SpectralToolbox Documentation

Release 0.1a

Daniele Bigoni

December 16, 2014

CONTENTS

Contents:

Created on Thu Mar 29 11:33:32 2012

@author: Daniele Bigoni (dabi@imm.dtu.dk)

CHAPTER

ONE

DESCRIPTION

Implementation of Spectral Methods in 1 dimension.

Available polynomials:

- *Jacobi Polynomials* or `Spectral1D.JACOBI`
- Hermite Physicist or `Spectral1D.HERMITEP`
- Hermite Function or `Spectral1D.HERMITEF`
- Hermite Probabilistic or `Spectral1D.HERMITEP_PROB`
- Laguerre Polynomial or `Spectral1D.LAGUERREP`
- Laguerre Function or `Spectral1D.LAGUERREF`
- ORTHPOL package (generation of recursion coefficients using [1]) or `Spectral1D.ORTHPOL`

Available quadrature rules (related to selected polynomials):

- Gauss or `Spectral1D.GAUSS`
- Gauss-Lobatto or `Spectral1D.GAUSSLOBATTO`
- Gauss-Radau or `Spectral1D.GAUSSRADAU`

Available quadrature rules (without polynomial selection):

- Kronrod-Patterson on the real line or `Spectral1D.KPN` (function `Spectral1D.kpn(n)`)
- Kronrod-Patterson uniform or `Spectral1D.KPU` (function `Spectral1D.kpu(n)`)
- Clenshaw-Curtis or `Spectral1D.CC` (function `Spectral1D.cc(n)`)
- Fejer's or `Spectral1D.FEJ` (function `Spectral1D.fej(n)`)

1.1 Jacobi Polynomials

Jacobi polynomials are defined on the domain $\Omega = [-1, 1]$ by the recurrence relation

$$xP_n^{(\alpha,\beta)}(x) = \frac{2(n+1)(n+\alpha+\beta+1)}{(2n+\alpha+\beta+1)(2n+\alpha+\beta+2)}P_{n+1}^{(\alpha,\beta)}(x)$$
$$+\frac{\beta^2-\alpha^2}{(2n+\alpha+\beta)(2n+\alpha+\beta+2)}P_n^{(\alpha,\beta)}(x) + \frac{2(n+\alpha)(n+\beta)}{(2n+\alpha+\beta)(2n+\alpha+\beta+1)}P_{n-1}^{(\alpha,\beta)}(x)$$

[1] 23. Gautschi, "Algorithm 726: ORTHPOL – a package of routines for generating orthogonal polynomials and Gauss-type quadrature rules". ACM Trans. Math. Softw., vol. 20, issue 1, pp. 21-62, 1994

with weight function

$$w(x;\alpha,\beta) = \frac{\Gamma(\alpha+\beta+2)}{2^{\alpha+\beta+1}\Gamma(\alpha+1)\Gamma(\beta+1)}(1-x)^{\alpha}(1+x)^{\beta}$$

Note: In probability theory, the Beta distribution is defined on $\Psi = [0,1]$ and its the Probability Distribution Function is

$$\rho_B(x;\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{\alpha-1}(1-x)**(\beta-1)$$

The relation betwen $w(x;\alpha,\beta)$ and $\rho_B(x;\alpha,\beta)$ for $x \in \Psi$ is

$$\rho_B(x;\alpha,\beta) = 2 * w(2*x-1;\beta-1,\alpha-1)$$

For example:

```
>>> from scipy import stats
>>> plot(xp,stats.beta(3,5).pdf(xp))
>>> plot(xp,2.*Bx(xx,4,2),'--')
>>> plot(xp,stats.beta(3,8).pdf(xp))
>>> plot(xp,2.*Bx(xx,7,2),'--')
```

CHAPTER

TWO

REFERENCES

CHAPTER

THREE

EXAMPLES

class SpectralToolbox.Spectral1D.**Poly1D**(*poly*, *params*, *sdout=<open file '<stderr>', mode 'w' at 0x2b18a359b270>*)

Initialization of the Polynomial instance.

Syntax: `p = Poly1D(poly,params)`

Input:

- `poly` = The orthogonal polynomial type desired
- `params` = The parameters needed by the selected polynomial
- `sdout` = (optional,default=sys.stderr) output stream for logging

Description: This method generates an instance of the Poly1D class, to be used in order to generate orthogonal basis of the polynomial type selected. Avaliable polynomial types can be selected using their string name or by predefined attributes

- 'Jacobi' or `Spectral1D.JACOBI`
- 'HermiteP' or `Spectral1D.HERMITEP`
- 'HermiteF' or `Spectral1D.HERMITEF`
- 'HermitePprob' or `Spectral1D.HERMITEP_PROB`
- 'LaguerreP' or `Spectral1D.LAGUERREP`
- 'LaguerreF' or `Spectral1D.LAGUERREF`
- 'ORTHPOL' or `Spectral1D.ORTHPOL`

Additional parameters are required for some polynomials.

