## Contents

1 Getting started 3  
1.1 Installation 3  
1.2 Trying it out 3  
1.3 The thermalblock demo explained 4  
1.4 Learning more 8  

2 Technical Overview 9  
2.1 Three Central Classes 9  
2.2 Base Classes 10  
2.3 Creating Discretizations 11  
2.4 Parameters 12  
2.5 Defaults 12  
2.6 The Reduction Process 12  

3 Environment Variables 15  

4 Release Notes 17  
4.1 pyMOR 0.4 (September 28, 2016) 17  
4.2 pyMOR 0.3 (March 2, 2015) 21  

5 API Documentation 23  
5.1 pymor package 23  

6 Demo Applications 323  
6.1 pymordemos package 323  

Bibliography 335
pyMOR is a software library for building model order reduction applications with the Python programming language. Its main focus lies on the application of reduced basis methods to parameterized partial differential equations. All algorithms in pyMOR are formulated in terms of abstract interfaces for seamless integration with external high-dimensional PDE solvers. Moreover, pure Python implementations of finite element and finite volume discretizations using the NumPy/SciPy scientific computing stack are provided for getting started quickly.
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 --plot-err --plot-solutions 3 2 3 32
```

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

```
./thermalblock.py --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:

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

```
(0,1)------------------(1,1)
<p>| | |
|                   |     |</p>
<table>
<thead>
<tr>
<th>$\mu_{11}$</th>
<th>$\mu_{12}$</th>
<th>$\mu_{13}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu_{21}$</td>
<td>$\mu_{22}$</td>
<td>$\mu_{23}$</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------</td>
<td>---------</td>
</tr>
</tbody>
</table>
(0,0)------------------(1,0)
```

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 a 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 of the 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

```
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:

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

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

d 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 `pymor.discretizations.interfaces.DiscretizationInterface.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_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 `h1_product` attribute of the discretization to the basis extension algorithm. We pass the same product to the reductor which uses it for computing the Riesz representatives required 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.

1.3. The thermalblock demo explained
```python
>>> from functools import partial
>>> extension_algorithm = partial(gram_schmidt_basis_extension, product=d.h1_product)
>>> reductor = partial(
...    reduce_coercive,
...    error_product=d.h1_product,
...    coercivity_estimator=ExpressionParameterFunctional('min(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 = d.parameter_space.sample_uniformly(4)
>>> print(samples[0])
{diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]}
```

Now we start the basis generation:

```python
>>> greedy_data = greedy(d, reductor, samples,
...    extension_algorithm=extension_algorithm,
...    use_estimator=True,
...    max_extensions=32)

01:44 greedy: Started greedy search on 4096 samples
01:44 greedy: Reducing ...
01:44 |    reduce_coercive: RB projection ...
...
01:45 |    reduce_coercive: Assembling error estimator ...
01:45 |      reduce_residual: Estimating residual range ...
01:45 |        estimate_image_hierarchical: Estimating image for basis vector -1 ..
... 01:45 |        estimate_image_hierarchical: Orthonormalizing ...
01:45 |      reduce_residual: Projecting residual operator ...
01:45 greedy: Estimating errors ...
01:47 greedy: Maximum error after 0 extensions: 9.86736953629 (mu = {diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]})
01:47 greedy: Computing solution snapshot for mu = {diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]} ...
01:47 |    StationaryDiscretization: Solving ThermalBlock_CG for {diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]} ...
01:47 greedy: Extending basis with solution snapshot ...
01:47 greedy: Reducing ...
...
01:47 greedy: Estimating errors ...
01:50 greedy: Maximum error after 1 extensions: 3.30998133771 (mu = {diffusion: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]})
01:50 greedy: Computing solution snapshot for mu = {diffusion: [0.1, 0.1, 1.0, 0.1, 0.1, 0.1]} ...
01:50 |    StationaryDiscretization: Solving ThermalBlock_CG for {diffusion: [0.1, 0.1, 1.0, 0.1, 0.1, 0.1]} ...
01:50 greedy: Extending basis with solution snapshot ...
...
...
03:52 greedy: Maximum number of 32 extensions reached.
03:52 greedy: Reducing once more ...
...
03:55 greedy: Greedy search took 130.547789097 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.vectorarrays.numpy.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)
>>> print(np.max(np.abs(gram_matrix - np.eye(32))))
1.16563426164e-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
  1.81310327e-01 2.69920752e-02 -1.79611519e-01 7.99676272e-03
  1.54092560e-01 5.76326362e-02 1.97982347e-01 -2.13775254e-02
  3.12892660e-02 -1.27037440e-01 -1.51352508e-02 3.36101087e-02
  2.05779889e-02 -4.96445984e-03 3.21176662e-02 -2.52674851e-02
  2.92150040e-02 3.23570362e-03 -4.14288199e-03 5.48325425e-03
  4.10728945e-03 -5.08125873e-04 2.71427033e-03 5.83210112e-05]]
>>> U_red = rc.reconstruct(u)
>>> print(U_red.dim)
20201
```

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),
```

1.3. The thermalblock demo explained 7
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!
CHAPTER 2

Technical Overview

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 subclass 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 can be 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. It becomes affordable to generate persistent keys for caching of computation results by generating state ids which uniquely identify the object’s state. Since the state cannot change, these ids have to be computed only once for the lifetime of the object.

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_args 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 instantiate finite element or finite volume operators. Finally these operators are used to instantiate one of the provided Discretization classes.

In pyMOR, analytical problems, Functions, DomainDescriptions, BoundaryInfos and Grids are all immutable, enabling efficient disk caching for the resulting Discretizations, persistent 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 ParameterType of a Parameter is simply obtained by replacing the arrays in the Parameter with their shape. I.e. a ParameterType specifies the names of the parameter components and their expected shapes.

The ParameterType of a Parametric object is determined by the class implementor during __init__ via a call to build_parameter_type, which can be used, to infer the ParameterType from the ParameterTypes of objects the given object depends upon. I.e. an Operator implementing the L2-product with some Function will inherit the ParameterType of the Function.

Reading the reference documentation on pyMOR's parameter handling facilities is strongly advised for implementors of Parametric classes.

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 programmatically 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`. 
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_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 0.4 (September 28, 2016)

With the pyMOR 0.4 release we have changed the copyright of pyMOR to

Copyright 2013-2016 pyMOR developers and contributors. All rights reserved.

Moreover, we have added a Contribution guideline to help new users with starting to contribute to pyMOR. Over 800 single commits have entered this release. For a full list of changes see here. pyMOR 0.4 contains contributions by Andreas Buhr, Michael Laier, Falk Meyer, Petar Mlinarić and Michael Schaefer. See here for more details.

