SciencesPo Computational Economics
Spring 2017

Florian Oswald

April 6, 2017

1 Function Approximation

Sciences Po, Spring 2017

1.1 Outline

1. Overview of Approximation Methods
    1. Interpolation
    2. Regression
2. Polynomial Interpolation
3. Spline Interpolation
4. Multidimensional Approximation

1.2 Approximation Methods

- Confronted with a non-analytic function f (i.e. something not like $log(x)$), we need a way to numerically represent f in a computer.
    - If your problem is to compute a value function in a dynamic problem, you don't have an *analytic representation* of V.
    - If you need to compute an equilibrium distribution for your model, you probably can't tell it's from one parametric family or another.
- Approximations use *data* of some kind which informs us about f. Most commonly, we know the function values $f(x_i)$ at a corresponding finite set of points $X = \{x_i\}_{i=1}^N$.
- The task of approximation is to take that data and tell us what the function value is at $f(y), y \notin X$.
- To an economist this should sound very familiar: take a dataset, learn it's structure, and make predictions.
- The only difference is that we can do much better here, because we have more degree's of freedom (we can choose our X in $Y = \beta X + \epsilon$)

1.3 Some Taxonomy

- Local Approximations: approximate function and it's derivative f, f' at a *single* point x_0. Taylor Series:

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2}f''(x_0) + \cdots + \frac{(x - x_0)^n}{n!}f^n(x_0)$$

- Interpolation or *Colocation*: find a function $\hat{f}$ that is a good fit to f, and require that $\hat{f}$ *passes through* the points. If we think of there being a *residual* $\epsilon_i = f(x_i) - \hat{f}(x_i)$ at each grid point i, this methods succeeds in setting $\epsilon_i = 0, \forall i$.
- Regression: Minimize some notion of distance (or squared distance) between $\hat{f}$ and f, without the requirement of pass through.

1.4 Doing Interpolation in Julia

- In practice, you will make heavy use of high-quality interpolation packages in julia.
- List in the end.
- Nevertheless, function approximation is *extremely* problem-specific, so sometimes a certain approach does not work for your problem.
- This is why we will go through the mechanics of some common methods.
- I would like you to know where to start drilling if you need to go and hack somebody elses code.

1.5 Interpolation Basics

- Let f be a smooth function mapping $\mathbb{R}^d \mapsto \mathbb{R}$, and define $\hat{f}(\cdot; c)$ to be our parametric approximation function. We generically define this as

$$\hat{f}(x; c) = \sum_{j=1}^{J} c_j \phi_j(x)$$

where * $\phi_j : \mathbb{R}^d \mapsto \mathbb{R}$ is called a **basis function**, * $c = c_1, c_2, \ldots, c_J$ is a coefficient vector * The integer J is the *order* of the interpolation. * Our problem is to choose (ϕ_i, c) in some way.

We will construct a *grid* of $M \geq J$ points $x_1, \ldots, x_M$ within the domain $\mathbb{R}^d$, and we will denote the *residuals* at each grid point by $\epsilon = \epsilon_1, \ldots, \epsilon_M$:

$$\begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_M \end{bmatrix} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_M) \end{bmatrix} - \begin{bmatrix} \phi_1(x_1) & \ldots & \phi_J(x_1) \\ \vdots & \ddots & \vdots \\ \phi_1(x_M) & \ldots & \phi_J(x_M) \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ \vdots \\ c_J \end{bmatrix}$$

- *Interpolation* or *colocation* occurs when $J = M$, i.e. we have a square matrix of basis functions, and can exactly solve this.
- We basically need to solve the system

$$\sum_{j=1}^{n} c_j \phi_j(x_i) = f(x_i), \forall i = 1, 2, \ldots, n$$

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

where the second line uses vector notation, and $\mathbf{y}$ has all values of f. * Solution: $\mathbf{c} = \mathbf{\Phi}^{-1} y$.

1.6 Regression Basics

- Clearly, on the previous slide we required that there are as many interpolation nodes as there are basis functions - we had J equations for M unknowns, so there exists a unique solution for c.
- We needed to *invert* $\mathbf{\Phi}$.
- If we have more, $M > J$ say, interpolation nodes than basis functions, we cannot do that. Instead we can define a loss function, and minimize it.
- In the case of squared loss, of course, this leads to the least squares solution:

$$e_i = f(x_i) - \sum_{j=1}^{n} c_j \phi_j(x_i)$$

$$\min_c e_i^2 \implies$$

$$c = (\Phi'\Phi)^{-1}\Phi' y$$

1.7 Spectral and Finite Element Methods

- **Spectral Methods** are such that the basis functions are non-zero over the entire domain of f.
  - Polynomial interpolation
  - Chebychev interpolation
- **Finite Element** methods are such that basis functions are non-zero only on a subset of the domain.
  - Splines
    - Linear splines, i.e. splines of degree 1, a.k.a. *linear approximation*
    - Higher order splines, mainly the *cubic spline*.

1.8 What makes a good Approximation?

- Should be arbitrarily accurate as we increase n.
- Φ Should be efficiently (fast) computable. If Φ were differentiable, we could easily get e.g. $\hat{f}'(x) = \sum_{j=1}^{J} c_j \phi_j'(x_i)$
- c Should be efficiently (fast) computable.

2 Polynomial Interpolation

- For any continuous real-valued function f on interval $[a, b]$, and an $\epsilon > 0$, there is a polynomial p, such that

$$||f - p||_\infty \equiv \sup_{x \in [a,b]} |f(x) - p(x)| < \epsilon$$

- However, the choice of p is critical. *Orthogonal Polynomials* have been shown to perform very well.

