WaveBlocksND Documentation

Release devel

R. Bourquin

June 29, 2012

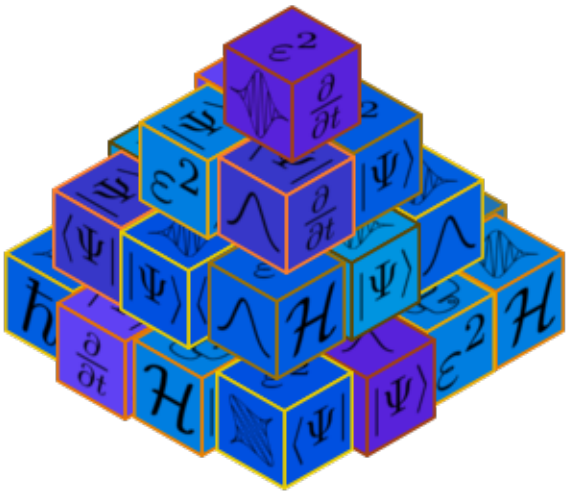
CONTENTS

1 Source code documentation **3**
1.1 WaveBlocks Classes 3

2 Etc **79**
2.1 Citation 79

3 Indices and tables **81**

Python Module Index **83**



Reusable building blocks for simulations with semiclassical wavepackets for solving the time-dependent Schrödinger equation.

Contents:

CHAPTER

ONE

SOURCE CODE DOCUMENTATION

1.1 WaveBlocks Classes

1.1.1 Basic numerics

ComplexMath

About the `ComplexMath` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

Some selected functions for complex math.

@author: R. Bourquin @copyright: Copyright (C) 2011, 2012 R. Bourquin @license: Modified BSD License

`ComplexMath.`**`continuate`**(*data*, *jump=6.283185307179586*, *reference=0.0*)

Make the given data continuous by removing all jumps of size $k \cdot$ jump but not touching jumps of any other size. This can be used to overcome issues with the branch cut along the negative axis. There may be arise problems with jumps that are of size nearly $k \cdot$ jump.

Parameters

- **data** – An array with the input data.
- **jump** – The basic size of jumps which will be removed. Default is 2π.
- **reference** (*A single float number.*) – This value allows the specify the starting point for continuation explicitely. It can be used together with `data`.

`ComplexMath.`**`cont_angle`**(*data*, *reference=None*)

Compute the angle of a complex number *not* constrained to the principal value and avoiding discontinuities at the branch cut. This function just applies 'continuate(.)' to the complex phase.

Parameters

- **data** – An array with the input data.

- **reference** (*A single float number.*) – This value allows the specify the starting point for continuation explicitely. It can be used together with `data`.

`ComplexMath.`**`cont_sqrt`**`(`*data*, *reference=None*`)`
Compute the complex square root (following the Riemann surface) yields a result *not* constrained to the principal value and avoiding discontinuities at the branch cut. This function applies 'continuate(.)' to the complex phase and computes the complex square root according to the formula $\sqrt{z} = \sqrt{r} \exp\left(i \cdot \frac{\phi}{2}\right)$.

Parameters

- **data** – An array with the input data.
- **reference** (*A single float number.*) – This value allows the specify the starting point for continuation explicitely. It can be used together with `data`.

Grid

About the `Grid` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND.`**`Grid`**
This class is an abstract interface to grids in general.

`get_dimension``()`
Return the dimension D of the grid.

DenseGrid

About the `DenseGrid` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND.`**`DenseGrid`**
This class is an abstract interface to dense grids in general.

`get_dimension``()`
Return the dimension D of the grid.

`is_regular``()`
Answers the question if the grid spacing is regular. (It can still be different along each axis!)

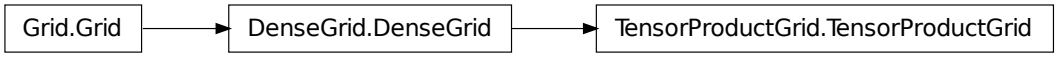
TensorProductGrid

About the `TensorProductGrid` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`TensorProductGrid`**(*limits*, *number_nodes*)
This class represents a dense tensor product grid. It can have an arbitrary dimension D. The grid nodes are enclosed in a hypercubic bounding box. This box can have different limits min_i, max_i along each axis x_i. In each of these intervalls we place N_i grid nodes. Note that the point max_i is not part of the grid. The grid interior is build as the tensor product of all the grid nodes along all the axes.

`get_axes`(*axes=None*)
Returns the one-dimensional grids along the axes.

Parameters **axes** (A single integer or a list of integers. If set to `None` (default) we return the data for all axes.) – The axes for which we want to get the grid.

Returns A list of ndarrays, each having a shape of $(1, ..., N_i, ..., 1)$. We return a list even if it contains just a single element.

get_dimension()
Return the dimension D of the grid.

get_extensions(*axes=None*)
Returns the extensions (length of the edges) of the bounding box.

Parameters **axes** (A single integer or a list of integers. If set to `None` (default) we return the extensions for all axes.) – The axes for which we want to get the extensions.

Returns A list of $|max_i - min_i|$ values.

get_limits(*axes=None*)
Returns the limits of the bounding box.

Parameters **axes** (A single integer or a list of integers. If set to `None` (default) we return the limits for all axes.) – The axes for which we want to get the limits.

Returns A list of (min_i, max_i) ndarrays.

get_meshwidths(*axes=None*)
Returns the meshwidths of the grid.

Parameters **axes** (A single integer or a list of integers. If set to `None` (default) we return the data for all axes.) – The axes for which we want to get the meshwidths.

Returns A list of h_i values or a single value.

get_nodes(*flat=True*, *split=False*)
Returns all grid nodes of the full tensor product grid.

Parameters

- **flat** (Boolean, default is `True`.) – Whether to return the grid with a *hypercubic* $(D, N_1, ..., N_D)$ shape or a *flat* $(D, \prod_i^D N_i)$ shape.
- **split** (Boolean, default is `False`.) – Whether to return the different components, one for each dimension inside a single ndarray or a list with ndarrays, with one item per dimension.

Returns Depends of the optional arguments.

get_number_nodes(*axes=None*, *overall=False*)
Returns the number of grid nodes along a set of axes.

Parameters

- **axes** (A single integer or a list of integers. If set to `None` (default) we return the data for all axes.) – The axes for which we want to get the number of nodes.
- **overall** (Boolean, default is `False`) – Compute the product $\prod_i^D N_i$ of the number N_i of grid nodes along each axis i specified.

Returns A list of N_i values or a single value N.

is_regular()
Answers the question if the grid spacing is regular. (It can still be different along each axis!)

GridWrapper

About the `GridWrapper` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND.`**`GridWrapper`**(*anarray*)

This class constructs a thin layer around an `ndarray` and wrappes it as `Grid` subclass for API compatibility. The array must have a shape of (D, N) with N the overall number of nodes.

Rather than using this class, one should try to elliminate the cases where it is used now.

`get_dimension`()

Return the dimension D of the grid.

`get_nodes`(*flat=True*, *split=False*)

Returns all grid nodes.

Parameters

- **flat** (Boolean, default is *True*.) – Whether to return the grid with a *hypercubic* $(D, N_1, ..., N_D)$ shape or a *flat* $(D, \prod_i^D N_i)$ shape. Note that the hypercubic shape is not implemented!
- **split** (Boolean, default is *False*.) – Whether to return the different components, one for each dimension inside a single ndarray or a list with ndarrays, with one item per dimension.

Returns Depends of the optional arguments.

`get_number_nodes`(*overall=False*)

Returns the number of grid nodes.

Parameters **overall** (Boolean, default is `False`) – Compute the product $N = \prod_i^D N_i$ of the number N_i of grid nodes along each dimension i specified.

Returns A list of N_i values or a single value N.

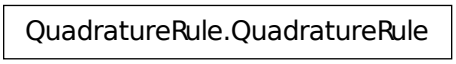
QuadratureRule

About the `QuadratureRule` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`QuadratureRule`**

This class is an abstract interface to quadrature rules in general.

`get_description``()`

Return a description of this quadrature rule object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

`get_dimension``()`

Returns The space dimension D of the quadrature rule.

`get_nodes``()`

Returns A two-dimensional ndarray containing the quadrature nodes γ_i. The array must have a shape of $(D, |\Gamma|)$.

`get_number_nodes``()`

Returns The number of quadrature nodes denoted by $|\Gamma|$ that are part of this quadrature rule $\Gamma = (\gamma, \omega)$.

`get_order``()`

Returns The order R of the quadrature rule.

`get_weights``()`

Returns A two-dimensional ndarray containing the quadrature weights ω_i. The array must have a shape of $(1, |\Gamma|)$.

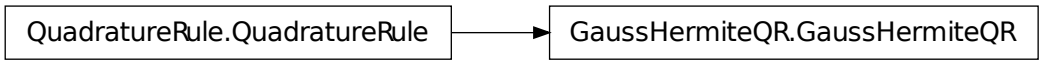
GaussHermiteQR

About the `GaussHermiteQR` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`GaussHermiteQR`**(*order*)

This class implements a Gauss-Hermite quadrature rule.

`get_description`()

Return a description of this quadrature rule object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

`get_dimension`()

Returns The space dimension D of the quadrature rule.

`get_nodes`()

Returns the quadrature nodes γ_i.

Returns An array containing the quadrature nodes γ_i.

`get_number_nodes`()

Returns The number of quadrature nodes denoted by $|\Gamma|$ that are part of this quadrature rule $\Gamma = (\gamma, \omega)$.

`get_order`()

Returns The order R of the quadrature rule.

`get_weights`()

Returns the quadrature weights ω_i.

Returns An array containing the quadrature weights ω_i.

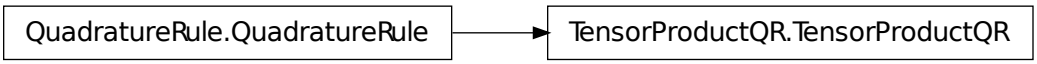
TensorProductQR

About the `TensorProductQR` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**TensorProductQR**(*rules*)

This class implements the construction of high dimensional quadrature rules from one-dimensional ones by taking tensor products.

get_description()

Return a description of this quadrature rule object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

get_dimension()

Returns The space dimension D of the quadrature rule.

get_nodes(*flat=True*, *split=False*)

Returns the quadrature nodes γ_i.

Parameters

- **flat** (Boolean, default is *True*.) – Dummy parameter for API compatibility with Grids.
- **split** (Boolean, default is *False*.) – Dummy parameter for API compatibility with Grids.

Returns An ndarray containing the quadrature nodes γ_i.

get_number_nodes()

Returns The number of quadrature nodes denoted by $|\Gamma|$ that are part of this quadrature rule $\Gamma = (\gamma, \omega)$.

get_order()

Returns The order R of the quadrature rule.

get_weights()

Returns the quadrature weights ω_i.

Returns An ndarray containing the quadrature weights ω_i.

MatrixExponential

About the MatrixExponential functions

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

This file contains several different algorithms to compute the matrix exponential. Currently we have an exponential based on Pade approximations and an Arnoldi iteration method.

@author: R. Bourquin @copyright: Copyright (C) 2007 V. Gradinaru @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

`MatrixExponential.`**`arnoldi`**(*A*, *v0*, *k*)

Arnoldi algorithm to compute the Krylov approximation H of a matrix A.

Parameters

- **A** – The matrix A of shape $N \times N$ to approximate.
- **v0** – The initial vector v_0 of length N. (Should be in matrix shape $(N, 1)$ for practical reasons.)
- **k** – The number k of Krylov steps performed.

Returns A tuple (V, H) where V is the large matrix of shape $N \times k$ containing the orthogonal vectors and H is the small matrix of shape $k \times k$ containing the Krylov approximation of A.

`MatrixExponential.`**`matrix_exp_arnoldi`**(*A*, *v*, *factor*, *k*)

Compute the solution of $v' = Av$ via k steps of a the Arnoldi krylov method.

Parameters

- **A** – The matrix A of shape $N \times N$.
- **v** – The vector v of length N.
- **factor** – An additional scalar factor α.
- **k** – The number k of Krylov steps performed.

Returns The (approximate) value of $\exp(-i\alpha A)\, v$.

`MatrixExponential.`**`matrix_exp_pade`**(*A*, *v*, *factor*)

Compute the solution of $v' = Av$ with a full matrix exponential via Pade approximation.

Parameters

- **A** – The matrix A of shape $N \times N$.
- **v** – The vector v of length N.
- **factor** – An additional scalar factor α.

Returns The (approximate) value of $\exp(-i\alpha A)\, v$

Utils

About the `Utils` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

Various small utility functions.

@author: R. Bourquin @copyright: Copyright (C) 2012 R. Bourquin @license: Modified BSD License

Utils.**meshgrid_nd**(*arrays*)
Like 'meshgrid()' but for arbitrary number of dimensions.

1.1.2 Basic quantum mechanics

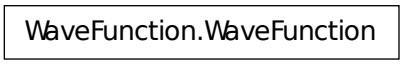
WaveFunction

About the WaveFunction class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**WaveFunction**(*parameters*)
This class represents a vector valued wavefuncton Ψ as used in the vector valued time-dependent Schroedinger equation. The state Ψ consists of the components ψ_0 to ψ_{N-1}.

get_grid()
Return the Grid instance representing the grid Γ. The wavefunction Ψ is evaluated on the grid nodes to get $\Psi(\Gamma)$.

get_number_components()
The number of components ψ_i the vector Ψ consists of.

get_values(*components=None*)
Get the wavefunction values $\psi(\Gamma)$ for each component ψ_i of Ψ.

Parameters **components** (A single integer or a list of integers. If set to *None* (default) we return the data for all components.) – The components i for which we want to get the wavefunction values ψ_i.

Returns A list of the values ψ_i for all components i.

kinetic_energy(*kinetic*, *summed=False*)
Calculate the kinetic energy $E_{\text{kin}} := \langle\Psi|T|\Psi\rangle$ of the different components ψ_i.

Parameters

- **kinetic** (A KineticOperator instance.) – The kinetic energy operator $T(\omega)$.
- **summed** – Whether to sum up the kinetic energies E_i of the individual components ψ_i. Default is *False*.

Returns A list with the kinetic energies of the individual components or the overall kinetic energy of the wavefunction. (Depending on the optional arguments.)

norm(*values=None*, *summed=False*, *components=None*)
Calculate the L^2 norm of the whole vector values wavefunction Ψ or some individual components ψ_i. The calculation is done in momentum space.

Parameters

- **values** – Allows to use this function for external data, similar to a static function.
- **summed** (Boolean, default is *False*.) – Whether to sum up the norms of the individual components.
- **components** (A single integer or a list of integers. If set to *None* (default) we compute the norm for all components.) – The components ψ_i of which the norms are calculated.

Returns The L^2 norm of Ψ or a list of L^2 norms of the specified components ψ_i.

potential_energy(*potential*, *summed=False*)
Calculate the potential energy $E_{\text{pot}} := \langle\Psi|V|\Psi\rangle$ of the different components ψ_i.

Parameters

- **potential** – The potential energy operator $V(x)$.
- **summed** – Whether to sum up the potential energies E_i of the individual components ψ_i. Default is *False*.

Returns A list with the potential energies of the individual components or the overall potential energy of the wavefunction. (Depending on the optional arguments.)

set_grid(*grid*)
Assign a new grid Γ to this WaveFunction instance.

Note: The user of this class has to make sure that the grid Γ and the wavefunction values $\Psi(\Gamma)$ are consistent with each other!