Polynomial	Parameters
Jacobi	(alpha,beta)
HermiteP	None
HermiteF	None
HermitePprob	None
LaguerreP	alpha
LaguerreF	alpha
ORTHPOL	see notes

Note: The ORTHPOL polynomials are built up using the "Multiple-Component Discretization Procedure" described in [4]. The following parameters describing the measure function are required in order to use the procedure for finding the recursion coefficients (alpha,beta) and have to be provided at construction time:

- `ncapm`: (int) maximum integer N0 (default = 500)
- `mc`: (int) number of component intervals in the continuous part of the spectrum

- `mp`: (int) number of points in the discrete part of the spectrum. If the measure has no discrete part, set mp=0
- `xp`, `yp`: (Numpy 1d-array) of dimension mp, containing the abscissas and the jumps of the point spectrum
- `mu`: (function) measure function that returns the mass (float) with arguments: `x` (float) absissa, `i` (int) interval number in the continuous part
- `irout`: (int) selects the routine for generating the recursion coefficients from the discrete inner product; `irout=1` selects the routine `sti`, `irout!=1` selects the routine `lancz`
- `finl`, `finr`: (bool) specify whether the extreme left/right interval is finite (false for infinite)
- `endl`, `endr`: (Numpy 1d-array) of dimension `mc` containing the left and right endpoints of the component intervals. If the first of these extends to -infinity, endl[0] is not being used by the routine.

Parameters `iq`, `quad`, `idelta` in [4] are suppressed. Instead the routine `qgp` of ORTHPOL [4] is used by default (`iq=0` and `idelta=2`)

`AssemblyDerivativeMatrix`(*x*, *N*, *k*)

AssemblyDerivativeMatrix(): Assemble the k-th derivative matrix using polynomials of order N.

Syntax: `Dk = AssemblyDerivativeMatrix(x,N,k)`

Input:

- x = (1d-array,float) Set of points on which to evaluate the polynomials
- N = (int) maximum order in the vanermonde matrix
- k = (int) derivative order

Output:

- Dk = Derivative matrix

Description: This function performs `D = linalg.solve(V.T, Vx.T)` where `V` and `Vx` are a Generalized Vandermonde Matrix and its derivative respectively.

Notes: For Chebyshev Polynomial, this function refers to the recursion form implemented in `PolynomialDerivativeMatrix`

`ChebyshevDerivativeCoefficients`(*fhat*)

ChebyshevDerivativeCoefficients(): computes the Chebyshev coefficients of the derivative of a function

Syntax: `dfhat = ChebyshevDerivativeCoefficients(fhat)`

Input:

- `fhat` = (1d-array,float) list of Chebyshev coefficients of the original function

Output:

- `dfhat` = (1d-array,float) list of Chebyshev coefficients of the derivative of the original function

Notes: Algorithm (5) from [1]

`DiscretePolynomialTransform`(*r*, *f*, *N*)

DiscretePolynomialTransform(): computes the Discrete Polynomial Transform of function values f

Syntax: `fhat = DiscretePolynomialTransform(r, f, N)`

Input:

- `r` = (1d-array,float) set of points on which to the polynomials are evaluated

[1] "Implemenenting Spectral Methods for Partial Differential Equations" by David A. Kopriva, Springer, 2009

- `f` = (1d-array,float) function values
- `N` = (int) maximum order in the generalized vanermonde matrix

Output: one of the two following output is given, depending on the length of r

- `(fhat, residues, rank,s)` = list of Polynomial coefficients and additional ouputs from function `numpy.linalg.lstsq` (if len(r) > N+1)

Description: If the Chebyshev polynomials are chosen and `r` contains Chebyshev-Gauss-Lobatto points, the Fast Chebyshev Transform is used. Otherwise uses the Generalized Vandermonde Matrix in order to transform from physical space to transform space.

See also:

FastChebyshevTransform

`FastChebyshevTransform`(*f*)

FastChebyshevTransform(): Returns the coefficients of the Fast Chebyshev Transform.

Syntax: `fhat = FastChebyshevTransform(f)`

Input:

- `f` = (1d-array,float) function values

Output:

- `fhat` = (1d-array,float) list of Polynomial coefficients

Warning: It is assumed that the values f are computed at Chebyshev-Gauss-Lobatto points.

Note: If f is odd, the vector is interpolated to even Chebyshev-Gauss-Lobatto points.

Note: Modification of algorithm (29) from [1]

`Gamma`(*N*)

Gamma(): returns the normalization constant for the N-th polynomial

Syntax: `g = Gamma(N)`

Input:

- `N` = polynomial order

Output:

- `g` = normalization constant

`GaussLobattoQuadrature`(*N*, *normed=False*, *left=None*, *right=None*)

GaussLobattoQuadrature(): Generates list of nodes for the Gauss-Lobatto quadrature rule using selected Polynomial basis

Syntax: `x = GaussLobattoQuadrature(N,[normed=False],[left=None],[right=None])`

Input:

- `N` = (int) accuracy level required
- `normed` = (optional,bool) whether the weights will be normalized or not
- `left` = (optional,float) containing the left endpoint (used by ORTHPOL)
- `right` = (optional,float) containing the right endpoint (used by ORTHPOL)

Output:

- `x` = (1d-array,float) containing the nodes
- `w` = (1d-array,float) containing the weights

Note: Available only for Jacobi Polynomials and ORTHPOL

`GaussQuadrature` (*N*, *normed=False*)

GaussQuadrature(): Generates list of nodes and weights for the Gauss quadrature rule using the selected Polynomial basis

Syntax: `(x,w) = GaussQuadrature(N,[normed=False])`

Input:

- `N` = (int) accuracy level required
- `normed` = (optional,bool) whether the weights will be normalized or not

Output:

- `x` = (1d-array,float) containing the nodes
- `w` = (1d-array,float) containing the weights

`GaussRadauQuadrature` (*N*, *normed=False*, *end=None*)

GaussRadauQuadrature(): Generates list of nodes for the Gauss-Radau quadrature rule using selected Polynomial basis

Syntax: ‘‘x = GaussRadauQuadrature(N,[normed=False],[end=None])’‘

Input:

