pyMOR

Release 0.3.2

December 09, 2015
pyMOR is a software library developed at the University of Münster for building model order reduction applications with the Python programming language. Its main focus lies on the reduction of parameterized partial differential equations using the reduced basis method. All algorithms in pyMOR are formulated in terms of abstract interfaces for seamless integration with external high-dimensional PDE-solver. Moreover, pure Python implementations of finite element and finite volume discretizations using the NumPy/SciPy scientific computing stack are provided for quick and easy prototyping.
CHAPTER 1

Getting started

1.1 Installation

Before trying out pyMOR, you need to install it. We provide packages for Ubuntu via our PPA:

```
sudo apt-add-repository ppa:pymor/stable
sudo apt-get update
sudo apt-get install python-pymor python-pymor-demos python-pymor-doc
```

Daily snapshots can be installed by using the pymor/daily PPA instead of pymor/stable. The current release can also be installed via pip. Please take a look at our README file for further details. The README also contains instructions for setting up a development environment for working on pyMOR itself.

1.2 Trying it out

While we consider pyMOR mainly as a library for building MOR applications, we ship a few example scripts. These can be found in the src/pymordemos directory of the source repository. Try launching one of them using the pymor-demo script contained in the python-pymor-demos package:

```
pymor-demo thermalblock --with-estimator --plot-err --plot-solutions 3 2 3 32
```

The demo scripts can also be launched directly from the source tree:

```
./thermalblock.py --with-estimator --plot-err --plot-solutions 3 2 3 32
```

This will solve and reduce the so called thermal block problem using the reduced basis method with a greedy basis generation algorithm. The thermal block problem consists in solving the stationary diffusion problem

\[
- \left[ d(x, \mu) \right] \frac{\partial u(x, \mu)}{\partial x} = 1 \quad \text{for } x \in \Omega \\
\quad u(x, \mu) = 0 \quad \text{for } x \in \partial \Omega
\]

on the domain \( \Omega = [0,1]^2 \) for the unknown \( u \). The domain is partitioned into \( XBLOCKS \times YBLOCKS \) blocks (\( XBLOCKS \) and \( YBLOCKS \) are the first two arguments to thermalblock.py). The thermal conductivity \( d(x, \mu) \) is constant on each block \( (i,j) \) with value \( \mu_{ij} \):

\[
\begin{array}{c|c|c}
0,0 & 1,0 & \ldots \\
\hline
\mu_{11} & \mu_{12} & \mu_{13} \\
\hline
\hline
\end{array}
\]
The real numbers $\mu_{ij}$ form the $XBLOCKS \times YBLOCKS$-dimensional parameter on which the solution depends. Running `thermalblock.py` will first produce plots of two detailed solutions of the problem for different randomly chosen parameters using linear finite elements. (The size of the grid can be controlled via the `--grid` parameter. The randomly chosen parameters will actually be the same for each run, since the random generator is initialized with a fixed default seed in `new_random_state`.)

After closing the window, the reduced basis for model order reduction is generated using a greedy search algorithm with error estimator. The third parameter `SNAPSHOTS` of `thermalblock.py` determines how many different values per parameter component $\mu_{ij}$ should be considered. I.e. the parameter training set for basis generation will have the size $SNAPSHOTS^{(XBLOCKS \times YBLOCKS)}$. After the basis of size 32 (the last parameter) has been computed, the quality obtained reduced model (on the 32-dimensional reduced basis space) is evaluated by comparing the solutions of the reduced and detailed models for new randomly chosen parameters. Finally, plots of the detailed and reduced solutions as well as the difference between the two are displayed for the random parameter which maximises reduction error.

### 1.3 The thermalblock demo explained

In the following we will walk through the thermal block demo step by step in an interactive Python shell. We assume that you are familiar with the reduced basis method and that you know the basics of Python programming as well as working with NumPy. (Note that our code will differ a bit from `thermalblock.py` as we will hardcode the various options the script offers and leave out some features.)

First, start a Python shell. We recommend using IPython

```python
ipython
```

You can paste the following input lines starting with `>>>` by copying them to the system clipboard and then executing `%paste` inside the IPython shell.

First, we will import the most commonly used methods and classes of pyMOR by executing:

```python
>>> from pymor.basic import *
Loading pymor version 0.3.0
```

Next we will instantiate a class describing the analytical problem we want so solve. In this case, a `ThermalBlockProblem`:

```python
>>> p = ThermalBlockProblem(num_blocks=(3, 2))
```

We want to discretize this problem using the finite element method. We could do this by hand, creating a `Grid`, instantiating `DiffusionOperatorP1` finite element diffusion operators for each subblock of the domain, forming a `LincombOperator` to represent the affine decomposition, instantiating a `L2ProductFunctionalP1` as right hand side, and putting it all together into a `StationaryDiscretization`. However, since `ThermalBlockProblem` derives form `EllipticProblem`, we can use a predefined `discretizer` to do the work for us. In this case, we use `discretize_elliptic_cg`:

```python
>>> d, d_data = discretize_elliptic_cg(p, diameter=1. / 100.)
```
is the `StationaryDiscretization` which has been created for us, whereas `d_data` contains some additional data, in this case the `Grid` and the `BoundaryInfo` which have been created during discretization. We can have a look at the grid:

```python
>>> print(d_data['grid'])
Tria-Grid on domain [0,1] x [0,1]
x0-intervals: 100, x1-intervals: 100
faces: 40000, edges: 60200, vertices: 20201
```

and, as always, we can display its class documentation using `help(d_data['grid'])`, or in the case of IPython `d_data['grid']?`.

Let’s solve the thermal block problem and visualize the solution:

```python
>>> U = d.solve([1.0, 0.1, 0.3, 0.1, 0.2, 1.0])
>>> d.visualize(U, title='Solution')
```

Each class in pyMOR that describes a `Parameter` dependent mathematical object, like the `StationaryDiscretization` in our case, derives from `Parametric` and determines the `Parameters` it expects during `__init__` by calling `build_parameter_type`. The resulting `ParameterType` is stored in the object’s `parameter_type` attribute. Let us have a look:

```python
>>> print(d.parameter_type)
{diffusion: (2, 3)}
```

This tells us, that the `Parameter` which `solve` expects should be a dictionary with one key ‘diffusion’ whose value is a NumPy array of shape (2, 3) corresponding to the block structure of the problem. However, by using the `parse_parameter` method, pyMOR is smart enough to correctly parse the input `[1.0, 0.1, 0.3, 0.1, 0.2, 1.0]`.

Next we want to use the `greedy` algorithm to reduce the problem. For this we need to choose a basis extension algorithm as well as a reductor which will perform the actual RB-projection. We will use `gram_schmidt_basis_extension` and `reduce_stationary_coercive`. The latter will also assemble an error estimator to estimate the reduction error. This will significantly speed up the basis generation, as we will only need to solve the high-dimensional problem for those parameters in the training set which are actually selected for basis extension. To control the condition of the reduced system matrix, we must ensure that the generated basis is orthonormal w.r.t. the H1-product on the solution space. For this we pass the basis extension algorithm the `h1_product` attribute of the discretization. We pass the same product to the reductor for computing the Riesz representatives for error estimation. Moreover, we have to provide a `ParameterFunctional` which computes a lower bound for the coercivity of the problem for a given parameter.

```python
>>> from functools import partial

>>> extension_algorithm = partial(gram_schmidt_basis_extension, product=d.h1_product)
>>> reductor = partial(reduce_stationary_coercive, error_product=d.h1_product,
                coercivity_estimator=GenericParameterFunctional(lambda mu: np.min(mu['diffusion']
                                                                                       , d.parameter_type))
```

Moreover, we need to select a `Parameter` training set. The discretization `d` already comes with a `ParameterSpace` which it has inherited from the analytical problem. We can sample our parameters from this space, which is a `CubicParameterSpace`. E.g.:

```python
>>> samples = list(d.parameter_space.sample_uniformly(4))
>>> print(samples[0])
{diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]}
```
>>> greedy_data = greedy(d, reductor, samples, 
...    extension_algorithm=extension_algorithm, 
...    use_estimator=True, max_extensions=32)
07:42|algorithms.greedy.greedy: Started greedy search on 4096 samples
07:42|algorithms.greedy.greedy: Reducing ...
07:42|algorithms.greedy.greedy: Estimating errors ...
...
07:44|algorithms.greedy.greedy: Maximum error after 0 extensions: 9.86736953629 \(\mu = \{\text{diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]\}}\)
07:44|algorithms.greedy.greedy: Extending with snapshot for \(\mu = \{\text{diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]\}}\)
...
15:26|algorithms.greedy.greedy: Maximum number of 32 extensions reached.
15:26|algorithms.greedy.greedy: Reducing once more ...
15:55|algorithms.greedy.greedy: Greedy search took 492.942929029 seconds

The `max_extensions` parameter defines how many basis vectors we want to obtain. `greedy_data` is a dictionary containing various data that has been generated during the run of the algorithm:

```python
>>> print(greedy_data.keys())
['reduction_data', 'reconstructor', 'time', 'basis', 'extensions', 'reduced_discretization', 'max_errs', 'max_err_mus']
```

The most important items are `reduced_discretization` and `reconstructor` which hold the reduced `Discretization` obtained from applying our reductor with the final reduced basis, as well as a reconstructor to reconstruct detailed solutions from the reduced solution vectors. The reduced basis is stored as `basis` item.

```python
>>> rd = greedy_data['reduced_discretization']
>>> rc = greedy_data['reconstructor']
>>> rb = greedy_data['basis']
```

All vectors in pyMOR are stored in so called `VectorArrays`. For example the solution \(U\) computed above is given as a `VectorArray` of length 1. For the reduced basis we have:

```python
>>> print(type(rb))
<class 'pymor.la.numpyvectorarray.NumpyVectorArray'>
>>> print(len(rb))
32
>>> print(rb.dim)
20201
```

Let us check if the reduced basis really is orthonormal with respect to the H1-product. For this we use the `apply2` method:

```python
>>> import numpy as np
>>> gram_matrix = d.h1_product.apply2(rb, rb, pairwise=False)
>>> print(np.max(np.abs(gram_matrix - np.eye(32))))
1.24982272795e-13
```

Looks good! We can now solve the reduced model for the same parameter as above. The result is a vector of coefficients w.r.t. the reduced basis, which is currently stored in \(rb\). To form the linear combination, we can use the reconstructor:

```python
>>> u = rd.solve([1.0, 0.1, 0.3, 0.1, 0.2, 1.0])
>>> print(u)
[[ 5.79477471e-01  5.91289054e-02  1.89924036e-01  1.89149529e-02
  1.81103127e-01  2.69928905e-02  1.89495298e-01  7.99676272e-03
  1.54092560e-01  5.76326362e-02  1.97982347e-01  2.05779889e-02
  6.81573489e-02  1.27037440e-01  2.52674851e-02]]
```

Chapter 1. Getting started
Finally we compute the reduction error and display the reduced solution along with the detailed solution and the error:

```python
>>> ERR = U - U_red
>>> print(d.h1_norm(ERR))
[ 0.00944595]
>>> d.visualize((U, U_red, ERR), legend=('Detailed', 'Reduced', 'Error'), separate_colorbars=True)
```

We can nicely observe that, as expected, the error is maximized along the jumps of the diffusion coefficient.

## 1.4 Learning more

As a next step, you should read our *Technical Overview* which discusses the most important concepts and design decisions behind pyMOR. After that you should be fit to delve into the reference documentation.

Should you have any problems regarding pyMOR, questions or feature requests, do not hesitate to contact us at our mailing list!
2.1 Three Central Classes

From a bird’s eye perspective, pyMOR is a collection of generic algorithms operating on objects of the following types:

**VectorArrays** Vector arrays are ordered collections of vectors. Each vector of the array must be of the same dimension. Subsets of vectors can be copied to a new array, appended to an existing array, deleted from the array or replaced by vectors of a different array. Basic linear algebra operations can be performed on the vectors of the array: vectors can be scaled in-place, the BLAS `axpy` operation is supported and scalar products between vectors can be formed. Linear combinations of vectors can be formed using the `lincomb` method. Moreover, various norms can be computed and selected components of the vectors can be extracted for empirical interpolation.

Each of these methods takes optional `ind` parameters to specify the subset of vectors on which to operate. If the parameter is not specified, the whole array is selected for the operation.

New vector arrays can be created using the `empty` and `zeros` method. As a convenience, many of Python’s math special methods are implemented in terms of the interface methods.

Note that there is not the notion of a single vector in pyMOR. The main reason for this design choice is to take advantage of vectorized implementations like `NumpyVectorArray` which internally store the vectors as two-dimensional NumPy arrays. As an example, the application of a linear matrix based operator to an array via the `apply` method boils down to a call to NumPy’s optimized `dot` method. If there were only lists of vectors in pyMOR, the above matrix-matrix multiplication would have to be expressed by a loop of matrix-vector multiplications. However, when working with external solvers, vector arrays will often be just lists of vectors. For this use-case we provide `ListVectorArray`, a vector array based on a Python list of vectors.

Associated to each vector array is a **VectorSpace**. A Vector space in pyMOR is simply the combination of a `VectorArray` class and an appropriate `subtype`. For `NumpyVectorArrays`, the subtype is a single integer denoting the dimension of the array. Subtypes for other array classes could, e.g., include a socket for communication with a specific PDE solver instance.

Two arrays in pyMOR are compatible (e.g. can be added) if they are from the same `VectorSpace`, i.e. they are instances of the same class and share the same subtype. The `VectorSpace` is also precisely the information needed to create new arrays of null vectors using the `make_array` class method. In fact `empty` and `zeros` are implemented by calling `make_array` with the `subtype` of the `VectorArray` instance for which they have been called.

**Operators** The main property of operators in pyMOR is that they can be applied to `VectorArrays` resulting in a new `VectorArray`. For this operation to be allowed, the operator's source `VectorSpace` must be identical with the `VectorSpace` of the given array. The result will be a vector array from the range space. An operator can be `linear` or not. The `apply_inverse` method provides an interface for (linear) solvers.
Operators in pyMOR are also used to represent bilinear forms via the `apply2` method. A functional in pyMOR is simply an operator with `VectorSpace(NumpyVectorArray, 1)` as range. Dually, a vector-like operator is an operator with a `VectorSpace(NumpyVectorArray, 1)` as source. Such vector-like operators are used in pyMOR to represent `Parameter` dependent vectors such as the initial data of an `InstationaryDiscretization`. For linear functionals and vector-like operators, the `as_vector` method can be called to obtain a vector representation of the operator as a `VectorArray` of length 1.

Linear combinations of operators can be formed using a `LincombOperator`. When such a linear combination is assembled, `assemble_lincomb` is called to ensure that, for instance, linear combinations of operators represented by a matrix lead to a new operator holding the linear combination of the matrices. The `projected` method is used to perform the reduced basis projection of a given operator. While each operator in pyMOR can be projected, specializations of this method ensure that, if possible, the projected operator will no longer depend on high-dimensional data.

Default implementations for many methods of the operator interface can be found in `OperatorBase`. Base classes for NumPy-based operators can be found in `pymor.operators.numpy`. Several methods for constructing new operators from existing ones are contained in `pymor.operators.constructions`.

Discretizations
Discretizations in pyMOR encode the mathematical structure of a given discrete problem by acting as container classes for operators. Each discretization object has `operators`, `functionals`, `vector_operators` and `products` dictionaries holding the `Operators` which appear in the formulation of the discrete problem. The keys in these dictionaries describe the role of the respective operator in the discrete problem.

Apart from describing the discrete problem, discretizations also implement algorithms for solving the given problem, returning `VectorArrays` with space `solution_space`. The solution is usually cached, s.t. subsequent solving of the problem for the same parameters reduces to looking up the solution in pyMOR’s cache.

While special discretization classes may be implemented which make use of the specific types of operators they contain (e.g. using some external high-dimensional solver for the problem), it is generally favourable to implement the solution algorithms only through the interfaces provided by the operators contained in the discretization, as this allows to use the same discretization class to solve high-dimensional and reduced problems. This has been done for the simple stationary and instationary discretizations found in `pymor.discretizations.basic`.

Discretizations can also implement `estimate` and `visualize` methods to estimate the discretization error of a computed solution and create graphic representations of `VectorArrays` from the `solution_space`.

## 2.2 Base Classes

While `VectorArrays` are mutable objects, both `Operators` and `Discretizations` are immutable in pyMOR: the application of an `Operator` to the same `VectorArray` will always lead to the same result, solving a `Discretization` for the same parameter will always produce the same solution array. This has two main benefits:

1. If multiple objects/algorithms hold references to the same `Operator` or `Discretization`, none of the objects has to worry that the referenced object changes without their knowledge.

2. The state of an immutable object is determined by the states of the objects that lead to the creation of the object. This property is used in pyMOR to create `state ids` for immutable objects which are used as keys in pyMOR’s caching backends.

A class can be made immutable in pyMOR by deriving from `ImmutableInterface`, which ensures that write access to the object’s attributes is prohibited after `__init__` has been executed. However, note that changes to private attributes (attributes whose name starts with `_`) are still allowed. It lies in the implementors responsibility to ensure that changes to these attributes do not affect the outcome of calls to relevant interface methods. As an example, a
call to `enable_caching` will set the objects private `__cache_region` attribute, which might affect the speed of a subsequent `solve` call, but not its result.

Of course, in many situations one may wish to change properties of an immutable object, e.g. the number of timesteps for a given discretization. This can be easily achieved using the `with_` method every immutable object has: a call of the form `o.with_(a=x, b=y)` will return a copy of `o` in which the attribute `a` now has the value `x` and the attribute `b` the value `y`. It can be generally assumed that calls to `with_` are inexpensive. The set of allowed arguments can be found in the `with_arguments` attribute.

All immutable classes in pyMOR and most other classes derive from `BasicInterface` which, through its meta class, provides several convenience features for pyMOR. Most notably, every subclass of `BasicInterface` obtains its own `logger` instance with a class specific prefix.

### 2.3 Creating Discretizations

pyMOR ships a small (and still quite incomplete) framework for creating finite element or finite volume discretizations based on the NumPy/Scipy software stack. To end up with an appropriate `Discretization`, one starts by instantiating an `analytical problem` which describes the problem we want to discretize. `analytical problems` contain `Functions` which define the analytical data functions associated with the problem and a `DomainDescription` that provides a geometrical definition of the domain the problem is posed on and associates a `BoundaryType` to each part of its boundary.

To obtain a `Discretization` from an `analytical problem` we use a `discretizer`. A discretizer will first mesh the computational domain by feeding the `DomainDescription` into a `domain_discretizer` which will return the `Grid` along with a `BoundaryInfo` associating boundary entities with `BoundaryTypes`. Next, the `Grid`, `BoundaryInfo` and the various data functions of the `analytical problem` are used to instatiate finite element or finite volume operators. Finally these operators are used to instatiate one of the provided `Discretization` classes.

In pyMOR, `analytical problems`, `Functions`, `DomainDescriptions`, `BoundaryInfos` and `Grids` are all immutable, ensuring that the resulting `Discretization` receives a proper `state id` to enable persistent disk caching over various runs of the applications written with pyMOR.

While pyMOR’s internal discretizations are useful for getting started quickly with model reduction experiments, pyMOR’s main goal is to allow the reduction of discretizations provided by external solvers. In order to do so, all that needs to be done is to provide `VectorArrays`, `Operators` and `Discretizations` which interact appropriately with the solver. pyMOR makes no assumption on how the communication with the solver is managed. For instance, communication could take place via a network protocol or job files. In particular it should be stressed that in general no communication of high-dimensional data between the solver and pyMOR is necessary: `VectorArrays` can merely hold handles to data in the solver’s memory or some on-disk database. Where possible, we favour, however, a deep integration of the solver with pyMOR by linking the solver code as a Python extension module. This allows Python to directly access the solver’s data structures which can be used to quickly add features to the high-dimensional code without any recompilation. A minimal example for such an integration using `pybindgen` can be found in the `src/pymordemos/minimal_cpp_demo` directory of the pyMOR repository. The `dune-pymor` repository contains experimental bindings for the DUNE software framework.

### 2.4 Parameters

pyMOR classes implement dependence on a parameter by deriving from the `Parametric` mix-in class. This class gives each instance a `parameter_type` attribute describing the form of `Parameters` the relevant methods of the object (`apply`, `solve`, `evaluate`, etc.) expect. A `Parameter` in pyMOR is basically a Python `dict` with strings as keys and NumPy arrays as values. Each such value is called a `Parameter` component. The
**2.5 Defaults**

pyMOR offers a convenient mechanism for handling default values such as solver tolerances, cache sizes, log levels, etc. Each default in pyMOR is the default value of an optional argument of some function. Such an argument is made a default by decorating the function with the `@defaults` decorator:

```python
@defaults('tolerance')
def some_algorithm(x, y, tolerance=1e-5):
    ...
```

Default values can be changed by calling `set_defaults`. A configuration file with all defaults defined in pyMOR can be obtained with `write_defaults_to_file`. This file can then be loaded, either programatically or automatically by setting the `PYMOR_DEFAULTS` environment variable.

As an additional feature, if `None` is passed as value for a function argument which is a default, its default value is used instead of `None`. This allows writing code of the following form:

```python
def method_called_by_user(U, V, tolerance_for_algorithm=None):
    ...
    algorithm(U, V, tolerance=tolerance_for_algorithm)
```

See the `defaults` module for more information.

**2.6 The Reduction Process**

The reduction process in pyMOR is handled by so called reductors which take arbitrary `Discretizations` and additional data (e.g. the reduced basis) to create reduced `Discretizations` along with reconstructor classes which allow to transform solution vectors of the reduced `Discretization` back to vectors of the solution space of the high-dimensional `Discretization` (e.g. by linear combination with the reduced basis). If proper offline/online decomposition is achieved by the reductor, the reduced `Discretization` will not store any high-dimensional data. Note that there is no inherent distinction between low- and high-dimensional `Discretizations` in pyMOR. The only difference lies in the different types of operators, the `Discretization` contains.

This observation is particularly apparent in the case of the classical reduced basis method: the operators and functionals of a given discrete problem are projected onto the reduced basis space whereas the structure of the problem (i.e. the type of `Discretization` containing the operators) stays the same. pyMOR reflects this fact by offering with `reduce_generic_rb` a generic algorithm which can be used to RB-project any discretization available to pyMOR. It should be noted however that this reductor is only able to efficiently offline/online-decompose affinely `Parameter`-dependent linear problems. Non-linear problems or such with no affine `Parameter` dependence require additional techniques such as empirical interpolation.

If you want to further dive into the inner workings of pyMOR, we highly recommend to study the source code of `reduce_generic_rb` and to step through calls of this method with a Python debugger, such as ipdb.
Environment Variables

pyMOR respects the following environment variables:

**PYMOR_CACHE_DISABLE** If 1, disable caching globally, overriding calls to `enable_caching`. This is mainly useful for debugging. See `pymor.core.cache` for more details.

**PYMOR_CACHE_MAX_SIZE** Maximum size of pyMOR’s default disk `CacheRegion`. If not set, the `pymor.core.cache._setup_default_regions.disk_max_size default` is used. The suffixes ‘k’ or ‘K’, ‘m’ or ‘M’ and ‘g’ or ‘G’ can be used to specify the amount as a number of kilobytes, megabytes or gigabytes.

**PYMOR_CACHE_MEMORY_MAX_KEYS** Maximum number of keys stored in pyMOR’s default memory `CacheRegion`. If not set, the `pymor.core.cache._setup_default_regions.memory_max_keys default` is used.

**PYMOR_CACHE_PATH** Location of pyMOR’s default disk `CacheRegion`. If not set, the `pymor.core.cache._setup_default_regions.disk_path default` is used.

**PYMOR_COLORS_DISABLE** If 1, disable coloring of logging output.

**PYMOR_COPY_DOCSTRINGS_DISABLE** By default, docstrings of methods in base classes are copied to overriding methods, if these do not define their own docstring. Setting this variable to 1 disables this feature. (We use this for when auto-generating API-documentation with sphinx.)

**PYMOR_DEFAULTS** If empty or `NONE`, do not load any `defaults` from file. Otherwise, a `:`-separated list of the paths to a Python scripts containing defaults.
4.1 pymor package

4.1.1 Subpackages

pymor.algorithms package

Submodules

basisextension module   This module contains algorithms for extending a reduced given basis by a new vector.

The methods are mainly designed to be used in conjunction with the greedy algorithm. Each method is of the form

extension_algorithm(basis, U, ...)

where basis and U are VectorArrays containing the old basis and new vectors to add. The methods return a tuple
new_basis, data where new_basis holds the extend basis and data is a dict containing additional information about the
extension process. The data dict at least has the key hierarchic whose value signifies if the new basis contains the old
basis as its first vectors.

If the basis extension fails, e.g. because the new vector is not linearly independent from the basis, an
ExtensionError exception is raised.

pymor.algorithms.basisextension.gram_schmidt_basis_extension(basis, U, product=product, copy_basis=copy_basis, copy_U=copy_U)

Extend basis using Gram-Schmidt orthonormalization.

Parameters

basis VectorArray containing the basis to extend.
U VectorArray containing the new basis vectors.
product The scalar product w.r.t. which to orthonormalize; if None, the Euclidean product is used.
copy_basis If copy_basis is False, the old basis is extended in-place.
copy_U If copy_U is False, the new basis vectors are removed from U.

Returns
new_basis  The extended basis.

extension_data  Dict containing the following fields:

  hierarchic  True if new_basis contains basis as its first vectors.

Raises

 ExtensionError  Gram-Schmidt orthonormalization fails. This is the case when no vector in U is linearly independent from the basis.

pymor.algorithms.basisextension.pod_basis_extension(basis, U, count=1, copy_basis=True, product=None, orthonormalize=True)

Extend basis with the first count POD modes of the projection of U onto the orthogonal complement of the basis.

Note that the provided basis is assumed to be orthonormal w.r.t. the provided scalar product!

Parameters

  basis  VectorArray containing the basis to extend. The basis is expected to be orthonormal w.r.t. product.

  U  VectorArray containing the vectors to which the POD is applied.

  count  Number of POD modes that are to be appended to the basis.

  product  The scalar product w.r.t. which to orthonormalize; if None, the Euclidean product is used.

  copy_basis  If copy_basis is False, the old basis is extended in-place.

  orthonormalize  If True, re-orthonormalize the new basis vectors obtained by the POD in order to improve numerical accuracy.

Returns

  new_basis  The extended basis.

extension_data  Dict containing the following fields:

  hierarchic  True if new_basis contains basis as its first vectors.

Raises

 ExtensionError  POD produces no new vectors. This is the case when no vector in U is linearly independent from the basis.

pymor.algorithms.basisextension.trivial_basis_extension(basis, U, copy_basis=True, copy_U=True)

Trivially extend basis by simply appending the new vectors.

We check if the new vectors are already contained in the basis, but we do not check for linear independence.

Parameters

  basis  VectorArray containing the basis to extend.
U VectorArray containing the new basis vectors.

copy_basis If copy_basis is False, the old basis is extended in-place.

copy_U If copy_U is False, the new basis vectors are removed from U.

Returns

new_basis The extended basis.

extension_data Dict containing the following fields:

  hierarchic True if new_basis contains basis as its first vectors.

Raises

ExtensionError Is raised if all vectors in U are already contained in the basis.

ei module This module contains algorithms for the empirical interpolation of operators.

The main work for generating the necessary interpolation data is handled by the ei_greedy method. The objects returned by this method can be used to instantiate an EmpiricalInterpolatedOperator.

As a convenience, the interpolate_operators method allows to perform the empirical interpolation of the Operators of a given discretization with a single function call.

pymor.algorithms.ei.deim(U, modes=None, error_norm=None, product=None)
Generate data for empirical interpolation using DEIM algorithm.

Given a VectorArray U, this method generates a collateral basis and interpolation DOFs for empirical interpolation of the vectors contained in U. The returned objects can also be used to instantiate an EmpiricalInterpolatedOperator.

The collateral basis is determined by the first pod modes of U.

Parameters

U A VectorArray of vectors to interpolate.

modes Dimension of the collateral basis i.e. number of POD modes of the vectors in U.

error_norm Norm w.r.t. which to calculate the interpolation error. If None, the Euclidean norm is used.

product Product Operator used for POD.

Returns

interpolation_dofs NumPy array of the DOFs at which the vectors are interpolated.

collateral_basis VectorArray containing the generated collateral basis.

data Dict containing the following fields:

  errors Sequence of maximum approximation errors during greedy search.
Generate data for empirical interpolation by a greedy search (EI-Greedy algorithm).

Given a VectorArray \( U \), this method generates a collateral basis and interpolation DOFs for empirical interpolation of the vectors contained in \( U \). The returned objects can also be used to instantiate an EmpiricalInterpolatedOperator.

The interpolation data is generated by a greedy search algorithm, adding in each loop the worst approximated vector in \( U \) to the collateral basis.

**Parameters**

- **U** A VectorArray of vectors to interpolate.
- **error_norm** Norm w.r.t. which to calculate the interpolation error. If None, the Euclidean norm is used.
- **target_error** Stop the greedy search if the largest approximation error is below this threshold.
- **max_interpolation_dofs** Stop the greedy search if the number of interpolation DOF (= dimension of the collateral basis) reaches this value.
- **projection** If ei, compute the approximation error by comparing the given vector by its interpolant. If orthogonal, compute the error by comparing with the orthogonal projection onto the span of the collateral basis.
- **product** If projection == ‘orthogonal’, the product which is used to perform the projection. If None, the Euclidean product is used.

**Returns**

- **interpolation_dofs** NumPy array of the DOFs at which the vectors are evaluated.
- **collateral_basis** VectorArray containing the generated collateral basis.
- **data** Dict containing the following fields:
  - **errors** Sequence of maximum approximation errors during greedy search.
  - **triangularity_errors** Sequence of maximum absolute values of interpolation matrix coefficients in the upper triangle (should be near zero).

Empirical operator interpolation using the EI-Greedy algorithm.

This is a convenience method for facilitating the use of ei_greedy. Given a Discretization, names of Operators, and a sample of Parameters, first the operators are evaluated on the solution snapshots of the discretization for the provided parameters. These evaluations are then used as input for ei_greedy. Finally the resulting interpolation data is used to create EmpiricalInterpolatedOperators and a new discretization with the interpolated operators is returned.

Note that this implementation creates ONE common collateral basis for all specified operators which might not be what you want.

**Parameters**
discretization The **Discretization** whose **Operators** will be interpolated.

operator_names List of keys in the **operators** dict of the discretization. The corresponding **Operators** will be interpolated.

sample A list of **Parameters** for which solution snapshots are calculated.

error_norm See `ei_greedy`.

target_error See `ei_greedy`.

max_interpolation_dofs See `ei_greedy`.

projection See `ei_greedy`.

product See `ei_greedy`.

**Returns**

ei_discretization **Discretization** with **Operators** given by **operator_names** replaced by `EmpiricalInterpolatedOperators`.

data Dict containing the following fields:

- **dofs** `NumPy array` of the DOFs at which the **Operators** have to be evaluated.
- **basis** `VectorArray` containing the generated collateral basis.
- **errors** Sequence of maximum approximation errors during greedy search.

**greedy module**

```python
pymor.algorithms.greedy.greedy(discretization, reductor, samples, initial_basis=None, use_estimator=True, error_norm=None, extension_algorithm=<function trivial_basis_extension>, target_error=None, max_extensions=None)
```

Greedy basis generation algorithm.

This algorithm generates a reduced basis by iteratively adding the worst approximated solution snapshot for a given training set to the reduced basis. The approximation error is computed either by directly comparing the reduced solution to the detailed solution or by using an error estimator (`use_estimator == True`). The reduction and basis extension steps are performed by calling the methods provided by the `reductor` and `extension_algorithm` arguments.

**Parameters**

- **discretization** The **Discretization** to reduce.
- **reductor** Reductor for reducing the given **Discretization**. This has to be a function of the form `reductor(discretization, basis, extends=None)`. If your reductor takes more arguments, use, e.g., `functools.partial`. The method has to return a tuple `(reduced_discretization, reconstructor, reduction_data)`. In case the last basis extension was hierarchic (see `extension_algorithm`), the extends argument is set to `(last_reduced_discretization, last_reconstructor, last_reduction_data)` which can be used by the reductor to speed up the reduction process. For an example see `reduce_stationary_affine_linear`.
- **samples** The set of **Parameter** samples on which to perform the greedy search.
- **initial_basis** The initial reduced basis with which the algorithm starts. If `None`, an empty basis is used as initial basis.
**use_estimator** If True, use `reduced_discretization.estimate()` to estimate the errors on the sample set. Otherwise a detailed simulation is performed to calculate the error.

**error_norm** If `use_estimator == False`, use this function to calculate the norm of the error. If None, the Euclidean norm is used.

**extension_algorithm** The extension algorithm to be used to extend the current reduced basis with the maximum error snapshot. This has to be a function of the form `extension_algorithm(old_basis, new_vector)`, which returns a tuple `(new_basis, extension_data)`, where `extension_data` is a dict at least containing the key `hierarchic`. `hierarchic` should be set to True if `new_basis` contains `old_basis` as its first vectors.

**target_error** If not None, stop the algorithm if the maximum (estimated) error on the sample set drops below this value.

**max_extensions** If not None, stop the algorithm after `max_extensions` extension steps.

**Returns**
Dict with the following fields

- **basis** The reduced basis.
- **reduced_discretization** The reduced `Discretization` obtained for the computed basis.
- **reconstructor** Reconstructor for `reduced_discretization`.
- **max_errs** Sequence of maximum errors during the greedy run.
- **max_err_mus** The parameters corresponding to `max_errs`.

**newton module**

```
pymor.algorithms.newton.newton(operator, rhs, initial_guess=None, mu=None, error_norm=None, miniter=0, maxiter=10, reduction=1e-10, abs_limit=1e-15, stagnation_window=0, stagnation_threshold=1e+99, return_stages=False, return_residuals=False)
```

Basic Newton algorithm.
This method solves the nonlinear equation

$$A(U, \mu) = V$$

for $U$ using the Newton method.

**Parameters**

- **operator** The `Operator A`. $A$ must implement the `jacobian` interface method.
- **rhs** `VectorArray` of length 1 containing the vector $V$.
- **initial_guess** If `None`, a `VectorArray` of length 1 containing an initial guess for the solution $U$.
- **mu** The `Parameter` for which to solve the equation.
- **error_norm** The norm with which the norm of the residual is computed. If `None`, the Euclidean norm is used.
- **miniter** Minimum amount of iterations to perform.
- **maxiter** Fail if the iteration count reaches this value without converging.
reduction  Finish if the residual norm has been reduced by this factor relative to the norm of the initial residual.
abs_limit  Finish if the residual norm is below this threshold.
stagnation_window  Finish if the residual norm has not been reduced by a factor of stagnation_threshold during the last stagnation_window iterations.
stagnation_threshold  See stagnation_window.
return_stages  If True return a VectorArray of the approximations of U after each iteration in the data dict.
return_residuals  If True return a VectorArray of all residual vectors which have been computed during the Newton iterations.

Returns
U  VectorArray of length 1 containing the computed solution
data  Dict containing the following fields:
   error_sequence  NumPy array containing the residual norms after each iteration.
   stages  See return_stages.
   residuals  See return_residuals.

Raises
NewtonError  Raised if the Newton algorithm failed to converge.

Defaults
miniter, maxiter, reduction, abs_limit, stagnation_window, stagnation_threshold (see pymor.core.defaults)

timestepping module  This module provides generic time-stepping algorithms for the solution of instationary problems.
The algorithms are generic in the sense that each algorithms operates exclusively on Operators and VectorArrays. In particular, the algorithms can also be used to turn an arbitrary stationary Discretization provided by an external library into an instationary Discretization.
Currently, implementations of explicit_euler and implicit_euler time-stepping are provided. The TimeStepperInterface defines a common interface that has to be fulfilled by the time-steppers that are used by InstationaryDiscretization. The classes ExplicitEulerTimeStepper and ImplicitEulerTimeStepper encapsulate explicit_euler and implicit_euler to provide this interface.

class pymor.algorithms.timestepping.ExplicitEulerTimeStepper(nt)
   Bases: pymor.algorithms.timestepping.TimeStepperInterface
   Implicit-Euler time-stepper.
   Solves equations of the form
   \[ M \frac{d}{dt} u + A(u, \mu, t) = F(\mu, t). \]
**nt**  The number of time-steps the time-stepper will perform.

---

### Methods

<table>
<thead>
<tr>
<th>pymor.algorithms.timestepping.ExplicitEulerTimeStepper</th>
<th>solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>pymor.algorithms.timestepping.ImmutableInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
<tr>
<td>pymor.algorithms.timestepping.BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

---

### Attributes

<table>
<thead>
<tr>
<th>pymor.algorithms.timestepping.ImmutableInterface</th>
<th>calculate_sid, sid, sid_failure, sid_ignore</th>
</tr>
</thead>
<tbody>
<tr>
<td>pymor.algorithms.timestepping.BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

---

**solve** *(initial_time, end_time, initial_data, operator, rhs=None, mass=None, mu=None, num_values=None)*

Apply time-stepper to the equation

\[
M \cdot d_t u + A(u, mu, t) = F(mu, t).
\]

**Parameters**

- **initial_time**  The time at which to begin time-stepping.
- **end_time**  The time until which to perform time-stepping.
- **initial_data**  The solution vector at *initial_time*.
- **operator**  The *Operator* $A$.
- **rhs**  The right hand side $F$ (either *VectorArray* of length 1 or *Operator* with *range.dim* == 1). If *None*, zero right hand side is assumed.
- **mass**  The *Operator* $M$. If *None*, the identity operator is assumed.
- **mu**  Parameter for which *operator* and *rhs* are evaluated. The current time is added to *mu* with key \_t.
- **num_values**  The number of returned vectors of the solution trajectory. If *None*, each intermediate vector that is calculated is returned.

**Returns**

*VectorArray* containing the solution trajectory.

---

**class**  pymor.algorithms.timestepping.ImplicitEulerTimeStepper *(nt, invert_options=None)*

**Bases:**  pymor.algorithms.timestepping.TimeStepperInterface

Implicit-Euler time-stepper.

Solves equations of the form

\[
M \cdot d_t u + A(u, mu, t) = F(mu, t).
\]

**Parameters**
The number of time-steps the time-stepper will perform.

invert_options The invert_options used to invert $M + dt^*A$.

Methods

<table>
<thead>
<tr>
<th>ImplicitEulerTimeStepper</th>
<th>ImmutableInterface</th>
<th>Bases:</th>
<th>pymor.core.interfaces.ImmutableInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>calculate_sid, sid, sid_failure, sid_ignore</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid,</td>
</tr>
<tr>
<td></td>
<td>with_arguments</td>
</tr>
</tbody>
</table>

solve

Applies time-stepper to the equation

$$M \times \frac{d}{dt} u + A(u, mu, t) = F(mu, t).$$

Parameters

- initial_time The time at which to begin time-stepping.
- end_time The time until which to perform time-stepping.
- initial_data The solution vector at initial_time.
- operator The Operator $A$.
- rhs The right hand side $F$ (either VectorArray of length 1 or Operator with range.dim == 1). If None, zero right hand side is assumed.
- mass The Operator $M$. If None, the identity operator is assumed.
- mu Parameter for which operator and rhs are evaluated. The current time is added to mu with key \_t.
- num_values The number of returned vectors of the solution trajectory. If None, each intermediate vector that is calculated is returned.

Returns

VectorArray containing the solution trajectory.

class pymor.algorithms.timestepping.TimeStepperInterface

Bases: pymor.core.interfaces.ImmutableInterface

Interface for time-stepping algorithms.

Algorithms implementing this interface solve time-dependent problems of the form

$$M \times \frac{d}{dt} u + A(u, mu, t) = F(mu, t).$$
Time-steppers used by `InstationaryDiscretization` have to fulfill this interface.

### Methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeStepper</td>
<td><code>interface</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>lock</code>, <code>unlock</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>add_attributes</code>, <code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>with_</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Interface</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td><code>calculate_sid</code>, <code>sid</code>, <code>sid_failure</code>, <code>sid_ignore</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>locked</code>, <code>logger</code>, <code>logging_disabled</code>, <code>name</code>, <code>uid</code>, <code>with_arguments</code></td>
</tr>
</tbody>
</table>

```python
solve(initial_time, end_time, initial_data, operator, rhs=None, mass=None, mu=None, num_values=None)
```

Apply time-stepper to the equation

\[
M \cdot \frac{d_t u}{dt} + A(u, mu, t) = F(mu, t).
\]

**Parameters**

- `initial_time` The time at which to begin time-stepping.
- `end_time` The time until which to perform time-stepping.
- `initial_data` The solution vector at `initial_time`.
- `operator` The `Operator` `A`.
- `rhs` The right hand side `F` (either `VectorArray` of length 1 or `Operator` with `range.dim == 1`). If `None`, zero right hand side is assumed.
- `mass` The `Operator` `M`. If `None`, the identity operator is assumed.
- `mu` Parameter for which `operator` and `rhs` are evaluated. The current time is added to `mu` with key `_t`.
- `num_values` The number of returned vectors of the solution trajectory. If `None`, each intermediate vector that is calculated is returned.

**Returns**

`VectorArray` containing the solution trajectory.

```python
pymor.algorithms.timestepping.explicit_euler(A, F, U0, t0, tl, nt, mu=None, num_values=None)
```

```python
pymor.algorithms.timestepping.implicit_euler(A, F, M, U0, t0, tl, nt, mu=None, invert_options=None, num_values=None)
```
pymor.analyticalproblems package

Submodules

advection module
class pymor.analyticalproblems.advection.InstationaryAdvectionProblem

```python
domain=RectDomain([[0
0], [1
1]]),
rhs=ConstantFunction(array(1.0)
2),
flux_function=ConstantFunction(array(0)
1),
flux_function_derivative=ConstantFunction(array(0)
1),
dirich-
let_data=ConstantFunction(array(2), ini-
tial_data=ConstantFunction(array(2), T=1, name=None)
```

Bases: pymor.core.interfaces.ImmutableInterface

Instationary advection problem.

The problem is to solve

\[
\begin{align*}
\frac{\partial u}{\partial t}(x, t, \mu) + f(u(x, t, \mu), t, \mu) &= s(x, t, \mu) \\
u(x, 0, \mu) &= u_0(x, \mu)
\end{align*}
\]

for u with t in [0, T], x in \(\Omega\).

Parameters

domain A DomainDescription of the domain \(\Omega\) the problem is posed on.

flux_function The Function \(f\). Note that the current time is handled as an additional \(\frac{\partial}{\partial t}\) component of the Parameter \(\mu\) which is passed to \(flux_function\) and is not part of the domain of the function. Moreover, \(flux_function.dim_domain\) has to be 1, whereas \(flux_function.shape_range\) has to be \((\text{dim } \Omega,)\).

flux_function_derivative The derivative of \(f\) with respect to \(u\).

rhs The Function \(s\). Note that the current time is handled as an additional \(\frac{\partial}{\partial t}\) component of the Parameter \(\mu\) which is passed to \(rhs\) and is not part of the domain of the function.

dirichlet_data Function providing the Dirichlet boundary values in global coordinates.

initial_data Function providing the initial values in global coordinates.

T The end time T.

name Name of the problem.

Methods

```
ImmutableInterface

BasicInterface

add_attributes, disable_logging, enable_logging,

has_interface_name, implementor_names, implementors, with_,

_setattr_
```

4.1. pymor package 25
Attributes

<table>
<thead>
<tr>
<th>InstationaryAdvectionProblem</th>
<th>dirichlet_data, domain, flux_function, flux_function_derivative, initial_data, rhs, T</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

domain

rhs

flux_function

flux_function_derivative

initial_data

dirichlet_data

T

burgers module

class pymor.analyticalproblems.burgers.Burgers2DBumpInitialData

Bases: pymor.functions.interfaces.FunctionInterface

Methods

Burgers2DBumpInitialData

FunctionInterface __call__, __setattr__

ImmutableInterface lock, unlock

BasicInterface add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__

Parametric build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

<table>
<thead>
<tr>
<th>Burgers2DBumpInitialData</th>
<th>dim_domain, shape_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

evaluate (x, mu=None)

Evaluate the function for given argument and Parameter.

class pymor.analyticalproblems.burgers.Burgers2DFlux (vx, vy)

Bases: pymor.functions.interfaces.FunctionInterface

Methods
Evaluate the function for given argument and Parameter.

class pymor.analyticalproblems.burgers.Burgers2DFluxDerivative (vx, vy)
Bases: pymor.functions.interfaces.FunctionInterface

Methods
Burgers2DFluxDerivative
evaluate
FunctionInterface
__call__
ImmutableInterface
lock, unlock
BasicInterface
add_attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, with_,
__setattr__
Parametric
build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes
Burgers2DFlux

dim_domain, shape_range
ImmutableInterface
calculate_sid, sid, sid_failure, sid_ignore
BasicInterface
locked, logger, logging_disabled, name, uid, with_arguments
Parametric
parameter_local_type, parameter_space, parameter_type,
parametric

evaluate (U, mu=None)
Evaluate the function for given argument and Parameter.

class pymor.analyticalproblems.burgers.Burgers2DProblem (vx=1.0, vy=1.0, torus=True,
initial_data_type='sin',
parameter_range=(1.0, 2.0))
Bases: pymor.analyticalproblems.advection.InstationaryAdvectionProblem

Two-dimensional Burgers-type problem.
The problem is to solve

4.1. pymor package
\[ \frac{\partial u(x, t, \mu)}{\partial t} + (v \cdot u(x, t, \mu))^\mu = 0 \]
\[ u(x, 0, \mu) = u_0(x) \]

for \( u \) with \( t \) in \([0, 0.3]\), \( x \) in \([0, 2] \times [0, 1]\).

**Parameters**

- **vx** The \( x \) component of the velocity vector \( v \).
- **vy** The \( y \) component of the velocity vector \( v \).
- **torus** If `True` impose periodic boundary conditions. Otherwise Dirichlet left and bottom, outflow top and right.
- **initial_data_type** Type of initial data (`'sin'` or `'bump'`).
- **parameter_range** The interval in which \( \mu \) is allowed to vary.

**Methods**

- **ImmutableInterface** `lock`, `unlock`
- **BasicInterface** `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`

**Attributes**

- **InstationaryAdvectionProblem** `dirichlet_data`, `domain`, `flux_function`, `flux_function_derivative`, `initial_data`, `rhs`, `T`
- **ImmutableInterface** `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- **BasicInterface** `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`

---

**class** `pymor.analyticalproblems.burgers.Burgers2DSinInitialData`

**Bases:** `pymor.functions.interfaces.FunctionInterface`

**Methods**

- **Burgers2DSinInitialData** `evaluate`, `evaluate_derivative`
- **FunctionInterface** `__call__`
- **ImmutableInterface** `lock`, `unlock`
- **BasicInterface** `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`
- **Parametric** `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

**Attributes**

- **Burgers2DSinInitialData** `dim_domain`, `shape_range`
- **ImmutableInterface** `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- **BasicInterface** `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`
- **Parametric** `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

**evaluate** `(x, mu=None)`

Evaluate the function for given argument and `Parameter`.  

---

Chapter 4. API Documentation
class pymor.analyticalproblems.burgers.BurgersBumpInitialData
Bases: pymor.functions.interfaces.FunctionInterface

Methods

<table>
<thead>
<tr>
<th>BurgersBumpInitialData</th>
<th>evaluate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionInterface</td>
<td><strong>call</strong></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>BurgersBumpInitialData</th>
<th>dim_domain, shape_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**evaluate** *(x, mu=None)*

Evaluate the function for given argument and **Parameter**.