Parameters **grid** – A new Grid instance.

set_values(*values*, *components=None*)
Assign new wavefunction values $\psi_i(\Gamma)$ for each component i of Ψ to the current WaveFunction instance.

Parameters

- **values** (*Each entry of the list has to be an ndarray.*) – A list with the new values of all components we want to change.
- **components** (A single integer or a list of integers. If set to *None* (default) we set the data for all components.) – The components i for which we want to set the new wavefunction values ψ_i.

Note: This method does NOT copy the input data arrays.

Raises ValueError If the list of *values* has the wrong length.

MatrixPotential

About the `MatrixPotential` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND.`**`MatrixPotential`**

This class represents a potential $V(x)$ with $x \in \mathbb{R}^D$. The potential is given as an analytic expression. Some calculations with the potential are supported. For example calculation of eigenvalues $\lambda_i(x)$ and eigenvectors $\nu_i(x)$ and numerical evaluation on a grid Γ.

`calculate_eigenvalues``()`

Calculate the eigenvalue $\lambda_0(x)$ of the potential $V(x)$.

Raises NotImplementedError This is an abstract base class.

`calculate_eigenvectors``()`

Calculate the eigenvectors $\nu_i(x)$ of the potential $V(x)$.

Raises NotImplementedError This is an abstract base class.

`calculate_exponential`(*factor=1*)

Calculate the matrix exponential $\exp(\alpha V)$.

Parameters **factor** – The prefactor α in the exponential.

Raises NotImplementedError This is an abstract base class.

`evaluate_at`(*grid*, *entry=None*)

Evaluate the potential $V(x)$ elementwise on a grid Γ.

Parameters

- **grid** – The grid containing the nodes γ_i we want to evaluate the potential at.
- **entry** – The indices (i, j) of the component $V_{i,j}(x)$ we want to evaluate or *None* to evaluate all entries.

Raises NotImplementedError This is an abstract base class.

`evaluate_eigenvalues_at`(*grid*, *entry=None*)

Evaluate the eigenvalues $\Lambda(x)$ elementwise on a grid Γ.

Parameters

- **grid** – The grid containing the nodes γ_i we want to evaluate the eigenvalues at.
- **entry** – The index i of the eigenvalue $\lambda_i(x)$ we want to evaluate or *None* to evaluate all eigenvalues.

Raises NotImplementedError This is an abstract base class.

evaluate_eigenvectors_at(*grid*, *entry=None*)
Evaluate the eigenvectors $\nu_i(x)$ elementwise on a grid Γ.

Parameters

- **grid** – The grid containing the nodes γ_i we want to evaluate the eigenvectors at.
- **entry** – The index i of the eigenvector $\nu_i(x)$ we want to evaluate or *None* to evaluate all eigenvectors.

Raises NotImplementedError This is an abstract base class.

evaluate_exponential_at(*grid*)
Evaluate the exponential of the potential matrix $V(x)$ on a grid Γ.

Parameters **grid** – The grid containing the nodes γ_i we want to evaluate the exponential at.

Raises NotImplementedError This is an abstract base class.

get_dimension()
Return the dimension D of the potential $V(x)$. The dimension is equal to the number of free variables x_i where $x := (x_1, x_2, ..., x_D)$.

get_number_components()
Return the number N of components the potential $V(x)$ supports. This is equivalent to the number of energy levels $\lambda_i(x)$.

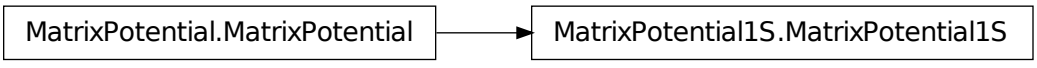
MatrixPotential1S

About the MatrixPotential1S class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**MatrixPotential1S**(*expression*, *variables*, ***kwargs*)

This class represents a scalar potential $V(x)$. The potential is given as an analytic 1×1 matrix expression. Some symbolic calculations with the potential are supported.

calculate_eigenvalues()

Calculate the eigenvalue $\lambda_0(x)$ of the potential $V(x)$. In the scalar case this is just equal to the matrix entry $V_{0,0}(x)$. Note: This function is idempotent and the eigenvalues are memoized for later reuse.

calculate_eigenvectors()

Calculate the eigenvector $\nu_0(x)$ of the potential $V(x)$. In the scalar case this is just the value 1. Note: This function is idempotent and the eigenvectors are memoized for later reuse.

calculate_exponential(*factor=1*)

Calculate the matrix exponential $\exp(\alpha V)$. In the case of this class the matrix is of size 1×1 thus the exponential simplifies to the scalar exponential function. Note: This function is idempotent.

Parameters **factor** – The prefactor α in the exponential.

calculate_hessian()

Calculate the Hessian matrix $\nabla^2 V$ of the potential $V(x)$ with $x \in \mathbb{R}^D$. For potentials which depend only one variable, this equals the second derivative and $D = 1$. Note that this function is idempotent.

calculate_jacobian()

Calculate the Jacobian matrix ∇V of the potential $V(x)$ with $x \in \mathbb{R}^D$. For potentials which depend only one variable, this equals the first derivative and $D = 1$. Note that this function is idempotent.

calculate_local_quadratic(*diagonal_component=None*)

Calculate the local quadratic approximation $U(x)$ of the potential's eigenvalue $\lambda(x)$. Note that this function is idempotent.

Parameters **diagonal_component** – Dummy parameter that has no effect here.

calculate_local_remainder(*diagonal_component=None*)

Calculate the non-quadratic remainder $W(x) = V(x) - U(x)$ of the quadratic Taylor approximation $U(x)$ of the potential's eigenvalue $\lambda(x)$. Note that this function is idempotent.

Parameters **diagonal_component** – Dummy parameter that has no effect here.

evaluate_at(*grid*, *entry=None*, *as_matrix=False*)

Evaluate the potential $V(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A Grid instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the potential at.
- **entry** (*A python tuple of two integers.*) – The indices (i, j) of the component $V_{i,j}(x)$ we want to evaluate or *None* to evaluate all entries. This has no effect here as we only have a single entry $V_{0,0}$.
- **as_matrix** – Dummy parameter which has no effect here.

Returns A list containing a single numpy ndarray of shape $(1, |\Gamma|)$.

evaluate_eigenvalues_at(*grid*, *entry=None*, *as_matrix=False*)

Evaluate the eigenvalue $\lambda_0(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A Grid instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the eigenvalue at.

- **entry** (*A python tuple of two integers.*) – The indices (i, j) of the component $\Lambda_{i,j}(x)$ we want to evaluate or *None* to evaluate all entries. If $j = i$ then we evaluate the eigenvalue $\lambda_i(x)$. This has no effect here as we only have a single entry λ_0.
- **as_matrix** – Dummy parameter which has no effect here.

Returns A list containing a single numpy ndarray of shape $(N_1, ..., N_D)$.

evaluate_eigenvectors_at(*grid*, *entry=None*)
Evaluate the eigenvector $\nu_0(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the eigenvector at.
- **entry** (*A singly python integer.*) – The index i of the eigenvector $\nu_i(x)$ we want to evaluate or *None* to evaluate all eigenvectors. This has no effect here as we only have a single entry ν_0.

Returns A list containing the numpy ndarrays, all of shape $(1, |\Gamma|)$.

evaluate_exponential_at(*grid*, *entry=None*)
Evaluate the exponential of the potential matrix $V(x)$ on a grid Γ.

Parameters **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the exponential at.

Returns The numerical approximation of the matrix exponential at the given grid nodes.

evaluate_hessian_at(*grid*, *component=None*)
Evaluate the potential's Hessian $\nabla^2 V(x)$ at some grid nodes Γ.

Parameters

- **grid** – The grid nodes Γ the Hessian gets evaluated at.
- **component** – Dummy parameter that has no effect here.

Returns The value of the potential's Hessian at the given nodes. The result is an `ndarray` of shape (D, D) is we evaluate at a single grid node or of shape $(|\Gamma|, D, D)$ if we evaluate at multiple nodes simultaneously.

evaluate_jacobian_at(*grid*, *component=None*)
Evaluate the potential's Jacobian $\nabla V(x)$ at some grid nodes Γ.

Parameters

- **grid** – The grid nodes Γ the Jacobian gets evaluated at.
- **component** – Dummy parameter that has no effect here.

Returns The value of the potential's Jacobian at the given nodes. The result is an `ndarray` of shape $(D, 1)$ is we evaluate at a single grid node or of shape $(D, |\Gamma|)$ if we evaluate at multiple nodes simultaneously.

evaluate_local_quadratic_at(*grid*, *diagonal_component=None*)
Numerically evaluate the local quadratic approximation $U(x)$ of the potential's eigenvalue $\lambda(x)$ at the given grid nodes Γ. This function is used for the homogeneous case.

Parameters

- **grid** – The grid nodes Γ the quadratic approximation gets evaluated at.
- **diagonal_component** – Dummy parameter that has no effect here.

Returns A list containing the values $V(\Gamma)$, $\nabla V(\Gamma)$ and $\nabla^2 V(\Gamma)$.

evaluate_local_remainder_at(*grid*, *position*, *diagonal_component=None*, *entry=None*)
Numerically evaluate the non-quadratic remainder $W(x)$ of the quadratic approximation $U(x)$ of the potential's eigenvalue $\lambda(x)$ at the given nodes Γ.

Parameters

- **grid** – The grid nodes Γ the remainder W gets evaluated at.
- **position** – The point $q \in \mathbb{R}^D$ where the Taylor series is computed.
- **diagonal_component** – Dummy parameter that has no effect here.
- **entry** – Dummy parameter that has no effect here.

Returns A list with a single entry consisting of an `ndarray` containing the values of $W(\Gamma)$. The array is of shape $(1, |\Gamma|)$.

get_dimension()
Return the dimension D of the potential $V(x)$. The dimension is equal to the number of free variables x_i where $x := (x_1, x_2, ..., x_D)$.

get_number_components()
Return the number N of components the potential $V(x)$ supports. This is equivalent to the number of energy levels $\lambda_i(x)$.

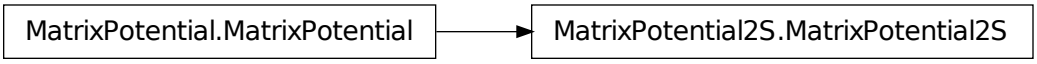
MatrixPotential2S

About the MatrixPotential2S class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**MatrixPotential2S**(*expression*, *variables*, ***kwargs*)
This class represents a matrix potential $V(x)$. The potential is given as an analytic 2×2 matrix expression. Some symbolic calculations with the potential are supported.

calculate_eigenvalues()
Calculate the two eigenvalues $\lambda_i(x)$ of the potential $V(x)$. We can do this by symbolic calculations. The multiplicities are taken into account. Note: This function is idempotent and the eigenvalues are memoized for later reuse.

calculate_eigenvectors()
Calculate the two eigenvectors $\nu_i(x)$ of the potential $V(x)$. We can do this by symbolic calculations. Note: This function is idempotent and the eigenvectors are memoized for later reuse.

calculate_exponential(*factor=1*)
Calculate the matrix exponential $\exp(\alpha V)$. In the case of this class the matrix is of size 2×2 thus the exponential can be calculated analytically for a general matrix. Note: This function is idempotent.

Parameters **factor** – The prefactor α in the exponential.

calculate_hessian()
Calculate the Hessian matrix $\nabla^2 \lambda_i$ of the potential's eigenvalues $\Lambda(x)$ with $x \in \mathbb{R}^D$. For potentials which depend only one variable, this equals the second derivative and $D = 1$. Note that this function is idempotent.

calculate_jacobian()
Calculate the Jacobian matrix $\nabla \lambda_i$ of the potential's eigenvalues $\Lambda(x)$ with $x \in \mathbb{R}^D$. For potentials which depend only one variable, this equals the first derivative and $D = 1$. Note that this function is idempotent.

calculate_local_quadratic(*diagonal_component=None*)
Calculate the local quadratic approximation matrix $U(x)$ of the potential's eigenvalues in $\Lambda(x)$. This function can be used for the homogeneous case and takes into account the leading component $\chi \in [0, \ldots, N-1]$. If the parameter i is not given, calculate the local quadratic approximation matrix $U(x)$ of all the potential's eigenvalues in Λ. This case can be used for the inhomogeneous case.

Parameters **diagonal_component** (Integer or `None` (default)) – Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i.

calculate_local_remainder(*diagonal_component=None*)
Calculate the non-quadratic remainder matrix $W(x) = V(x) - U(x)$ of the quadratic approximation matrix $U(x)$ of the potential's eigenvalue matrix $\Lambda(x)$. In the homogeneous case the matrix U is given by $U(x) = \text{diag}([u_i, \ldots, u_i])$ where in the inhomogeneous case it is given by $U(x) = \text{diag}([u_0, \ldots, u_{N-1}])$.

Parameters **diagonal_component** (Integer or `None` (default)) – Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i. If set to `None` the inhomogeneous case is computed.

evaluate_at(*grid*, *entry=None*, *as_matrix=True*)
Evaluate the potential $V(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the potential at.
- **entry** (*A python tuple of two integers.*) – The indices (i, j) of the component $V_{i,j}(x)$ we want to evaluate or *None* to evaluate all entries.
- **as_matrix** – Dummy parameter which has no effect here.

Returns A list containing 4 numpy ndarrays of shape $(1, |\Gamma|)$.

evaluate_eigenvalues_at(*grid*, *entry=None*, *as_matrix=False*)
Evaluate the eigenvalues $\lambda_i(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid Γ containing the nodes γ_i we want to evaluate the eigenvalues at.
- **entry** (*A python tuple of two integers.*) – The indices (i, j) of the component $\Lambda_{i,j}(x)$ we want to evaluate or *None* to evaluate all entries. If $j = i$ then we evaluate the eigenvalue $\lambda_i(x)$.

- **as_matrix** – Whether to include the off-diagonal zero entries of $\Lambda_{i,j}(x)$ in the return value.

Returns A list containing the numpy `ndarray`, all of shape $(1, |\Gamma|)$.

evaluate_eigenvectors_at(*grid*, *entry=None*)
Evaluate the two eigenvectors $\nu_i(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the eigenvectors at.
- **entry** (*A singly python integer.*) – The index i of the eigenvector $\nu_i(x)$ we want to evaluate or *None* to evaluate all eigenvectors.

Returns A list containing the numpy ndarrays, all of shape $(N, |\Gamma|)$.

evaluate_exponential_at(*grid*)
Evaluate the exponential of the potential matrix $V(x)$ on a grid Γ.

Parameters **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the exponential at.

Returns The numerical approximation of the matrix exponential at the given grid nodes. A list contains the exponentials for all entries (i, j), each having the same shape as the grid.

evaluate_hessian_at(*grid*, *component=None*)
Evaluate the list of Hessian matrices $\nabla^2 \lambda_i(x)$ at some grid nodes Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid nodes Γ the Hessian gets evaluated at.
- **component** – Dummy parameter that has no effect here.

Returns The value of the potential's Hessian at the given nodes. The result is an `ndarray` of shape (D, D) is we evaluate at a single grid node or of shape $(|\Gamma|, D, D)$ if we evaluate at multiple nodes simultaneously.

evaluate_jacobian_at(*grid*, *component=None*)
Evaluate the list of Jacobian matrices $\nabla \lambda_i(x)$ at some grid nodes Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid nodes Γ the Jacobian gets evaluated at.
- **component** – Dummy parameter that has no effect here.

