Smolyak Documentation

Release 0.0.1

EconForge

April 08, 2016

Contents

Contents:

CHAPTER 1

Mathematical Background

This page is still a work in progress. I will try to provide a very basic description of each of the pieces that are necessary.

1.1 Smolyak Grid

One of the standard forms of building an approximation grid is to use a simple tensor-product. While this seems relatively trivial for low dimensions, but the number of points to evaluate quickly becomes intractable for larger dimensions. In Bellman (1961) this problem is referred to as the "curse of dimensionality." Two years after Bellman's paper, Sergey Smolyak introduced a numerical technique where the number of grid points needed to approximate grew polynomially instead of exponentially. As stated in Judd, Maliar, Maliar, Valero; the idea behind this technique was "that some elements produced by tensor-product rules are more important for representing multidimensional functions than the others." The tensor-product typically takes as parameters the dimension of the grid and the number of points, n to be evaluated at in each dimension which produces a grid with n^d points (Note: There are other ways of doing this where the number of points is different in each dimension, but the resulting number of points is similar). The Smolyak grid takes an "accuracy" parameter μ and the number of dimensions as parameters. The number of points in the grid is determined by the dimension and μ. The number of Smolyak grid points at $\mu = 1$ is $1 + 2d$, at $\mu = 2$ it is $1 + 4d + 4d(d-1)$, etc... Notice that the number of grid points grows linearly at $\mu = 1$, and quadratically at $\mu = 2$.

The standard construction of a Smolyak grid uses nested sets of points. One typically uses the extrema of the Chebyshev Polynomials, which are known as the Chebyshev-Gauss-Lobatto points. We will continue our description using these points since the code is implemented using them. The nested sets require two conditions. First, that each set S_i has 2^{i-1} points for $i \geq 2$ and 1 if $i = 1$. Secondly, that the sets are nested. The first four nested sets are:

$$
\begin{aligned}
i = 1 \ : \ S_1 &= \{0\} \\
i = 2 \ : \ S_2 &= \{-1, 0, 1\} \\
i = 3 \ : \ S_3 &= \{-1, -\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}, 1\} \\
i = 4 \ : \ S_4 &= \{-1, -\frac{\sqrt{2+\sqrt{2}}}{2}, -\frac{1}{\sqrt{2}}, -\frac{\sqrt{2-\sqrt{2}}}{2}, 0, \frac{\sqrt{2-\sqrt{2}}}{2}, \frac{1}{\sqrt{2}}, \frac{\sqrt{2+\sqrt{2}}}{2}, 1\}
\end{aligned}
$$

One then takes the tensor product of the unidimensional sets and then picks out the products that satisfy $d \leq \sum_{j=1}^{d} i_j \leq d + \mu$ where i_j is the index of the unidimensional sets. For example if the parameters were $d = 2, \mu = 2$ then you would build the first four nested sets of points (as shown above) and then take all of the tensor products that satisfied $2 \leq i_1 + i_2 \leq 4$ which would give you $S_1 \times S_1, S_1 \times S_2, S_2 \times S_1, S_2 \times S_2$. Then these would be your points. We will define the set of grid points to be $\mathcal{H}^{d,\mu} := \bigcup_{d \leq |i^*| \leq d+\mu} \prod i = 1^{d+\mu} S_i$ where $i^* = \begin{bmatrix} i_1 & \dots i_d \end{bmatrix}$.

You can see how there would be repeated points and hence this method could be improved upon. This is one of the key results of the Judd, Maliar, Maliar, Valero (2013) paper. Instead of building nested sets, they build disjoint sets A_i

such that $A_1 = \{0\}$ and $A_i = S_i \backslash S_{i-1}$ for all $i \geq 2$. Then the points are taken from the tensor-products of these sets in the same fashion as described above.

The following is a computationally efficient way of finding these grid points. The first step is to create the first $\mu + 1$ unidimensional disjoint sets A_i as described above. Then one should create a vector of possible values i.e. $\big[1, 2, \ldots, \mu + 1\big]$. It is important to note that these possible values only range from 1 to $\mu + 1$ since the smallest possible index is 1 (To see this think of given d indexes. The smallest the first $d - 1$ of them can be is 1 which would sum to $d - 1$ hence the last index could only be valued up to $\mu + 1$. We can then check all of the combinations (with replacement) of these to find the sets of indexes that would work. Once we have these we can permute them to capture all of the indexes that would work. Then you only take the tensor-products of these sets (Add reference to our code smol_inds and build_grid here).

1.2 Smolyak Basis Polynomials

Generated documentation:

CHAPTER 2

smolyak

2.1 smolyak Package

2.1.1 smolyak Package

grid Module

This file contains a class that builds a Smolyak Grid. The hope is that it will eventually contain the interpolation routines necessary so that the given some data, this class can build a grid and use the Chebychev polynomials to interpolate and approximate the data.

Method based on Judd, Maliar, Maliar, Valero 2013 (W.P)

Authors

- Chase Coleman (ccoleman@stern.nyu.edu)
- Spencer Lyon (slyon@stern.nyu.edu)

References

Judd, Kenneth L, Lilia Maliar, Serguei Maliar, and Rafael Valero. 2013. "Smolyak Method for Solving Dynamic Economic Models: Lagrange Interpolation, Anisotropic Grid and Adaptive Domain".