---

class pymor.analyticalproblems.burgers.BurgersFlux(v)
Bases: pymor.functions.interfaces.FunctionInterface

Methods

<table>
<thead>
<tr>
<th>BurgersFlux</th>
<th>evaluate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionInterface</td>
<td><strong>call</strong></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>BurgersFlux</th>
<th>dim_domain, shape_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**evaluate** *(U, mu=None)*

Evaluate the function for given argument and **Parameter**.

---

4.1. pymor package
class pymor.analyticalproblems.burgers.BurgersFluxDerivative(v)
Bases: pymor.functions.interfaces.FunctionInterface

Methods

BurgersFluxDerivative
FunctionInterface
__call__
ImmutableInterface
unlock
BasicInterface
add_attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, with_,
__setattr__
Parametric
build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes

BurgersFluxDerivative
dim_domain, shape_range
ImmutableInterface
calculate_sid, sid, sid_failure, sid_ignore
BasicInterface
locked, logger, logging_disabled, name, uid,
with_arguments
Parametric
parameter_local_type, parameter_space, parameter_type,
parametric

evaluate(U, mu=None)
Evaluate the function for given argument and Parameter.

class pymor.analyticalproblems.burgers.BurgersProblem(v=1.0, circle=True, initial_data_type='sin', parameter_range=(1.0, 2.0))
Bases: pymor.analyticalproblems.advection.InstationaryAdvectionProblem

One-dimensional Burgers-type problem.
The problem is to solve

\[ \frac{\partial u(x, t, \mu)}{\partial t} + \frac{\partial}{\partial x} (v \cdot u(x, t, \mu)^\mu) = 0 \]
\[ u(x, 0, \mu) = u_0(x) \]

for \( u \) with \( t \) in \([0, 0.3]\), \( x \) in \([0, 2]\).

Parameters

v  The velocity \( v \).
circle  If \( True \) impose periodic boundary conditions. Otherwise Dirichlet left, outflow right.
initial_data_type  Type of initial data (‘sin’ or ‘bump’).
parameter_range  The interval in which \( \mu \) is allowed to vary.

Methods

ImmutableInterface
unlock
BasicInterface
add_attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, with_,
__setattr__
Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstationaryAdvectionProblem</td>
<td>flux_function, flux_function_derivative, initial_data, rhs, T</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

class pymor.analyticalproblems.burgers.BurgersSinInitialData

Bases: pymor.functions.interfaces.FunctionInterface

Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>BurgersSinInitialData</td>
<td>evaluate</td>
</tr>
<tr>
<td>FunctionInterface</td>
<td><strong>call</strong>, <strong>setattr</strong></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>unlock, add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><strong>call</strong>, has_interface_name, implementor_names, implementors, with_</td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BurgersSinInitialData</td>
<td>dim_domain, shape_range</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

evaluate(x, mu=None)

Evaluate the function for given argument and Parameter.

elegant module

class pymor.analyticalproblems.elliptic.EllipticProblem

Bases: pymor.core.interfaces.ImmutableInterface

Affinely decomposed linear elliptic problem.
The problem consists in solving

4.1. pymor package
\( K \) for \( u \).

**Parameters**

- **domain** A `DomainDescription` of the domain the problem is posed on.
- **rhs** The `Function` \( f(x, \mu) \). `rhs.dim_domain` has to agree with the dimension of `domain`, whereas `rhs.shape_range` has to be tuple().
- **diffusion_functions** List containing the `Functions` \( d_k(x) \), each having `shape_range` of either `tuple()` or `(dim domain, dim domain)`.
- **diffusion_functionals** List containing the `ParameterFunctionals` \( \theta_k(\mu) \). If `len(diffusion_functions) == 1`, `diffusion_functionals` is allowed to be `None`, in which case no parameter dependence is assumed.
- **dirichlet_data** `Function` providing the Dirichlet boundary values in global coordinates.
- **neumann_data** `Function` providing the Neumann boundary values in global coordinates.
- **parameter_space** `ParameterSpace` for the problem.
- **name** Name of the problem.

**Methods**

- `ImmutableInterface.lock`, `ImmutableInterface.unlock`
- `BasicInterface.add_attributes`, `BasicInterface.disable_logging`, `BasicInterface.enable_logging`, `BasicInterface.has_interface_name`, `BasicInterface.implementor_names`, `BasicInterface.implementors`, `BasicInterface.with_`, `__setattr__`

**Attributes**

- `EllipticProblem.diffusion_functionals`, `EllipticProblem.diffusion_functions`, `EllipticProblem.dirichlet_data`, `EllipticProblem.domain`, `EllipticProblem.rhs`
- `ImmutableInterface.calculate_sid`, `ImmutableInterface.sid`, `ImmutableInterface.sid_failure`, `ImmutableInterface.sid_ignore`
- `BasicInterface.locked`, `BasicInterface.logger`, `BasicInterface.logging_disabled`, `BasicInterface.name`, `BasicInterface.uid`, `BasicInterface.with_arguments`

**thermalblock module**

```python
class pymor.analyticalproblems.thermalblock.ThermalBlockDiffusionFunction(x, y, nx, ny):
    pass
```

Bases: `pymor.functions.interfaces.FunctionInterface`
**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>evaluate</code></td>
<td>Evaluate the function for given argument and Parameter.</td>
</tr>
<tr>
<td><code>build_parameter_type</code>, <code>local_parameter</code>, <code>parse_parameter</code>, <code>strip_parameter</code></td>
<td></td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dim_domain</code>, <code>shape_range</code></td>
<td></td>
</tr>
<tr>
<td><code>calculate_sid</code>, <code>sid</code>, <code>sid_failure</code>, <code>sid_ignore</code></td>
<td></td>
</tr>
<tr>
<td><code>locked</code>, <code>logger</code>, <code>logging_disabled</code>, <code>name</code>, <code>uid</code>, <code>with_arguments</code></td>
<td></td>
</tr>
</tbody>
</table>

**evaluate** \((x, \mu=None)\)

Evaluate the function for given argument and Parameter.

```python
class pymor.analyticalproblems.thermalblock.ThermalBlockProblem(num_blocks=(3, 3), parameter_range=(0.1, 1), rhs=ConstantFunction(array(1.0), 2))
```

**Bases:** `pymor.analyticalproblems.elliptic.EllipticProblem`

Analytical description of a 2D ‘thermal block’ diffusion problem.

This problem is to solve the elliptic equation

\[- \left[ \frac{d}{d(x, \mu)} u(x, \mu) \right] = f(x, \mu)\]

on the domain \([0,1]^2\) with Dirichlet zero boundary values. The domain is partitioned into \(nx \times ny\) blocks and the diffusion function \(d(x, \mu)\) is constant on each such block \((i,j)\) with value \(\mu_{ij}\).

```
<table>
<thead>
<tr>
<th>\mu_{11}</th>
<th>\mu_{12}</th>
<th>\mu_{13}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\mu_{21}</td>
<td>\mu_{22}</td>
<td>\mu_{23}</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
```

The Problem is implemented as an `EllipticProblem` with the characteristic functions of the blocks as `diffusion_functions`.

**Parameters**

- **num_blocks** The tuple \((nx, ny)\)
- **parameter_range** A tuple \((\mu_{\text{min}}, \mu_{\text{max}})\). Each `Parameter` component \(\mu_{ij}\) is allowed to lie in the interval \([\mu_{\text{min}}, \mu_{\text{max}}]\).
**rhs**  The Function $f(x, \mu)$.

### Methods

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_</td>
</tr>
<tr>
<td></td>
<td><strong>setattr</strong></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>EllipticProblem</th>
<th>diffusion_functionals, diffusion_functions, dirichlet_data, domain, rhs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

## pymor.core package

### Submodules

**backports module**  This module contains pure python backports of library features from python >= 3 to 2.7

**class pymor.core.backports.abstractclassmethod(calling_method)**

**Bases:** classmethod

A decorator indicating abstract classmethods.

Similar to abstractmethod.

**Usage:**

```python
class C(metaclass=ABCMeta):
    @abstractclassmethod
    def my_abstract_classmethod(cls, ...):
        ...
```

‘abstractclassmethod’ is deprecated. Use ‘classmethod’ with ‘abstractmethod’ instead.

### Methods

<table>
<thead>
<tr>
<th>classmethod</th>
<th><strong>new</strong></th>
</tr>
</thead>
</table>

**class pymor.core.backports.abstractstaticmethod(calling_method)**

**Bases:** staticmethod

A decorator indicating abstract staticmethods.

Similar to abstractmethod.

**Usage:**

```python
class C(metaclass=ABCMeta):
    @abstractstaticmethod
    def my_abstract_staticmethod(...):
        ...
```

Methods

| staticmethod | __new__ |

**cache module**  This module provides the caching facilities of pyMOR.

Any class that wishes to provide cached method calls should derive from **CacheableInterface**. Methods which are to be cached can then be marked using the **cached** decorator.

To ensure consistency, **CacheableInterface** derives from **ImmutableInterface**: The return value of a cached method call should only depend on its arguments as well as the immutable state of the class instance.

Making this assumption, the keys for cache lookup are created from the following data:

1. the instance’s state id (see **ImmutableInterface**) if available, else the instance’s unique id (see **BasicInterface**),
2. the method’s **__name__**,
3. the state id of each argument if available, else its pickled state.
4. the state of pyMOR’s global **defaults**.

Note, however, that instances of **ImmutableInterface** are allowed to have mutable private attributes. It is the implementors responsibility not to break things. (See this warning.)

Backends for storage of cached return values derive from **CacheRegion**. Currently two backends are provided for memory-based and disk-based caching (**MemoryRegion** and **SQLiteRegion**). The available regions are stored in the module level **cache_regions** dict. The user can add additional regions (e.g. multiple disk cache regions) as required. **CacheableInterface** has a **region** attribute through which a key of the **cache_regions** dict can provided to select a cache region which should be used by the instance. (Setting **region** to **None** or **’none’** disables caching.)

By default, a ‘memory’ and a ‘disk’ cache region are automatically configured. The path and maximum size of the disk region as well as the maximum number of keys of the memory cache region can be configured via the `pymor.core.cache._setup_default_regions.disk_path`, `pymor.core.cache._setup_default_regions.disk_max_size` and `pymor.core.cache._setup_default_regions.memory_max_keys` **defaults**. (Note that changing these defaults will result in changed state ids, so moving a disk cache and changing the default path accordingly will result in cache misses.) As an alternative, these defaults can be overridden by the **PYMOR_CACHE_PATH**, **PYMOR_CACHE_MAX_SIZE** and **PYMOR_CACHE_MEMORY_MAX_KEYS** environment variables. (These variables do not enter state id calculation and are therefore the preferred way to configure caching.)

There are multiple ways to disable and enable caching in pyMOR:

1. Calling **disable_caching**(enable_caching).
2. Setting **cache_regions[region].enabled** to False or True.
3. Calling **CacheableInterface.disable_caching**(CacheableInterface.enable_caching).

Caching of a method is only active, if caching is enabled on global, region and instance level. For debugging purposes, it is moreover possible to set the environment variable **PYMOR_CACHE_DISABLE=1** which overrides any call to **enable_caching**.

A cache region can be emptied using **CacheRegion.clear**. The function **clear_caches** clears each cache region registered in **cache_regions**.

4.1. **pymor package**
class pymor.core.cache.CacheRegion
Bases: object

Base class for all pyMOR cache regions.

Methods

- **CacheRegion.clear**, **get**, **set**

Attributes

- **CacheRegion.enabled**
  If `False` caching is disabled for this region.

- **clear()**
  Clear the entire cache region.

class pymor.core.cache.CacheableInterface
Bases: pymor.core.interfaces.ImmutableInterface

Base class for anything that wants to use our built-in caching.

Methods

- **CacheableInterface.disable_caching**, **enable_caching**
- **ImmutableInterface.lock**, **unlock**
- **BasicInterface.add_attributes**, **disable_logging**, **enable_logging**, **has_interface_name**, **implementor_names**, **implementors**, **with_**, **__setattr__**

Attributes

- **CacheableInterface.cache_region**
- **ImmutableInterface.calculate_sid**, **sid**, **sid_failure**, **sid_ignore**
- **BasicInterface.locked**, **logger**, **logging_disabled**, **name**, **uid**, **with_arguments**

cache_region
Name of the CacheRegion to use. Must correspond to a key in pymor.core.cache.cache_regions. If `None` or `none`, caching is disabled.

disable_caching()
Disable caching for this instance.

enable_caching(region)
Enable caching for this instance.

Parameters

- **region** Name of the CacheRegion to use. Must correspond to a key in pymor.core.cache.cache_regions. If `None` or `none`, caching is disabled.
class pymor.core.cache.MemoryRegion(max_keys)
Bases: pymor.core.cache.CacheRegion

Methods
MemoryRegion clear, get, set

Attributes
MemoryRegion NO_VALUE
CacheRegion enabled

class pymor.core.cache.SQLiteRegion(path, max_size)
Bases: pymor.core.cache.CacheRegion

Methods
SQLiteRegion clear, get, housekeeping, set

Attributes
SQLiteRegion enabled

class pymor.core.cache.cached(function)
Bases: object

Decorator to make a method of CacheableInterface actually cached.

Methods
cached __call__, __get__

__call__(im_self, *args, **kwargs)
Via the magic that is partial functions returned from __get__, im_self is the instance object of the class
we're decorating a method of and [kw]args are the actual parameters to the decorated method

__get__(instance, instancetype)
Implement the descriptor protocol to make decorating instance method possible. Return a partial function
where the first argument is the instance of the decorated instance object.

pymor.core.cache.clear_caches()
Clear all cache regions.
	pymor.core.cache.disable_caching()
Globally disable caching.
	pymor.core.cache.enable_caching()
Globally enable caching.
**decorators module**  Created on Fri Nov 2 10:12:55 2012 Collection of function/class based decorators.

---

class pymor.coredecorators.**DecoratorBase**(func)

Bases: object

A base for all decorators that does the common automagic

Methods

```python
DecoratorBase __get__
```

```python
__get__ (obj, ownerClass=None)
```

Return a wrapper that binds self as a method of obj (!)

---

class pymor.coredecorators.**DecoratorWithArgsBase**

Bases: object

A base for all decorators with args that sadly can do little common automagic

---

class pymor.coredecorators.** Deprecated**(alt='no alternative given')

Bases: pymor.coredecorators.**DecoratorBase**

This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

---

```python
DecoratorBase __get__
```

---

pymor.coredecorators.**contains_contract**(string)

---

pymor.coredecorators.**contract**(arg, **kwargs)

Decorator for adding contracts to functions.

It is smart enough to support functions with variable number of arguments and keyword arguments.

There are three ways to specify the contracts. In order of precedence:

- As arguments to this decorator. For example:

  ```python
  @contract(a='int,>0',b='list[N],N>0',returns='list[N]')
  def my_function(a, b):
      # ...
      pass
  ```

- As annotations (supported only in Python 3):
Using :type: and :rtype: tags in the function’s docstring:

```python
@contract
def my_function(a: 'int, >0', b: 'list[N], N>0') -> 'list[N]':
    # ...
pass
```

```python
pymor.core.decorators.fixup_docstring(doc)
```

This replaces all dots with underscores in contract lines. This is necessary to circumvent type identifier checking in pycontracts itself.

### defaults module

This module contains pyMOR’s facilities for handling default values.

A default value in pyMOR is always the default value of some function argument. To mark the value of an optional function argument as an user-modifiable default value, use the :func:`defaults:` decorator. As an additional feature, if `None` is passed as value for such an argument, its default value is used instead of `None`. This is useful for writing code of the following form:

```python
@default('option')
def algorithm(U, option=42):
    ...

def method_called_by_user(V, option_for_algorithm=None):
    ...
    algorithm(U, option=option_for_algorithm)
    ...
```

If the user does not provide `option_for_algorithm` to `method_called_by_user`, the default `42` is automatically chosen without the implementor of `method_called_by_user` having to care about this.

The user interface for handling default values in pyMOR is provided by `set_defaults`, `load_defaults_from_file`, `write_defaults_to_file` and `print_defaults`.

If pyMOR is imported, it will automatically search for configuration files named `pymor_defaults.py` in the current working directory. The first file found will be loaded via `load_defaults_from_file`. However, for your security, this file will only be loaded, if it is owned by the user running the Python interpreter. (`load_defaults_from_file` uses `exec` to load the configuration.) As an alternative, the environment variable `PYMOR_DEFAULTS` can be used to specify the path of a configuration file. If empty or set to `NONE`, no configuration file will be loaded whatsoever.

**Warning:** The state of pyMOR’s global defaults enters the calculation of each state id (see `pymor.core.interfaces`). Thus, if you first instantiate an immutable object and then change the defaults, the resulting object will have a different state id than if you first change the defaults. (This is necessary as the object can save internal state upon initialization, which depends on the state of the global defaults.) As a consequence, the key generated for `caching` will depend on the time the defaults have been changed. While no wrong results will be produced, changing defaults at different times will cause unnecessary cache misses and will pollute the cache with duplicate entries.

As a rule of thumb, defaults should be set once and for all at the start of a pyMOR application. Anything else should be considered a dirty hack. (pyMOR will warn you if it gets the impression you are trying to hack it.)
class pymor.core.defaults.DefaultContainer
    Bases: object

    Internal singleton class holding all default values defined in pyMOR.
    Not to be used directly.

Methods

    DefaultContainer | check_consistency, get, import_all, keys, update

pymor.core.defaults.defaults(*args, **kwargs)
    Function decorator for marking function arguments as user-configurable defaults.

    If a function decorated with defaults is called, the values of the marked default parameters are set to the values defined via load_defaults_from_file or set_defaults if no value is provided by the caller of the function. If no value is specified using these methods, the default value provided by in the function signature is used.

    Moreover, if None is passed as a value for a default argument, the argument is set to its default value, as well.

    If the argument arg of function f in sub-module m of package p is marked as a default value, its value will be changeable by the aforementioned methods under the path p.m.f.arg.

    The defaults decorator can also be used for user code.

Parameters

    args  List of strings containing the names of the arguments of the decorated function to mark as pyMOR defaults.

    Each of these arguments has to be a keyword argument (with a default value).

    **kwargs  If a method of a class is decorated, the fully qualified name of the method should be provided, as this name cannot be derived at decoration time in Python 2.

pymor.core.defaults.defaults_sid()
    Return a state id for pyMOR’s global defaults.

    This method is used for the calculation of state ids of immutable objects (see pymor.core.interfaces) and for cache key generation.

    For other uses see the implementation of pymor.operators.numpy.NumpyMatrixBasedOperator.assemble.

pymor.core.defaults.load_defaults_from_file(filename='./pymor_defaults.py')
    Loads default values define in a configuration file.

    Such configuration files can be created via write_defaults_to_file. The file is loaded via Python’s exec function, so be very careful with configuration files you have not created your own. You have been warned!

    (Note that defaults should only be changed/loaded before state ids have been calculated. See this warning for details.)

Parameters

    filename  Path of the configuration file.
pymor.core.defaults.print_defaults(import_all=True, shorten_paths=2)
Print all default values set in pyMOR.

Parameters
import_all While print_defaults will always print all defaults defined in loaded configuration files or set via set_defaults, default values set in the function signature can only be printed, if the modules containing these functions have been imported. If import_all is set to True, print_defaults will therefore first import all of pyMOR’s modules, to provide a complete lists of defaults.

shorten_paths Shorten the paths of all default values by shorten_paths components. The last two path components will always be printed.

pymor.core.defaults.set_defaults(defaults, check=True)
Set default values.
This method sets the default value of function arguments marked via the defaults decorator, overriding default values specified in the function signature or set earlier via load_defaults_from_file or previous set_defaults calls.

(Note that defaults should only be changed/load before state ids have been calculated. See this warning for details.)

Parameters
defaults Dictionary of default values. Keys are the full paths of the default values. (See defaults.)
check If True, recursively import all packages associated to the paths of the set default values. Then check if defaults with the provided paths have actually been defined using the defaults decorator.

pymor.core.defaults.write_defaults_to_file(filename='./pymor_defaults.py', packages=('pymor',), code=True, file=True, user=True)
Write the currently set default values in pyMOR to a configuration file.
The resulting file is an ordinary Python script and can be modified by the user at will. It can be loaded in a later session using load_defaults_from_file.

Parameters
filename Name of the file to write to.
packages List of package names. To discover all default values that have been defined using the defaults decorator, write_defaults_to_file will recursively import all sub-modules of the named packages before creating the configuration file.

code If False, ignore default values, defined in function signatures.
file If False, ignore default values loaded from a configuration file.
user If False, ignore default values provided via set_defaults.

dynamic module This is an empty module usable as a placeholder(-dict) for dynamically created types and such
exceptions module

**class** pymor.core.exceptions.**AccuracyError**
* Bases: exceptions.Exception

Is raised if the result of a computation is inaccurate

**Methods**

```python
Exception __new__
```

**Attributes**

```plaintext
BaseException args, message
```

**class** pymor.core.exceptions.**ConstError**
* Bases: exceptions.Exception

I get thrown when you try to add a new member to a locked class instance

**Methods**

```python
Exception __new__
```

**Attributes**

```plaintext
BaseException args, message
```

**class** pymor.core.exceptions.**ExtensionError**
* Bases: exceptions.Exception

Is raised if a (basis) extension algorithm fails.

This will mostly happen during a basis extension when the new snapshot is already in the span of the basis.

**Methods**

```python
Exception __new__
```

**Attributes**

```plaintext
BaseException args, message
```

**class** pymor.core.exceptions.**InversionError**
* Bases: exceptions.Exception

Is raised if an operator inversion algorithm fails.

**Methods**

```python
Exception __new__
```
Internally, pyMOR, Release 0.3.2

**Attributes**

```python
BaseException | args, message
```

**class** `pymor.core.exceptions.NewtonError`

**Bases:** `exceptions.Exception`

Is raised if the Newton algorithm fails to converge.

**Methods**

```python
Exception | __new__
```

**Attributes**

```python
BaseException | args, message
```

**Interfaces module** This module provides base classes from which most classes in pyMOR inherit.

The purpose of these classes is to provide some common functionality for all objects in pyMOR. The most notable features provided by `BasicInterface` are the following:

1. `BasicInterface` sets class `UberMeta` as metaclass which itself inherits from `abc.ABCMeta`, so it is possible to define interface classes with abstract methods using the `abstractmethod` decorator. There are also decorators for abstract class methods, static methods, and properties.

2. Using metaclass magic, each `class` deriving from `BasicInterface` comes with its own logger instance accessible through its `logger` attribute. The logger prefix is automatically set to the class name.

3. Logging can be disabled and re-enabled for each instance using the `BasicInterface.disable_logging` and `BasicInterface.enable_logging` methods.

4. An instance can be made immutable using `BasicInterface.lock`. If an instance is locked, each attempt to change one of its attributes raises an exception. Private attributes (of the form `_name`) are exempted from this rule. Locked instances can be unlocked again using `BasicInterface.unlock`.

5. `BasicInterface.with_` can be used to create a copy of an instance with some changed attributes. E.g.

```python
obj.with_(a=x, b=y)
```

creates a copy with the `a` and `b` attributes of `obj` set to `x` and `y`. (Note that in general `a` and `b` do not necessarily have to correspond to class attributes of `obj`; it is up to the implementor to interpret the provided arguments.) `BasicInterface.with_arguments` holds the set of allowed arguments.

`BasicInterface` provides a default implementation of `with_` which works as follows:

- The argument names of the classes `__init__` method are looked up. If the instance has an attribute of the same name for each `__init__` argument, `with_arguments` returns the argument names of `__init__`, otherwise an empty set is returned and the `with_` functionality is disabled.

- If the above condition is satisfied, a call to `with_` results in the creation of a new instance where the arguments of `with_` are passed through to `__init__`. The missing `__init__` arguments are taken from the corresponding instance attributes.

6. `BasicInterface.uid` provides a unique id for each instance. While `id(obj)` is only guaranteed to be unique among all living Python objects, `BasicInterface.uid` will be (almost) unique among all pyMOR objects that have ever existed, including previous runs of the application. This is achieved by building the id from a

4.1. **pymor package** 43
uuid4 which is newly created for each pyMOR run and a counter which is increased for any object that requests an uid.

7. If not set by the user to another value, BasicInterface.name is generated from the class name and the uid of the instance.

ImmutableInterface derives from BasicInterface and adds the following functionality:

1. Using more metaclass magic, each instance which derives from ImmutableInterface is locked after its __init__ method has returned.

2. If possible, a unique state id for the instance is calculated and stored as sid attribute. If sid calculation fails, sid_failure is set to a string giving a reason for the failure.

The basic idea behind state ids is that for an immutable object the result of any member function call is already pre-determined by the objects state id, the function’s arguments and the state of pyMOR’s global defaults.

The sid is constructed as a tuple containing:

- the class of the instance
- for each __init__ argument its name and
  - its sid if it has one
  - its value if it is an instance of NoneType, str, int, float or bool
  - its value if it is a numpy array of short size

For tuple, list or dict instances, the calculation is done by recursion. If none of these cases apply, sid calculation fails.

- the state of all defaults which have been set by the user

| Warning: | Default values defined in the function signature do not enter the sid calculation. Thus, if you change default values in your code, pyMOR will not be aware of these changes! As a consequence, you should always take care to clear your cache if you change in-code default values. |

Note that a sid contains only object references to the sids of the provided __init__ arguments. This structure is preserved by pickling resulting in relatively short string representations of the sid.

3. ImmutableInterface.sid_ignore can be set to a tuple of __init__ argument names, which should be excluded from sid calculation.

4. sid generation (with all its overhead) can be disabled by setting ImmutableInterface.calculate_sid to False.

5. sid generation can be disabled completely in pyMOR by calling disable_sid_generation. It can be activated again by calling enable_sid_generation.

---

class pymor.core.interfaces.BasicInterface

Bases: object

Base class for most classes in pyMOR.

Methods

BasicInterface.add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with___setattr__
Attributes

| BasicInterface | locked, logger, logging_disabled, name, uid, with_arguments |

**locked**

*True* if the instance is made immutable using *lock*.

**logger**

A per-class instance of *logging.Logger* with the class name as prefix.

**logging_disabled**

*True* if logging has been disabled.

**name**

The name of the instance. If not set by the user, the name is generated from the class name and the *uid* of the instance.

**uid**

A unique id for each instance. The uid is obtained by using *UIDProvider* and should be unique for all pyMOR objects ever created.

**with_arguments**

Set of allowed keyword arguments for *with_*.

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

**add_attributes(**kwargs**)**

Add attributes to a locked instance.

**disable_logging**(doit=True)

Disable logging output for this instance.

**enable_logging**(doit=True)

Enable logging output for this instance.

**classmethod has_interface_name()**

*True* if the class name ends with *Interface*. Used for introspection.

**classmethod implementor_names**(descend=False)

For convenience I return a list of my implementor names instead of class objects

**classmethod implementors**(descend=False)

I return a, potentially empty, list of my subclass-objects. If descend is True I traverse my entire subclass hierarchy and return a flattened list.

**lock**(doit=True)

Make the instance immutable.

Trying to change an attribute after locking raises a *ConstError*. Private attributes (of the form *_attribute*) are exempted from this rule.

**unlock()**

Make the instance mutable again, after it has been locked using *lock*.

**with_(**kwargs**)**

Returns a copy with changed attributes.

The default implementation is to call *_with_via_init(**kwargs)**.

---

4.1. pymor package
**kwargs Names of attributes to change with their new values. Each attribute name has to be contained in `with_arguments`.

Returns
Copy of `self` with changed attributes.

class pymor.core.interfaces.ImmutableInterface
Bases: pymor.core.interfaces.BasicInterface
Base class for immutable objects in pyMOR.

Instances of `ImmutableInterface` are immutable in the sense that they are `locked` after `__init__` returns.

Warning: For instances of `ImmutableInterface`, the following should always be true

- The result of any member function call is determined by the function's arguments together with the state id of the corresponding instance and the current state of pyMOR's global defaults.
- While, in principle, you are allowed to modify private members after instance initialization, this should never affect the outcome of future method calls. In particular, if you update any internal state after initialization, you have to ensure that this state is not affected by possible changes of the global `defaults`.

Methods
- `ImmutableInterface` lock
- `BasicInterface` add_attributes, disable_logging, enable_logging,
  has_interface_name, implementor_names, implementors, with_,
  __setattr__

Attributes
- `ImmutableInterface` calculate_sid, sid, sid_failure, sid_ignore
- `BasicInterface` locked, logger, logging_disabled, name, uid,
  with_arguments

calculate_sid
If `True`, a unique id describing the state of the instance is calculated after `__init__` returns, based on the states of the provided arguments. For further details see `pymor.core.interfaces`.

sid
The objects state id. If sid generation is disabled or fails, this attribute is not set.

sid_failure
If sid generation fails, a string describing the reason for the failure.

sid_ignore
Tuple of `__init__` arguments not to include in sid calculation. The default it `{'name', `cache_region'}`

__metaclass__
alias of `ImmutableMeta`

lock (doit=True)
Make the instance immutable.
Trying to change an attribute after locking raises a `ConstError`. Private attributes (of the form `_attribute`) are exempted from this rule.

```python
unlock()
```

Make the instance mutable again, after it has been locked using `lock`.

---

```python
class pymor.core.interfaces.ImmutableMeta(name, bases, namespace)
    Bases: pymor.core.interfaces.UberMeta
    Metaclass for ImmutableInterface.

    Methods
    ABCMeta  register, __instancecheck__, __subclasscheck__,
    type     mro, __subclasses__

    Attributes
    ImmutableMeta  init_arguments_never_warn, sids_created
```

---

```python
class pymor.core.interfaces.UID
    Bases: object
    Provides unique, quickly computed ids by combining a session UUID4 with a counter.

    Attributes
    UID  counter, prefix, uid
```

---

```python
class pymor.core.interfaces.UberMeta(name, bases, namespace)
    Bases: abc.ABCMeta

    Methods
    UberMeta  __new__
    ABCMeta   register, __instancecheck__, __subclasscheck__,
    type     mro, __subclasses__

    static  __new__ (classname, bases, classdict)
    I copy contract decorations and docstrings from base class methods to deriving classes. I also forward
    “abstract{class|static}method” decorations in the base class to “[class|static]method” decorations in the
    new subclass.

    Copying of docstrings can be prevented by setting the `PYMOR_COPY_DOCSTRINGS_DISABLE` environ-
    ment variable to `1`.

    class pymor.core.interfaces.abstractclassmethod(callable_method)
    Bases: pymor.core.backports.abstractclassmethod

    I mark my wrapped function with an additional `__isabstractclassmethod__` member, where my
    abstractclassmethod_base sets `__isabstractmethod__` = True.

    Methods
    classmethod  __new__
```

4.1. pymor package
class `pymor.core.interfaces.abstractstaticmethod` *(callable_method)*  
Bases: `pymor.core.backports.abstractstaticmethod`  
I mark my wrapped function with an additional `__isabstractstaticmethod__` member, where my abstractclass-method_base sets `__isabstractmethod__ = True`.

Methods

| staticmethod | `__new__` |

`pymor.core.interfaces.disable_sid_generation()`  
Globally disable the generation of state ids.

`pymor.core.interfaces.enable_sid_generation()`  
Globally enable the generation of state ids.

`pymor.core.interfaces.inject_sid(obj, context, *args)`  
Add a state id sid to an object.  
The purpose of this methods is to inject state ids into objects which do not derive from `ImmutableInterface`. If `obj` is an instance of `BasicInterface`, it is locked, if it is an `numpy.ndarray`, its `writable` flag is set to `False`.  
It is the callers responsibility to ensure that the given parameters uniquely describe the state of `obj`, and that `obj` does not change its state after the call of `inject_sid`. For an example see `pymor.analyticalproblems.EllipticProblem`.  

**Parameters**  
`obj` The object which shall obtain a sid.  
`context` A hashable, picklable, immutable object, describing the context in which `obj` was created.  
`*args` List of parameters which in the given context led to the creation of `obj`.

**logger module**  

class `pymor.core.logger.ColoredFormatter`  
Bases: `logging.Formatter`  
A logging.Formatter that inserts tty control characters to color loglevel keyword output. Coloring can be disabled by setting the `PYMOR_COLORS_DISABLE` environment variable to `1`.

**Methods**

| ColoredFormatter | `format, formatTime` |
| Formatter         | `converter, formatException, usesTime` |
class pymor.core.logger.DummyLogger

Bases: object

Methods

DummyLogger: critical, debug, error, exception, getChild, getEffectiveLevel, info, isEnabledFor, log, nop, warn, warning

Attributes

DummyLogger: propagate

pymor.core.logger.formatter_message (message, use_color)

pymor.core.logger.getLogger (module, level=None, filename='')

Get the logger of the respective module for pyMOR’s logging facility.

Parameters

module Name of the module.
level If set, logger.setLevel(level) is called (see setLevel).
filename If not empty, path of an existing file where everything logged will be written to.

Defaults

filename (see pymor.core.defaults)

pymor.core.logger.set_log_levels (levels={'pymor': 'INFO'})

Set log levels for pyMOR’s logging facility.

Parameters

levels Dict of log levels. Keys are names of loggers (see logging.getLogger), values are the log levels to set for the loggers of the given names (see setLevel).

Defaults

levels (see pymor.core.defaults)

pickle module This module contains methods for object serialization.

Instead of importing serialization functions from Python’s pickle module directly, you should use the dump, dumps, load, loads functions defined here.

Moreover, dumps_function provides a way to serialize function objects which cannot be serialized by dumps. Note, however, that its use should be avoided since it uses non-portable implementation details of CPython to achieve its goals.

4.1. pymor package 49
class pymor.core.pickle.Module (mod)
    Bases: object

pymor.core.pickle._global_names (code_object)
    Return all names in code_object.co_names which are used in a LOAD_GLOBAL statement.

pymor.core.pickle.dumps_function (function)
    Tries hard to pickle a function object:
    1. The function’s code object is serialized using the marshal module.
    2. For all global names used in the function’s code object the corresponding object in the function’s global
       namespace is pickled. In case this object is a module, the modules __package__ name is pickled.
    3. All default arguments are pickled.
    4. All objects in the function’s closure are pickled.
    Note that also this is heavily implementation specific and will probably only work with CPython. If possible,
    avoid using this method.

pymor.core.pickle.loads_function (s)
    Restores a function serialized with dumps_function.

pymor.discretizations package

Submodules

basic module
class pymor.discretizations.basic.DiscretizationBase (operators, functionals, vector_operators, products=None, estimator=None, visualizer=None, cache_region='disk', name=None)
    Bases: pymor.discretizations.interfaces.DiscretizationInterface
    Base class for Discretizations providing some common functionality.

Methods
<table>
<thead>
<tr>
<th>DiscretizationBase</th>
<th>estimate, visualize</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiscretizationInterface</td>
<td></td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, _setattr</td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes
**estimate** (*U, mu=None*)

Estimate the discretization error for a given solution.

**Parameters**
- **U**  The solution obtained by solve.
- **mu**  Parameter for which *U* has been obtained.

**Returns**

The estimated error.

**visualize** (*U, **kwargs*)

Visualize a solution VectorArray *U*.

**Parameters**
- **U**  The VectorArray from solution_space that shall be visualized.

---

**class** `pymor.discretizations.basic.InstationaryDiscretization` (*T, initial_data, operator, rhs=None, mass=None, time_stepper=None, num_values=None, products=None, operators=None, functionals=None, vector_operators=None, parameter_space=None, estimator=None, visualizer=None, cache_region='disk', name=None)

**Bases:** `pymor.discretizations.basic.DiscretizationBase`

Generic class for discretizations of instationary problems.

This class describes instationary problems given by the equations:

\[
M \cdot \frac{\partial}{\partial t} u(t, \mu) + L(u(\mu), t, \mu) = F(t, \mu) \\
u(0, \mu) = u_0(\mu)
\]

for *t* in [0, *T*], where *L* is a (possibly non-linear) time-dependent *Operator*, *F* is a time-dependent linear *Functional*, and *u_0* the initial data. The mass *Operator* *M* is assumed to be linear, time-independent and Parameter-independent.
Parameters

T  The end-time T.

initial_data  The initial data u_0. Either a VectorArray of length 1 or (for the Parameter-dependent case) a vector-like Operator (i.e. a linear Operator with source.dim == 1) which applied to NumpyVectorArray(np.array([1])) will yield the initial data for a given Parameter.

operator  The Operator L.

rhs  The Functional F.

mass  The mass Operator M. If None the identity is assumed.

time_stepper  T time-stepper to be used by solve. Has to satisfy the TimeStepperInterface.

num_values  The number of returned vectors of the solution trajectory. If None, each intermediate vector that is calculated is returned.

products  A dict of product Operators defined on the discrete space the problem is posed on. For each product a corresponding norm is added as a method of the discretization.

operators  A dict of Operators associated with the discretization.

functionals  A dict of (output) Functionals associated with the discretization.

vector_operators  A dict of vector-like Operators associated with the discretization.

parameter_space  The ParameterSpace for which the discrete problem is posed.

estimator  An error estimator for the problem. This can be any object with an estimate(U, mu, discretization) method. If estimator is not None an estimate(U, mu) method is added to the discretization which will call estimator.estimate(U, mu, self).

visualizer  A visualizer for the problem. This can be any object with a visualize(U, discretization, ...) method. If visualizer is not None a visualize(U, *args, **kwargs) method is added to the discretization, which forwards its arguments to the visualizer’s visualize method.

cache_region  None or name of the cache region to use. See pymor.core.cache.

name  Name of the discretization.
The end-time.

**initial_data**

The initial data $u_0$ given by a vector-like Operator. The same as `vector_operators[‘initial_data’]`.

**operator**

The Operator $L$. The same as `operators[‘operator’]`.

**rhs**

The Functional $F$. The same as `functionals[‘rhs’]`.

**mass**

The mass operator $M$. The same as `operators[‘mass’]`.

**time_stepper**

The provided time-stepper.

**with_(**kwargs)**

Returns a copy with changed attributes.

The default implementation is to call `_with_via_init(**kwargs)`.

**Parameters**

**kwargs**  Names of attributes to change with their new values. Each attribute name has to be contained in `with_arguments`.

**Returns**

Copy of `self` with changed attributes.

class pymor.discretizations.basic.StationaryDiscretization

Bases: pymor.discretizations.basic.DiscretizationBase

Generic class for discretizations of stationary problems.

This class describes discrete problems given by the equation:
with a linear functional $F$ and a (possibly non-linear) operator $L$.

**Parameters**

**operator** The $\text{Operator} L$.

**rhs** The $\text{Functional} F$.

**products** A dict of inner product $\text{Operators}$ defined on the discrete space the problem is posed on. For each product a corresponding norm is added as a method of the discretization.

**operators** A dict of $\text{ Operators}$ associated with the discretization.

**functionals** A dict of (output) $\text{ Functionals}$ associated with the discretization.

**vector_operators** A dict of vector-like $\text{Operators}$ associated with the discretization.

**parameter_space** The $\text{ParameterSpace}$ for which the discrete problem is posed.

**estimator** An error estimator for the problem. This can be any object with an $\text{ estimate}(U, \mu, \text{discretization})$ method. If $\text{estimator}$ is not $\text{ None}$ an $\text{ estimate}(U, \mu)$ method is added to the discretization which will call $\text{estimator.estimate}(U, \mu, \text{self})$.

**visualizer** A visualizer for the problem. This can be any object with a $\text{ visualize}(U, \text{discretization}, ...) $ method. If $\text{visualizer}$ is not $\text{ None}$ a $\text{ visualize}(U, *\text{args}, **\text{kwargs})$ method is added to the discretization, which forwards its arguments to the visualizer’s $\text{ visualize}$ method.

**cache_region** $\text{ None}$ or name of the cache region to use. See $\text{ pymor.core.cache}$.

**name** Name of the discretization.

**Methods**

<table>
<thead>
<tr>
<th>StationaryDiscretization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discretization</td>
</tr>
<tr>
<td>CacheableInterface</td>
</tr>
<tr>
<td>ImmutableInterface</td>
</tr>
<tr>
<td>BasicInterface</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parametric</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

| Attributes |

<table>
<thead>
<tr>
<th>StationaryDiscretization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discretization</td>
</tr>
<tr>
<td>CacheableInterface</td>
</tr>
<tr>
<td>ImmutableInterface</td>
</tr>
<tr>
<td>BasicInterface</td>
</tr>
<tr>
<td>Parametric</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operator, rhs, sid_ignore</th>
</tr>
</thead>
<tbody>
<tr>
<td>functionals, linear, operators, products, solution_space, vector_operators</td>
</tr>
<tr>
<td>cache_region</td>
</tr>
<tr>
<td>calculate_sid, sid, sid_failure</td>
</tr>
<tr>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>
**operator**
The `Operator` L. The same as `operators['operator']`.

**rhs**
The `Functional` F. The same as `functionals['rhs']`.

`with_(**kwargs)`
Returns a copy with changed attributes.
The default implementation is to call `with_via_init(**kwargs)`.

**Parameters**

**kwargs** Names of attributes to change with their new values. Each attribute name has to be contained in `with_arguments`.

**Returns**
Copy of `self` with changed attributes.

---

**interfaces module**

**class** `pymor.discretizations.interfaces.DiscretizationInterface`


Describes a discretization.

A discretization is an object describing a discrete problem via its type and the `Operators` it contains and which can solved for a given `Parameter` resulting in a solution `VectorArray`.

**Methods**

- `DiscretizationInterface.solve`, `visualize`
- `CacheableInterface.disable_caching`, `enable_caching`
- `ImmutableInterface.lock`, `unlock`
- `BasicInterface.add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`
- `Parametric.build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

**Attributes**

- `DiscretizationInterface.functionals`, `linear`, `operators`, `products`, `solution_space`, `vector_operators`
- `CacheableInterface.cache_region`
- `ImmutableInterface.calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `BasicInterface.locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`
- `Parametric.parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

**solution_space**
`VectorSpace` of the `VectorArrays` returned by `solve`.

**linear**
`True` if the discretization describes a linear Problem.
operators

Dictionary of all Operators contained in the discretization. (Compare the implementation of pymor.reductors.basic.reduce_generic_rb.)

functionals

Same as operators but for Functionals.

vector_operators

Same as operators but for Operators representing vectors, i.e. linear Operators with source.dim == 1.

products

Same as Operators but for inner product operators associated with the discretization.

estimate \((U, \mu=None)\)

Estimate the discretization error for a given solution.

Parameters

- **U** The solution obtained by solve.
- **\mu** Parameter for which \(U\) has been obtained.

Returns

The estimated error.

solve \((\mu=None, **kwargs)\)

Solve for the Parameter \(\mu\).

The result will be cached by default.

Parameters

- **\mu** Parameter for which to solve.

Returns

The solution given as a VectorArray.

visualize \((U, **kwargs)\)

Visualize a solution VectorArray \(U\).

Parameters

- **U** The VectorArray from solution_space that shall be visualized.

pymor.discretizers package

Submodules

advection module
Discretizes an `InstationaryAdvectionProblem` using the finite volume method. Simple explicit Euler time-stepping is used for time-discretization.

**Parameters**

- **analytical_problem** The `InstationaryAdvectionProblem` to discretize.
- **diameter** If not `None`, `diameter` is passed to the `domain_discretizer`.
- **nt** The number of time-steps.
- **num_flux** The numerical flux to use in the finite volume formulation. Allowed values are `'lax_friedrichs'`, `'engquist_osher'`, `'simplified_engquist_osher'`. (See `pymor.operators.fv`.)
- **lxf_lambda** The stabilization parameter for the Lax-Friedrichs numerical flux. (Ignored, if different flux is chosen.)
- **eo_gausspoints** Number of Gauss points for the Engquist-Osher numerical flux. (Ignored, if different flux is chosen.)
- **eo_intervals** Number of sub-intervals to use for integration when using Engquist-Osher numerical flux. (Ignored, if different flux is chosen.)
- **num_values** The number of returned vectors of the solution trajectory. If `None`, each intermediate vector that is calculated is returned.
- **domain_discretizer** Discretizer to be used for discretizing the analytical domain. This has to be a function `domain_discretizer(domain_description, diameter, ...)`. If further arguments should be passed to the discretizer, use `functools.partial`. If `None`, `discretize_domain_default` is used.
- **grid** Instead of using a domain discretizer, the `Grid` can also be passed directly using this parameter.
- **boundary_info** A `BoundaryInfo` specifying the boundary types of the grid boundary entities. Must be provided if `grid` is specified.

**Returns**

- **discretization** The `Discretization` that has been generated.
- **data** Dictionary with the following entries:
  - **grid** The generated `Grid`.
  - **boundary_info** The generated `BoundaryInfo`.
elliptic module

`pymor.discretizers.elliptic.discretize_elliptic_cg`  
`discretize_elliptic_cg`  
`discretize_elliptic_cg`  

Discretizes an `EllipticProblem` using finite elements.

**Parameters**

- `analytical_problem` The `EllipticProblem` to discretize.
- `diameter` If not `None`, `diameter` is passed to the `domain_discretizer`.
- `domain_discretizer` Discretizer to be used for discretizing the analytical domain. This has to be a function  
  `domain_discretizer(domain_description, diameter, ...)`. If further arguments should be passed to the discretizer, use `functools.partial`. If `None`, `discretize_domain_default` is used.
- `grid` Instead of using a domain discretizer, the `Grid` can also be passed directly using this parameter.
- `boundary_info` A `BoundaryInfo` specifying the boundary types of the grid boundary entities. Must be provided if `grid` is specified.

**Returns**

- `discretization` The `Discretization` that has been generated.
- `data` Dictionary with the following entries:
  - `grid` The generated `Grid`.
  - `boundary_info` The generated `BoundaryInfo`.

`pymor.discretizers.elliptic.discretize_elliptic_fv`  
`discretize_elliptic_fv`  
`discretize_elliptic_fv`  

Discretizes an `EllipticProblem` using the finite volume method.

**Parameters**

- `analytical_problem` The `EllipticProblem` to discretize.
- `diameter` If not `None`, `diameter` is passed to the `domain_discretizer`.
- `domain_discretizer` Discretizer to be used for discretizing the analytical domain. This has to be a function  
  `domain_discretizer(domain_description, diameter, ...)`. If further arguments should be passed to the discretizer, use `functools.partial`. If `None`, `discretize_domain_default` is used.
- `grid` Instead of using a domain discretizer, the `Grid` can also be passed directly using this parameter.
- `boundary_info` A `BoundaryInfo` specifying the boundary types of the grid boundary entities. Must be provided if `grid` is specified.

**Returns**

- `discretization` The `Discretization` that has been generated.
**pymor.domaindescriptions package**

**Submodules**

**basic module**

```python
class pymor.domaindescriptions.basic.CircleDomain(domain=(0, 1))
Bases: pymor.domaindescriptions.interfaces.DomainDescriptionInterface

Describes a domain with the topology of a circle, i.e. a line with identified end points.

Parameters

domain List [x_l, x_r] providing the left and right endpoint.
```

**Attributes**

```python
circle  domain, width
DomainDescriptionInterface boundary_types, has_dirichlet, has_neumann,
has_only_dirichlet, has_only_dirichlet_neumann,
has_only_neumann
ImmutableInterface calculate_sid, sid, sid_failure, sid_ignore
BasicInterface locked, logger, logging_disabled, name, uid, with_arguments
```

```python
domain
__repr__(x) <==> repr(x)
```

```python
class pymor.domaindescriptions.basic.CylindricalDomain(domain=((0, 0), [1, 1]),
top=BoundaryType('dirichlet'),
bottom=BoundaryType('dirichlet'))
Bases: pymor.domaindescriptions.interfaces.DomainDescriptionInterface

Describes a cylindrical domain.

BoundaryTypes can be associated edgewise.

Parameters

domain List of two points defining the lower-left and upper-right corner of the domain. The left and right edge are identified.
```
The BoundaryType of the top edge.