4.1.1 Release highlights

FEniCS and deal.II support

pyMOR now includes wrapper classes for integrating PDE solvers written with the dolfin library of the FEniCS project. For a usage example, see pymordemos.thermalblock_simple.discretize_fenics. Experimental support for deal.II can be found in the pymor-deal.II repository of the pyMOR GitHub organization.

Parallelization of pyMOR’s reduction algorithms

We have added a parallelization framework to pyMOR which allows parallel execution of reduction algorithms based on a simple WorkerPool interface [#14]. The greedy [#155] and ei_greedy algorithms [#162] have been refactored to utilize this interface. Two WorkerPool implementations are shipped with pyMOR: IPythonPool utilizes the parallel computing features of IPython, allowing parallel algorithm execution in large heterogeneous clusters of computing nodes. MPIPool can be used to benefit from existing MPI-based parallel HPC computing architectures [#161].
Support classes for MPI distributed external PDE solvers

While pyMOR’s VectorArray, Operator and Discretization interfaces are agnostic to the concrete (parallel) implementation of the corresponding objects in the PDE solver, external solvers are often integrated by creating wrapper classes directly corresponding to the solvers data structures. However, when the solver is executed in an MPI distributed context, these wrapper classes will then only correspond to the rank-local data of a distributed VectorArray or Operator.

To facilitate the integration of MPI parallel solvers, we have added MPI helper classes [#163] in pymor.vectorarrays.mpi, pymor.operators.mpi and pymor.discretizations.mpi that allow an automatic wrapping of existing sequential bindings for MPI distributed use. These wrapper classes are based on a simple event loop provided by pymor.tools.mpi, which is used in the interface methods of the wrapper classes to dispatch into MPI distributed execution of the corresponding methods on the underlying MPI distributed objects.

The resulting objects can be used on MPI rank 0 (including interactive Python sessions) without any further changes to pyMOR or the user code. For an example, see pymordemos.thermalblock_simple.discretize_fenics.

New reduction algorithms

- adaptive_greedy uses adaptive parameter training set refinement according to [HDO11] to prevent overfitting of the reduced model to the training set [#213].
- reduce_parabolic reduces linear parabolic problems using reduce_generic_rb and assembles an error estimator similar to [GP05], [HO08]. The parabolic_mor demo contains a simple sample application using this reductor [#190].
- The estimate_image and estimate_image_hierarchical algorithms can be used to find an as small as possible space in which the images of a given list of operators for a given source space are contained for all possible parameters mu. For possible applications, see reduce_residual which now uses estimate_image_hierarchical for Petrov-Galerkin projection of the residual operator [#223].

Copy-on-write semantics for VectorArrays

The copy method of the VectorArray interface is now assumed to have copy-on-write semantics. I.e., the returned VectorArray will contain a reference to the same data as the original array, and the actual data will only be copied when one of the arrays is changed. Both NumpyVectorArray and ListVectorArray have been updated accordingly [#55]. As a main benefit of this approach, immutable objects having a VectorArray as an attribute now can safely create copies of the passed VectorArrays (to ensure the immutability of their state) without having to worry about unnecessarily increased memory consumption.

Improvements to pyMOR’s discretization toolkit

- An unstructured triangular Grid is now provided by UnstructuredTriangleGrid. Such a Grid can be obtained using the discretize_gmsh method, which can parse Gmsh output files. Moreover, this method can generate Gmsh input files to create unstructured meshes for an arbitrary PolygonalDomain [#9].
- Basic support for parabolic problems has been added. The discretize_parabolic_cg and discretize_parabolic_fv methods can be used to build continuous finite element or finite volume Discretizations from a given pymor.analyticalproblems.parabolic.ParabolicProblem. The parabolic demo demonstrates the use of these methods [#189].
- The pymor.discretizers.disk module contains methods to create stationary and instationary affinely decomposed Discretizations from matrix data files and an .ini file defining the given problem.
• *EllipticProblems* can now also contain advection and reaction terms in addition to the diffusion part. *discretize_elliptic_cg* has been extended accordingly [#211].

• The *continuous Galerkin* module has been extended to support Robin boundary conditions [#110].

• *BitmapFunction* allows to use grayscale image data as *data Functions* [#194].

• For the visualization of time-dependent data, the colorbars can now be rescaled with each new frame [#91].

**Caching improvements**

• *state id* generation is now based on deterministic pickling. In previous version of pyMOR, the *state id* of immutable objects was computed from the state ids of the parameters passed to the object’s *__init__* method. This approach was complicated and error-prone. Instead, we now compute the *state id* as a hash of a deterministic serialization of the object’s state. While this approach is more robust, it is also slightly more expensive. However, due to the object’s immutability, the *state id* only has to be computed once, and state ids are now only required for storing results in persistent cache regions (see below). Computing such results will usually be much more expensive than the *state id* calculation [#106].

• *CacheRegions* now have a *persistent* attribute indicating whether the cache data will be kept between program runs. For persistent cache regions the *state id* of the object for which the cached method is called has to be computed to obtain a unique persistent id for the given object. For non-persistent regions the object’s *uid* can be used instead. *pymor.core.cache_regions* now by default contains 'memory', 'disk' and 'persistent' cache regions [#182], [#121].

• *defaults* can now be marked to not affect *state id* computation. In previous version of pyMOR, changing any *default* value caused a change of the *state id* pyMOR’s defaults dictionary, leading to cache misses. While this in general is desirable, as, for instance, changed linear solver default error tolerances might lead to different solutions for the same *Discretization* object, it is clear for many I/O related defaults, that these will not affect the outcome of any computation. For these defaults, the *defaults* decorator now accepts a *sid_ignore* parameter, to exclude these defaults from *state id* computation, preventing changes of these defaults causing cache misses [#81].

• As an alternative to using the *@cached* decorator, *cached_method_call* can be used to cache the results of a function call. This is now used in *solve* to enable parsing of the input parameter before it enters the cache key calculation [#231].

### 4.1.2 Additional new features

• *apply_inverse_adjoint* has been added to the *Operator* interface [#133].

• Support for complex values in *NumpyVectorArray* and *NumpyMatrixOperator* [#131].

• New *ProductParameterFunctional*. This *ParameterFunctional* represents the product of a given list of *ParameterFunctionals*.

• New *SelectionOperator* [#105]. This *Operator* represents one *Operator* of a given list of *Operators*, depending on the evaluation of a provided *ParameterFunctional*.

• New block matrix operators [#215]. *BlockOperator* and *BlockDiagonalOperator* represent block matrices of *Operators* which can be applied to appropriately shaped *BlockVectorArrays*.