Returns The value of the potential's Jacobian at the given nodes. The result is a list of `ndarray` each of shape $(D, 1)$ is we evaluate at a single grid node or of shape $(D, |\Gamma|)$ if we evaluate at multiple nodes simultaneously.

evaluate_local_quadratic_at(*grid*, *diagonal_component=None*)
Numerically evaluate the local quadratic approximation matrix $U(x)$ of the potential's eigenvalues in $\Lambda(x)$ at the given grid nodes Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid Γ containing the nodes γ we want to evaluate the quadratic approximation at.
- **diagonal_component** – Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i.

Returns A list of tuples or a single tuple. Each tuple (λ, J, H) contains the the evaluated eigenvalues $\lambda_i(\Gamma)$, the Jacobian $J(\Gamma)$ and the Hessian $H(\Gamma)$ in this order.

evaluate_local_remainder_at(*grid*, *position*, *diagonal_component=None*, *entry=None*)
Numerically evaluate the non-quadratic remainder $W(x)$ of the quadratic approximation $U(x)$ of the potential's eigenvalue $\Lambda(x)$ at the given nodes Γ.

Warning: do not set the `diagonal_component` and the `entry` parameter both to `None`.

Parameters

- **grid** – The grid nodes Γ the remainder W gets evaluated at.
- **position** – The point $q \in \mathbb{R}^D$ where the Taylor series is computed.
- **diagonal_component** (Integer or `None` (default)) – Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i and whose remainder matrix $W(x) = V(x) - \operatorname{diag}([u_i, \ldots, u_i])$ we evaluate. If set to `None` the inhomogeneous case given by $W(x) = V(x) - \operatorname{diag}([u_0, \ldots, u_{N-1}])$ is computed.
- **entry** (*A python tuple of two integers.*) – The entry (i, j) of the remainder matrix W that is evaluated.

Returns A list with N^2 `ndarray` elements or a single `ndarray`. Each containing the values of $W_{i,j}(\Gamma)$. Each array is of shape $(1, |\Gamma|)$.

get_dimension()
Return the dimension D of the potential $V(x)$. The dimension is equal to the number of free variables x_i where $x := (x_1, x_2, ..., x_D)$.

get_number_components()
Return the number N of components the potential $V(x)$ supports. This is equivalent to the number of energy levels $\lambda_i(x)$.

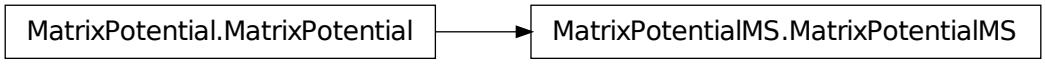
MatrixPotentialMS

About the `MatrixPotentialMS` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND`.**MatrixPotentialMS**(*expression*, *variables*, ***kwargs*)
This class represents a matrix potential $V(x)$. The potential is given as an analytic $N \times N$ matrix expression.

All methods use pure numerical techniques because symbolical calculations are unfeasible for 3 or more energy levels.

calculate_eigenvalues()
Calculate all the eigenvalues $\lambda_i(x)$ of the potential $V(x)$. We can not do this by symbolic calculations, hence the function has an empty implementation. We compute the eigenvalues by numercial techniques in the corresponding *evaluate_eigenvalues_at* function.

calculate_eigenvectors()
Calculate all the eigenvectors $\nu_i(x)$ of the potential $V(x)$. We can not do this by symbolic calculations, hence the function has an empty implementation. We compute the eigenvectors by numercial techniques in the corresponding *evaluate_eigenvectors_at* function.

calculate_exponential(*factor=1*)
Calculate the matrix exponential $\exp(\alpha V)$. In the case of this class the matrix is of size $N \times N$ thus the exponential can not be calculated analytically for a general matrix. We use numerical approximations to determine the matrix exponential. We just store the prefactor α for use during numerical evaluation.

Parameters **factor** – The prefactor α in the exponential.

calculate_hessian()
Calculate the Hessian matrix $\nabla^2\lambda_i$ of the potential's eigenvalues $\Lambda(x)$ with $x \in \mathbb{R}^D$. For potentials which depend only one variable, this equals the second derivative and $D = 1$. Note that this function is idempotent.

calculate_jacobian()
Calculate the Jacobian matrix $\nabla\lambda_i$ of the potential's eigenvalues $\Lambda(x)$ with $x \in \mathbb{R}^D$. For potentials which depend only one variable, this equals the first derivative and $D = 1$. Note that this function is idempotent.

calculate_local_quadratic(*diagonal_component=None*)
Calculate the local quadratic approximation matrix $U(x)$ of the potential's eigenvalues in $\Lambda(x)$. This function can be used for the homogeneous case and takes into account the leading component $\chi \in [0, \ldots, N-1]$. If the parameter i is not given, calculate the local quadratic approximation matrix $U(x)$ of all the potential's eigenvalues in Λ. This case can be used for the inhomogeneous case.

Parameters **diagonal_component** – Dummy parameter which has no effect here.

calculate_local_remainder(*diagonal_component=None*)
Calculate the non-quadratic remainder matrix $W(x) = V(x) - U(x)$ of the quadratic approximation matrix $U(x)$ of the potential's eigenvalue matrix $\Lambda(x)$. In the homogeneous case the matrix U is given by $U(x) = \text{diag}([u_i, \ldots, u_i])$ where in the inhomogeneous case it is given by $U(x) = \text{diag}([u_0, \ldots, u_{N-1}])$.

Parameters **diagonal_component** (Integer or `None` (default)) – Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i. If set to `None` the inhomogeneous case is computed.

evaluate_at(*grid*, *entry=None*, *as_matrix=True*)
Evaluate the potential $V(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the potential at.
- **entry** (*A python tuple of two integers.*) – The indices (i, j) of the component $V_{i,j}(x)$ we want to evaluate or *None* to evaluate all entries.
- **as_matrix** – Dummy parameter which has no effect here.

Returns A list containing N^2 numpy ndarrays of shape $(1, |\Gamma|)$.

evaluate_eigenvalues_at(*grid*, *entry=None*, *as_matrix=False*, *sorted=True*)
Evaluate the eigenvalues $\lambda_i(x)$ elementwise on a grid Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the eigenvalues at.
- **entry** (*A python tuple of two integers.*) – The indices (i, j) of the component $\Lambda_{i,j}(x)$ we want to evaluate or *None* to evaluate all entries. If $j = i$ then we evaluate the eigenvalue $\lambda_i(x)$.
- **as_matrix** – Whether to include the off-diagonal zero entries of $\Lambda_{i,j}(x)$ in the return value.

Returns A list containing the numpy ndarrays, all of shape $(1, |\Gamma|)$.

evaluate_eigenvectors_at(*grid*, *sorted=True*)
Evaluate the eigenvectors $\nu_i(x)$ elementwise on a grid Γ.

Parameters **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the eigenvectors at.

Returns A list containing the N numpy ndarrays, all of shape $(D, |\Gamma|)$.

evaluate_exponential_at(*grid*)
Evaluate the exponential of the potential matrix $V(x)$ on a grid Γ.

Parameters **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid containing the nodes γ_i we want to evaluate the exponential at.

Returns The numerical approximation of the matrix exponential at the given grid nodes. A list contains the exponentials for all entries (i, j), each having a shape of $(1, |\Gamma|)$.

evaluate_hessian_at(*grid*, *component=None*)
Evaluate the list of Hessian matrices $\nabla^2\lambda_i(x)$ at some grid nodes Γ for one or all eigenvalues.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid nodes Γ the Hessian gets evaluated at.
- **component** – The index i of the eigenvalue λ_i.

Returns The value of the potential's Hessian at the given nodes. The result is an `ndarray` of shape (D, D) is we evaluate at a single grid node or of shape $(D, D, |\Gamma|)$ if we evaluate at multiple nodes simultaneously.

evaluate_jacobian_at(*grid*, *component=None*)
Evaluate the list of Jacobian matrices $\nabla\lambda_i(x)$ at some grid nodes Γ for one or all eigenvalues.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid nodes Γ the Jacobian gets evaluated at.
- **component** – The index i of the eigenvalue λ_i.

Returns The value of the potential's Jacobian at the given nodes. The result is a list of `ndarray` each of shape $(D, 1)$ is we evaluate at a single grid node or of shape $(D, |\Gamma|)$ if we evaluate at multiple nodes simultaneously.

evaluate_local_quadratic_at(*grid*, *diagonal_component=None*)
Numerically evaluate the local quadratic approximation matrix $U(x)$ of the potential's eigenvalues in $\Lambda(x)$ at the given grid nodes Γ.

Parameters

- **grid** (A `Grid` instance. (Numpy arrays are not directly supported yet.)) – The grid Γ containing the nodes γ we want to evaluate the quadratic approximation at.
- **diagonal_component** – Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i.

Returns A list of tuples or a single tuple. Each tuple (λ, J, H) contains the the evaluated eigenvalue $\lambda_i(\Gamma)$, its Jacobian $J(\Gamma)$ and its Hessian $H(\Gamma)$ in this order.

evaluate_local_remainder_at(*grid*, *position*, *diagonal_component=None*, *entry=None*)

Numerically evaluate the non-quadratic remainder $W(x)$ of the quadratic approximation $U(x)$ of the potential's eigenvalue $\Lambda(x)$ at the given nodes Γ.

Warning: do not set the `diagonal_component` and the `entry` parameter both to `None`.

Parameters

- **grid** – The grid nodes Γ the remainder W gets evaluated at.
- **position** – The point $q \in \mathbb{R}^D$ where the Taylor series is computed.
- **diagonal_component** (Integer or `None` (default)) – Specifies the index i of the eigenvalue λ_i that gets expanded into a Taylor series u_i and whose remainder matrix $W(x) = V(x) - \text{diag}([u_i, \ldots, u_i])$ we evaluate. If set to `None` the inhomogeneous case given by $W(x) = V(x) - \text{diag}([u_0, \ldots, u_{N-1}])$ is computed.
- **entry** (*A python tuple of two integers.*) – The entry (i, j) of the remainder matrix W that is evaluated.

Returns A list with N^2 `ndarray` elements or a single `ndarray`. Each containing the values of $W_{i,j}(\Gamma)$. Each array is of shape $(1, |\Gamma|)$.

get_dimension()

Return the dimension D of the potential $V(x)$. The dimension is equal to the number of free variables x_i where $x := (x_1, x_2, ..., x_D)$.

get_number_components()

Return the number N of components the potential $V(x)$ supports. This is equivalent to the number of energy levels $\lambda_i(x)$.

1.1.3 Wavepackets

BasisShape

About the `BasisShape` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class WaveBlocksND.**BasisShape**

This class defines the abstract interface to basis shapes. A basis shape is essentially all information and operations related to the set $\mathcal{K}$ of multi-indices k.

Basis shapes must be immutable objects.

contains(*k*)

Checks if a given multi-index k is part of the basis set $\mathcal{K}$.

Parameters **k** (*tuple*) – The multi-index we want to test.

Raises **NotImplementedError** Abstract interface.

get_basis_size(*extended=False*)

Returns the size $|\mathcal{K}|$ of the basis. The size is the number of distinct multi-indices k that belong to the basis $\mathcal{K}$.

get_description()

Return a description of this basis shape object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current basis shape. A description never contains any data.

get_dimension()

Returns the dimension D of the basis shape $\mathcal{K}$. This is defined as the number of components each multi-index $k = (k_0, \ldots, k_{D-1})$ has.

get_neighbours(*k*, *direction=None*)

Returns a list of all multi-indices that are neighbours of a given multi-index k. A direct neighbours is defines as $(k_0, \ldots, k_d \pm 1, \ldots, k_{D-1}) \forall d \in [0 \ldots D-1]$.

Parameters

- **k** (*tuple*) – The multi-index of which we want to get the neighbours.
- **direction** (*int*) – The direction $0 \leq d < D$ in which we want to find the neighbours $k \pm e_d$.

Returns A list containing the pairs (d, k').

Raises **NotImplementedError** Abstract interface.

get_node_iterator(*mode='lex'*)

Returns an iterator to iterate over all basis elements k.

Parameters **mode** (*string*) – The mode by which we iterate over the indices. Default is 'lex' for lexicographical order. Supported is also 'chain', for the chain-like mode, details see the manual.

Raises **NotImplementedError** Abstract interface.

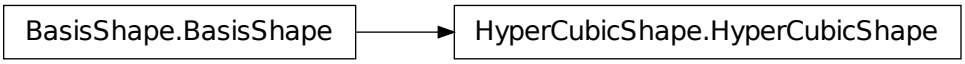
HyperCubicShape

About the `HyperCubicShape` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`HyperCubicShape`**(*limits*)

This class implements the hypercubic basis shape which is the full dense basis set. A basis shape is essentially all information and operations related to the set $\mathcal{K}$ of multi-indices k.

`contains`(*k*)

Checks if a given multi-index k is part of the basis set $\mathcal{K}$.

Parameters **k** (*tuple*) – The multi-index we want to test.

`extend`()

Extend the basis shape such that (at least) all neighbours of all boundary nodes are included in the extended basis shape.

`get_basis_size`(*extended=False*)

Returns the size $|\mathcal{K}|$ of the basis. The size is the number of distinct multi-indices k that belong to the basis $\mathcal{K}$.

`get_description`()

Return a description of this basis shape object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current basis shape. A description never contains any data.

`get_dimension`()

Returns the dimension D of the basis shape $\mathcal{K}$. This is defined as the number of components each multi-index $k = (k_0, \ldots, k_{D-1})$ has.

`get_limits`()

Returns the upper limit K_d for all directions d. :return: A tuple of the maximum of the multi-index in each direction.

`get_neighbours`(*k*, *selection=None*, *direction=None*)

Returns a list of all multi-indices that are neighbours of a given multi-index k. A direct neighbour is defined as $(k_0, \ldots, k_d \pm 1, \ldots, k_{D-1}) \forall d \in [0 \ldots D-1]$.

Parameters

- **k** (*tuple*) – The multi-index of which we want to get the neighbours.
- **selection** (string with fixed values `forward`, `backward` or `all`. The values `all` is equivalent to the value `None` (default).) –
- **direction** (*int*) – The direction $0 \leq d < D$ in which we want to find the neighbours $k \pm e_d$.

Returns A list containing the pairs (d, k').

get_node_iterator(*mode='lex'*, *direction=None*)
Returns an iterator to iterate over all basis elements k.

Parameters

- **mode** (*string*) – The mode by which we iterate over the indices. Default is 'lex' for lexicographical order. Supported is also 'chain', for the chain-like mode, details see the manual.
- **direction** (*integer.*) – If iterating in *chainmode* this specifies the direction the chains go.

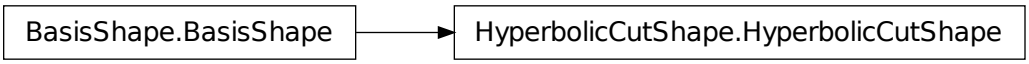
HyperbolicCutShape

About the `HyperbolicCutShape` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**HyperbolicCutShape**(D, K)
This class implements the hyperbolic cut basis shape which is a special type of sparse basis set. A basis shape is essentially all information and operations related to the set $\mathcal{K}$ of multi-indices k. The hyperbolic cut shape in D dimensions and with *sparsity* K is defined as the set

$$\mathcal{K}(D,K) := \{(k_0, \ldots, k_{D-1}) | k_d \geq 0 \forall d \in [0, \ldots, D-1] \wedge \prod_{d=0}^{D-1} (1 + k_d) \leq K\}$$

contains(k)
Checks if a given multi-index k is part of the basis set $\mathcal{K}$.