- ‘‘N” = (int) accuracy level required
- `normed` = (optional,bool) whether the weights will be normalized or not
- `end` = (optional,float) containing the endpoint (used by ORTHPOL)

Output:

- ‘‘x” = (1d-array,float) containing the nodes
- ‘‘w” = (1d-array,float) weights

Note: Available only for Laguerre Polynomials/Functions and ORTHPOL

`GradEvaluate` (*r*, *N*, *k*, *norm=True*)

GradEvaluate(): evaluate the `k`-th derivative of the `N`-th order polynomial at points `r`

Syntax: `P = GradEvaluate(r,N,k[,norm=True])`

Input:

- `r` = (1d-array,float) set of points on which to evaluate the polynomial
- `N` = (int) order of the polynomial
- `k` = (int) order of the derivative
- `norm` = (bool) whether to return normalized (True) or non normalized (False) polynomials

Output:

- `P` = Polynomial evaluated on `r`

GradVandermonde1D(*r*, *N*, *k*, *norm=True*)

GradVandermonde1D(): Initialize the `k`-th gradient of the modal basis `N` at `r`

Syntax: `V = GradVandermonde1D(r,N,k,[norm])`

Input:

- `r` = (1d-array,float) set of `M` points on which to evaluate the polynomials
- `N` = (int) maximum order in the vanermonde matrix
- `k` = (int) derivative order
- `norm` = (optional,boolean) True -> orthonormal polynomials, False -> non orthonormal polynomials

Output:

- `V` = (2d-array(`MxN`),float) Generalized Vandermonde matrix

GramSchmidt(*p*, *N*, *w*)

GramSchmidt(): creates a Generalized Vandermonde Matrix of orthonormal polynomials with respect to the weights `w`

Syntax: `V = GramSchmidt(p, N, w)`

Input:

- `p` = (1d-array,float) points at which to evaluate the new polynomials
- `N` = (int) the maximum order of the polynomials
- `w` = (1d-array,float) weights to be used for the orthogonoalization

Output:

- `V` = Generalized Vandermonde Matrix containing the new orthogonalized polynomials

Description: Takes the points where the polynomials have to be evaluated and computes a Generalized Gram Schmidth procedure, where a weighted projection is used. If `w==1` then the usual inner product is used for the orthogonal projection.

InverseDiscretePolynomialTransform(*r*, *fhat*, *N*)

InverseDiscretePolynomialTransform(): computes the nodal values from the modal form fhat.

Syntax: `f = InverseDiscretePolynomialTransform(r, fhat, alpha, beta, N)`

Input:

- `x` = (1d-array,float) set of points on which to the polynomials are evaluated
- `fhat` = (1d-array,float) list of Polynomial coefficients
- `N` = (int) maximum order in the generalized vanermonde matrix

Output:

- `f` = (1d-array,float) function values

Description: If the Chebyshev polynomials are chosen and r contains Chebyshev-Gauss-Lobatto points, the Inverse Fast Chebyshev Transform is used. Otherwise uses the Generalized Vandermonde Matrix in order to transform from transform space to physical space.

See also:

InverseFastChebyshevTransform

InverseFastChebyshevTransform(*fhat*)
InverseFastChebyshevTransform(): Returns the coefficients of the Inverse Fast Chebyshev Transform.

Syntax: `f = InverseFastChebyshevTransform(fhat)`

Input:

- `fhat` = (1d-array,float) list of Polynomial coefficients

Output:

- `f` = (1d-array,float) function values

Note: If f is odd, the vector is padded with a zero value (highest freq.)

Note: Modification of algorithm (29) from [1]

LegendreDerivativeCoefficients(*fhat*)
LegendreDerivativeCoefficients(): computes the Legendre coefficients of the derivative of a function

Syntax: `dfhat = LegendreDerivativeCoefficients(fhat)`

Input:

- `fhat` = (1d-array,float) list of Legendre coefficients of the original function

Output:

- `dfhat` = (1d-array,float) list of Legendre coefficients of the derivative of the original function

Notes: Algorithm (4) from [1]

PolyInterp(*x*, *f*, *xi*, *order*)
PolyInterp(): Interpolate function values `f` from points `x` to points `xi` using Forward and Backward Polynomial Transform

Syntax: `fi = PolyInterp(x, f, xi)`

Input:

- `x` = (1d-array,float) set of `N` original points where `f` is evaluated
- `f` = (1d-array,float) set of `N` function values
- `xi` = (1d-array,float) set of `M` points where the function is interpolated
- `order` = (integer) order of polynomial interpolation

Output:

- `fi` = (1d-array,float) set of `M` function values

Notes:

Quadrature(*N*, *quadType=None*, *normed=False*, *left=None*, *right=None*, *end=None*)
Quadrature(): Generates list of nodes and weights for the `quadType` quadrature rule using the selected Polynomial basis

Syntax: `(x,w) = Quadrature(N, [quadType=None], [normed=False], [left=None], [right=None], [end=None])`

Input:

- `N` = (int) accuracy level required
- `quadType` = (`AVAIL_POLYS`) type of quadrature to be used. Default is Gauss quadrature rule.

- `normed` = (optional,bool) whether the weights will be normalized or not
- `left` = (optional,float) containing the left endpoint (used by ORTHPOL Gauss-Lobatto rules)
- `right` = (optional,float) containing the right endpoint (used by ORTHPOL Gauss-Lobatto rules)
- `end` = (optional,float) containing the endpoint (used by ORTHPOL Gauss-Radau rules)

Output:

- `x` = (1d-array,float) containing the nodes
- `w` = (1d-array,float) containing the weights

CHAPTER

FOUR

SPECTRAL ND

Created on Mon Jul 9 11:35:12 2012

@author: Daniele Bigoni (dabi@imm.dtu.dk)

Implementation of Spectral Methods in n dimension.