  Definition - Orthogonal Polynomials: an orthogonal polynomial sequence is a family of polynomials such that any two different polynomials in the sequence are orthogonal to each other under some inner product.

- There are many families that satisfy this. See [@judd-book] [1] table 6.3 for an overview.
- We will now look at a widely used family, the **Chebyshev polynomial**.

2.1 Chebyshev Nodes

- *Chebyshev Nodes* are defined in the interval $[-1, 1]$ as

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

- Which maps to general interval $[a, b]$ as

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2k-1}{2n}\pi\right), k = 1, \ldots, n$$

- Chebyshev nodes are **not** evenly spaced: there are more points towards the boundaries. You might remember that from Gaussian Integration nodes.

```
In [1]: using Plots
        using FastGaussQuadrature
        nodes = gausschebyshev(11)  # generate 11 Chebyshev Nodes
        Plots.scatter(nodes,ones(11),grid=false,ylims=(0.9,1.1),m=(:+,:red),label="nodes")
```

```
INFO: Recompiling stale cache file /Users/florian.oswald/.julia/lib/v0.5/Plots.ji for module Plo
```

2.2 What Polynomial to use? What form for Φ?

- In principle the *monomial basis* could be used. It is just the power functions of x: $1, x, x^2, x^3, \ldots$
- Stacking this up for each evaluation node (Chebyshev, or any other), gives the *Vandermonde Matrix*:

$$V = \begin{bmatrix} 1 & x_1 & \ldots & x_1^{n-2} & x_1^{n-1} \\ 1 & x_2 & \ldots & x_2^{n-2} & x_2^{n-1} \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & x_m & \ldots & x_m^{n-2} & x_m^{n-1} \end{bmatrix}$$

for the case with m evaluation nodes for x, and n basis functions for each x_i. * V is often ill-conditioned and thus a bad choice. * A much better polynomial basis is - *surprise, surprise* - the *Chebyshev Polynomial basis*.

2.3 Evaluating the Chebyshev Polynomial and Basis Function

- As before, this Basis is defined in $[-1, 1]$, so for general $x \in [a, b]$ we normalize x to

$$z_i = 2\frac{x_i - a}{b - a} - 1$$

- There are several ways to obtain the value of the order j Chebyshev polynomial at z, e.g. to get $\sum_{i=0}^{n} c_i T_i(z)$.

  1. use definition $T_j(z) = \cos(\arccos(z)j)$. Or

2. Recursively we have that $\phi_j(x) = T_{j-1}(z)$, and

$$\begin{aligned} T_0(z) &= 1 \\ T_1(z) &= z \\ T_{i+1}(z) &= 2zT_i(z) - T_{i-1}(z), i = 1, \ldots, n \end{aligned}$$

2.3.1 Constructing Φ as T evaluated *at* the Chebyshev Nodes

- Combining Chebyshev nodes evaluated at the roots T of the Cheby polynomial to construct Φ is a particularly good idea.
- Doing so, we obtain an interpolation matrix Φ with typical element

$$\phi_{ij} = \cos\left(\frac{(n - i + 0.5)(j-1)\pi}{n}\right)$$

- And we obtain that Φ is indeed orthogonal

$$\Phi^T\Phi = \text{diag}\{n, n/2, n/2, \ldots, n/2\}$$

2.4 (Chebyshev) Interpolation Proceedure

- Let's summarize this proceedure.
- Instead of Chebyshev polynomials we could be using any other suitable family of polynomials.
- To obtain a Polynomial interpolant $\hat{f}$, we need:

  1. a function to f interpolate. We need to be able to get the function values somehow.
  2. A set of (Chebyshev) interpolation nodes at which to compute f
  3. An interpolation matrix Φ that corresponds to the nodes we have chosen.
  4. A resulting coefficient vector c

- To obtain the value of the interpolation at x' off our grid, we also need a way to evaluate $\Phi(x')$.

  1. Evaluate the Basis function Φ at x': get $\Phi(x')$
  2. obtain new values as $y = \Phi(x')c$.

2.5 Polynomial Interpolation with `Julia`: `ApproxFun.jl`

- `ApproxFun.jl` is a Julia package based on the Matlab package `chebfun`. It is quite amazing.
- More than just *function approximation*. This is a toolbox to actually *work* with functions.
- given 2 functions f, g, we can do algebra with them, i.e. $h(x) = f(x) + g(x)^2$
- We can differentiate and integrate
- Solve ODE's and PDE's
- represent periodic functions
- Head over to the website and look at the readme.

```
In [2]: # works even with discontinuities!
        using Plots
        using ApproxFun
```

```
        ff = x->sign(x-0.1)/2 + cos(4*x);  # sign introduces a jump at 0.1
        x  = Fun(identity)         # set up a function space
        space(x)
        f  = ff(x)         # define ff on that space
        plot(f)         # plot
```

```
INFO: Recompiling stale cache file /Users/florian.oswald/.julia/lib/v0.5/ApproxFun.ji for module
```

```
In [3]: # whats the first deriv of that function at at 0.785?
        println(f'(0.785))
        # and close to the jump, at 0.100001?
        println(f'(0.1000001))
        # integral of f?
        g1 = cumsum(f)
        g = g1 + f(-1)
        integral = norm(f-g)
        integral
```

```
-0.006370611665976966
-1.5576748429316383
```

```
Out[3]: 2.152565021925387
```

- The main purpose of this package is to manipulate analytic functions, i.e. function with an algebraic representation.
- There is the possibility to supply a set of data points and fit a polynomial:

```
In [4]: S = Chebyshev(1..2);
        n = 50;
        p = linspace(1,2,n);    # a non-default grid
        v = exp.(p);            # values at the non-default grid
        V = Array(Float64,n,n); # Create a Vandermonde matrix by evaluating the basis at the gri
        for k = 1:n
            V[:,k] = Fun(S,[zeros(k-1);1]).(p)
        end
        f = Fun(S,V\v);
        println("approx = $(f(1.1))")
        println("true = $(exp(1.1))")

        # Check out more examples on the [docs](http://juliaapproximation.github.io/ApproxFun.jl
```

```
approx = 3.0041660237524423
true = 3.0041660239464334
```

2.6 Splines: Piecewise Polynomial Approximation

- Splines are a finite element method, i.e. there are regions of the function domain where some basis functions are zero.