Parameters **k** (*tuple*) – The multi-index k we want to test.

extend()
Extend the basis shape such that (at least) all neighbours of all boundary nodes are included in the extended basis shape.

get_basis_size(*extended=False*)
Returns the size $|\mathcal{K}|$ of the basis. The size is the number of distinct multi-indices k that belong to the basis $\mathcal{K}$.

get_description()
Return a description of this basis shape object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current basis shape. A description never contains any data.

get_dimension()
Returns the dimension D of the basis shape $\mathcal{K}$. This is defined as the number of components each multi-index $k = (k_0, \ldots, k_{D-1})$ has.

get_limits()
Returns the upper limit K which is the same for all directions d.

Returns A tuple of the maximum of the multi-index in each direction.

get_neighbours(*k*, *selection=None*, *direction=None*)
Returns a list of all multi-indices that are neighbours of a given multi-index k. A direct neighbour is defined as $(k_0, \ldots, k_d \pm 1, \ldots, k_{D-1}) \forall d \in [0 \ldots D-1]$.

Parameters

- **k** (*tuple*) – The multi-index of which we want to get the neighbours.
- **selection** (string with fixed values `forward`, `backward` or `all`. The values `all` is equivalent to the value `None` (default).) –
- **direction** (*int*) – The direction $0 \leq d < D$ in which we want to find the neighbours $k \pm e_d$.

Returns A list containing the pairs (d, k').

get_node_iterator(*mode='lex'*, *direction=None*)
Returns an iterator to iterate over all basis elements $k \in \mathcal{K}$.

Parameters

- **mode** (*string*) – The mode by which we iterate over the indices. Default is `lex` for lexicographical order. Supported is also `chain`, for the chain-like mode, details see the manual.
- **direction** (*integer.*) – If iterating in *chainmode* this specifies the direction the chains go.

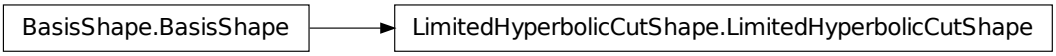
LimitedHyperbolicCutShape

About the `LimitedHyperbolicCutShape` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`LimitedHyperbolicCutShape`**(*D*, *K*, *limits*)

This class implements the hyperbolic cut basis shape which is a special type of sparse basis set. A basis shape is essentially all information and operations related to the set $\mathcal{K}$ of multi-indices k. The hyperbolic cut shape in D dimensions with *sparsity* S and limits $K = (K_0, \ldots, K_{D-1})$ is defined as the set

$$\mathcal{K}(D, S, K) := \{(k_0, \ldots, k_{D-1}) | 0 \leq k_d < K_d \forall d \in [0, \ldots, D-1] \wedge \prod_{d=0}^{D-1} (1 + k_d) \leq S\}$$

`contains`(*k*)

Checks if a given multi-index k is part of the basis set $\mathcal{K}$.

Parameters **k** (*tuple*) – The multi-index k we want to test.

`extend`(*tight=True*)

Extend the basis shape such that (at least) all neighbours of all boundary nodes are included in the extended basis shape.

Parameters **tight** – Whether to cut off the long tails.

Param Boolean, default is `False`

`get_basis_size`(*extended=False*)

Returns the size $|\mathcal{K}|$ of the basis. The size is the number of distinct multi-indices k that belong to the basis $\mathcal{K}$.

`get_description`()

Return a description of this basis shape object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current basis shape. A description never contains any data.

`get_dimension`()

Returns the dimension D of the basis shape $\mathcal{K}$. This is defined as the number of components each multi-index $k = (k_0, \ldots, k_{D-1})$ has.

`get_limits`()

Returns the upper limit K_d for all directions d.

Returns A tuple of the maximum of the multi-index in each direction.

`get_neighbours`(*k*, *selection=None*, *direction=None*)

Returns a list of all multi-indices that are neighbours of a given multi-index k. A direct neighbour is defined as $(k_0, \ldots, k_d \pm 1, \ldots, k_{D-1}) \forall d \in [0 \ldots D-1]$.

Parameters

- **k** (*tuple*) – The multi-index of which we want to get the neighbours.

- **selection** (string with fixed values `forward`, `backward` or `all`. The values `all` is equivalent to the value `None` (default).) –
- **direction** (*int*) – The direction $0 \leq d < D$ in which we want to find the neighbours $k \pm e_d$.

Returns A list containing the pairs (d, k').

get_node_iterator(*mode='lex'*, *direction=None*)

Returns an iterator to iterate over all basis elements $k \in \mathcal{K}$.

Parameters

- **mode** (*string*) – The mode by which we iterate over the indices. Default is `lex` for lexicographical order. Supported is also `chain`, for the chain-like mode, details see the manual.
- **direction** (*integer.*) – If iterating in *chainmode* this specifies the direction the chains go.

Wavepacket

About the `Wavepacket` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND`.**Wavepacket**(*parameters*)

This class is an abstract interface to wavepackets in general.

clone()

Clone the wavepacket. Return a new copy of the wavepacket and make sure that all references between the two wavepackets get broken.

Raises NotImplementedError Abstract interface.

gen_id()

Generate an (unique) ID per wavepacket instance.

get_dimension()

Returns The space dimension D of the wavepacket Ψ.

get_id()

Return the packet ID of this wavepacket instance. The ID may be used for storing packets in associative lists.

Returns The ID of the current instance.

get_number_components()

Returns The number N of components the wavepacket Ψ has.

set_id(*anid*)
Manually set a new ID for the current wavepacket instance.

Parameters **anid** (*int*) – The new ID.

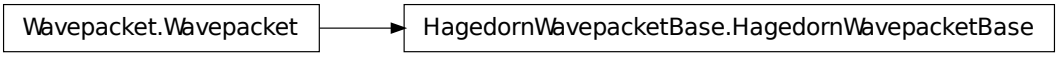
HagedornWavepacketBase

About the HagedornWavepacketBase class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**HagedornWavepacketBase**(*parameters*)
This class implements the abstract Wavepacket interface and contains code common to all types of Hagedorn wavepackets.

clone()
Clone the wavepacket. Return a new copy of the wavepacket and make sure that all references between the two wavepackets get broken.

Raises NotImplementedError Abstract interface.

gen_id()
Generate an (unique) ID per wavepacket instance.

get_basis_shape(*component=None*)
Retrieve the basis shapes $\mathcal{K}_i$ for each component i.

Parameters **component** (*int*) – The component i whose basis shape we request. (Default is None which means to return the basis shapes for all components.

Returns The basis shape for an individual component or a list with all shapes.

get_coefficient(*component*, *index*)
Retrieve a single coefficient c_k^i of the specified component Φ_i of :math:Psi‘.

Parameters

- **component** – The index i of the component Φ_i we want to update.

- **index** (A tuple of D integers.) – The multi-index k of the coefficient c_k^i we want to update.

Returns A single complex number.

get_coefficient_vector()
Retrieve the coefficients for all components Φ_i simultaneously.

Returns The coefficients c^i of all components Φ_i stacked into a single long column vector.

get_coefficients(*component=None*)
Returns the coefficients c^i for some component Φ_i of Ψ or all the coefficients c of all components.

Note: this method copies the data arrays.

Parameters **component** (int (Default is `None` meaning all)) – The index i of the component we want to retrieve.

Returns A single ndarray with the coefficients of the given component or a list containing the ndarrays for each component. Each ndarray is two-dimensional with a shape of $(|\mathcal{K}_i|, 1)$.

get_dimension()

Returns The space dimension D of the wavepacket Ψ.

get_eps()
Retrieve the semi-classical scaling parameter ε of the wavepacket.

Returns The value of ε.

get_id()
Return the packet ID of this wavepacket instance. The ID may be used for storing packets in associative lists.

Returns The ID of the current instance.

get_number_components()

Returns The number N of components the wavepacket Ψ has.

get_quadrature()
Return the `Quadrature` subclass instance used computing inner rpoducts and evaluating brakets.

Returns The current `Quadrature` subclass instance.

norm(*component=None*, *summed=False*)
Calculate the L^2 norm $\langle\Psi|\Psi\rangle$ of the wavepacket Ψ.

Parameters

- **component** (int or `None`.) – The index i of the component Φ_i whose norm is calculated. The default value is `None` which means to compute the norms of all N components.
- **summed** (Boolean, default is `False`.) – Whether to sum up the norms $\langle\Phi_i|\Phi_i\rangle$ of the individual components Φ_i.

Returns The norm of Ψ or the norm of Φ_i or a list with the N norms of all components. Depending on the values of `component` and `summed`.

set_basis_shape(*basis_shape*, *component=None*)
Set the basis shape $\mathcal{K}$ of a given component or for all components.

Parameters

- **basis_shape** (A subclass of `BasisShape`.) – The basis shape for an individual component or a list with all N shapes.

- **component** (*int*) – The component i whose basis shape we want to set. (Default is `None` which means to set the basis shapes for all components.

set_coefficient(*component*, *index*, *value*)
Set a single coefficient c_k^i of the specified component Φ_i of :math:Psi‘.

Parameters

- **component** – The index i of the component Φ_i we want to update.
- **index** (A tuple of D integers.) – The multi-index k of the coefficient c_k^i we want to update.
- **value** – The new value of the coefficient c_k^i.

Raises ValueError For invalid indices i or k.

set_coefficient_vector(*vector*)
Set the coefficients for all components Φ_i simultaneously.

Note: This function does *NOT* copy the input data! This is for efficiency as this routine is used in the innermost loops.

Parameters **vector** (*A two-dimensional ndarray of appropriate shape.*) – The coefficients of all components as a single long column vector.

set_coefficients(*values*, *component=None*)
Update all the coefficients c of Ψ or update the coefficients c^i of the components Φ_i only.

Note: this method copies the data arrays.

Parameters

- **values** (*An ndarray of suitable shape or a list of ndarrays.*) – The new values of the coefficients c^i of Φ_i.
- **component** (int (Default is `None` meaning all)) – The index i of the component we want to update with new coefficients.

Raises ValueError For invalid component indices i.

set_id(*anid*)
Manually set a new ID for the current wavepacket instance.

Parameters **anid** (*int*) – The new ID.

set_quadrature(*quadrature*)
Set the `Quadrature` subclass instance used for computing inner products and evaluating brakets.

Parameters **quadrature** – The new `Quadrature` subclass instance.

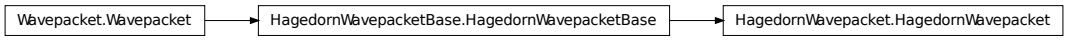
HagedornWavepacket

About the `HagedornWavepacket` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**HagedornWavepacket**(*dimension*, *ncomponents*, *eps*)

This class represents homogeneous vector valued Hagedorn wavepackets Ψ with N components in D space dimensions.

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__
x.__hash__() <==> hash(x)

__init__(*dimension*, *ncomponents*, *eps*)
Initialize a new homogeneous Hagedorn wavepacket.

Parameters

- **dimension** – The space dimension D the packet has.
- **ncomponents** – The number N of components the packet has.
- **eps** – The semi-classical scaling parameter ε of the basis functions.

Returns An instance of HagedornWavepacket.

static __new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
x.__repr__() <==> repr(x)

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__()

Returns A string describing the Hagedorn wavepacket Ψ.

static __subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

evaluate_at(*grid*, *component=None*, *prefactor=False*)
Evaluate the Hagedorn wavepacket Ψ at the given nodes γ.

Parameters

- **grid** (A class having a `get_nodes(...)()` method.) – The grid :math:Gamma‘ containing the nodes γ.
- **component** – The index i of a single component Φ_i to evaluate. (Defaults to `None` for evaluating all components.)
- **prefactor** (bool, default is `False`.) – Whether to include a factor of $\frac{1}{\sqrt{\det(Q)}}$.

Returns A list of arrays or a single array containing the values of the Φ_i at the nodes γ.

evaluate_basis_at(*grid*, *component*, *prefactor=False*)
Evaluate the basis functions ϕ_k recursively at the given nodes γ.

Parameters

- **grid** (A class having a `get_nodes(...)()` method.) – The grid :math:Gamma‘ containing the nodes γ.
- **component** – The index i of a single component Φ_i to evaluate. We need this to choose the correct basis shape.
- **prefactor** (bool, default is `False`.) – Whether to include a factor of $\frac{1}{\sqrt{\det(Q)}}$.

Returns A two-dimensional ndarray H of shape $(|\mathcal{K}_i|, |\Gamma|)$ where the entry $H[\mu(k), i]$ is the value of $\phi_k(\gamma_i)$.

gen_id()
Generate an (unique) ID per wavepacket instance.

get_basis_shape(*component=None*)
Retrieve the basis shapes $\mathcal{K}_i$ for each component i.

Parameters **component** (*int*) – The component i whose basis shape we request. (Default is `None` which means to return the basis shapes for all components.

Returns The basis shape for an individual component or a list with all shapes.

get_coefficient(*component*, *index*)
Retrieve a single coefficient c_k^i of the specified component Φ_i of :math:Psi‘.

Parameters

- **component** – The index i of the component Φ_i we want to update.
- **index** (A tuple of D integers.) – The multi-index k of the coefficient c_k^i we want to update.

Returns A single complex number.

get_coefficient_vector()
Retrieve the coefficients for all components Φ_i simultaneously.

Returns The coefficients c^i of all components Φ_i stacked into a single long column vector.

get_coefficients(*component=None*)
Returns the coefficients c^i for some component Φ_i of Ψ or all the coefficients c of all components.

Note: this method copies the data arrays.

Parameters **component** (int (Default is `None` meaning all)) – The index i of the component we want to retrieve.

Returns A single ndarray with the coefficients of the given component or a list containing the ndarrays for each component. Each ndarray is two-dimensional with a shape of $(|\mathcal{K}_i|, 1)$.

get_description()
Return a description of this wavepacket object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

get_dimension()

Returns The space dimension D of the wavepacket Ψ.

get_eps()
Retrieve the semi-classical scaling parameter ε of the wavepacket.

Returns The value of ε.

get_id()
Return the packet ID of this wavepacket instance. The ID may be used for storing packets in associative lists.

Returns The ID of the current instance.

get_number_components()

Returns The number N of components the wavepacket Ψ has.

get_parameters(*component=None*, *aslist=False*)
Get the Hagedorn parameter set Π of the wavepacket Ψ.

Parameters

- **component** – Dummy parameter for API compatibility with the inhomogeneous packets.
- **aslist** – Return a list of N parameter tuples. This is for API compatibility with inhomogeneous packets.

Returns The Hagedorn parameter set $\Pi = (q, p, Q, P, S)$ in this order.

get_quadrature()
Return the `Quadrature` subclass instance used computing inner rpoducts and evaluating brakets.

Returns The current `Quadrature` subclass instance.

norm(*component=None*, *summed=False*)
Calculate the L^2 norm $\langle\Psi|\Psi\rangle$ of the wavepacket Ψ.

Parameters

- **component** (int or `None`.) – The index i of the component Φ_i whose norm is calculated. The default value is `None` which means to compute the norms of all N components.
- **summed** (Boolean, default is `False`.) – Whether to sum up the norms $\langle\Phi_i|\Phi_i\rangle$ of the individual components Φ_i.