It uses the package `Spectral1D` as basic polynomials in order to construct higher dimensional rules by tensor product.

class `SpectralToolbox.SpectralND.`**`PolyND`**(*polys*)

Initialization of the N-dimensional Polynomial instance

Syntax: `p = PolyND(polys)`

Input:

- `polys` = (list,Spectral1D.Poly1D) list of polynomial instances of the class `Spectral1D.Poly1D`

See also:

Spectral1D.Poly1D

`GaussLobattoQuadrature`(*Ns*, *normed=False*, *warnings=True*)

GaussLobattoQuadrature(): computes the tensor product of the Guass Lobatto Points and weights

Syntax: `(x,w) = GaussLobattoQuadrature(Ns,[normed=False],[warnings=True])`

Input:

- `Ns` = (list,int) n-dimensional list with the order of approximation of each polynomial
- `normed` = (optional,boolean) whether the weights will be normalized or not
- `warnings` = (optional,boolean) set whether to ask for confirmation when it is required to allocate more then 100Mb of memory

Output:

- `x` = tensor product of the collocation points
- `w` = tensor product of the weights

> **Warning:** The lengths of `Ns` has to be conform to the number of polynomials with which you have instantiated `PolyND`

`GaussQuadrature`(*Ns*, *normed=False*, *warnings=True*)

GaussQuadrature(): computes the tensor product of the Guass Points and weights

Syntax: `(x,w) = GaussQuadrature(Ns, [normed=False],[warnings=True])`

Input:

- `Ns` = (list,int) n-dimensional list with the order of approximation of each polynomial
- `normed` = (optional,boolean) whether the weights will be normalized or not
- `warnings` = (optional,boolean) set whether to ask for confirmation when it is required to allocate more then 100Mb of memory

Output:

- `x` = tensor product of the collocation points
- `w` = tensor product of the weights

> **Warning:** The lengths of `Ns` has to be conform to the number of polynomials with which you have instantiated `PolyND`

GradVandermonde(*rs*, *Ns*, *ks*, *norms=None*, *usekron=True*, *output=True*, *warnings=True*)

GradVandermonde(): initialize the tensor product of the k-th gradient of the modal basis.

Syntax: `V = GradVandermonde(r,N,k,[norms=None],[usekron=True],[output=True],[warning`

Input:

- `rs` = (list of 1d-array,float) n-dimensional list of set of points on which to evaluate the polynomials (by default they are not the kron product of the points. See `usekron` option)
- `Ns` = (list,int) n-dimensional list with the maximum orders of approximation of each polynomial
- `ks` = (list,int) n-dimensional list with derivative orders
- `norms` = (default=None,list,boolean) n-dimensional list of boolean, True -> orthonormal, False -> orthogonal, None -> all orthonormal
- `usekron` = (optional,boolean) set whether to apply the kron product of the single dimensional Vandermonde matrices or to multiply column-wise. kron(rs) and usekron==False is equal to rs and usekron==True
- `output` = (optional,boolean) set whether to print out information about memory allocation
- `warnings` = (optional,boolean) set whether to ask for confirmation when it is required to allocate more then 100Mb of memory

OUTPUT:

- `V` = Tensor product of the Generalized Vandermonde matrices

> **Warning:** The lengths of `Ns` , `rs` , `ks` , `norms` has to be conform to the number of polynomials with which you have instantiated `PolyND`

GradVandermondePascalSimplex(*rs*, *N*, *ks*, *norms=None*, *usekron=True*, *output=True*, *warnings=True*)

GradVandermondePascalSimplex(): initialize k-th gradient of the modal basis up to the total order N

Syntax: `V = GradVandermonde(r,N,k,[norms=None],[output=True],[warnings=True])`

Input:

- `rs` = (list of 1d-array,float) n-dimensional list of set of points on which to evaluate the polynomials (by default they are not the kron product of the points. See `usekron` option)
- `N` = (int) the maximum orders of the polynomial basis
- `ks` = (list,int) n-dimensional list with derivative orders

- `norms` = (default=None,list,boolean) n-dimensional list of boolean, True -> orthonormal, False -> orthogonal, None -> all orthonormal
- `usekron` = (optional,boolean) set whether to apply the kron product of the single dimensional Vandermonde matrices or to multiply column-wise. kron(rs) and usekron==False is equal to rs and usekron==True
- `output` = (optional,boolean) set whether to print out information about memory allocation
- `warnings` = (optional,boolean) set whether to ask for confirmation when it is required to allocate more then 100Mb of memory

OUTPUT:

- `V` = Generalized Vandermonde matrix up to the N-th order

> **Warning:** The lengths of `rs` , `ks` , `norms` has to be conform to the number of polynomials with which you have instantiated `PolyND`

`Quadrature`(*Ns*, *quadTypes=None*, *left=None*, *right=None*, *end=None*, *normed=False*, *warnings=True*)

GaussQuadrature(): computes the tensor product of the Guass Points and weights

Syntax: `(x,w) = GaussQuadrature(Ns, [quadTypes=None], [normed=False],[warnings=True])`

Input:

