$ is the same order as the last coefficient i.e.

$$d^j(x|\theta) = \sum_{i=0}^{j} \theta_i \psi_i(x)$$

$$|d(x) - d^j(x|\theta)| \approx O(\theta_j)$$

for any $x \in [-1, 1]$ and for any j.

- In practice, if estimated θ_j is too large, increase j

Chebyshev Polynomials

- Decades of research and applied work are testament to the usefulness of Chebshev Polynomials
- Words of wisdom from from John Boyd:
  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.

Distance Functions

- Suppose that we've chosen our basis functions
- Next step: Distance metric
- Two approaches
  1. Minimize norm over θ (for some weights)

$$||R(\cdot,|\theta)|| = \; < R(\cdot,|\theta), R(\cdot,|\theta) > \; = \int_\Omega R^2(x,|\theta)w(x)dx$$

  2. Project onto space of other functions, $\{\phi_i(\cdot)\}$ (*projection directions/test functions*) i.e. for $i = 1, \ldots, n$

$$0 = P_i(\theta) = \; < R(\cdot,|\theta), \phi_i(\cdot) > \; = \int_\Omega R(x,|\theta)\phi_i(x)w(x)dx$$

    - Idea: Residuals orthogonal to span of projection directions
    - Cannot 'predict' residuals within this span
    - Exact identification: Requires $n = j + 1$

Projection Directions 1: Least Squares

- The FOC to the following problem (when $w(x) = 1$)

$$\min_{\theta} \int_{\Omega} R^2(\mathbf{x}|\theta) d\mathbf{x}$$

- Equivalent to projection direction given by

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

- Problems
  1. Projection directions could be highly correlated
  2. Projection direction depends on initial guess for θ

Projection Directions 2: Subdomain

- Alternative is to split into $j+1$ subdomains (Ω_i) with a step function projection direction

$$\phi_i(\mathbf{x}) = \begin{cases} 1, & \text{if } \ \mathbf{x} \in \Omega_i \\ 0, & \text{otherwise} \end{cases}$$

- This is the same as solving the following system

$$\int_{\Omega_i} R(\mathbf{x}|\theta) d\mathbf{x} = 0, \ \ i = 1, \ldots, j+1$$

Projection Directions 3: Collocation

- Also called *pseudospectral* or *method of selected points*
- Directions given by *Dirac Delta* function, $\delta(\cdot)$
  - Assigns zero weight to all points except zero
  - Turns a continuous integration discrete

$$\phi_i(x) = \delta(x - x_i)$$

- Integral vanishes! Boils down to system

$$R(\mathbf{x}_i|\theta) = 0, \; i = 1, \ldots, j+1$$

- Best points: Use the zeros of the $(j+1)$th order Chebyshev polynomial
  - This is called *orthogonal collocation*

Projection Directions 4: Galerkin

- Project onto original set of basis functions i.e.

$$\phi_i(\mathbf{x}) = \psi_{i-1}(\mathbf{x})$$

$$\implies \int_\Omega \psi_i(\mathbf{x}) R(\mathbf{x}|\theta) d\mathbf{x} = 0, \;\; i = 1, \ldots, j+1$$

- Interpretation: Residual has to be orthogonal to each of the basis functions
- Generally very accurate, but complicated to code/takes a long time

Projection Directions 5: Moment-Matching

- Project onto $j+1$ monomials

$$\phi_i(\mathbf{x}) = x^{i-1}$$

- Looks familiar when you write it out...

$$\int_\Omega R(x, |\theta) x^{i-1} dx = 0 \text{ for } i = 1, \ldots, j+1$$

- Intuitive, but often not as efficient as alternative approaches
  - Monomials close to multicollinear; not much space to span

Example

- Back to our deterministic NCG model

$$c_t + k_{t+1} - k_t^{\alpha} - (1-\delta)k_t = 0$$

$$c_t^{-\sigma} - \beta(\alpha k_{t+1}^{\alpha-1} + 1 - \delta)c_{t+1}^{-\sigma}$$

- Considerable flexibility in how we want to solve
  1. Approximate $c_t(k_t|\theta)$. Use RC to define $k_{t+1}(k_t|\theta)$
  2. Approximate $k_{t+1}(k_t|\theta)$. Use RC to define $c_t(k_t|\theta)$
- Do it two ways:
  1. Chebyshev Polynomials and Chebyshev Collocation Weights
  2. Legendre Polynomials and Galerkin Weights