Returns The norm of Ψ or the norm of Φ_i or a list with the N norms of all components. Depending on the values of `component` and `summed`.

set_basis_shape(*basis_shape*, *component=None*)
Set the basis shape $\mathcal{K}$ of a given component or for all components.

Parameters

- **basis_shape** (A subclass of `BasisShape`.) – The basis shape for an individual component or a list with all N shapes.
- **component** (*int*) – The component i whose basis shape we want to set. (Default is `None` which means to set the basis shapes for all components.

set_coefficient(*component*, *index*, *value*)
Set a single coefficient c_k^i of the specified component Φ_i of :math:Psi‘.

Parameters

- **component** – The index i of the component Φ_i we want to update.
- **index** (A tuple of D integers.) – The multi-index k of the coefficient c_k^i we want to update.
- **value** – The new value of the coefficient c_k^i.

Raises ValueError For invalid indices i or k.

set_coefficient_vector(*vector*)
Set the coefficients for all components Φ_i simultaneously.

Note: This function does *NOT* copy the input data! This is for efficiency as this routine is used in the innermost loops.

Parameters **vector** (*A two-dimensional ndarray of appropriate shape.*) – The coefficients of all components as a single long column vector.

set_coefficients(*values*, *component=None*)
Update all the coefficients c of Ψ or update the coefficients c^i of the components Φ_i only.

Note: this method copies the data arrays.

Parameters

- **values** (*An ndarray of suitable shape or a list of ndarrays.*) – The new values of the coefficients c^i of Φ_i.
- **component** (int (Default is `None` meaning all)) – The index i of the component we want to update with new coefficients.

Raises ValueError For invalid component indices i.

set_id(*anid*)
Manually set a new ID for the current wavepacket instance.

Parameters **anid** (*int*) – The new ID.

set_parameters(*Pi*, *component=None*)
Set the Hagedorn parameters Π of the wavepacket Ψ.

Parameters

- **Pi** – The Hagedorn parameter set $\Pi = (q, p, Q, P, S)$ in this order.
- **component** – Dummy parameter for API compatibility with the inhomogeneous packets.

set_quadrature(*quadrature*)
Set the `Quadrature` subclass instance used for computing inner products and evaluating brakets.

Parameters **quadrature** – The new `Quadrature` subclass instance.

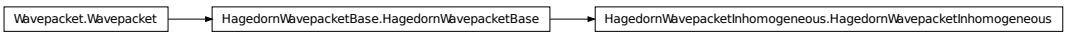
HagedornWavepacketInhomogeneous

About the `HagedornWavepacketInhomogeneous` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`HagedornWavepacketInhomogeneous`**(*dimension*, *ncomponents*, *eps*)

This class represents inhomogeneous vector valued Hagedorn wavepackets Ψ with N components in D space dimensions.

`__delattr__`
x.__delattr__('name') <==> del x.name

`__format__`()
default object formatter

`__getattribute__`
x.__getattribute__('name') <==> x.name

`__hash__`
x.__hash__() <==> hash(x)

`__init__`(*dimension*, *ncomponents*, *eps*)
Initialize a new in homogeneous Hagedorn wavepacket.

Parameters

- **dimension** – The space dimension D the packet has.
- **ncomponents** – The number N of components the packet has.
- **eps** – The semi-classical scaling parameter ε of the basis functions.

Returns An instance of `HagedornWavepacketInhomogeneous`.

static `__new__`(*S*, ...) → a new object with type S, a subtype of T

`__reduce__`()
helper for pickle

`__reduce_ex__`()
helper for pickle

`__repr__`
x.__repr__() <==> repr(x)

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__()

Returns A string describing the Hagedorn wavepacket Ψ.

static __subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

evaluate_at(*grid*, *component=None*, *prefactor=False*)
Evaluate the Hagedorn wavepacket Ψ at the given nodes γ.

Parameters

- **grid** (A class having a `get_nodes(...)()` method.) – The grid :math:Gamma' containing the nodes γ.
- **component** – The index i of a single component Φ_i to evaluate. (Defaults to `None` for evaluating all components.)
- **prefactor** (bool, default is `False`.) – Whether to include a factor of $\frac{1}{\sqrt{\det(Q)}}$.

Returns A list of arrays or a single array containing the values of the Φ_i at the nodes γ.

evaluate_basis_at(*grid*, *component*, *prefactor=False*)
Evaluate the basis functions ϕ_k recursively at the given nodes γ.

Parameters

- **grid** (A class having a `get_nodes(...)()` method.) – The grid :math:Gamma' containing the nodes γ.
- **component** – The index i of a single component Φ_i to evaluate.
- **prefactor** (bool, default is `False`.) – Whether to include a factor of $\frac{1}{\sqrt{\det(Q)}}$.

Returns A two-dimensional ndarray H of shape $(|\mathcal{K}_i|, |\Gamma|)$ where the entry $H[\mu(k), i]$ is the value of $\phi_k(\gamma_i)$.

gen_id()
Generate an (unique) ID per wavepacket instance.

get_basis_shape(*component=None*)
Retrieve the basis shapes $\mathcal{K}_i$ for each component i.

Parameters **component** (*int*) – The component i whose basis shape we request. (Default is `None` which means to return the basis shapes for all components.

Returns The basis shape for an individual component or a list with all shapes.

get_coefficient(*component*, *index*)
Retrieve a single coefficient c_k^i of the specified component Φ_i of :math:Psi'.

Parameters

- **component** – The index i of the component Φ_i we want to update.
- **index** (A tuple of D integers.) – The multi-index k of the coefficient c_k^i we want to update.

Returns A single complex number.

get_coefficient_vector()
Retrieve the coefficients for all components Φ_i simultaneously.

Returns The coefficients c^i of all components Φ_i stacked into a single long column vector.

get_coefficients(*component=None*)
Returns the coefficients c^i for some component Φ_i of Ψ or all the coefficients c of all components.

Note: this method copies the data arrays.

Parameters **component** (int (Default is `None` meaning all)) – The index i of the component we want to retrieve.

Returns A single ndarray with the coefficients of the given component or a list containing the ndarrays for each component. Each ndarray is two-dimensional with a shape of $(|\mathcal{K}_i|, 1)$.

get_description()
Return a description of this wavepacket object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

get_dimension()

Returns The space dimension D of the wavepacket Ψ.

get_eps()
Retrieve the semi-classical scaling parameter ε of the wavepacket.

Returns The value of ε.

get_id()
Return the packet ID of this wavepacket instance. The ID may be used for storing packets in associative lists.

Returns The ID of the current instance.

get_number_components()

Returns The number N of components the wavepacket Ψ has.

get_parameters(*component=None*, *aslist=False*)
Get the Hagedorn parameter set Π_i of each component :math‘Phi_i‘ of the wavepacket Ψ.

Parameters

- **component** – The index i of the component Φ_i whose parameters Π_i we want to get.
- **aslist** – Dummy parameter for API compatibility with the homogeneous packets.

Returns A list with all parameter sets Π_i or a single parameter set. The parameters $\Pi_i = (q_i, p_i, Q_i, P_i, S_i)$ are always in this order.

get_quadrature()
Return the `Quadrature` subclass instance used computing inner rpoducts and evaluating brakets.

Returns The current `Quadrature` subclass instance.

norm(*component=None*, *summed=False*)
Calculate the L^2 norm $\langle\Psi|\Psi\rangle$ of the wavepacket Ψ.

Parameters

- **component** (int or `None`.) – The index i of the component Φ_i whose norm is calculated. The default value is `None` which means to compute the norms of all N components.
- **summed** (Boolean, default is `False`.) – Whether to sum up the norms $\langle\Phi_i|\Phi_i\rangle$ of the individual components Φ_i.

Returns The norm of Ψ or the norm of Φ_i or a list with the N norms of all components. Depending on the values of `component` and `summed`.

set_basis_shape(*basis_shape*, *component=None*)
Set the basis shape $\mathcal{K}$ of a given component or for all components.

Parameters

- **basis_shape** (A subclass of `BasisShape`.) – The basis shape for an individual component or a list with all N shapes.
- **component** (*int*) – The component i whose basis shape we want to set. (Default is `None` which means to set the basis shapes for all components.

set_coefficient(*component*, *index*, *value*)
Set a single coefficient c_k^i of the specified component Φ_i of :math:Psi‘.

Parameters

- **component** – The index i of the component Φ_i we want to update.
- **index** (A tuple of D integers.) – The multi-index k of the coefficient c_k^i we want to update.
- **value** – The new value of the coefficient c_k^i.

Raises ValueError For invalid indices i or k.

set_coefficient_vector(*vector*)
Set the coefficients for all components Φ_i simultaneously.

Note: This function does *NOT* copy the input data! This is for efficiency as this routine is used in the innermost loops.

Parameters **vector** (*A two-dimensional ndarray of appropriate shape.*) – The coefficients of all components as a single long column vector.

set_coefficients(*values*, *component=None*)
Update all the coefficients c of Ψ or update the coefficients c^i of the components Φ_i only.

Note: this method copies the data arrays.

Parameters

- **values** (*An ndarray of suitable shape or a list of ndarrays.*) – The new values of the coefficients c^i of Φ_i.
- **component** (int (Default is `None` meaning all)) – The index i of the component we want to update with new coefficients.

Raises ValueError For invalid component indices i.

set_id(*anid*)
Manually set a new ID for the current wavepacket instance.

Parameters **anid** (*int*) – The new ID.

set_parameters(*Pi*, *component=None*)
Set the Hagedorn parameter set Π_i of each component :math‘Phi_i‘ of the wavepacket Ψ.

Parameters

- **Pi** (*A single tuple or a list of tuples*) – The parameter sets $\Pi_i = (q_i, p_i, Q_i, P_i, S_i)$ with its values in this order.
- **component** – The index i of the component Φ_i whose parameters Π_i we want to update.

set_quadrature(*quadrature*)
Set the Quadrature subclass instance used for computing inner products and evaluating brakets.

Parameters **quadrature** – The new Quadrature subclass instance.

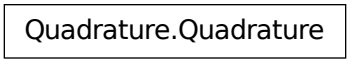
Quadrature

About the Quadrature class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**Quadrature**
This class is an abstract interface to quadratures in general.

build_matrix()
Calculate the matrix elements of $\langle\Psi|f|\Psi\rangle$ for a general function $f(x)$ with $x \in \mathbb{R}^D$. Note that the arguments may vary through subclasses!

Raises **NotImplementedError** Abstract interface.

get_description()
Return a description of this quadrature object. A description is a dict containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

get_qr()
Return the QuadratureRule subclass instance used for quadrature.

Returns The current instance of the quadrature rule.

quadrature()
Performs the quadrature of $\langle\Psi|f|\Psi\rangle$ for a general function $f(x)$ with $x \in \mathbb{R}^D$. Note that the arguments may vary through subclasses!

Raises **NotImplementedError** Abstract interface.

set_qr(*QR*)
Set the QuadratureRule subclass instance used for quadrature.

Parameters **QR** – The new QuadratureRule instance.

transform_nodes()
Transform the quadrature nodes such that they fit the given wavepacket. Note that the arguments may vary through subclasses!

Raises **NotImplementedError** Abstract interface.

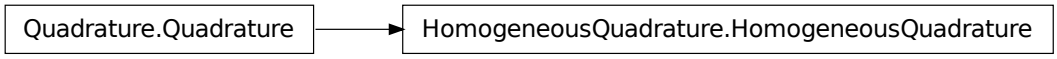
HomogeneousQuadrature

About the HomogeneousQuadrature class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**HomogeneousQuadrature**(*QR=None*)

build_matrix(*packet*, *operator=None*)
Calculate the matrix elements of $\langle\Psi|f|\Psi\rangle$ for a general function $f(x)$ with $x \in \mathbb{R}^D$. The matrix is computed without including the coefficients c_k^i.

Parameters

- **packet** – The wavepacket Ψ.
- **operator** – A matrix-valued function $f(q, x) : \mathbb{R} \times \mathbb{R}^D \to \mathbb{R}^{N \times N}$.

Returns A square matrix of size $\sum_i^N |\mathcal{K}_i| \times \sum_j^N |\mathcal{K}_j|$.

get_description()
Return a description of this quadrature object. A description is a dict containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

get_qr()
Return the QuadratureRule subclass instance used for quadrature.

Returns The current instance of the quadrature rule.

quadrature(*packet*, *operator=None*, *summed=False*, *component=None*, *diag_component=None*)
Performs the quadrature of $\langle\Psi|f|\Psi\rangle$ for a general function $f(x)$ with $x \in \mathbb{R}^D$.

Parameters

- **packet** – The wavepacket Ψ.
- **operator** – A matrix-valued function $f(x) : \mathbb{R}^D \to \mathbb{R}^{N \times N}$.
- **summed** (bool, default is `False`.) – Whether to sum up the individual integrals $\langle \Phi_i | f_{i,j} | \Phi_j \rangle$.
- **component** – Request only the i-th component of the result. Remember that $i \in [0, N^2 - 1]$.
- **diag_component** – Request only the i-th component from the diagonal entries, here $i \in [0, N - 1]$. Note that `component` takes precedence over `diag_component` if both are supplied. (Which is discouraged)

Returns The value of the braket $\langle \Psi | f | \Psi \rangle$. This is either a scalar value or a list of N^2 scalar elements depending on the value of `summed`.

set_qr(*QR*)
Set the `QuadratureRule` subclass instance used for quadrature.

Parameters **QR** – The new `QuadratureRule` instance.

transform_nodes(*Pi*, *eps*, *QR=None*)
Transform the quadrature nodes γ such that they fit the given wavepacket Φ [Π].

Parameters

- **Pi** – The parameter set Π of the wavepacket.
- **eps** – The value of ε of the wavepacket.
- **QR** – An optional quadrature rule $\Gamma = (\gamma, \omega)$ providing the nodes. If not given the internal quadrature rule will be used.

Returns A two-dimensional ndarray of shape $(D, |\Gamma|)$ where $|\Gamma|$ denotes the total number of quadrature nodes.

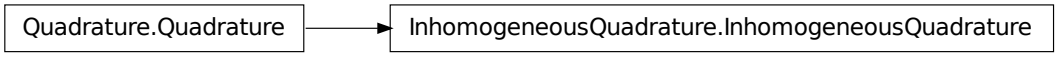
InhomogeneousQuadrature

About the `InhomogeneousQuadrature` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND`.**InhomogeneousQuadrature**(*QR=None*)

build_matrix(*pacbra*, *packet=None*, *operator=None*)
Calculate the matrix elements of $\langle\Psi|f|\Psi'\rangle$ for a general function $f(x)$ with $x \in \mathbb{R}^D$. The matrix is computed without including the coefficients c_k^i.

Parameters

- **pacbra** – The wavepacket Ψ from the bra with N components.
- **packet** – The wavepacket Ψ' from the ket with N' components.
- **operator** – A matrix-valued function $f(q, x) : \mathbb{R} \times \mathbb{R}^D \to \mathbb{R}^{N \times N'}$.

Returns A matrix of size $\sum_i^N |\mathcal{K}_i| \times \sum_j^{N'} |\mathcal{K}'_j|$.

get_description()
Return a description of this quadrature object. A description is a `dict` containing all key-value pairs necessary to reconstruct the current instance. A description never contains any data.

get_qr()
Return the `QuadratureRule` subclass instance used for quadrature.

Returns The current instance of the quadrature rule.

mix_parameters(*Pibra*, *Piket*)
Mix the two parameter sets Π_i and Π_j from the bra and the ket wavepackets $\Phi[\Pi_i]$ and $\Phi'[\Pi_j]$.