The BoundaryType of the bottom edge.

**Methods**

<table>
<thead>
<tr>
<th>CylindricalDomain</th>
<th><strong>repr</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>lock, unlock</td>
</tr>
</tbody>
</table>

| BasicInterface | add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with__, __setattr__ |

**Attributes**

<table>
<thead>
<tr>
<th>CylindricalDomain</th>
<th>bottom, diameter, domain, height, lower_left, top, upper_right, volume, width</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainDescription</td>
<td>boundary_types, has_dirichlet, has_neumann, has_only_dirichlet, has_only_dirichletneumann, has_only_neumann</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

```python
class pymor.domaindescriptions.basic.LineDomain(domain=(0, 1), left=BoundaryType('dirichlet'), right=BoundaryType('dirichlet')):
    Bases: pymor.domaindescriptions.interfaces.DomainDescriptionInterface

    Describes an interval domain.

    BoundaryTypes can be associated edgewise.

    Parameters
    domain List [x_l, x_r] providing the left and right endpoint.
    left The BoundaryType of the left endpoint.
    right The BoundaryType of the right endpoint.

    Methods
    __repr__ | | __repr__ |
    ImmutableInterface | lock, unlock |
    BasicInterface | add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with__, __setattr__ |
```

**Attributes**
class pymor.domaindescriptions.basic.RectDomain(domin=[0, 0], [1, 1]),
left=BoundaryType('dirichlet'),
right=BoundaryType('dirichlet'),
top=BoundaryType('dirichlet'),
bottom=BoundaryType('dirichlet'))

Bases: pymor.domaindescriptions.interfaces.DomainDescriptionInterface

Describes a rectangular domain.

BoundaryTypes can be associated edgewise.

Parameters

- **domain** List of two points defining the lower-left and upper-right corner of the domain.
- **left** The BoundaryType of the left edge.
- **right** The BoundaryType of the right edge.
- **top** The BoundaryType of the top edge.
- **bottom** The BoundaryType of the bottom edge.

Methods

- **__repr__()** <==> repr(x)

Attributes

- **bottom**, **diameter**, **domain**, **height**, **left**, **lower_left**, **right**, **top**, **upper_right**, **volume**, **width**

- **has_dirichlet**, **has_neumann**, **has_only_dirichlet**, **has_only_dirichlet_neumann**, **has_only_neumann**

- **calculate_sid**, **sid**, **sid_failure**, **sid_ignore**

- **locked**, **logger**, **logging_disabled**, **name**, **uid**, **with_arguments**
right
top
bottom
__repr__() <==> repr(x)

class pymor.domaindescriptions.basic.TorusDomain(domain=[0, 0], [1, 1])
Bases: pymor.domaindescriptions.interfaces.DomainDescriptionInterface

Describes a domain with the topology of a torus.

Parameters
domain List of two points defining the lower-left and upper-right corner of the domain. The left and right edge
are identified, as well as the bottom and top edge

Methods
TorusDomain__repr__
ImmutableInterface__lock
BasicInterfaceadd_attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, with_,
__setattr__

Attributes
TorusDomain diameter, domain, height, lower_left, upper_right, volume, width
DomainDescriptionInterfaceboundary_types, has_dirichlet, has_neumann,
has_only_dirichlet, has_only_dirichletneumann,
has_only_neumann
ImmutableInterfacecalculate_sid, sid, sid_failure, sid_ignore
BasicInterface locked, logger, logging_disabled, name, uid, with_arguments
domain
__repr__() <==> repr(x)

boundarytypes module
class pymor.domaindescriptions.boundarytypes.BoundaryType(type_)
Bases: pymor.core.interfaces.ImmutableInterface

Represents a boundary type, i.e. Dirichlet, Neumann, etc.

By defining a global registry of possible boundary types, we prevent hard to track down errors due to typos.
Only boundary types that have been registered before using register_type can be instantiated.

The boundary types which are registered by default are ‘dirichlet’, ‘neumann’ and ‘robin’.

Parameters
type_ Name of the boundary type as a string.
Methods

<table>
<thead>
<tr>
<th>BoundaryType</th>
<th>register_type, <strong>eq</strong>, <strong>hash</strong>, <strong>ne</strong>, <strong>repr</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>lock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging,</td>
</tr>
<tr>
<td></td>
<td>has_interface_name, implementor_names, implementors, with_,</td>
</tr>
<tr>
<td></td>
<td><strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>BoundaryType</th>
<th>types</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

types

Set of the names of registered boundary types.

__eq__(other)

`x.__eq__(y) <==> x==y`

__hash__()

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

__ne__(other)

`x.__ne__(y) <==> x!=y`

__repr__()

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

__str__()

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

classmethod register_type(name)

Register a new BoundaryType with name name.

interfaces module

class pymor.domaindescriptions.interfaces.DomainDescriptionInterface

Bases: pymor.core.interfaces.ImmutableInterface

Describes a geometric domain along with its boundary.

Methods

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>lock, unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging,</td>
</tr>
<tr>
<td></td>
<td>has_interface_name, implementor_names, implementors, with_,</td>
</tr>
<tr>
<td></td>
<td><strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>DomainDescriptionInterface</th>
<th>boundary_types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>has_dirichlet, has_neumann,</td>
</tr>
<tr>
<td></td>
<td>has_only_dirichlet, has_only_dirichletneumann,</td>
</tr>
<tr>
<td></td>
<td>has_only_neumann</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

boundary_types

Set of BoundaryTypes the domain has.
pyMOR, Release 0.3.2

pymor.domaindiscretizers package

Submodules

default module
pymor.domaindiscretizers.default.discretize_domain_default (domain_description, diameter=0.01, grid_type=None)

Discretize a DomainDescription using a sensible default implementation.

This method can discretize the following DomainDescriptions:

<table>
<thead>
<tr>
<th>DomainDescription</th>
<th>grid_type</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>RectDomain</td>
<td>TriaGrid</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>RectGrid</td>
<td></td>
</tr>
<tr>
<td>CylindricalDomain</td>
<td>TriaGrid</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>RectGrid</td>
<td></td>
</tr>
<tr>
<td>TorusDomain</td>
<td>TriaGrid</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>RectGrid</td>
<td></td>
</tr>
<tr>
<td>LineDomain</td>
<td>OnedGrid</td>
<td>X</td>
</tr>
<tr>
<td>CircleDomain</td>
<td>OnedGrid</td>
<td>X</td>
</tr>
</tbody>
</table>

Parameters

- **domain_description** A DomainDescription of the domain to discretize.
- **diameter** Maximal diameter of the codim-0 entities of the generated Grid.
- **grid_type** The class of the Grid which is to be constructed. If None, a default choice is made according to the table above.

Returns

- **grid** The generated Grid.
- **boundary_info** The generated BoundaryInfo.

pymor.functions package

Submodules

basic module
class pymor.functions.basic.ConstantFunction (value=array(1.0), dim_domain=1, name=None)

Bases: pymor.functions.basic.FunctionBase

A constant Function

f: R^d -> R^shape(c), f(x) = c

Parameters

- **value** The constant c.
**dim_domain**  The dimension $d$.

**name**  The name of the function.

---

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ConstantFunction</code></td>
<td><code>evaluate</code>, <code>__repr__</code>, <code>__str__</code></td>
</tr>
<tr>
<td><code>FunctionBase</code></td>
<td><code>__add__</code>, <code>__mul__</code>, <code>__neg__</code>, <code>__radd__</code>, <code>__rmul__</code>, <code>__sub__</code></td>
</tr>
<tr>
<td><code>FunctionInterface</code></td>
<td></td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>__setattr__</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td><code>add_attributes</code>, <code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>with_</code>, <code>__setattr__</code></td>
</tr>
<tr>
<td><code>Parametric</code></td>
<td><code>build_parameter_type</code>, <code>local_parameter</code>, <code>parse_parameter</code>, <code>strip_parameter</code></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Interface</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FunctionInterface</code></td>
<td><code>dim_domain</code>, <code>shape_range</code></td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>calculate_sid</code>, <code>sid</code>, <code>sid_failure</code>, <code>sid_ignore</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td><code>locked</code>, <code>logger</code>, <code>logging_disabled</code>, <code>name</code>, <code>uid</code>, <code>with_arguments</code></td>
</tr>
<tr>
<td><code>Parametric</code></td>
<td><code>parameter_local_type</code>, <code>parameter_space</code>, <code>parameter_type</code>, <code>parametric</code></td>
</tr>
</tbody>
</table>

---

**__repr__** $(\text{x}) \iff \text{repr}(x)$

**__str__** $(\text{x}) \iff \text{str}(x)$

**evaluate** $(x, \mu=\text{None})$

Evaluate the function for given argument and *Parameter*.

---

**class pymor.functions.basic.ExpressionFunction** $(\text{expression}, \text{dim_domain}=\text{1}, \text{shape_range}=\text{()}, \text{parameter_type}=\text{None}, \text{name}=\text{None})$

**Bases:** `pymor.functions.basic.GenericFunction`  

Turns a Python expression given as a string into a *Function*.

Some *NumPy* arithmetic functions like ‘sin’, ‘log’, ‘min’ are supported. For a full list see the *functions* class attribute.

**Warning:** *eval* is used to evaluate the given expression. As a consequence, using this class with expression strings from untrusted sources will cause mayhem and destruction!

---

**Parameters**

- **expression**  A Python expression of one variable $x$ and a parameter $\mu$ given as a string.
- **dim_domain**  The dimension of the domain.
- **shape_range**  The shape of the values returned by the expression.
- **parameter_type**  The *ParameterType* the expression accepts.
- **name**  The name of the function.

---

**Methods**

---

4.1. *pymor* package
ExpressionFunction.

GenericFunction: __reduce__, __repr__

FunctionBase: __add__, __mul__, __neg__, __radd__, __rmul__, __sub__

FunctionInterface:

ImmutableInterface: lock, unlock

BasicInterface: add_attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, with_,
__setattr__

Parametric: build_parameter_type, local_parameter, parse_parameter,
strip_parameter

__reduce__()
helper for pickle

__repr__() <=> repr(x)

class pymor.functions.basic.FunctionBase
Bases: pymor.functions.interfaces.FunctionInterface

Base class for Functions providing some common functionality.

Methods

FunctionBase: __add__, __mul__, __neg__, __radd__, __rmul__, __sub__

FunctionInterface: __call__

ImmutableInterface: lock, unlock

BasicInterface: add_attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, with_,
__setattr__

Parametric: build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes

FunctionInterface: dim_domain, shape_range

ImmutableInterface: calculate_sid, sid, sid_failure, sid_ignore

BasicInterface: locked, logger, logging_disabled, name, uid, with_arguments

Parametric: parameter_local_type, parameter_space, parameter_type,
parametric

__add__(other)
Returns a new LincombFunction representing the sum of two functions, or of one function and a constant.

__mul__(other)
Returns a new LincombFunction representing the product of a function by a scalar.
__neg__(self)
    Returns a new LincombFunction representing the function scaled by -1.

__radd__(other)
    Returns a new LincombFunction representing the sum of two functions, or of one function and a constant.

__rmul__(other)
    Returns a new LincombFunction representing the product of a function by a scalar.

__sub__(other)
    Returns a new LincombFunction representing the difference of two functions, or of one function and a constant.

class pymor.functions.basic.GenericFunction(mapping, dim_domain=1, shape_range=(), parameter_type=None, name=None)
Bases: pymor.functions.basic.FunctionBase
Wrapper making an arbitrary Python function between NumPy arrays a proper Function.
Note that a GenericFunction can only be dumps_function will be tried as a last resort. For this reason, it is usually preferable to use ExpressionFunction instead, which always can be serialized.

Parameters
mapping  The function to wrap. If parameter_type is None, the function is of the form mapping(x). If parameter_type is not None, the function has to have the signature mapping(x, mu). Moreover, the function is expected to be vectorized, i.e.:

    mapping(x).shape == x.shape[:-1] + shape_range.

dim_domain  The dimension of the domain.
shape_range  The shape of the values returned by the mapping.
parameter_type  The ParameterType the mapping accepts.
name  The name of the function.

Methods
GenericFunction evaluate, __str__
FunctionBase __add__, __mul__, __neg__, __radd__, __rmul__, __sub__
ImmutableInterface __lock, __unlock
BasicInterface add_attributes, disable_logging, enable_logging,
    has_interface_name, implementor_names, implementors, with_,
    __setattr__
Parametric build_parameter_type, local_parameter, parse_parameter,
    strip_parameter

Attributes
FunctionInterface dim_domain, shape_range
ImmutableInterface calculate_sid, sid, sid_failure, sid_ignore
BasicInterface locked, logger, logging_disabled, name, uid, with_arguments
Parametric parameter_local_type, parameter_space, parameter_type, parametric
class pymor.functions.basic.LincombFunction (functions, coefficients, name=None)
   Bases: pymor.functions.basic.FunctionBase

   A Function representing a linear combination of Functions.

   The linear coefficients can be provided as scalars or ParameterFunctionals. Alternatively, if no linear coefficients are given, the missing coefficients become part of the Parameter the functions expects.

   Parameters
   functions  List of Functions whose linear combination is formed.
   coefficients A list of linear coefficients. A linear coefficient can either be a fixed number or a ParameterFunctional.
   name       Name of the function.

   Methods
   LincombFunction.evaluate, evaluate_coefficients
   FunctionBase.__add__, __mul__, __neg__, __radd__, __rmul__, __sub__
   FunctionInterface.__call__
   ImmutableInterface.lock, unlock
   BasicInterface.add_attributes, disable_logging, enable_logging,
      has_interface_name, implementor_names, implementors, with_,
      __setattr__
   Parametric.build_parameter_type, local_parameter, parse_parameter,
      strip_parameter

   Attributes
   LincombFunction     coefficients, functions
   FunctionInterface   dim_domain, shape_range
   ImmutableInterface  calculate_sid, sid, sid_failure, sid_ignore
   BasicInterface      locked, logger, logging_disabled, name, uid, with_arguments
   Parametric          parameter_local_type, parameter_space, parameter_type, parametric

functions
coefficients
evaluate (x, mu=None)
   Evaluate the function for given argument and Parameter.

evaluate_coefficients (mu)
   Compute the linear coefficients for a given Parameter mu.
class pymor.functions.interfaces.FunctionInterface
Bases: pymor.core.interfaces.ImmutableInterface, pymor.parameters.base.Parametric

Interface for Parameter dependent analytical functions.

Every function is a map of the form
\[ f(\mu): \Omega \rightarrow \mathbb{R}^{\text{shape\_range}} \]

The returned values are NumPy arrays of arbitrary (but fixed) shape. Note that NumPy distinguished between one-dimensional arrays of length 1 (with shape (1,)) and zero-dimensional scalar arrays (with shape tuple()). In pyMOR, we usually expect scalar-valued functions to have shape_range == tuple().

While the function might raise an error if it is evaluated for an argument not in the domain \( \Omega \), the exact behavior is left undefined.

Functions are vectorized in the sense, that if \( x.n\text{dim} = k \), then
\[ f(x, \mu)[i_0, i_1, \ldots, i(k-2)] = f(x[i_0, i_1, \ldots, i(k-2)], \mu), \]
in particular \( f(x, \mu).\text{shape} == x.\text{shape}[\ldots] + \text{shape\_range} \).

Methods

<table>
<thead>
<tr>
<th>FunctionInterface</th>
<th>evaluate, <strong>call</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>lock, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>FunctionInterface</th>
<th>dim_domain, shape_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

dim_domain
The dimension \( d > 0 \).

shape_range
The shape of the function values.

__call__(...) \( \Leftrightarrow x(...) \)

evaluate (x, mu=None)
Evaluate the function for given argument and Parameter.

pymor.grids package

Submodules

boundaryinfos module
class pymor.grids.boundaryinfos.

AllDirichletBoundaryInfo(grid)

Bases: pymor.grids.interfaces.BoundaryInfoInterface

BoundaryInfo where BoundaryType('dirichlet') is attached to each boundary entity.

Methods

AllDirichletBoundaryInfo

BoundaryInfo Interface

check_boundary_types, dirichlet_boundaries, dirichlet_mask,

neumann_boundaries, neumann_mask, no_boundary_type_mask,

unique_boundary_type_mask

CacheableInterface

disable_caching, enable_caching

ImmutableInterface

lock, unlock

BasicInterface

add_attributes, disable_logging, enable_logging,

has_interface_name, implementor_names, implementors, with_,

_setattr_

Attributes

BoundaryInfo Interface

boundary_types, has_dirichlet, has_neumann,

has_only_dirichlet, has_only_dirichlet_neumann,

has_only_neumann

CacheableInterface

cache_region

ImmutableInterface

calculate_sid, sid, sid_failure, sid_ignore

BasicInterface

locked, logger, logging_disabled, name, uid, with_arguments

mask (boundary_type, codim)

retval[i] is True if the codim-codim entity of global index i is associated to the BoundaryType boundary_type.

class pymor.grids.boundaryinfos.

BoundaryInfoFromIndicators(grid, indicators, assert_unique_type=None, assert_some_type=None)

Bases: pymor.grids.interfaces.BoundaryInfoInterface

BoundaryInfo where the BoundaryTypes are determined by indicator functions.

Parameters

grid The grid to which the BoundaryInfo is associated.

indicators Dict where each key is a BoundaryType and the corresponding value is a boolean valued function defined on the analytical domain which indicates if a point belongs to a boundary of the given BoundaryType. (The indicator functions must be vectorized.)
Attributes

- `BoundaryInfo` attributes:
  - `boundary_types`, `has_dirichlet`, `has_neumann`,
  - `has_only_dirichlet`, `has_only_dirichlet_neumann`,
  - `has_only_neumann`

- `CacheableInterface` attribute: `cache_region`

- `ImmutableInterface` attributes:
  - `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`

- `BasicInterface` attributes:
  - `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`

**mask** `(boundary_type, codim)`

`retval[i]` is `True` if the `codim-codim` entity of global index `i` is associated to the `BoundaryType boundary_type`.

class `pymor.grids.boundaryinfos.EmptyBoundaryInfo(grid)`

Bases: `pymor.gridsinterfaces.BoundaryInfoInterface`

`BoundaryInfo` with no `BoundaryTypes` attached to any boundary.

Methods

- `EmptyBoundaryInfo.__init__`
- `BoundaryInfo._check_boundary_types`, `dirichlet_boundaries`, `dirichlet_mask`,
  - `neumann_boundaries`, `neumann_mask`, `no_boundary_type_mask`,
  - `unique_boundary_type_mask`

- `CacheableInterface.disable_caching`, `enable_caching`

- `ImmutableInterface.lock`, `unlock`

- `BasicInterface.add_attributes`, `disable_logging`, `enable_logging`,
  - `has_interface_name`, `implementor_names`, `implementors`, `with__`,
  - `__setattr__`

Attributes

- `BoundaryInfo` attributes:
  - `boundary_types`, `has_dirichlet`, `has_neumann`,
  - `has_only_dirichlet`, `has_only_dirichlet_neumann`,
  - `has_only_neumann`

- `CacheableInterface` attribute: `cache_region`

- `ImmutableInterface` attributes:
  - `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`

- `BasicInterface` attributes:
  - `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`

**mask** `(boundary_type, codim)`

`retval[i]` is `True` if the `codim-codim` entity of global index `i` is associated to the `BoundaryType boundary_type`.

class `pymor.grids.boundaryinfos.SubGridBoundaryInfo(subgrid, grid, grid_boundary_info, new_boundary_type=None)`

Bases: `pymor.gridsinterfaces.BoundaryInfoInterface`

Derives a `BoundaryInfo` for a `SubGrid`.

Parameters

- `subgrid` The `SubGrid` for which a `BoundaryInfo` is created.
- `grid` The parent `Grid`.
grid_boundary_info The BoundaryInfo of the parent Grid from which to derive the BoundaryInfo.

new_boundary_type The BoundaryType which is assigned to the new boundaries of subgrid. If None, no BoundaryType is assigned.

---

### Methods

<table>
<thead>
<tr>
<th>SubGridBoundaryInfo</th>
<th>BoundaryInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_boundary_types, dirichlet_boundaries, dirichlet_mask, neumann_boundaries, neumann_mask, no_boundary_type_mask, unique_boundary_type_mask</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CacheableInterface</th>
<th>ImmutableInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable_caching, enable_caching</td>
<td></td>
</tr>
<tr>
<td>lock, unlock</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BasicInterface</th>
<th>CacheableInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <em>setattr</em>_</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>CacheableInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundary_types, has_dirichlet, has_neumann, has_only_dirichlet, has_only_dirichlet_neumann, has_only_neumann</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>BasicInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
<td></td>
</tr>
<tr>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
<td></td>
</tr>
</tbody>
</table>

---

### constructions module

**pymor.grids.constructions.flatten_grid(grid)**

This method is used by our visualizers to render n-dimensional grids which cannot be embedded into R^n by duplicating vertices which would have to be mapped to multiple points at once. (Think of grids on rectangular domains with identified edges.)

**Parameters**

**grid** The Grid to flatten.

**Returns**

**subentities** The subentities(0, grid.dim) relation for the flattened grid.

**coordinates** The coordinates of the codim-grid.dim entities.

**entity_map** Maps the indices of the codim-grid.dim entities of the flattened grid to the indices of the corresponding entities in the original grid.

---

**defaultimpl module**
class pymor.grids.defaultimpl.AffineGridDefaultImplementations
    Bases: object

    Provides default implementations for AffineGrids.

class pymor.grids.defaultimpl.ConformalTopologicalGridDefaultImplementations
    Bases: object

    Provides default informations for ConformalTopologicalGrids.

class pymor.grids.defaultimpl_reference_element_default_Implementations
    Bases: object

    Provides default implementations for ReferenceElements.

interfaces module
class pymor.grids.interfaces.AffineGridInterface
    Bases: pymor.grids.defaultimpl.AffineGridDefaultImplementations,
          pymor.grids.interfaces.ConformalTopologicalGridInterface

    Topological grid with geometry where each codim-0 entity is affinely mapped to the same
    ReferenceElement.

    The grid is completely determined via the subentity relation given by subentities and the
    embeddings given by embeddings. In addition, only size and reference_element have to be
    implemented. Cached default implementations for all other methods are provided by
    AffineGridDefaultImplementations.

    Methods

    AffineGridInterface: centers, diameters, embeddings, integration_elements,
                        jacobian_inverse_transposed, quadrature_points,
                        reference_element, subentities, unit_outer_normals,
                        volumes, volumes_inverse

    ConformalTopologicalGridInterface: boundary_mask, neighbours, size, superentities,
                                        super_entity_indices

    CacheableInterface: disable_caching, enable_caching

    ImmutableInterface: lock, unlock

    BasicInterface: add_attributes, disable_logging, enable_logging,
                    has_interface_name, implementor_names, implementors,
                    with_, __setattr__

Attributes

AffineGridInterface: dim_outer
ConformalTopologicalGridInterface
CacheableInterface: cache_region
ImmutableInterface: calculate_sid, sid, sid_failure, sid_ignore
BasicInterface: locked, logger, logging_disabled, name, uid,
                with_arguments

dim_outer
    The dimension of the space into which the grid is embedded.
centers \( \text{(codim)} \)
\( \text{retval}[e] \) is the barycenter of the codim-codim entity with global index \( e \).

diameters \( \text{(codim)} \)
\( \text{retval}[e] \) is the diameter of the codim-codim entity with global index \( e \).

embeddings \( \text{(codim)} \)
Returns tuple \((A, B)\) where \( A[e] \) and \( B[e] \) are the linear part and the translation part of the map from the reference element of \( e \) to \( e \).

For \( \text{codim} > 0 \), we provide a default implementation by taking the embedding of the codim-1 parent entity \( e_0 \) of \( e \) with lowest global index and composing it with the subentity_embedding of \( e \) into \( e_0 \) determined by the reference element.

integration_elements \( \text{(codim)} \)
\( \text{retval}[e] \) is given as \( \sqrt{\text{det}(A^T \cdot A)} \), where \( A = \text{embeddings(codim)[0][e]} \).

jacobian_inverse_transposed \( \text{(codim)} \)
\( \text{retval}[e] \) is the transposed (pseudo-)inverse of the jacobian of \( \text{embeddings(codim)[e]} \).

quadrature_points \( \text{(codim, order=None, npoints=None, quadrature_type='default')} \)
\( \text{retval}[e] \) is an array of quadrature points in global coordinates for the codim-codim entity with global index \( e \).

The quadrature is of order \( \text{order} \) or has \( \text{npoints} \) integration points. To integrate a function \( f \) over \( e \) one has to form
\[
\text{np.dot}(f(\text{quadrature_points(codim, order)[e]}), \text{reference_element(codim).quadrature(order)[1]} * \text{integration_elements(codim)[e]}) \text{ # NOQA}
\]

reference_element \( \text{(codim)} \)
The \( \text{ReferenceElement} \) of the codim-codim entities.

subentities \( \text{(codim, subentity_codim)} \)
\( \text{retval}[e,s] \) is the global index of the \( s \)-th codim-subentity_codim subentity of the codim-codim entity with global index \( e \).

Only \( \text{subentities(codim, codim+1)} \) has to be implemented; a default implementation is provided which evaluates \( \text{subentities(codim, subentity_codim)} \) by computing the transitive closure of \( \text{subentities(codim, codim+1)} \).

unit_outer_normals \( \text{()} \)
\( \text{retval}[e,i] \) is the unit outer normal to the \( i \)-th codim-1 subentity of the codim-0 entity with global index \( e \).

volumes \( \text{(codim)} \)
\( \text{retval}[e] \) is the (dim-codim)-dimensional volume of the codim-codim entity with global index \( e \).

volumes_inverse \( \text{(codim)} \)
\( \text{retval}[e] = \frac{1}{\text{volumes(codim)[e]}} \).

class pymor.grids.interfaces.AffineGridWithOrthogonalCentersInterface
Bases: pymor.grids.interfaces.AffineGridInterface

\text{AffineGrid} \text{ with an additional orthogonal_centers method.}
## orthogonal_centers()

`retval[e]` is a point inside the codim-0 entity with global index `e` such that the line segment from `retval[e]` to `retval[e2]` is always orthogonal to the codim-1 entity shared by the codim-0 entities with global index `e` and `e2`.

(This is mainly useful for gradient approximation in finite volume schemes.)

```python
class pymor.grids.interfaces.BoundaryInfoInterface
    Bases: pymor.core.cache.CacheableInterface

Provides `BoundaryTypes` for the boundaries of a given `ConformalTopologicalGrid`.

For every `BoundaryType` and codimension a mask is provided, marking grid entities of the respective type and codimension by their global index.
```

### Methods

```python
BoundaryInfoInterface
    check_boundary_types, dirichlet_boundaries, dirichlet_mask, mask, neumann_boundaries, neumann_mask, no_boundary_type_mask, unique_boundary_type_mask,
CacheableInterface
    disable_caching, enable_caching
ImmutableInterface
    lock, unlock
BasicInterface
    add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__
```
boundary_types
set of all BoundaryTypes the grid has.

mask (boundary_type, codim)
retval[i] is True if the codim-codim entity of global index i is associated to the BoundaryType boundary_type.

no_boundary_type_mask (codim)
retval[i] is True if the codim-codim entity of global index i is associated to no BoundaryType.

unique_boundary_type_mask (codim)
retval[i] is True if the codim-codim entity of global index i is associated to one and only one BoundaryType.

class pymor.grids.interfaces.ConformalTopologicalGridInterface
Bases: pymor.grids.defaultimpl.ConformalTopologicalGridDefaultImplementations,
pymor.core.cache.CacheableInterface

A topological grid without hanging nodes.

The grid is completely determined via the subentity relation given by subentities. In addition, only size has to be implemented, cached default implementations for all other methods are provided by ConformalTopologicalGridDefaultImplementations.

Methods

<table>
<thead>
<tr>
<th>ConformalTopologicalGridInterface</th>
<th>boundaries, boundary_mask, neighbours, size, subentities, superentities, superentity_indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheableInterface</td>
<td>disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>lock, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>ConformalTopologicalGridInterface</th>
<th>cache_region</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td></td>
</tr>
</tbody>
</table>

dim
The dimension of the grid.

boundaries (codim)
Returns the global indices of all codim-codim boundary entities.

By definition, a codim-1 entity is a boundary entity if it has only one codim-0 superentity. For codim != 1, a codim-codim entity is a boundary entity if it has a codim-1 sub/super-entity.

boundary_mask (codim)
retval[e] is true iff the codim-codim entity with global index e is a boundary entity.

By definition, a codim-1 entity is a boundary entity if it has only one codim-0 superentity. For codim != 1, a codim-codim entity is a boundary entity if it has a codim-1 sub/super-entity.
**neighbours** *(codim, neighbour_codim, intersection_codim=None)*

`retval[e,n]` is the global index of the $n$-th codim-neighbour_codim entity of the codim-codim entity $e$ that shares with $e$ a subentity of codimension intersection_codim.

If `intersection_codim == None`, it is set to `codim + 1` if `codim == neighbour_codim` and to `min(codim, neighbour_codim)` otherwise.

The default implementation is to compute the result from `subentities(codim, intersection_codim)` and `superentities(intersection_codim, neighbour_codim)`.

**size** *(codim)*

The number of entities of codimension `codim`.

**subentities** *(codim, subentity_codim)*

`retval[e,s]` is the global index of the $s$-th codim-subentity_codim subentity of the codim-codim entity with global index $e$.

Only `subentities(codim, codim+1)` has to be implemented; a default implementation is provided which evaluates `subentities(codim, subentity_codim)` by computing the transitive closure of `subentities(codim, codim+1)`.

**superentities** *(codim, superentity_codim)*

`retval[e,s]` is the global index of the $s$-th codim-superentity_codim superentity of the codim-codim entity with global index $e$.

`retval[e]` is sorted by global index.

The default implementation is to compute the result from `subentities(superentity_codim, codim)`.

**superentity_indices** *(codim, superentity_codim)*

`retval[e,s]` is the local index of the codim-codim entity $e$ in the codim-superentity_codim superentity `superentities(codim, superentity_codim)[e,s]`.

---

**class** `pymor.grids.interfaces.ReferenceElementInterface`

**Bases:** `pymor.grids.defaultimpl.ReferenceElementDefaultImplementations`, `pymor.core.cache.CacheableInterface`

Defines a reference element.

All reference elements have the property that all subentities of a given codimension are of the same type. I.e. a three-dimensional reference element cannot have triangles and rectangles as faces at the same time.

**Methods**

- `center`, `mapped_diameter`, `quadrature`, `quadrature_info`, `quadrature_types`, `size`, `sub_reference_element`, `subentities`, `subentity_embedding`, `unit_outer_normals`, `__call__`
- `CacheableInterface`: `disable_caching`, `enable_caching`
- `ImmutableInterface`: `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `BasicInterface`: `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`

**Attributes**

- `ReferenceElementInterface`: `dim`, `volume`
- `CacheableInterface`: `cache_region`
- `ImmutableInterface`: `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `BasicInterface`: `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`
dim
   The dimension of the reference element

volume
   The volume of the reference element

__call__(...) <==> x(...)

center()
   Coordinates of the barycenter.

mapped_diameter(A)
   The diameter of the reference element after transforming it with the matrix A (vectorized).

quadrature(order=None, npoints=None, quadrature_type='default')
   Returns tuple (P, W) where P is an array of quadrature points with corresponding weights W.
   The quadrature is of order order or has npoints integration points.

quadrature_info()
   Returns a tuple of dicts (O, N) where O[quadrature_type] is a list of orders which are implemented for
   quadrature_type and N[quadrature_type] is a list of the corresponding numbers of integration points.

size(codim)
   Number of subentities of codimension codim.

sub_reference_element(codim)
   Returns the reference relement of the codim-codim subentities.

subentities(codim, subentity_codim)
   subentities(c,sc)[i,j] is, with respect to the indexing inside the reference element, the index of the j-th
   codim-subentity_codim subentity of the i-th codim-codim subentity of the reference element.

subentity_embedding(subentity_codim)
   Returns a tuple (A, B) which defines the embedding of the codim-subentity_codim subentities into the
   reference element.
   For subentity_codim > 1', the embedding is by default given recursively via 'suben-
   tity_embedding(subentity_codim - 1) and sub_reference_element(subentity_codim - 1).suben-
   tity_embedding(1) choosing always the superentity with smallest index.

unit_outer_normals()
   retval[e] is the unit outer-normal vector to the codim-1 subentity with index e.

oned module

class pymor.grids.oned.OnedGrid(domain=(0, 1), num_intervals=4, identify_left_right=False)
   Bases: pymor.grids.interfaces.AffineGridWithOrthogonalCentersInterface

   One-dimensional Grid on an interval.

Parameters

domain  Tuple (left, right) containing the left and right boundary of the domain.

num_intervals  The number of codim-0 entities.

Methods
OnedGrid  embeddings, orthogonal_centers, size, subentities, visualize, __reduce__, __str__
AffineGridInterface  centers, diameters, integration_elements, jacobian_inverse_transposed, quadrature_points, unit_outer_normals, volumes, volumes_inverse
ConformalTopologicalGridInterface  boundaries, neighbours, superentities, superentity_indices
CacheableInterface  disable_caching, enable_caching
ImmutableInterface  lock, unlock
BasicInterface  add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with, __setattr__

Attributes

<table>
<thead>
<tr>
<th>OnedGrid</th>
<th>dim, dim_outer, reference_element</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheableInterface</td>
<td>cache_region</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

__reduce__()  
helper for pickle

__str__() <==> str(x)

embeddings (codim)
Returns tuple \((A, B)\) where \(A[e]\) and \(B[e]\) are the linear part and the translation part of the map from the reference element of \(e\) to \(e\).

For \(codim > 0\), we provide a default implementation by taking the embedding of the codim-1 parent entity \(e_0\) of \(e\) with lowest global index and composing it with the subentity_embedding of \(e\) into \(e_0\) determined by the reference element.

orthogonal_centers ()
\(retval[e]\) is a point inside the codim-0 entity with global index \(e\) such that the line segment from \(retval[e]\) to \(retval[e_2]\) is always orthogonal to the codim-1 entity shared by the codim-0 entities with global index \(e\) and \(e_2\).

(This is mainly useful for gradient approximation in finite volume schemes.)

size (codim=0)
The number of entities of codimension \(codim\).

subentities (codim, subentity_codim)
\(retval[e,s]\) is the global index of the \(s\)-th codim-subentity_codim subentity of the codim-codim entity with global index \(e\).

Only subentities(codim, codim+1) has to be implemented; a default implementation is provided which evaluates subentities(codim, subentity_codim) by computing the transitive closure of subentities(codim, codim+1).

visualize (U, codim=2, **kwargs)
Visualize scalar data associated to the grid as a plot.

Parameters

\(U\) VectorArray of the data to visualize. If \(len(U) > 1\), the data is visualized as a time series of plots. Alternatively, a tuple of VectorArrays can be provided, in which case several plots are made into
the same axes. The lengths of all arrays have to agree.

**codim** The codimension of the entities the data in $U$ is attached to (either 0 or 1).

**kwargs** See `visualize_matplotlib_1d`

---

**rect module**

**class** `pymor.grids.rect.RectGrid(num_intervals=(2, 2), domain=[0, 0], [1, 1], identify_left_right=False, identify_bottom_top=False)`

Bases: `pymor.grids.interfaces.AffineGridWithOrthogonalCentersInterface`

Basic implementation of a rectangular *Grid* on a rectangular domain.

The global face, edge and vertex indices are given as follows

```
x1
^  |
| 6--10---7--11---8
|  |  |  |
3 2 4 3 5
|  |  |  |
3---8---4---9---5
|  |  |  |
0 0 1 1 2
|  |  |  |
0---6---1---7---2 --> x0
```

**Parameters**

- **num_intervals** Tuple $(n0, n1)$ determining a grid with $n0 \times n1$ codim-0 entities.
- **domain** Tuple $(ll, ur)$ where $ll$ defines the lower left and $ur$ the upper right corner of the domain.

**Methods**

<table>
<thead>
<tr>
<th><code>RectGrid</code></th>
<th>embeddings, global_to_structured, orthogonal_centers, size, structured_to_global, subentities, vertex_coordinates, visualize, <strong>reduce</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AffineGridInterface</code></td>
<td>centers, diameters, integration_elements, jacobian_inverse_transposed, quadrature_points, unit_outer_normals, volumes, volumes_inverse</td>
</tr>
<tr>
<td><code>ConformalTopologicalGridInterface</code></td>
<td>facets, faces, boundary_mask, neighbours, superentities, superentity_indices</td>
</tr>
<tr>
<td><code>CacheableInterface</code></td>
<td>disable_caching, enable_caching</td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td>lock, unlock</td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with, <strong>setattr</strong></td>
</tr>
</tbody>
</table>
RectGrid  |  dim, dim_outer, reference_element
---|---
CacheableInterface  |  cache_region
ImmutableInterface  |  calculate_sid, sid, sid_failure, sid_ignore
BasicInterface  |  locked, logger, logging_disabled, name, uid, with_arguments

___reduce___()  
helper for pickle

___str___() <==> str(x)

embeddings (codim=0)  
Returns tuple (A, B) where A[e] and B[e] are the linear part and the translation part of the map from the reference element of e to e.

For codim > 0, we provide a default implementation by taking the embedding of the codim-1 parent entity e_0 of e with lowest global index and composing it with the subentity_embedding of e into e_0 determined by the reference element.

global_to_structured (codim)  
Returns an array which maps global codim-codim indices to structured indices.

I.e. if GTS = global_to_structured(codim) and STG = structured_to_global(codim), then STG[GTS[:, 0], GTS[:, 1]] == numpy.arange(size(codim)).

orthogonal_centers ()  
retval[e] is a point inside the codim-0 entity with global index e such that the line segment from retval[e] to retval[e2] is always orthogonal to the codim-1 entity shared by the codim-0 entities with global index e and e2.

(This is mainly useful for gradient approximation in finite volume schemes.)

size (codim=0)  
The number of entities of codimension codim.

structured_to_global (codim)  
Returns an array which maps structured indices to global codim-codim indices.

In other words structured_to_global(codim)[i, j] is the global index of the i-th in x0-direction and j-th in x1-direction codim-codim entity of the grid.

subentities (codim, subentity_codim)  
retval[e,s] is the global index of the s-th codim-subentity_codim subentity of the codim-codim entity with global index e.

Only subentities(codim, codim+1) has to be implemented; a default implementation is provided which evaluates subentities(codim, subentity_codim) by computing the transitive closure of subentities(codim, codim+1).

vertex_coordinates (dim)  
Returns an array of the x_dim coordinates of the grid vertices.

I.e.

centers(2)[structured_to_global(2)[i, j]] == np.array([vertex_coordinates(0)[i], vertex_coordinates(1)[j]]

visualize (U, codim=2, **kwargs)  
Visualize scalar data associated to the grid as a patch plot.

Parameters
**U** VectorArray of the data to visualize. If \( \text{len}(U) > 1 \), the data is visualized as a time series of plots.
Alternatively, a tuple of VectorArrays can be provided, in which case a subplot is created for each entry of the tuple. The lengths of all arrays have to agree.

codim  The codimension of the entities the data in \( U \) is attached to (either 0 or 2).
kwargs  See visualize_patch

---

**referenceelements module**

**class** pymor.grids.referenceelements.Line

**Bases:** pymor.grids.interfaces.ReferenceElementInterface

**Methods**

| Line | center, mapped_diameter, quadrature, quadrature_info, size, sub_reference_element, subentities, subentity_embedding, unit_outer_normals |
| ReferenceElement | quadrature_types, __call__ |
| CacheableInterface | disable_caching, enable_caching |
| ImmutableInterface | lock, unlock |
| BasicInterface | add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__ |

**Attributes**

| Line | dim, volume |
| CacheableInterface | cache_region |
| ImmutableInterface | calculate_sid, sid, sid_failure, sid_ignore |
| BasicInterface | locked, logger, logging_disabled, name, uid, with_arguments |

**center ()**

Coordinates of the barycenter.

**mapped_diameter**(A)

The diameter of the reference element after transforming it with the matrix \( A \) (vectorized).

**quadrature**(order=\( \text{None} \), npoints=\( \text{None} \), quadrature_type=\( \text{'default'} \))

Returns tuple \( (P, W) \) where \( P \) is an array of quadrature points with corresponding weights \( W \).

The quadrature is of order \( \text{order} \) or has \( \text{npoints} \) integration points.

**quadrature_info ()**

Returns a tuple of dicts \( (O, N) \) where \( O[\text{quadrature_type}] \) is a list of orders which are implemented for \( \text{quadrature_type} \) and \( N[\text{quadrature_type}] \) is a list of the corresponding numbers of integration points.

**size**(codim)

Number of subentities of codimension \( \text{codim} \).

**sub_reference_element**(codim)

Returns the reference element of the codim-codim subentities.

**subentities**(codim, subentity_codim)

\( \text{subentities}(c, sc)[i, j] \) is, with respect to the indexing inside the reference element, the index of the \( j \)-th codim-subentity_codim subentity of the \( i \)-th codim-codim subentity of the reference element.
**subentity_embedding** *(subentity_codim)*  
Returns a tuple *(A, B)* which defines the embedding of the codim-*subentity_codim* subentities into the reference element.

For *subentity_codim > 1*, the embedding is by default given recursively via ‘subentity_embedding(subentity_codim - 1)’ and *sub_reference_element(subentity_codim - 1).subentity_embedding(1)* choosing always the superentity with smallest index.

**unit_outer_normals** ()  
*retval[e]* is the unit outer-normal vector to the codim-1 subentity with index *e*.

class `pymor.grids.referenceelements.Point`  
Bases: `pymor.grids.interfaces.ReferenceElementInterface`

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>center</code></td>
<td>Coordinates of the barycenter.</td>
</tr>
<tr>
<td><code>mapped_diameter(A)</code></td>
<td>The diameter of the reference element after transforming it with the matrix <em>A</em> (vectorized).</td>
</tr>
<tr>
<td><code>quadrature(order=None, npoints=None, quadrature_type='default')</code></td>
<td>Returns tuple <em>(P, W)</em> where <em>P</em> is an array of quadrature points with corresponding weights <em>W</em>.</td>
</tr>
<tr>
<td><code>quadrature_info()</code></td>
<td>Returns a tuple of dicts <em>(O, N)</em> where <em>O[quadrature_type]</em> is a list of orders which are implemented for <em>quadrature_type</em> and <em>N[quadrature_type]</em> is a list of the corresponding numbers of integration points.</td>
</tr>
<tr>
<td><code>size(codim)</code></td>
<td>Number of subentities of codimension <em>codim</em>.</td>
</tr>
<tr>
<td><code>sub_reference_element(codim)</code></td>
<td>Returns the reference relement of the codim-<em>codim</em> subentities.</td>
</tr>
<tr>
<td><code>subentities(codim, subentity_codim)</code></td>
<td><em>subentities(c,sc)[i,j]</em> is, with respect to the indexing inside the reference element, the index of the <em>j</em>-th codim-<em>subentity_codim</em> subentity of the <em>i</em>-th codim-<em>codim</em> subentity of the reference element.</td>
</tr>
</tbody>
</table>

**center()**  
Coordinates of the barycenter.

**mapped_diameter(A)**  
The diameter of the reference element after transforming it with the matrix *A* (vectorized).

**quadrature(order=None, npoints=None, quadrature_type='default')**  
Returns tuple *(P, W)* where *P* is an array of quadrature points with corresponding weights *W*.

The quadrature is of order *order* or has *npoints* integration points.

**quadrature_info()**  
Returns a tuple of dicts *(O, N)* where *O[quadrature_type]* is a list of orders which are implemented for *quadrature_type* and *N[quadrature_type]* is a list of the corresponding numbers of integration points.

**size(codim)**  
Number of subentities of codimension *codim*.

**sub_reference_element(codim)**  
Returns the reference relement of the codim-*codim* subentities.

**subentities(codim, subentity_codim)**  
*subentities(c,sc)[i,j]* is, with respect to the indexing inside the reference element, the index of the *j*-th codim-*subentity_codim* subentity of the *i*-th codim-*codim* subentity of the reference element.
subentity_embedding \((\text{subentity_codim})\)

Returns a tuple \((A, B)\) which defines the embedding of the codim-\text{subentity_codim} subentities into the reference element.

For \text{subentity_codim} > 1', the embedding is by default given recursively via \text{subentity_embedding}(\text{subentity_codim} - 1) and \text{sub_reference_element}(\text{subentity_codim} - 1).\text{subentity_embedding}(1) choosing always the superentity with smallest index.

unit_outer_normals()

\text{retval}[e] is the unit outer-normal vector to the codim-1 subentity with index \(e\).

class \text{pymor.grids.referenceelements.Square}

Bases: \text{pymor.grids.interfaces.ReferenceElementInterface}

Methods

\begin{tabular}{|l|l|}
\hline
\text{Square} & \text{center, mapped_diameter, quadrature, quadrature_info, size, sub_reference_element, subentities, subentity_embedding, unit_outer_normals} \\
\hline
\text{ReferenceElementInterface} & \text{quadrature_types, __call__} \\
\hline
\text{CacheableInterface} & \text{disable_caching, enable_caching} \\
\hline
\text{ImmutableInterface} & \text{lock, unlock} \\
\hline
\text{BasicInterface} & \text{add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with\_., __setattr__} \\
\hline
\end{tabular}

Attributes

\begin{tabular}{|l|l|}
\hline
\text{Square} & \text{dim, volume} \\
\hline
\text{CacheableInterface} & \text{cache_region} \\
\hline
\text{ImmutableInterface} & \text{calculate_sid, sid, sid_failure, sid_ignore} \\
\hline
\text{BasicInterface} & \text{locked, logger, logging_disabled, name, uid, with_arguments} \\
\hline
\end{tabular}

center()

Coordinates of the barycenter.

mapped_diameter \((A)\)

The diameter of the reference element after transforming it with the matrix \(A\) (vectorized).

quadrature \((\text{order=\text{None}, npoints=\text{None}, quadrature_type=\text{\textquoteleft default\textquoteright}})\)

Returns tuple \((P, W)\) where \(P\) is an array of quadrature points with corresponding weights \(W\).

The quadrature is of order \text{order} or has \text{npoints} integration points.

quadrature_info()

Returns a tuple of dicts \((O, N)\) where \(O[\text{quadrature_type}]\) is a list of orders which are implemented for \text{quadrature_type} and \(N[\text{quadrature_type}]\) is a list of the corresponding numbers of integration points.

size \((\text{codim})\)

Number of subentities of codimension \text{codim}.

sub_reference_element \((\text{codim})\)

Returns the reference relement of the codim-\text{codim} subentities.

subentities \((\text{codim, subentity_codim})\)

\text{subentities}(c,sc)[i,j] is, with respect to the indexing inside the reference element, the index of the \(j\)-th codim-\text{subentity_codim} subentity of the \(i\)-th codim-\text{codim} subentity of the reference element.
subentity_embedding (subentity_codim)

Returns a tuple \((A, B)\) which defines the embedding of the codim-\(\text{subentity_codim}\) subentities into the reference element.

For \(\text{subentity_codim} > 1\), the embedding is by default given recursively via ‘subentity_embedding(subentity_codim - 1)’ and ‘sub_reference_element(subentity_codim - 1).subentity_embedding(1)’ choosing always the superentity with smallest index.

unit_outer_normals()

\(\text{retval}[e]\) is the unit outer-normal vector to the codim-1 subentity with index \(e\).

class pymor.grids.referenceelements.Triangle

Bases: pymor.grids.interfaces.ReferenceElementInterface

Methods

<table>
<thead>
<tr>
<th>Triangle</th>
<th>center, mapped_diameter, quadrature, quadrature_info, size, sub_reference_element, subentities, subentity_embedding, unit_outer_normals</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReferenceElementInterface</td>
<td>quadrature_types, __call__</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>__caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>__lock, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Triangle</th>
<th>dim, volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheableInterface</td>
<td>cache_region</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

center()

Coordinates of the barycenter.

mapped_diameter (A)

The diameter of the reference element after transforming it with the matrix \(A\) (vectorized).

quadrature (order=None, npoints=None, quadrature_type='default')

Returns tuple \((P, W)\) where \(P\) is an array of quadrature points with corresponding weights \(W\).