- `Ns` = (list,int) n-dimensional list with the order of approximation of each polynomial
- `quadTypes` = (list,''Spectral1D.AVAIL_QUADPOINTS'') n-dimensional list of quadrature point types chosen among Gauss, Gauss-Radau, Gauss-Lobatto (using the definition in `Spectral1D`). If None, Gauss points will be generated by default
- `left`: (list,float) list of left values used by ORTHPOL for Gauss-Lobatto rules (the dimensions where the value is not used can be set to anything)
- `right`: (list,float) list of left values used by ORTHPOL for Gauss-Lobatto rules (the dimensions where the value is not used can be set to anything)
- `end`: (list,float) list of left values used by ORTHPOL for Gauss-Radau rules (the dimensions where the value is not used can be set to anything)
- `normed` = (optional,boolean) whether the weights will be normalized or not
- `warnings` = (optional,boolean) set whether to ask for confirmation when it is required to allocate more then 100Mb of memory

Output:

- `x` = tensor product of the collocation points
- `w` = tensor product of the weights

> **Warning:** The lengths of `Ns` has to be conform to the number of polynomials with which you have instantiated `PolyND`

CHAPTER

FIVE

SPARSE GRIDS

Created on Mon Jul 9 14:08:30 2012

@author: Daniele Bigoni (dabi@imm.dtu.dk)

Implementation of Sparse Grid (Smolyak) methods for numerical integration.

This implementation is a porting of the Sparse-Grid package developped in MatLab(R) at http://www.sparse-grids.de
The code is extended with Clenshaw-Curtis and Fejer quadrature rules [1]

class `SpectralToolbox.SparseGrids.`**`SparseGrid`**(*qrule*, *dim*, *k*, *sym*)

Initialization of the Sparse Grid instance.

Syntax: sg = SparseGrid(qrule, dim, k, sym)

Input:

- qrule = Function of 1D integration rule
- dim = dimension of the integration problem
- k = Accuracy level. The rule will be exact for polynomial up to total order 2k-1
- sym = (optional) only used for own 1D quadrature rule (type not "KPU",...). If sym is supplied and not=0, the code will run faster but will produce incorrect results if 1D quadrature rule is asymmetric.

Description: Several 1D integration rules are available to be chosen for the `qrule` input parameter

- KPU = Nested rule for unweighted integral over [0,1]
- KPN = Nested rule for integral with Gaussian weight
- GQU = Gaussian quadrature for unweighted integral over [0,1] (Gauss-Legendre)
- GQN = Gaussian quadrature for integral with Gaussian weight (Gauss-Hermite)
- CC = Clenshaw-Curtis quadrature for unweighted integral over [-1,1]
- FEJ = Fejer's quadrature for unweighted integral over [-1,1]
- func = any function provided by the user that accept level l and returns nodes n and weights w for univariate quadrature rule with polynomial exactness 2l-1 as [n w] = feval(func,level)

`sparseGrid`()

sparseGrid(): main function for generating nodes & weights for sparse grids intergration

Syntax: (n,w) = sparseGrid()

Output:

- n = matrix of nodes with dim columns
- w = row vector of corresponding weights

SpectralToolbox.SparseGrids.**GQU**(*): function for generating 1D Gaussian quadrature rules for unweighted integral over [0, 1] (Gauss-Legendre)*

Syntax: (n,w) = GQU(l)

Input: l = level of accuracy of the quadrature rule

Output: n = nodes w = weights

SpectralToolbox.SparseGrids.**GQN**(*): function for generating 1D Gaussian quadrature for integral with Gaussian weight (Gauss-Hermite)*

Syntax: (n,w) = GQU(l)

Input: l = level of accuracy of the quadrature rule

Output: n = nodes w = weights

SpectralToolbox.SparseGrids.**KPU**(*l*)

KPU(): function for generating 1D Nested rule for unweighted integral over [0,1] Syntax:

(n,w) = GQU(l)

Input: l = level of accuracy of the quadrature rule

Output: n = nodes w = weights

SpectralToolbox.SparseGrids.**KPN**(*l*)

KPN(): function for generating 1D Nested rule for integral with Gaussian weight Syntax:

(n,w) = GQU(l)

Input: l = level of accuracy of the quadrature rule

Output: n = nodes w = weights

CHAPTER

SIX

MISCELLANEOUS

Created on Wed Feb 27

@author: Daniele Bigoni (dabi@dtu.dk)

Collection of miscellaneous functions used in the SpectralToolbox

class SpectralToolbox.Misc.**ExpandingArray**(*initData*, *allocInitDim=None*, *dtype=<type 'numpy.float64'>*, *maxIncrement=None*)

ExpandingArray is used for the dynamic allocation of memory in applications where the total allocated memory needed cannot be predicted. Memory is preallocated with increases of 50% all the time data exceed the allocated memory.

Initialization of the Expanding Array.

```
>>> EA = ExpandingArray(initData,[allocInitDim=None,[dtype=np.float64,[maxIncrement=None]]])
```

Parameters

- **initData** (*ndarray*) – InitialData with which to be initially filled. This must provide the number of dimensions of the array
- **allocInitDim** (*1darray-integer*) – Initial allocated dimension (optional)
- **dtype** (*dtype*) – type for the data that will be contained in EA (optional,default=np.float64)
- **maxIncrement** (*integer*) – upper limit for the allocation increment

concatenate(*X*, *axis=0*)

Concatenate data to the existing array. If needed the array is resized in the axis direction by a factor of 50%.

Parameters

- **X** (*ndarray*) – data to be concatenated to the array. Note that X.shape[i]==EA.shape()[i] is required for all i!=axis
- **axis** (*integer*) – axis along which to concatenate the additional data (optional)

```
>>> import numpy as np
>>> EA = Misc.ExpandingArray(np.random.rand(25,4))
>>> EA.shape()
(25, 4)
>>> EA.getAllocArray().shape
(25, 4)
>>> EA.concatenate(np.random.rand(13,4))
>>> EA.shape()
(38, 4)
```

getAllocArray()
Return the allocated array.