Important Note

- If $j < \infty$, we are *not* guaranteed that a set of θ's exist that zero out the distance function
- Do not use residuals as accuracy metric!
  - Residuals tell you either
    1. Gap at at certain points (collocation)
    2. Average gap over a region
  - Even if residuals non-zero, could be decent elsewhere
- "Proof is in the pudding"
  - Best accuracy metric: Euler Equation Errors

Dealing with Issues I

- If a set of θ do not seem to do well/exist, can always try hybrid techniques

$$\min_{\theta} \sum_{i=1}^{n} < R(\cdot|\theta), \phi_i >^2$$

  - Solution guaranteed to exist (may not be exact zero or unique)

- Done this way, you can also add *overidentification* i.e. for some $m > n$

$$\min_{\theta} \sum_{i=1}^{m} < R(\cdot|\theta), \phi_i >^2$$

note since $n = j + 1$, here there are more conditions than coefficients

Dealing with Issues II

- If a set of θ do not seem to do well/exist, notice we still have the freedom of weights!
- Projection directions are *not* weights; results still valid under different weights
- Often very helpful to place more weight on steady state e.g. let $\hat{X}$ be an ϵ-ball around the steady state, then

$$w(x) > 1 \;\forall x \in \hat{X} \text{ and } \; w(x) = 1 \;\forall x \in \hat{X}^{\complement}$$

Multistep Scheme

- Can exploit fact that polynomials are orthogonal
    - Each additional one is 'spanning' a space the previous ones did not
- Suggests 'Multi-step' step solution scheme for large j
    1. Solve model for some $j' < j$
    2. Use the optimal solution for j' as a guess for $j' + 1$
    3. Continue until we reach a solution for j
- Remarkably effective, especially since straight solution of very high-dimensional problems is finicky

Alternative: Finite Elements

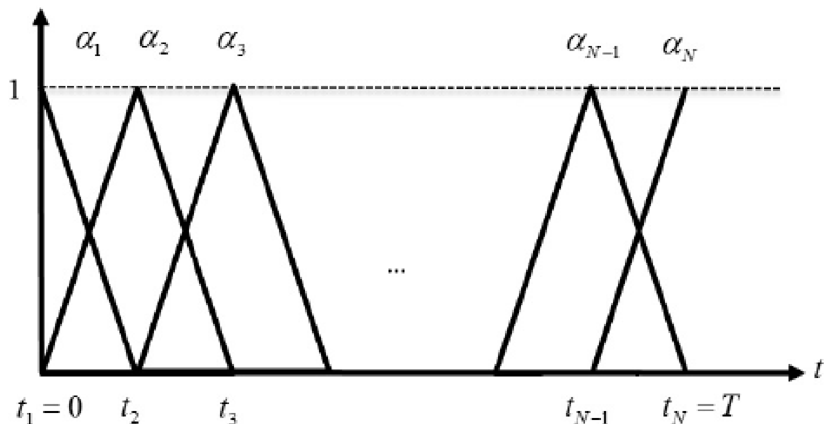
- Use as basis functions a series of tent functions
- Break up domain into N different subdomains (not necessarily equidistant)
  - $\{k_1, \ldots, k_N\}$, where $\underline{k} = k_1$ and $\bar{k} = k_N$
  - $k_i < k_{i+1}$ for any $i < N$
- 'Tent' basis functions given by (for $i \in \{2, N-1\}$)

$$\psi_i(k) = \begin{cases} \frac{k-k_{i-1}}{k_i-k_{i-1}}, & k \in [k_{i-1}, k_i] \\ \frac{k_{i+1}-k}{k_{i+1}-k_i}, & k \in [k_i, k_{i+1}] \\ 0, & o/w \end{cases}$$

and for the first and last functions

$$\psi_1(k) = \begin{cases} \frac{k_1-k}{k_1-k_0}, & k \in [k_0, k_1] \\ 0, & o/w \end{cases}, \ \psi_N(k) = \begin{cases} \frac{k-k_{N-1}}{k_N-k_{N-1}}, & k \in [k_{N-1}, k_N] \\ 0, & o/w \end{cases}$$