Krueger, Dirk, and Felix Kubler. 2004. "Computing Equilibrium in OLG Models with Stochastic Production." Journal of Economic Dynamics and Control 28 (7) (April): 1411-1436.

smolyak.grid.**num_grid_points**(*d*, *mu*)

Checks the number of grid points for a given d, mu combination.

Parameters **d, mu** : int

The parameters d and mu that specify the grid

Returns **num** : int

The number of points that would be in a grid with params d, mu

Notes

This function is only defined for mu = 1, 2, or 3

smolyak.grid.**m_i**(*i*)

Compute one plus the "total degree of the interpolating polynoimals" (Kruger & Kubler, 2004). This shows up many times in Smolyak's algorithm. It is defined as:

$$m_i = \begin{cases} 1 & \text{if } i = 1 \\ 2^{i-1} + 1 & \text{if } i \geq 2 \end{cases}$$

Parameters **i** : int

The integer i which the total degree should be evaluated

Returns **num** : int

Return the value given by the expression above

smolyak.grid.**cheby2n**(*x*, *n*, *kind=1.0*)

Computes the first $n + 1$ Chebychev polynomials of the first kind evaluated at each point in x .

Parameters **x** : float or array(float)

A single point (float) or an array of points where each polynomial should be evaluated

n : int

The integer specifying which Chebychev polynomial is the last to be computed

kind : float, optional(default=1.0)

The "kind" of Chebychev polynomial to compute. Only accepts values 1 for first kind or 2 for second kind

Returns **results** : array (float, ndim=x.ndim+1)

The results of computation. This will be an $(n + 1 \times dim \dots)$ where $(dim \dots)$ is the shape of x. Each slice along the first dimension represents a new Chebychev polynomial. This dimension has length $n + 1$ because it includes ϕ_0 which is equal to 1 $\forall x$

smolyak.grid.**s_n**(*n*)

Finds the set S_n , which is the n th Smolyak set of Chebychev extrema

Parameters **n** : int

The index n specifying which Smolyak set to compute

Returns **s_n** : array (float, ndim=1)

An array containing all the Chebychev extrema in the set S_n

smolyak.grid.**a_chain**(*n*)

Finds all of the unidimensional disjoint sets of Chebychev extrema that are used to construct the grid. It improves on past algorithms by noting that $A_n = S_n$ [evens] except for $A_1 = \{0\}$ and $A_2 = \{-1, 1\}$. Additionally, $A_n = A_{n+1}$ [odds] This prevents the calculation of these nodes repeatedly. Thus we only need to calculate biggest of the S_n's to build the sequence of A_n 's

Parameters **n** : int

This is the number of disjoint sets from Sn that this should make

Returns **A_chain** : dict (int -> list)

This is a dictionary of the disjoint sets that are made. They are indexed by the integer corresponding

smolyak.grid.**phi_chain**(*n*)

For each number in 1 to n, compute the Smolyak indices for the corresponding basis functions. This is the n in ϕ_n

Parameters **n** : int

The last Smolyak index n for which the basis polynomial indices should be found

Returns **aphi_chain** : dict (int -> list)

A dictionary whose keys are the Smolyak index n and values are lists containing all basis polynomial subscripts for that Smolyak index

smolyak.grid.**smol_inds**(*d*, *mu*)

Finds all of the indices that satisfy the requirement that $d \leq \sum_{i=1}^{d} \leq d + \mu$.

Parameters **d** : int

The number of dimensions in the grid

mu : int or array (int, ndim=1)

The parameter mu defining the density of the grid. If an array, there must be d elements and an anisotropic grid is formed

Returns **true_inds** : array

A 1-d Any array containing all d element arrays satisfying the constraint

Notes

This function is used directly by build_grid and poly_inds

smolyak.grid.**build_grid**(*d*, *mu*, *inds=None*)

Use disjoint Smolyak sets to construct Smolyak grid of degree d and density parameter mu

The return value is an $n \times d$ Array, where n is the number of points in the grid

Parameters **d** : int

The number of dimensions in the grid

mu : int

The density parameter for the grid

inds : list (list (int)), optional (default=None)

The Smolyak indices for parameters d and mu. Should be computed by calling *smol_inds(d, mu)*. If None is given, the indices are computed using this function call

Returns **grid** : array (float, ndim=2)

The Smolyak grid for the given d, mu

smolyak.grid.**build_B**(*d*, *mu*, *pts*, *b_inds=None*, *deriv=False*)

Compute the matrix B from equation 22 in JMMV 2013 Translation of dolo.numeric.interpolation.smolyak.SmolyakBasic

Parameters **d** : int

The number of dimensions on the grid

mu : int or array (int, ndim=1, legnth=d)

The mu parameter used to define grid

pts : array (float, dims=2)

Arbitrary d-dimensional points. Each row is assumed to be a new point. Often this is the smolyak grid returned by calling *build_grid(d, mu)*

b_inds : array (int, ndim=2)

The polynomial indices for parameters a given grid. These should be computed by calling *poly_inds(d, mu)*.

deriv : bool

Whether or not to compute the values needed for the derivative matrix B_prime.