Returns allocated array

Return type ndarray

```
>>> import numpy as np
>>> EA = Misc.ExpandingArray(np.random.rand(25,4))
>>> EA.shape()
(25, 4)
>>> EA.getAllocArray().shape
(25, 4)
>>> EA.concatenate(np.random.rand(13,4))
>>> EA.shape()
(38, 4)
>>> EA.getAllocArray().shape
(45, 4)
```

getDataArray()
Get the view of the array with data.

Returns allocated array

Return type ndarray

```
>>> EA.shape()
(38, 4)
>>> EA.getAllocArray().shape
(45, 4)
>>> EA.getDataArray().shape
(38, 4)
```

shape()
Returns the shape of the data inside the array. Note that the allocated memory is always bigger or equal to `shape()`.

Returns shape of the data

Return type tuple of integer

```
>>> import numpy as np
>>> EA = Misc.ExpandingArray(np.random.rand(25,4))
>>> EA.shape()
(25, 4)
>>> EA.getAllocArray().shape
(25, 4)
>>> EA.concatenate(np.random.rand(13,4))
>>> EA.shape()
(38, 4)
>>> EA.getAllocArray().shape
(45, 4)
```

trim(*N*, *axis=0*)
Trim the axis dimension of N elements. The allocated data is not reinitialized or deallocated. Only the dimensions of the view are redefined.

Parameters

- **N** (*integer*) – number of elements to be removed along the `axis` dimension
- **axis** (*integer*) – axis along which to remove elements (optional)

```
>>> EA = Misc.ExpandingArray(np.random.rand(4,2))
>>> EA.getDataArray()
array([[ 0.42129746,  0.76220921],
       [ 0.9238783 ,  0.11256142],
       [ 0.42031437,  0.87349243],
       [ 0.83187297,  0.555708  ]])
>>> EA.trim(2,axis=0)
>>> EA.getDataArray()
array([[ 0.42129746,  0.76220921],
       [ 0.9238783 ,  0.11256142]])
>>> EA.getAllocArray()
array([[ 0.42129746,  0.76220921],
       [ 0.9238783 ,  0.11256142],
       [ 0.42031437,  0.87349243],
       [ 0.83187297,  0.555708  ]])
```

SpectralToolbox.Misc.**MultiIndex**(*d*, *N*)

Generates the multi index ordering for the construction of multidimensional Generalized Vandermonde matrices

Parameters

- **d** (*integer*) – dimension of the simplex
- **N** (*integer*) – maximum value of the sum of the indices

Returns array containing the ordered multi-indices

Return type 2d-array of integer

```
>>> Misc.MultiIndex(2,3)
array([[0, 0],
       [1, 0],
       [0, 1],
       [2, 0],
       [1, 1],
       [0, 2],
       [3, 0],
       [2, 1],
       [1, 2],
       [0, 3]])
```

SpectralToolbox.Misc.**almostEqual**(*x*, *y*, *tol*)

Check equality of two arrays objects up to a certain tolerance

Parameters

- **x,y** (*numpy.ndarray objects of floats*) – values to be compared
- **tol** (*float*) – tolerance to be used

Returns `true` if equal, `false` otherwise

Return type bool

```
>>> eps2 = 2.*Misc.machineEpsilon(np.float64)
>>> Misc.almostEqual(np.array([2.]),np.array([2.+0.5*eps2]),eps2)
True
>>> Misc.almostEqual(np.array([2.]),np.array([2.+2.*eps2]),eps2)
False
```

SpectralToolbox.Misc.**almostEqualList**(*xArray*, *y*, *tol*)

Check equality of a list of floats against an iterable value up to certain tolerance

Parameters

- **xArray** (*2d-array of floats*) – values to be compared to `y`
- **y** (*iterable objects of floats*) – values to be compared to
- **tol** (*float*) – tolerance to be used

Returns array of booleans containing true where equal, false elsewhere.

Return type 1d-array of bool

Syntax: `b = almostEqualList(xArray,y,tol)`

```
>>> eps2 = 2.*Misc.machineEpsilon(np.float64)
>>> X = np.random.rand(4,2)
>>> Misc.almostEqualList(X,X[1,:],eps2)
array([False,  True, False, False], dtype=bool)
```

`SpectralToolbox.Misc.`**`argsort_insertion`**(*X*, *tol*, *start_idx=1*, *end_idx=None*)

Implements the insertion sort with `binary_search`. Returns permutation indices.

Parameters

- **X** (*2d-array of floats*) – values ordered by row according to the `compare` function
- **tol** (*float*) – tolerance to be used
- **start_idx,end_idx** (*int*) – starting and ending indices for the ordering (optional)

Returns permutation indices

Return type 1d-array of integers

```
>>> X = np.random.rand(5,2)
>>> X
array([[ 0.56865133,  0.18490129],
       [ 0.01411459,  0.46076606],
       [ 0.64384365,  0.24998971],
       [ 0.47840414,  0.32554137],
       [ 0.12961966,  0.43712056]])
>>> perm = Misc.argsort_insertion(X,eps2)
>>> X[perm,:]
array([[ 0.01411459,  0.46076606],
       [ 0.12961966,  0.43712056],
       [ 0.47840414,  0.32554137],
       [ 0.56865133,  0.18490129],
       [ 0.64384365,  0.24998971]])
```