Parameters

- **Pibra** – The parameter set Π_i from the bra part wavepacket.
- **Piket** – The parameter set Π_j from the ket part wavepacket.

Returns The mixed parameters q_0 and Q_S. (See the theory for details.)

quadrature(*pacbra*, *packet=None*, *operator=None*, *summed=False*, *component=None*, *diag_component=None*)
Performs the quadrature of $\langle\Psi|f|\Psi'\rangle$ for a general function $f(x)$ with $x \in \mathbb{R}^D$.

Parameters

- **pacbra** – The wavepacket Ψ from the bra with N components.
- **packet** – The wavepacket Ψ' from the ket with N' components.
- **operator** – A matrix-valued function $f(x) : \mathbb{R}^D \to \mathbb{R}^{N \times N'}$.
- **summed** (bool, default is `False`.) – Whether to sum up the individual integrals $\langle\Phi_i|f_{i,j}|\Phi'_j\rangle$.
- **component** – Request only the i-th component of the result. Remember that $i \in [0, N \cdot N' - 1]$.
- **diag_component** – Request only the i-th component from the diagonal entries, here $i \in [0, N' - 1]$. Note that `component` takes precedence over `diag_component` if both are supplied. (Which is discouraged)

Returns The value of the braket $\langle\Psi|f|\Psi'\rangle$. This is either a scalar value or a list of $N \cdot N'$ scalar elements depending on the value of `summed`.

set_qr(*QR*)
Set the `QuadratureRule` subclass instance used for quadrature.

Parameters **QR** – The new `QuadratureRule` instance.

transform_nodes(*Pibra*, *Piket*, *eps*, *QR=None*)
Transform the quadrature nodes γ such that they fit the given wavepackets $\Phi[\Pi_i]$ and $\Phi'[\Pi_j]$ best.

Parameters

- **Pibra** – The parameter set Π_i from the bra part wavepacket.
- **Piket** – The parameter set Π_j from the ket part wavepacket.
- **eps** – The value of ε of the wavepacket.
- **QR** – An optional quadrature rule $\Gamma = (\gamma, \omega)$ providing the nodes. If not given the internal quadrature rule will be used.

Returns A two-dimensional ndarray of shape $(D, |\Gamma|)$ where $|\Gamma|$ denotes the total number of quadrature nodes.

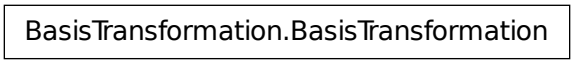
BasisTransformation

About the `BasisTransformation` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`BasisTransformation`**(*potential*)

This class is the interface definition for general basis transformation procedures. The transformation switches between the canonical basis of the potential $V(x)$ and its eigenbasis $\Lambda(x)$ consisting of the energy levels $\lambda_i(x)$ with $i \in [0, \ldots, N]$.

`__delattr__`
x.__delattr__('name') <==> del x.name

`__format__`()
default object formatter

`__getattribute__`
x.__getattribute__('name') <==> x.name

`__hash__`
x.__hash__() <==> hash(x)

`__init__`(*potential*)
Create a new `BasisTransformation` instance for a given potential matrix $V(x)$.

Parameters **potential** (A `MatrixPotential` instance.) – The potential underlying the basis transformation.

static **__new__** (*S*, ...) → a new object with type S, a subtype of T

__reduce__ ()
helper for pickle

__reduce_ex__ ()
helper for pickle

__repr__
x.__repr__() <==> repr(x)

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__ () → int
size of object in memory, in bytes

__str__
x.__str__() <==> str(x)

static **__subclasshook__** ()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

transform_to_canonical (*transformable*)
Do nothing, implement an identity transformation.

transform_to_eigen (*transformable*)
Do nothing, implement an identity transformation.

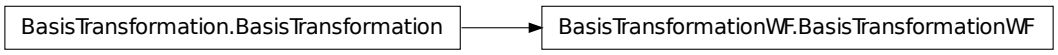
BasisTransformationWF

About the `BasisTransformationWF` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**BasisTransformationWF**(*potential*, *grid=None*)

This class implements basis transformations of wavefunctions $\psi(x)$ between the canonical basis of and the basis $\Lambda(x)$ spanned by the eigenvectors $\nu_i(x)$ of the potential $V(x)$.

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__
x.__hash__() <==> hash(x)

__init__(*potential*, *grid=None*)
Create a new BasisTransformation instance for a given potential matrix $V(x)$.

Parameters

- **potential** (A MatrixPotential instance.) – The potential underlying the basis transformation.
- **grid** (A Grid subclass instance.) – The grid.

static __new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
x.__repr__() <==> repr(x)

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__
x.__str__() <==> str(x)

static __subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

set_grid(*grid*)
Set the grid Γ containing the nodes γ on which the wavefunction ψ was evaluated. The N eigenvectors ν_i will be evaluated on the same grid nodes.

Parameters **grid** (A `Grid` subclass instance.) – The grid

transform_to_canonical(*wavefunction*)
Transform the evaluated wavefunction $\psi(\Gamma)$ given in the eigenbasis to the canonical basis.

Parameters **wavefunction** (A `WaveFunction` instance.) – The wavefunction to transform.

Returns Another `WaveFunction` instance containing the transformed wavefunction $\psi'(\Gamma)$.

transform_to_eigen(*wavefunction*)
Transform the evaluated wavefunction $\psi'(\Gamma)$ given in the canonical basis to the eigenbasis.

Parameters **wavefunction** (A `WaveFunction` instance.) – The wavefunction to transform.

Returns Another `WaveFunction` instance containing the transformed wavefunction $\psi(\Gamma)$.

BasisTransformationHAWP

About the BasisTransformationHAWP class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND`.**BasisTransformationHAWP**(*potential*, *builder=None*)
This class implements basis transformations of Hagedorn wavepackets $\Psi(x)$ between the canonical basis of and the basis $\Lambda(x)$ spanned by the eigenvectors $\nu_i(x)$ of the potential $V(x)$.

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__
x.__hash__() <==> hash(x)

__init__ (*potential*, *builder=None*)
Create a new BasisTransformationHAWP instance for a given potential matrix $V(x)$.

Parameters

- **potential** (A MatrixPotential instance.) – The potential underlying the basis transformation.
- **builder** (A Quadrature subclass instance.) – An object that can compute this matrix.

static __new__ (*S*, ...) → a new object with type S, a subtype of T

__reduce__ ()
helper for pickle

__reduce_ex__ ()
helper for pickle

__repr__
x.__repr__() <==> repr(x)

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__ () → int
size of object in memory, in bytes

__str__
x.__str__() <==> str(x)

static __subclasshook__ ()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

set_matrix_builder (*builder*)
Set the matrix builder. It is responsible for computing the matrix elements $\langle\phi_i|V_{i,j}|\phi_j\rangle$. This matrix is used during the basis transformation.

Parameters **builder** (A Quadrature subclass instance.) – An object that can compute this matrix.

transform_to_canonical (*wavepacket*)
Transform the wavepacket Ψ given in the eigenbasis to the canonical basis.

Note that this method acts destructively on the given Wavepacket instance. If this is not desired, clone the packet before handing it over to this method.

Parameters **wavepacket** (A Wavepacket subclass instance.) – The Hagedorn wavepacket to transform.

Returns Another Wavepacket instance containing the transformed wavepacket Ψ'.

transform_to_eigen(*wavepacket*)
Transform the wavepacket Ψ' given in the canonical basis to the eigenbasis.

Note that this method acts destructively on the given Wavepacket instance. If this is not desired, clone the packet before handing it over to this method.

Parameters **wavepacket** (A Wavepacket subclass instance.) – The Hagedorn wavepacket to transform.

Returns Another Wavepacket instance containing the transformed wavepacket Ψ.

1.1.4 Observables

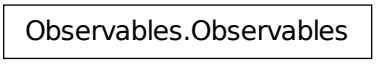
Observables

About the Observables class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**Observables**
This class is the interface definition for general observable computation procedures.

kinetic_energy(*ket*, *T*)
Compute the kinetic energy $E_{\text{kin}} := \langle\psi|T|\psi\rangle$.

Parameters

- **ket** – The object denoted by ψ.
- **T** – The kinetic energy operator T.

Raises NotImplementedError Abstract interface.

potential_energy(*ket*, *potential*)
Compute the potential energy $E_{\text{pot}} := \langle\psi|V|\psi\rangle$.

Parameters

- **ket** – The object denoted by ψ.
- **potential** – The potential $V(x)$.

Raises NotImplementedError Abstract interface.

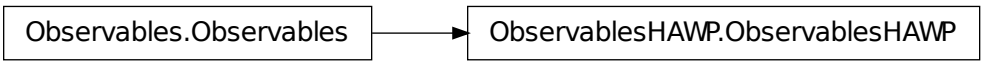
ObservablesHAWP

About the `ObservablesHAWP` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`ObservablesHAWP`**(*quadrature=None*)

This class implements observable computation for Hagedorn wavepackets Ψ.

`apply_gradient`(*wavepacket*, *component*)

Compute the effect of the gradient operator $-i\varepsilon^2\nabla_x$ on the basis functions $\phi(x)$ of a component Φ_i of the Hagedorn wavepacket Ψ.

Parameters

- **wavepacket** (A `HagedornWavepacketBase` subclass instance.) – The wavepacket Ψ containing Φ_i.
- **component** (*int*) – The index i of the component Φ_i.

Returns Extended basis shape $\dot{\mathcal{K}}$ and new coefficients c'.

`kinetic_energy`(*wavepacket*, *component=None*, *summed=False*)

Compute the kinetic energy $E_{\text{kin}} := \langle\Psi|T|\Psi\rangle$ of the different components Φ_i of the wavepacket Ψ.

Parameters

- **wavepacket** (A `HagedornWavepacketBase` subclass instance.) – The wavepacket Ψ of which we compute the kinetic energy.
- **component** (Integer or `None`.) – The index i of the component Φ_i whose kinetic energy we want to compute. If set to `None` the computation is performed for all N components.
- **summed** (*Boolean*) – Whether to sum up the kinetic energies E_i of the individual components Φ_i. Default is *False*.

Returns A list with the kinetic energies of the individual components or the overall kinetic energy of the wavepacket. (Depending on the optional arguments.)

potential_energy (*wavepacket*, *potential*, *component=None*, *summed=False*)
Compute the potential energy $E_{\text{pot}} := \langle\Psi|V|\Psi\rangle$ of the different components Φ_i of the wavepacket Ψ.

Parameters

- **wavepacket** (A `HagedornWavepacketBase` subclass instance.) – The wavepacket Ψ of which we compute the potential energy.
- **potential** – The potential $V(x)$.
- **component** (Integer or `None`.) – The index i of the component Φ_i whose potential energy we want to compute. If set to `None` the computation is performed for all N components.
- **summed** (*Boolean*) – Whether to sum up the potential energies E_i of the individual components Φ_i. Default is *False*.

Returns A list with the potential energies of the individual components or the overall potential energy of the wavepacket. (Depending on the optional arguments.)

set_quadrature (*quadrature*)
Set the quadrature.

Parameters **quadrature** (A `Quadrature` subclass instance.) – A quadrature for computing the integrals. Quadrature is only used for the computation of the potential energy $\langle\Psi|V(x)|\Psi\rangle$ but not for the kinetic energy.

1.1.5 Time propagation

KineticOperator

About the `KineticOperator` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND.`**KineticOperator** (*grid*, *eps=None*)
This class represents the kinetic operator T in Fourier space.

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__
x.__hash__() <==> hash(x)

__init__(*grid*, *eps=None*)
Compute the Fourier transformation of the position space representation of the kinetic operator T.

Parameters

- **grid** – The position space grid Γ of which we compute its Fourier transform Ω.
- **eps** – The semi-classical scaling parameter ε. (optional)

static __new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
x.__repr__() <==> repr(x)

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__
x.__str__() <==> str(x)

static __subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

calculate_exponential(*factor=1.0*)
Calculate the exponential $\exp(\alpha\underline{\omega} \cdot \underline{\omega})$ used in the Strang splitting.

Parameters **factor** – The prefactor α. It defaults to 1 but is ususally set to $-\frac{i}{2}\varepsilon^2\tau$ by the caller.

calculate_operator(*eps=None*)
Calculate the kinetic operator $\hat{T} = \frac{\varepsilon^4}{2}\underline{\omega} \cdot \underline{\omega}$ in Fourier space.

Parameters **eps** – The semi-classical scaling parameter ε. It has to be given here or during the initialization of the current instance.

evaluate_at(*grid=None*)
Evaluate the kinetic operator $\hat{T} = \frac{\varepsilon^4}{2}\underline{\omega} \cdot \underline{\omega}$ in Fourier space. This returns an numpy `ndarray` by using a specific $\underline{\omega}$.

Parameters **grid** – Unused dummy parameter.

evaluate_exponential_at(*grid=None*)
Evaluate the exponential $\exp(\alpha\hat{T})$ in Fourier space. This returns a numpy `ndarray` by using a specific $\underline{\omega}$. The factor α can be set by the corresponding method.

Parameters **grid** – Unused dummy parameter.

get_fourier_grid_axes()
Return the grid axes of the Fourier space grid Ω.

Propagator

About the `Propagator` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND`.**Propagator**
Propagators can numerically simulate the time evolution of quantum states $\psi(x,t)$ as described by the time-dependent Schroedinger equation

$$i\varepsilon^2\frac{\partial}{\partial t}\psi(x,t) = H\psi(x,t)$$

where the semi-classical scaling parameter $\varepsilon > 0$ is already included. The Hamiltonian operator H is defined as

$$H = T + V(x) = -\frac{\varepsilon^4}{2}\Delta + V(x)$$

get_number_components()

Returns The number N components of $\psi(x,t)$.

Raises NotImplementedError This is an abstract base class.

get_potential()
Returns the potential $V(x)$ used for time propagation.

Returns A `MatrixPotential` subclass instance.

propagate()
Given the wavefunction ψ at time t, calculate the new :math:psi‘ at time $t + \tau$. We do exactly one timestep of size τ here.

Raises NotImplementedError This is an abstract base class.

FourierPropagator

About the `FourierPropagator` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND`.**FourierPropagator**(*potential*, *initial_values*, *para*)
This class can numerically propagate given initial values $\Psi(x_0, t_0)$ on a potential hypersurface $V(x)$. The propagation is done with a Strang splitting of the time propagation operator $\exp(-\frac{i}{\varepsilon^2}\tau H)$.

get_number_components()
Get the number N of components of Ψ.

Returns The number N.

get_operators()
Get the kinetic and potential operators $T(\Omega)$ and $V(\Gamma)$.

Returns A tuple (T, V) containing two `ndarrays`.

get_potential()
Returns the potential $V(x)$ used for time propagation.

Returns A `MatrixPotential` subclass instance.

get_wavefunction()
Get the wavefunction that stores the current data $\Psi(\Gamma)$.

Returns The `WaveFunction` instance.

propagate()
Given the wavefunction values $\Psi(\Gamma)$ at time t, calculate new values $\Psi'(\Gamma)$ at time $t + \tau$. We perform exactly one single timestep of size τ within this function.