Alternative: Finite Elements



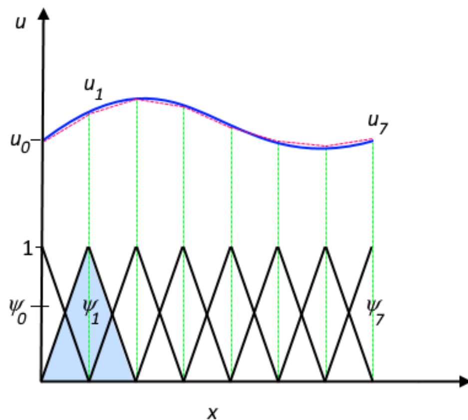
Alternative: Finite Elements

- Basis functions necessarily continuous
- Approximation is piecewise linear
- Collocation distance function best ($|\theta| = N$)

$$d^j(k|\theta) = d(k_j)$$

- Convergence
  - Need lots of points...
  - But not too bad in practice!
    - Very simple evaluation (linear)
    - Very orthogonal
- Used *a lot* in engineering

Alternative: Finite Elements



More States

- All of our basis functions so far have been one-dimensional
- How to handle 2 or more states? Three basic ways
  1. Discretize AR(1) processes
  2. Tensor Basis
  3. 'Trimmed' Tensor Basis
     - 3.1 Complete Polynomials
     - 3.2 Smolyak's Algorithm

Discretizing

- Suppose we want to approximate $k^\star(k, z)$

1. Start by discretizing z into a finite-state Markov process with n elements
2. For each state, $\{z_1, z_2, \ldots, z_n\}$, assume a unidimensional basis function i.e.

$$k^\star(k|z_m) = \sum_{i=0}^{j} \theta_i^m \psi_i(k) \text{ for } m = 1, \ldots, n$$

3. Solve system for $n(j+1)$ coefficients, $\{\theta_i^m\}_{i=0,m=1}^{j,n}$

Aside: Alternate Discretization

- Tauchenizing performs poorly when ρ very large
- Rouwenhurst Method: Based on Binomial $\rightarrow$ Normal
  1. Break domain into a grid with equally spaced points and

     $$\underline{z} = \mu_z - \sigma_z \sqrt{n-1}, \quad \bar{z} = \mu_z + \sigma_z \sqrt{n-1}$$

     where σ_z is unconditional volatility
  2. When $n = 2$, transition matrix is

     $$P^2 = \begin{bmatrix} p & 1-p \\ 1-q & q \end{bmatrix}$$

     where $p = q = \frac{1+\rho}{2}$
  3. For $n \geq 3$, $P^n = [\hat{P}^n_{1,\cdot}; .5\hat{P}^n_{2:n-1,\cdot}; \hat{P}^n_{n,\cdot}]$, where $\hat{P}^n$ given by

$$\hat{P}^n = p \begin{bmatrix} P^{n-1} & \mathbf{0} \\ \mathbf{0}' & 0 \end{bmatrix} + (1-p) \begin{bmatrix} \mathbf{0} & P^{n-1} \\ 0 & \mathbf{0}' \end{bmatrix} + (1-q) \begin{bmatrix} \mathbf{0}' & 0 \\ P^{n-1} & \mathbf{0} \end{bmatrix} + q \begin{bmatrix} 0 & \mathbf{0}' \\ \mathbf{0} & P^{n-1} \end{bmatrix}$$

Tensor Bases

- Tensors build multidimensional basis functions by finding the Kronecker product of all unidimensional basis functions
  - i.e. every possible multiple combination of them
- If we have n different states, s_i, and want to do a j-order approximation...

$$d^j(\cdot|\theta) = \sum_{i_1=0}^{j} \cdots \sum_{i_n=0}^{j} \theta_{i_1,\ldots,i_n} \prod_{k=1}^{n} \psi_{i_k}^k(s_k)$$

- Could vary j or basis functions across states if we wanted
- Advantage: If unidimensional basis is orthogonal, so is the tensor basis
- Disadvantage: Curse of dimensionality acute
  - $|\Theta| = (j+1)^n$, where n is number of states
  - 5 states and order 9 $\implies$ 100,000 coefficients

Tensor Bases: Multistep

- Since Tensor Bases orthogonal, multistep works well
- 2 states e.g. z and k: 'box' method
  - Add right and bottom sides to coefficient matrix with each update (set to zero)
  - Use previous solution as guess (interior of matrix)

$$\psi_0(k)\psi_0(z) \rightarrow \begin{bmatrix} \psi_0(k)\psi_0(z) & \psi_0(k)\psi_1(z) \\ \psi_1(k)\psi_0(z) & \psi_1(k)\psi_1(z) \end{bmatrix}$$