`SpectralToolbox.Misc.`**`binary_search`**(*X*, *val*, *lo*, *hi*, *tol*, *perm=None*)

Search for the minimum X bigger than val

Parameters

- **X** (*2d-array of floats*) – values ordered by row according to the `compare` function
- **val** (*1d-array of floats*) – value to be compared to
- **lo,hi** (*integer*) – staring and ending indices
- **tol** (*float*) – tolerance to be used
- **perm** (*1d-array of integers*) – possible permutation to be used prior to the search (optional)

Returns index pointing to the maximum X smaller than val. If `perm` is provided, `perm[idx]` points to the maximum X smaller than val

Return type integer

```
>>> X = np.arange(1,5).reshape((4,1))
>>> X
array([[1],
       [2],
       [3],
       [4]])
>>> Misc.binary_search(X,np.array([2.5]),0,4,eps2)
>>> idx = Misc.binary_search(X,np.array([2.5]),0,4,eps2)
>>> idx
2
>>> X[idx,:]
array([3])
```

`SpectralToolbox.Misc.`**`compare`**(*x*, *y*, *tol*)

Compares two iterable objects up to a certain tolerance

Parameters

- **x,y** (*iterable objects of floats*) – values to be compared
- **tol** (*float*) – tolerance to be used

Returns -1 if `(x-y) < tol`, 1 if `(x-y) > tol`, 0 otherwise

Return type integer

```
>>> eps2 = 2.*Misc.machineEpsilon(np.float64)
>>> Misc.compare(np.array([2.]),np.array([2.+0.5*eps2]),eps2)
0
>>> Misc.compare(np.array([2.]),np.array([2.+2.*eps2]),eps2)
-1
```

`SpectralToolbox.Misc.`**`findOverlapping`**(*XF*, *X*, *tol*)

Finds overlapping points of `XF` on `X` grids of points. The two grids are ordered with respect to `Misc.compare()`.

Parameters

- **XF,X** (*2d-array of floats*) – values ordered by row according to the `Misc.compare()`.
- **tol** (*float*) – tolerance to be used

Returns true values for overlapping points of `XF` on `X`, false for not overlapping points. Note: the overlapping return argument is a true-false indexing for `XF`.

Return type 1d-array of bool

Example

```
>>> XF
array([[ -1.73205081e+00,   0.00000000e+00],
       [ -1.00000000e+00,  -1.00000000e+00],
       [ -1.00000000e+00,   0.00000000e+00],
       [ -1.00000000e+00,   1.00000000e+00],
       [  0.00000000e+00,  -1.73205081e+00],
       [  0.00000000e+00,  -1.00000000e+00],
       [  0.00000000e+00,   2.16406754e-16],
       [  0.00000000e+00,   1.00000000e+00],
```

```
       [  0.00000000e+00,   1.73205081e+00],
       [  1.00000000e+00,  -1.00000000e+00],
       [  1.00000000e+00,   0.00000000e+00],
       [  1.00000000e+00,   1.00000000e+00],
       [  1.73205081e+00,   0.00000000e+00]])
>>> X
array([[ -1.73205081e+00,   0.00000000e+00],
       [ -1.00000000e+00,  -1.00000000e+00],
       [ -1.00000000e+00,   0.00000000e+00],
       [ -1.00000000e+00,   1.00000000e+00],
       [  0.00000000e+00,  -1.00000000e+00],
       [  2.16406754e-16,   0.00000000e+00],
       [  0.00000000e+00,   1.00000000e+00],
       [  1.00000000e+00,  -1.00000000e+00],
       [  1.00000000e+00,   0.00000000e+00],
       [  1.00000000e+00,   1.00000000e+00],
       [  1.73205081e+00,   0.00000000e+00]])
>>> tol = 2. * Misc.machineEpsilon()
>>> bool_idx_over = Misc.findOverlapping(XF,X,tol)
>>> XF[np.logical_not(bool_idx_over),:]
array([[ 0.        , -1.73205081],
       [ 0.        ,  1.73205081]])
```

SpectralToolbox.Misc.**machineEpsilon**(*func=<type 'float'>*)

Returns the abolute machine precision for the type passed as argument

Parameters **func** (*dtype*) – type

Returns absolute machine precision

Return type float

```
>>> Misc.machineEpsilon(np.float64)
2.2204460492503131e-16
>>> Misc.machineEpsilon(np.float128)
1.084202172485504434e-19
```

SpectralToolbox.Misc.**powerset**(*iterable*)

Compute the power set of an iterable object.

SpectralToolbox.Misc.**unique_cuts**(*X*, *tol*, *retIdxs=False*)

Returns the unique values and a list of arrays of boolean indicating the positions of the unique values. If retIdx is true, then it returns the group of indices with the same values as a indicator function (true-false array)

Created on Wed Feb 27

@author: Daniele Bigoni (dabi@dtu.dk)

CHAPTER

SEVEN

DESCRIPTION

built up in order to provide flexibility for both:]

1. the types of polynomials to be used per direction
2. the accuracy to be used per direction

The types of polynomials available are all the ones included in the module `Spectral1D`. The rules don't need to be symmetric and the accuracy per each direction can vary.

For rules with Heterogeneous accuracy, two sparse grids will be construced: one partial sparse grid and one full sparse grid (up to the maximum accuracy). The values computed for the partial sparse grid can then be used to interpolate on the points of the full sparse grid. This latter rule can then be used to compute the integral.