The quadrature is of order \(\text{order}\) or has \(\text{npoints}\) integration points.

quadrature_info()

Returns a tuple of dicts \((O, N)\) where \(O[\text{quadrature_type}]\) is a list of orders which are implemented for \(\text{quadrature_type}\) and \(N[\text{quadrature_type}]\) is a list of the corresponding numbers of integration points.

size (codim)

Number of subentities of codimension \(\text{codim}\).

sub_reference_element (codim)

Returns the reference relement of the codim-\(\text{codim}\) subentities.

subentities (codim, subentity_codim)

\(\text{subentities}(c, sc)[i, j]\) is, with respect to the indexing inside the reference element, the index of the \(j\)-th codim-\(\text{subentity_codim}\) subentity of the \(i\)-th codim-\(\text{codim}\) subentity of the reference element.
**subentity_embedding**(subentity_codim)

Returns a tuple \((A, B)\) which defines the embedding of the codim-
subentity_codim subentities into the reference element.

For \(\text{subentity_codim} > 1\) the embedding is by default given recursively via \(\text{subentity_embedding(subentity_codim - 1)}\) and \(\text{sub_reference_element(subentity_codim - 1).subentity_embedding(1)}\) choosing always the superentity with smallest index.

**unit_outer_normals()**

\(\text{retval[e]}\) is the unit outer-normal vector to the codim-1 subentity with index \(e\).

---

**subgrid module**

**class** `pymor.grids.subgrid.SubGrid(grid, entities)`

**Bases:** `pymor.grids.interfaces.AffineGridInterface`

A subgrid of a `Grid`.

Given a `Grid` and a list of codim-0 entities we construct the minimal subgrid of the grid, containing all the given entities.

**Parameters**

- `grid` *Grid* of which a subgrid is to be created.
- `entities` *NumPy array* of global indices of the codim-0 entities which are to be contained in the subgrid.

**Methods**

- `SubGrid` `embeddings`, `indices_from_parent_indices`, `parent_indices`, `size`, `subentities`
- `AffineGridInterface` `centers`, `diameters`, `integration_elements`, `jacobian_inverse_transposed`, `quadrature_points`, `unit_outer_normals`, `volumes`, `volumes_inverse`
- `ConformalTopologicalGridInterface` `boundary_mask`, `neighbours`, `superentities`, `superentity_indices`
- `CacheableInterface` `disable_caching`, `enable_caching`
- `ImmutableInterface` `lock`, `unlock`
- `BasicInterface` `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`

**Attributes**

- `SubGrid` `parent_grid`, `reference_element`
- `AffineGridInterface` `dim_outer`
- `ConformalTopologicalGridInterface`
- `CacheableInterface` `cache_region`
- `ImmutableInterface` `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `BasicInterface` `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`

**parent_grid**

The `Grid` from which the subgrid was constructed. `Subgrid` only stores a `weakref` to the grid, so accessing this property might return `None` if the original grid has been destroyed.
embeddings \((\text{codim})\)

Returns tuple \((A, B)\) where \(A[e]\) and \(B[e]\) are the linear part and the translation part of the map from the reference element of \(e\) to \(e\).

For \(\text{codim} > 0\), we provide a default implementation by taking the embedding of the codim-1 parent entity \(e_0\) of \(e\) with lowest global index and composing it with the subentity_embedding of \(e\) into \(e_0\) determined by the reference element.

indices_from_parent_indices \((\text{ind}, \text{codim})\)

Maps a NumPy array of indices of codim-codim entities of the parent grid to indices of the subgrid.

Raises

ValueError Not all provided indices correspond to entities contained in the subgrid.

parent_indices \((\text{codim})\)

\(\text{reval}[e]\) is the index of the \(e\)-th codim-codim entity in the parent grid.

size \((\text{codim})\)

The number of entities of codimension \(\text{codim}\).

subentities \((\text{codim}, \text{subentity_codim})\)

\(\text{reval}[e,s]\) is the global index of the \(s\)-th codim-subentity_codim subentity of the codim-codim entity with global index \(e\).

Only \(\text{subentities}(\text{codim}, \text{codim}+1)\) has to be implemented; a default implementation is provided which evaluates \(\text{subentities}(\text{codim}, \text{subentity_codim})\) by computing the transitive closure of \(\text{subentities}(\text{codim}, \text{codim}+1)\).

tria module
class pymor.grids.tria.TriaGrid \((\text{num_intervals}=(2, 2), \text{domain}=(\{0, 0\}, \{1, 1\}), \text{identify_left_right}=False, \text{identify_bottom_top}=False)\)

Bases: pymor.grids.interfaces.AffineGridWithOrthogonalCentersInterface

Basic implementation of a triangular grid on a rectangular domain.

The global face, edge and vertex indices are given as follows

<table>
<thead>
<tr>
<th>6</th>
<th>10</th>
<th>7</th>
<th>11</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>10</td>
<td>18</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>11</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>2</td>
<td>26</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
<td>24</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
<td>24</td>
<td>13</td>
</tr>
</tbody>
</table>

Parameters
**num_intervals**  Tuple \((n_0, n_1)\) determining a grid with \(n_0 \times n_1\) codim-0 entities.

**domain**  Tuple \((ll, ur)\) where \(ll\) defines the lower left and \(ur\) the upper right corner of the domain.

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TriaGrid</code></td>
<td>embeddings, orthogonal_centers, size, subentities, visualize, <strong>reduce</strong>, <strong>str</strong></td>
</tr>
<tr>
<td><code>AffineGridInterface</code></td>
<td>centers, diameters, integration_elements, jacobian_inverse_transposed, quadrature_points, unit_outer_normals, volumes, volumes_inverse</td>
</tr>
<tr>
<td><code>ConformalTopologicalGridInterface</code></td>
<td>boundaries, boundary_mask, neighbours, superentities, superentity_indices</td>
</tr>
<tr>
<td><code>CacheableInterface</code></td>
<td>disable_caching, enable_caching</td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td>lock, unlock</td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TriaGrid</code></td>
<td>dim, dim_outer, reference_element</td>
</tr>
<tr>
<td><code>CacheableInterface</code></td>
<td>cache_region</td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

**__reduce__**( )

helper for pickle

**__str__**( ) \(\iff \) \(\text{str}(x)\)

**embeddings**(codim=0)

Returns tuple \((A, B)\) where \(A[e]\) and \(B[e]\) are the linear part and the translation part of the map from the reference element of \(e\) to \(e\).

For \(\text{codim} > 0\), we provide a default implementation by taking the embedding of the codim-1 parent entity \(e_0\) of \(e\) with lowest global index and composing it with the subentity_embedding of \(e\) into \(e_0\) determined by the reference element.

**size**(codim=0)

The number of entities of codimension \(\text{codim}\).

**subentities**(codim, subentity_codim)

\(\text{retval}[e, s]\) is the global index of the \(s\)-th codim-\(\text{subentity_codim}\) subentity of the codim-\(\text{codim}\) entity with global index \(e\).

Only \(\text{subentities}(\text{codim}, \text{codim}+1)\) has to be implemented; a default implementation is provided which evaluates \(\text{subentities}(\text{codim}, \text{subentity_codim})\) by computing the transitive closure of \(\text{subentities}(\text{codim}, \text{codim}+1)\).

**visualize**(U, codim=2, **kwargs)

Visualize scalar data associated to the grid as a patch plot.

**Parameters**

- \(U\)  VectorArray of the data to visualize. If \(\text{len}(U) > 1\), the data is visualized as a time series of plots. Alternatively, a tuple of VectorArrays can be provided, in which case a subplot is created for each entry of the tuple. The lengths of all arrays have to agree.
The codimension of the entities the data in $U$ is attached to (either 0 or 2).

**kwargs

See `visualize_patch`

### pymor.gui package

#### Submodules

**gl module**  This module provides a widget for displaying patch plots of scalar data assigned to 2D-grids using OpenGL. This widget is not intended to be used directly. Instead, use `visualize_patch` or `PatchVisualizer`.

```python
class pymor.gui.gl.ColorBarWidget (parent, U=None, vmin=None, vmax=None)
Bases: QGLWidget

Methods
ColorBarWidget initializeGL, paintGL, resizeGL, set
```

```python
class pymor.gui.gl.GLPatchWidget (parent, grid, vmin=None, vmax=None, bounding_box=(0, 0), (1, 1), codim=2)
Bases: QGLWidget

Methods
GLPatchWidget initializeGL, paintGL, resizeGL, set, set_coordinates
```

```python
pymor.gui.gl.compile_vertex_shader (source)
Compile a vertex shader from source.
```

```python
pymor.gui.gl.link_shader_program (vertex_shader)
Create a shader program with from compiled shaders.
```

**matplotlib module**  This module provides a widgets for displaying plots of scalar data assigned to one- and two-dimensional grids using matplotlib. This widget is not intended to be used directly. Instead, use `visualize_matplotlib_1d` or `Matplotlib1DVisualizer`.

```python
class pymor.gui.matplotlib.Matplotlib1DWidget (parent, grid, count, vmin=None, vmax=None, legend=None, codim=1, separate_plots=False, dpi=100)
Bases: FigureCanvasQTAgg

Methods
Matplotlib1DWidget set
```

4.1. pymor package
class `pymor.gui.matplotlib.MatplotlibPatchWidget` *(parent, grid, bounding_box=None, vmin=None, vmax=None, codim=2, dpi=100)*

Bases: `FigureCanvasQTAgg`

Methods

**MatplotlibPatchWidget**

`set`

**qt module**

This module provides a few methods and classes for visualizing data associated to grids. We use the PySide bindings for the Qt widget toolkit for the GUI.

class `pymor.gui.qt.Matplotlib1DVisualizer` *(grid, codim=1, block=False)*

Bases: `pymor.core.interfaces.BasicInterface`

Visualize scalar data associated to a one-dimensional Grid as a plot.

The grid’s `ReferenceElement` must be the line. The data can either be attached to the subintervals or vertices of the grid.

Parameters

grid The underlying `Grid`.

codim The codimension of the entities the data in `U` is attached to (either 0 or 1).

block If `True`, block execution until the plot window is closed.

Methods

**Matplotlib1DVisualizer**

`visualize`

`add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `unlock`, `with_`, `__setattr__`

Attributes

**BasicInterface**

`locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`

`visualize` *(U, discretization, title=None, legend=None, block=None)*

Visualize the provided data.

Parameters

U `VectorArray` of the data to visualize. If `len(U) > 1`, the data is visualized as a time series of plots. Alternatively, a tuple of `VectorArrays` can be provided, in which case several plots are made into the same axes. The lengths of all arrays have to agree.

discretization Filled in `pymor.discretizations.DiscretizationBase.visualize` (ignored).

title Title of the plot.

legend Description of the data that is plotted. Most useful if `U` is a tuple in which case `legend` has to be a tuple of strings of the same length.

block If `True` block execution until the plot window is closed. If `None`, use the default provided during instantiation.
class pymor.gui.qt.PatchVisualizer(grid, bounding_box=(0, 0), [1, 1], codim=2, backend=None, block=False)

Bases: pymor.core.interfaces.BasicInterface

Visualize scalar data associated to a two-dimensional Grid as a patch plot.

The grid’s ReferenceElement must be the triangle or square. The data can either be attached to the faces or vertices of the grid.

Parameters

grid  The underlying Grid.

bounding_box  A bounding box in which the grid is contained.

codim  The codimension of the entities the data in U is attached to (either 0 or 2).

backend  Plot backend to use (‘gl’ or ‘matplotlib’).

block  If True block execution until the plot window is closed.

Methods

PatchVisualizer

BasicInterface

add_attributes, disable_logging, enable_logging,

has_interface_name, implementor_names, implementors, lock, unlock,

with..., __setattr__

Attributes

BasicInterface

locked, logger, logging_disabled, name, uid, with_arguments

visualize(U, discretization, title=None, legend=None, separate_colorbars=False, block=None, filename=None, columns=2)

Visualize the provided data.

Parameters

U  VectorArray of the data to visualize. If len(U) > 1, the data is visualized as a time series of plots. Alternatively, a tuple of VectorArrays can be provided, in which case a subplot is created for each entry of the tuple. The lengths of all arrays have to agree.

discretization  Filled in pymor.discretizations.DiscretizationBase.visualize (ignored).

title  Title of the plot.

legend  Description of the data that is plotted. Most useful if U is a tuple in which case legend has to be a tuple of strings of the same length.

separate_colorbars  If True, use separate colorbars for each subplot.

block  If True, block execution until the plot window is closed. If None, use the default provided during instantiation.

filename  If specified, write the data to a VTK-file using pymor.tools.vtkio.write_vtk instead of displaying it.

columns  The number of columns in the visualizer GUI in case multiple plots are displayed at the same time.
class pymor.gui.qt.PlotMainWindow(U, plot, length=1, title=None)

Bases: object

Base class for plot main windows.

Methods

- rewind, slider_changed, speed_changed, step_backward, step_forward, to_end, toggle_play, update_solution

pymor.gui.qt._launch_qt_app(main_window_factory, block)

Wrapper to display plot in a separate process.

pymor.gui.qt.visualize_matplotlib_1d(grid, U, codim=1, title=None, legend=None, separate_plots=False, block=False)

Visualize scalar data associated to a one-dimensional Grid as a plot.

The grid’s ReferenceElement must be the line. The data can either be attached to the subintervals or vertices of the grid.

Parameters

- grid  The underlying Grid.
- U  VectorArray of the data to visualize. If len(U) > 1, the data is visualized as a time series of plots. Alternatively, a tuple of VectorArrays can be provided, in which case several plots are made into the same axes. The lengths of all arrays have to agree.
- codim  The codimension of the entities the data in U is attached to (either 0 or 1).
- title  Title of the plot.
- legend  Description of the data that is plotted. Most useful if U is a tuple in which case legend has to be a tuple of strings of the same length.
- separate_plots  If True, use subplots to visualize multiple VectorArrays.
- block  If True, block execution until the plot window is closed.

pymor.gui.qt.visualize_patch(grid, U, bounding_box=[[0, 0], [1, 1]], codim=2, title=None, legend=None, separate_colorbars=False, backend='gl', block=False, columns=2)

Visualize scalar data associated to a two-dimensional Grid as a patch plot.

The grid’s ReferenceElement must be the triangle or square. The data can either be attached to the faces or vertices of the grid.

Parameters

- grid  The underlying Grid.
- U  VectorArray of the data to visualize. If len(U) > 1, the data is visualized as a time series of plots. Alternatively, a tuple of VectorArrays can be provided, in which case a subplot is created for each entry of the tuple. The lengths of all arrays have to agree.
**bounding_box**  A bounding box in which the grid is contained.

**codim**  The codimension of the entities the data in $U$ is attached to (either 0 or 2).

**title**  Title of the plot.

**legend**  Description of the data that is plotted. Most useful if $U$ is a tuple in which case **legend** has to be a tuple of strings of the same length.

**separate_colorbars**  If True, use separate colorbars for each subplot.

**backend**  Plot backend to use (‘gl’ or ‘matplotlib’).

**block**  If True, block execution until the plot window is closed.

**columns**  The number of columns in the visualizer GUI in case multiple plots are displayed at the same time.

---

**Defaults**

backend (see pymor.core.defaults)

---

**pymor.la package**

**Submodules**

---

**basic module**

**class pymor.la.basic.InducedNorm**(product, raise_negative, tol, name)

**Bases:** pymor.core.interfaces.ImmutableInterface, pymor.parameters.base.Parametric

Instantiated by induced_norm. Do not use directly.

**Methods**

<table>
<thead>
<tr>
<th>InducedNorm</th>
<th><strong>call</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>lock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_.<strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>calculate_sid, sid, sid_failure, sid_ignore</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

___call__(...) \(\iff x(...)\)

**pymor.la.basic.cat_arrays**(vector_arrays)

Return a new VectorArray which a concatenation of the arrays in `vector_arrays`.  

---

**4.1. pymor package**
\textbf{pymor.la.basic\textunderscore induced\_norm}\,(\textit{product, raise\_negative=True, tol=1e-10, name=None})

The induced norm of a scalar product.

\begin{verbatim}
    product.apply2(U, U, mu=mu, pairwise=True)
\end{verbatim}

The norm of a the vectors in a \texttt{VectorArray} $U$ is calculated by calling

In addition, negative norm squares of absolute value smaller than $tol$ are clipped to 0. If $raise\_negative$ is True, a \texttt{ValueError} exception is raised if there are still negative norm squares afterwards.

\begin{description}
\item[Parameters]
\begin{itemize}
\item \textit{product}  The scalar product \texttt{Operator} for which the norm is to be calculated.
\item \textit{raise\_negative}  If True, raise an exception if calculated norm is negative.
\item \textit{tol}  See above.
\end{itemize}
\end{description}

\begin{description}
\item[Returns]
\begin{itemize}
\item \textit{norm}  A function \texttt{norm(U, mu=None)} taking a \texttt{VectorArray} $U$ as input together with the \texttt{Parameter} $mu$ which is passed to the product.
\end{itemize}
\end{description}

\begin{description}
\item[Defaults]
\begin{itemize}
\item \texttt{raise\_negative, tol} (see \texttt{pymor.core.defaults})
\end{itemize}
\end{description}

\begin{description}
\item[genericsolvers module]  This module contains some iterative linear solvers which only use the \texttt{Operator} interface
\end{description}

\textbf{pymor.la.genericsolvers\textunderscore apply\_inverse}\,(\textit{op, rhs, options=None})

Solve linear equation system.

\begin{verbatim}
    applies the inverse of \textit{op} to the vectors in \textit{rhs}.
\end{verbatim}

\begin{description}
\item[Parameters]
\begin{itemize}
\item \textit{op}  The linear, non-parametric \texttt{Operator} to invert.
\item \textit{rhs}  \texttt{VectorArray} of right-hand sides for the equation system.
\item \textit{options}  \texttt{invert\_options} to use. (See \texttt{invert\_options}.)
\end{itemize}
\end{description}

\begin{description}
\item[Returns]
\begin{itemize}
\item \texttt{VectorArray} of the solution vectors.
\end{itemize}
\end{description}
Returns invert options (with default values) for arbitrary linear Operators.

Parameters

default_solver Default solver to use (generic_lgmres, least_squares_generic_lsmr, least_squares_generic_lsqr).
default_least_squares_solver Default solver to use for least squares problems (least_squares_generic_lsmr, least_squares_generic_lsqr).
generic_lgmres_tol See scipy.sparse.linalg.lgmres.
generic_lgmres_maxiter See scipy.sparse.linalg.lgmres.
generic_lgmres_inner_m See scipy.sparse.linalg.lgmres.
generic_lgmres_outer_k See scipy.sparse.linalg.lgmres.
least_squares_generic_lsmr_damp See scipy.sparse.linalg.lsmr.
least_squares_generic_lsmr_atol See scipy.sparse.linalg.lsmr.
least_squares_generic_lsmr_btol See scipy.sparse.linalg.lsmr.
least_squares_generic_lsmr_conlim See scipy.sparse.linalg.lsmr.
least_squares_generic_lsmr_maxiter See scipy.sparse.linalg.lsmr.
least_squares_generic_lsmr_show See scipy.sparse.linalg.lsmr.
least_squares_generic_lsqr_damp See scipy.sparse.linalg.lsqr.
least_squares_generic_lsqr_atol See scipy.sparse.linalg.lsqr.
least_squares_generic_lsqr_btol See scipy.sparse.linalg.lsqr.
least_squares_generic_lsqr_conlim See scipy.sparse.linalg.lsqr.
least_squares_generic_lsqr_iter_lim See scipy.sparse.linalg.lsqr.
least_squares_generic_lsqr_show See scipy.sparse.linalg.lsqr.

Returns
A tuple of all possible invert options.
**Defaults**

default_solver, default_least_squares_solver, generic_lgmres_tol, generic_lgmres_maxiter,
generic_lgmres_inner_m, generic_lgmres_outer_k, least_squares_generic_lsmr_damp,
least_squares_generic_lsmr_atol, least_squares_generic_lsmr_btol, least_squares_generic_lsmr_conlim,
least_squares_generic_lsmr_maxiter, least_squares_generic_lsmr_show, least_squares_generic_lsqr_atol,
least_squares_generic_lsqr_btol, least_squares_generic_lsqr_conlim, least_squares_generic_lsqr_iter_lim,
least_squares_generic_lsqr_show (see pymor.core.defaults)

```python
pymor.la.genericsolvers.lgmres(A, b, x0=None, tol=1e-05, maxiter=1000, M=None, callback=None, inner_m=30, outer_k=3, outer_y=None, store_outer_Av=True)
```

```python
pymor.la.genericsolvers.lsmr(A, b, damp=0.0, atol=1e-06, btol=1e-06, conlim=100000000.0, maxiter=None, show=False)
```

```python
pymor.la.genericsolvers.lsqr(A, b, damp=0.0, atol=1e-08, btol=1e-08, conlim=100000000.0, iter_lim=None, show=False)
```

**gram_schmidt module**

```python
pymor.la.gram_schmidt.gram_schmidt(A, product=None, atol=1e-13, rtol=1e-13, offset=0, find_duplicates=True, reiterate=True, reiteration_threshold=0.1, check=True, check_tol=0.001, copy=False)
```

Orthonormalize a `VectorArray` using the stabilized Gram-Schmidt algorithm.

**Parameters**

- **A** The `VectorArray` which is to be orthonormalized.
- **product** The scalar product w.r.t. which to orthonormalize, given as a linear `Operator`. If `None` the Euclidean product is used.
- **atol** Vectors of norm smaller than `atol` are removed from the array.
- **rtol** Relative tolerance used to detect linear dependent vectors (which are then removed from the array).
- **offset** Assume that the first `offset` vectors are already orthogonal and start the algorithm at the `offset + 1`-th vector.
- **find_duplicates** If `True`, eliminate duplicate vectors before the main loop.
- **reiterate** If `True`, orthonormalize again if the norm of the orthogonalized vector is much smaller than the norm of the original vector.
- **reiteration_threshold** If `reiterate` is `True`, re-orthonormalize if the ratio between the norms of the orthogonalized vector and the original vector is smaller than this value.
- **check** If `True`, check if the resulting `VectorArray` is really orthonormal.
- **check_tol** Tolerance for the check.
copy  If True, create a copy of A instead of modifying A itself.

Returns
The orthonormalized VectorArray.

Defaults
atol, rtol, find_duplicates, reiterate, reiteration_threshold, check, check_tol (see pymor.core.defaults)

interfaces module
class pymor.la.interfaces.VectorArrayInterface
Bases: pymor.core.interfaces.BasicInterface

Interface for vector arrays.

A vector array should be thought of as a list of (possibly high-dimensional) vectors. While the vectors themselves will be inaccessible in general (e.g. because they are managed by external code on large systems), operations on the vectors like addition can be performed via the interface.

It is moreover assumed that the number of vectors is small enough such that scalar data associated to each vector can be handled on the Python side. I.e. methods like l2_norm or gramian will always return NumPy arrays.

An implementation of the interface via NumPy arrays is given by NumpyVectorArray. In general, it is the implementors decision how memory is allocated internally (e.g. continuous block of memory vs. list of pointers to the individual vectors.) Thus no general assumptions can be made on the costs of operations like appending to or removing vectors from the array. As a hint for ‘continuous block of memory’ implementations, VectorArray constructors should provide a reserve keyword argument which allows to specify to what size the array is assumed to grow.

Most methods provide ind and/or o_ind arguments which are used to specify on which vectors the method is supposed to operate. If ind (o_ind) is None the whole array is selected. Otherwise, ind can be a single index in range(len(self)), a list of indices or a one-dimensional NumPy array of indices. An index can be repeated in which case the corresponding vector is selected several times.

Methods
VectorArray Interface: almost_equal, amax, append, axpy, check_ind, check_ind_unique, components, copy, dot, empty, gramian, l1_norm, l2_norm, len_ind, len_ind_unique, lincomb, make_array, remove, replace, scal, sup_norm, zeros, __add__, __iadd__, __imul__, __isub__, __len__, __mul__, __neg__, __radd__, __sub__
BasicInterface: add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with_, __setattr__

Attributes
VectorArrayInterface: data, dim, space, subtype
BasicInterface: locked, logger, logging_disabled, name, uid, with_arguments

4.1. pymor package
data
Implementors can provide a data property which returns a NumPy array of shape \((\text{len}(v), v.\text{dim})\) containing the data stored in the array. Access should be assumed to be slow and is mainly intended for debugging/visualization purposes or to once transfer data to pyMOR and further process it using NumPy. In the case of NumpyVectorArray, an actual view of the internally used NumPy array is returned, so changing it, will alter the VectorArray. Thus, you cannot assume to own the data returned to you, in general.

dim
The dimension of the vectors in the array.

space
VectorSpace array the array belongs to.

subtype
Can be any Python object with a sensible implementation of \_eq_. Two arrays are compatible (e.g. can be added) if they are instances of the same class and share the same subtype. A valid subtype has to be provided to make_array and the resulting array will be of that subtype. By default, the subtype of an array is simply None. For NumpyVectorArray, the subtype is a single integer denoting the dimension of the array. Subtypes for other array classes could, e.g., include a socket for communication with a specific PDE solver instance.

\_add\_ (other)
The pairwise sum of two VectorArrays.

\_iadd\_ (other)
In-place pairwise addition of VectorArrays.

\_imul\_ (other)
In-place product by a scalar.

\_isub\_ (other)
In-place pairwise difference of VectorArrays.

\_len\_ ()
The number of vectors in the array.

\_mul\_ (other)
Product by a scalar.

\_neg\_ ()
Product by -1.

\_radd\_ (other)
The pairwise sum of two VectorArrays.

\_sub\_ (other)
The pairwise difference of two VectorArrays.

almost_equal (other, ind=None, o_ind=None, rtol=None, atol=None)
Check vectors for equality.

Equality of two vectors should be defined as in pymor.tools.float_cmp_all.

The subtypes of self and other have to agree. If the length of self (ind) resp. other (o_ind) is 1, the one specified vector is compared to all vectors of the other summand. Otherwise the length of ind and o_ind have to agree.

Parameters

other A VectorArray containing the vectors to compare with.

ind Indices of the vectors that are to be compared (see class documentation).
**Indices of the vectors in other** that are to be compared (see class documentation).

**rtol** See `pymor.tools.float_cmp_all`  
**atol** See `pymor.tools.float_cmp_all`  

**Returns**  
NumPy array of the truth values of the comparison.

### amax

**amax** *(ind=None)*  
The maximum absolute value of the vectors contained in the array.

**Parameters**  
**ind** Indices of the vectors whose maximum absolute value is to be calculated (see class documentation).

**Returns**  
**max_ind** NumPy array containing for each vector an index at which the maximum is attained.  
**max_val** NumPy array containing for each vector the maximum absolute value of its components.

### append

**append** *(other, o_ind=None, remove_from_other=False)*  
Append vectors to the array.

**Parameters**  
**other** A VectorArray containing the vectors to be appended.  
**o_ind** Indices of the vectors that are to be appended (see class documentation).  
**remove_from_other** If True, the appended vectors are removed from other. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.

### axpy

**axpy** *(alpha, x, ind=None, x_ind=None)*  
BLAS AXPY operation.

This method forms the sum  
\[
\text{self}[\text{ind}] = \alpha \times \text{x}[\text{x_ind}] + \text{self}[\text{ind}]
\]

The subtypes of self and x have to agree. If the length of x (x_ind) is 1, the same x vector is used for all vectors in self. Otherwise, the lengths of self (ind) and x (x_ind) have to agree. If alpha is a scalar, each x vector is multiplied with the same factor alpha. Otherwise, alpha has to be a one-dimensional NumPy array of the same length as self (ind) containing the coefficients for each x vector.

**Parameters**  
**alpha** The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in x are multiplied.  
**x** A VectorArray containing the x-summands.  
**ind** Indices of the vectors of self that are to be added (see class documentation). Repeated indices are forbidden.  
**x_ind** Indices of the vectors in x that are to be added (see class documentation). Repeated indices are allowed.
**check_ind**(ind)
Check if *ind* is an admissible list of indices in the sense of the class documentation.

**check_ind_unique**(ind)
Check if *ind* is an admissible list of non-repeated indices in the sense of the class documentation.

**components**(component_indices, ind=None)
Extract components of the vectors contained in the array.

**Parameters**

- **component_indices** List or 1D *NumPy array* of indices of the vector components that are to be returned.
- **ind** Indices of the vectors whose components are to be calculated (see class documentation).

**Returns**

A *NumPy array* result such that result[i, j] is the component_indices[j]-th component of the ind[i]-th vector of the array.

**copy**(ind=None)
Returns a copy of a subarray.

**Parameters**

- **ind** Indices of the vectors that are to be copied (see class documentation).

**Returns**

A copy of the *VectorArray*.

**dot**(other, pairwise, ind=None, o_ind=None)
Returns the scalar products between *VectorArray* elements.

**Parameters**

- **other** A *VectorArray* containing the second factors.
- **pairwise** See return value documentation.
- **ind** Indices of the vectors whose scalar products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in *other* whose scalar products are to be taken (see class documentation).

**Returns**

If pairwise is *True*, returns a *NumPy array* result such that :
result[i] = (self[ind][i], other[o_ind][i]).
If pairwise is *False*, returns a *NumPy array* result such that :
result[i, j] = (self[ind][i], other[o_ind][j]).

**empty**(reserve=0)
Create an empty *VectorArray* of the same *subtype*.

**Parameters**
reserve Hint for the backend to which length the array will grow.

Returns
An empty VectorArray.

gramian \((ind=\text{None})\)
Shorthand for \(\text{dot}(\text{self}, \text{pairwise}=\text{False}, \text{ind}=\text{ind}, \text{o\_ind}=\text{ind})\).

\[ \text{l1\_norm}(\text{ind}=\text{None}) \]
The l1-norms of the vectors contained in the array.

Parameters
\text{ind} Indices of the vectors whose norms are to be calculated (see class documentation).

Returns
A NumPy array result such that \(\text{result}[i]\) contains the norm of \(\text{self}[\text{ind}][i]\).

\[ \text{l2\_norm}(\text{ind}=\text{None}) \]
The l2-norms of the vectors contained in the array.

Parameters
\text{ind} Indices of the vectors whose norms are to be calculated (see class documentation).

Returns
A NumPy array result such that \(\text{result}[i]\) contains the norm of \(\text{self}[\text{ind}][i]\).

\[ \text{len\_ind}(\text{ind}) \]
Return the number of specified indices.

\[ \text{len\_ind\_unique}(\text{ind}) \]
Return the number of specified unique indices.

\[ \text{lincomb}(\text{coefficients}, \text{ind}=\text{None}) \]
Returns linear combinations of the vectors contained in the array.

Parameters
\text{coefficients} A NumPy array of dimension 1 or 2 containing the linear coefficients. \text{coefficients.shape[-1]} has to agree with \(\text{len}(\text{self})\).
\text{ind} Indices of the vectors which are linear combined (see class documentation).

Returns
A VectorArray result such that :
\[
\begin{align*}
\text{result}[i] &= \text{self}[j] \times \text{coefficients}[i,j] \\
\text{in case coefficients is of dimension 2, otherwise len(result) == 1 and} \\
\text{result}[1] &= \text{self}[j] \times \text{coefficients}[j].
\end{align*}
\]
**classmethod make_array** *(subtype=None, count=0, reserve=0)*

Create a VectorArray of null vectors.

**Parameters**

subtype The subtype, the created array should have. What a valid subtype is, is determined by the respective array implementation.

count The number of null vectors to create. For count == 0, an empty array is returned.

reserve A hint for the backend to which length the array will grow.

**remove** *(ind=None)*

Remove vectors from the array.

**Parameters**

ind Indices of the vectors that are to be removed (see class documentation).

**replace** *(other, ind=None, o_ind=None, remove_from_other=False)*

Replace vectors of the array.

**Parameters**

other A VectorArray containing the replacement vectors.

ind Indices of the vectors that are to be replaced (see class documentation). Repeated indices are forbidden.

o_ind Indices of the replacement vectors (see class documentation). len(ind) has to agree with len(o_ind). Repeated indices are allowed.

remove_from_other If True, the new vectors are removed from other. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.

**scal** *(alpha, ind=None)*

BLAS SCAL operation (in-place scalar multiplication).

This method calculates

\[
\text{self[ind]} = \alpha \times \text{self[ind]}
\]

If alpha is a scalar, each vector is multiplied by this scalar. Otherwise, alpha has to be a one-dimensional NumPy array of the same length as self (ind) containing the factors for each vector.

**Parameters**

alpha The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in self are multiplied.

ind Indices of the vectors of self that are to be scaled (see class documentation). Repeated indices are forbidden.

**sup_norm** *(ind=None)*

The l-infinity--norms of the vectors contained in the array.

**Parameters**

ind Indices of the vectors whose norms are to be calculated (see class documentation).
Returns

A NumPy array result such that result[i] contains the norm of self[ind][i].

zeros(count=1)

Create a VectorArray of null vectors of the same subtype.

Parameters

count  The number of vectors.

Returns

A VectorArray containing count vectors with each component zero.

class pymor.la.interfaces.VectorSpace(space_type, subtype=None)

Bases: pymor.core.interfaces.BasicInterface

Class describing a vector space.

A vector space is simply the combination of a VectorArray type and a subtype. This data is exactly sufficient to construct new arrays using the make_array method. (See the implementation of zeros.)

A VectorArray U is contained in a vector space space, if

type(U) == space.type and U.subtype == space.subtype

Methods

empty, zeros, __contains__, __eq__, __ne__

BasicInterface

attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, lock, unlock,
with, __setattr__

Attributes

<table>
<thead>
<tr>
<th>VectorSpace</th>
<th>dim, subtype, type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

type

The type of VectorArrays in the space.

subtype

The subtype used to construct arrays of the given space.

__contains__(other)

A VectorArray is contained in the space, iff it is an instance of its type and has the same subtype.

__eq__(other)

x.__eq__(y) <=> x==y

__ne__(other)

x.__ne__(y) <=> x!=y

empty(reserve=0)

Create an empty VectorArray.

Parameters
reserve  Hint for the backend to which length the array will grow.

returns
An empty VectorArray.

zeros (count=1)
Create a VectorArray of null vectors

parameters
count The number of vectors.

returns
A VectorArray containing count vectors with each component zero.

inverse module
pymor.la.inverse.inv_transposed_two_by_two(A)
Efficiently compute the transposed inverses of a NumPy array of 2x2-matrices

   retval[i1,...,ik,m,n] = numpy.linalg.inv(A[i1,...,ik,:,:]).

pymor.la.inverse.inv_two_by_two(A)
Efficiently compute the inverses of a NumPy array of 2x2-matrices

   retval[i1,...,ik,m,n] = numpy.linalg.inv(A[i1,...,ik,:,:]).

listvectorarray module
class pymor.la.listvectorarray.ListVectorArray (vectors, subtype=(), copy=True)
Bases: pymor.la.interfaces.VectorArrayInterface

VectorArray implementation via a python list of vectors.

The subtypes a ListVectorArray can have are tuples (vector_type, vector_subtype) where vector_type is a subclass of VectorInterface and vector_subtype is a valid subtype for vector_type.

methods
ListVectorArray: almost_equal, amax, append, axpy, components, copy, dot, gramian, l1_norm, l2_norm, lincomb, make_array, remove, replace, scal, __len__, __str__

VectorArray: check_ind, check_ind_unique, empty, len_ind, len_ind_unique, sup_norm, zeros, __add__, __iadd__, __imul__, __isub__, __mul__, __neg__, __radd__, __sub__, __rsub__, __rtruediv__, __radd__, __rsub__

BasicInterface: add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with, __setattr__

attributes
__len__()  
The number of vectors in the array.

__str__() <==> str(x)

almost_equal (other, ind=None, o_ind=None, rtol=None, atol=None)  
Check vectors for equality.

Equality of two vectors should be defined as in pymor.tools.float_cmp_all.

The subtypes of self and other have to agree. If the length of self (ind) resp. other (o_ind) is 1, the one specified vector is compared to all vectors of the other summand. Otherwise the length of ind and o_ind have to agree.

Parameters
other  A VectorArray containing the vectors to compare with.
ind  Indices of the vectors that are to be compared (see class documentation).
o_ind  Indices of the vectors in other that are to be compared (see class documentation).
rtol  See pymor.tools.float_cmp_all
atol  See pymor.tools.float_cmp_all

Returns
NumPy array of the truth values of the comparison.

amax (ind=None)  
The maximum absolute value of the vectors contained in the array.

Parameters
ind  Indices of the vectors whose maximum absolute value is to be calculated (see class documentation).

Returns
max_ind  NumPy array containing for each vector an index at which the maximum is attained.
max_val  NumPy array containing for each vector the maximum absolute value of its components.

append (other, o_ind=None, remove_from_other=False)  
Append vectors to the array.

Parameters
other  A VectorArray containing the vectors to be appended.
o_ind  Indices of the vectors that are to be appended (see class documentation).
remove_from_other  If True, the appended vectors are removed from other. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.
**axpy** *(alpha, x, ind=None, x_ind=None)*

BLAS AXPY operation.

This method forms the sum

\[ \text{self}[\text{ind}] = \alpha \times \text{x}[\text{x_ind}] + \text{self}[\text{ind}] \]

The subtypes of *self* and *x* have to agree. If the length of *x* (*x_ind*) is 1, the same *x* vector is used for all vectors in *self*. Otherwise, the lengths of *self* (*ind*) and *x* (*x_ind*) have to agree. If *alpha* is a scalar, each *x* vector is multiplied with the same factor *alpha*. Otherwise, *alpha* has to be a one-dimensional *NumPy array* of the same length as *self* (*ind*) containing the coefficients for each *x* vector.

**Parameters**

- **alpha** The scalar coefficient or one-dimensional *NumPy array* of coefficients with which the vectors in *x* are multiplied.
- **x** A *VectorArray* containing the *x*-summands.
- **ind** Indices of the vectors of *self* that are to be added (see class documentation). Repeated indices are forbidden.
- **x_ind** Indices of the vectors in *x* that are to be added (see class documentation). Repeated indices are allowed.

**components** *(component_indices, ind=None)*

Extract components of the vectors contained in the array.

**Parameters**

- **component_indices** List or 1D *NumPy array* of indices of the vector components that are to be returned.
- **ind** Indices of the vectors whose components are to be calculated (see class documentation).

**Returns**

A *NumPy array* *result* such that *result[i, j]* is the *component_indices[j]*-th component of the *ind[i]*-th vector of the array.

**copy** *(ind=None)*

Returns a copy of a subarray.

**Parameters**

- **ind** Indices of the vectors that are to be copied (see class documentation).

**Returns**

A copy of the *VectorArray*.

**dot** *(other, pairwise, ind=None, o_ind=None)*

Returns the scalar products between *VectorArray* elements.

**Parameters**

- **other** A *VectorArray* containing the second factors.
- **pairwise** See return value documentation.
**ind**  Indices of the vectors whose scalar products are to be taken (see class documentation).

**o_ind**  Indices of the vectors in *other* whose scalar products are to be taken (see class documentation).

---

**Returns**
If pairwise is *True*, returns a **NumPy array result** such that :

\[
\text{result}[i] = (\text{self}[\text{ind}][i], \text{other}[\text{o_ind}][i]).
\]

If pairwise is *False*, returns a **NumPy array result** such that :

\[
\text{result}[i, j] = (\text{self}[\text{ind}][i], \text{other}[\text{o_ind}][j]).
\]

---

**gramian**(ind=None)
Shorthand for *dot*(self, pairwise=False, ind=ind, o_ind=ind).

---

**l1_norm**(ind=None)
The 1-norms of the vectors contained in the array.

**Parameters**

- **ind**: Indices of the vectors whose norms are to be calculated (see class documentation).

---

**Returns**
A **NumPy array result** such that result[i] contains the norm of self[ind][i].

---

**l2_norm**(ind=None)
The 2-norms of the vectors contained in the array.

**Parameters**

- **ind**: Indices of the vectors whose norms are to be calculated (see class documentation).

---

**Returns**
A **NumPy array result** such that result[i] contains the norm of self[ind][i].

---

**lincomb**(coefficients, ind=None)
Returns linear combinations of the vectors contained in the array.

**Parameters**

- **coefficients**: A **NumPy array** of dimension 1 or 2 containing the linear coefficients. coefficients.shape[-1] has to agree with len(self).
- **ind**: Indices of the vectors which are linear combined (see class documentation).

---

**Returns**
A **VectorArray result** such that :

\[
\text{result}[i] = \text{self}[j] * \text{coefficients}[i,j]
\]
in case coefficients is of dimension 2, otherwise len(result) == 1 and

\[
\text{result}[1] = \text{self}[j] * \text{coefficients}[j].
\]
remove (ind=None)
Remove vectors from the array.

Parameters
ind  Indices of the vectors that are to be removed (see class documentation).

replace (other, ind=None, o_ind=None, remove_from_other=False)
Replace vectors of the array.

Parameters
other  A VectorArray containing the replacement vectors.
ind  Indices of the vectors that are to be replaced (see class documentation). Repeated indices are forbidden.
o_ind  Indices of the replacement vectors (see class documentation). len(ind) has to agree with len(o_ind). Repeated indices are allowed.
remove_from_other  If True, the new vectors are removed from other. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.

scal (alpha, ind=None)
BLAS SCAL operation (in-place scalar multiplication).
This method calculates
self[ind] = alpha*self[ind]

If alpha is a scalar, each vector is multiplied by this scalar. Otherwise, alpha has to be a one-dimensional NumPy array of the same length as self (ind) containing the factors for each vector.

Parameters
alpha  The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in self are multiplied.
ind  Indices of the vectors of self that are to be scaled (see class documentation). Repeated indices are forbidden.

class pymor.la.listvectorarray. NumpyListVectorArray (vectors, subtype=(), copy=True)
Bases: pymor.la.listvectorarray.ListVectorArray

Methods
ListVectorArray: almost_equal, amax, append, axpy, components, copy, dot, gramian, l1_norm, l2_norm, lincomb, make_array, remove, replace, scal, __len__, __str__,
VectorArray: check_ind, check_ind_unique, empty, len_ind, len_ind_unique, sup_norm, zeros, __add__, __iadd__, __imul__, __isub__, __mul__, __neg__, __radd__, __sub__,
BasicInterface: add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with, __setattr__

Attributes
class pymor.la.listvectorarray.NumpyVector

Bases: pymor.la.listvectorarray.VectorInterface

Vector stored in a NumPy 1D-array.

Methods

NumpyVector: almost_equal, amax, axpy, components, copy, dot, l1_norm, l2_norm, make_zeros, scal
VectorInterface: sup_norm
BasicInterface: add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with_, __setattr__

Attributes

NumpyVector: data, dim, subtype
BasicInterface: locked, logger, logging_disabled, name, uid, with_arguments

class pymor.la.listvectorarray.VectorInterface

Bases: pymor.core.interfaces.BasicInterface

Interface for vectors.

This Interface is mainly intended to be used in conjunction with ListVectorArray. In general, all pyMOR objects operate on VectorArrays instead of single vectors! All methods of the interface have a direct counterpart in the VectorArray interface.

Methods

VectorInterface: almost_equal, amax, axpy, components, copy, dot, l1_norm, l2_norm, make_zeros, scal, sup_norm
BasicInterface: attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with_, __setattr__

Attributes

VectorInterface: dim, subtype
BasicInterface: locked, logger, logging_disabled, name, uid, with_arguments

numpysolvers module  This module contains wrappers for linear solvers which operate directly on NumPy matrices.
**pymor.la.numpysolvers.apply_inverse** *(matrix, U, options=None)*

Solve linear equation system. Applies the inverse of *matrix* to the row vectors in *U*.

See *sparse_options* for documentation of all possible options for sparse matrices. This method is called by `pymor.core.NumpyMatrixOperator.apply_inverse` and usually should not be used directly.

**Parameters**

- **matrix** The NumPy matrix to invert.
- **U** 2-dimensional NumPy array containing as row vectors the right-hand sides of the linear equation systems to solve.
- **options** *invert_options* to use. (See *invert_options*.)

**Returns**

NumPy array of the solution vectors.

**pymor.la.numpysolvers.dense_options** *(default_solver='solve', default_least_squares_solver='least_squares_lstsq', least_squares_lstsq_rcond=-1.0)*

Returns *invert_options* (with default values) for dense NumPy matrices.

**Parameters**

- **default_solver** Default dense solver to use (solve, least_squares_lstsq, generic_lgmres, least_squares_generic_lsmr, least_squares_generic_lsqr).
- **default_least_squares_solver** Default solver to use for least squares problems (least_squares_lstsq, least_squares_generic_lsmr, least_squares_generic_lsqr).
- **least_squares_lstsq_rcond** See `numpy.linalg.lstsq`.

**Returns**

A tuple of all possible *invert_options*.

**Defaults**

default_solver, default_least_squares_solver, least_squares_lstsq_rcond (see `pymor.core.defaults`)

**pymor.la.numpysolvers.invert_options** *(matrix=None, sparse=None)*

Returns *invert_options* (with default values) for a given NumPy matrix. See *sparse_options* for documentation of all possible options for sparse matrices.

**Parameters**

- **matrix** The matrix for which to return the options.
- **sparse** Instead of providing a matrix via the *matrix* argument, *sparse* can be set to `True` or `False` to request the invert options for sparse or dense matrices.
Returns
A tuple of all possible invert options.
Returns `invert_options` (with default values) for sparse NumPy matrices.

### Parameters

**default_solver**  Default sparse solver to use (spsolve, bicgstab, bicgstab_spilu, pyamg, pyamg_rs, pyamg_sa, generic_lgmres, least_squares_generic_lsmr, least_squares_generic_lsqr).
**default_least_squares_solver**  Default solver to use for least squares problems (least_squares_generic_lsmr, least_squares_generic_lsqr).

bicgstab_tol  See scipy.sparse.linalg.bicgstab.
bicgstab_maxiter  See scipy.sparse.linalg.bicgstab.
splu_drop_tol  See scipy.sparse.linalg.spilu.
splu_fill_factor  See scipy.sparse.linalg.spilu.
splu_drop_rule  See scipy.sparse.linalg.spilu.
splu_permc_spec  See scipy.sparse.linalg.spilu.
spsolve_permc_spec  See scipy.sparse.linalg.spsolve.
lgmres_tol  See scipy.sparse.linalg.lgmres.
lgmres_maxiter  See scipy.sparse.linalg.lgmres.
lgmres_inner_m  See scipy.sparse.linalg.lgmres.
lgmres_outer_k  See scipy.sparse.linalg.lgmres.
least_squares_lsmr_damp  See scipy.sparse.linalg.lsmr.
least_squares_lsmr_atol  See scipy.sparse.linalg.lsmr.
least_squares_lsmr_btol  See scipy.sparse.linalg.lsmr.
least_squares_lsmr_conlim  See scipy.sparse.linalg.lsmr.
least_squares_lsmr_maxiter  See scipy.sparse.linalg.lsmr.
least_squares_lsmr_show  See scipy.sparse.linalg.lsmr.
least_squares_lsqr_damp  See scipy.sparse.linalg.lsqr.
least_squares_lsqr_atol  See scipy.sparse.linalg.lsqr.
least_squares_lsqr_btol  See scipy.sparse.linalg.lsqr.
least_squares_lsqr_conlim  See scipy.sparse.linalg.lsqr.
least_squares_lsqr_iter_lim  See scipy.sparse.linalg.lsqr.
least_squares_lsqr_show  See scipy.sparse.linalg.lsqr.
pyamg_tol  Tolerance for PyAMG blackbox solver.
pyamg_maxiter  Maximum iterations for PyAMG blackbox solver.
pyamg_verb  Verbosity flag for PyAMG blackbox solver.
pyamg_rs_strength  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_CF  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_presmoother  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_postsmoother  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_max_levels  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_max_coarse  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_coarse_solver  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_cycle  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_accel  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_tol  Parameter for PyAMG Ruge-Stuben solver.
pyamg_rs_maxiter  Parameter for PyAMG Ruge-Stuben solver.
pyamg_sa_symmetry  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_strength  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_aggregate  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_smooth  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_presmoother  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_postsmoother  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_improve_candidates  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_max_levels  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_diagonal_dominance  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_coarse_solver  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_cycle  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_accel  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_tol  Parameter for PyAMG Smoothed-Aggregation solver.
pyamg_sa_maxiter  Parameter for PyAMG Smoothed-Aggregation solver.