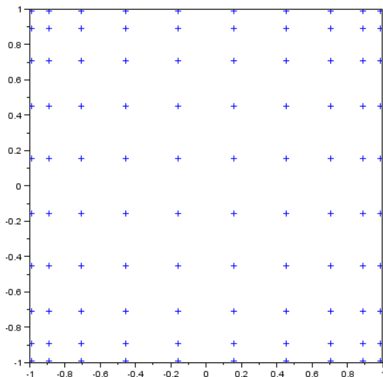
$$\rightarrow \begin{bmatrix} \psi_0(k)\psi_0(z) & \psi_0(k)\psi_1(z) & \psi_0(k)\psi_2(z) \\ \psi_1(k)\psi_0(z) & \psi_1(k)\psi_1(z) & \psi_1(k)\psi_2(z) \\ \psi_2(k)\psi_0(z) & \psi_2(k)\psi_1(z) & \psi_2(k)\psi_2(z) \end{bmatrix} \rightarrow$$

- Formula: Store coefficients in a line: Add $2N-1$ zeros each step
  - Matrix location of new coefficients: c_n from $n = 1, \ldots, 2N-1$

$$i = \min(c_n, N), \quad j = \min(2N - c, N)$$

Tensor Bases: Collocation

- Grid: Cartesian product of individual collocation grids
- Any component of a tensor is zero $\implies$ tensor is zero
    - Zeros of the unidimensional basis functions are also zeros of the tensor



Sparse Grid I: Complete Polynomials

- Tensor bases are preferred method for most people
- But how to deal with curse of dimensionality? Build a grid that is *sparse*
- Two popular ways: First are *Complete Polynomials*
  - Don't employ all elements of the Tensor
  - Keep only those whose order is less than κ for some $\kappa \geq j$

$$P_\kappa^n = \{\psi_{i_1}^1 \psi_{i_2}^2 \dots \psi_{i_n}^n | \sum_{l=1}^{n} i_l \leq \kappa\}$$

- e.g. Chebyshev polynomials of order $j = 4$ and 3 states
  - 125 coefficients in the full Tensor
  - Set $\kappa = 6 \implies$ Only 72 coefficients
- Second is *Smolyak's Algorithm* (later)

Complete Polynomials: Example

- 3D Multistep: Similar to two-dimensions, but in a cube
  - At each step add

$$(N-1) \times (2N-1) + N^2$$

    coefficients

- Add investment efficiency to RBC model (everything else same)

$$\mu_t = \rho_\mu \mu_{t-1} + \sigma_\mu \epsilon_{2,t}$$

$$k_{t+1} = (1-\delta)k_t + i_t e^{\mu_t}$$

Sparse Grid II: Smolyak's Algorithm

- More efficient (and complicated) way to reduce grid
- Smolyak's Algorithm
  - Collocation method
  - Judiciously select grid
  - Grid size grows polynomially (much slower than exponential)
- Clusters points around
  1. Corners of domain of Chebyshev
  2. Central cross of domain of Chebyshev
- Very useful (and used a lot)!
  1. Malin et al. (2011, JEDC) use to globally solve model with 20 continuous state variables!
  2. Bocola (2016, JPE) uses it estimate (not calibrate) a large scale, non-linear default model
- **The** most powerful tool you have at your disposal for large scale, global, relatively well-behaved problems

Laying the Base

- Assumes Chebyshev polynomials/collocation

1. **Transform the Domain of the State Variables**
   - For state variable $\tilde{x}_l$, with $l = 1, \ldots, n$ on a domain $[a, b]$, use linear translation
   $$x_l = 2\frac{\tilde{x}_l - a}{b - a} - 1$$
   - Puts entire grid in n-dimensional cube $[-1, 1]^n$
2. **Set the Order of the Polynomial**
   - Define $m_1 = 1$
   - Set $m_i = 2^{i-1} + 1$, where $m_i - 1$ is the order of the approximating polynomial
   - e.g.
   $$m_1 = 1, \quad m_2 = 3, \quad m_3 = 5, \quad m_4 = 9, \quad m_5 = 17, \ldots$$