Returns **B** : array (float, ndim=2)

The matrix B that represents the Smolyak polynomial corresponding to grid

B_Prime : array (float, ndim=3), optional (default=false)

This will be the 3 dimensional array representing the gradient of the Smolyak polynomial at each of the points. It is only returned when *deriv=True*

class `smolyak.grid.`**`SmolyakGrid`**(*d*, *mu*, *lb=None*, *ub=None*)

Bases: `object`

This class currently takes a dimension and a degree of polynomial and builds the Smolyak Sparse grid. We base this on the work by Judd, Maliar, Maliar, and Valero (2013).

Parameters **d** : int

The number of dimensions in the grid

mu : int or array(int, ndim=1, length=d)

The "density" parameter for the grid

Examples

```
>>> s = SmolyakGrid(3, 2)
>>> s
Smolyak Grid:
    d: 3
    mu: 2
    npoints: 25
    B: 0.65% non-zero
>>> ag = SmolyakGrid(3, [1, 2, 3])
>>> ag
Anisotropic Smolyak Grid:
    d: 3
    mu: 1 x 2 x 3
    npoints: 51
    B: 0.68% non-zero
```

Attributes

d	int	This is the dimension of grid that you are building
mu	int	mu is a parameter that defines the fineness of grid that we want to build
lb	array (float, ndim=2)	This is an array of the lower bounds for each dimension
ub	array (float, ndim=2)	This is an array of the upper bounds for each dimension
cube_grid	array (float, ndim=2)	The Smolyak sparse grid on the domain $[-1, 1]^d$
grid:	array (float, ndim=2)	The sparse grid, transformed to the user-specified bounds for the domain
inds	list (list (int))	This is a lists of lists that contains all of the indices
B	array (float, ndim=2)	This is the B matrix that is used to do lagrange interpolation
B_L	array (float, ndim=2)	Lower triangle matrix of the decomposition of B
B_U	array (float, ndim=2)	Upper triangle matrix of the decomposition of B

Methods

cube2dom(pts)	Takes a point(s) and transforms it(them) from domain [-1, 1]^d
dom2cube(pts)	Takes a point(s) and transforms it(them) into the [-1, 1]^d domain
plot_grid()	Beautifully plots the grid for the 2d and 3d cases

cube2dom(*pts*)
Takes a point(s) and transforms it(them) from domain [-1, 1]^d back into the desired domain

dom2cube(*pts*)
Takes a point(s) and transforms it(them) into the [-1, 1]^d domain

plot_grid()
Beautifully plots the grid for the 2d and 3d cases

Parameters **None** :

Returns **None** :

interp Module

This file contains the interpolation routines for the grids that are built using the grid.py file in the smolyak package... Write more doc soon.

smolyak.interp.**find_theta**(*sg*, *f_on_grid*)
Given a SmolyakGrid object and the value of the function on the points of the grid, this function will return the coefficients theta

class smolyak.interp.**SmolyakInterp**(*sg*, *f_on_grid*)
Bases: object

This class is going to take several inputs. It will need a SmolyakGrid object to be passed in and the values of the function evaluated at the grid points

Methods

interpolate(pts[, interp, deriv, deriv_th, ...])	Basic Lagrange interpolation, with optional first derivatives
update_theta(f_on_grid)	

interpolate (*pts*, *interp=True*, *deriv=False*, *deriv_th=False*, *deriv_X=False*)

Basic Lagrange interpolation, with optional first derivatives (gradient)

Parameters **pts** : array (float, ndim=2)

A 2d array of points on which to evaluate the function. Each row is assumed to be a new d-dimensional point. Therefore, pts must have the same number of columns as `si.SGrid.d`

interp : bool, optional(default=false)

Whether or not to compute the actual interpolation values at pts

deriv : bool, optional(default=false)

Whether or not to compute the gradient of the function at each of the points. This will have the same dimensions as pts, where each column represents the partial derivative with respect to a new dimension.

deriv_th : bool, optional(default=false)

Whether or not to compute the ???? derivative with respect to the Smolyak polynomial coefficients (maybe?)

deriv_X : bool, optional(default=false)

Whether or not to compute the ???? derivative with respect to grid points

Returns **rets** : list (array(float))

A list of arrays containing the objects asked for. There are 4 possible objects that can be computed in this function. They will, if they are called for, always be in the following order:

1. Interpolation values at pts
2. Gradient at pts
3. ???? at pts
4. ???? at pts

If the user only asks for one of these objects, it is returned directly as an array and not in a list.

Notes

This is a stripped down port of `dolo.SmolyakBasic.interpolate`

TODO: There may be a better way to do this

TODO: finish the docstring for the 2nd and 3rd type of derivatives

update_theta (*f_on_grid*)

util Module

smolyak.util.**permute**(*a*)

Creates all unique combinations of a list a that is passed in. Function is based off of a function written by John Lettman: TCHS Computer Information Systems. My thanks to him.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

S

S