• *from_file* factory method for *NumpyVectorArray* and *NumpyMatrixOperator* [#118]. *NumpyVectorArray.from_file* and *NumpyMatrixOperator.from_file* can be used to construct such objects from data files of various formats (MATLAB, matrix market, NumPy data files, text).
• **ListVectorArray-based NumpyMatrixOperator** [#164]. The playground now contains NumpyListVectorArrayMatrixOperator which can apply NumPy/SciPy matrices to a ListVectorArray. This Operator is mainly intended for performance testing purposes. The thermalblock demo now has an option --list-vector-array for using this operator instead of NumpyMatrixOperator.

• Log indentation support [#230]. pyMOR’s log output can now be indented via the logger.block(msg) context manager to reflect the hierarchy of subalgorithms.

• Additional INFO2 and INFO3 log levels [#212]. Loggers now have additional info2 and info3 methods to highlight important information (which does fall in the ‘warning’ category).

• Default implementation of as_vector for functionals [#107]. OperatorBase.as_vector now contains a default implementation for functionals by calling apply_adjoint.

• pycontracts has been removed as a dependency of pyMOR [#127].

• Test coverage has been raised to 80 percent.

### 4.1.3 Backward incompatible changes

• VectorArray implementations have been moved to the pymor.vectorarrays sub-package [#89].

• The dot method of the VectorArray interface has been split into dot and pairwise_dot [#76]. The pairwise parameter of dot has been removed, always assuming pairwise == False. The method pairwise_dot corresponds to the pairwise == True case. Similarly the pairwise parameter of the apply2 method of the Operator interface has been removed and a pairwise_apply2 method has been added.

• almost_equal has been removed from the VectorArray interface [#143]. As a replacement, the new method pymor.algorithms.basic.almost_equal can be used to compare VectorArrays for almost equality by the norm of their difference.

• lincomb has been removed from the Operator interface [#83]. Instead, a LincombOperator should be directly instantiated.

• Removal of the options parameter of apply_inverse in favor of solver_options attribute [#122]. The options parameter of OperatorInterface.apply_inverse has been replaced by the solver_options attribute. This attribute controls which fixed (linear) solver options are used when apply_inverse is called. See here <https://github.com/pymor/pymor/pull/184> for more details.

• Renaming of reductors for coercive problems [#224]. pymor.reductors.linear.reduce_stationary_affine_linear and pymor.reductors.stationary.reduce_stationary_coercive have been renamed to pymor.reductors.coercive.reduce_coercive and pymor.reductors.coercive.reduce_coercive_simple. The old names are deprecated and will be removed in pyMOR 0.5.

• Non-parametric objects have now parameter_type {} instead of None [#84].

• Sampling methods of ParameterSpaces now return iterables instead of iterators [#108].

• Caching of solve is now disabled by default [#178]. Caching of solve must now be explicitly enabled by using pymor.core.cache.CacheableInterface.enable_caching.

• The default value for extension_algorithm parameter of greedy has been removed [#82].

• Changes to ei_greedy [#159], [#160]. The default for the projection parameter has been changed from 'orthogonal' to 'ei' to let the default algorithm agree with literature. In addition a copy parameter with default True has been added. When copy is True, the input data is copied before executing the
algorithm, ensuring, that the original VectorArray is left unchanged. When possible, copy should be set to False in order to reduce memory consumption.

• The copy parameter of pymor.algorithms.gram_schmidt.gram_schmidt now defaults to True [#123].
• with_ has been moved from BasicInterface to ImmutableInterface [#154].
• BasicInterface.add_attributes has been removed [#158].
• Auto-generated names no longer contain the uid [#198]. The auto-generated name of pyMOR objects no longer contains their uid. Instead, the name is now simply set to the class name.
• Python fallbacks to Cython functions have been removed [#145]. In order to use pyMOR’s discretization toolkit, building of the _unstructured, inplace, relations Cython extension modules is now required.

4.1.4 Further improvements

• [#78] update apply_inverse signature
• [#115] [algorithms.gram_schmidt] silence numpy warning
• [#144] L2ProductP1 uses wrong quadrature rule in 1D case
• [#147] Debian doc packages have weird title
• [#151] add tests for ‘almost_equal’ using different norms
• [#156] Let thermal block demo use error estimator by default
• [#195] Add more tests / fixtures for operators in pymor.operators.constructions
• [#197] possible problem in caching
• [#207] No useful error message in case PySide.QtOpenGL cannot be imported
• [#209] Allow ‘pip install pymor’ to work even when numpy/scipy are not installed yet
• [#219] add minimum versions for dependencies
• [#228] merge fixes in python3 branch back to master
• [#269] Provide a helpful error message when cython modules are missing
• [#276] Infinite recursion in apply for IdentityOperator * scalar

4.2 pyMOR 0.3 (March 2, 2015)

• Introduction of the vector space concept for even simpler integration with external solvers.
• Addition of a generic Newton algorithm.
• Support for Jacobian evaluation of empirically interpolated operators.
• A new, easy to use mechanism for setting and accessing default values.
• Serialization via the pickle module is now possible for each class in pyMOR. (See the new ‘analyze_pickle’
demo.)

• Addition of generic iterative linear solvers which can be used in conjunction with any operator satisfying py-
MOR’s operator interface. Support for least squares solvers and PyAMG (http://www.pyamg.org/).

• An improved SQLite-based cache backend.

• Improvements to the built-in discretizations: support for bilinear finite elements and addition of a finite volume
diffusion operator.

• Test coverage has been raised from 46% to 75%.

Over 500 single commits have entered this release. A full list of all changes can be obtained under the following
address: https://github.com/pymor/pymor/compare/0.2.2...0.3.0
5.1 pymor package

5.1.1 Subpackages

pymor.algorithms package

Submodules

adaptivegreedy module

class pymor.algorithms.adaptivegreedy.AdaptiveSampleSet (parameter_space)
    Bases: pymor.coreinterfaces.BasicInterface

An adaptive parameter sample set.

Used by adaptive_greedy.

Methods

<table>
<thead>
<tr>
<th>AdaptiveSampleSet.map_vertex_to_mu, refine, visualize</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>AdaptiveSampleSet</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>
pymor.algorithms.adaptivegreedy._estimate(mu, rd=None, d=None, rc=None, error_norm=None)

Called by adaptive_greedy.

pymor.algorithms.adaptivegreedy.adaptive_greedy(discretization, reductor, parameter_space=None, initial_basis=None, use_estimator=True, error_norm=None, extension_algorithm=func.gram_schmidt_basis_extension, target_error=None, max_extensions=None, validation_mus=0, rho=1.1, gamma=0.2, theta=0.0, visualize=False, visualize_vertex_size=80, pool=<pymor.parallel.dummy.DummyPool object>)

Greedy basis generation algorithm with adaptively refined training set.