Returns
A tuple of all possible invert options.

Defaults
default_solver, default_least_squares_solver, bicgstab_tol, bicgstab_maxiter, spilu_drop_tol, 
spilu_fill_factor, spilu_drop_rule, spilu_permc_spec, spsolve_permc_spec, spsolve_keep_factorization, 
lgmres_tol, lgmres_maxiter, lgmres_inner_m, lgmres_outer_k, least_squares_lsmr_damp, 
least_squares_lsmr_atol, least_squares_lsmr_btol, least_squares_lsmr_conlim, least_squares_lsmr_maxiter, 
least_squares_lsmr_show, least_squares_lspar_atol, least_squares_lspar_btol, least_squares_lspar_conlim, 
least_squares_lspar_iterlim, least_squares_lspar_show, pyamg_tol, pyamg_maxiter, pyamg_accel, 
pyamg_rs_strength, pyamg_rs_CF, pyamg_rs_postsmoother, pyamg_rs_max_levels, pyamg_rs_max_coarse, 
pyamg_rs_coarse_solver, pyamg_rs_cycle, pyamg_rs_accel, pyamg_rs_tol, pyamg_rs_maxiter, 
pyamg_sa_symmetry, pyamg_sa_strength, pyamg_sa_aggregate, pyamg_sa_smooth, pyamg_sa_presmoother, 
pyamg_sa_postsmoother, pyamg_sa_improve_candidates, pyamg_sa_max_levels, pyamg_sa_max_coarse, 
pyamg_sa_diagonal_dominance, pyamg_sa_coarse_solver, pyamg_sa_cycle, pyamg_sa_accel, pyamg_sa_tol, 
pyamg_sa_maxiter (see pymor.core.defaults)

numpyvectorarray module
class pymor.la.numpyvectorarray.NumpyVectorArray(instance, dtype=None, copy=False, or- 
der=None, subok=False)
Bases: pymor.la.interfaces.VectorArrayInterface
VectorArray implementation via NumPy arrays.

This is the default VectorArray type used by all Operators implemented directly in pyMOR. Reduced Operators will also expect NumpyVectorArrays.

Note that this class is just a thin wrapper around the underlying NumPy array. Thus, while operations like axpy or VectorArrayInterface.dot will be quite efficient, removing or appending vectors will be costly.

Methods

NumpyVectorArray: almost_equal, amax, append, axpy, components, copy, dot, l1_norm, l2_norm, lincomb, make_array, remove, replace, scal, __len__, __repr__, __str__

VectorArrayInterface: check_ind, check_ind_unique, empty, gramian, len_ind, len_ind_unique, sup_norm, zeros, __add__, __iadd__, __imul__, __isub__, __mul__, __neg__, __radd__, __rsub__, __rtruediv__

BasicInterface: add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with_, __setattr__

Attributes

NumpyVectorArray: data, dim, subtype
VectorArrayInterface: space
BasicInterface: locked, logger, logging_disabled, name, uid, with_arguments

__len__() - The number of vectors in the array.
__repr__() <=> repr(x)
__str__() <=> str(x)

almost_equal (other, ind=None, o_ind=None, rtol=None, atol=None) - Check vectors for equality.

Equality of two vectors should be defined as in pymor.tools.float_cmp_all.

The subtypes of self and other have to agree. If the length of self (ind) resp. other (o_ind) is 1, the one specified vector is compared to all vectors of the other summand. Otherwise the length of ind and o_ind have to agree.

Parameters

other A VectorArray containing the vectors to compare with.
ind Indices of the vectors that are to be compared (see class documentation).
o_ind Indices of the vectors in other that are to be compared (see class documentation).
rtol See pymor.tools.float_cmp_all
atol See pymor.tools.float_cmp_all

Returns

NumPy array of the truth values of the comparison.
amax \((\text{ind} = \text{None})\)

The maximum absolute value of the vectors contained in the array.

**Parameters**

- **ind** Indices of the vectors whose maximum absolute value is to be calculated (see class documentation).

**Returns**

- **max_ind** NumPy array containing for each vector an index at which the maximum is attained.
- **max_val** NumPy array containing for each vector the maximum absolute value of its components.

append \((\text{other}, o\_\text{ind} = \text{None}, \text{remove}\_\text{from}\_\text{other} = \text{False})\)

Append vectors to the array.

**Parameters**

- **other** A VectorArray containing the vectors to be appended.
- **o_ind** Indices of the vectors that are to be appended (see class documentation).
- **remove_from_other** If True, the appended vectors are removed from other. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.

axpy \((\alpha, x, \text{ind} = \text{None}, x\_\text{ind} = \text{None})\)

BLAS AXPY operation.

This method forms the sum

\[
\text{self}[\text{ind}] = \alpha \times x[\text{x\_ind}] + \text{self}[\text{ind}]
\]

The subtypes of self and x have to agree. If the length of x (x\_ind) is 1, the same x vector is used for all vectors in self. Otherwise, the lengths of self (ind) and x (x\_ind) have to agree. If alpha is a scalar, each x vector is multiplied with the same factor alpha. Otherwise, alpha has to be a one-dimensional NumPy array of the same length as self (ind) containing the coefficients for each x vector.

**Parameters**

- **alpha** The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in x are multiplied.
- **x** A VectorArray containing the x-summands.
- **ind** Indices of the vectors of self that are to be added (see class documentation). Repeated indices are forbidden.
- **x\_ind** Indices of the vectors in x that are to be added (see class documentation). Repeated indices are allowed.

components \((\text{component\_indices}, \text{ind} = \text{None})\)

Extract components of the vectors contained in the array.

**Parameters**

- **component\_indices** List or 1D NumPy array of indices of the vector components that are to be returned.
- **ind** Indices of the vectors whose components are to be calculated (see class documentation).
Returns

A NumPy array result such that result[i, j] is the component_indices[j]-th component of the ind[i]-th vector of the array.

copy (ind=None)

Returns a copy of a subarray.

Parameters

ind Indices of the vectors that are to be copied (see class documentation).

Returns

A copy of the VectorArray.

dot (other, pairwise, ind=None, o_ind=None)

Returns the scalar products between VectorArray elements.

Parameters

other A VectorArray containing the second factors.
pairwise See return value documentation.
ind Indices of the vectors whose scalar products are to be taken (see class documentation).
o_ind Indices of the vectors in other whose scalar products are to be taken (see class documentation).

Returns

If pairwise is True, returns a NumPy array result such that :

result[i] = ( self[ind][i], other[o_ind][i] ).

If pairwise is False, returns a NumPy array result such that :

result[i, j] = ( self[ind][i], other[o_ind][j] ).

l1_norm (ind=None)

The l1-norms of the vectors contained in the array.

Parameters

ind Indices of the vectors whose norms are to be calculated (see class documentation).

Returns

A NumPy array result such that result[i] contains the norm of self[ind][i].

l2_norm (ind=None)

The l2-norms of the vectors contained in the array.

Parameters

ind Indices of the vectors whose norms are to be calculated (see class documentation).
Returns
A NumPy array result such that result[i] contains the norm of self[ind][i].

`lincomb(coefficients, ind=None)`
Returns linear combinations of the vectors contained in the array.

Parameters
coefficients A NumPy array of dimension 1 or 2 containing the linear coefficients. coefficients.shape[-1] has to agree with len(self).

ind Indice of the vectors which are linear combined (see class documentation).

Returns
A VectorArray result such that:

result[i] = self[j] * coefficients[i,j]
in case coefficients is of dimension 2, otherwise len(result) == 1 and

`remove(ind=None)`
Remove vectors from the array.

Parameters
ind Indices of the vectors that are to be removed (see class documentation).

`replace(other, ind=None, o_ind=None, remove_from_other=False)`
Replace vectors of the array.

Parameters
other A VectorArray containing the replacement vectors.

ind Indices of the vectors that are to be replaced (see class documentation). Repeated indices are forbidden.

o_ind Indices of the replacement vectors (see class documentation). len(ind) has to agree with len(o_ind). Repeated indices are allowed.

remove_from_other If True, the new vectors are removed from other. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.

`scal(alpha, ind=None)`
BLAS SCAL operation (in-place scalar multiplication).

This method calculates

self[ind] = alpha*self[ind]

If alpha is a scalar, each vector is multiplied by this scalar. Otherwise, alpha has to be a one-dimensional NumPy array of the same length as self (ind) containing the factors for each vector.
**alpha** The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in `self` are multiplied.

**ind** Indices of the vectors of `self` that are to be scaled (see class documentation). Repeated indices are forbidden.

```python
pymor.la.numpyvectorarray.NumpyVectorSpace(dim)
```

Shorthand for `VectorSpace(NumpyVectorArray, dim)`.

**pod module**

```python
pymor.la.pod.pod(A, modes=None, product=None, tol=4e-08, symmetrize=False, orthonormalize=True, check=True, check_tol=1e-10)
```

Proper orthogonal decomposition of `A`.

If the `VectorArray A` is viewed as a linear map

```latex
A: \mathbb{R}^{\text{len}(A)} \longrightarrow \mathbb{R}^{\text{dim}(A)}
```

then the return value of this method is simply the `VectorArray` of left-singular vectors of the singular value decomposition of `A` with the scalar product on \( \mathbb{R}^{\text{dim}(A)} \) given by `product` and the scalar product on \( \mathbb{R}^{\text{len}(A)} \) being the Euclidean product.

**Parameters**

- **A** The `VectorArray` for which the POD is to be computed.
- **modes** If not `None` only the first `modes` POD modes (singular vectors) are returned.
- **products** Scalar product `Operator` w.r.t. which the POD is computed.
- **tol** Singular values smaller than this value multiplied by the largest singular value are ignored.
- **symmetrize** If `True`, symmetrize the gramian again before proceeding.
- **orthonormalize** If `True`, orthonormalize the computed POD modes again using `la.gram_schmidt.gram_schmidt`.
- **check** If `True`, check the computed POD modes for orthonormality.
- **check_tol** Tolerance for the orthonormality check.

**Returns**

- **POD** `VectorArray` of POD modes.
- **SVALS** Sequence of singular values.

**Defaults**

- `tol`, `symmetrize`, `orthonormalize`, `check`, `check_tol` (see `pymor.core.defaults`)
pymor.operators package

Submodules

**basic module**

**class** pymor.operators.basic.OperatorBase

**Bases**: pymor.operators.interfaces.OperatorInterface

Base class for Operators providing some default implementations.

When implementing a new operator, it is usually advisable to derive from this class.

**Methods**

- apply2, apply_adjoint, apply_inverse, as_vector, assemble, jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__
- apply, assemble_lincomb, restricted
- lock
- add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__
- build_parameter_type, local_parameter, parse_parameter, strip_parameter

**Attributes**

<table>
<thead>
<tr>
<th>OperatorBase</th>
<th>invert_options</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>linear, range, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**__add__(other)**
Sum of two operators

**__mul__(other)**
Product of operator by a scalar

**__radd__(other)**
Sum of two operators

**__str__() \leftrightarrow \text{str}(x)**

**apply2(V, U, pairwise, U_ind=None, V_ind=None, mu=None, product=None)**
Treat the operator as a 2-form by calculating $(V, A(U))$.

In particular, if $(\cdot, \cdot)$ is the Euclidean product and $A$ is a linear operator given by multiplication with a matrix $M$, then

$$A\cdot \text{apply2}(V, U) = V^T \cdot M \cdot U$$

**Parameters**

- $V$ VectorArray of the left arguments $V$.
- $U$ VectorArray of the right arguments $U$. 
pairwise  If False, the 2-form is applied to all combinations of vectors in V and U, i.e.

\[ L.apply2(V, U).shape = (\text{len(V\_ind)}, \text{len(U\_ind)}). \]

If True, the vectors in V and U are applied in pairs, i.e. V and U must be of the same length and we have

\[ L.apply2(V, U).shape = (\text{len(V\_ind)},) = (\text{len(U\_ind)},). \]

V\_ind  The indices of the vectors in V to which the operator shall be applied. (See the VectorArray documentation for further details.)

U\_ind  The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)

mu  The Parameter for which to evaluate the operator.

product  The scalar product used in the expression \((V, A(U))\) given as an Operator. If None, the euclidean product is chosen.

Returns

A NumPy array of all 2-form evaluations.

apply_adjoint  \((U, \text{ind=none}, \text{mu=none}, \text{source\_product=none}, \text{range\_product=none})\)

Apply the adjoint operator.

For a linear operator A the adjoint A^* of A is given by

\[ (A^*v, u)_s = (v, Au)_r \]

where \((,)_s\) and \((,)_r\) denote the scalar products on the source and range space of A. If A and the two products are given by the matrices M, P_s and P_r, then:

\[ A^*v = P_s^{-1} * M^T * P_r * v \]

with M^T denoting the transposed of M. Thus, if \((,)_s\) and \((,)_r\) are the euclidean products, A^*v is simply given by multiplication of the matrix of A with v from the left.

Parameters

U  VectorArray of vectors to which the adjoint operator is applied.

ind  The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)

mu  The Parameter for which to apply the adjoint operator.

source\_product  The scalar product Operator on the source space. If None, the euclidean product is chosen.

range\_product  The scalar product Operator on the range space. If None, the euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.
**apply_inverse** *(U, ind=None, mu=None, options=None)*

Apply the inverse operator.

**Parameters**
- **U** `VectorArray` of vectors to which the inverse operator is applied.
- **ind** The indices of the vectors in `U` to which the operator shall be applied. (See the `VectorArray` documentation for further details.)
- **mu** The `Parameter` for which to evaluate the inverse operator.
- **options** Dictionary of options for the inversion algorithm. The dictionary has to contain the key `type` whose value determines which inversion algorithm is to be used. All other items represent options specific to this algorithm. `options` can also be given as a string, which is then interpreted as the type of inversion algorithm. If `options` is `None`, a default algorithm with default options is chosen. Available algorithms and their default options are provided by `invert_options`.

**Returns**
- `VectorArray` of the inverse operator evaluations.

**Raises**
- **InversionError** The operator could not be inverted.

**as_vector** *(mu=None)*

Return a vector representation of a linear functional or vector operator.

This method may only be called on linear functionals, i.e. linear `Operators` with `NumpyVectorSpace(1)` as range, or on operators describing vectors, i.e. linear `Operators` with `NumpyVectorSpace(1)` as source.

In the case of a functional, the identity

```
self.as_vector(mu).dot(U) == self.apply(U, mu)
```

holds, whereas in the case of a vector-like operator we have

```
self.as_vector(mu) == self.apply(NumpyVectorArray(1), mu).
```

**Parameters**
- **mu** The `Parameter` for which to return the vector representation.

**Returns**
- `V` `VectorArray` of length 1 containing the vector representation. `V` belongs to `self.source` for functionals and to `self.range` for vector-like operators.

**assemble** *(mu=None)*

Assemble the operator for a given parameter.

What the result of the assembly is strongly depends on the given operator. For instance, a matrix-based operator will assemble its matrix, a `LincombOperator` will try to form the linear combination of its operators, whereas an arbitrary operator might simply return a `FixedParameterOperator`. The only assured property of the assembled operator is that it no longer depends on a `Parameter`. 
Parameters
mu The Parameter for which to assemble the operator.

Returns
Parameter-independent, assembled Operator.

jacobian(U, mu=None)
Return the operator’s Jacobian.

Parameters
U Length 1 VectorArray containing the vector for which to compute the Jacobian.
mu The Parameter for which to compute the Jacobian.

Returns
Operator representing the Jacobian.

projected(source_basis, range_basis, product=None, name=None)
Project the operator to subspaces of the source and range space.

Denote self by A. Given a scalar product ( , ), and vectors b_1, ..., b_N, c_1, ..., c_M, the projected operator A_P is defined by

\[ [ A_P(e_j) ]_i = ( c_i, A(b_j) ) \]

for all i,j, where e_j denotes the j-th canonical basis vector of R^N.

In particular, if the c_i are orthonormal w.r.t. the given product, then A_P is the coordinate representation w.r.t. the b_i/c_i bases of the restriction of A to span(b_i) concatenated with the orthogonal projection onto span(c_i).

From another point of view, if A is viewed as a bilinear form (see apply2) and ( , ) is the Euclidean product, then A_P represents the matrix of the bilinear form restricted span(b_i) / span(c_i) (w.r.t. the b_i/c_i bases).

How the projected operator is realized will depend on the implementation of the operator to project. While a projected NumpyMatrixOperator will again be a NumpyMatrixOperator, only a generic pymor.operators.basic.ProjectedOperator will be returned in general. (Note that the latter will not be suitable to obtain an efficient offline/online-decomposition for reduced basis schemes.)

A default implementation is provided in OperatorBase.

**Warning:** No check is performed whether the b_i and c_j are orthonormal or linear independent.

Parameters
source_basis The b_1, ..., b_N as a VectorArray or None. If None, no restriction of the source space is performed.
range_basis The c_1, ..., c_M as a VectorArray. If None, no projection in the range space is performed.
product An Operator representing the scalar product. If None, the Euclidean product is chosen.
name Name of the projected operator.
Returns
The projected Operator.

class pymor.operators.basic.ProjectedOperator(operator, source_basis, range_basis, product=None, copy=True, name=None)

Bases: pymor.operators.basic.OperatorBase

Generic Operator for representing the projection of an Operator to a subspace.

This class is not intended to be instantiated directly. Instead, you should use the projected method of the given Operator.

Parameters
operator The Operator to project.
source_basis See projected.
range_basis See projected.
product See projected.
copy If True, make a copy of the provided source_basis and range_basis. This is usually necessary, as VectorArrays are not immutable.
name Name of the projected operator.

Methods
apply, assemble, jacobian, projected_to_subbasis
OperatorBase apply2, apply_adjoint, apply_inverse, as_vector, lincomb, projected, __add__, __mul__, __radd__, __str__
OperatorInterface assemble_lincomb, restricted
ImmutableInterface __add__ attributes, disable_logging, enable_logging,
BasicInterface has_interface_name, implementor_names, implementors, with_, _setattr_
Parametric build_parameter_type, local_parameter, parse_parameter,

Attributes
ProjectedOperator linear
OperatorBase invert_options
OperatorInterface range, source
ImmutableInterface calculate_sid, sid, sid_failure, sid_ignore
BasicInterface locked, logger, logging_disabled, name, uid, with_arguments
Parametric parameter_local_type, parameter_space, parameter_type, parametric

apply(U, ind=None, mu=None)
Apply the operator.

Parameters
U VectorArray of vectors to which the operator is applied.
The indices of the vectors in $U$ to which the operator shall be applied. (See the `VectorArray` documentation for further details.)

- **mu**: The `Parameter` for which to evaluate the operator.

Returns
`VectorArray` of the operator evaluations.

**assemble** *(mu=None)*
Assemble the operator for a given parameter.

What the result of the assembly is strongly depends on the given operator. For instance, a matrix-based operator will assemble its matrix, a `LincombOperator` will try to form the linear combination of its operators, whereas an arbitrary operator might simply return a `FixedParameterOperator`. The only assured property of the assembled operator is that it no longer depends on a `Parameter`.

**Parameters**
- **mu**: The `Parameter` for which to assemble the operator.

**Returns**
Parameter-independent, assembled `Operator`.

**jacobian** *(U, mu=None)*
Return the operator's Jacobian.

**Parameters**
- **U**: Length 1 `VectorArray` containing the vector for which to compute the Jacobian.
- **mu**: The `Parameter` for which to compute the Jacobian.

**Returns**
`Operator` representing the Jacobian.

**projected_to_subbasis** *(dim_source=None, dim_range=None, name=None)*
See `NumpyMatrixOperator.projected_to_subbasis`.

**cg module**
This module provides some operators for continuous finite element discretizations.

**class** `pymor.operators.cg.DiffusionOperatorP1` *(grid, boundary_info, diffusion_function=None, diffusion_constant=None, dirichlet_clear_columns=False, dirichlet_clear_diag=False, name=None)*
`Bases: pymor.operators.numpy.NumpyMatrixBasedOperator`

Diffusion `Operator` for linear finite elements.

The operator is of the form

$$(Lu)(x) = c \begin{bmatrix} d(x) & u(x) \end{bmatrix}$$
The function \( d \) can be scalar- or matrix-valued. The current implementation works in one and two dimensions, but can be trivially extended to arbitrary dimensions.

**Parameters**

- **grid** The `Grid` for which to assemble the operator.
- **boundary_info** `BoundaryInfo` for the treatment of Dirichlet boundary conditions.
- **diffusion_function** The `Function` \( d(x) \) with `shape_range == tuple()` or `shape_range = (grid.dim_outer, grid.dim_outer)`. If `None`, constant one is assumed.
- **diffusion_constant** The constant \( c \). If `None`, \( c \) is set to one.
- **dirichlet_clear_columns** If `True`, set columns of the system matrix corresponding to Dirichlet boundary DOFs to zero to obtain a symmetric system matrix. Otherwise, only the rows will be set to zero.
- **dirichlet_clear_diag** If `True`, also set diagonal entries corresponding to Dirichlet boundary DOFs to zero (e.g. for affine decomposition). Otherwise they are set to one.
- **name** Name of the operator.

**Methods**

- `apply`, `apply_adjoint`, `apply_inverse`, `as_vector`, `assemble`, `export_matrix`
- `apply2`, `jacobian`, `lincomb`, `projected`, `__add__`, `__mul__`, `__radd__`, `__str__`
- `assemble`, `assemble_lincomb`, `restricted`
- `apply2`, `jacobian`, `lincomb`, `projected`, `__add__`, `__mul__`, `__radd__`, `__str__`
- `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`
- `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

**Attributes**

- `sparse`
- `invert_options`, `linear`
- `range`, `source`
- `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`
- `parameter_local_type`, `parameter_space`, `parameter_type`, `parameter_type`

```python
class pymor.operators.cg.DiffusionOperatorQ1(  
    grid,  
    boundary_info,  
    diffusion_function=None,  
    diffusion_constant=None,  
    dirichlet_clear_columns=False,  
    dirichlet_clear_diag=False,  
    name=None  
)
```

Bases: `pymor.operators.numpy.NumpyMatrixBasedOperator`

Diffusion `Operator` for bilinear finite elements.

The operator is of the form
The function $d$ can be scalar- or matrix-valued. The current implementation works in two dimensions, but can be trivially extended to arbitrary dimensions.

**Parameters**

- **grid** The `Grid` for which to assemble the operator.
- **boundary_info** `BoundaryInfo` for the treatment of Dirichlet boundary conditions.
- **diffusion_function** The `Function` $d(x)$ with shape_range == tuple() or shape_range == (grid.dim_outer, grid.dim_outer). If `None`, constant one is assumed.
- **diffusion_constant** The constant $c$. If `None`, $c$ is set to one.
- **dirichlet_clear_columns** If `True`, set columns of the system matrix corresponding to Dirichlet boundary DOFs to zero to obtain a symmetric system matrix. Otherwise, only the rows will be set to zero.
- **dirichlet_clear_diag** If `True`, also set diagonal entries corresponding to Dirichlet boundary DOFs to zero (e.g. for affine decomposition). Otherwise they are set to one.
- **name** Name of the operator.

**Methods**

<table>
<thead>
<tr>
<th>NumpyMatrixBasedOperator</th>
<th>apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, jacobian, lincomb, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>assemble_lincomb, restricted</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><strong>str</strong></td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with__, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>DiffusionOperatorQ1</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>invert_options, linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**Class** `pymor.operators.cg.InterpolationOperator` *(grid, function)*

Bases: `pymor.operators.numpy.NumpyMatrixBasedOperator`

Lagrange interpolation operator for continuous finite element spaces.

**Parameters**

- **grid** The `Grid` on which to interpolate.
**function** The *Function* to interpolate.

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td><code>apply</code>, <code>apply_adjoint</code>, <code>apply_inverse</code>, <code>as_vector</code>, <code>assemble</code>, <code>export_matrix</code></td>
</tr>
<tr>
<td>OperatorBase</td>
<td><code>apply2</code>, <code>jacobian</code>, <code>lincomb</code>, <code>projected</code>, <code>__add__</code>, <code>__mul__</code>, <code>__radd__</code>, <code>__str__</code></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td><code>assemble</code>, <code>assemble_lincomb</code>, <code>restricted</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>__eq__</code>, <code>__ne__</code>, <code>__hash__</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>add_attributes</code>, <code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>with__</code>, <code>__setattr__</code></td>
</tr>
<tr>
<td>Parametric</td>
<td><code>build_parameter_type</code>, <code>local_parameter</code>, <code>parse_parameter</code>, <code>strip_parameter</code></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterpolationOperator</td>
<td><code>linear</code>, <code>source</code></td>
</tr>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td><code>invert_options</code>, <code>sparse</code></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td><code>range</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>calculate_sid</code>, <code>sid</code>, <code>sid_failure</code>, <code>sid_ignore</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>locked</code>, <code>logger</code>, <code>logging_disabled</code>, <code>name</code>, <code>uid</code>, <code>with_arguments</code></td>
</tr>
<tr>
<td>Parametric</td>
<td><code>parameter_local_type</code>, <code>parameter_space</code>, <code>parameter_type</code>, <code>parameter_type</code></td>
</tr>
</tbody>
</table>

#### class pymor.operators.cg.L2ProductFunctionalP1(grid, function, boundary_info=None, dirichlet_data=None, neumann_data=None, order=2, name=None)

*Functional* representing the scalar product with an *L2-Function* for linear finite elements.

Boundary treatment can be performed by providing `boundary_info` and `dirichlet_data`, in which case the DOFs corresponding to Dirichlet boundaries are set to the values provided by `dirichlet_data`. Neumann boundaries are handled by providing a `neumann_data` function.

The current implementation works in one and two dimensions, but can be trivially extended to arbitrary dimensions.

#### Parameters

- **grid** `Grid` for which to assemble the functional.
- **function** The *Function* with which to take the scalar product.
- **boundary_info** `BoundaryInfo` determining the Dirichlet and Neumann boundaries or `None`. If `None`, no boundary treatment is performed.
- **dirichlet_data** `Function` providing the Dirichlet boundary values. If `None`, constant-zero boundary is assumed.
- **neumann_data** `Function` providing the Neumann boundary values. If `None`, constant-zero is assumed.
- **order** Order of the Gauss quadrature to use for numerical integration.
- **name** The name of the functional.
Methods

```
NumpyMatrixBasedOperator
  apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix
OperatorBase
  apply2, jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__
OperatorInterface
  assemble_lincomb, restricted
ImmutableInterface
  __add__, __mul__, __radd__, __str__
BasicInterface
  add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__
Parametric
  build_parameter_type, local_parameter, parse_parameter, strip_parameter
```

Attributes

```
L2ProductFunctionalP1
  range, sparse
NumpyMatrixBasedOperator
  invert_options, linear
OperatorInterface
  source
ImmutableInterface
  calculate_sid, sid, sid_failure, sid_ignore
BasicInterface
  locked, logger, logging_disabled, name, uid, with_arguments
Parametric
  parameter_local_type, parameter_space, parameter_type, parametric
```

class pymor.operators.cg.L2ProductFunctionalQ1 (grid, function, boundary_info=None, dirichlet_data=None, neumann_data=None, order=2, name=None)

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator

Functional representing the scalar product with an L2-Function for bilinear finite elements.

Boundary treatment can be performed by providing boundary_info and dirichlet_data, in which case the DOFs corresponding to Dirichlet boundaries are set to the values provided by dirichlet_data.

The current implementation works in two dimensions, but can be trivially extended to arbitrary dimensions.

Parameters

- **grid**  *Grid* for which to assemble the functional.
- **function**  *Function* with which to take the scalar product.
- **boundary_info**  *BoundaryInfo* determining the Dirichlet boundaries or *None*. If *None*, no boundary treatment is performed.
- **dirichlet_data**  *Function* providing the Dirichlet boundary values. If *None*, constant-zero boundary is assumed.
- **neumann_data**  *Function* providing the Neumann boundary values. If *None*, constant-zero is assumed.
- **order**  Order of the Gauss quadrature to use for numerical integration.
- **name**  The name of the functional.

Methods
class pymor.operators.cg.L2ProductP1(grid, boundary_info, dirichlet_clear_rows=True, dirichlet_clear_columns=False, dirichlet_clear_diag=False, name=None)

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator

Operator representing the L2-product between linear finite element functions.

To evaluate the product use the apply2 method.

The current implementation works in one and two dimensions, but can be trivially extended to arbitrary dimensions.

Parameters

grid  The Grid for which to assemble the product.

boundary_info  BoundaryInfo for the treatment of Dirichlet boundary conditions.

dirichlet_clear_rows  If True, set the rows of the system matrix corresponding to Dirichlet boundary DOFs to zero. (Useful when used as mass matrix in time-stepping schemes.)

dirichlet_clear_columns  If True, set columns of the system matrix corresponding to Dirichlet boundary DOFs to zero (to obtain a symmetric matrix).

dirichlet_clear_diag  If True, also set diagonal entries corresponding to Dirichlet boundary DOFs to zero (e.g. for affine decomposition). Otherwise, if either dirichlet_clear_rows or dirichlet_clear_columns is True, the diagonal entries are set to one.

name  The name of the product.

Methods
### NumpyMatrixBasedOperator

- `apply`, `apply_adjoint`, `apply_inverse`, `as_vector`, `assemble`, `export_matrix`

### OperatorBase

- `apply2`, `jacobian`, `lincomb`, `projected`, `__add__`, `__mul__`, `__radd__`, `__str__`

### OperatorInterface

- `assemble_lincomb`, `restricted`

### ImmutableInterface

- `lock`, `unlock`

### BasicInterface

- `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with__`, `__setattr__`

### Parametric

- `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

---

#### Attributes

<table>
<thead>
<tr>
<th>NumpyMatrixBasedOperator</th>
<th><code>sparse</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>invert_options</code>, <code>linear</code></td>
<td></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td><code>range</code>, <code>source</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>calculate_sid</code>, <code>sid</code>, <code>sid_failure</code>, <code>sid_ignore</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>locked</code>, <code>logger</code>, <code>logging_disabled</code>, <code>name</code>, <code>uid</code>, <code>with_arguments</code></td>
</tr>
<tr>
<td>Parametric</td>
<td><code>parameter_local_type</code>, <code>parameter_space</code>, <code>parameter_type</code>, <code>parametric</code></td>
</tr>
</tbody>
</table>

---

**class** `pymor.operators.cg.L2ProductQ1` (grid, boundary_info, dirichlet_clear_rows=True, dirichlet_clear_columns=False, dirichlet_clear_diag=False, name=None)

**Bases:** `pymor.operators.numpy.NumpyMatrixBasedOperator`

**Operator** representing the L2-product between bilinear finite element functions.

To evaluate the product use the `apply2` method.

The current implementation works in two dimensions, but can be trivially extended to arbitrary dimensions.

**Parameters**

- **grid** The `Grid` for which to assemble the product.
- **boundary_info** `BoundaryInfo` for the treatment of Dirichlet boundary conditions.
- **dirichlet_clear_rows** If `True`, set the rows of the system matrix corresponding to Dirichlet boundary DOFs to zero. (Useful when used as mass matrix in time-stepping schemes.)
- **dirichlet_clear_columns** If `True`, set columns of the system matrix corresponding to Dirichlet boundary DOFs to zero (to obtain a symmetric matrix).
- **dirichlet_clear_diag** If `True`, also set diagonal entries corresponding to Dirichlet boundary DOFs to zero (e.g. for affine decomposition). Otherwise, if either `dirichlet_clear_rows` or `dirichlet_clear_columns` is `True`, the diagonal entries are set to one.
- **name** The name of the product.

---

**Methods**

---

4.1. `pymor` package
constructions module  Module containing some constructions to obtain new operators from old ones.

## class pymor.operators.constructions.ComponentProjection

**Source:** `pymor.operators.constructions.ComponentProjection(components, source, name=None)`

**Operator:** representing the projection of a Vector on some of its components.

### Parameters

- **components** List or 1D NumPy array of the indices of the vector components that are to be extracted by the operator.
- **source** Source VectorSpace of the operator.
- **name** Name of the operator.

### Methods

- **ComponentProjection**
  - `apply`, `apply_adjoint`, `apply_inverse`, `as_vector`, `assemble`, `export_matrix`

- **OperatorBase**
  - `apply2`, `jacobian`, `lincomb`, `projected`, `__add__`, `__mul__`, `__radd__`, `__str__`

- **OperatorInterface**
  - `assemble_lincomb`, `restricted`

- **ImmutableInterface**
  - `assemble_lincomb`, `restricted`

- **BasicInterface**
  - `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`

- **Parametric**
  - `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

### Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L2ProductQ1</code></td>
</tr>
<tr>
<td><code>NumpyMatrixBasedOperator</code></td>
</tr>
<tr>
<td><code>OperatorInterface</code></td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
</tr>
<tr>
<td><code>Parametric</code></td>
</tr>
</tbody>
</table>
**ComponentProjection**

**linear**

**OperatorBase**

invert_options

**OperatorInterface**

range, source

**ImmutableInterface**

calculate_sid, sid, sid_failure, sid_ignore

**BasicInterface**

locked, logger, logging_disabled, name, uid, with_arguments

**Parametric**

parameter_local_type, parameter_space, parameter_type, parametric

### apply \((U, \text{ind}=None, \text{mu}=None)\)

Apply the operator.

**Parameters**

- **U** VectorArray of vectors to which the operator is applied.
- **ind** The indices of the vectors in \(U\) to which the operator shall be applied. (See the VectorArray documentation for further details.)
- **mu** The Parameter for which to evaluate the operator.

**Returns**

VectorArray of the operator evaluations.

### restricted \((dofs)\)

Restrict the operator range to a given set of degrees of freedom.

This method returns a restricted version \(\text{restricted}\_\text{op}\) of the operator along with an array \(\text{source}\_\text{dofs}\) such that for any VectorArray \(U\) in \(\text{self}\_\text{source}\) the following is true:

```python
self.apply(U, mu).components(dofs) == restricted_op.apply(NumpyVectorArray(U.components(source_dofs)), mu)
```

Such an operator is mainly useful for empirical interpolation where the evaluation of the original operator only needs to be known for few selected degrees of freedom. If the operator has a small stencil, only few source_dofs will be needed to evaluate the restricted operator which can make its evaluation very fast compared to evaluating the original operator. Note that the interface does not make any assumptions on the efficiency of evaluating the restricted operator.

**Parameters**

- **dofs** One-dimensional NumPy array of degrees of freedom in the operator range to which to restrict.

**Returns**

- **restricted_op** The restricted operator as defined above. The operator will have NumpyVectorSpace \((\text{len(source}\_\text{dofs}))\) as source and NumpyVectorSpace \((\text{len(dofs)})\) as range.
- **source_dofs** One-dimensional NumPy array of source degrees of freedom as defined above.

```python
class pymor.operators.constructions.Concatenation(second, first, name=None)
Bases: pymor.operators.basic.OperatorBase

Operator representing the concatenation of two Operators.
```
second  The Operator which is applied as second operator.

first  The Operator which is applied as first operator.

name  Name of the operator.

---

Methods

<table>
<thead>
<tr>
<th>OperatorBase</th>
<th>apply, apply_adjoint, restricted</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_inverse, as_vector, assemble, jacobian, lincomb, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>assemble_lincomb</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>lock, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with__, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

---

Attributes

<table>
<thead>
<tr>
<th>OperatorBase</th>
<th>invert_options</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>linear, range, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

---

**apply** $(U, ind=None, mu=None)$

Apply the operator.

**Parameters**

$U$ VectorArray of vectors to which the operator is applied.

$ind$ The indices of the vectors in $U$ to which the operator shall be applied. (See the VectorArray documentation for further details.)

$mu$ The Parameter for which to evaluate the operator.

**Returns**

VectorArray of the operator evaluations.

---

**apply_adjoint** $(U, ind=None, mu=None, source_product=None, range_product=None)$

Apply the adjoint operator.

For a linear operator $A$ the adjoint $A^*$ of $A$ is given by

$$(A^*v, u)_s = (v, Au)_r$$

where $(,)_s$ and $(,)_r$ denote the scalar products on the source and range space of $A$. If $A$ and the two products are given by the matrices $M, P_s$ and $P_r$, then:

$$A^*v = P_s^*(-1) * M^T * P_r * v$$

with $M^T$ denoting the transposed of $M$. Thus, if $(,)_s$ and $(,)_r$ are the euclidean products, $A^*v$ is simply given by multiplication of the matrix of $A$ with $v$ from the left.
Parameters

U: VectorArray of vectors to which the adjoint operator is applied.

ind: The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)

mu: The Parameter for which to apply the adjoint operator.

source_product: The scalar product Operator on the source space. If None, the euclidean product is chosen.

range_product: The scalar product Operator on the range space. If None, the euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.

restricted(dofs)

Restrict the operator range to a given set of degrees of freedom.

This method returns a restricted version restricted_op of the operator along with an array source_dofs such that for any VectorArray U in self.source the following is true:

```
self.apply(U, mu).components(dofs) == restricted_op.apply(NumpyVectorArray(U.components(source_dofs)), mu)
```

Such an operator is mainly useful for empirical interpolation where the evaluation of the original operator only needs to be known for few selected degrees of freedom. If the operator has a small stencil, only few source_dofs will be needed to evaluate the restricted operator which can make its evaluation very fast compared to evaluating the original operator. Note that the interface does not make any assumptions on the efficiency of evaluating the restricted operator.

Parameters

dofs: One-dimensional NumPy array of degrees of freedom in the operator range to which to restrict.

Returns

restricted_op: The restricted operator as defined above. The operator will have NumpyVectorSpace (len(source_dofs)) as source and NumpyVectorSpace (len(dofs)) as range.

source_dofs: One-dimensional NumPy array of source degrees of freedom as defined above.

class pymor.operators.constructions.ConstantOperator(value, source, copy=True, name=None)

Bases: pymor.operators.basic.OperatorBase

A constant Operator always returning the same vector.

Parameters

value: A VectorArray of length 1 containing the vector which is returned by the operator.

source: Source VectorSpace of the operator.

copy: If True, store a copy of vector instead of vector itself.
**pyMOR, Release 0.3.2**

**name**  Name of the operator.

---

**Methods**

<table>
<thead>
<tr>
<th>ConstantOperator</th>
<th>apply, jacobian, projected</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply, apply_adjoint, apply_inverse, as_vector, assemble, lincomb, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>assemble_lincomb, restricted</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <em>setattr</em></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

---

**Attributes**

<table>
<thead>
<tr>
<th>ConstantOperator</th>
<th>linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>invert_options</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

---

**apply** *(U, ind=None, mu=None)*

Apply the operator.

**Parameters**

- **U**  *VectorArray* of vectors to which the operator is applied.
- **ind**  The indices of the vectors in *U* to which the operator shall be applied. (See the *VectorArray* documentation for further details.)
- **mu**  The *Parameter* for which to evaluate the operator.

**Returns**

*VectorArray* of the operator evaluations.

---

**jacobian** *(U, mu=None)*

Return the operator’s Jacobian.

**Parameters**

- **U**  *Length 1 VectorArray* containing the vector for which to compute the Jacobian.
- **mu**  The *Parameter* for which to compute the Jacobian.

**Returns**

*Operator* representing the Jacobian.

---

**projected** *(source_basis, range_basis, product=None, name=None)*

Project the operator to subspaces of the source and range space.
Denote `self` by `A`. Given a scalar product `( , )`, and vectors `b_1, ..., b_N, c_1, ..., c_M`, the projected operator `A_P` is defined by

\[
[ A_P(e_j) ]_i = ( c_i, A(b_j) )
\]

for all `i,j`, where `e_j` denotes the `j`-th canonical basis vector of `R^N`.

In particular, if the `c_i` are orthonormal w.r.t. the given product, then `A_P` is the coordinate representation w.r.t. the `b_i/c_i` bases of the restriction of `A` to `span(b_i)` concatenated with the orthogonal projection onto `span(c_i)`.

From another point of view, if `A` is viewed as a bilinear form (see `apply2`) and `( , )` is the Euclidean product, then `A_P` represents the matrix of the bilinear form restricted `span(b_i) / span(c_i)` (w.r.t. the `b_i/c_i` bases).

How the projected operator is realized will depend on the implementation of the operator to project. While a projected `NumpyMatrixOperator` will again be a `NumpyMatrixOperator`, only a generic `pymor.operators.basic.ProjectedOperator` will be returned in general. (Note that the latter will not be suitable to obtain an efficient offline/online-decomposition for reduced basis schemes.)

A default implementation is provided in `OperatorBase`.

**Warning:** No check is performed whether the `b_i` and `c_j` are orthonormal or linear independent.

### Parameters

- **source_basis** The `b_1, ..., b_N` as a `VectorArray` or `None`. If `None`, no restriction of the source space is performed.
- **range_basis** The `c_1, ..., c_M` as a `VectorArray`. If `None`, no projection in the range space is performed.
- **product** An `Operator` representing the scalar product. If `None`, the Euclidean product is chosen.
- **name** Name of the projected operator.

### Returns

The projected `Operator`.

```python
class pymor.operators.constructions.FixedParameterOperator (operator, mu=None, name=None)
```

**Bases:** `pymor.operators.basic.OperatorBase`

Makes an `Operator Parameter`-independent by providing it a fixed `Parameter`.

### Parameters

- **operator** The `Operator` to wrap.
- **mu** The fixed `Parameter` that will be fed to the `apply` method of `operator`.

### Methods
**apply** \((U, \text{ind}=\text{None}, \text{mu}=\text{None})\)

Apply the operator.

**Parameters**

- **U** *VectorArray* of vectors to which the operator is applied.
- **\text{ind}** The indices of the vectors in \(U\) to which the operator shall be applied. (See the *VectorArray* documentation for further details.)
- **\text{mu}** The *Parameter* for which to evaluate the operator.

**Returns**

*VectorArray* of the operator evaluations.

**apply_adjoint** \((U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None})\)

Apply the adjoint operator.

For a linear operator \(A\) the adjoint \(A^*\) of \(A\) is given by

\[
(A^*v, u)_s = (v, Au)_r
\]

where \((,)_s\) and \((,)_r\) denote the scalar products on the source and range space of \(A\). If \(A\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[
A^*v = P_s^{-1} * M^T * P_r * v
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \((,)_s\) and \((,)_r\) are the euclidean products, \(A^*v\) is simply given by multiplication of the matrix of \(A\) with \(v\) from the left.

**Parameters**

- **U** *VectorArray* of vectors to which the adjoint operator is applied.
- **\text{ind}** The indices of the vectors in \(U\) to which the operator shall be applied. (See the *VectorArray* documentation for further details.)
mu  The Parameter for which to apply the adjoint operator.

source_product  The scalar product Operator on the source space. If None, the euclidean product is chosen.

rangeproduct  The scalar product Operator on the range space. If None, the euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.

apply_inverse(U, ind=None, mu=None, options=None)

Apply the inverse operator.

Parameters

U  VectorArray of vectors to which the inverse operator is applied.

ind  The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)

mu  The Parameter for which to evaluate the inverse operator.

options  Dictionary of options for the inversion algorithm. The dictionary has to contain the key ‘type’ whose value determines which inversion algorithm is to be used. All other items represent options specific to this algorithm. options can also be given as a string, which is then interpreted as the type of inversion algorithm. If options is None, a default algorithm with default options is chosen. Available algorithms and their default options are provided by invert_options.

Returns

VectorArray of the inverse operator evaluations.

Raises

InversionError  The operator could not be inverted.

class pymor.operators.constructions.IdentityOperator(space, name=None)

Bases: pymor.operators.basic.OperatorBase

The identity Operator.

In other word

op.apply(U) == U

Parameters

space  The VectorSpace the operator acts on.

name  Name of the operator.
apply \((U, \text{ind}=\text{None}, \mu=\text{None})\)

Apply the operator.

**Parameters**

- \(U\) `VectorArray` of vectors to which the operator is applied.
- \(\text{ind}\) The indices of the vectors in \(U\) to which the operator shall be applied. (See the `VectorArray` documentation for further details.)
- \(\mu\) The `Parameter` for which to evaluate the operator.

**Returns**

`VectorArray` of the operator evaluations.

apply_adjoint \((U, \text{ind}=\text{None}, \mu=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None})\)

Apply the adjoint operator.

For a linear operator \(A\) the adjoint \(A^*\) of \(A\) is given by

\[
(A^* v, u)_s = (v, A u)_r
\]

where \((.,.)_s\) and \((.,.)_r\) denote the scalar products on the source and range space of \(A\). If \(A\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[
A^* v = P_s^{-1} * M^T * P_r * v
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \((.,.)_s\) and \((.,.)_r\) are the euclidean products, \(A^* v\) is simply given by multiplication of the matrix of \(A\) with \(v\) from the left.

**Parameters**

- \(U\) `VectorArray` of vectors to which the adjoint operator is applied.
- \(\text{ind}\) The indices of the vectors in \(U\) to which the operator shall be applied. (See the `VectorArray` documentation for further details.)
mu  The Parameter for which to apply the adjoint operator.

source_product  The scalar product Operator on the source space. If None, the euclidean product is chosen.

range_product  The scalar product Operator on the range space. If None, the euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.

class pymor.operators.constructions.LincombOperator(operators, coefficients, name=None)
Bases: pymor.operators.basic.OperatorBase
An operator representing a linear combination of arbitrary Operators.

Parameters

operators  List of Operators whose linear combination is formed.

coefficients  A list of linear coefficients. A linear coefficient can either be a fixed number or a ParameterFunctional.

name  Name of the operator.

Methods

LincombOperator.apply, apply_adjoint, as_vector, assemble, evaluate_coefficients, jacobian, projected, projected_to_subbasis
OperatorBase.apply2, apply_inverse, lincomb, __add__, __mul__, __radd__, __str__
OperatorInterface.assemble, assemble_lincomb, restricted
ImmutableInterface.add_attributes, disable_logging, enable_logging,
  has_interface_name, implementor_names, implementors, with_, __setattr__
Parametric.build_parameter_type, local_parameter, parse_parameter,
  strip_parameter

Attributes

OperatorBase  invert_options
OperatorInterface  linear, range, source
ImmutableInterface  calculate_sid, sid, sid_failure, sid_ignore
BasicInterface  locked, logger, logging_disabled, name, uid, with_arguments
Parametric  parameter_local_type, parameter_space, parameter_type, parametric

apply(U, ind=None, mu=None)
Apply the operator.

Parameters

U  VectorArray of vectors to which the operator is applied.
apply_adjoint\( (U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source\_product}=\text{None}, \text{range\_product}=\text{None}) \)

Apply the adjoint operator.