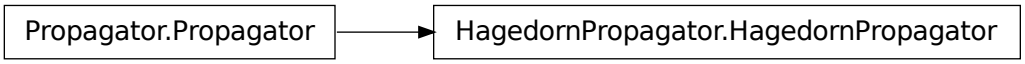
HagedornPropagator

About the `HagedornPropagator` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`HagedornPropagator`**(*parameters*, *potential*, *packets*=[ ])

This class can numerically propagate given initial values Ψ in a potential $V(x)$. The propagation is done for a given set of homogeneous Hagedorn wavepackets neglecting interaction.

`add_wavepacket`(*packet*)

Add a new wavepacket Ψ to the list of propagated wavepackets.

Parameters **packet** (A tuple (Ψ, χ) with Ψ a `HagedornWavepacket` instance and χ an integer.) – The new wavepacket Ψ and its leading component $\chi \in [0, N-1]$.

`get_number_components`()

Returns The number N of components Φ_i of Ψ.

`get_potential`()

Returns the potential $V(x)$ used for time propagation.

Returns A `MatrixPotential` subclass instance.

`get_wavepackets`(*packet=None*)

Return the wavepackets $\{\Psi_i\}_i$ that take part in the time propagation by the current `HagedornPropagator` instance.

Parameters **packet** (Integer or `None`) – The index i (in this list) of a single packet Ψ_i that is to be returned. If set to `None` (default) return the full list with all packets.

Returns A list of `HagedornWavepacket` instances or a single instance.

`propagate`()

Given a wavepacket Ψ at time t compute the propagated wavepacket at time $t + \tau$. We perform exactly one timestep of size τ here. This propagation is done for all packets in the list $\{\Psi_i\}_i$ and neglects any interaction between two packets.

`set_wavepackets`(*packetlist*)

Set the list $\{\Psi_i\}_i$ of wavepackets that the propagator will propagate.

Parameters **packetlist** (A list of (Ψ_i, χ_i) tuples.) – A list of new wavepackets Ψ_i and their leading components χ_i to propagate.

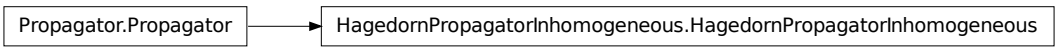
HagedornPropagatorInhomogeneous

About the `HagedornPropagatorInhomogeneous` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

`WaveBlocksND.`**`HagedornPropagatorInhomogeneous`**
alias of `WaveBlocksND.HagedornPropagatorInhomogeneous`

SimulationLoop

About the `SimulationLoop` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class `WaveBlocksND.`**`SimulationLoop`**(*parameters*)
This class acts as the main simulation loop. It owns a propagator that propagates a set of initial values during a

time evolution. It is responsible for preparing the simulation, setting up initial values and store the simulation data with the help of an `IOManager` instance.

end_simulation()
Do the necessary cleanup after a simulation. For example request the IOManager to write the data and close the output files. Shut down the simulation process. :raise NotImplementedError: This is an abstract base class.

prepare_simulation()
Set up a propagator for the simulation loop. Set the potential and initial values according to the configuration. :raise NotImplementedError: This is an abstract base class.

run_simulation()
Run the simulation. This method will implement the central loop running over all timesteps. Inside of this loop it will call the *propagate* method of its propagator and save the simulation results. :raise NotImplementedError: This is an abstract base class.

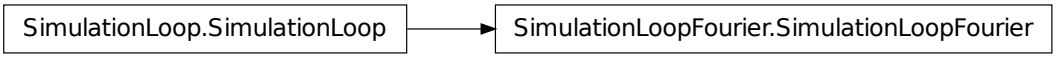
SimulationLoopFourier

About the `SimulationLoopFourier` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**SimulationLoopFourier**(*parameters*)
This class acts as the main simulation loop. It owns a propagator that propagates a set of initial values during a time evolution.

end_simulation()
Do the necessary cleanup after a simulation. For example request the `IOManager` to write the data and close the output files.

prepare_simulation()
Set up a Fourier propagator for the simulation loop. Set the potential and initial values according to the configuration.

Raises ValueError For invalid or missing input data.

run_simulation()
Run the simulation loop for a number of time steps.

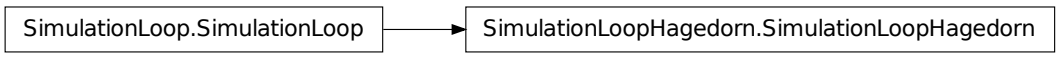
SimulationLoopHagedorn

About the `SimulationLoopHagedorn` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class `WaveBlocksND.`**`SimulationLoopHagedorn`**(*parameters*)

This class acts as the main simulation loop. It owns a propagator that propagates a set of initial values during a time evolution.

`end_simulation`()

Do the necessary cleanup after a simulation. For example request the `IOManager` to write the data and close the output files.

`prepare_simulation`()

Set up a Hagedorn propagator for the simulation loop. Set the potential and initial values according to the configuration.

Raises ValueError For invalid or missing input data.

`run_simulation`()

Run the simulation loop for a number of time steps.

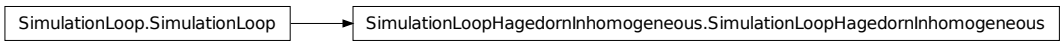
SimulationLoopHagedornInhomogeneous

About the `SimulationLoopHagedornInhomogeneous` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**SimulationLoopHagedornInhomogeneous**(*parameters*)

This class acts as the main simulation loop. It owns a propagator that propagates a set of initial values during a time evolution.

end_simulation()

Do the necessary cleanup after a simulation. For example request the IOManager to write the data and close the output files.

prepare_simulation()

Set up a Hagedorn propagator for the simulation loop. Set the potential and initial values according to the configuration.

Raises ValueError For invalid or missing input data.

run_simulation()

Run the simulation loop for a number of time steps.

1.1.6 Simulation result storage I/O

IOManager

About the IOManager class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class WaveBlocksND.**IOManager**

An IOManager class that can save various simulation results into data files. For storing the data we use the well established HDF5 file format. An IOManager instace abstracts the input and output operations and translates requests into low-level operations.

create_block(*blockid=None*, *groupid='global'*)

Create a data block with the specified block ID. Each data block can store several chunks of information, and there can be an arbitrary number of data blocks per file.

Parameters **blockid** – The ID for the new data block. If not given the blockid will be choosen automatically. The block ID has to be unique.

Returns The block ID of the created block.

create_file(*parameters*, *filename='simulation_results.hdf5'*)

Set up a new :py:class'IOManager' instance. The output file is created and opened.

Parameters

- **parameters** – A ParameterProvider instance containing the current simulation parameters. This is only used for determining the size of new data sets.
- **filename** – The filename (optionally with filepath) of the file we try to create. If not given the default value from *GlobalDefaults* is used.

create_group(*groupid=None*)

Create a data group with the specified group ID. Each data group can contain an arbitrary number of data blocks, and there can be an arbitrary number of data groups per file.

Parameters **groupid** – The ID for the new data group. If not given the group ID will be choosen automatically. The group ID has to be unique.

Returns The group ID of the created group.

finalize()

Close the open output file and reset the internal information.

find_timestep_index(*timegridpath*, *timestep*)

Lookup the index for a given timestep. This assumes the timegrid array is strictly monotone.

get_block_ids(*groupid=None*, *grouped=False*)

Return a list containing the IDs for all blocks in the current file structure.

Parameters

- **groupid** – An optional group ID. If given we return only block IDs for blocks which are a member of this group. If it is *None* we return all block IDs.
- **grouped** – If *True* we group the block IDs by their group into lists. This option is only relevant in case the *groupid* is not given.

get_group_ids(*exclude=[]*)

Return a list containing the IDs for all groups in the current file structure.

Parameters **exclude** – A list of group IDs to exclude. Per default no group is excluded.

get_group_of_block(*blockid*)

Return the ID of the group a given block belongs to or *None* if there is no such data block.

Parameters **blockid** – The ID of the given block.

get_number_blocks(*groupid=None*)
Return the number of data blocks in the current file structure.

Parameters **groupid** – An optional group ID. If given we count only data blocks which are a member of this group. If it is *None* (default) we count all data blocks.

get_number_groups()
Return the number of data block groups in the current file structure.

must_resize(*path*, *slot*, *axis=0*)
Check if we must resize a given dataset and if yes, resize it.

open_file(*filename='simulation_results.hdf5'*)
Load a given file that contains the results from another simulation.

Parameters **filename** – The filename (optionally with filepath) of the file we try to load. If not given the default value from *GlobalDefaults* is used.

split_data(*data*, *axis*)
Split a multi-dimensional data block into slabs along a given axis.

Parameters

- **data** – The data tensor given.
- **axis** – The axis along which to split the data.

Returns A list of slices.

IOM_plugin_parameters

About the IOM_plugin_parameters class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling simulation parameter data.

@author: R. Bourquin @copyright: Copyright (C) 2011, 2012 R. Bourquin @license: Modified BSD License

IOM_plugin_parameters.**add_parameters**(*self*, *blockid='global'*)
Add storage for the simulation parameters.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_parameters.**delete_parameters**(*self*, *blockid='global'*)
Remove the stored simulation parameters.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_parameters.**has_parameters**(*self*, *blockid='global'*)
Ask if the specified data block has the desired data tensor.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_parameters.**load_parameters**(*self*, *blockid='global'*)
Load the simulation parameters.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_parameters.**save_parameters**(*self*, *parameters*, *blockid='global'*)
Save the simulation parameters.

Parameters

- **parameters** – The simulation parameters to store.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_parameters.**update_parameters**(*self*, *parameters*, *blockid='global'*)
Update the parameters by some new values.

Parameters

- **parameters** – The parameters containing updated values.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_grid

About the IOM_plugin_grid class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling grid data.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

IOM_plugin_grid.**add_grid**(*self*, *parameters*, *blockid=0*)
Add storage for a tensor product grid.

Parameters

- **parameters** – A ParameterProvider instance containing at least the keys *number_nodes* and *dimension*.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_grid.**delete_grid**(*self*, *blockid=0*)
Remove the stored grid.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_grid.**has_grid**(*self*, *blockid=0*)
Ask if the specified data block has the desired data tensor.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_grid.**load_grid**(*self*, *blockid=0*)
Load the grid nodes.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_grid.**save_grid**(*self*, *gridnodes*, *blockid=0*)
Save the grid nodes.

Parameters

- **gridnodes** – The grid nodes to store.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_wavefunction

About the `IOM_plugin_wavefunction` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling wavefunction data.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

`IOM_plugin_wavefunction.`**`add_wavefunction`**(*self*, *parameters*, *timeslots=None*, *blockid=0*)
Add storage for the sampled wavefunction.

Parameters

- **parameters** – A `ParameterProvider` instance containing at least the keys *number_nodes* and *ncomponents*.
- **timeslots** – The number of time slots we need. Can be `None` to get automatically growing datasets.
- **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavefunction.`**`delete_wavefunction`**(*self*, *blockid=0*)
Remove the stored wavefunction.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavefunction.`**`has_wavefunction`**(*self*, *blockid=0*)
Ask if the specified data block has the desired data tensor.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavefunction.`**`load_wavefunction`**(*self*, *timestep=None*, *blockid=0*)
Load the wavefunction values.

Parameters

- **timestep** – Load only the data of this timestep.
- **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavefunction.`**`load_wavefunction_timegrid`**(*self*, *blockid=0*)
Load the wavefunction timegrid.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavefunction.`**`save_wavefunction`**(*self*, *wavefunctionvalues*, *timestep=None*, *blockid=0*)
Save the values ψ_i of a `WaveFunction` instance.

Parameters

- **wavefunctionvalues** (*A list of ndarrays.*) – A list of the values to save.
- **timestep** – The timestep at which we save the data.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_fourieroperators

About the `IOM_plugin_fourieroperators` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling the propagation operators that appear in the Fourier algorithm.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

`IOM_plugin_fourieroperators.`**`add_fourieroperators`**(*self*, *parameters*, *blockid=0*)
Add storage for the Fourier propagation operators.

Parameters

- **parameters** – A `ParameterProvider` instance containing at least the keys *ncomponents* and *number_grid_nodes*.
- **blockid** – The ID of the data block to operate on.

`IOM_plugin_fourieroperators.`**`delete_fourieroperators`**(*self*, *blockid=0*)
Remove the stored Fourier operators.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_fourieroperators.`**`has_fourieroperators`**(*self*, *blockid=0*)
Ask if the specified data block has the desired data tensor.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_fourieroperators.`**`load_fourieroperators`**(*self*, *blockid=0*)
Load the Fourier operators.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_fourieroperators.`**`save_fourieroperators`**(*self*, *operators*, *blockid=0*)
Save the kinetic and potential operator to a file.

Parameters

- **operators** – The operators to save, given as tuple (T, V).
- **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket

About the `IOM_plugin_wavepacket` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling homogeneous Hagedorn wavepacket data.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

`IOM_plugin_wavepacket.`**`add_wavepacket`**(*self*, *parameters*, *timeslots=None*, *blockid=0*)
Add storage for the homogeneous wavepackets.

Parameters

- **parameters** – An `ParameterProvider` instance with at least the keys `dimension` and `ncomponents`.
- **timeslots** – The number of time slots we need. Can be `None` to get automatically growing datasets.
- **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavepacket.`**`delete_wavepacket`**(*self*, *blockid=0*)
Remove the stored wavepackets.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavepacket.`**`has_wavepacket`**(*self*, *blockid=0*)
Ask if the specified data block has the desired data tensor.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavepacket.`**`load_wavepacket_basisshapes`**(*self*, *the_hash=None*, *blockid=0*)
Load the basis shapes by hash.

Parameters

- **the_hash** – The hash of the basis shape whose description we want to load.
- **blockid** – The ID of the data block to operate on.

`IOM_plugin_wavepacket.`**`load_wavepacket_coefficients`**(*self*, *timestep=None*, *get_hashes=False*, *component=None*, *blockid=0*)
Load the wavepacket coefficients.

Parameters

- **timestep** – Load only the data of this timestep.
- **get_hashes** – Return the corresponding basis shape hashes.
- **component** – Load only data from this component.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket.**load_wavepacket_description**(*self*, *blockid=0*)
Load the wavepacket description.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket.**load_wavepacket_parameters**(*self*, *timestep=None*, *blockid=0*)
Load the wavepacket parameters.

Parameters

- **timestep** – Load only the data of this timestep.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket.**load_wavepacket_timegrid**(*self*, *blockid=0*)
Load the wavepacket timegrid.

Parameters **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket.**save_wavepacket_basisshapes**(*self*, *basisshape*, *blockid=0*)
Save the basis shapes of the Hagedorn wavepacket to a file.

Parameters

- **basisshape** – The basis shape of the Hagedorn wavepacket.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket.**save_wavepacket_coefficients**(*self*, *coefficients*, *basisshapes*, *timestep=None*, *blockid=0*)
Save the coefficients of the Hagedorn wavepacket to a file. Warning: we do only save tha hash of the basis shapes here! You have to save the basis shape with the corresponding function too.

Parameters

- **coefficients** (A `list` with N suitable `ndarrays`.) – The coefficients of the Hagedorn wavepacket.
- **basisshapes** (A `list` with N `BasisShape` subclass instances.) – The corresponding basis shapes of the Hagedorn wavepacket.
- **timestep** – The timestep at which we save the data.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket.**save_wavepacket_description**(*self*, *descr*, *blockid=0*)
Save the description of this wavepacket.

Parameters

- **descr** – The description.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_wavepacket.**save_wavepacket_parameters**(*self*, *parameters*, *timestep=None*, *blockid=0*)
Save the parameter set Π of the Hagedorn wavepacket Ψ to a file.