- As such, they provide a very flexible framework for approximation instead of high-order polynomials.
  - Keep in mind that Polynomials basis functions are non-zero on the entire domain. Remember the Vandermonde matrix.
- They bring some element of local approximation back into our framework. What happens at one end of the domain to the function is not important to what happens at the other end.
- Looking back at the previous plot of random data: we are searching for **one** polynomial to fit **all** those wiggles. A spline will allow us to design **different** polynomials in different parts of the domain.

2.6.1 Splines: Basic Setup

- The fundamental building block is the *knot vector*, or the *breakpoints vector* $\mathbf{z}$ of length p. An element of $\mathbf{z}$ is z_i.
- $\mathbf{z}$ is ordered in ascending order.
- $\mathbf{z}$ spans the domain $[a, b]$ of our function, and we have that $a = z_1, b = z_p$
- A spline is of *order* k if the polynomial segments are k-th order polynomials.
- Literature: [@deboor] [**?**] is the definitive reference for splines.

2.6.2 Splines: Characterization

- Given p knots, there are $p-1$ polynomial segments of order k, each characterized by $k+1$ coefficients, i.e. a total of $(p-1)(k+1)$ parameters.
- However, we also require the spline to be continuous and differentiable of degree $k-1$ at the $p-2$ interior breakpoints.
- Imposing that uses up an additional $k(p-2)$ conditions.
- We are left with $n = (p-1)(k+1) - k(p-2) = p+k-1$ free parameters.
- A Spline of order k and p knots can thus be written as a linear combination of it's $n = p+k-1$ basis functions.

2.6.3 Splines: Show some Basis Functions

```
In [5]: using ApproXD
        # Pkg.clone("https://github.com/floswald/ApproXD.jl")
        bs = BSpline(7,3,0,1) #7 knots, degree 3 in [0,1]
        # how many basis functions? (go back 1 slide.)
        # getNumCoefs(bs)
        B = full(getBasis(collect(linspace(0,1.0,500)),bs))
        plot(linspace(0,1.0,500),B,layout=(3,3),grid=false,ylim=(-0.1,1.1),label=transpose(["Bas

WARNING: the no-op `transpose` fallback is deprecated, and no more specific `transpose` method f
 in depwarn(::String, ::Symbol) at ./deprecated.jl:64
 in transpose(::String) at ./deprecated.jl:770
 in (::Base.##207#208)(::Tuple{Int64,String}) at ./<missing>:0
 in collect(::Base.Generator{Base.Prod2{Base.OneTo{Int64},Array{String,1}},Base.##207#208}) at .
 in transpose(::Array{String,1}) at ./arraymath.jl:416
 in include_string(::String, ::String) at ./loading.jl:441
```

```
 in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /Users/florian.oswald/.julia/v0.5/IJulia/src/
 in eventloop(::ZMQ.Socket) at /Users/florian.oswald/.julia/v0.5/IJulia/src/eventloop.jl:8
 in (::IJulia.##13#19)() at ./task.jl:360
while loading In[5], in expression starting on line 7
```

```
In [6]: # Notice that placing each of those panels on top of each other generates a sparse matri
        println(getBasis(collect(linspace(0,1.0,5)),bs))
        plot(linspace(0,1.0,500),B,grid=false,ylim=(-0.1,1.1),legend=false)
```

```
[1, 1]  =  1.0
[1, 2]  =  0.0
[2, 2]  =  0.03125
[1, 3]  =  0.0
[2, 3]  =  0.46875
[1, 4]  =  0.0
[2, 4]  =  0.479167
[3, 4]  =  0.166667
[2, 5]  =  0.0208333
[3, 5]  =  0.666667
[4, 5]  =  0.0208333
[3, 6]  =  0.166667
[4, 6]  =  0.479167
[3, 7]  =  0.0
[4, 7]  =  0.46875
[4, 8]  =  0.03125
[5, 9]  =  1.0
```

2.7 B-Splines: Definition

- We mostly use Basis Splines, or **B-Splines**.
- Here is a recursive definition of a B-Spline (and what is used in `ApproXD`):
- Denote the j-th basis function of degree k with knot vector $\mathbf{z}$ at x as $B_j^{k,\mathbf{z}}(x)$
- Again, there are $n = k + p - 1$ B's (where p= `length(z)`)
- We can define $B_j^{k,\mathbf{z}}(x)$ recursively like this:

$$B_j^{k,\mathbf{z}}(x) = \frac{x - z_{j-k}}{z_j - z_{j-k}} B_{j-1}^{k-1,\mathbf{z}}(x) + \frac{z_{j+1} - x}{z_{j+1} - z_{j+1-k}} B_j^{k-1,\mathbf{z}}(x), j = 1, \ldots, n$$

- The recursion starts with

$$B_j^{0,\mathbf{z}}(x) = \begin{cases} 1 & \text{if } z_j \leq x < z_{j+1} \\ 0 & \text{otherwise.} \end{cases}$$

- For this formulation to work, we need to extend the knot vector for $j < 1, j > p$:

$$z_j = \begin{cases} a & \text{if } j \leq 1 \\ b & \text{if } j \geq p \end{cases}$$

- And we need to set the endpoints

$$B_0^{k-1,\mathbf{z}} = B_n^{k-1,\mathbf{z}} = 0$$

- You may see that this gives rise to a triangular computation strategy, as pointed out here.