For a linear operator \( A \) the adjoint \( A^\ast \) of \( A \) is given by

\[
(A^\ast v, u)_s = (v, Au)_r
\]

where \((,)_s\) and \((,)_r\) denote the scalar products on the source and range space of \( A \). If \( A \) and the two products are given by the matrices \( M, P_s \) and \( P_r \), then:

\[
A^\ast v = P_s^{-1} \cdot M^T \cdot P_r \cdot v
\]

with \( M^T \) denoting the transposed of \( M \). Thus, if \((,)_s\) and \((,)_r\) are the euclidean products, \( A^\ast v \) is simply given by multiplication of the matrix of \( A \) with \( v \) from the left.

Parameters

\( U \) \( \text{VectorArray} \) of vectors to which the adjoint operator is applied.

\( \text{ind} \) The indices of the vectors in \( U \) to which the operator shall be applied. (See the \( \text{VectorArray} \) documentation for further details.)

\( \text{mu} \) The \( \text{Parameter} \) for which to apply the adjoint operator.

\( \text{source\_product} \) The scalar product \( \text{Operator} \) on the source space. If \( \text{None} \), the euclidean product is chosen.

\( \text{range\_product} \) The scalar product \( \text{Operator} \) on the range space. If \( \text{None} \), the euclidean product is chosen.

Returns

\( \text{VectorArray} \) of the adjoint operator evaluations.

as_vector\( (\text{mu}=\text{None}) \)

Return a vector representation of a linear functional or vector operator.

This method may only be called on linear functionals, i.e. linear \( \text{Operators} \) with \( \text{NumpyVectorSpace}(1) \) as range, or on operators describing vectors, i.e. linear \( \text{Operators} \) with \( \text{NumpyVectorSpace}(1) \) as source.

In the case of a functional, the identity

\[
\text{self.as\_vector(}\text{mu}\text{)}.\text{dot(U)} = \text{self.apply(U, mu)}
\]

holds, whereas in the case of a vector-like operator we have

\[
\text{self.as\_vector(}\text{mu}\text{)} = \text{self.apply(}\text{NumpyVectorArray(1)}, \text{mu}).
\]
**assemble** *(mu=None)*

Assemble the operator for a given parameter.

What the result of the assembly is strongly depends on the given operator. For instance, a matrix-based operator will assemble its matrix, a `LincombOperator` will try to form the linear combination of its operators, whereas an arbitrary operator might simply return a `FixedParameterOperator`. The only assured property of the assembled operator is that it no longer depends on a `Parameter`.

**Parameters**

- **mu** The `Parameter` for which to assemble the operator.

**Returns**

Parameter-independent, assembled `Operator`.

**evaluate_coefficients** *(mu)*

Compute the linear coefficients of the linear combination for a given parameter.

**Parameters**

- **mu** `Parameter` for which to compute the linear coefficients.

**Returns**

List of linear coefficients.

**jacobian** *(U, mu=None)*

Return the operator’s Jacobian.

**Parameters**

- **U** Length 1 `VectorArray` containing the vector for which to compute the Jacobian.
- **mu** The `Parameter` for which to compute the Jacobian.

**Returns**

`Operator` representing the Jacobian.

**projected** *(source_basis, range_basis, product=None, name=None)*

Project the operator to subspaces of the source and range space.

Denote `self` by A. Given a scalar product `( , )`, and vectors \( b_1, ..., b_N, c_1, ..., c_M \), the projected operator \( A_P \) is defined by

\[
\{ A_P(e_j) \}_i = ( c_i, A(b_j) )
\]

for all i,j, where \( e_j \) denotes the j-th canonical basis vector of \( R^N \).
In particular, if the $c_i$ are orthonormal w.r.t. the given product, then $A_P$ is the coordinate representation w.r.t. the $b_i/c_i$ bases of the restriction of $A$ to $\text{span}(b_i)$ concatenated with the orthogonal projection onto $\text{span}(c_i)$.

From another point of view, if $A$ is viewed as a bilinear form (see `apply2`) and $(\cdot, \cdot)$ is the Euclidean product, then $A_P$ represents the matrix of the bilinear form restricted $\text{span}(b_i) / \text{span}(c_i)$ (w.r.t. the $b_i/c_i$ bases).

How the projected operator is realized will depend on the implementation of the operator to project. While a projected `NumpyMatrixOperator` will again be a `NumpyMatrixOperator`, only a generic `pymor.operators.basic.ProjectedOperator` will be returned in general. (Note that the latter will not be suitable to obtain an efficient offline/online-decomposition for reduced basis schemes.)

A default implementation is provided in `OperatorBase`.

**Warning:** No check is performed whether the $b_i$ and $c_j$ are orthonormal or linear independent.

**Parameters**

- **source_basis** The $b_1, \ldots, b_N$ as a `VectorArray` or `None`. If `None`, no restriction of the source space is performed.
- **range_basis** The $c_1, \ldots, c_M$ as a `VectorArray`. If `None`, no projection in the range space is performed.
- **product** An `Operator` representing the scalar product. If `None`, the Euclidean product is chosen.
- **name** Name of the projected operator.

**Returns**

The projected `Operator`.

**projected_to_subbasis** *(dim_source=\text{None}, \text{dim_range}=\text{None}, \text{name}=\text{None})*

See `NumpyMatrixOperator.projected_to_subbasis`.

**class** `pymor.operators.constructions.VectorArrayOperator` *(\text{array}, \quad \text{transposed}=\text{False}, \quad \text{copy}=\text{True}, \text{name}=\text{None})*

Bases: `pymor.operators.basic.OperatorBase`

Wraps a `VectorArray` as an `Operator`.

If `transposed == False`, the operator maps from `NumpyVectorSpace(len(array))` to `array.space` by forming linear combinations of the vectors in the array with given coefficient arrays.

If `transposed == True`, the operator maps from `array.space` to `NumpyVectorSpace(len(array))` by forming the scalar products of the arument with the vectors in the given array.

**Parameters**

- **array** The `VectorArray` which is to be treated as an operator.
- **transposed** See description above.
- **copy** If `True`, store a copy of `array` instead of `array` itself.
- **name** The name of the operator.
Methods

| VectorArrayOperator | apply, apply_adjoint, as_vector, assemble_lincomb |
| OperatorBase        | apply2, apply_inverse, assemble, jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__ |
| OperatorInterface   | restricted |
| ImmutableInterface  | lock |
| BasicInterface      | add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__ |
| Parametric          | build_parameter_type, local_parameter, parse_parameter, strip_parameter |

Attributes

| VectorArrayOperator | linear |
| OperatorBase       | invert_options |
| OperatorInterface  | range, source |
| ImmutableInterface | calculate_sid, sid, sid_failure, sid_ignore |
| BasicInterface     | locked, logger, logging_disabled, name, uid, with_arguments |
| Parametric         | parameter_local_type, parameter_space, parameter_type, parametric |

**apply** \((U, ind=None, mu=None)\)

Apply the operator.

**Parameters**

- **U** `VectorArray` of vectors to which the operator is applied.
- **ind** The indices of the vectors in \(U\) to which the operator shall be applied. (See the `VectorArray` documentation for further details.)
- **mu** The `Parameter` for which to evaluate the operator.

**Returns**

`VectorArray` of the operator evaluations.

**apply_adjoint** \((U, ind=None, mu=None, source_product=None, range_product=None)\)

Apply the adjoint operator.

For a linear operator \(A\) the adjoint \(A^*\) of \(A\) is given by

\[
(A^*v, u)_s = (v, Au)_r
\]

where \((, )_s\) and \((, )_r\) denote the scalar products on the source and range space of \(A\). If \(A\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[
A^*v = P_s^{-1} \times M^T \times P_r \times v
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \((, )_s\) and \((, )_r\) are the euclidean products, \(A^*v\) is simply given by multiplication of the matrix of \(A\) with \(v\) from the left.

**Parameters**

- **U** `VectorArray` of vectors to which the adjoint operator is applied.
**ind** The indices of the vectors in $U$ to which the operator shall be applied. (See the VectorArray documentation for further details.)

**mu** The Parameter for which to apply the adjoint operator.

**source_product** The scalar product Operator on the source space. If None, the euclidean product is chosen.

**range_product** The scalar product Operator on the range space. If None, the euclidean product is chosen.

Returns
VectorArray of the adjoint operator evaluations.

**as_vector** *(mu=None)*
Return a vector representation of a linear functional or vector operator.

This method may only be called on linear functionals, i.e. linear Operators with NumpyVectorSpace(1) as range, or on operators describing vectors, i.e. linear Operators with NumpyVectorSpace(1) as source.

In the case of a functional, the identity

$$\text{self.as_vector}(\mu).\text{dot}(U) == \text{self.apply}(U, \mu)$$

holds, whereas in the case of a vector-like operator we have

$$\text{self.as_vector}(\mu) == \text{self.apply}(\text{NumpyVectorArray}(1), \mu).$$

Parameters

**mu** The Parameter for which to return the vector representation.

Returns

V VectorArray of length 1 containing the vector representation. V belongs to self.source for functionals and to self.range for vector-like operators.

**assemble_lincomb** *(operators, coefficients, name=None)*
Try to assemble a linear combination of the given operators.

This method is called in the assemble method of LincombOperator on the first of its operator. If an assembly of the given linear combination is possible, e.g. the linear combination of the system matrices of the operators can be formed, then the assembled operator is returned. Otherwise, the method returns None to indicate that assembly is not possible.

Parameters

**operators** List of Operators whose linear combination is formed.

**coefficients** List of the corresponding linear coefficients.

**name** Name of the assembled operator.

Returns

The assembled Operator if assembly is possible, otherwise None.
class pymor.operators.constructions.VectorFunctional(vector, product=None, copy=True, name=None)

Bases: pymor.operators.constructions.VectorArrayOperator

Wrap a vector as a linear Functional.

Given a vector \( \mathbf{v} \) of dimension \( d \), this class represents the functional

\[
\mathbf{f}: \mathbb{R}^d \rightarrow \mathbb{R}^1 \\
\mathbf{u} \mapsto (\mathbf{u}, \mathbf{v})
\]

where \((,\)\) denotes the scalar product given by \( product \).

In particular, if \( product \) is None

\[
\text{VectorFunctional}(\mathbf{vector}).\text{as_vector()} = \mathbf{vector}.
\]

If \( product \) is not none, we obtain

\[
\text{VectorFunctional}(\mathbf{vector}).\text{as_vector()} = \text{product.apply(}\mathbf{vector}\text{)}.
\]

Parameters

\( \text{vector} \) \hspace{1em} \text{VectorArray} of length 1 containing the vector \( \mathbf{v} \).

\( \text{product} \) \hspace{1em} \text{Operator} representing the scalar product to use.

\( \text{copy} \) \hspace{1em} If True, store a copy of \( \text{vector} \) instead of \( \text{vector} \) itself.

\( \text{name} \) \hspace{1em} Name of the operator.

Methods

- VectorArrayOperator
  - apply, apply_adjoint, as_vector, assemble_lincomb
- OperatorBase
  - apply2, apply_inverse, assemble, jacobian, lincomb, projected,
    __add__, __mul__, __radd__, __str__
- OperatorInterface
- ImmutableInterface
- BasicInterface
  - add_attributes, disable_logging, enable_logging,
    has_interface_name, implementor_names, implementors, with_,
    __setattr__
- Parametric
  - build_parameter_type, local_parameter, parse_parameter,
    strip_parameter

Attributes

- VectorFunctional
  - linear, range
- OperatorBase
  - invert_options
- OperatorInterface
  - source
- ImmutableInterface
  - calculate_sid, sid, sid_failure, sid_ignore
- BasicInterface
  - locked, logger, logging_disabled, name, uid, with_arguments
- Parametric
  - parameter_local_type, parameter_space, parameter_type, parametric
**class** `pymor.operators.constructions.VectorOperator`

**Bases:** `pymor.operators.constructions.VectorArrayOperator`

Wrap a vector as a vector-like `Operator`.

Given a vector \( v \) of dimension \( d \), this class represents the operator

\[
\begin{align*}
op: & \mathbb{R}^1 \longrightarrow \mathbb{R}^d \\
x & \longmapsto xv
\end{align*}
\]

In particular

\[
\text{VectorOperator}(\text{vector}).\text{as_vector()} == \text{vector}
\]

**Parameters**

- **vector** `VectorArray` of length 1 containing the vector \( v \).
- **copy** If `True`, store a copy of `vector` instead of `vector` itself.
- **name** Name of the operator.

**Methods**

- `VectorArrayOperator` `apply`, `apply_adjoint`, `as_vector`, `assemble_lincomb`
- `OperatorBase` `apply2`, `apply_inverse`, `assemble`, `jacobian`, `lincomb`, `projected`, `__add__`, `__mul__`, `__radd__`, `__str__`
- `OperatorInterface` `restricted`
- `ImmutableInterface` `lock`, `unlock`
- `BasicInterface` `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`
- `Parametric` `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

**Attributes**

- `VectorOperator` `linear`, `source`
- `OperatorBase` `invert_options`
- `OperatorInterface` `range`
- `ImmutableInterface` `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `BasicInterface` `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`
- `Parametric` `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

**class** `pymor.operators.constructions.ZeroOperator`

**Bases:** `pymor.operators.basic.OperatorBase`

The `Operator` which maps every vector to zero.

**Parameters**

- **source** Source `VectorSpace` of the operator.
- **range** Range `VectorSpace` of the operator.
- **name** Name of the operator.
Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroOperator</td>
<td>apply, assemble_lincomb, projected</td>
</tr>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_adjoint, apply_inverse, as_vector, assemble, jacobian, lincomb, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>restrict</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>lock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with__, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroOperator</td>
<td>linear</td>
</tr>
<tr>
<td>OperatorBase</td>
<td>invert_options</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

apply \((U, \text{ind=None, \text{mu=None}})\)

Apply the operator.

Parameters

- **U** VectorArray of vectors to which the operator is applied.
- **ind** The indices of the vectors in \(U\) to which the operator shall be applied. (See the VectorArray documentation for further details.)
- **mu** The Parameter for which to evaluate the operator.

Returns

VectorArray of the operator evaluations.

assemble_lincomb \((\text{operators, coefficients, name=None})\)

Try to assemble a linear combination of the given operators.

This method is called in the `assemble` method of `LincombOperator` on the first of its operator. If an assembly of the given linear combination is possible, e.g. the linear combination of the system matrices of the operators can be formed, then the assembled operator is returned. Otherwise, the method returns `None` to indicate that assembly is not possible.

Parameters

- **operators** List of Operators whose linear combination is formed.
- **coefficients** List of the corresponding linear coefficients.
- **name** Name of the assembled operator.

Returns
The assembled Operator if assembly is possible, otherwise None.

**projected** *(source_basis, range_basis, product=None, name=None)*

Project the operator to subspaces of the source and range space.

Denote *self* by *A*. Given a scalar product (*, *), and vectors *b_*1, ..., *b_*N, *c_*1, ..., *c_*M, the projected operator *A_*P is defined by

\[
[A_P(e_j)]_i = (c_i, A(b_j))
\]

for all *i, j*, where *e_*j denotes the *j*-th canonical basis vector of \( \mathbb{R}^N \).

In particular, if the *c_*j are orthonormal w.r.t. the given product, then *A_*P is the coordinate representation w.r.t. the *b_*j/*c_*j bases of the restriction of *A* to span(*b_*j) concatenated with the orthogonal projection onto span(*c_*j).

From another point of view, if *A* is viewed as a bilinear form (see `apply2`) and (**, **) is the Euclidean product, then *A_*P represents the matrix of the bilinear form restricted span(*b_*j) / span(*c_*j) (w.r.t. the *b_*j/*c_*j bases).

How the projected operator is realized will depend on the implementation of the operator to project. While a projected *NumpyMatrixOperator* will again be a *NumpyMatrixOperator*, only a generic *pymor.operators.basic.ProjectedOperator* will be returned in general. (Note that the latter will not be suitable to obtain an efficient offline/online-decomposition for reduced basis schemes.)

A default implementation is provided in *OperatorBase*.

**Warning:** No check is performed whether the *b_*j and *c_*j are orthonormal or linear independent.

**Parameters**

- **source_basis** The *b_*1, ..., *b_*N as a *VectorArray* or *None*. If *None*, no restriction of the source space is performed.
- **range_basis** The *c_*1, ..., *c_*M as a *VectorArray*. If *None*, no projection in the range space is performed.
- **product** An *Operator* representing the scalar product. If *None*, the Euclidean product is chosen.
- **name** Name of the projected operator.

**Returns**

The projected *Operator*.

ei module

class *pymor.operators.ei.EmpiricalInterpolatedOperator* *(operator, interpolation_dofs, collateral_basis, triangular, name=None)*

Bases: *pymor.operators.basic.OperatorBase*

Interpolate an *Operator* using Empirical Operator Interpolation.

Let *L* be an *Operator*, \( 0 <= c_*1, ..., c_*M <= L.range.dim \) indices of interpolation DOFs and \( b_*1, ..., b_*M \) in \( \mathbb{R}^*(L.range.dim) \) collateral basis vectors. If moreover \( \psi(j(U)) \) denotes the *j*-th component of *U*, the empirical interpolation \( L_{EI} \) of *L* w.r.t. the given data is given by
Since the original operator only has to be evaluated at the given interpolation DOFs, `EmpiricalInterpolatedOperator` calls `restricted'(interpolation_dofs)` to obtain a restricted version of the operator which is stored and later used to quickly obtain the required evaluations. If the `restricted` method is not implemented, the full operator will be evaluated (which will lead to the same result, but without any speedup).

The interpolation DOFs and the collateral basis can be generated using the algorithms provided in the `pymor.algorithms.ei` module.

**Parameters**

- **operator** The `Operator` to interpolate. The operator must implement a `restricted` method as described above.
- **interpolation_dofs** List or 1D NumPy array of the interpolation DOFs \( c_1, \ldots, c_M \).
- **collateral_basis** VectorArray containing the collateral basis \( b_1, \ldots, b_M \).
- **triangular** If `True`, assume that \( \psi_i(c_j)(b_j) = 0 \) for \( i < j \), which means that the interpolation matrix is triangular.
- **name** Name of the operator.

**Methods**

- `apply`, `jacobian`, `projected` (from `OperatorBase`)
- `assemble`, `assemble_lincomb`, `restricted` (from `OperatorInterface`)
- `add`, `__add__`, `__mul__`, `__radd__`, `__str__`, `__setattr__` (from `ImmutableInterface`)
- `has_interface_name`, `implementor_names`, `implementors` (from `BasicInterface`)
- `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter` (from `Parametric`)

**Attributes**

- `invert_options` (from `OperatorBase`)
- `linear`, `range`, `source` (from `OperatorInterface`)
- `calculate_sid`, `sid`, `sid_failure`, `sid_ignore` (from `ImmutableInterface`)
- `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments` (from `BasicInterface`)
- `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric` (from `Parametric`)

**apply** `(U, ind=None, mu=None)`

Apply the operator.

**Parameters**

- **U** VectorArray of vectors to which the operator is applied.
**ind**  The indices of the vectors in $U$ to which the operator shall be applied. (See the `VectorArray` documentation for further details.)

**mu**  The `Parameter` for which to evaluate the operator.

**Returns**

`VectorArray` of the operator evaluations.

**jacobian** ($U, mu=None$)

Return the operator’s Jacobian.

**Parameters**

- **U**  Length 1 `VectorArray` containing the vector for which to compute the Jacobian.
- **mu**  The `Parameter` for which to compute the Jacobian.

**Returns**

`Operator` representing the Jacobian.

**projected** ($source\_basis, range\_basis, product=None, name=None$)

Project the operator to subspaces of the source and range space.

Denote `self` by $A$. Given a scalar product $(\cdot, \cdot)$, and vectors $b_1, \ldots, b_N, c_1, \ldots, c_M$, the projected operator $A_P$ is defined by

\[
A_P(e_j)_i = (c_i, A(b_j))
\]

for all $i,j$, where $e_j$ denotes the $j$-th canonical basis vector of $R^N$.

In particular, if the $c_i$ are orthonormal w.r.t. the given product, then $A_P$ is the coordinate representation w.r.t. the $b_i/c_i$ bases of the restriction of $A$ to $span(b_i)$ concatenated with the orthogonal projection onto $span(c_i)$.

From another point of view, if $A$ is viewed as a bilinear form (see `apply2`) and $(\cdot, \cdot)$ is the Euclidean product, then $A_P$ represents the matrix of the bilinear form restricted $span(b_i) / span(c_i)$ (w.r.t. the $b_i/c_i$ bases).

How the projected operator is realized will depend on the implementation of the operator to project. While a projected `NumpyMatrixOperator` will again be a `NumpyMatrixOperator`, only a generic `pymor.operators.basic.ProjectedOperator` will be returned in general. (Note that the latter will not be suitable to obtain an efficient offline/online-decomposition for reduced basis schemes.)

A default implementation is provided in `OperatorBase`.

**Warning:**  No check is performed whether the $b_i$ and $c_j$ are orthonormal or linear independent.

**Parameters**

- **source\_basis**  The $b_1, \ldots, b_N$ as a `VectorArray` or `None`. If `None`, no restriction of the source space is performed.
- **range\_basis**  The $c_1, \ldots, c_M$ as a `VectorArray`. If `None`, no projection in the range space is performed.
- **product**  An `Operator` representing the scalar product. If `None`, the Euclidean product is chosen.
**name**  Name of the projected operator.

**Returns**
The projected Operator.

class pymor.operators.ei.ProjectedEmpiricalInterpolatedOperator

(restricted_operator, interpolation_matrix, source_basis_dofs, projected_collateral_basis, triangular, name=None)

Bases: pymor.operators.basic.OperatorBase

A projected EmpiricalInterpolatedOperator.

Not intended to be used directly. Instead use projected.

**Methods**

<table>
<thead>
<tr>
<th>OperatorBase</th>
<th>apply, apply2, apply_adjoint, apply_inverse, as_vector, assemble, lincomb, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>assemble, lincomb, restricted</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>lock, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with___setattr__</td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

**apply** *(U, ind=None, mu=None)*

Apply the operator.

**Parameters**

*U*  VectorArray of vectors to which the operator is applied.

*ind*  The indices of the vectors in *U* to which the operator shall be applied. (See the VectorArray documentation for further details.)

*mu*  The Parameter for which to evaluate the operator.

**Returns**
VectorArray of the operator evaluations.

\[ \text{jacobian} \left( U, \mu=\text{None} \right) \]

Return the operator’s Jacobian.

Parameters

- **U**: Length 1 VectorArray containing the vector for which to compute the Jacobian.
- **mu**: The Parameter for which to compute the Jacobian.

Returns

Operator representing the Jacobian.

fv module  This module provides some operators for finite volume discretizations.

class pymor.operators.fv.DiffusionOperator(grid, boundary_info, diffusion_function=\text{None},
\text{diffusion_constant}=\text{None}, name=\text{None})

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator

Finite Volume Diffusion Operator.

The operator is of the form

\[
(Lu)(x) = c \left[ d(x) \ u(x) \right]
\]

Parameters

- **grid**: The Grid over which to assemble the operator.
- **boundary_info**: BoundaryInfo for the treatment of Dirichlet boundary conditions.
- **diffusion_function**: The the scalar-valued Function \( d(x) \). If \( \text{None} \), constant one is assumed.
- **diffusion_constant**: The constant \( c \). If \( \text{None} \), \( c \) is set to one.
- **name**: Name of the operator.

Methods

- apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix
- apply2, jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__
- assemble_lincomb, restricted
- add, lock, unlock
- add_attributes, disable_logging, enable_logging,
  has_interface_name, implementor_names, implementors, with_,
  __setattr__
- build_parameter_type, local_parameter, parse_parameter,
  strip_parameter

Attributes
class pymor.operators.fv.EngquistOsherFlux(flux, flux_derivative, gausspoints=5, intervals=1)

Bases: pymor.operators.fv.NumericalConvectiveFluxInterface

Engquist-Osher numerical flux.

If \( f \) is the analytical flux, and \( f' \) its derivative, the Engquist-Osher flux is given by

\[
F(U_{\text{in}}, U_{\text{out}}, \text{normal}, \text{vol}) = \text{vol} \times [c^+(U_{\text{in}}, \text{normal}) + c^-(U_{\text{out}}, \text{normal})]
\]

\[
c^+(U_{\text{in}}, \text{normal}) = f(0) \text{normal} + \max(f'(s) \text{normal}, 0) \, ds_{s=0}
\]

\[
c^-(U_{\text{out}}, \text{normal}) = \min(f'(s) \text{normal}, 0) \, ds_{s=0}
\]

Parameters

- **flux** Function defining the analytical flux \( f \).
- **flux_derivative** Function defining the analytical flux derivative \( f' \).
- **gausspoints** Number of Gauss quadrature points to be used for integration.
- **intervals** Number of subintervals to be used for integration.

Methods

- EngquistOsherFlux.evaluate_stage1, evaluate_stage2
- ImmutableInterface.lock
- BasicInterface.add_attributes, disable_logging, enable_logging,
  has_interface_name, implementor_names, implementors, with_, __setattr__
- Parametric.build_parameter_type, local_parameter, parse_parameter,
  strip_parameter

Attributes

- ImmutableInterface.calculate_sid, sid, sid_failure, sid_ignore
- BasicInterface.locked, logger, logging_disabled, name, uid, with_arguments
- Parametric parameter_local_type, parameter_space, parameter_type, parametric

class pymor.operators.fv.L2Product(grid, name=None)

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator

4.1. pymor package
Operator representing the L2-product between finite volume functions.

To evaluate the product use the apply2 method.

Parameters

grid  The Grid for which to assemble the product.

name  The name of the product.

Methods

NumpyMatrixBasedOperator: apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix
OperatorBase: apply2, jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__
OperatorInterface: assemble_lincomb, restricted
ImmutableInterface: __add__, __mul__, __radd__, __str__
BasicInterface: add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with__, __setattr__
Parametric: build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

| L2Product | sparse |
| NumpyMatrixBasedOperator | invert_options, linear |
| OperatorInterface | range, source |
| ImmutableInterface | calculate_sid, sid, sid_failure, sid_ignore |
| BasicInterface | locked, logger, logging_disabled, name, uid, with_arguments |
| Parametric | parameter_local_type, parameter_space, parameter_type, parametric |

class pymor.operators.fv.L2ProductFunctional (grid, function=None, boundary_info=None, dirichlet_data=None, diffusion_function=None, diffusion_constant=None, neumann_data=None, order=1, name=None)

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator

Finite volume Functional representing the scalar product with an L2-Function.

Additionally boundary conditions can be enforced by providing dirichlet_data and neumann_data functions.

Parameters

grid  Grid for which to assemble the functional.

function  The Function with which to take the scalar product or None.

boundary_info  BoundaryInfo determining the Dirichlet and Neumann boundaries or None. If None, no boundary treatment is performed.

dirichlet_data  Function providing the Dirichlet boundary values. If None, constant-zero boundary is assumed.
diffusion_function  See DiffusionOperator. Has to be specified in case dirichlet_data is given.

diffusion_constant  See DiffusionOperator. Has to be specified in case dirichlet_data is given.

neumann_data  Function providing the Neumann boundary values. If None, constant-zero is assumed.

order  Order of the Gauss quadrature to use for numerical integration.

name  The name of the functional.

Methods

<table>
<thead>
<tr>
<th>NumpyMatrixBasedOperator</th>
<th>apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, jacobian, lincomb, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>assemble, lincomb, restricted</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><strong>add</strong>, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with__, <strong>setattr</strong></td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>L2ProductFunctional</th>
<th>range, sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>invert_options, linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

class pymor.operators.fv.LaxFriedrichsFlux (flux, lxf_lambda=1.0)

    Bases: pymor.operators.fv.NumericalConvectiveFluxInterface

Lax-Friedrichs numerical flux.

If f is the analytical flux, the Lax-Friedrichs flux F is given by

\[ F(U_{in}, U_{out}, normal, vol) = vol \times \left\{ \frac{\text{normal}(f(U_{in}) + f(U_{out}))}{2} + \frac{(U_{in} - U_{out})}{2\lambda} \right\} \]

Parameters

<table>
<thead>
<tr>
<th>flux  Function defining the analytical flux f.</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxf_lambda  The stabilization parameter (\lambda).</td>
</tr>
</tbody>
</table>

Methods
class `pymor.operators.fv.LinearAdvectionLaxFriedrichs` (grid, boundary_info, velocity_field, lxf_lambda=1.0, name=None)

Bases: `pymor.operators.numpy.NumpyMatrixBasedOperator`

Linear advection finite Volume Operator using Lax-Friedrichs flux.

The operator is of the form

\[ L(u, \mu)(x) = (v(x, \mu)u(x)) \]

See `LaxFriedrichsFlux` for the definition of the Lax-Friedrichs flux.

Parameters

- **grid** `Grid` over which to assemble the operator.
- **boundary_info** `BoundaryInfo` determining the Dirichlet and Neumann boundaries.
- **velocity_field** `Function` defining the velocity field \( v \).
- **lxf_lambda** The stabilization parameter \( \lambda \).
- **name** The name of the operator.

Methods

- **NumpyMatrixBase**: `apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix`
- **OperatorBase**: `apply2, jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__`
- **OperatorInterface**: `assemble_lincomb, restricted`
- **ImmutableInterface**: `calculate_sid, sid, sid_failure, sid_ignore`
- **BasicInterface**: `has_interface_name, implementor_names, implementors, with__setattr__`
- **Parametric**: `build_parameter_type, local_parameter, parse_parameter, strip_parameter`

Attributes
class `pymor.operators.fv.NonlinearAdvectionOperator` (grid, boundary_info, numerical_flux, dirichlet_data=None, name=None)

Bases: `pymor.operators.basic.OperatorBase`

Nonlinear finite volume advection Operator.

The operator is of the form

\[ L(u, \mu)(x) = f(u(x), \mu) \]

**Note:** For Neumann boundaries, currently only zero boundary values are implemented.

**Parameters**

- **grid** `Grid` for which to evaluate the operator.
- **boundary_info** `BoundaryInfo` determining the Dirichlet and Neumann boundaries.
- **numerical_flux** The `NumericalConvectiveFlux` to use.
- **dirichlet_data** `Function` providing the Dirichlet boundary values. If `None`, constant-zero boundary is assumed.
- **name** The name of the operator.

**Methods**

- `apply`, `restricted`, `with_`
- `apply2`, `apply_adjoint`, `apply_inverse`, `as_vector`, `assemble`, `jacobian`, `lincomb`, `projected`, `__add__`, `__mul__`, `__radd__`, `__str__`
- `assemble_lincomb`
- `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `__setattr__`
- `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

**Attributes**

- `invert_options`, `linear`, `sparse`
- `range`, `source`
- `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`
- `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`
apply \((U, \text{ind}=\text{None}, \mu=\text{None})\)

Apply the operator.

**Parameters**

- \(U\) VectorArray of vectors to which the operator is applied.
- \(\text{ind}\) The indices of the vectors in \(U\) to which the operator shall be applied. (See the VectorArray documentation for further details.)
- \(\mu\) The Parameter for which to evaluate the operator.

**Returns**

VectorArray of the operator evaluations.

restricted \((\text{dofs})\)

Restrict the operator range to a given set of degrees of freedom.

This method returns a restricted version \(\text{restricted\_op}\) of the operator along with an array \(\text{source\_dofs}\) such that for any VectorArray \(U\) in \(\text{self.source}\) the following is true:

\[
\text{self.apply}(U, \mu).\text{components}(\text{dofs})
\]
\[
== \text{restricted\_op}.\text{apply}(\text{NumpyVectorArray}(U.\text{components}(\text{source\_dofs})), \mu)
\]

Such an operator is mainly useful for empirical interpolation where the evaluation of the original operator only needs to be known for few selected degrees of freedom. If the operator has a small stencil, only few \(\text{source\_dofs}\) will be needed to evaluate the restricted operator which can make its evaluation very fast compared to evaluating the original operator. Note that the interface does not make any assumptions on the efficiency of evaluating the restricted operator.

**Parameters**

- \(\text{dofs}\) One-dimensional NumPy array of degrees of freedom in the operator range to which to restrict.

**Returns**

- \(\text{restricted\_op}\) The restricted operator as defined above. The operator will have \(\text{NumpyVectorSpace (len(source\_dofs))}\) as source and \(\text{NumpyVectorSpace (len(dofs))}\) as range.
- \(\text{source\_dofs}\) One-dimensional NumPy array of source degrees of freedom as defined above.

with_ \((**\text{kwargs})\)

Returns a copy with changed attributes.

The default implementation is to call \_with\_via\_init(**\text{kwargs}).

**Parameters**

- \(**\text{kwargs}\) Names of attributes to change with their new values. Each attribute name has to be contained in \text{with\_arguments}.

**Returns**

Copy of \(\text{self}\) with changed attributes.
class pymor.operators.fv.NumericalConvectiveFluxInterface
Bases: pymor.core.interfaces.ImmutableInterface, pymor.parameters.base.Parametric

Interface for numerical convective fluxes for finite volume schemes.

Numerical fluxes defined by this interfaces are functions of the form $F(U_{inner}, U_{outer}, \text{unit\_outer\_normal, edge\_volume, } \mu)$.

The flux evaluation is vectorized and happens in two stages:

1. evaluate_stagel receives a NumPy array $U$ of all values which appear as $U_{inner}$ or $U_{outer}$ for the edges the flux shall be evaluated at and returns a tuple of NumPy arrays each of the same length as $U$.

2. evaluate_stag2 receives the reordered stage1_data for each edge as well as the unit outer normal and the volume of the edges.

stage1_data is given as follows: If $R_l$ is $l$-th entry of the tuple returned by evaluate_stagel, the $l$-th entry $D_l$ of of the stage1_data tuple has the shape $(\text{num\_edges}, 2) + R_l.shape[1:]$. If for edge $k$ the values $U_{inner}$ and $U_{outer}$ are the $i$-th and $j$-th value in the $U$ array provided to evaluate_stagel, we have

\[ D_l[k, 0] = R_l[i], \quad D_l[k, 1] = R_l[j]. \]

evaluate_stag2 returns a NumPy array of the flux evaluations for each edge.

Methods

- NumericalConvectiveFluxInterface: evaluate_stagel, evaluate_stag2
- ImmutableInterface: lock, unlock
- BasicInterface: add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__
- Parametric: build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

- ImmutableInterface: recalculate_sid, sid, sid_failure, sid_ignore
- BasicInterface: locked, logger, logging_disabled, name, uid, with_arguments
- Parametric: parameter_local_type, parameter_space, parameter_type, parametric

class pymor.operators.fv.SimplifiedEngquistOsherFlux (flux, flux_derivative)
Bases: pymor.operators.fv.NumericalConvectiveFluxInterface

Engquist-Osher numerical flux. Simplified Implementation for special case.

For the definition of the Enquist-Osher flux see EngquistOsherFlux. This class provides a faster and more accurate implementation for the special case that $f(0) = 0$ and the derivative of $f$ only changes sign at $0$.

Parameters

- flux Function defining the analytical flux $f$.
- flux_derivative Function defining the analytical flux derivative $f'$. 

4.1. pymor package 161
### Methods

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>evaluate_stage1, evaluate_stage2, unlock, add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong>, calculate_sid, sid, sid_failure, sid_ignore, locked, logger, logging_disabled, name, uid, with_arguments, build_parameter_type, local_parameter, parse_parameter, strip_parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Parametric</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>calculate_sid, sid, sid_failure, sid_ignore, locked, logger, logging_disabled, name, uid, with_arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

```python
pymor.operators.fv.nonlinear_advection_engquist_osher_operator(grid, boundary_info, flux, flux_derivative, gausspoints=5, intervals=1, dirichlet_data=None, name=None)
```

Instantiate a **NonlinearAdvectionOperator** using **EngquistOsherFlux**.

```python
pymor.operators.fv.nonlinear_advection_lax_friedrichs_operator(grid, boundary_info, flux, lxf_lambda=1.0, dirichlet_data=None, name=None)
```

Instantiate a **NonlinearAdvectionOperator** using **LaxFriedrichsFlux**.

```python
pymor.operators.fv.nonlinear_advection_simplified_engquist_osher_operator(grid, boundary_info, flux, flux_derivative, dirichlet_data=None, name=None)
```

Instantiate a **NonlinearAdvectionOperator** using **SimplifiedEngquistOsherFlux**.

```
```

Interfaces module
class pymor.operators.interfaces.OperatorInterface
Bases: pymor.core.interfaces.ImmutableInterface, pymor.parameters.base.Parametric

Interface for Parameter dependent discrete operators.

An operator in pyMOR is simply a mapping which for any given Parameter maps vectors from its source VectorSpace to vectors in its range VectorSpace.

Note that there is no special distinction between functionals and operators in pyMOR. A functional is simply an operator with NumpyVectorSpace(1) as its range VectorSpace.

Methods

<table>
<thead>
<tr>
<th>OperatorInterface</th>
<th>apply, apply2, apply_adjoint, apply_inverse, as_vector, assemble, assemble_lincomb, jacobian, lincomb, projected, restricted, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>lock, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>OperatorInterface</th>
<th>invert_options, linear, range, source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

invert_options

OrderedDict of possible options for apply_inverse. Each key is a type of inversion algorithm which can be used to invert the operator. invert_options[k] is a dict containing all options along with their default values which can be set for algorithm k. We always have invert_options[k]['type'] == k such that invert_options[k] can be passed directly to apply_inverse.

linear

True if the operator is linear.

source

The source VectorSpace.

range

The range VectorSpace.

__add__(other)

Sum of two operators

__mul__(other)

Product of operator by a scalar

__radd__(other)

Sum of two operators

apply (U, ind=None, mu=None)

Apply the operator.

Parameters

U VectorArray of vectors to which the operator is applied.
ind  The indices of the vectors in $U$ to which the operator shall be applied. (See the VectorArray documentation for further details.)

mu  The Parameter for which to evaluate the operator.

Returns
VectorArray of the operator evaluations.

apply2($V, U, pairwise, U_ind=None, V_ind=None, mu=None, product=None$)
Treat the operator as a 2-form by calculating $(V, A(U))$.
In particular, if $(, )$ is the Euclidean product and $A$ is a linear operator given by multiplication with a matrix $M$, then
$A.apply2(V, U) = V^T*M*U$

Parameters
V  VectorArray of the left arguments $V$.
U  VectorArray of the right arguments $U$.
pairwise  If False, the 2-form is applied to all combinations of vectors in $V$ and $U$, i.e.

$$L.apply2(V, U).shape = (\text{len}(V_ind), \text{len}(U_ind)).$$

If True, the vectors in $V$ and $U$ are applied in pairs, i.e. $V$ and $U$ must be of the same length and we have

$$L.apply2(V, U).shape = (\text{len}(V_ind),) = (\text{len}(U_ind),).$$

V_ind  The indices of the vectors in $V$ to which the operator shall be applied. (See the VectorArray documentation for further details.)
U_ind  The indices of the vectors in $U$ to which the operator shall be applied. (See the VectorArray documentation for further details.)
mu  The Parameter for which to evaluate the operator.
product  The scalar product used in the expression $(V, A(U))$ given as an Operator. If None, the euclidean product is chosen.

Returns
A NumPy array of all 2-form evaluations.

apply_adjoint($U, ind=None, mu=None, source_product=None, range_product=None$)
Apply the adjoint operator.
For a linear operator $A$ the adjoint $A^*$ of $A$ is given by

$$(A^*v, u)_s = (v, Au)_r$$

where $(, )_s$ and $(, )_r$ denote the scalar products on the source and range space of $A$. If $A$ and the two products are given by the matrices $M$, $P_s$ and $P_r$, then:

$$A^*v = P_s^*(-1) * N^T * P_r * v$$
with $M^T$ denoting the transposed of $M$. Thus, if $(\cdot,\cdot)_s$ and $(\cdot,\cdot)_r$ are the euclidean products, $A^*v$ is simply given by multiplication of the matrix of $A$ with $v$ from the left.

**Parameters**

- **U** : `VectorArray` of vectors to which the adjoint operator is applied.
- **ind** : The indices of the vectors in $U$ to which the operator shall be applied. (See the `VectorArray` documentation for further details.)
- **mu** : The `Parameter` for which to apply the adjoint operator.
- **source_product** : The scalar product `Operator` on the source space. If `None`, the euclidean product is chosen.
- **range_product** : The scalar product `Operator` on the range space. If `None`, the euclidean product is chosen.

**Returns**

`VectorArray` of the adjoint operator evaluations.

**apply_inverse** *(U, ind=None, mu=None, options=None)*

Apply the inverse operator.

**Parameters**

- **U** : `VectorArray` of vectors to which the inverse operator is applied.
- **ind** : The indices of the vectors in $U$ to which the operator shall be applied. (See the `VectorArray` documentation for further details.)
- **mu** : The `Parameter` for which to evaluate the inverse operator.
- **options** : Dictionary of options for the inversion algorithm. The dictionary has to contain the key ‘`type`’ whose value determines which inversion algorithm is to be used. All other items represent options specific to this algorithm. `options` can also be given as a string, which is then interpreted as the type of inversion algorithm. If `options` is `None`, a default algorithm with default options is chosen. Available algorithms and their default options are provided by `invert_options`.

**Returns**

`VectorArray` of the inverse operator evaluations.

**Raises**

`InversionError` : The operator could not be inverted.

**as_vector** *(mu=None)*

Return a vector representation of a linear functional or vector operator.

This method may only be called on linear functionals, i.e. linear `Operators` with `NumpyVectorSpace(1)` as `range`, or on operators describing vectors, i.e. linear `Operators` with `NumpyVectorSpace(1)` as `source`.

In the case of a functional, the identity

$$
\text{self.as_vector}(\mu) \cdot (U) = \text{self.apply}(U, \mu)
$$
holds, whereas in the case of a vector-like operator we have

\[
\text{self.as_vector(mu)} == \text{self.apply(NumpyVectorArray(1), mu)}.\]

### Parameters

- **mu**  The Parameter for which to return the vector representation.

### Returns

- **V**  VectorArray of length 1 containing the vector representation.  \(V\) belongs to \(\text{self.source}\) for functionals and to \(\text{self.range}\) for vector-like operators.

#### assemble (mu=None)

Assemble the operator for a given parameter.

What the result of the assembly is strongly depends on the given operator. For instance, a matrix-based operator will assemble its matrix, a LincombOperator will try to form the linear combination of its operators, whereas an arbitrary operator might simply return a FixedParameterOperator. The only assured property of the assembled operator is that it no longer depends on a Parameter.

### Parameters

- **mu**  The Parameter for which to assemble the operator.

### Returns

Parameter-independent, assembled Operator.

#### assemble_lincomb (operators, coefficients, name=None)

Try to assemble a linear combination of the given operators.

This method is called in the assemble method of LincombOperator on the first of its operator. If an assembly of the given linear combination is possible, e.g. the linear combination of the system matrices of the operators can be formed, then the assembled operator is returned. Otherwise, the method returns None to indicate that assembly is not possible.

### Parameters

- **operators**  List of Operators whose linear combination is formed.
- **coefficients**  List of the corresponding linear coefficients.
- **name**  Name of the assembled operator.

### Returns

The assembled Operator if assembly is possible, otherwise None.

#### jacobian (U, mu=None)

Return the operator’s Jacobian.

### Parameters

- **U**  Length 1 VectorArray containing the vector for which to compute the Jacobian.
- **mu**  The Parameter for which to compute the Jacobian.
Returns

Operator representing the Jacobian.

**static lincomb** (operators, coefficients, name=None)

DEPRECATED! Use `pymor.operators.constructions.LincombOperator` instead.

**projected** (source_basis, range_basis, product=None, name=None)

Project the operator to subspaces of the source and range space.

Denote `self` by A. Given a scalar product ( , ), and vectors b_1, ..., b_N, c_1, ..., c_M, the projected operator A_P is defined by

\[
[A_P(e_j)]_i = (c_i, A(b_j))
\]

for all i,j, where e_j denotes the j-th canonical basis vector of \( \mathbb{R}^N \).

In particular, if the c_i are orthonormal w.r.t. the given product, then A_P is the coordinate representation w.r.t. the b_i/c_i bases of the restriction of A to \( \text{span}(b_i) \) concatenated with the orthogonal projection onto \( \text{span}(c_i) \).

From another point of view, if A is viewed as a bilinear form (see `apply2`) and ( , ) is the Euclidean product, then A_P represents the matrix of the bilinear form restricted \( \text{span}(b_i) / \text{span}(c_i) \) (w.r.t. the b_i/c_i bases).

How the projected operator is realized will depend on the implementation of the operator to project. While a projected `NumpyMatrixOperator` will again be a `NumpyMatrixOperator`, only a generic `pymor.operators.basic.ProjectedOperator` will be returned in general. (Note that the latter will not be suitable to obtain an efficient offline/online-decomposition for reduced basis schemes.)

A default implementation is provided in `OperatorBase`.

**Warning:** No check is performed whether the b_i and c_j are orthonormal or linear independent.

**Parameters**

- `source_basis` The b_1, ..., b_N as a `VectorArray` or `None`. If `None`, no restriction of the source space is performed.
- `range_basis` The c_1, ..., c_M as a `VectorArray`. If `None`, no projection in the range space is performed.
- `product` An `Operator` representing the scalar product. If `None`, the Euclidean product is chosen.
- `name` Name of the projected operator.

**Returns**

The projected `Operator`.

**restricted** (dofs)

Restrict the operator range to a given set of degrees of freedom.

This method returns a restricted version restricted_op of the operator along with an array `source_dofs` such that for any `VectorArray` U in `self.source` the following is true:

```python
self.apply(U, mu).components(dofs)
== restricted_op.apply(NumpyVectorArray(U.components(source_dofs)), mu))
```
Such an operator is mainly useful for empirical interpolation where the evaluation of the original operator only needs to be known for few selected degrees of freedom. If the operator has a small stencil, only few source_dofs will be needed to evaluate the restricted operator which can make its evaluation very fast compared to evaluating the original operator. Note that the interface does not make any assumptions on the efficiency of evaluating the restricted operator.

**Parameters**

dofs One-dimensional NumPy array of degrees of freedom in the operator range to which to restrict.

**Returns**

restricted_op The restricted operator as defined above. The operator will have NumpyVectorSpace (len(source_dofs)) as source and NumpyVectorSpace (len(dofs)) as range.

source_dofs One-dimensional NumPy array of source degrees of freedom as defined above.

---

**numpy module**  This module provides the following NumPy based Operators:

- NumpyMatrixOperator wraps a 2D NumPy array as a Operator.
- NumpyMatrixBasedOperator should be used as base class for all Operators which assemble into a NumpyMatrixOperator.
- NumpyGenericOperator wraps an arbitrary Python function between NumPy arrays as an Operator.

---

```python
class pymor.operators.numpy.NumpyGenericOperator(mapping, dim_source=1, dim_range=1, linear=False, parameter_type=None, name=None)
```

Wraps an arbitrary Python function between NumPy arrays as a an Operator.

**Parameters**

- mapping The function to wrap. If parameter_type is None, the function is of the form mapping(U) and is expected to be vectorized. In particular:

  ```
  mapping(U).shape == U.shape[:-1] + (dim_range,).
  ```

  If parameter_type is not None, the function has to have the signature mapping(U, mu).

- dim_source Dimension of the operator’s source.

- dim_range Dimension of the operator’s range.

- linear Set to True if the provided mapping is linear.

- parameter_type The ParameterType of the Parameters the mapping accepts.

- name Name of the operator.

**Methods**
apply \((U, \text{ind}=\text{None}, \mu=\text{None})\)

Apply the operator.

Parameters

- **U** VectorArray of vectors to which the operator is applied.
- **ind** The indices of the vectors in \(U\) to which the operator shall be applied. (See the VectorArray documentation for further details.)
- **\mu** The Parameter for which to evaluate the operator.