Parameters

- **parameters** (A `list` containing the five `ndarrays` like (q, p, Q, P, S)) – The parameter set of the Hagedorn wavepacket.
- **timestep** – The timestep at which we save the data.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_inhomogwavepacket

About the `IOM_plugin_inhomogwavepacket` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling homogeneous Hagedorn wavepacket data.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

`IOM_plugin_inhomogwavepacket.`**`add_inhomogwavepacket`**(*self*, *parameters*, *timeslots=None*, *blockid=0*)

Add storage for the inhomogeneous wavepackets.

Parameters **parameters** – An `ParameterProvider` instance with at least the keys `dimension` and `ncomponents`.

`IOM_plugin_inhomogwavepacket.`**`delete_inhomogwavepacket`**(*self*, *blockid=0*)

Remove the stored wavepackets.

`IOM_plugin_inhomogwavepacket.`**`has_inhomogwavepacket`**(*self*, *blockid=0*)

Ask if the specified data block has the desired data tensor.

`IOM_plugin_inhomogwavepacket.`**`load_inhomogwavepacket_basisshapes`**(*self*, *the_hash=None*, *blockid=0*)

Load the basis shapes by hash.

`IOM_plugin_inhomogwavepacket.`**`save_inhomogwavepacket_basisshapes`**(*self*, *basisshape*, *blockid=0*)

Save the basis shapes of the Hagedorn wavepacket to a file.

Parameters **coefficients** – The basis shapes of the Hagedorn wavepacket.

`IOM_plugin_inhomogwavepacket.`**`save_inhomogwavepacket_coefficients`**(*self*, *coefficients*, *basisshapes*, *timestep=None*, *blockid=0*)

Save the coefficients of the Hagedorn wavepacket to a file. Warning: we do only save tha hash of the basis shapes here! You have to save the basis shape with the corresponding function too.

Parameters

- **coefficients** (A `list` with N suitable `ndarrays`.) – The coefficients of the Hagedorn wavepacket.
- **basisshapes** (A `list` with N `BasisShape` subclass instances.) – The corresponding basis shapes of the Hagedorn wavepacket.

`IOM_plugin_inhomogwavepacket.`**`save_inhomogwavepacket_parameters`**(*self*, *parameters*, *timestep=None*, *blockid=0*)

Save the parameter set Π of the Hagedorn wavepacket Ψ to a file.

Parameters **parameters** (A `list` containing the five `ndarrays` like (q, p, Q, P, S)) – The parameter set of the Hagedorn wavepacket.

IOM_plugin_norm

About the `IOM_plugin_norm` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling norm data.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

`IOM_plugin_norm.`**`add_norm`**(*self*, *parameters*, *timeslots=None*, *blockid=0*)
Add storage for the norms.

Parameters

- **parameters** – A `ParameterProvider` instance containing at least the key *ncomponents*.
- **timeslots** – The number of time slots we need. Can be `None` to get automatically growing datasets.
- **blockid** – The ID of the data block to operate on.

`IOM_plugin_norm.`**`delete_norm`**(*self*, *blockid=0*)
Remove the stored norms.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_norm.`**`has_norm`**(*self*, *blockid=0*)
Ask if the specified data block has the desired data tensor.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_norm.`**`load_norm`**(*self*, *timestep=None*, *split=False*, *blockid=0*)
Load the norm data.

Parameters

- **timestep** – Load only the data of this timestep.
- **split** – Split the data array into one array for each component.
- **blockid** – The ID of the data block to operate on.

`IOM_plugin_norm.`**`load_norm_timegrid`**(*self*, *blockid=0*)
Load the timegrid corresponding to the norm data.

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_norm.`**`save_norm`**(*self*, *norm*, *timestep=None*, *blockid=0*)
Save the norm of wavefunctions or wavepackets.

Parameters

- **norm** – The norm values to save.

- **timestep** – The timestep at which we save the data.
- **blockid** – The ID of the data block to operate on.

IOM_plugin_energy

About the `IOM_plugin_energy` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

IOM plugin providing functions for handling energy data.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

`IOM_plugin_energy.`**`add_energy`**(*self*, *parameters*, *timeslots=None*, *blockid=0*, *key=('kin', 'pot')*)

Parameters

- **parameters** – A `ParameterProvider` instance containing at least the key *ncomponents*.
- **timeslots** – The number of time slots we need. Can be `None` to get automatically growing datasets.
- **blockid** – The ID of the data block to operate on.
- **key** (Tuple of valid identifier strings that are `kin`, `pot` and `tot`. Default is `("kin", "pot")`.) – Specify which energies to save. All are independent.

`IOM_plugin_energy.`**`delete_energy`**(*self*, *blockid=0*)

Remove the stored energies

Parameters **blockid** – The ID of the data block to operate on.

`IOM_plugin_energy.`**`has_energy`**(*self*, *blockid=0*, *key=('kin', 'pot')*)

Ask if the specified data block has the desired data tensor.

Parameters

- **blockid** – The ID of the data block to operate on.
- **key** (Tuple of valid identifier strings that are `kin`, `pot` and `tot`. Default is `("kin", "pot")`.) – Specify which energies to save. All are independent.

`IOM_plugin_energy.`**`load_energy`**(*self*, *timestep=None*, *split=False*, *blockid=0*, *key=('kin', 'pot')*)

Load the energy data.

Parameters

- **timestep** – Load only the data of this timestep.
- **split** – Split the array into arrays for each component.
- **blockid** – The ID of the data block to operate on.
- **key** (Tuple of valid identifier strings that are `kin`, `pot` and `tot`. Default is `("kin", "pot")`.) – Specify which energies to save. All are independent.

IOM_plugin_energy.**load_energy_timegrid**(*self*, *blockid=0*, *key=('kin', 'pot')*)
Load the time grid for specified energies.

Parameters

- **blockid** – The ID of the data block to operate on.
- **key** (Tuple of valid identifier strings that are `kin`, `pot` and `tot`. Default is `("kin", "pot")`.) – Specify which energies to save. All are independent.

IOM_plugin_energy.**save_energy**(*self*, *energies*, *timestep=None*, *blockid=0*, *key=('kin', 'pot')*)
Save the kinetic and potential energies to a file.

Parameters

- **energies** – A tuple containing the energies. The order is important, it has to match the order in the `key` argument. Per default the order has to be $(E_{\text{kin}}, E_{\text{pot}})$.
- **timestep** – The timestep at which we save the data.
- **blockid** – The ID of the data block to operate on.
- **key** (Tuple of valid identifier strings that are `kin`, `pot` and `tot`. Default is `("kin", "pot")`.) – Specify which energies to save. All are independent.

1.1.7 Other classes

GlobalDefaults

About the `GlobalDefaults` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

This file contains some global defaults, for example file names for output files. If a `ParameterProvider` instance is asked about a key which it does not know about it tries to look it up here to see if a default value is available.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

ParameterLoader

About the `ParameterLoader` class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram

Class documentation

class WaveBlocksND.**ParameterLoader**

load_from_dict(*adict*)
Construct a ParameterProvider instance from a common python key-value dict.

Parameters **adict** – A plain python *dict* with key-value pairs.

Returns A ParameterProvider instance.

load_from_file(*filepath*)
Read the parameters from a configuration file.

Parameters **filepath** – Path to the configuration file.

Returns A ParameterProvider instance.

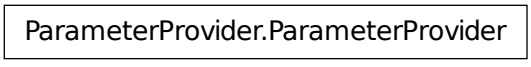
ParameterProvider

About the ParameterProvider class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Inheritance diagram



Class documentation

class WaveBlocksND.**ParameterProvider**

compute_parameters()
Compute some further parameters from the given ones.

get_parameters()
Return a copy of the dict containing all parameters. @return: A copy of the dict containing all parameters. The dict will be copied.

get_timemanager()
Return the embedded I{TimeManager} instance.

set_parameters(*params*)
Overwrite the dict containing all parameters with a newly provided dict with (possibly) changed parameters. @param params: A I{ParameterProvider} instance or a dict with new parameters. The values will be deep-copied. No old values will remain.

update_parameters(*params*)
Overwrite the dict containing all parameters with a newly provided dict with (possibly) changed parameters. @param params: A I{ParameterProvider} instance or a dict with new parameters. The values will be deep-copied. Old values are only overwritten if we have got new values.

FileTools

About the FileTools class

The WaveBlocks Project

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

Class documentation

The WaveBlocks Project

This file contains various functions for finding and retrieving the files that contain parameter settings and simulation results. Note: The terms 'path' and 'ID' are used as synonyms here. Each simulation ID is just the basename of the path or the configuration file.

@author: R. Bourquin @copyright: Copyright (C) 2010, 2011, 2012 R. Bourquin @license: Modified BSD License

FileTools.**compare_by**(*namea*, *nameb*, *pattern*, *ldel='['*, *mdel='='*, *rdel=']'*, *as_string=True*)
Compare two simulation IDs with respect to a (numerical) value in the ID.

Parameters

- **namea** – The first name in the comparison
- **nameb** – The second name in the comparison
- **pattern** – The pattern whose (numerical) value is used for sorting
- **ldel** – Left delimiter of the pattern
- **mdel** – Middle delimiter of the pattern
- **rdel** – Right delimiter of the pattern

- **as_string** – Determines if the values for `pattern` get converted to floats

Returns A boolean answer if the IDs are the same w.r.t the pattern.

`FileTools.`**`gather_all`**(*stringlist*, *pattern*)
Collects all simulation IDs which contain a specific pattern from a given list.

Parameters

- **stringlist** – A list with the simulation IDs
- **pattern** – The pattern

Returns A list of simulation IDs that contain the given pattern.

`FileTools.`**`get_max_by`**(*stringlist*, *pattern*, *ldel='['*, *mdel='='*, *rdel=']'*, *as_string=False*)
Get the maximum of a list with simulation IDs with respect to a (numerical) value in the ID. This is just a simple convenience function so that the user needs not to remember if the sort order is ascending or descending which plays no role for iteration.

Parameters

- **stringlist** – A list with the simulation IDs
- **pattern** – The pattern whose (numerical) value is used for sorting
- **ldel** – Left delimiter of the pattern
- **mdel** – Middle delimiter of the pattern
- **rdel** – Right delimiter of the pattern
- **as_string** – Determines if the values for `pattern` get converted to floats

Returns A sorted list of simulation IDs.

`FileTools.`**`get_min_by`**(*stringlist*, *pattern*, *ldel='['*, *mdel='='*, *rdel=']'*, *as_string=False*)
Get the minimum of a list with simulation IDs with respect to a (numerical) value in the ID. This is just a simple convenience function so that the user needs not to remember if the sort order is ascending or descending which plays no role for iteration.

Parameters

- **stringlist** – A list with the simulation IDs
- **pattern** – The pattern whose (numerical) value is used for sorting
- **ldel** – Left delimiter of the pattern
- **mdel** – Middle delimiter of the pattern
- **rdel** – Right delimiter of the pattern
- **as_string** – Determines if the values for `pattern` get converted to floats

Returns A sorted list of simulation IDs.

`FileTools.`**`get_number_simulations`**(*path*)
Get the number of simulations at hand below the given path.

Parameters **path** – The path under which we search for a output file.

Returns The number of simulations result directories.

`FileTools.`**`get_parameters_file`**(*path*)
Search for a configuration file containing the simulation parameters under a given path. Note that in case there are more than one .py file under the given path we just return the first one found!

Parameters **path** – The path under which we search for a configuration file.

Returns The path (filename) of the configuration file.

`FileTools.`**`get_result_dirs`**(*path*)
Lists all simulations (IDs) that can be found under the given path.

Parameters **path** – The filesystem path under which we search for simulations.

Returns A list of simulation IDs.

`FileTools.`**`get_results_file`**(*path*)
Search for a file containing the simulation results under a given path. Note that in case there are more than one .hdf5 file under the given path we just return the first one found!

Parameters **path** – The path under which we search for a output file.

Returns The path (filename) of the output file.

`FileTools.`**`group_by`**(*stringlist*, *pattern*, *ldel='['*, *mdel='='*, *rdel=']'*, *as_string=True*)
Groups simulation IDs with respect to a pattern.

Parameters

- **stringlist** – A list with the simulation IDs
- **pattern** – The pattern used for grouping
- **ldel** – Left delimiter of the pattern
- **mdel** – Middle delimiter of the pattern
- **rdel** – Right delimiter of the pattern
- **as_string** – Determines if the values for `pattern` get converted to floats. Not used here.

Returns A list of groups of simulation IDs.

`FileTools.`**`intersect_by`**(*lista*, *listb*, *pattern*, *ldel='['*, *mdel='='*, *rdel=']'*, *as_string=True*)
Find the intersection of two lists containing simulation IDs.

Parameters

- **lista** – A first list with the simulation IDs
- **listb** – A second list with the simulation IDs
- **pattern** – The pattern whose numerical value is used for sorting
- **ldel** – Left delimiter of the pattern
- **mdel** – Middle delimiter of the pattern
- **rdel** – Right delimiter of the pattern
- **as_string** – Determines if the values for `pattern` get converted to floats

Returns A sorted list of simulation IDs.

`FileTools.`**`name_contains`**(*name*, *pattern*)
Checks if a simulation ID contains a given pattern.

Parameters

- **name** – The full simulation ID.
- **pattern** – The pattern in question.

Returns A boolean answer.

FileTools.**sort_by**(*stringlist*, *pattern*, *ldel='['*, *mdel='='*, *rdel=']'*, *as_string=False*)
Sorts simulation IDs with respect to a (numerical) value in the ID.

Parameters

- **stringlist** – A list with the simulation IDs
- **pattern** – The pattern whose (numerical) value is used for sorting
- **ldel** – Left delimiter of the pattern
- **mdel** – Middle delimiter of the pattern
- **rdel** – Right delimiter of the pattern
- **as_string** – Determines if the values for `pattern` get converted to floats

Returns A sorted list of simulation IDs.

CHAPTER

TWO

ETC

2.1 Citation

For citation of this project please use one of the following bibtex snippets. The code for one-dimensional simulations:

```
@misc{waveblocks,
    author = {R. Bourquin and V. Gradinaru},
    title = {{WaveBlocks}: Reusable building blocks for simulations with semiclassical wavepackets},
    year = {2010, 2011},
    url = {\url{https://github.com/raoulbq/WaveBlocks}},
    howpublished={\url{https://github.com/raoulbq/WaveBlocks}}
}
```

The code for multi-dimensional simulations:

```
@misc{waveblocksnd,
    author = {R. Bourquin and V. Gradinaru},
    title = {{WaveBlocks}: Reusable building blocks for simulations with semiclassical wavepackets},
    year = {2010, 2011, 2012},
    url = {\url{https://github.com/raoulbq/WaveBlocksND}},
    howpublished={\url{https://github.com/raoulbq/WaveBlocksND}}
}
```

CHAPTER

THREE

INDICES AND TABLES

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

PYTHON MODULE INDEX

c

ComplexMath, 3

f

FileTools, 74

g

GlobalDefaults, 72

i

IOM_plugin_energy, 71
IOM_plugin_fourieroperators, 66
IOM_plugin_grid, 64
IOM_plugin_inhomogwavepacket, 69
IOM_plugin_norm, 70
IOM_plugin_parameters, 63
IOM_plugin_wavefunction, 65
IOM_plugin_wavepacket, 67

m

MatrixExponential, 11

u

Utils, 12

w

WaveBlocksND, 30