The Grid

3. **Building the Gauss-Lobotto Nodes**
   - Find *extrema* (not zeros) of Chebyshev polynomials
   - Build sets

$$\mathcal{G}^i = \{\zeta_1^i, \dots, \zeta_{m_i}^i\} \subset [-1, 1]$$

   where

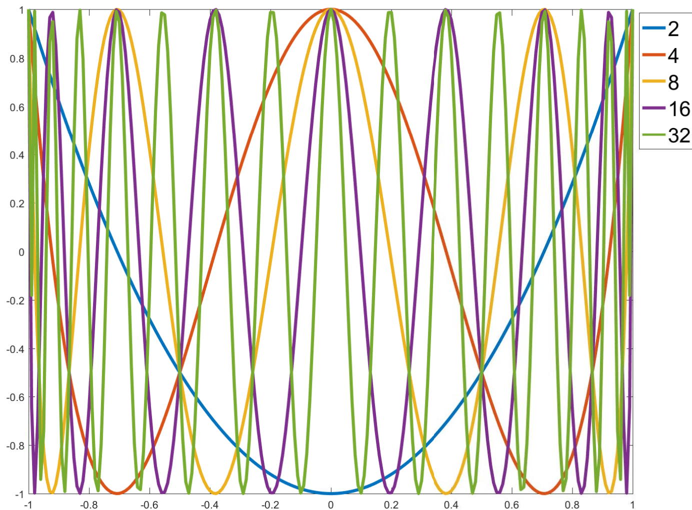
$$\zeta_j^i = -\cos\left(\frac{j-1}{m_i - 1}\pi\right), \quad j = 1, \dots, m_i$$

   - Initial set $\mathcal{G}^1 = \{0\}$
   - Crucial! Since $m_i = 2^{i-1} + 1$, grids are nested e.g.

$$\mathcal{G}^2 = \{-1, 0, 1\}$$

$$\mathcal{G}^3 = \{-1, -\cos\left(\frac{\pi}{4}\right), 0, -\cos\left(\frac{3\pi}{4}\right), 1\}$$

Nested Extrema



Sparse Grid

4. **Building a Sparse Grid**
   - Select an accuracy metric $q > n$
     - Larger $q \implies$ more accurate
     - Experience suggests $q = n + 2$ or $q = n + 3$ work well
   - Sparse grid defined to be union of the Cartesian products

$$G(q,n) = \cup_{q-n+1 \leq |\mathbf{i}| \leq q} \left(\mathcal{G}^{i_1} \times \cdots \times \mathcal{G}^{i_n} \right)$$

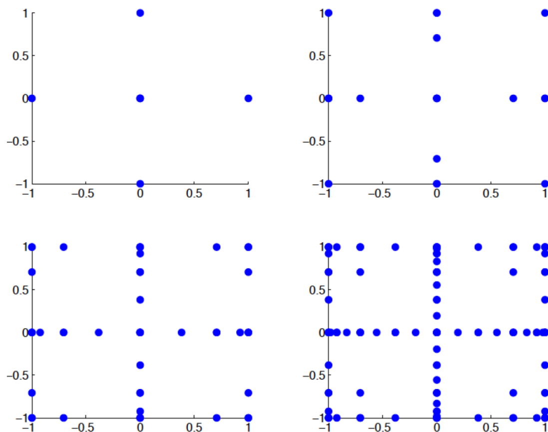
   where $|i| = \sum_{l=1}^{n} i_l$ (l just indexes dimension)
   - e.g.

$$G(3,2) = \cup_{2 \leq |\mathbf{i}| \leq 3} \left(G^{i_1} \times G^{i_2} \right)$$

$$= \left(\mathcal{G}^1 \times \mathcal{G}^1\right) \cup \left(\mathcal{G}^1 \times \mathcal{G}^2\right) \cup \left(\mathcal{G}^2 \times \mathcal{G}^1\right)$$