Returns

VectorArray of the operator evaluations.
Attributes

<table>
<thead>
<tr>
<th>NumpyMatrixBasedOperator</th>
<th>invert_options, linear, sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>range, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

```
sparse
  True if the operator assembles into a sparse matrix, False if the operator assembles into a dense matrix, None if unknown.
```

```
apply (U, ind=None, mu=None)
  Apply the operator.

Parameters
  U VectorArray of vectors to which the operator is applied.
  ind The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)
  mu The Parameter for which to evaluate the operator.

Returns
  VectorArray of the operator evaluations.
```

```
apply_adjoint (U, ind=None, mu=None, source_product=None, range_product=None)
  Apply the adjoint operator.
  For a linear operator A the adjoint $A^*$ of A is given by
  $$(A^*v, u)_s = (v, Au)_r$$
  where $( , )_s$ and $( , )_r$ denote the scalar products on the source and range space of A. If A and the two products are given by the matrices $M$, $P_s$ and $P_r$, then:
  $$A^*v = P_s^{(-1)} \ast M^T \ast P_r \ast v$$
  with $M^T$ denoting the transposed of $M$. Thus, if $( , )_s$ and $( , )_r$ are the euclidean products, $A^*v$ is simply given by multiplication of the matrix of A with v from the left.

Parameters
  U VectorArray of vectors to which the adjoint operator is applied.
  ind The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)
  mu The Parameter for which to apply the adjoint operator.
  source_product The scalar product Operator on the source space. If None, the euclidean product is chosen.
  range_product The scalar product Operator on the range space. If None, the euclidean product is chosen.
Returns
VectorArray of the adjoint operator evaluations.

apply_inverse(U, ind=None, mu=None, options=None)
Apply the inverse operator.

Parameters
U VectorArray of vectors to which the inverse operator is applied.
ind The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)
mu The Parameter for which to evaluate the inverse operator.
options Dictionary of options for the inversion algorithm. The dictionary has to contain the key ‘type’ whose value determines which inversion algorithm is to be used. All other items represent options specific to this algorithm. options can also be given as a string, which is then interpreted as the type of inversion algorithm. If options is None, a default algorithm with default options is chosen. Available algorithms and their default options are provided by invert_options.

Returns
VectorArray of the inverse operator evaluations.

Raises
InversionError The operator could not be inverted.

as_vector(mu=None)
Return a vector representation of a linear functional or vector operator.

This method may only be called on linear functionals, i.e. linear Operators with NumpyVectorSpace(1) as range, or on operators describing vectors, i.e. linear Operators with NumpyVectorSpace(1) as source.

In the case of a functional, the identity

\[ \text{self.as_vector}(\mu) \cdot \text{dot}(U) = \text{self.apply}(U, \mu) \]

holds, whereas in the case of a vector-like operator we have

\[ \text{self.as_vector}(\mu) = \text{self.apply} \left( \text{NumpyVectorArray}(1), \mu \right) \].

Parameters
mu The Parameter for which to return the vector representation.

Returns
V VectorArray of length 1 containing the vector representation. V belongs to self.source for functionals and to self.range for vector-like operators.

assemble(mu=None)
Assemble the operator for a given parameter.
What the result of the assembly is strongly depends on the given operator. For instance, a matrix-based operator will assemble its matrix, a LincombOperator will try to form the linear combination of its operators, whereas an arbitrary operator might simply return a FixedParameterOperator. The only assured property of the assembled operator is that it no longer depends on a Parameter.

**Parameters**

- **mu**: The Parameter for which to assemble the operator.

**Returns**

Parameter-independent, assembled Operator.

### export_matrix(filename, matrix_name=None, output_format='matlab', mu=None)

Save the matrix of the operator to a file.

**Parameters**

- **filename**: Name of output file.
- **matrix_name**: The name, the output matrix is given. (Comment field is used in case of Matrix Market output_format.) If None, the Operator’s name is used.
- **output_format**: Output file format. Either matlab or matrixmarket.

### NumpyMatrixOperator(matrix, name=None)

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator

Wraps a 2D NumPy array as an Operator.

**Parameters**

- **matrix**: The NumPy array which is to be wrapped.
- **name**: Name of the operator.

**Methods**

- apply, apply_adjoint, apply_inverse, as_vector, assemble, assemble_lincomb, projected_to_subbasis
- apply2, jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__
- restricted
- add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, __setattr__
- build_parameter_type, local_parameter, parse_parameter, strip_parameter

**Attributes**
apply \((U, \text{ind}=\text{None}, \text{mu}=\text{None})\)  
Apply the operator.

Parameters

\(U\) VectorArray of vectors to which the operator is applied.

\(\text{ind}\) The indices of the vectors in \(U\) to which the operator shall be applied. (See the VectorArray documentation for further details.)

\(\text{mu}\) The Parameter for which to evaluate the operator.

Returns

VectorArray of the operator evaluations.

apply_adjoint \((U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None})\)  
Apply the adjoint operator.

For a linear operator \(A\) the adjoint \(A^*\) of \(A\) is given by

\[
(A^*v, u)_s = (v, Au)_r
\]

where \((,)_s\) and \((,)_r\) denote the scalar products on the source and range space of \(A\). If \(A\) and the two products are given by the matrices \(M\), \(P_s\) and \(P_r\), then:

\[
A^*v = P_s^\dagger \cdot M^T \cdot P_r \cdot v
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \((,)_s\) and \((,)_r\) are the euclidean products, \(A^*v\) is simply given by multiplication of the matrix of \(A\) with \(v\) from the left.

Parameters

\(U\) VectorArray of vectors to which the adjoint operator is applied.

\(\text{ind}\) The indices of the vectors in \(U\) to which the operator shall be applied. (See the VectorArray documentation for further details.)

\(\text{mu}\) The Parameter for which to apply the adjoint operator.

\(\text{source_product}\) The scalar product \(\text{Operator}\) on the source space. If \(\text{None}\), the euclidean product is chosen.

\(\text{range_product}\) The scalar product \(\text{Operator}\) on the range space. If \(\text{None}\), the euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.
apply_inverse \( (U, \text{ind}=\text{None}, \mu=\text{None}, \text{options}=\text{None}) \)
Apply the inverse operator.

**Parameters**

- **U** : VectorArray of vectors to which the inverse operator is applied.
- **ind** : The indices of the vectors in \( U \) to which the operator shall be applied. (See the VectorArray documentation for further details.)
- **\mu** : The Parameter for which to evaluate the inverse operator.
- **options** : Dictionary of options for the inversion algorithm. The dictionary has to contain the key ‘type’ whose value determines which inversion algorithm is to be used. All other items represent options specific to this algorithm. options can also be given as a string, which is then interpreted as the type of inversion algorithm. If options is None, a default algorithm with default options is chosen. Available algorithms and their default options are provided by invert_options.

**Returns**

VectorArray of the inverse operator evaluations.

**Raises**

InversionError  The operator could not be inverted.

as_vector \( (\mu=\text{None}) \)
Return a vector representation of a linear functional or vector operator.

This method may only be called on linear functionals, i.e. linear Operators with NumpyVectorSpace(1) as range, or on operators describing vectors, i.e. linear Operators with NumpyVectorSpace(1) as source.

In the case of a functional, the identity

self.as_vector(\mu).dot(U) == self.apply(U, \mu)

holds, whereas in the case of a vector-like operator we have

self.as_vector(\mu) == self.apply(NumpyVectorArray(1), \mu).

**Parameters**

- **\mu** : The Parameter for which to return the vector representation.

**Returns**

V : VectorArray of length 1 containing the vector representation. V belongs to self.source for functionals and to self.range for vector-like operators.

assemble \( (\mu=\text{None}) \)
Assemble the operator for a given parameter.

What the result of the assembly is strongly depends on the given operator. For instance, a matrix-based operator will assemble its matrix, a LincombOperator will try to form the linear combination of its operators, whereas an arbitrary operator might simply return a FixedParameterOperator. The only assured property of the assembled operator is that it no longer depends on a Parameter.
Parameters
mu The Parameter for which to assemble the operator.

Returns
Parameter-independent, assembled Operator.

assemble_lincomb(operators, coefficients, name=None)
Try to assemble a linear combination of the given operators.
This method is called in the assemble method of LincombOperator on the first of its operator. If an assembly of the given linear combination is possible, e.g. the linear combination of the system matrices of the operators can be formed, then the assembled operator is returned. Otherwise, the method returns None to indicate that assembly is not possible.

Parameters
operators List of Operators whose linear combination is formed.
coefficients List of the corresponding linear coefficients.
name Name of the assembled operator.

Returns
The assembled Operator if assembly is possible, otherwise None.

projected_to_subbasis(dim_source=None, dim_range=None, name=None)
Project the operator to a subbasis.
The purpose of this method is to further project an operator that has been obtained through projected to subbases of the original projection bases, i.e.

\[ \text{op.projected(s_basis, r_basis, prod).projected_to_subbasis(dim_source, dim_range)} \]

should be the same as

\[ \text{op.projected(s_basis.copy(range(dim_source)), r_basis.copy(range(dim_range)), prod)} \]

For a NumpyMatrixOperator this amounts to extracting the upper-left (dim_range, dim_source) corner of the matrix it wraps.

Parameters
dim_source Dimension of the source subbasis.
dim_range Dimension of the range subbasis.

Returns
The projected Operator.
This module contains the implementation of pyMOR’s parameter handling facilities.

A `Parameter` in pyMOR is basically a `dict` of NumPy arrays. Each item of the dict is called a parameter component. The `ParameterType` of a `Parameter` is a dict of the shapes of the parameter components, i.e.

```python
mu.parameter_type['component'] == mu['component'].shape
```

Classes which represent mathematical objects depending on parameters, e.g. `Functions`, `Operators`, `Discretizations` derive from the `Parametric` mixin. Each `Parametric` object has a `parameter_type` attribute holding the `ParameterType` of the parameters the object’s `evaluate`, `apply`, `solve`, etc. methods expect. Note that the `ParameterType` of the given `Parameter` is allowed to actually be a superset (in the obvious sense) of the object’s `ParameterType`.

The `ParameterType` of a `Parametric` object is determined in its `__init__` method by calling `build_parameter_type` which computes the `ParameterType` as the union of the `ParameterTypes` of the objects given to the method. This way, an, e.g., `Operator` can inherit the `ParameterTypes` of the data functions it depends upon.

A `Parametric` object can have a `ParameterSpace` assigned to it by setting the `parameter_space` attribute. (The `ParameterType` of the space has to agree with the `ParameterType` of the object.) The `parse_parameter` method parses a user input according to the object’s `ParameterType` to make it a `Parameter` (e.g. if the `ParameterType` consists of a single one-dimensional component, the user can simply supply a list of numbers of the right length). Moreover, if given a `Parameter`, it checks whether the `Parameter` has an appropriate `ParameterType`. Thus `parse_parameter` should always be called by the implementor for any given parameter argument. The `local_parameter` method is used to extract the local parameter components of the given `Parameter` and performs some name mapping. (See the documentation of `build_parameter_type` for details.)

class pymor.parameters.base.Parameter(v)

    Bases: dict

    Class representing a parameter.

    A `Parameter` is simply a `dict` where each key is a string and each value is a NumPy array. We call an item of the dictionary a parameter component.

    A `Parameter` differs from an ordinary `dict` in the following ways:

    • It is ensured that each value is a NumPy array.
    • We overwrite `copy` to ensure that not only the `dict` but also the NumPy arrays are copied.
    • The `allclose` method allows to compare `Parameters` for equality in a mathematically meaningful way.
    • Each `Parameter` has a `sid` property providing a unique state id. (See `pymor.core.interfaces`.)
    • We override `__str__` to ensure alphanumerical ordering of the keys and more or less pretty printing of the values.
    • The `parameter_type` property can be used to obtain the `ParameterType` of the parameter.
    • Use `from_parameter_type` to construct a `Parameter` from a `ParameterType` and user supplied input.

Parameters

v Anything that `dict` accepts for the construction of a dictionary.

Methods
Attributes

| Parameter | parameter_type, sid |

**parameter_type**
The ParameterType of the Parameter.

**sid**
The state id of the Parameter. (See pymor.core.interfaces.)

**allclose** *(mu)*
Compare to Parameters using float_cmp_all.

**Parameters**

- **mu** The Parameter with which to compare.

**Returns**
True if both Parameters have the same ParameterType and all parameter components are almost equal, else False.

**classmethod from_parameter_type** *(mu, parameter_type=None)*
Takes a user input mu and interprets it as a Parameter according to the given ParameterType. Depending on the ParameterType, mu can be given as a Parameter, dict, tuple, list, array or scalar. For the exact details, please refer to the source code.

**Parameters**

- **mu** The user input which shall be interpreted as a Parameter.
- **parameter_type** The ParameterType w.r.t. which mu is to be interpreted.

**Returns**
The resulting Parameter.

** Raises**
ValueError is raised if mu cannot be interpreted as a Parameter of ParameterType parameter_type.

```python
class pymor.parameters.base.ParameterType(t)
Bases: dict

Class representing a parameter type.

A parameter type is simply a dictionary with strings as keys and tuples of natural numbers as values. The keys are the names of the parameter components and the tuples their expected shape. (Compare Parameter.)
```
Apart from checking the correct format of its values, the only difference between a ParameterType and an ordinary dict is, that ParameterType orders its keys alphabetically.

Parameters

If t is an object with a parameter_type attribute, a copy of this ParameterType is created. Otherwise, t can be anything from which a dict can be constructed.

Methods

ParameterType: clear, copy, fromkeys, items, iteritems, iterkeys, itervalues, keys, pop, popitem, update, values

dict: get, has_key, setdefault, viewitems, viewkeys, viewvalues, __contains__, __getitem__, __new__, __sizeof__

Attributes

<table>
<thead>
<tr>
<th>ParameterType</th>
<th>sid</th>
</tr>
</thead>
</table>

class pymor.parameters.base.Parametric

Bases: object

Mixin class for objects representing mathematical entities depending on a Parameter.

Each such object has a ParameterType stored in the parameter_type attribute, which should be set by the implementor during __init__ using the build_parameter_type method. Methods expecting the Parameter (typically evaluate, apply, solve, etc.) should accept an optional argument mu defaulting to None. This mu should then be fed into parse_parameter to obtain a Parameter of correct ParameterType from the (user supplied) input mu. The local parameter components (see build_parameter_type) can be extracted using local_type.

Methods

Parametric: build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

| Parametric | parameter_local_type, parameter_space, parameter_type, parametric |

parameter_type

The ParameterType of the Parameters the object expects or None, if the object actually is parameter independent.

parameter_local_type

The ParameterType of the parameter components which are introduced by the object itself and are not inherited by other objects it depends on. None if there are no such components. (See build_parameter_type)

parameter_space

ParameterSpace the parameters are expected to lie in or None.

parametric

True if the object really depends on a parameter, i.e. parameter_type is not None.
**build_parameter_type** (local_type=None, global_names=None, local_global=False, inherits=None, provides=None)

Builds the *ParameterType* of the object. Should be called by **__init__**.

The *ParameterType* of a *Parametric* object is determined by the parameter components the object itself requires for evaluation, and by the parameter components required by the objects the object depends upon for evaluation. We speak of local and inherited parameter components. The *ParameterType* of the local parameter components are provided via the *local_type* parameter, whereas the *Parametric* objects from which parameter components are inherited are provided as the *inherits* parameter.

Since the implementor does not necessarily know the future use of the object, a mapping between the names of the local parameter components and their intended global names (from the user perspective) can be provided via the *global_names* parameter. This mapping of names will be usually provided by the user when instantiating the class. (E.g. a *Function* evaluating \( x \to x^a \) could depend on a local parameter component named ‘base’, whereas the user wishes to name the component ‘decay_rate’.) If such a mapping is not desired, the *local_global* parameter must be set to **True**. (To later extract the local parameter components with their local names from a given *Parameter* use the *local_parameter* method.)

After the name mapping, all parameter components (local or inherited by one of the objects provided via *inherits*) with the same name are treated as identical and are thus required to have the same shapes. The object’s *ParameterType* is then made up by the shapes of all parameter components appearing.

Additionally components of the resulting *ParameterType* can be removed by specifying them via the *provides dict*. The idea is that the object itself may provide parameter components to the inherited objects which thus should not become part of the object’s own parameter type. (A typical application would be *InstationaryDiscretization*, which provides a time parameter component to its (time-dependent) operators during time-stepping.)

**Note:** As parameter components of the *ParameterTypes* of different objects are treated as equal if they have the same name, renaming a local parameter component is not merely a convenience feature but can also have a semantic meaning by identifying local parameter components with inherited ones.

**Parameters**

- **local_type** *ParameterType* of the local parameter components.
- **global_names** A *dict* of the form `{‘localname’: ‘globalname’, ...}` defining a name mapping specifying global parameter component names for each key of *local_type*. If **None** and *local_type* is not **None**, *local_global* must be set to **True**.
- **local_global** If **True**, treat the names of the local parameter components as global names of these components. In this case, *global_names* must be **None**.
- **inherits** Iterable where each entry is a *Parametric* object whose *ParameterType* shall become part of the built *ParameterType*.
- **provides** *Dict* of parameter component names and their shapes which are provided by the object itself to the objects in the *inherited* list. The parameter components listed here will, thus, not become part of the object’s *ParameterType*.

**local_parameter** (*mu*)

Extract the local parameter components with their local names from a given *Parameter*. See *build_parameter_type* for details.

**parse_parameter** (*mu*)

Interpret a user supplied parameter *mu* as a *Parameter*.  

---

### 4.1. pymor package

179
If \( \mu \) is not already a Parameter, \( \text{Parameter.from_parameter_type} \) is used, to make \( \mu \) a parameter of the correct ParameterType. If \( \mu \) is already a Parameter, it is checked if its ParameterType matches our own. (It is actually allowed that the ParameterType of \( \mu \) is a superset of our own ParameterType in the obvious sense.)

**Parameters**

\( \mu \) The input to parse as a Parameter.

**strip\_parameter\( (\mu) \)**

Remove all components of the Parameter \( \mu \) which are not part of the object’s ParameterType.

Otherwise \text{strip\_parameter} behaves like parse\_parameter.

This method is mainly useful for caching where the key should only contain the relevant parameter components.

---

**functionals module**

class \text{pymor.parameters.functionals.ExpressionParameterFunctional}(\text{expression, parameter\_type, name=None})

Bases: \text{pymor.parameters.functionals.GenericParameterFunctional}

Turns a Python expression given as a string into a ParameterFunctional.

Some NumPy arithmetic functions like ‘sin’, ‘log’, ‘min’ are supported. For a full list see the \text{functions} class attribute.

**Warning:** \text{eval} is used to evaluate the given expression. As a consequence, using this class with expression strings from untrusted sources will cause mayhem and destruction!

**Parameters**

\text{expression} The Python expression for the functional as a string.

\text{parameter\_type} The ParameterType of the Parameters the functional takes.

**Methods**

ExpressionParameterFunctional \_\_repr__

GenericParameter \_\_repr__

ParameterFunctional \_\_interface\_name

ImmutableInterface \_\_lock, unlock

BasicInterface add\_attributes, disable\_logging, enable\_logging, has\_interface\_name, implementor\_names, implementors, with\. __setattr__

Parametric build\_parameter\_type, local\_parameter, parse\_parameter, strip\_parameter

**Attributes**
ExpressionParameterFunctional

ImmutableInterface calculate_sid, sid, sid_failure, sid_ignore

BasicInterface locked, logger, logging_disabled, name, uid, with_arguments

Parametric parameter_local_type, parameter_space, parameter_type, parametric

__reduce__() helper for pickle

__repr__() <==> repr(x)

class pymor.parameters.functionals_GenericParameterFunctional(mapping, parameter_type, name=None)

Bases: pymor.parameters.interfaces.ParameterFunctionalInterface

A wrapper making an arbitrary Python function a ParameterFunctional

Note that a GenericParameterFunctional can only be dumps_function will be tried as a last resort. For this reason, it is usually preferable to use ExpressionParameterFunctional instead, which always can be serialized.

Parameters

parameter_type The ParameterType of the Parameters the functional takes.
mapping The function to wrap. The function has signature mapping(mu).
name The name of the functional.

Methods

class pymor.parameters.functionals_ProjectionParameterFunctional(component_name, component_shape, coordinates=None, name=None)

Bases: pymor.parameters.interfaces.ParameterFunctionalInterface

4.1. pymor package
ParameterFunctional returning a component of the given parameter.

Parameters

component_name  The name of the component to return.
component_shape  The shape of the component.
coordinates  If not None, return \texttt{mu[component_name][coordinates]} instead of \texttt{mu[component_name]}.
name  Name of the functional.

Methods

\texttt{ProjectionParameterFunctional}

\texttt{ParameterFunctionalInterface}

\texttt{ImmutableInterface}  \texttt{lock, unlock}

\texttt{BasicInterface}  \texttt{add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, setattr__}

\texttt{Parametric}  \texttt{build_parameter_type, local_parameter, parse_parameter, strip_parameter}

Attributes

\texttt{ImmutableInterface}  \texttt{calculate_sid, sid, sid_failure, sid_ignore}

\texttt{BasicInterface}  \texttt{locked, logger, logging_disabled, name, uid, with_arguments}

\texttt{Parametric}  \texttt{parameter_local_type, parameter_space, parameter_type, parametric}

\texttt{evaluate(mu=\texttt{None})}

Evaluate the functional for the given Parameter \texttt{mu}.
__call__(...) \Leftrightarrow \text{x(\ldots)}

\text{evaluate}(\mu=\text{None})

Evaluate the functional for the given Parameter $\mu$.

class pymor.parameters.interfaces.ParameterSpaceInterface
Bases: pymor.core.interfaces.ImmutableInterface

Interface for parameter spaces.

Methods

<table>
<thead>
<tr>
<th>Base Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterSpaceInterface, ImmutableInterface, unlock</td>
</tr>
<tr>
<td>BasicInterface, add_attributes, disable_logging, enable_logging,</td>
</tr>
<tr>
<td>has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Base Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterSpaceInterface, parameter_type</td>
</tr>
<tr>
<td>ImmutableInterface, calculate_sid, sid, sid_failure, sidignore</td>
</tr>
<tr>
<td>BasicInterface, locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

$\text{parameter_type}$

$\text{ParameterType}$ of the space.

$\text{contains}(\mu)$

$\text{True}$ if $\mu$ is contained in the space.

class pymor.parameters.spaces.CubicParameterSpace
Bases: pymor.parameters.interfaces.ParameterSpaceInterface

Simple ParameterSpace where each summand is an n-cube.

Parameters

$\text{parameter_type}$ The $\text{ParameterType}$ of the space.

$\text{minimum}$ The minimum for each matrix entry of each Parameter component. Must be $\text{None}$ if $\text{ranges}$ is not $\text{None}$.

$\text{maximum}$ The maximum for each matrix entry of each Parameter component. Must be $\text{None}$ if $\text{ranges}$ is not $\text{None}$.

$\text{ranges}$ dict whose keys agree with $\text{parameter_type}$ and whose values are tuples (min, max) specifying the minimum and maximum of each matrix entry of corresponding Parameter component. Must be $\text{None}$ if minimum and maximum are specified.

Methods
**CubicParameterSpace** contains **parse_parameter**, **sample_randomly**, **sample_uniformly**, **__repr__**, **__str__**.

**ImmutableInterface** lock, unlock

**BasicInterface** add_attributes, disable_logging, enable_logging, **has_interface_name**, implementor_names, implementors, **with_**, **__setattr__**

### Attributes

**ParameterSpaceInterface** parameter_type

**ImmutableInterface** calculate_sid, sid, sid_failure, sid_ignore

**BasicInterface** locked, logger, logging_disabled, name, uid, **with_arguments**

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

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

**contains**(mu)

*True if mu is contained in the space.*

**sample_randomly**(count=None, random_state=None, seed=None)

Iterator sampling random **Parameters** from the space.

**sample_uniformly**(counts)

Iterator sampling uniformly **Parameters** from the space.

---

**pymor.playground package**

**Subpackages**

**pymor.playground.demos package**

**Submodules**

**parabolic module**

**pymor.playground.demos.parabolic.parabolic_demo()**

---

**pymor.playground.functions package**

**Submodules**

**bitmap module**

**class** **pymor.playground.functions.bitmap.BitmapFunction**(filename, bounding_box=[[0.0, 0.0], [1.0, 1.0]], range=[0.0, 1.0])

*Define a 2D function via a grayscale image.*

---

184 Chapter 4. API Documentation
filename  Path of the image representing the function.
bounding_box  Lower left and upper right coordinates of the domain of the function.
range  A pixel of value $p$ is mapped to $(p / 255.) \times \text{range}[1] + \text{range}[0]$.

### Methods

<table>
<thead>
<tr>
<th>BitmapFunction</th>
<th>evaluate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionBase</td>
<td><strong>add</strong>, <strong>mul</strong>, <strong>neg</strong>, <strong>radd</strong>, <strong>rmul</strong>, <strong>sub</strong></td>
</tr>
<tr>
<td>FunctionInterface</td>
<td>unlock</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with_, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>BitmapFunction</th>
<th>dim_domain, shape_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**evaluate** $(x, mu=None)$

Evaluate the function for given argument and Parameter.

**expression_function module**

**class** `pymor.playground.functions.expression_function.ExpressionFunction(expressions, variables='x y z')`

<table>
<thead>
<tr>
<th>Bases:</th>
<th>pymor.core.interfaces.BasicInterface</th>
</tr>
</thead>
</table>

**Methods**

<table>
<thead>
<tr>
<th>ExpressionFunction</th>
<th><strong>call</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with_, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

**Attributes**

| BasicInterface | locked, logger, logging_disabled, name, uid, with_arguments |

**__call__ (...) <==> x(...)**

**pymor.playground.grids package**

**Submodules**

**4.1. pymor package**
gmsh module

class `pymor.playground.grids.gmsh.GmshGrid(gmsh_file)`

    Bases: `pymor.grids.interfaces.AffineGridInterface`

Methods

<table>
<thead>
<tr>
<th>GmshGrid</th>
<th>embeddings, size, subentities, test_instances, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>AffineGridInterface</td>
<td>centers, diameters, integration_elements,</td>
</tr>
<tr>
<td></td>
<td>jacobian_inverse_transposed, quadrature_points,</td>
</tr>
<tr>
<td></td>
<td>unit_outer_normals, volumes, volumes_inverse</td>
</tr>
<tr>
<td>ConformalTopologicalGridInterface</td>
<td>boundary_mask, neighbours, superentities,</td>
</tr>
<tr>
<td></td>
<td>superentity_indices</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>lock, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging,</td>
</tr>
<tr>
<td></td>
<td>has_interface_name, implementor_names, implementors,</td>
</tr>
<tr>
<td></td>
<td>with_, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>GmshGrid</th>
<th>dim, dim_outer, reference_element</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheableInterface</td>
<td>cache_region</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid,</td>
</tr>
<tr>
<td></td>
<td>with_arguments</td>
</tr>
</tbody>
</table>

__str__() ↔ str(x)

embeddings (codim=0)

    Returns tuple (A, B) where A[e] and B[e] are the linear part and the translation part of the map from the reference element of e to e.

    For codim > 0, we provide a default implementation by taking the embedding of the codim-1 parent entity e_0 of e with lowest global index and composing it with the subentity_embedding of e into e_0 determined by the reference element.

size (codim=0)

    The number of entities of codimension codim.

subentities (codim=0, subentity_codim=None)

    retval[e,s] is the global index of the s-th codim-subentity_codim subentity of the codim-codim entity with global index e.

    Only subentities(codim, codim+1) has to be implemented; a default implementation is provided which evaluates subentities(codim, subentity_codim) by computing the transitive closure of subentities(codim, codim+1).

class `pymor.playground.grids.gmsh.GmshParseError`

    Bases: exceptions.Exception

Methods

| Exception              | __new__ |

Attributes
pymor.playground.grids.gmsh.parse_gmsh_file(f)

pymor.playground.la package

Submodules

blockvectorarray module

class pymor.playground.la.blockvectorarray.BlockVectorArray(blocks, copy=False)
Bases: pymor.la.interfaces.VectorArrayInterface

VectorArray implementation

Methods

<table>
<thead>
<tr>
<th>BlockVectorArray</th>
<th>almost_equal, amax, append, axpy, block, components, copy, dot, 11_norm, 12_norm, lincomb, make_array, remove, replace, scal, sup_norm, <strong>len</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorArray</td>
<td>Interface</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, with_, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>BlockVectorArray</th>
<th>data, dim, num_blocks, subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorArray</td>
<td>space</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
</tbody>
</table>

__len__() The number of vectors in the array.

almost_equal (other, ind=None, o_ind=None, rtol=None, atol=None) Check vectors for equality.

Equality of two vectors should be defined as in pymor.tools.float_cmp_all.

The subtypes of self and other have to agree. If the length of self (ind) resp. other (o_ind) is 1, the one specified vector is compared to all vectors of the other summand. Otherwise the length of ind and o_ind have to agree.

Parameters

other A VectorArray containing the vectors to compare with.
ind Indices of the vectors that are to be compared (see class documentation).
o_ind Indices of the vectors in other that are to be compared (see class documentation).
rtol See pymor.tools.float_cmp_all
**atol** See pymor.tools.float_cmp_all

Returns
NumPy array of the truth values of the comparison.

**amax** *(ind=None)*
The maximum absolute value of the vectors contained in the array.

Parameters

*ind* Indices of the vectors whose maximum absolute value is to be calculated (see class documentation).

Returns

*max_ind* NumPy array containing for each vector an index at which the maximum is attained.

*max_val* NumPy array containing for each vector the maximum absolute value of its components.

**append** *(other, o_ind=None, remove_from_other=False)*
Append vectors to the array.

Parameters

*other* A VectorArray containing the vectors to be appended.

*o_ind* Indices of the vectors that are to be appended (see class documentation).

*remove_from_other* If True, the appended vectors are removed from *other*. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.

**axpy** *(alpha, x, ind=None, x_ind=None)*
BLAS AXPY operation.

This method forms the sum

\[
\text{self}[\text{ind}] = \alpha \times \text{x}[\text{x_ind}] + \text{self}[\text{ind}]
\]

The subtypes of *self* and *x* have to agree. If the length of *x* (*x_ind*) is 1, the same *x* vector is used for all vectors in *self*. Otherwise, the lengths of *self* (*ind*) and *x* (*x_ind*) have to agree. If *alpha* is a scalar, each *x* vector is multiplied with the same factor *alpha*. Otherwise, *alpha* has to be a one-dimensional NumPy array of the same length as *self* (*ind*) containing the coefficients for each *x* vector.

Parameters

*alpha* The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in *x* are multiplied.

*x* A VectorArray containing the *x*-summands.

*ind* Indices of the vectors of *self* that are to be added (see class documentation). Repeated indices are forbidden.

*x_ind* Indices of the vectors in *x* that are to be added (see class documentation). Repeated indices are allowed.

**block** *(ind)*
Returns a copy of each block (no slicing).
**components** *(component_indices, ind=None)*

Extract components of the vectors contained in the array.

**Parameters**

- **component_indices** List or 1D NumPy array of indices of the vector components that are to be returned.
- **ind** Indices of the vectors whose components are to be calculated (see class documentation).

**Returns**

A NumPy array result such that result[i, j] is the component_indices[j]-th component of the ind[i]-th vector of the array.

**copy** *(ind=None)*

Returns a copy of a subarray.

**Parameters**

- **ind** Indices of the vectors that are to be copied (see class documentation).

**Returns**

A copy of the VectorArray.

**dot** *(other, pairwise, ind=None, o_ind=None)*

Returns the scalar products between VectorArray elements.

**Parameters**

- **other** A VectorArray containing the second factors.
- **pairwise** See return value documentation.
- **ind** Indices of the vectors whose scalar products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in other whose scalar products are to be taken (see class documentation).

**Returns**

If pairwise is True, returns a NumPy array result such that:

```
result[i] = (self[ind][i], other[o_ind][i]).
```

If pairwise is False, returns a NumPy array result such that:

```
result[i, j] = (self[ind][i], other[o_ind][j]).
```

**l1_norm** *(ind=None)*

The l1-norms of the vectors contained in the array.

**Parameters**

- **ind** Indices of the vectors whose norms are to be calculated (see class documentation).

**Returns**

A NumPy array result such that result[i] contains the norm of self[ind][i].
12_norm(ind=None)
The l2-norms of the vectors contained in the array.

Parameters
ind Indices of the vectors whose norms are to be calculated (see class documentation).

Returns
A NumPy array result such that result[i] contains the norm of self[ind][i].

lincomb(coefficients, ind=None)
Returns linear combinations of the vectors contained in the array.

Parameters
coefficients A NumPy array of dimension 1 or 2 containing the linear coefficients. coefficients.shape[-1] has to agree with len(self).
ind Indices of the vectors which are linear combined (see class documentation).

Returns
A VectorArray result such that:
result[i] = self[j] * coefficients[i,j]
in case coefficients is of dimension 2, otherwise len(result) == 1 and

remove(ind=None)
Remove vectors from the array.

Parameters
ind Indices of the vectors that are to be removed (see class documentation).

replace(other, ind=None, o_ind=None, remove_from_other=False)
Replace vectors of the array.

Parameters
other A VectorArray containing the replacement vectors.
ind Indices of the vectors that are to be replaced (see class documentation). Repeated indices are forbidden.
o_ind Indices of the replacement vectors (see class documentation). len(ind) has to agree with len(o_ind). Repeated indices are allowed.
remove_from_other If True, the new vectors are removed from other. For list-like implementations this can be used to prevent unnecessary copies of the involved vectors.

scal(alpha, ind=None)
BLAS SCAL operation (in-place scalar multiplication).
This method calculates
self[ind] = alpha*self[ind]

If \( \alpha \) is a scalar, each vector is multiplied by this scalar. Otherwise, \( \alpha \) has to be a one-dimensional NumPy array of the same length as \( \text{self} \) (\( \text{ind} \)) containing the factors for each vector.

Parameters

- **alpha** The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in \( \text{self} \) are multiplied.
- **ind** Indices of the vectors of \( \text{self} \) that are to be scaled (see class documentation). Repeated indices are forbidden.

**sup_norm** \((\text{ind} = \text{None})\)

The l-infinity–norms of the vectors contained in the array.

Parameters

- **ind** Indices of the vectors whose norms are to be calculated (see class documentation).

Returns

A NumPy array result such that result[i] contains the norm of self[ind][i].

Submodules

progressbar module

pymor.reductors package

Submodules

**basic module**

**class** `pymor.reductors.basic.GenericRBReconstructor(RB)`

Bases: `pymor.core.interfaces.BasicInterface`

Simple reconstructor forming linear combinations with a reduced basis.

**Methods**

- `reconstruct`, `restricted_to_subbasis`
- `add_attributes`, `disable_logging`, `enable_logging`,
  `has_interface_name`, `implementor_names`, `implementors`, `lock`,
  `unlock`, `with`, `__getattr__`, `__setattr__`

**Attributes**

- `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`

**reconstruct** \((U)\)

Reconstruct high-dimensional vector from reduced vector \( U \).
restricted_to_subbasis(dim)
See projected_to_subbasis.

class pymor.reductors.basic.SubbasisReconstructor(dim, dim_subbasis, old_reconstructor=None)
Bases: pymor.core.interfaces.BasicInterface
Returned by reduce_to_subbasis.

Methods

Attributes

reconstruct(U)
Reconstruct high-dimensional vector from reduced vector U.

def reduce_generic_rb(discretization, RB, operator_product=None, vector_product=None, disable_caching=True, extends=None)

Generic reduced basis reductor.
Replaces each Operator of the given Discretization with the projection onto the span of the given reduced basis.

Parameters
discretization The Discretization which is to be reduced.
RB VectorArray containing the reduced basis on which to project.
operator_product Scalar product for the projection of the Operators. (See projected.)
vector_product Scalar product for the projection of vector-like Operators. (A typical vector-like operator would be the initial_data Operator of an InstationaryDiscretization holding the initial data of a Cauchy problem.)
disable_caching If True, caching of solutions is disabled for the reduced Discretization.
extends Set by greedy to the result of the last reduction in case the basis extension was hierarchic. Currently ignored by this reductor.

Returns
rd The reduced Discretization.
rc The reconstructor providing a reconstruct(U) method which reconstructs high-dimensional solutions from solutions U of the reduced Discretization.
reduction_data Additional data produced by the reduction process. Currently empty.
Further reduce a \texttt{Discretization} to the subbasis formed by the first \texttt{dim} basis vectors.

This is achieved by calling \texttt{projected_to_subbasis} for each operator of the given \texttt{Discretization}. Additionally, if a reconstructor for the \texttt{Discretization} is provided, its \texttt{restricted_to_subbasis} method is also called to obtain a reconstructor for the further reduced \texttt{Discretization}. Otherwise \texttt{SubbasisReconstructor} is used (which will be less efficient).

**Parameters**

- \texttt{discretization} The \texttt{Discretization} to further reduce.
- \texttt{dim} The dimension of the subbasis.
- \texttt{reconstructor} Reconstructor for \texttt{discretization} or \texttt{None}.

**Returns**

- \texttt{rd} The further reduced \texttt{Discretization}.
- \texttt{rc} Reconstructor for \texttt{rd}.

### linear module

**Class** \texttt{pymor.reductors.linear.StationaryAffineLinearReducedEstimator} \((\text{estimator_matrix}, \text{coercivity_estimator})\)

Bases: \texttt{pymor.core.interfaces.ImmutableInterface}

Instatiated by \texttt{reduce_stationary_affine_linear}.

Not to be used directly.

**Methods**

- \texttt{StationaryAffineLinearReducedEstimator.estimate, RestrictedToSubbasis}
- \texttt{ImmutableInterface.lock, unlock}
- \texttt{BasicInterface.add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with, \_\_setattr\_}

**Attributes**

- \texttt{ImmutableInterface.calculate_sid, sid, sid_failure, sid_ignore}
- \texttt{BasicInterface.locked, logger, logging_disabled, name, uid, with_arguments}

Reducer for linear \texttt{StationaryDiscretizations} with affinely decomposed operator and rhs.

**Note:** The reducer \texttt{reduce_stationary_coercive} can be used for arbitrary coercive \texttt{StationaryDiscretizations} and offers an improved error estimator with better numerical stability.
This reductor uses `reduce_generic_rb` for the actual RB-projection. The only addition is an error estimator. The estimator evaluates the norm of the residual with respect to a given inner product.

**Parameters**

- **discretization** The `Discretization` which is to be reduced.
- **RB** `VectorArray` containing the reduced basis on which to project.
- **error_product** Scalar product `Operator` used to calculate Riesz representative of the residual. If `None`, the Euclidean product is used.
- **coercivity_estimator** `None` or a `ParameterFunctional` returning a lower bound for the coercivity constant of the given problem. Note that the computed error estimate is only guaranteed to be an upper bound for the error when an appropriate coercivity estimate is specified.
- **disable_caching** If `True`, caching of solutions is disabled for the reduced `Discretization`.
- **extends** Set by `greedy` to the result of the last reduction in case the basis extension was `hierarchic`. Used to prevent re-computation of Riesz representatives already obtained from previous reductions.

**Returns**

- **rd** The reduced `Discretization`.
- **rc** The reconstructor providing a `reconstruct(U)` method which reconstructs high-dimensional solutions from solutions `U` of the reduced `Discretization`.
- **reduction_data** Additional data produced by the reduction process. In this case the computed Riesz representatives. (Compare the `extends` parameter.)

---

**residual module**

**class** `pymor.reductors.residual.NonProjectedReconstructor` *(product)*

Bases: `pymor.core.interfaces.ImmutableInterface`

Returned by `reduce_residual`.

Not to be used directly.

**Methods**

<table>
<thead>
<tr>
<th>NonProjectedReconstructor</th>
<th><code>reconstruct</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td><code>lock</code>, <code>unlock</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>add_attributes</code>, <code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>with_.</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th><code>calculate_sid</code>, <code>sid</code>, <code>sid_failure</code>, <code>sid_ignore</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td><code>locked</code>, <code>logger</code>, <code>logging_disabled</code>, <code>name</code>, <code>uid</code>, <code>with_arguments</code></td>
</tr>
</tbody>
</table>
class pymor.reductors.residual.NonProjectedResidualOperator (operator, functional, product)

Bases: pymor.reductors.residual.ResidualOperator

Returned by reduce_residual.

Not to be used directly.

Methods

<table>
<thead>
<tr>
<th>NonProjectedResidualOperator</th>
<th>apply, projected_to_subbasis</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_adjoint, apply_inverse, as_vector, assemble, jacobian, lincomb, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>assemble_lincomb, restricted</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>deck, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>add_attributes, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, with, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>OperatorBase</th>
<th>invert_options</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>linear, range, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>calculate_sid, sid, sid_failure, sid_ignore</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid, with_arguments</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

apply (U, ind=None, mu=None)

Apply the operator.

Parameters

- U VectorArray of vectors to which the operator is applied.
- ind The indices of the vectors in U to which the operator shall be applied. (See the VectorArray documentation for further details.)
- mu The Parameter for which to evaluate the operator.

Returns

VectorArray of the operator evaluations.

class pymor.reductors.residual.ResidualOperator (operator, functional, name=None)

Bases: pymor.operators.basic.OperatorBase

Returned by reduce_residual.

Methods
residualOperator

apply, projected_to_subbasis

OperatorBase
apply2, apply_adjoint, apply_inverse, as_vector, assemble,
jacobian, lincomb, projected, __add__, __mul__, __radd__, __str__

OperatorInterface
assemble_lincomb, restricted

ImmutableInterface
lock, unlock

BasicInterface
add_attributes, disable_logging, enable_logging,
has_interface_name, implementor_names, implementors, with_,
__setattr__

Parametric
build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes

OperatorBase
invert_options

OperatorInterface
linear, range, source

ImmutableInterface
calculate_sid, sid, sid_failure, sid_ignore

BasicInterface
locked, logger, logging_disabled, name, uid, with_arguments

Parametric
parameter_local_type, parameter_space, parameter_type,
parametric

apply \((U, \ ind=\text{None}, \ mu=\text{None})\)

Apply the operator.

Parameters

\(U\) VectorArray of vectors to which the operator is applied.

\(\ ind\) The indices of the vectors in \(U\) to which the operator shall be applied. (See the VectorArray documentation for further details.)

\(\ mu\) The Parameter for which to evaluate the operator.

Returns

VectorArray of the operator evaluations.

pymor.reductors.residual.reduce_residual \(\text{operator, functional=\text{None}, RB=\text{None}, product=\text{None}, extends=\text{None}}\)

Generic reduced basis residual reductor.

Given an operator and a functional, the concatenation of residual operator with the Riesz isomorphism is given by:

\[
\text{riesz_residual.apply}(U, \mu) = \text{product.apply_inverse}(\text{operator.apply}(U, \mu) - \text{functional.as_vector}(\mu))
\]

This reductor determines a low-dimensional subspace of image of a reduced basis space under \(\text{riesz_residual}\), computes an orthonormal basis \(\text{residual_range}\) of this range spaces and then returns the Petrov-Galerkin projection

\[
\text{projected_riesz_residual} = \text{riesz_residual.projected(\text{source_basis}=\text{RB}, \text{range_basis}=\text{residual_range})}
\]

of the \(\text{riesz_residual}\) operator. Given an reduced basis coefficient vector \(u\), the dual norm of the residual can then be computed as

196 Chapter 4. API Documentation
Moreover, a `reconstructor` is provided such that

```
reconstructor.reconstruct(projected_riesz_residual.apply(u, mu))
== riesz_residual.apply(RB.lincomb(u), mu)
```

### Parameters

- **operator**: See definition of `riesz_residual`.
- **functional**: See definition of `riesz_residual`. If `None`, zero right-hand side is assumed.
- **RB**: `VectorArray` containing a basis of the reduced space onto which to project.
- **product**: Scalar product `Operator` w.r.t. which to compute the Riesz representatives.
- **extends**: Set by `greedy` to the result of the last reduction in case the basis extension was `hierarchic`. Used to prevent re-computation of `residual_range` basis vectors already obtained from previous reductions.

### Returns

- **projected_riesz_residual**: See above.
- **reconstructor**: See above.
- **reduction_data**: Additional data produced by the reduction process. (Compare the `extends` parameter.)

---

**stationary module**

class pymor.reductors.stationary.StationaryCoerciveEstimator(residual, residual_range_dims, coercivity_estimator)

Bases: `pymor.core.interfaces.ImmutableInterface`

Instatiated by `reduce_stationary_coercive`.

Not to be used directly.

**Methods**

- `add_attributes`, `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `with_`, `__setattr__`

**Attributes**

- `calculate_sid`, `sid`, `sid_failure`, `sid_ignore`
- `locked`, `logger`, `logging_disabled`, `name`, `uid`, `with_arguments`
Reductor for `StationaryDiscretizations` with coercive operator.

This reductor uses `reduce_generic_rb` for the actual RB-projection. The only addition is an error estimator. The estimator evaluates the dual norm of the residual with respect to a given inner product. We use `reduce_residual` for improved numerical stability. (See “A. Buhr, C. Engwer, M. Ohlberger, S. Rave, A Numerically Stable A Posteriori Error Estimator for Reduced Basis Approximations of Elliptic Equations, Proceedings of the 11th World Congress on Computational Mechanics, 2014.”)

### Parameters

- **discretization** The `Discretization` which is to be reduced.
- **RB** `VectorArray` containing the reduced basis on which to project.
- **error_product** Scalar product `Operator` used to calculate Riesz representative of the residual. If `None`, the Euclidean product is used.
- **coercivity_estimator** `None` or a `ParameterFunctional` returning a lower bound for the coercivity constant of the given problem. Note that the computed error estimate is only guaranteed to be an upper bound for the error when an appropriate coercivity estimate is specified.
- **disable_caching** If `True`, caching of solutions is disabled for the reduced `Discretization`.
- **extends** Set by `greedy` to the result of the last reduction in case the basis extension was `hierarchic`. Used to prevent re-computation of residual range basis vectors already obtained from previous reductions.

### Returns

- **rd** The reduced `Discretization`.
- **rc** The reconstructor providing a `reconstruct(U)` method which reconstructs high-dimensional solutions from solutions `U` of the reduced `Discretization`.
- **reduction_data** Additional data produced by the reduction process. (Compare the `extends` parameter.)

---

**pymor.tools package**

**Submodules**

- **arguments module**
  - `pymor.tools.arguments.method_arguments(func)`
    - Returns the names of the arguments of a given method (without `self`).

- **floatcmp module**
  - `pymor.tools.floatcmp.float_cmp(x, y, rtol=3.552713678800501e-15, atol=3.552713678800501e-15)`
    - Compare `x` and `y` component-wise for almost equality.
    - For scalars we define almost equality as
float_cmp(x, y) <=> |x - y| <= atol + |y|*rtol

Note: Numpy’s allclose method uses the same definition but treats arrays containing infinities as close if the infinities are at the same places and all other entries are close. In our definition, arrays containing infinities can never be close which seems more appropriate in most cases.

Parameters
x, y NumPy arrays to be compared. Have to be broadcastable to the same shape.
rtol The relative tolerance.
atol The absolute tolerance.

Defaults
rtol, atol (see pymor.core.defaults)

pymor.tools.floatcmp.float_cmp_all(x, y, rtol=None, atol=None)
Compare x and y for almost equality.
Returns True if all components of x are almost equal to the corresponding components of y.
See float_cmp.

frozendict module
class pymor.tools.frozendict.FrozenDict(*args, **kwargs)
Bases: dict

An immutable dictionary.

Methods
dict copy, fromkeys, get, has_key, items, iteritems, iterkeys, itervalues, keys, values, viewitems, viewkeys, viewvalues, __contains__, __getitem__, __sizeof__

Attributes
FrozenDict clear, pop, popitem, setdefault, update

inplace module
pymor.tools.inplace.iadd_masked(U, V, U_ind)
Indexed, masked in-place addition.
This is the same as
U[U_ind] += V

with two exceptions:
1. Negative indices are skipped.
2. If the same index is repeated, all additions are performed, not only the last one.

\[
\text{pymor.tools.inplace.\texttt{isub_masked}}(U, V, U_{\text{ind}})
\]
Indexed, masked in-place subtraction.
This is the same as
\[
U[U_{\text{ind}}] -= V
\]

with two exceptions:
1. Negative indices are skipped.
2. If the same index is repeated, all subtractions are performed, not only the last one.

\[
\text{pprint module}
\]
\[
\text{pymor.tools.pprint.\texttt{format_array}}(\text{array}, \text{compact\_print=False})
\]
Creates a formatted string representation of a NumPy array.

Parameters

compact\_print If True, return a shorter version of string representation.

Returns

The string representation.

Defaults

compact\_print (see pymor.core.defaults)

\[
\text{quadratures module}
\]
\[
\text{class pymor.tools.quadratures.\texttt{GaussQuadratures}}
\]
Bases: object
Gauss quadrature on the interval [0, 1]

Methods

\[
\text{GaussQuadratures | iter\_quadrature, maxpoints, quadrature}
\]

Attributes

\[
\text{GaussQuadratures | a, order\_map, orders, points, weights}
\]

\[
\text{classmethod iter\_quadrature (order=None, npoints=None)}
\]
iterates over a quadrature tuple wise

\[
\text{classmethod quadrature (order=None, npoints=None)}
\]
returns tuple (P, W) where P is an array of Gauss points with corresponding weights W for the given integration order “order” or with “npoints” integration points
random module