2.8 B-Splines: Derivatives and Integrals

- This is another very nice thing about B-Splines.
- The derivative wrt to it's argument x is

$$\frac{dB_j^{k,\mathbf{z}}(x)}{dx} = \frac{k}{z_j - z_{j-k}} B_{j-1}^{k-1,\mathbf{z}}(x) + \frac{k}{z_{j+1} - z_{j+1-k}} B_j^{k-1,\mathbf{z}}(x), j = 1, \ldots, n$$

- Similarly, the Integral is just the sum over the basis functions:

$$\int_a^x B_j^{k,\mathbf{z}}(y)dy = \sum_{i=j}^{n} \frac{z_i - z_{i-k}}{k} B_{i+1}^{k+1,\mathbf{z}}(x)$$

2.9 Linear B-Spline: A useful special case

- This is *connecting the dots with a straight line*
- This may incur some approximation error if the underlying function is very curved between the dots.
- However, it has some benefits:
  - it is shape-preserving,
  - it is fast,
  - it is easy to build.
- For a linear spline with evenly spaced breakpoints, this becomes almost trivial.
  - Let's define $h = \frac{b-a}{n-1}$ as the distance between breakpoints.
  - Our basis function becomes very simple, giving us a measure of how far x is from the next knot:

$$\phi_j(x) = \begin{cases} 1 - \frac{|x - z_j|}{h} & \text{if } |x - z_j| \leq h \\ 0 & \text{otherwise} \end{cases}$$

- Notice that each interior basis function (i.e. not 0 and not n) has witdth $2h$.

```
In [7]: bs = BSpline(9,1,0,1) #9 knots, degree 1 in [0,1]
        B = full(getBasis(collect(linspace(0,1.0,500)),bs))
        plot(linspace(0,1.0,500),B,layout=(3,3),grid=false,ylim=(-0.1,1.1),label=transpose(["Bas
```

2.10 Linear B-Spline: Evaluation

- In order to evaluate the linear interpolator, we need to know only one thing: Which knot span is active, i.e. what is j s.t. $x \in [z_j, z_{j+1}]$?
- This is a classic problem in computer science. Binary search.
- `julia` implements `searchsortedlast`.
- Once we know j, it's easy to get the interpolated value as

$$\hat{f}(x) = \frac{x - z_j}{h} f(z_{j+1}) + \frac{z_{j+1} - x}{h} f(z_j)$$

2.11 The Importance of Knot Placement

- We just talked about *equally spaced knots*. This is just a special case.
- B-Splines give us the flexibility to place the knots where we want.
- Contrary to Polynomial interpolations (where we cannot choose the evaluation nodes), this is very helpful in cases where we know that a function is very curved in a particular region.
- Canonical Example: Runge's function: $f(x) = (1 + 25x^2)^{-1}$.
- Also: If you know that your function has a kink (i.e. a discontinuous first derivative) at $\hat{x}$, then you can stack breakpoints on top of each other *at* $\hat{x}$

```
In [8]: h(x) = (1+25x^2)^(-1)
        plot(h)
```

2.12 Interpolation with `Interpolations.jl`

```
# Nearest-neighbor interpolation
itp = interpolate(a, BSpline(Constant()), OnCell())
v = itp[5.4]   # returns a[5]

# (Multi)linear interpolation
itp = interpolate(A, BSpline(Linear()), OnGrid())
v = itp[3.2, 4.1]  # returns 0.9*(0.8*A[3,4]+0.2*A[4,4]) + 0.1*(0.8*A[3,5]+0.2*A[4,5])

# Quadratic interpolation with reflecting boundary conditions
# Quadratic is the lowest order that has continuous gradient
itp = interpolate(A, BSpline(Quadratic(Reflect())), OnCell())

# Linear interpolation in the first dimension, and no interpolation (just lookup) in the second
itp = interpolate(A, (BSpline(Linear()), NoInterp()), OnGrid())
v = itp[3.65, 5]  # returns  0.35*A[3,5] + 0.65*A[4,5]
```

```
In [9]: # In-place interpolation
        # restart kernel!
        using Interpolations
        A = rand(10,5)
        itp = interpolate!(A, Interpolations.BSpline(Quadratic(InPlace())), OnCell())
        itp[0.2,0.9]
```

```
INFO: Recompiling stale cache file /Users/florian.oswald/.julia/lib/v0.5/Interpolations.ji for m
```

```
Out[9]: 0.7077664255335139
```

- Very often we need a **gridded** interpolation. I.e. we supply the function values on an irregular grid.
- For this occasion, the `GriddedInterpolation` type is useful.
- For now this only works in 3 modes:
  - Gridded(Linear())
  - Gridded(Constant()) nearest neighbor
  - NoInterp (you must supply index ON grid)

```
In [10]: A = rand(20)
         A_x = collect(1.0:2.0:40.0)
         knots = (A_x,)
         itp = interpolate(knots, A, Gridded(Linear()))
         itp[2.0]

Out[10]: 0.20974428433562797

In [11]: # 2D
         A = rand(8,20)
         knots = ([x^2 for x = 1:8], [0.2y for y = 1:20])
         itp = interpolate(knots, A, Gridded(Linear()))
         itp[4,4.1]

Out[11]: 1.0187081420597015

In [12]: # we can mix modes across dimensions!
         itp = interpolate(knots, A, (Gridded(Linear()),Gridded(Constant())))
         itp[4,4.1]

Out[12]: 0.6971995070839345
```