This method extends pyMOR’s default greedy greedy basis generation algorithm by adaptive refinement of the parameter training set according to [HDO11] to prevent overfitting of the reduced basis to the training set. This is achieved by estimating the reduction error on an additional validation set of parameters. If the ratio between the estimated errors on the validation set and the validation set is larger than rho, the training set is refined using standard grid refinement techniques.

Parameters

discretization See greedy.
reductor See greedy.
parameter_space The ParameterSpace for which to compute the reduced model. If None, the parameter space of the discretization is used.
initial_basis See greedy.
use_estimator See greedy.
error_norm See greedy.
extension_algorithm See greedy.
target_error See greedy.
max_extensions See greedy.
validation_mus

One of the following:

- a list of Parameters to use as validation set,
- a positive number indicating the number of random parameters to use as validation set,
- a non-positive number, indicating the negative number of random parameters to use as validation set in addition to the centers of the elements of the adaptive training set.

rho Maximum allowed ratio between maximum estimated error on validation set vs. maximum estimated error on training set. If the ratio is larger, the training set is refined.
**gamma**  Weight of the age penalty term in the training set refinement indicators.

**theta**  Ratio of training set elements to select for refinement. (One element is always refined.)

**visualize**  If True, visualize the refinement indicators. (Only available for 2 and 3 dimensional parameter spaces.)

**visualize_vertex_size**  Size of the vertices in the visualization.

**pool**  See greedy.

---

### 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.
- **extensions**  Number of greedy iterations.
- **max_errs**  Sequence of maximum errors during the greedy run.
- **max_err_mus**  The parameters corresponding to max_errs.
- **max_val_errs**  Sequence of maximum errors on the validation set.
- **max_val_err_mus**  The parameters corresponding to max_val_errs.
- **refinements**  Number of refinements made in each extension step.
- **training_set_sizes**  The final size of the training set in each extension step.
- **time**  Duration of the algorithm.
- **reduction_data**  Reduction data returned by the last reductor call.

---

### basic module

Module containing some basic but generic linear algebra algorithms.

```python
pymor.algorithms.basic.almost_equal(U, V, U_ind=None, V_ind=None, product=None, norm=None, rtol=1e-14, atol=1e-14)
```

Compare U and V for almost equality.

The vectors of U and V are compared in pairs for almost equality. Two vectors \( \mathbf{u} \) and \( \mathbf{v} \) are considered almost equal iff

\[
||\mathbf{u} - \mathbf{v}||_2 \leq atol + ||\mathbf{v}||_2 \times rtol.
\]

The norm to be used can be specified via the norm or product parameter.

If the length of U (U_ind) resp. V (V_ind) is 1, the single specified vector is compared to all vectors of the other array. Otherwise, the lengths of both indexed arrays have to agree.

---

Parameters
U, V VectorArrays to be compared.

U_ind, V_ind Indices of the vectors that are to be compared (see VectorArray).

product If specified, use this inner product Operator to compute the norm. product and norm are mutually exclusive.

norm If specified, must be a callable, which is used to compute the norm or, alternatively, one of the string ‘l1’, ‘l2’, ‘sup’, in which case the respective VectorArray norm methods are used. product and norm are mutually exclusive. If neither is specified, norm='l2' is assumed.

rtol The relative tolerance.

atol The absolute tolerance.

Defaults

rtol, atol (see pymor.core.defaults)

basisextension module

This module contains algorithms for extending a given reduced 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

```python
extension_algorithm(basis, U, ...)
```

where basis and U are VectorArrays containing the old basis and new vectors to be added. The methods return a tuple new_basis, data where new_basis holds the extended 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.

```python
pymor.algorithms.basisextension.gram_schmidt_basis_extension(basis, U, product=None, copy_basis=True, copy_U=True)
```

Extend basis using Gram-Schmidt orthonormalization.

Parameters

basis VectorArray containing the basis to extend.

U VectorArray containing the new basis vectors.

product The inner 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 has failed. This is the case when no vector in \( U \) is linearly independent of 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 defect of the projection of \( U \) onto the current basis.

**Warning:** The provided basis is assumed to be orthonormal w.r.t. the given inner 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 shall be appended to the basis.
- **product**  The inner 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 has produced no new vectors. This is the case when no vector in \( U \) is linearly independent of the basis.
pyMOR, Release 0.4.2

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

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 Raised when 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 be used to instantiate an EmpiricalInterpolatedOperator (with triangular=False).
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  Inner product Operator used for the 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.

pymor.algorithms.ei.ei_greedy(U, error_norm=None, atol=None, rtol=None, 
max_interpolation_dofs=None, projection='ei', product=None, 
copy=True, pool=<pymor.parallel.dummy.DummyPool object>)

Generate data for empirical interpolation using 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 be used to instantiate an EmpiricalInterpolatedOperator (with triangular=True).

The interpolation data is generated by a greedy search algorithm, where in each loop iteration the worst approximated vector in U is added 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.
atol  Stop the greedy search if the largest approximation error is below this threshold.
rtol  Stop the greedy search if the largest relative 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 vectors by their interpolants. If 'orthogonal', compute the error by comparing with the orthogonal projection onto the span of the collateral basis.
product  If projection == 'orthogonal', the inner product which is used to perform the projection. If None, the Euclidean product is used.
copy  If False, U will be modified during executing of the algorithm.
pool  If not None, the WorkerPool to use for parallelization.

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).

---

```python
def pymor.algorithms.ei.interpolate_operators(
    discretization, operator_names, parameter_sample,
    error_norm=None, atol=None, rtol=None, max_interpolation_dofs=None,
    projection='ei', product=None, pool=<pymor.parallel.dummy.DummyPool object>)
```

Empirical operator interpolation using the EI-Greedy algorithm.

This is a convenience method to facilitate 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.
- **parameter_sample**  A list of `Parameters` for which solution snapshots are calculated.
- **error_norm**  See `ei_greedy`.
- **atol**  See `ei_greedy`.
- **rtol**  See `ei_greedy`.
- **max_interpolation_dofs**  See `ei_greedy`.
- **projection**  See `ei_greedy`.
- **product**  See `ei_greedy`.
- **pool**  If not None, the `WorkerPool` to use for parallelization.