CHAPTER

EIGHT

REFERENCES

CHAPTER

NINE

EXAMPLES

Let's consider the following space $\Omega = [-\infty, \infty] times [0,1]$ with the associated measures $\mu_1(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$ and $\mu_2(x) = 1$ for each dimension. The product measure is given by $\mu(\bar{x}) = \prod_{i=1}^{n} \mu_i(x_i)$. We will consider the function

$$f(\bar{x}) = x_1^{p_1} \cdot x_2^{p_2}$$

```
>>> f_mult = lambda x,y,xp,yp: x**xp * y**yp
```

with exact value of the integral given by

$$\int_{\Omega} f(\bar{x})\mu(\bar{x}) = \frac{2^{-1+p_2}\left(1+(-1)^{p_2}\right)\Gamma\left(\frac{1+p_2}{2}\right)}{(1+p_1)\sqrt{\pi}}$$

```
>>> def I_mult(q,p):
>>>     return 2.**(-.5+.5*(-1.+q)) * (1.+(-1.)**q) * scipy.special.gamma((1.+q)/2.) / ((1.+p)*np.sq
```

Let $p_1 = 2$ and $p_2 = 4$. We can obtain a sparse grid composed using Hermite basis (`Spectral1D.HERMITEP_PROB`) and Legendre basis (`Spectral1D.JACOBI`) with orders 2 and 4 respectively.

```
>>> from SpectralToolbox import HeterogeneousSparseGrids as HSG
>>> pH = Spectral1D.Poly1D(Spectral1D.HERMITEP_PROB,None)
>>> pL = Spectral1D.Poly1D(Spectral1D.JACOBI,[0.0,0.0])
>>> polys = [pH,pL]
>>> Ns = [2,4]
>>> sg = HSG.HSparseGrid(polys,Ns)
>>> (XF,W,X) = sg.sparseGrid()
[SG] Sparse Grid Generation [=========================================] 100%
[SG] Sparse Grid Generation: 0.01s
>>> XF[:,1] = (XF[:,1]+1.)/2.
>>> X[:,1] = (X[:,1]+1.)/2.
>>> plt.figure()
>>> plt.plot(XF[:,0],XF[:,1],'o')
>>> plt.plot(X[:,0],X[:,1],'or')
```

The resulting partial and full sparse grids are shown in the following figure.

The values on the partial grid can be computed and then the interpolation is taken over the full sparse grid.

```
>>> fX = f_mult(X[:,0],X[:,1],Q,P)
>>> fXF = sg.sparseGridInterp(X,fX,XF)
[SG] Sparse Grid Interpolation [=======================================] 100%
[SG] Sparse Grid Interpolation: 0.00s
```

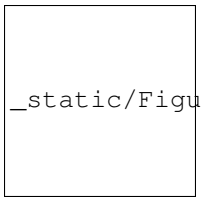

Figure 9.1: Partial (red) and full (blue) sparse grid. The full sparse grid is overlapping over the partial sparse grid.

Finally the error of the quadrature rule is

```
>>> IErr = np.abs(0.5*np.dot(fXF,W)-I_mult(Q,P))
>>> print IErr
3.33066907388e-16
```

class SpectralToolbox.HeterogeneousSparseGrids.**HSparseGrid**(*polys*, *Ns*, *tol=None*, *sdout=<open file '<stderr>', mode 'w' at 0x2b18a359b270>*)

Heterogeneous Sparse Grid class

Constructor of Heterogeneous Sparse Grid object (this does not allocate the sparse grid)

Parameters

- **polys** (list of Spectral1D.Poly1D) – orthogonal polynomials to be used as basis functions
- **Ns** (*list of integers*) – accuracy for each dimension. It can be a list of accuracies or a single accuracy, in which case uniform accuracy is assumed
- **tol** (*float*) – tolerance to be used when comparing points of the grid (optional, default=:py:func:*Misc.machineEpsilon()*)
- **sdout** (*stream*) – default output stream for the class (optional,default=‘‘sys.stderr‘‘)

Note: one of the following must hold: len(polys)==len(Ns) or len(Ns)==1, in which case the same order is used for all the directions.

Example

```
>>> from SpectralToolbox import HeterogeneousSparseGrids as HSG
>>> pH = Spectral1D.Poly1D(Spectral1D.HERMITEP_PROB,None)
>>> pL = Spectral1D.Poly1D(Spectral1D.JACOBI,[0.0,0.0])
>>> polys = [pH,pL]
>>> Ns = [2,4]
>>> sg = HSG.HSparseGrid(polys,Ns)
```

sparseGrid(*heter=False*)

Generates the full and partial sparse grids

Parameters **heter** (*bool*) – if Ns is homogeneous, this parameter will force the output of the partial sparse grid as well

Returns

tuple (XF,WF,X) containing:

- XF: full grid points

- `WF`: full grid weights
- `X`: partial grid points

Example

```
>>> (XF,W,X) = sg.sparseGrid()
[SG] Sparse Grid Generation [=========================================] 100%
[SG] Sparse Grid Generation: 0.01s
```

sparseGridInterp(*X*, *fX*, *XF*)

Interpolate values of the Sparse Grid using 1D Polynomial interpolation along cuts.

Parameters

- **X** (*2d-array of floats*) – partial grid on which a function has been evaluated
- **fX** (*1d-array of floats*) – values for the points in the partial grid
- **XF** (*2d-array of floats*) – full grid on which to interpolate

Returns `fXF` the interpolated values on the full grid

Return type 1d-array of floats

..note:: The partial and full grid must be overlapping

Example

```
>>> fXF = sg.sparseGridInterp(X,fX,XF)
[SG] Sparse Grid Interpolation [=======================================] 100%
[SG] Sparse Grid Interpolation: 0.00s
```

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

S