- What about vector valued interpolations?
- Suppose we have a function $f : \mathbb{R} \mapsto \mathbb{R}^2$
- Economics example:

$$f(x) = \begin{pmatrix} \text{savings}(x) \\ \text{consumption}(x) \end{pmatrix}$$

```
In [13]: using FixedSizeArrays
         using Interpolations
         # a = rand(2,200)  # like f(x) above
         a = Float64[log(1+i)/j for i in 1:2, j in linspace(1,3,200)]
         b = reinterpret(Vec{2,Float64}, a, (200,))
         bitp = scale(interpolate(b, Interpolations.BSpline(Quadratic(Reflect())), OnCell()),lin

         # just compute the sum to demonstrate speed
         function foo(itp, X)
```

```
            s = zero(eltype(itp))
            for x in X
                s += itp[x]
            end
            s
        end

        X = rand(10^7)

        # After warming up with one call to `foo(bitp, X)`:
        foo(bitp, X);
        @time foo(bitp, X)
```

```
  0.138240 seconds (131 allocations: 7.750 KB)
```

```
Out[13]: Vec(-1.2477332586079907e8,-1.977610425796057e8)
```

```
In [14]: using Plots
         v=bitp[1.5]
         println("cons=$(v[1])")
         println("save=$(v[2])")
         plot(linspace(1,3,200),a',labels=["cons","save"]',yticks=convert(Array{Float64},v))
         vline!([1.5])
         hline!([v[1],v[2]])


         # say you have assets of 0.5:
```

```
cons=0.46209812681126966
save=0.7324082026493517
```

```
WARNING: the no-op `transpose` fallback is deprecated, and no more specific `transpose` method f
 in depwarn(::String, ::Symbol) at ./deprecated.jl:64
 in transpose(::String) at ./deprecated.jl:770
 in ctranspose at ./operators.jl:315 [inlined]
 in (::Base.##209#210)(::Tuple{Int64,String}) at ./<missing>:0
 in copy!(::Array{String,2}, ::Base.Generator{Base.Prod2{Base.OneTo{Int64},Array{String,1}},Base
 in _collect(::Type{String}, ::Base.Generator{Base.Prod2{Base.OneTo{Int64},Array{String,1}},Base
 in ctranspose(::Array{String,1}) at ./arraymath.jl:417
 in include_string(::String, ::String) at ./loading.jl:441
 in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /Users/florian.oswald/.julia/v0.5/IJulia/src/
 in eventloop(::ZMQ.Socket) at /Users/florian.oswald/.julia/v0.5/IJulia/src/eventloop.jl:8
 in (::IJulia.##13#19)() at ./task.jl:360
while loading In[14], in expression starting on line 5
```

2.13 The CompEcon Toolbox of Miranda and Fackler

- another good alternative:
- CompEcon.jl

2.14 Mulitidimensional Approximation

- Up to now, most of what we did was in one dimesion.
- Economic problems *often* have more dimension than that.
  - The number of state variables in your value functions are the number of dimensions.
- We can readily extend what we learned into more dimensions.
- However, we will quickly run into feasibility problems: hello *curse of dimensionality*.

2.15 Tensor Product of univariate Basis Functions: Product Rule

- One possibility is to approximate e.g. the 2D function $f(x, y)$ by

$$\hat{f}(x, y) = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{i,j} \phi_i^x(x) \phi_j^y(y)$$

  - here ϕ_i^x is the basis function in x space,
  - you can see that the coefficient vector $c_{i,j}$ is indexed in two dimensions now.
  - Notice that our initial notation was general enough to encompass this case, as we defined the basis functions as $\mathbb{R}^d \mapsto \mathbb{R}$. So with the product rule, this mapping is just given by $\phi_i^x(x)\phi_j^y(y)$.
- This formulation requires that we take the product of $\phi_i^x(x), \phi_j^y(y)$ at *all* combinations of their indices, as is clear from the summations.
- This is equivalent to the tensor product between ϕ_i^x and ϕ_j^y.

2.16 Computing Coefficients from Tensor Product Spaces

- Extending this into D dimensions, where in each dim i we have n_i basis functions, we get

$$\hat{f}(x_1, x_2, \ldots, x_D) = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \cdots \sum_{i_D=1}^{n_D} c_{i_1, i_2, \ldots, i_D} \phi_{i_1}(x_1) \phi_{i_2}(x_2) \ldots \phi_{i_D}(x_D)$$

- In Vector notation

$$\hat{f}(x_1, x_2, \ldots, x_D) = \left[\phi_D(x_D) \otimes \phi_{D-1}(x_{D-1}) \otimes \cdots \otimes \phi_1(x_1)\right] c$$

  where c is is an $n = \Pi_{i=1}^{D} n_i$ column vector
- The solution is the interpolation equation as before,

$$\begin{aligned} \Phi c &= y \\ \Phi &= \Phi_D \otimes \Phi_{D-1} \otimes \cdots \otimes \Phi_1 \end{aligned}$$