**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.
  - **triangularity_errors**  Sequence of maximum absolute values of interpolation matrix coefficients in the upper triangle (should be near zero).
error module

```python
pymor.algorithms.error.reduction_error_analysis(reduced_discretization, discretization=None, reconstructor=None, test_mus=10, basis_sizes=0, random_seed=None, estimator=True, condition=False, error_norms=(), error_norm_names=None, estimator_norm_index=0, custom=(), plot=False, plot_custom_logarithmic=True, pool=<pymor.parallel.dummy.DummyPool object>)
```

Analyze the model reduction error.

The maximum model reduction error is estimated by solving the reduced `Discretization` for given random `Parameters`.

**Parameters**

- **reduced_discretization**  The reduced `Discretization`.
- **discretization**  The high-dimensional `Discretization`. Must be specified if `error_norms` are given.
- **reconstructor**  The reconstructor for `reduced_discretization`. Must be specified if `error_norms` are given.
- **test_mus**  Either a list of `Parameters` to compute the errors for, or the number of parameters which are sampled randomly from `parameter_space` (if given) or `reduced_discretization`. `parameter_space`.
- **basis_sizes**  Either a list of reduced basis dimensions to consider, or the number of dimensions (which are then selected equidistantly, always including the maximum reduced space dimension). The dimensions are input for `~pymor.reductors.basic.reduce_to_subbasis` to quickly obtain smaller reduced `Discretizations` from `rb_discretization`.
- **random_seed**  If `test_mus` is a number, use this value as random seed for drawing the `Parameters`.
- **estimator**  If `True` evaluate the error estimator of `reduced_discretization` on the test `Parameters`.
- **condition**  If `True`, compute the condition of the reduced system matrix for the given test `Parameters` (can only be specified if `rb_discretization` is an instance of `StationaryDiscretization` and `rb_discretization.operator` is linear).
- **error_norms**  List of norms in which to compute the model reduction error.
- **error_norm_names**  Names of the norms given by `error_norms`. If `None`, the name attributes of the given norms are used.
- **estimator_norm_index**  When `estimator` is `True` and `error_norms` are specified, this is the index of the norm in `error_norms` w.r.t. which to compute the effectivity of the estimator.
- **custom**  List of custom functions which are evaluated for each test `Parameter` and basis size. The functions must have the signature

```python
def custom_value(reduced_discretization, discretization=d, reconstructor, mu, dim):
    pass
```

5.1. pymor package
plot If True, generate a plot of the computed quantities w.r.t. the basis size.
plot_custom_logarithmic If True, use a logarithmic y-axis to plot the computed custom values.
pool If not None, the WorkerPool to use for parallelization.

Returns
Dict with the following fields

- **mus** The test Parameters which have been considered.
- **basis_sizes** The reduced basis dimensions which have been considered.
- **norms** NumPy array of the norms of the high-dimensional solutions w.r.t. all given test Parameters, reduced basis dimensions and norms in error_nors. (Only present when error_norms has been specified.)
- **max_norms** Maxima of norms over the given test Parameters.
- **max_norm_mus** Parameters corresponding to max_norms.
- **errors** NumPy array of the norms of the model reduction errors w.r.t. all given test Parameters, reduced basis dimensions and norms in error_norms. (Only present when error_norms has been specified.)
- **max_errors** Maxima of errors over the given test Parameters.
- **max_error_mus** Parameters corresponding to max_errors.
- **rel_errors** errors divided by norms. (Only present when error_norms has been specified.)
- **max_rel_errors** Maxima of rel_errors over the given test Parameters.
- **max_rel_error_mus** Parameters corresponding to max_rel_errors.
- **error_norm_names** Names of the given error_norms. (Only present when error_norms has been specified.)
- **estimates** NumPy array of the model reduction error estimates w.r.t. all given test Parameters and reduced basis dimensions. (Only present when estimator is True.)
- **max_estimate** Maxima of estimates over the given test Parameters.
- **max_estimate_mus** Parameters corresponding to max_estimate.
- **effectivities** errors divided by estimates. (Only present when estimator is True and error_norms has been specified.)
- **min_effectivities** Minima of effectivities over the given test Parameters.
- **min_effectivity_mus** Parameters corresponding to min_effectivities.
- **max_effectivities** Maxima of effectivities over the given test Parameters.
- **max_effectivity_mus** Parameters corresponding to max_effectivities.
- **errors** NumPy array of the reduced system matrix conditions w.r.t. all given test Parameters and reduced basis dimensions. (Only present when conditions is True.)
- **max_conditions** Maxima of conditions over the given test Parameters.
**max_condition_mus** *Parameters* corresponding to `max_conditions`.

**custom_values** NumPy array of custom function evaluations w.r.t. all given test *Parameters*, reduced basis dimensions and functions in `custom`. (Only present when `custom` has been specified.)

**max_custom_values** Maxima of `custom_values` over the given test *Parameters*.

**max_custom_values_mus** *Parameters* corresponding to `max_custom_values`.

**time** Time (in seconds) needed for the error analysis.

**summary** String containing a summary of all computed quantities for the largest (last) considered basis size.

**figure** The figure containing the generated plots. (Only present when `plot` is True.)

---

**genericsolvers module**

This module contains some iterative linear solvers which only use the *Operator* interface.

---

**pymor.algorithms.genericsolvers.apply_inverse**(op, rhs, options=None)

Solve linear equation system.

Applies the inverse of `op` to the vectors in `rhs`.

**Parameters**

- **op** The linear, non-parametric *Operator* to invert.
- **rhs** VectorArray of right-hand sides for the equation system.
- **options** The solver options to use. (See `options`.)

**Returns**

VectorArray of the solution vectors.

---

**pymor.algorithms.genericsolvers.lgmres**(A, b, x0=None, tol=1e-05, maxiter=1000, M=None, callback=None, inner_m=30, outer_k=3, outer_v=None, store_outer_Av=True)

**pymor.algorithms.genericsolvers.lsmr**(A, b, damp=0.0, atol=1e-06, btol=1e-06, conlim=100000000.0, maxiter=None, show=False)

**pymor.algorithms.genericsolvers.lsqr**(A, b, damp=0.0, atol=1e-08, btol=1e-08, conlim=100000000.0, iter_lim=None, show=False)

---

5.1. *pymor package*
pymor.algorithms.genericsolvers.options

(default_solver='generic_lgmres',
default_least_squares_solver='least_squares_generic_lsmr',
generic_lgmres_tol=1e-05,
generic_lgmres_maxiter=1000,
generic_lgmres_inner_m=39,
generic_lgmres_outer_k=3,
least_squares_generic_lsmr_damp=0.0,
least_squares_generic_lsmr_atol=1e-06,
least_squares_generic_lsmr_btol=1e-06,
least_squares_generic_lsmr_conlim=100000000.0,
least_squares_generic_lsmr_maxiter=None,
least_squares_generic_lsmr_show=False,
least_squares_generic_lsmr_damp=0.0,
least_squares_generic_lsmr_atol=1e-06,
least_squares_generic_lsmr_btol=1e-06,
least_squares_generic_lsmr_conlim=100000000.0,
least_squares_generic_lsmr_maxiter=None,
least_squares_generic_lsmr_show=False)

Returns solver_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 possible values for `solver_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`)

**gram_schmidt module**

```python
pymor.algorithms.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=True)
```

Orthonormalize a `VectorArray` using the stabilized Gram-Schmidt algorithm.