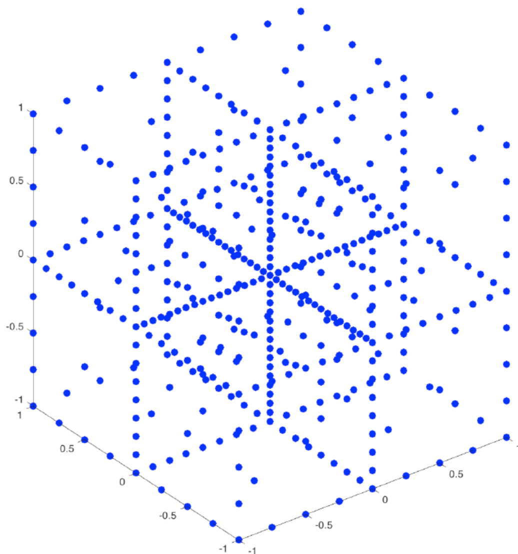
$$= \{ \underbrace{(0,0)}_{(\mathcal{G}^1 \times \mathcal{G}^1)}, \underbrace{(0,-1),(0,1)}_{(\mathcal{G}^1 \times \mathcal{G}^2)/(0,0)}, \underbrace{(-1,0),(1,0)}_{(\mathcal{G}^2 \times \mathcal{G}^1)/(0,0)} \}$$

Sparse Grids: 2 Dimensions



Left to right: $G(3,2)$, $G(4,2)$, $G(5,2)$, and $G(6,2)$

Sparse Grids: 3 Dimensions



Sparse Grid: Important Properties

- Points cluster around
  1. Central cross
  2. Corners of domain
- Nested: $G(q, n) \subset G(q+1, n)$
- Grid size grows polynomially on n^{q-n}
  - e.g. When $q = n + 2$...

n	$G(q, n)$	5^n
2	13	25
3	25	125
4	41	625
5	61	3125
...	...	...
12	313	244,140,625

Implementing

5. **Building the Tensor Product**
   - Use Chebyshev Polynomials to build tensor-product multivariate polynomial
   $$\psi_i(x_i) = T_{i-1}(x_i)$$
   - These will *not yet* be our function approximation!
   $$p^{i_1,\dots,i_n}(x|\theta) = \sum_{l_1=1}^{m_{i_1}} \cdots \sum_{l_n=1}^{m_{i_n}} \theta_{l_1 \dots l_n} \psi_{l_1}(x_1) \dots \psi_{l_n}(x_n)$$
   where $|i| = \sum_{l=1}^{n} i_l$, $x_i \in [-1, 1]$, $x = \{x_1, \dots, x_n$, and θ stacks all the coefficients, $\theta_{l_1 \dots l_n}$
   - Now, define
   $$p^{|i|}(x|\theta) = \sum_{i_1,\dots,i_n \text{ s.t. } |i|=\sum_{l=1}^{n} i_l} p^{i_1,\dots,i_n}(x|\theta)$$

Tensor Product: Example

- Tensor product example: $n = 2$ and $q = 3$ (recall $m_1 = 1$, $m_2 = 3$)

$$p^{1,1}(x|\theta) = \sum_{l_1=1}^{m_1} \sum_{l_2=1}^{m_1} \theta_{l_1 l_2} \psi_{l_1}(x_1) \psi_{l_2}(x_2) = \theta_{11}$$

$$p^{1,2}(x|\theta) = \sum_{l_1=1}^{m_1} \sum_{l_2=1}^{m_2} \theta_{l_1 l_2} \psi_{l_1}(x_1) \psi_{l_2}(x_2) = \theta_{11} + \theta_{12} T_1(x_2) + \theta_{13} T_2(x_2)$$

$$p^{2,1}(x|\theta) = \sum_{l_1=1}^{m_2} \sum_{l_2=1}^{m_1} \theta_{l_1 l_2} \psi_{l_1}(x_1) \psi_{l_2}(x_2) = \theta_{11} + \theta_{21} T_1(x_1) + \theta_{31} T_2(x_1)$$

- Implies

$$p^{|2|}(x|\theta) = p^{1,1}(x|\theta)$$

$$p^{|3|}(x|\theta) = p^{1,2}(x|\theta) + p^{2,1}(x|\theta)$$

Tensor Product: Properties

- Some coefficients repeated in Tensor e.g. θ_{11}
- Number of coefficients = Cardinality of $G(q, n)$
- Given function values at a grid point, $d(\cdot)$, coefficients have a linear, closed form solution
    - Motivates this particular structure in the first place