```python
pyMOR.tools.random.new_random_state(seed=42)
```

Returns a new NumPy RandomState.

**Parameters**

- `seed`: Seed to use for initializing the random state.

**Returns**

New RandomState object.

**Defaults**

- `seed` (see `pymor.core.defaults`)

relations module

```python
pyMOR.tools.relations.inverse_relation(R, size_rhs=None, with_indices=False)
```

Computes the inverse relation of a relation.

If \( r \) is a relation, then the inverse relation \( r_i \) is defined by

\[
x r i y \iff y r x
\]

**Parameters**

- `R`: 2D NumPy array of integers representing a relation \( r \) on the natural numbers via
  \[
  x r y \iff (x < R.size[0] \text{ and } y \in R[x])
  \]

  Rows of \( R \) which are to short are padded with -1.

- `size_rhs`: Can be provided for speedup. Has to be greater than \( R.max() \).

- `with_indices`: If True, also return the matrix \( RINV \).

**Returns**

- `RINV`: 2D NumPy array representation of the inverse relation.
- `RINVI`: NumPy array such that for \( RINV[i, j] \neq -1 \):
  \[
  R[RINV[i, j], RINVI[i, j]] = i.
  \]

Only returned if `with_indices` is True.

timing module

```python
class pyMOR.tools.timing.Timer(section, log=<logging.Logger object>)
```

Bases: object

You can use me as a context manager, plain instance or decorator to time execution of a code scope:
with Timer() as timer:
    do_some_stuff()
    do more stuff()
#outputs time in (s)

### OR ###

@timing.Timer('name', logging.debug)
def function(*args):
    do_stuff

function(1)
#outputs time in (s) to logging.debug

### OR ###

timer = timing.Timer()
timer.start()
do_stuff()
timer.stop()
print(timer.dt)

Methods

| Timer | start, stop |

pymor.tools.timing.busywait(amount)

vtkio module

pymor.tools.vtkio._write_meta_file(filename_base, steps, fn_tpl)
  Outputs a collection file for a series of vtu files
  This DOES NOT WORK for the currently used legacy vtk format below

pymor.tools.vtkio.write_vtk(grid, data, filename_base, codim=2, binary vtk=True, last_step=None)
  Output grid-associated data in (legacy) vtk format

Parameters

grid a Grid with triangular or rectilinear reference element

data VectorArrayInterface instance with either cell (ie one datapoint per codim 0 entity) or vertex (ie one datapoint per codim 2 entity) data in each array element

filename_base common component for output files in timeseries

last_step if set must be <= len(data) to restrict output of timeseries

4.1.2 Submodules

basic module

This module imports some commonly used methods and classes.
You can use `from pymor.basic import *` in interactive session to have the most important parts of pyMOR directly available.

`version module`
5.1 pymordemos package

5.1.1 Submodules

analyze_pickle module

Analyze pickled data demo.


This demo loads a pickled reduced discretization, solves for random parameters, estimates the reduction errors and then visualizes these estimates. If the detailed discretization and the reconstructor are also provided, the estimated error is visualized in comparison to the real reduction error.

The needed data files are created by the thermal block demo, by setting the ‘--pickle’ option.

Arguments: REDUCED_DATA File containing the pickled reduced discretization. SAMPLES Number of samples to test with.

Options:

--detailed=DETAILED_DATA File containing the high-dimensional discretization and the reconstructor.

--error-norm=NORM Name of norm in which to compute the errors.

--ndim=NDIM Number of reduced basis dimensions for which to estimate the error.

pymordemos.analyze_pickle.analyze_pickle_convergence(args)

pymordemos.analyze_pickle.analyze_pickle_demo(args)

pymordemos.analyze_pickle.analyze_pickle_histogram(args)
pyMOR, Release 0.3.2

burgers module

Burgers demo.
Solves a two-dimensional Burgers-type equation. See pymor.analyticalproblems.burgers for more details.

Usage:
```
```

Arguments: EXP Exponent

Options:
```
--grid=NI              Use grid with (2*NI)*NI elements [default: 60].
--grid-type=TYPE      Type of grid to use (rect, tria) [default: rect].
--initial-data=TYPE    Select the initial data (sin, bump) [default: sin]
--lxf-lambda=VALUE    Parameter lambda in Lax-Friedrichs flux [default: 1].
--nt=COUNT            Number of time steps [default: 100].
--not-periodic        Solve with dirichlet boundary conditions on left and bottom boundary.
--num-flux=FLUX       Numerical flux to use (lax_friedrichs, engquist_osh) [default: lax_friedrichs].
-h, --help            Show this message.
--vx=XSPEED           Speed in x-direction [default: 1].
--vy=YSPEED           Speed in y-direction [default: 1].
```

pymordemos.burgers.burgers_demo(args)

burgers_ei module

Burgers with EI demo.
Model order reduction of a two-dimensional Burgers-type equation (see pymor.analyticalproblems.burgers) using the reduced basis method with empirical operator interpolation.

Usage: burgers_ei.py [options] EXP_MIN EXP_MAX EI_SNAPSHOTS EISIZE SNAPSHOTS RBSIZE

Arguments: EXP_MIN Minimal exponent
```
EXP_MAX Maximal exponent
EI_SNAPSHOTS Number of snapshots for empirical interpolation.
EISIZE Number of interpolation DOFs.
SNAPSHOTS Number of snapshots for basis generation.
RBSIZE Size of the reduced basis
```

Options:
```
--grid=NI              Use grid with (2*NI)*NI elements [default: 60].
--grid-type=TYPE      Type of grid to use (rect, tria) [default: rect].
--initial-data=TYPE    Select the initial data (sin, bump) [default: sin]
--lxf-lambda=VALUE  Parameter lambda in Lax-Friedrichs flux [default: 1].
--not-periodic  Solve with dirichlet boundary conditions on left and bottom boundary.
--nt=COUNT  Number of time steps [default: 100].
--num-flux=FLUX  Numerical flux to use (lax_friedrichs, engquist_osher) [default: lax_friedrichs].
-h, --help  Show this message.
-p, --plot-err  Plot error.
--plot-ei-err  Plot empirical interpolation error.
--plot-error-landscape  Calculate and show plot of reduction error vs. basis sizes.
--plot-error-landscape-N=COUNT  Number of basis sizes to test [default: 10]
--plot-error-landscape-M=COUNT  Number of collateral basis sizes to test [default: 10]
--plot-solutions  Plot some example solutions.
--test=COUNT  Use COUNT snapshots for stochastic error estimation [default: 10].
--vx=XSpeed  Speed in x-direction [default: 1].
--vy=YSpeed  Speed in y-direction [default: 1].

pymordemos.burgers_ei.burgers_demo(args)

elliptic module

Simple demonstration of solving the Poisson equation in 2D using pyMOR’s builtin discretizations.

Usage: elliptic.py [-fv] [-rect] PROBLEM-NUMBER DIRICHLET-NUMBER NEUMANN-NUMBER NEUMANN-COUNT

Arguments: PROBLEM-NUMBER {0,1}, selects the problem to solve
DIRICHLET-NUMBER {0,1,2}, selects the Dirichlet data function
NEUMANN-NUMBER {0,1}, selects the Neumann data function
NEUMANN-COUNT 0: no neumann boundary 1: right edge is neumann boundary 2: right+top edges are neumann boundary 3: right+top+bottom edges are neumann boundary

Options:
-h, --help  Show this message.
--fv  Use finite volume discretization instead of finite elements.
--rect  Use RectGrid instead of TriaGrid.

pymordemos.elliptic.elliptic_demo(args)

5.1. pymordemos package
**elliptic2 module**

Simple demonstration of solving the Poisson equation in 2D using pyMOR’s built-in discretizations.

**Usage:** elliptic2.py [-fv] PROBLEM-NUMBER N

**Arguments:** PROBLEM-NUMBER {0,1}, selects the problem to solve

N Triangle count per direction

**Options:**

- `-h, --help` Show this message.

- `--fv` Use finite volume discretization instead of finite elements.

```
pymordemos.elliptic2.elliptic2_demo(args)
```

**elliptic_oned module**

Proof of concept for solving the Poisson equation in 1D using linear finite elements and our grid interface

**Usage:** elliptic_oned.py [-fv] PROBLEM-NUMBER N

**Arguments:** PROBLEM-NUMBER {0,1}, selects the problem to solve

N Grid interval count

**Options:**

- `-h, --help` Show this message.

- `--fv` Use finite volume discretization instead of finite elements.

```
pymordemos.elliptic_oned.elliptic_oned_demo(args)
```

**thermalblock module**

Thermalblock demo.

**Usage:**

```
```

**Arguments:**

- XBLOCKS Number of blocks in x direction.
- YBLOCKS Number of blocks in y direction.
- SNAPSHOTS Number of snapshots for basis generation per component. In total SNAPSHOTS^(XBLOCKS * YBLOCKS).
- RBSIZE Size of the reduced basis

**Options:**

- `-e, --with-estimator` Use error estimator.

- `--estimator-norm=NORM` Norm (trivial, h1) in which to calculate the residual [default: h1].

```
Chapter 5. Demo Applications
```
--extension-alg=ALG  Basis extension algorithm (trivial, gram_schmidt, h1_gram_schmidt) to be used [default: h1_gram_schmidt].
--grid=NI  Use grid with 2*NI*NI elements [default: 100].
-h, --help  Show this message.
--pickle=PREFIX  Pickle reduced discretization, as well as reconstructor and high-dimensional discretization to files with this prefix.
-p, --plot-err  Plot error.
--plot-solutions  Plot some example solutions.
--reductor=RED  Reductor (error estimator) to choose (traditional, residual_basis) [default: residual_basis]
--test=COUNT  Use COUNT snapshots for stochastic error estimation [default: 10].

pymordemos.thermalblock.thermalblock_demo (args)

thermalblock_gui module

Thermalblock with GUI demo

Usage:


Arguments:

XBLOCKS Number of blocks in x direction.
YBLOCKS Number of blocks in y direction.
SNAPSHOTS Number of snapshots for basis generation per component. In total SNAPSHOTS^(XBLOCKS * YBLOCKS).
RBSIZE Size of the reduced basis

Options:

--estimator-norm=NORM  Norm (trivial, h1) in which to calculate the residual [default: h1].
--grid=NI  Use grid with 2*NI*NI elements [default: 60].
--testing  load the gui and exit right away (for functional testing)
-h, --help  Show this message.

class pymordemos.thermalblock_gui.AllPanel (parent, reduced_sim, detailed_sim)
Bases: object

class pymordemos.thermalblock_gui.DetailedSim (args)
Bases: pymordemos.thermalblock_gui.SimBase

Methods

DetailedSim solve
thermalblock_pod module

Thermalblock with POD demo.

Usage:

```
XBLOCKS YBLOCKS SNAPSHOTS RBSIZE
```

Arguments:  
- XBLOCKS Number of blocks in x direction.  
- YBLOCKS Number of blocks in y direction.  
- SNAPSHOTS Number of snapshots for basis generation per component. In total SNAPSHOTS^(XBLOCKS * YBLOCKS).  
- RBSIZE Size of the reduced basis

Options:

- **--grid=NI** Use grid with 2*NI*NI elements [default: 100].  
- -h, --help Show this message.  
- -p, --plot-err Plot error.
--plot-solutions Plot some example solutions.
--pod-norm=NORM Norm (trivial, $h_1$) w.r.t. which to calculate the POD [default: $h_1$].
--test=COUNT Use COUNT snapshots for stochastic error estimation [default: 10].

pymordemos.thermalblock_pod.thermalblock_demo(args)
## Symbols

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong>()</td>
<td>(pymor.functions.basic.FunctionBase method)</td>
<td>66</td>
</tr>
<tr>
<td><strong>add</strong>()</td>
<td>(pymor.la.interfaces.VectorArrayInterface method)</td>
<td>98</td>
</tr>
<tr>
<td><strong>add</strong>()</td>
<td>(pymor.operators.basic.OperatorBase method)</td>
<td>120</td>
</tr>
<tr>
<td><strong>add</strong>()</td>
<td>(pymor.operators.interfaces.OperatorInterface method)</td>
<td>163</td>
</tr>
<tr>
<td><strong>call</strong>()</td>
<td>(pymor.core.cache.cached method)</td>
<td>37</td>
</tr>
<tr>
<td><strong>call</strong>()</td>
<td>(pymor.functions.interfaces.FunctionInterface method)</td>
<td>69</td>
</tr>
<tr>
<td><strong>call</strong>()</td>
<td>(pymor.grids.interfaces.ReferenceElementInterface method)</td>
<td>78</td>
</tr>
<tr>
<td><strong>call</strong>()</td>
<td>(pymor.la.basic.InducedNorm method)</td>
<td>93</td>
</tr>
<tr>
<td><strong>call</strong>()</td>
<td>(pymor.parameters.interfaces.ParameterFunctionalInterface method)</td>
<td>183</td>
</tr>
<tr>
<td><strong>contains</strong>()</td>
<td>(pymor.la.interfaces.VectorSpace method)</td>
<td>103</td>
</tr>
<tr>
<td><strong>eq</strong>()</td>
<td>(pymor.domaindescriptions.boundarytypes.BoundaryType method)</td>
<td>63</td>
</tr>
<tr>
<td><strong>eq</strong>()</td>
<td>(pymor.la.interfaces.VectorSpace method)</td>
<td>103</td>
</tr>
<tr>
<td><strong>get</strong>()</td>
<td>(pymor.core.cache.cached method)</td>
<td>37</td>
</tr>
<tr>
<td><strong>hash</strong>()</td>
<td>(pymor.domaindescriptions.boundarytypes.BoundaryType method)</td>
<td>63</td>
</tr>
<tr>
<td><strong>iadd</strong>()</td>
<td>(pymor.la.interfaces.VectorArrayInterface method)</td>
<td>98</td>
</tr>
<tr>
<td><strong>imul</strong>()</td>
<td>(pymor.la.interfaces.VectorArrayInterface method)</td>
<td>98</td>
</tr>
<tr>
<td><strong>isub</strong>()</td>
<td>(pymor.la.interfaces.VectorArrayInterface method)</td>
<td>98</td>
</tr>
<tr>
<td><strong>len</strong>()</td>
<td>(pymor.la.listvectorarray.ListVectorArray method)</td>
<td>105</td>
</tr>
<tr>
<td><strong>len</strong>()</td>
<td>(pymor.la.numpyvectorarray.NumpyVectorArray method)</td>
<td>115</td>
</tr>
<tr>
<td><strong>len</strong>()</td>
<td>(pymor.playground.la.blockvectorarray.BlockVectorArray method)</td>
<td>187</td>
</tr>
<tr>
<td><strong>ne</strong>()</td>
<td>(pymor.domaindescriptions.boundarytypes.BoundaryType method)</td>
<td>63</td>
</tr>
<tr>
<td><strong>neg</strong>()</td>
<td>(pymor.functions.basic.FunctionBase method)</td>
<td>66</td>
</tr>
<tr>
<td><strong>neg</strong>()</td>
<td>(pymor.la.interfaces.VectorArrayInterface method)</td>
<td>98</td>
</tr>
<tr>
<td><strong>new</strong>()</td>
<td>(pymor.core.interfaces.UberMeta static method)</td>
<td>47</td>
</tr>
<tr>
<td><strong>reduce</strong>()</td>
<td>(pymor.functions.basic.ExpressionFunction method)</td>
<td>66</td>
</tr>
<tr>
<td><strong>reduce</strong>()</td>
<td>(pymor.grids.oned.OnedGrid method)</td>
<td>79</td>
</tr>
<tr>
<td><strong>reduce</strong>()</td>
<td>(pymor.grids.rect.RectGrid method)</td>
<td>81</td>
</tr>
<tr>
<td><strong>reduce</strong>()</td>
<td>(pymor.grids.tria.TriaGrid method)</td>
<td>88</td>
</tr>
<tr>
<td><strong>reduce</strong>()</td>
<td>(pymor.parameters.functionals.ExpressionParameterFunctional method)</td>
<td>181</td>
</tr>
<tr>
<td><strong>repr</strong>()</td>
<td>(pymor.domaindescriptions.basic.CircleDomain method)</td>
<td>59</td>
</tr>
<tr>
<td><strong>radd</strong>()</td>
<td>(pymor.la.interfaces.VectorArrayInterface method)</td>
<td>98</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BasicInterface (class in pymor.core.interfaces), 44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitmapFunction (class in pymor.playground.functions.bitmap), 184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>block() (pymor.playground.la.blockvectorarray.BlockVectorArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>method), 188</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlockVectorArray (class in pymor.playground.la.blockvectorarray,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bottom (pymor.domaindescriptions.basic.CylindricalDomain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attribute), 60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bottom (pymor.domaindescriptions.basic.RectDomain attribute), 62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boundaries() (pymor.grids.interfaces.ConformalTopologicalGrid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface method), 76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boundary_mask() (pymor.grids.interfaces.ConformalTopologicalGrid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface method), 76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boundary_types (pymor.domaindescriptions.interfaces_DOMAIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DescriptionInterface attribute), 63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boundary_types (pymor.grids.interfaces.BoundaryInfoInterface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attribute), 76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BoundaryInfoFromIndicators (class in pymor.grids.boundaryinfos,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BoundaryInfoInterface (class in pymor.grids.interfaces), 75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BoundaryType (class in pymor.domaindescriptions.boundarytypes,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>build_parameter_type() (pymor.parameters.base.Parametric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>method), 178</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burgers2DBumpInitialData (class in pymor.analyticalproblems.burgers,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burgers2DFlux (class in pymor.analyticalproblems.burgers), 26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burgers2DFluxDerivative (class in pymor.analyticalproblems.burgers,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burgers2DProblem (class in pymor.analyticalproblems.burgers), 27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burgers2DSinInitialData (class in pymor.analyticalproblems.burgers,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>burgers_demo() (module pymordemos.burgers), 206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>burgers_demo() (module pymordemos.burgers_ei), 207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BurgersBumpInitialData (class in pymor.analyticalproblems.burgers,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BurgersFlux (class in pymor.analyticalproblems.burgers), 29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BurgersFluxDerivative (class in pymor.analyticalproblems.burgers,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BurgersProblem (class in pymor.analyticalproblems.burgers), 29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BurgersSinInitialData (class in pymor.analyticalproblems.burgers,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busywait() (module pymor.tools.timing), 202</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_region (pymor.core.cache.CacheableInterface attribute), 36</td>
</tr>
<tr>
<td>CacheableInterface (class in pymor.core.cache), 36</td>
</tr>
<tr>
<td>Array (class in pymor.core.cache), 37</td>
</tr>
<tr>
<td>CacheRegion (class in pymor.core.cache), 35</td>
</tr>
<tr>
<td>calculate_sid (pymor.core.interfaces.ImmutableInterface attribute), 46</td>
</tr>
<tr>
<td>cat_arrays() (module pymor.la.basic), 93</td>
</tr>
<tr>
<td>center() (pymor.grids.interfaces.ReferenceElementInterface method), 78</td>
</tr>
<tr>
<td>center() (pymor.grids.referenceelements.Line method), 78</td>
</tr>
<tr>
<td>center() (pymor.grids.referenceelements.Point method), 78</td>
</tr>
<tr>
<td>center() (pymor.grids.referenceelements.Square method), 85</td>
</tr>
<tr>
<td>center() (pymor.grids.referenceelements.Triangle method), 85</td>
</tr>
<tr>
<td>centers() (pymor.grids.interfaces.AffineGridInterface method), 85</td>
</tr>
<tr>
<td>check_ind() (pymor.la.interfaces.VectorArrayInterface method), 99</td>
</tr>
<tr>
<td>check_ind_unique() (pymor.la.interfaces.VectorArrayInterface method), 100</td>
</tr>
<tr>
<td>CircleDomain (class in pymor.domaindescriptions.basic), 59</td>
</tr>
<tr>
<td>clear() (pymor.core.cache.CacheRegion method), 36</td>
</tr>
<tr>
<td>clear_caches() (module pymor.core.cache), 37</td>
</tr>
<tr>
<td>coefficients (pymor.functions.basic.LincombFunction attribute), 68</td>
</tr>
<tr>
<td>ColorBarWidget (class in pymor.gui.gl), 89</td>
</tr>
<tr>
<td>ColoredFormatter (class in pymor.core.logger), 48</td>
</tr>
<tr>
<td>compile_vertex_shader() (module pymor.gui.gl), 89</td>
</tr>
<tr>
<td>ComponentProjection (class in pymor.operators.constructions), 132</td>
</tr>
<tr>
<td>components() (pymor.la.interfaces.VectorArrayInterface method), 100</td>
</tr>
<tr>
<td>components() (pymor.la.listvectorarray.ListVectorArray method), 106</td>
</tr>
<tr>
<td>components() (pymor.la.numpyvectorarray.NumpyVectorArray method), 116</td>
</tr>
<tr>
<td>components() (pymor.playground.la.blockvectorarray.BlockVectorArray method), 188</td>
</tr>
<tr>
<td>Concatenation (class in pymor.operators.constructions), 133</td>
</tr>
<tr>
<td>ConformalTopologicalGridDefaultImplementations (class in pymor.grids.defaultimpl), 73</td>
</tr>
<tr>
<td>ConformalTopologicalGridInterface (class in pymor.grids.interfaces), 76</td>
</tr>
<tr>
<td>ConstantFunction (class in pymor.functions.basic), 64</td>
</tr>
<tr>
<td>ConstantOperator (class in pymor.operators.constructions), 135</td>
</tr>
</tbody>
</table>
Index
pyMOR, Release 0.3.2

elliptic2_demo() (in module pymordemos.elliptic2), 208
elliptic_demo() (in module pymordemos.elliptic), 207
elliptic_oned_demo() (in module pymordemos.elliptic_oned), 208
EllipticProblem (class in pymor.analyticalproblems.elliptic), 31
evaluations() (pymor.grids.interfaces.AffineGridInterface method), 74
embeddings() (pymor.grids.oned.OnedGrid method), 79
embeddings() (pymor.grids.rect.RectGrid method), 81
embeddings() (pymor.grids.subgrid.SubGrid method), 86
embeddings() (pymor.grids.tria.TriaGrid method), 88
embeddings() (pymor.playground.grids.gmsh.GmshGrid method), 186
EmpiricalInterpolatedOperator (class in pymor.operators.ei), 150
empty() (pymor.la.interfaces.VectorArrayInterface method), 100
empty() (pymor.la.interfaces.VectorSpace method), 103
EmptyBoundaryInfo (class in pymor.grids.boundaryinfos), 71
enable_caching() (in module pymor.core.cache), 37
enable_caching() (pymor.core.cache.CacheableInterface method), 36
enable_logging() (pymor.core.interfaces.BasicInterface method), 45
enable_sid_generation() (in module pymor.core.interfaces), 48
enabled (pymor.core.cache.CacheRegion attribute), 36
EngquistOsherFlux (class in pymor.operators.fv), 155
estimate() (pymor.discretizations.basic.DiscretizationBase method), 51
estimate() (pymor.discretizations.interfaces.DiscretizationInterface method), 56
evaluate() (pymor.analyticalproblems.burgers.Burgers2DBumpInitialData method), 26
evaluate() (pymor.analyticalproblems.burgers.Burgers2DFluxInitialData method), 27
evaluate() (pymor.analyticalproblems.burgers.Burgers2DFluxDerivativeInitialData method), 27
evaluate() (pymor.analyticalproblems.burgers.Burgers2DSinInitialData method), 29
evaluate() (pymor.analyticalproblems.burgers.BurgersBumpInitialData method), 29
evaluate() (pymor.analyticalproblems.burgers.BurgersFluxInitialData method), 29
evaluate() (pymor.analyticalproblems.burgers.BurgersFluxDerivativeInitialData method), 30
evaluate() (pymor.analyticalproblems.burgers.BurgersSinInitialData method), 31
evaluate() (pymor.analyticalproblems.thermalblock.ThermalBlockDiffusionFunction method), 65
evaluate() (pymor.functions.basic.ConstantFunction method), 68
evaluate() (pymor.functions.basic.LincombFunction method), 68
evaluate() (pymor.functions.interfaces.FunctionInterface method), 69
evaluate() (pymor.functions.parameters.functional.GenericParameterFunctional method), 181
evaluate() (pymor.functions.parameters.functional.ProjectionParameterFunctional method), 182
evaluate() (pymor.functions.parameters.interfaces.ParameterFunctionalInterface method), 183
evaluate() (pymor.playground.functions.bitmap.BitmapFunction method), 185
evaluate_coefficients() (pymor.functions.basic.LincombFunction method), 68
evaluate_coefficients() (pymor.operators.constructions.LincombOperator method), 143
explicit_euler() (in module pymor.algorithms.timestepping), 24
ExplicitEulerTimeStepper (class in pymor.algorithms.timestepping), 21
export_matrix() (pymor.operators.numpy.NumpyMatrixBasedOperator method), 172
ExpressionFunction (class in pymor.functions.basic), 65
ExpressionFunction (class in pymor.playground.functions.expression_function), 185
ExpressionParameterFunctional (class in pymor.functions.parameters.functional), 180
ExpressionError (class in pymor.core.exceptions), 42
FixedParameterOperator (class in pymor.operators.constructions), 137
fixup_docstring() (in module pymor.core.decorators), 39
float_cmp() (module pymor.tools.floatcmp), 198
float_cmp_all() (module pymor.tools.floatcmp), 199
flux_function (pymor.analyticalproblems.advection.InstationaryAdvectionProblem method), 26
flux_function_derivative (pymor.analyticalproblems.advection.InstationaryAdvectionProblem method), 26
FrozenDict (class in pymor.tools.frozendict), 199
functionals (pymor.discretizations.interfaces.DiscretizationInterface attribute), 56
218 Index
FunctionBase (class in pymor.functions.basic), 66
FunctionInterface (class in pymor.functions.interfaces), 68
functions (pymor.functions.basic.LincombFunction attribute), 68

G
GaussQuadratures (class in pymor.tools.quadratures), 200
GenericFunction (class in pymor.functions.basic), 67
GenericParameterFunctional (class in pymor.parameters.functionals), 181
GenericRBReconstructor (class in pymor.reductors.basic), 191
getLogger() (in module pymor.core.logger), 49
global_to_structured() (pymor.grids.rect.RectGrid method), 81
GLPatchWidget (class in pymor.gui.gl), 89
GmshGrid (class in pymor.playground.grids.gmsh), 186
GmshParseError (class in pymor.playground.grids.gmsh), 186
gram_schmidt() (in module pymor.la.gram_schmidt), 96
gram_schmidt_basis_extension() (in module pymor.algorithms.basisextension), 15
gramian() (pymor.la.interfaces.VectorArrayInterface method), 101
gramian() (pymor.la.listvectorarray.ListVectorArray method), 107
greedy() (in module pymor.algorithms.greedy), 19

H
has_interface_name() (pymor.core.interfaces.BasicInterface class method), 45
iadd_masked() (in module pymor.tools.inplace), 199
IdentityOperator (class in pymor.operators.constructions), 139
ImmutableInterface (class in pymor.core.interfaces), 46
ImmutableMeta (class in pymor.core.interfaces), 47
implementor_names() (pymor.core.interfaces.BasicInterface class method), 45
implementors() (pymor.core.interfaces.BasicInterface class method), 45
implicit_euler() (in module pymor.algorithms.timestepping), 24
ImplicitEulerTimeStepper (class in pymor.algorithms.timestepping), 22
indices_from_parent_indices() (pymor.grids.subgrid.SubGrid method), 87
induced_norm() (in module pymor.la.basic), 93
InducedNorm (class in pymor.la.basic), 93
initial_data (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 26
initial_data (pymor.discretizations.basic.InstationaryDiscretization attribute), 53
inject_sid() (in module pymor.core.interfaces), 48
InstationaryAdvectionProblem (class in pymor.analyticalproblems.advection), 25
InstationaryDiscretization (class in pymor.discretizations.basic), 51
integration_elements() (pymor.grids.interfaces.AffineGridInterface method), 74
interpolate_operators() (in module pymor.algorithms.ei), 18
InterpolationOperator (class in pymor.operators.cg), 127
inv_transposed_two_by_two() (in module pymor.la.inverse), 104
inv_two_by_two() (in module pymor.la.inverse), 104
inverse_relation() (in module pymor.tools.relations), 201
InversionError (class in pymor.core.exceptions), 42
invert_options (pymor.operators.interfaces.OperatorInterface attribute), 163
invert_options() (in module pymor.la.genericsolvers), 94
invert_options() (in module pymor.la.numpysolvers), 110
isub_masked() (in module pymor.operators.inplace), 200
iter_quadrature() (pymor.tools.quadratures.GaussQuadratures class method), 200

J
jacobian() (pymor.operators.basic.OperatorBase method), 123
jacobian() (pymor.operators.basicProjectedOperator method), 125
jacobian() (pymor.operators.constructions.ConstantOperator method), 136
jacobian() (pymor.operators.constructions.LincombOperator method), 143
jacobian() (pymor.operators.ei.EmpiricalInterpolatedOperator method), 152
jacobian() (pymor.operators.ei.ProjectEmpiricalInterpolatedOperator method), 154
jacobian() (pymor.operators.interfaces.OperatorInterface method), 166
jacobian_inverse_transposed() (pymor.grids.interfaces.AffineGridInterface method), 74

L
l1_norm() (pymor.la.interfaces.VectorArrayInterface method), 101
l1_norm() (pymor.la.listvectorarray.ListVectorArray method), 107
l1_norm() (pymor.la.numpyvectorarray.NumpyVectorArray method), 117

Index
Index
name (pymor.core.interfaces.BasicInterface attribute), 45
neighbours() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 76
new_random_state() (in module pymor.tools.random), 201
newton() (in module pymor.algorithms.newton), 20
NewtonError (class in pymor.core.exceptions), 43
no_boundary_type_mask() (pymor.grids.interfaces.BoundaryInfoInterface method), 76
nonlinear_advection_engquist_osher_operator() (in module pymor.operators.fv), 162
nonlinear_advection_lax_friedrichs_operator() (in module pymor.operators.fv), 162
nonlinear_advection_simplified_engquist_osher_operator() (in module pymor.operators.fv), 162
NonlinearAdvectionOperator (class in pymor.operators.fv), 159
NonProjectedReconstructor (class in pymor.reductors.residual), 194
NonProjectedResidualOperator (class in pymor.reductors.residual), 194
NumericalConvectiveFluxInterface (class in pymor.operators.fv), 161
NumpyGenericOperator (class in pymor.operators.numpy), 168
NumpyListVectorArray (class in pymor.la.listvectorarray), 108
NumpyMatrixBasedOperator (class in pymor.operators.numpy), 169
NumpyMatrixOperator (class in pymor.operators.numpy), 172
NumpyVector (class in pymor.la.vectorarray), 109
NumpyVectorArray (class in pymor.la.listvectorarray), 114
NumpyVectorSpace() (in module pymor.la.vectorarray), 119
orthogonal_centers() (pymor.grids.oned.OnedGrid method), 79
orthogonal_centers() (pymor.grids.rect.RectGrid method), 81
parabolic_demo() (in module pymor.playground.demos.parabolic), 184
Parameter (class in pymor.parameters.base), 176
parameter_local_type (pymor.parameters.base.Parametric attribute), 178
parameter_space (pymor.parameters.base.Parametric attribute), 178
parameter_type (pymor.parameters.base.Parametric attribute), 177
parameter_type (pymor.parameters.interfaces.ParameterSpaceInterface attribute), 183
ParameterFunctionalInterface (class in pymor.parameters.interfaces), 182
ParameterSpaceInterface (class in pymor.parameters.interfaces), 183
ParamType (class in pymor.parameters.base), 177
Parametric (class in pymor.parameters.base), 178
Parametric (pymor.parameters.base.Parametric attribute), 178
ParamRuler (class in pymordemos.thermalblock_gui), 210
parent_grid (pymor.grids.subgrid.SubGrid attribute), 86
parent_indices() (pymor.grids.subgrid.SubGrid method), 87
parse_gmsh_file() (in module pymor.playground.grids.gmsh), 187
parse_parameter() (pymor.parameters.base.Parametric method), 179
PatchVisualizer (class in pymor.gui.qt), 91
PlotMainWindow (class in pymor.gui.qt), 92
pod() (in module pymor.la.pod), 119
pod_basis_extension() (in module pymor.algorithms.basisextension), 16
Point (class in pymor.grids.referenceelements), 83
print_defaults() (in module pymor.core.defaults), 40
products (pymor.discretizations.interfaces.DiscretizationInterface attribute), 56
projected() (pymor.operators.basic.OperatorBase method), 123
projected() (pymor.operators.constructions.ConstantOperator method), 136
projected() (pymor.operators.constructions.LincombOperator method), 143
projected() (pymor.operators.constructions.ZeroOperator method), 150
pymor.reductors.residual (module), 194
pymor.reductors.stationary (module), 197
pymor.tools (module), 198
pymor.tools.arguments (module), 198
pymor.tools.floatcmp (module), 198
pymor.tools.frozendict (module), 199
pymor.tools.inplace (module), 199
pymor.tools.pprint (module), 200
pymor.tools.quadratures (module), 200
pymor.tools.random (module), 201
pymor.tools.relations (module), 201
pymor.tools.timing (module), 201
pymor.tools.vtkio (module), 202
pymor.version (module), 203
pymordemos (module), 205
pymordemos.analyze_pickle (module), 205
pymordemos.burgers (module), 206
pymordemos.burgers_ei (module), 206
pymordemos.elliptic (module), 207
pymordemos.elliptic2 (module), 208
pymordemos.elliptic_oned (module), 208
pymordemos.thermalblock (module), 208
pymordemos.thermalblock_gui (module), 209
pymordemos.thermalblock_pod (module), 210

R
range (pymor.operators.interfaces.OperatorInterface attribute), 163
RBGui (class in pymordemos.thermalblock_gui), 210
reconstruct() (pymor.reductors.basic.GenericRBReconstructor method), 191
reconstruct() (pymor.reductors.basic.SubbasisReconstructor method), 192
RectDomain (class in pymor.domaindescriptions.basic), 61
RectGrid (class in pymor.grids.rect), 80
reduce_generic_rb() (in module pymor.reductors.basic), 192
reduce_residual() (in module pymor.reductors.residual), 196
reduce_stationary_affine_linear() (in module pymor.reductors.linear), 193
reduce_stationary_coercive() (in module pymor.reductors.stationary), 197
reduce_to_subbasis() (in module pymor.reductors.basic), 192
ReducedSim (class in pymordemos.thermalblock_gui), 210
reference_element() (pymor.grids.interfaces.AffineGridInterface method), 74
ReferenceElementDefaultImplementations (class in pymor.grids.defaultimpl), 73
ReferenceElementInterface (class in pymor.grids.interfaces), 77
register_type() (pymor.domaindescriptions.boundarytypes.BoundaryType class method), 63
remove() (pymor.la.interfaces.VectorArrayInterface method), 102
remove() (pymor.la.listvectorarray.ListVectorArray method), 107
remove() (pymor.la.numpyvectorarray.NumpyVectorArray method), 118
remove() (pymor.playground.la.blockvectorarray.BlockVectorArray method), 190
replace() (pymor.la.interfaces.VectorArrayInterface method), 102
replace() (pymor.la.listvectorarray.ListVectorArray method), 108
replace() (pymor.la.numpyvectorarray.NumpyVectorArray method), 118
replace() (pymor.playground.la.blockvectorarray.BlockVectorArray method), 190
ResidualOperator (class in pymor.reductors.residual), 195
restricted() (pymor.operators.constructions.ComponentProjection method), 133
restricted() (pymor.operators.constructions.Concatenation method), 135

Q
quadrature() (pymor.grids.interfaces.ReferenceElementInterface method), 78
quadrature() (pymor.grids.referenceelements.Line method), 82
quadrature() (pymor.grids.referenceelements.Point method), 83
quadrature() (pymor.grids.referenceelements.Square method), 84
quadrature() (pymor.grids.referenceelements.Triangle method), 85
quadrature() (pymor.tools.quadratures.GaussQuadratures class method), 200
quadrature_info() (pymor.grids.interfaces.ReferenceElementInterface method), 78
quadrature_info() (pymor.grids.referenceelements.Line method), 82
quadrature_info() (pymor.grids.referenceelements.Point method), 83
quadrature_info() (pymor.grids.referenceelements.Square method), 84
quadrature_info() (pymor.grids.referenceelements.Triangle method), 85
quadrature_points() (pymor.grids.interfaces.AffineGridInterface method), 74

Index 223
restricted() (pymor.operators.fv.NonlinearAdvectionOperator method), 160
restricted() (pymor.operators.interfaces.OperatorInterface method), 167
restricted_to_subbasis() (pymor.reductors.basic.GenericRBReconstructor method), 191
rhs (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 26
rhs (pymor.analyticalproblems.elliptic.EllipticProblem attribute), 32
rhs (pymor.discretizations.basic.InstationaryDiscretization attribute), 53
rhs (pymor.discretizations.basic.StationaryDiscretization attribute), 55
right (pymor.domaindescriptions.basic.LineDomain attribute), 61
right (pymor.domaindescriptions.basic.RectDomain attribute), 61

S
sample_randomly() (pymor.parameters.spaces.CubicParameterSpace method), 184
sample_uniformly() (pymor.parameters.spaces.CubicParameterSpace method), 184
scal() (pymor.la.interfaces.VectorArrayInterface method), 102
scal() (pymor.la.listvectorarray.ListVectorArray method), 108
scal() (pymor.la.numpyvectorarray.NumpyVectorArray method), 118
scal() (pymor.playground.la.blockvectorarray.BlockVectorArray method), 190
set_defaults() (in module pymor.core.defaults), 41
set_log_levels() (in module pymor.core.logger), 49
shape_range (pymor.functions.interfaces.FunctionInterface attribute), 69
sid (pymor.core.interfaces.ImmutableInterface attribute), 46
sid (pymor.parameters.base.Parameter attribute), 177
sid_failure (pymor.core.interfaces.ImmutableInterface attribute), 46
sid_ignore (pymor.core.interfaces.ImmutableInterface attribute), 46
SimBase (class in pymordemos.thermalblock_gui), 210
SimPanel (class in pymordemos.thermalblock_gui), 210
SimplifiedEngquistOsherFlux (class in pymor.operators.fv), 161
size() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 77
size() (pymor.grids.interfaces.ReferenceElementInterface method), 78

size() (pymor.grids.oned.OnedGrid method), 79
size() (pymor.grids.rect.RectGrid method), 81
size() (pymor.grids.referenceelements.Line method), 82
size() (pymor.grids.referenceelements.Point method), 83
size() (pymor.grids.referenceelements.Square method), 84
size() (pymor.grids.referenceelements.Triangle method), 85
size() (pymor.grids.subgrid.SubGrid method), 87
size() (pymor.grids.tria.TriaGrid method), 88
size() (pymor.playground.grids.gmsh.GmshGrid method), 186
solution_space (pymor.discretizations.interfaces.DiscretizationInterface attribute), 55
solve() (pymor.algorithms.timestepping.ExplicitEulerTimeStepper method), 22
solve() (pymor.algorithms.timestepping.ImplicitEulerTimeStepper method), 23
solve() (pymor.algorithms.timestepping.TimeStepperInterface method), 24
solve() (pymor.discretizations.interfaces.DiscretizationInterface method), 56
source (pymor.operators.interfaces.OperatorInterface attribute), 163
space (pymor.la.interfaces.VectorArrayInterface attribute), 98
sparse (pymor.operators.numpy.NumpyMatrixBasedOperator attribute), 170
sparse_options() (in module pymor.la.numpysolvers), 111
SQLiteRegion (class in pymor.core.cache), 37
Square (class in pymor.grids.referenceelements), 84
StationaryAffineLinearReducedEstimator (class in pymor.reductors.linear), 193
StationaryCoerciveEstimator (class in pymor.reductors.stationary), 197
StationaryDiscretization (class in pymor.discretizations.basic), 53
strip_parameter() (pymor.parameters.base.Parametric method), 180
structed_to_global() (pymor.grids.rect.RectGrid method), 81
sub_reference_element() (pymor.grids.interfaces.ReferenceElementInterface method), 78
sub_reference_element() (pymor.grids.referenceelements.Line method), 82
sub_reference_element() (pymor.grids.referenceelements.Point method), 83
sub_reference_element() (pymor.grids.referenceelements.Square method), 84
sub_reference_element() (pymor.grids.referenceelements.Triangle method), 85
SubbasisReconstructor (class in pymor.reductors.basic), 192
subentities() (pymor.grids.interfaces.AffineGridInterface method), 74
subentities() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 77
subentities() (pymor.grids.interfaces.ReferenceElementInterface method), 78
subentities() (pymor.grids.oned.OnedGrid method), 79
subentities() (pymor.grids.rect.RectGrid method), 81
subentities() (pymor.grids.referenceelements.Line method), 82
subentities() (pymor.grids.referenceelements.Point method), 83
subentities() (pymor.grids.referenceelements.Square method), 84
subentities() (pymor.grids.referenceelements.Triangle method), 85
subentities() (pymor.grids.subgrid.SubGrid method), 87
subentities() (pymor.playground.grids.gmsh.GmshGrid method), 186
subentity_embedding() (pymor.grids.interfaces.ReferenceElementInterface method), 78
subentity_embedding() (pymor.grids.referenceelements.Line method), 82
subentity_embedding() (pymor.grids.referenceelements.Point method), 83
subentity_embedding() (pymor.grids.referenceelements.Square method), 84
subentity_embedding() (pymor.grids.referenceelements.Triangle method), 85
SubGrid (class in pymor.grids.subgrid), 86
SubGridBoundaryInfo (class in pymor.grids.boundaryinfos), 71
subtype (pymor.la.interfaces.VectorArrayInterface attribute), 98
subtype (pymor.la.interfaces.VectorSpace attribute), 103
sup_norm() (pymor.la.interfaces.VectorArrayInterface method), 102
sup_norm() (pymor.playground.la.blockvectorarray.BlockVectorArray method), 191
superentities() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 77
superentity_indices() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 77
T
T (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 26
T (pymor.discretizations.basic.InstationaryDiscretization attribute), 53
thermalblock_demo() (in module pymor.demos.thermalblock), 209
thermalblock_demo() (in module pymor.demos.thermalblock_pod), 211
ThermalBlockDiffusionFunction (class in pymor.analyticalproblems.thermalblock), 32
ThermalBlockProblem (class in pymor.analyticalproblems.thermalblock), 33
time_stepper (pymor.discretizations.basic.InstationaryDiscretization attribute), 53
Timer (class in pymor.tools.timing), 201
timeStepperInterface (class in pymor.algorithms.timestepping), 23
top (pymor.domaindescriptions.basic.CylindricalDomain attribute), 60
top (pymor.domaindescriptions.basic.RectDomain attribute), 62
TorusDomain (class in pymor.domaindescriptions.basic), 62
TriaGrid (class in pymor.grids.tria), 87
Triangle (class in pymor.grids.referenceelements), 85
trivial_basis_extension() (in module pymor.algorithms.basisextension), 16
type (pymor.la.interfaces.VectorSpace attribute), 103
type (pymor.domaindescriptions.boundarytypes.BoundaryType attribute), 63
U
UberMeta (class in pymor.core.interfaces), 47
UID (class in pymor.core.interfaces), 47
uid (pymor.core.interfaces.BasicInterface attribute), 45
unique_boundary_type_mask() (pymor.grids.interfaces.BoundaryInfoInterface method), 76
unit_outer_normals() (pymor.grids.interfaces.AffineGridInterface method), 74
unit_outer_normals() (pymor.grids.interfaces.ReferenceElementInterface method), 78
unit_outer_normals() (pymor.grids.referenceelements.Line method), 83
unit_outer_normals() (pymor.grids.referenceelements.Point method), 84
unit_outer_normals() (pymor.grids.referenceelements.Square method), 85
unit_outer_normals() (pymor.grids.referenceelements.Triangle method), 86
unlock() (pymor.core.interfaces.BasicInterface method), 45
unlock() (pymor.core.interfaces.ImmutableInterface method), 47

V
vector_operators (pymor.discretizations.interfaces.DiscretizationInterface attribute), 56
vector_type (pymor.la.listvectorarray.NumpyListVectorArray attribute), 109
VectorArrayInterface (class in pymor.la.interfaces), 97
VectorArrayOperator (class in pymor.operators.constructions), 144
VectorFunctional (class in pymor.operators.constructions), 147
VectorInterface (class in pymor.la.listvectorarray), 109
VectorSpace (class in pymor.la.interfaces), 103
vertex_coordinates() (pymor.grids.rect.RectGrid method), 81
visualize() (pymor.discretizations.basic.DiscretizationBase method), 51
visualize() (pymor.discretizations.interfaces.DiscretizationInterface method), 56
visualize() (pymor.grids.oned.OnedGrid method), 79
visualize() (pymor.grids.rect.RectGrid method), 81
visualize() (pymor.grids.tria.TriaGrid method), 88
visualize() (pymor.gui.qt.Matplotlib1DVisualizer method), 90
visualize() (pymor.gui.qt.PatchVisualizer method), 91
visualize_matplotlib_1d() (in module pymor.gui.qt), 92
volume (pymor.grids.interfaces.ReferenceElementInterface attribute), 78
volumes() (pymor.grids.interfaces.AffineGridInterface method), 74
volumes_inverse() (pymor.grids.interfaces.AffineGridInterface method), 74

W
with_() (pymor.core.interfaces.BasicInterface method), 45
with_() (pymor.discretizations.basic.InstationaryDiscretization method), 53
with_() (pymor.discretizations.basic.StationaryDiscretization method), 55

Z
ZeroOperator (class in pymor.operators.constructions), 148
zeros() (pymor.la.interfaces.VectorArrayInterface method), 103
zeros() (pymor.la.interfaces.VectorSpace method), 104
write_defaults_to_file() (in module pymor.core.defaults), 41
write_vtk() (in module pymor.tools.vtkio), 202