**Parameters**

- `A` - The `VectorArray` which is to be orthonormalized.
- `product` - The inner product `Operator` w.r.t. which to orthonormalize. 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 orthonormal and start the algorithm at the `offset + 1`-th vector.
- `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` in-place.

**Returns**

The orthonormalized `VectorArray`.

**Defaults**

- `atol`, `rtol`, `reiterate`, `reiteration_threshold`, `check`, `check_tol` (see `pymor.core.defaults`)
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). 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_coercive.
- **samples** The set of Parameter samples on which to perform the greedy search.
- **initial_basis** The initial reduced basis at the start of the algorithm. 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 discretization.solve() is called to compute the exact model reduction 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.
- **atol** If not None, stop the algorithm if the maximum (estimated) error on the sample set drops below this value.
- **rtol** If not None, stop the algorithm if the maximum (estimated) relative error on the sample set drops below this value.
- **max_extensions** If not None, stop the algorithm after max_extensions extension steps.
- **pool** If not None, the WorkerPool to use for parallelization.

**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.

extensions  Number of performed basis extensions.

time  Total runtime of the algorithm.

reduction_data  The reduction_data returned by the last reconstructor call.

---

image module

pymor.algorithms.image.estimate_image  

Estimate the image of given Operators for all mu.

Let operators be a list of Operators with common source and range, and let vectors be a list of VectorArrays or vector-like Operators in the range of these operators. Given a VectorArray domain of vectors in the source of the operators, this algorithms determines a VectorArray image of range vectors such that the linear span of image contains:

• op.apply(U, mu=mu) for all operators op in operators, for all possible Parameters mu and for all VectorArrays U contained in the linear span of domain,

• U for all VectorArrays in vectors,

• v.as_vector(mu) for all Operators in vectors and all possible Parameters mu.

The algorithm will try to choose image as small as possible. However, no optimality is guaranteed.

Parameters

operators  See above.

evectors  See above.

domain  See above. If None, an empty domain VectorArray is assumed.

extends  For some operators, e.g. EmpiricalInterpolatedOperator, as well as for all elements of vectors, image is estimated independently from the choice of domain. If extends is True, such operators are ignored. (This is useful in case these vectors have already been obtained by earlier calls to this function.)

orthonormalize  Compute an orthonormal basis for the linear span of image using the gram_schmidt algorithm.

product  Inner product Operator w.r.t. which to orthonormalize.

riesz_representatives  If True, compute Riesz representatives of the vectors in image before orthonormalizing (useful for dual norm computation when the range of the operators is a dual space).
Returns

The `VectorArray` image.

Raises

`ImageCollectionError`  Is raised when for a given `Operator` no image estimate is possible.

```
pymor.algorithms.image.estimate_image_hierarchical(operators=(), vectors=(), domain=None, extends=None, orthonormalize=True, product=None, riesz_representatives=False)
```

Estimate the image of given `Operators` for all mu.

This is an extended version of `estimate_image`, which calls `estimate_image` individually for each vector of domain.

As a result, the vectors in the returned `image VectorArray` will be ordered by the domain vector they correspond to (starting with vectors which correspond to the functionals and to `Operators` for which the image is estimated independently from domain).

This function also returns an `image_dims` list, such that the first `image_dims[i+1]` vectors of `image` correspond to the first `i` vectors of `domain` (the first `image_dims[0]` vectors correspond to `vectors` and to `Operators` with fixed image estimate).

Parameters

- `operators`  See `estimate_image`.
- `vectors`  See `estimate_image`.
- `domain`  See `estimate_image`.
- `extends`  When additional vectors have been appended to the domain `VectorArray` after `estimate_image_hierarchical` has been called, and `estimate_image_hierarchical` shall be called again for the extended domain array, `extends` can be set to `(image, image_dims)`, where `image`, `image_dims` are the return values of the last `estimate_image_hierarchical` call. The old domain vectors will then be skipped during computation and `image`, `image_dims` will be modified in-place.
- `orthonormalize`  See `estimate_image`.
- `product`  See `estimate_image`.
- `riesz_representatives`  See `estimate_image`.

Returns

- `image`  See above.
- `image_dims`  See above.
Raises

ImageCollectionError Is raised when for a given Operator no image estimate is possible.

newton module

pymor.algorithms.newton.newton(operator, rhs, initial_guess=None, mu=None, error_norm=None, least_squares=False, miniter=0, maxiter=100, rtol=1.0, atol=1.0, stagnation_window=3, stagnation_threshold=0.9, 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.
least_squares If True, use a least squares linear solver (e.g. for residual minimization).
miniter Minimum amount of iterations to perform.
maxiter Fail if the iteration count reaches this value without converging.
rtol Finish when the residual norm has been reduced by this factor relative to the norm of the initial residual.
atol Finish when the residual norm is below this threshold.
stagnation_window Finish when 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 intermediate approximations of \( U \) after each iteration.
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, rtol, atol, stagnation_window, stagnation_threshold (see `pymor.core.defaults`)

---

**pod module**

`pymor.algorithms.pod.pod(A, modes=None, product=None, rtol=4e-08, atol=0.0, l2_mean_err=0.0, symmetrize=False, orthonormalize=True, check=True, check_tol=1e-10)`

Proper orthogonal decomposition of A.