2.16.1 The Problem with Tensor Product of univariate Basis Functions

- What's the problem?
- Well, solving $\Phi c = y$ is hard.
- If we have as many evaluation points as basis functions in each dimension, i.e. if each single Φ_i is a square matrix, Φ is of size (n,n).
- Inverting this is *extremely* hard even for moderately sized problems.
- Sometimes it's not even possible to allocate Φ in memory.
- Here it's important to remember the sparsity structure of a spline basis function.

```
In [15]: using Plots
         using ApproXD
         bs = ApproXD.BSpline(7,3,0,1) #7 knots, degree 3 in [0,1]
         n = 500
         eval_points = collect(linspace(0,1.0,n))
         B = full(ApproXD.getBasis(eval_points,bs))
         ys = [string("Basis",i) for i = 1:8]
         xs = eval_points
         heatmap(xs,ys,B')
```

- This is a cubic spline basis. at most $k + 1 = 4$ basis are non-zero for any x.

```
In [16]: heatmap(xs,ys,B'.>0)
```

2.17 Using Sparsity of Splines

- It may be better to store the splines in sparse format.
- Look at object `B` by typing `B` and `typeof(B)`
- There are sparse system solvers available.
- Creating and storing the inverse of Φ destroys the sparsity structure (inverse of a sparse matrix is not sparse), and may not be a good idea.
- Look back at Section **??**
- We only have to sum over the non-zero entries! Every other operation is pure cost.
- This is implemented in `ApproXD.jl` for example via

```
function evalTensor2{T}(mat1::SparseMatrixCSC{T,Int64},
                        mat2::SparseMatrixCSC{T,Int64},
                        c::Vector{T})
```

2.18 High Dimensional Functions: Introducing the Smolyak Grid

- This is a modification of the Tensor product rule.
- It elemininates points from the full tensor product according to their *importance* for the quality of approximation.
- The user controls this quality parameter, thereby increasing/decreasing the size of the grid.
- [@jmmv] [2]is a complete technical reference for this method.
- [@maliar-maliar] [3]chapter 4 is very good overview of this topic, and the basis of this part of the lecture.

2.19 The Smolyak Grid in 2 Dimensions

- Approximation level $\mu \in \mathbb{N}$ governs the quality of the approximation.
- Start with a unidimensional grid of points x:

$$x = \left\{-1, \frac{-1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}, 1\right\}$$

  which are 5 Chebyshev nodes (it's not important that those are Chebyshev nodes, any grid will work).
- A 2D tensor product $x \otimes x$ gives 25 grid points

$$x \otimes x = \left\{(-1,-1), (-1, \frac{-1}{\sqrt{2}}), \ldots, (1,1)\right\}$$

- The Smolyak method proceeds differently.
- We construct three nested sets:

$$\begin{aligned} i = 1 &: S_1 = \{0\} \\ i = 2 &: S_2 = \{0, -1, 1\} \\ i = 3 &: S_3 = \left\{-1, \tfrac{-1}{\sqrt{2}}, 0, \tfrac{1}{\sqrt{2}}, 1\right\} \end{aligned}$$

- Then, we construct all possible 2D tensor products using elements from these nested sets in a table (next slide).
- Finally, we select only those elements of the table, that satisfy the Smolyak rule:

$$i_1 + i_2 \leq d + \mu$$

  where i_1, i_2 are column and row index, respectively, and d, μ are the number of dimensions and the quality of approximation.

3 The Smolyak Grid in 2D: Tensor Table

3.1 Selecting Elements

- Denote the Smolyak grid for d dimensions at level μ by $\mathcal{H}^{d,\mu}$.
- if $\mu = 0$ we have $i_1 + i_2 \leq 2$. Only one point satisfies this, and

$$\mathcal{H}^{2,0} = \{(0,0)\}$$

- if $\mu = 1$ we have $i_1 + i_2 \leq 3$. Three cases satisfy this:

  1. $i_1 = 1, i_2 = 1 \rightarrow (0,0)$
  2. $i_1 = 1, i_2 = 2 \rightarrow (0,0), (0,-1), (0,1)$
  3. $i_1 = 2, i_2 = 1 \rightarrow (0,0), (-1,0), (1,0)$

  – Therefore, the unique elements from the union of all of those is

$$\mathcal{H}^{2,1} = \{(0,0), (-1,0), (1,0), (0,-1), (0,1)\}$$

- if $\mu = 2$ we have $i_1 + i_2 \leq 4$. Six cases satisfy this:

Table 3 Tensor products of unidimensional nested grid points for the two-dimensional case.