Approximating Function

6. **Building the Interpolating Function in n Dimensions**
   - The Smolyak function that interpolates on $G(q,n)$ is

$$d(x|\theta,q,n) = \sum_{\max(n,q-n+1)\leq|i|\leq q} (-1)^{q-|i|} \binom{n-1}{q-|i|} p^{|i|}(x|\theta)$$

   i.e. a weighted sum of the Tensors

   - $\binom{n-1}{q-|i|}$ is a combination, not a vector
     - Number of different ways you can select $q-|i|$ elements from a set of size $n-1$ (order doesn't matter)

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Approximating Function

- Application: Again to $n = 2$ and $q = 3$

$$d(x|\theta, q, n) = \sum_{2 \leq |i| \leq 3} (-1)^{3-|i|} \binom{1}{3-|i|} p^{|i|}(x|\theta)$$

$$= -\binom{1}{1} p^{|2|}(x|\theta) + \binom{1}{0} p^{|3|}(x|\theta)$$

$$= p^{1,2}(x|\theta) + p^{2,1}(x|\theta) - p^{1,1}(x|\theta)$$

$$= \theta_{11} + \theta_{21} T_1(x_1) + \theta_{31} T_2(x_1) + \theta_{12} T_1(x_2) + \theta_{13} T_2(x_2)$$

- Intuition scales up with dimensions/q

Solving

7. **Solving for the Polynomial Coefficients**
   - By construction, number of coefficients $M = \#\theta$ is exactly equal to the grid size i.e. $M = \#G(q, n)$
   - Evaluate residuals given approximation at each grid point (collocation)
     - M nonlinear equations
     - M unknowns
   - Nonlinear solver delivers θ

Smolyak Speed-Up

- Multi-step technically possible, but rather complicated
- Better strategy: Combine perturbation and projection
  1. First-order perturbation of model
  2. Alter coefficients $\implies$ Initial guess = perturbed function
- If first $n+1$ coefficients are linear (constant), we have our interpolating function

$$\theta_0 = d_{ss}$$

$$\theta_i = d_i \times \frac{i_{UB} - i_{LB}}{2} \text{ for each state } i$$

- Comes from

$$d_i(s_1, \ldots, s_n|\theta) = f(2\frac{s_1 - 1_{LB}}{1_{UB} - 1_{LB}} - 1, \ldots, 2\frac{s_n - n_{LB}}{n_{UB} - n_{LB}} - 1|\theta)$$

Dynamic Programming/Projection

- Alternative to spline interpolation
  - Interpolate between known grid points with projected function
  - Simple, closed-form solution for coefficients
- Consider a 1D VFI update step with N gridpoints
  - At update step i, we have (at $x_1, x_2, \ldots, x_N$)

$$\{V_1^i, V_2^i, \ldots, V_N^i\}$$

  - Approximate value function with an $N-1$ degree polynomial

$$V_1^i = \theta_0 + \theta_1 T_1(x_1) + \ldots \theta_{N-1} T_{N-1}(x_1)$$

$$V_2^i = \theta_0 + \theta_1 T_1(x_2) + \ldots \theta_{N-1} T_{N-1}(x_2)$$

$$\ldots$$

$$V_N^i = \theta_0 + \theta_1 T_1(x_N) + \ldots \theta_{N-1} T_{N-1}(x_N)$$

Dynamic Programming/Projection

- Notice this is a N-dimensional linear system in θ

$$\underbrace{V}_{N\times 1} = \underbrace{T(x)}_{N\times N}\underbrace{\Theta}_{N\times 1}$$

$$\implies \Theta = [T(x)]^{-1}V$$

- Works best when $\{x_j\}_{j=1}^{N}$ are Chebyshev zeros/extrema
- $T(x)$ non-invertible, least-squares residual problem usually works $\implies$

$$\Theta = [T(x)'T(x)]^{-1}T(x)'V$$

  - Could weight some states more heavily if we want to

Dynamic Programming/Projection

- Multidimensional Interpolation: Use
  1. Kronecker product
  2. Tensors
- With n dimensions (and N points along each)

$$\underbrace{V}_{N^n \times 1} = \underbrace{T}_{N^n \times N^n} \underbrace{\Theta}_{N^n \times 1}$$

- Where $T = T_n(x) \otimes T_{n-1(x)} \otimes \cdots \otimes T_1(x)$
  - No need to invert large matrix
  - Closed-form result from tensor algebra

$$\Theta = \left[[T_n(x)]^{-1} \otimes [T_{n-1(x)}]^{-1} \otimes \cdots \otimes [T_1(x)]^{-1} \right] V$$