Viewing the VectorArray A as a A.dim x len(A) matrix, the return value of this method is the VectorArray of left-singular vectors of the singular value decomposition of A, where the inner product on R^(dim(A)) is given by product and the inner product on R^(len(A)) is the Euclidean inner 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.
- **product** Inner product Operator w.r.t. which the POD is computed.
- **rtol** Singular values smaller than this value multiplied by the largest singular value are ignored.
- **atol** Singular values smaller than this value are ignored.
- **l2_mean_err** Do not return more modes than needed to bound the mean l2-approximation error by this value. I.e. the number of returned modes is at most

\[
\arg\min_N \{ \frac{1}{\text{len}(A)} \times \sum_{n=N+1}^{\infty} s_n^2 \leq l2\text{-mean}_err^2 \}
\]

where s_n denotes the n-th singular value.
- **symmetrize** If True, symmetrize the Gramian again before proceeding.
- **orthonormalize** If True, orthonormalize the computed POD modes again using the `gram_schmidt` algorithm.
- **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

rtol, atol, l2_mean_err, symmetrize, orthonormalize, check, check_tol (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 algorithm 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 used by `InstationaryDiscretization`. The classes `ExplicitEulerTimeStepper` and `ImplicitEulerTimeStepper` encapsulate `explicit_euler` and `implicit_euler` to provide this interface.

```python
class pymor.algorithms.timestepping.ExplicitEulerTimeStepper(nt)
    Bases: pymor.algorithms.timestepping.TimeStepperInterface

Explicit Euler time-stepper.
Solves equations of the form

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

Parameters

nt  The number of time-steps the time-stepper will perform.

Methods

ExplicitEulerTimeStepper
ImmutablerInterface.generate_sid, unlock, with_
BasicInterface disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

Attributes

ImmutablerInterface add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface locked, logger, logging_disabled, name, uid

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

Apply time-stepper to the equation

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

**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.

```python
class pymor.algorithms.timestepping.ImplicitEulerTimeStepper(nt,
                  solver_options='operator')
```

Bases: pymor.algorithms.timestepping.TimeStepperInterface

 Implicit Euler time-stepper.

Solves equations of the form

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

**Parameters**

- **nt** The number of time-steps the time-stepper will perform.
- **solver_options** The solver_options used to invert \( M + dt \times A \). The special values 'mass' and 'operator' are recognized, in which case the solver_options of M (resp. A) are used.

**Methods**

- `ImplicitEulerTimeStepper`
- `ImmutableInterface.generate_sid, unlock, with_`
- `BasicInterface disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__`
solve \((\text{initial\_time}, \text{end\_time}, \text{initial\_data}, \text{operator}, \text{rhs} = \text{None}, \text{mass} = \text{None}, \text{mu} = \text{None}, \text{num\_values} = \text{None})\)

Apply time-stepper to the equation

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

**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.TimeStepperInterface

Bases: pymor.core.interfaces.ImmutableInterface

Interface for time-stepping algorithms.

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

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

Time-steppers used by InstationaryDiscretization have to fulfill this interface.

**Methods**
Attributes

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

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

**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.

---

pymor.algorithms.timestepping.explıcit_euler(*A*, *F*, *U0*, *t0*, *tl*, *nt*, *mu=None*, *num_values=None*)

pymor.algorithms.timestepping.implicit_euler(*A*, *F*, *M*, *U0*, *t0*, *tl*, *nt*, *mu=None*, *num_values=None*, *solver_options='operator'*)

---

pymor.analyticalproblems package

**Submodules**

- advection module
class pymor.analyticalproblems.advection.InstationaryAdvectionProblem(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),
  dirichlet_data=ConstantFunction(array(0), 2),
  initial_data=ConstantFunction(array(1), 2), T=1, name=None)

Bases: pymor.core.interfaces.ImmutableInterface

Instationary advection problem.

The problem is to solve the scalar conservation law:

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

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

**Parameters**

- **domain** A `DomainDescription` of the domain \( \Omega \) the problem is posed on.
- **rhs** The `Function` \( s \). Note that the current time is handled as an additional \( '_t' \) component of the `Parameter` \( \mu \) passed to `rhs`.
- **flux_function** The `Function` \( f \). Note that the current time is handled as an additional \( '_t' \) component of the `Parameter` \( \mu \) which is passed to `flux_function`. \( \text{flux_function.dim_domain} \) has to be \( 1 \), whereas \( \text{flux_function.shape_range} \) has to be \((\text{dim} \ \Omega,)\).
- **flux_function_derivative** The derivative of \( f \) with respect to \( u \).
- **dirichlet_data** `Function` providing the Dirichlet boundary values.
- **initial_data** `Function` providing the initial values \( u_0 \).
- **T** The final time \( T \).
- **name** Name of the problem.

**Methods**

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

**Attributes**

---

5.1. pymor package 45
burgers module

```python
class pymor.analyticalproblems.burgers.Burgers2DBumpInitialData
    Bases: pymor.functions.interfaces.FunctionInterface

    Methods

    Attributes

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

    class pymor.analyticalproblems.burgers.Burgers2DFlux(vx, vy)
        Bases: pymor.functions.interfaces.FunctionInterface
```

**Excerpt from pymor.analyticalproblems.burgers**

**InstationaryAdvectionProblem**

- `domain`
- `rhs`
- `flux_function`
- `flux_function_derivative`
- `dirichlet_data`
- `initial_data`
- `T`

**burgers module**

**Burgers2DBumpInitialData**

- `evaluate`

**Burgers2DFlux**

- `(vx, vy)`

**Methods**

**Attributes**

**evaluate**(*)

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

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

Attributes

- `Burgers2DFluxdim_domain, shape_range`
- `ImmutableInterfaceadd_with_arguments, sid, sid_ignore, with_arguments`
- `BasicInterfacebase_locked, logger, logging_disabled, name, uid`
- `Parametricbuild_parameter_type, local_parameter, parse_parameter, strip_parameter`

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

class `pymor.analyticalproblems.burgers.Burgers2DFluxProblem`(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

5.1. pymor package
for $u$ with $t$ in $[0, 0.3]$, $x$ in $[0, 2] \times [0, 1]$.

**Parameters**

$\mathbf{v}$ The $x$ component of the velocity vector $\mathbf{v}$.

$\mathbf{v}$ The $y$ component of the velocity vector $\mathbf{v}$.

**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.
**evaluate** \((x, \text{mu}=\text{None})\)
Evaluate the function for given argument \(x\) and Parameter \(\text{mu}\).

```python
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>call</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>
```

**evaluate** \((x, \text{mu}=\text{None})\)
Evaluate the function for given argument \(x\) and Parameter \(\text{mu}\).