	$S_{i_1} \backslash S_{i_2}$	$i_2 = 1$ **0**	$i_2 = 2$ **0, −1, 1**	$i_2 = 3$ **0, −1, 1, $\frac{-1}{\sqrt{2}}$, $\frac{1}{\sqrt{2}}$**
$i_1 = 1$	0	$(0,0)$	$(0,0),(0,-1),(0,1)$	$(0,0),(0,-1),(0,1),(0,\frac{-1}{\sqrt{2}}),(0,\frac{1}{\sqrt{2}})$
$i_1 = 2$	0	$(0,0)$	$(0,0),(0,-1),(0,1)$	$(0,0),(0,-1),(0,1),(0,\frac{-1}{\sqrt{2}}),(0,\frac{1}{\sqrt{2}})$
	−1	$(-1,0)$	$(-1,0),(-1,-1),(-1,1)$	$(-1,0),(-1,-1),(-1,1),(-1,\frac{-1}{\sqrt{2}}),(-1,\frac{1}{\sqrt{2}})$
	1	$(1,0)$	$(1,0),(1,-1),(1,1)$	$(1,0),(1,-1),(1,1),(1,\frac{-1}{\sqrt{2}}),(1,\frac{1}{\sqrt{2}})$
$i_1 = 3$	0	$(0,0)$	$(0,0),(0,-1),(0,1)$	$(0,0),(0,-1),(0,1),(0,\frac{-1}{\sqrt{2}}),(0,\frac{1}{\sqrt{2}})$
	−1	$(-1,0)$	$(-1,0),(-1,-1),(-1,1)$	$(-1,0),(-1,-1),(-1,1),(-1,\frac{-1}{\sqrt{2}}),(-1,\frac{1}{\sqrt{2}})$
	1	$(1,0)$	$(1,0),(1,-1),(1,1)$	$(1,0),(1,-1),(1,1),(1,\frac{-1}{\sqrt{2}}),(1,\frac{1}{\sqrt{2}})$
	$\frac{-1}{\sqrt{2}}$	$(\frac{-1}{\sqrt{2}},0)$	$(\frac{-1}{\sqrt{2}},0),(\frac{-1}{\sqrt{2}},-1),(\frac{-1}{\sqrt{2}},1)$	$(\frac{-1}{\sqrt{2}},0),(\frac{-1}{\sqrt{2}},-1),(\frac{-1}{\sqrt{2}},1),(\frac{-1}{\sqrt{2}},\frac{-1}{\sqrt{2}}),(\frac{-1}{\sqrt{2}},\frac{1}{\sqrt{2}})$
	$\frac{1}{\sqrt{2}}$	$(\frac{1}{\sqrt{2}},0)$	$(\frac{1}{\sqrt{2}},0),(\frac{1}{\sqrt{2}},-1),(\frac{1}{\sqrt{2}},1)$	$(\frac{1}{\sqrt{2}},0),(\frac{1}{\sqrt{2}},-1),(\frac{1}{\sqrt{2}},1),(\frac{1}{\sqrt{2}},\frac{-1}{\sqrt{2}}),(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}})$

[@maliar-maliar] table 3: All Tensor Products

1. $i_1 = 1, i_2 = 1$
2. $i_1 = 1, i_2 = 2$
3. $i_1 = 2, i_2 = 1$
4. $i_1 = 1, i_2 = 3$
5. $i_1 = 2, i_2 = 2$
6. $i_1 = 3, i_2 = 1$

– Therefore, the unique elements from the union of all of those is

$$\mathcal{H}^{2,2} = \left\{(-1,1),(0,1),(1,1),(-1,0),(0,0),(1,0),(-1,-1),(0,-1),(1,-1),\left(\frac{-1}{\sqrt{2}},0\right),\left(\frac{1}{\sqrt{2}},0\right),\left(0,\frac{-1}{\sqrt{2}}\right)\right.$$

- Note that those elements are on the diagonal from top left to bottom right expanding through all the tensor products on table 3.

3.2 Size of Smolyak Grids

- The Smolyak grid grows much slower (at order d to a power of μ) than the Tensor grid (exponential growth)

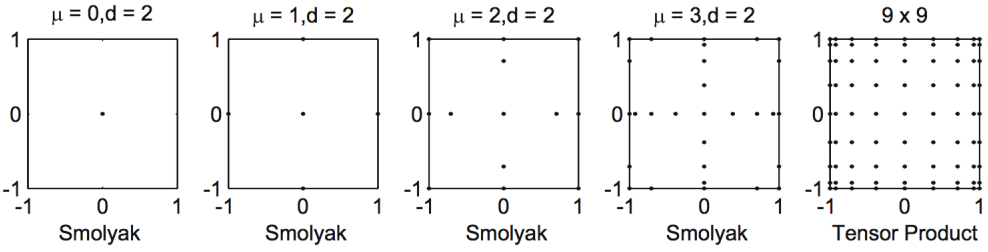

Figure 2 Smolyak grids versus tensor-product grid.

[@maliar-maliar] figure 2: Tensor vs Smolyak in 2D

Table 4 Number of grid points: tensor-product grid with five points in each dimension versus Smolyak grid.

		Smolyak grid		
d	Tensor-product grid 5^d	$\mu = 1$	$\mu = 2$	$\mu = 3$
1	5	3	5	9
2	25	5	13	29
10	9,765,625	21	221	1581
20	95,367,431,640,625	41	841	11561

Notes: d is the number of state variables; μ is the approximation level.

[@maliar-maliar] figure 4: Tensor vs Smolyak in 2D, number of grid points

3.3 Smolyak Polynomials

- Corresponding to the construction of grid points, there is the Smolyak way of constructing polynomials.
- This works exactly as before. We start with a one-dimensional set of basis functions (again Chebyshev here, again irrelevant):

$$\left\{1, x, 2x^2 - 1, 4x^3 - 3x, 8x^4 - 8x^2 + 1\right\}$$

- Three nested sets:

$$\begin{aligned} i = 1 &: S_1 = \{1\} \\ i = 2 &: S_2 = \{1, x, 2x^2 - 1\} \\ i = 3 &: S_3 = \left\{1, x, 2x^2 - 1, 4x^3 - 3x, 8x^4 - 8x^2 + 1\right\} \end{aligned}$$

- Denoting $\mathcal{P}^{d,\mu}$ the Smolyak polynomial, we follow exactly the same steps as for the grids to select elements of the full tensor product table 5:

3.4 Smolyak Interpolation

This proceeds as in the previouses cases:

1. Evaluate f at all grid points $\mathcal{H}^{d,\mu}$.
2. Evaluate the set of basis functions given by $\mathcal{P}^{d,\mu}$ at all grid points $\mathcal{H}^{d,\mu}$.
3. Solve for the interpolating coefficients by inverting the Basis function matrix.

3.5 Extensions

- There is a lot of redundancy in computing the grids the way we did it.
- More sophisticated approaches take care not to compute repeated elements.

Table 5 Tensor products of unidimensional nested polynomial basis for the two-dimensional case.

	$S_{i_1} \backslash S_{i_2}$	$i_2 = 1$ 1	$i_2 = 2$ $1, y, 2y^2 - 1$	$i_2 = 3$ $1, y, 2y^2 - 1, 4y^3 - 3y, 8y^4 - 8y^2 + 1$
$i_1 = 1$	1	1	$1, y, 2y^2 - 1$	$1, y, 2y^2 - 1, 4y^3 - 3y, 8y^4 - 8y^2 + 1$
$i_1 = 2$	1	1	$1, y, 2y^2 - 1,$	$1, y, 2y^2 - 1, 4y^3 - 3y, 8y^4 - 8y^2 + 1$
	x	x	$x, xy, x\left(2y^2 - 1\right),$	$x, xy, x\left(2y^2 - 1\right), x\left(4y^3 - 3y\right), \ldots$
	$2x^2 - 1$	$2x^2 - 1$	$2x^2 - 1, (2x^2 - 1)y, (2x^2 - 1)(2y^2 - 1)$	$2x^2 - 1, (2x^2 - 1)y, (2x^2 - 1)(2y^2 - 1), \ldots$
$i_1 = 3$	1	1	$1, y, 2y^2 - 1$	$1, y, 2y^2 - 1, 4y^3 - 3y, 8y^4 - 8y^2 + 1$
	x	x	$x, xy, x\left(2y^2 - 1\right)$	$x, xy, x\left(2y^2 - 1\right), x\left(4y^3 - 3y\right), \ldots$
	$2x^2 - 1$	$2x^2 - 1$	$2x^2 - 1, (2x^2 - 1)y, (2x^2 - 1)(2y^2 - 1)$	$\left(2x^2 - 1\right), \left(2x^2 - 1\right)y, \ldots$
	$4x^3 - 3x$	$4x^3 - 3x$	$\left(4x^3 - 3x\right), \left(4x^3 - 3x\right)y, \ldots$	$\left(4x^3 - 3x\right), \left(4x^3 - 3x\right)y, \ldots$
	$8x^4 - 8x^2 + 1$	$8x^4 - 8x^2 + 1$	$\left(8x^4 - 8x^2 + 1\right), \left(8x^4 - 8x^2 + 1\right)y, \ldots$	$\left(8x^4 - 8x^2 + 1\right), \left(8x^4 - 8x^2 + 1\right)y, \ldots$

[@maliar-maliar] figure 5: All Smolyak Polynomials in 2D

3.6 Smolyak Grids in Julia

- There are at least 2 julia packages that implement this idea:
  - https://github.com/EconForge/Smolyak (can't find the function)
  - https://github.com/alancrawford/Smolyak (seems broken)

4 Using `alancrawford/Smolyak`

From the unit tests of that package:

```
In [17]: # Pkg.clone("https://github.com/alancrawford/Smolyak.git")
         using Smolyak

         # here is the true function
         # in a real world application, evaluating this is the biggest cost
         # we would like to evaluate only a few times.

         slopes = rand(4)
         truefun(x) = 1.1 + slopes[1]*x[1] - slopes[2]*x[2]^2 + (slopes[3]*x[3])^3 * slopes[4] *

         # choose approx level in each dim
         mu = [1,2,2,1]
         D = length(mu)
         lb = -2*ones(length(mu))
         ub = 12*ones(length(mu))
         sg = SmolyakGrid(mu,lb,ub)
         sb = SmolyakBasis(sg)
         makeBasis!(sb)
         sp = SmolyakPoly(sb)

         for i in 1:sb.NumPts
             sp.Value[i] = truefun(sg.xGrid[i])          # Assign true fvals to poly
         end
```

```
        make_pinvBFt!(sp,sb)
        Smolyak.makeCoef!(sp)

        # make basis on random point
        NumObs = 50
        X = Vector{Float64}[ Float64[lb[d]+( ub[d]- lb[d])*rand() for d in 1:D] for i in 1:NumO
        sbX = SmolyakBasis(X,mu,lb,ub,0,0)
        makeBasis!(sbX)
        spX = SmolyakPoly(sbX)
        copy!(spX.Coef,sp.Coef)   #ăassign precomputed coefs
        ynew = makeValue!(spX,sbX) # Interpolated Values
        valsnew = [truefun(i) for i in X]
        scatter(ynew,valsnew,ylabel="truth",xlabel="approx")

****** MAX OF 2 DERIVATIVES ******



      MethodError: no method matching Smolyak.SmolyakBasis(::Array{Array{Float64,1},1}, ::Arra
  Closest candidates are:
    Smolyak.SmolyakBasis(::Array{T,N}, ::Union{Array{Int64,1},Int64}, ::Array{Float64,1}, ::Ar
    Smolyak.SmolyakBasis(::Array{T,N}, ::Union{Array{Int64,1},Int64}, ::Array{Float64,1}) at /
    Smolyak.SmolyakBasis(::Array{T,N}, ::Union{Array{Int64,1},Int64}) at /Users/florian.oswald
    ...
```

References

[1] Kenneth L. Judd. *Numerical methods in economics*. The MIT Press, 1998.

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

[3] Lilia Maliar and Serguei Maliar. Numerical methods for large scale dynamic economic models. *Handbook of Computational Economics*, 3:325, 2013.