```python
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>call</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**evaluate** \((U, \text{mu}=\text{None})\)
Evaluate the function for given argument \(x\) and Parameter \(\text{mu}\).
class pymor.analyticalproblems.burgers.BurgersFluxDerivative(v)
Bases: pymor.functions.interfaces.FunctionInterface

Methods

BurgersFluxDerivative
FunctionInterface
call
ImmutableInterface
generate_sid, unlock, with_
BasicInterface
disable_logging, enable_logging, has_interface_name,
implementor_names, implementors, lock, __setattr__
Parametric
build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes

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

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

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] and 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
```python
class pymor.analyticalproblems.burgers.BurgersSinInitialData
Bases: pymor.functions.interfaces.FunctionInterface

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

Attributes
BurgersSinInitialData dim_domain, shape_range
ImmutableInterface add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface locked, logger, logging_disabled, name, uid
Parametric parameter_local_type, parameter_space, parameter_type,
        parametric

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

elliptic module
class pymor.analyticalproblems.elliptic.EllipticProblem(
    domain=RectDomain([[0, 0], [1, 1]]),
    rhs=ConstantFunction(array(1.0), 2),
    diffusion_functions=None,
    diffusion_functionals=None,
    advection_functions=None,
    advection_functionals=None,
    reaction_functions=None,
    reaction_functionals=None,
    dirichlet_data=None,
    neumann_data=None,
    robin_data=None,
    parameter_space=None,
    name=None
)

Bases: pymor.core.interfaces.ImmutableInterface

Affinely decomposed linear elliptic problem.

The problem consists in solving

$$
\begin{vmatrix}
K_d & K_v & K_r \\
- \theta_{d,k} & d_k(x) & u(x, \mu) + \theta_{v,k} & v_k(x) & u(x, \mu) + \theta_{r,k} \\
& \varepsilon_k(x) & u(x, \mu) = f(x, \mu)
\end{vmatrix}
$$

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 ()
- **diffusion_functions** List containing the Functions \( d_k(x) \), each having shape_range of either () or (dim domain, dim domain).
- **diffusion_functionals** List containing the ParameterFunctionals \( \theta_{d,k}(\mu) \). If len(diffusion_functions) == 1, diffusion_functionals is allowed to be None, in which case no parameter dependence is assumed.
- **advection_functions** List containing the Functions \( v_k(x) \), each having shape_range of (dim domain,).
- **advection_functionals** List containing the ParameterFunctionals \( \theta_{v,k}(\mu) \). If len(advection_functions) == 1, advection_functionals is allowed to be None, in which case no parameter dependence is assumed.
- **reaction_functions** List containing the Functions \( r_k(x) \), each having shape_range of ()
- **reaction_functionals** List containing the ParameterFunctionals \( \theta_{r,k}(\mu) \). If len(reaction_functions) == 1, reaction_functionals is allowed to be None, in which case no parameter dependence is assumed.
- **dirichlet_data** Function providing the Dirichlet boundary values.
- **neumann_data** Function providing the Neumann boundary values.
- **robin_data** Tuple of two Functions providing the Robin parameter and boundary values.
- **parameter_space** ParameterSpace for the problem.
name Name of the problem.

Methods

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

Attributes

<table>
<thead>
<tr>
<th>EllipticProblem</th>
</tr>
</thead>
<tbody>
<tr>
<td>advection_functionals, advection_functions, diffusion_functionals, diffusion_functions, dirichlet_data, domain, neumann_data, reaction_functionals, reaction_functions, rhs, robin_data</td>
</tr>
<tr>
<td>ImmutableInterface</td>
</tr>
<tr>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
</tr>
<tr>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

domain

rhs

diffusion_functions

diffusion_functionals

advection_functions

advection_functionals

reaction_functions

reaction_functionals

dirichlet_data

neumann_data

robin_data

helmholtz module

class pymor.analyticalproblems.helmholtz.HelmholtzProblem (domain=RectDomain([[0 0], [1 1]]), rhs=None, parameter_range=(0.0, 100.0), dirichlet_data=None, neumann_data=None) Bases: pymor.analyticalproblems.elliptic.EllipticProblem

Helmholtz equation problem.

This problem is to solve the Helmholtz equation

\[- u(x, k) - k^2 u(x, k) = f(x, k)\]
on a given domain.

**Parameters**

- **domain** A *DomainDescription* of the domain the problem is posed on.
- **rhs** The *Function* \( f(x, \mu) \).
- **parameter_range** A tuple \((k_{\min}, k_{\max})\) describing the interval in which \( k \) is allowed to vary.
- **dirichlet_data** *Function* providing the Dirichlet boundary values.
- **neumann_data** *Function* providing the Neumann boundary values.
- **name** Name of the problem.

**Methods**

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

**Attributes**

- *EllipticProblem*
  - `advection_functionals`, `advection_functions`, `diffusion_functionals`, `diffusion_functions`, `dirichlet_data`, `domain`, `neumann_data`, `reaction_functionals`, `reaction_functions`, `rhs`, `robin_data`
- *ImmutableInterface*
  - `add_with_arguments`, `sid`, `sid_ignore`, `with_arguments`
- *BasicInterface*
  - `locked`, `logger`, `logging_disabled`, `name`, `uid`

**parabolic module**

class `pymor.analyticalproblems.parabolic.ParabolicProblem`:

```python
domain=RectDomain([[0 0], [1 1]]),
rhs=ConstantFunction(array(1.0), 2),
diffusion_functions=(ConstantFunction(array(1.0), 2),),
diffusion_functionals=None,
dirichlet_data=None,
neumann_data=None,
initial_data=ConstantFunction(array(1.0), 2),
T=1,
```

Bases: `pymor.core.interfaces.ImmutableInterface`

Affinely decomposed linear parabolic problem.

The problem consists in solving
\[ \begin{align*}
\frac{\partial}{\partial t} u(x, t, \mu) - \sum_{k=0}^{K} \theta_k(\mu) d_k(x) u(x, t, \mu) &= f(x, t, \mu) \\
\quad u(x, 0, \mu) &= u_0(x, \mu)
\end{align*} \]

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

**Parameters**

- **domain** A `DomainDescription` of the domain 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 \( () \).
- **diffusion_functions** List containing the `Functions` \( d_k(x) \), each having `shape_range` of either \( () \) or \((\text{dim domain}, \text{dim domain})\).
- **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.
- **neumann_data** `Function` providing the Neumann boundary values.
- **initial_data** `Function` providing the initial values.
- **T** The final time \( T \).
- **parameter_space** `ParameterSpace` for the problem.
- **name** Name of the problem.

**Methods**

- `ParabolicProblem.elliptic_part, from_elliptic`
- `ImmutableInterface.add_with_arguments, sid, unlock, with_`
- `BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__`

**Attributes**

- `ParabolicProblem.diffusion_functionals, diffusion_functions, dirichlet_data, domain, initial_data, neumann_data, rhs, T`
- `ImmutableInterface.add_with_arguments, sid, sid_ignore, with_arguments`
- `BasicInterface.added, locked, logger, logging_disabled, name, uid`

**domain**

**rhs**

**diffusion_functions**

**diffusion_functionals**

**dirichlet_data**

**neumann_data**
class pymor.analyticalproblems.thermalblock.ThermalBlockDiffusionFunction(x, y, nx, ny)

Bases: pymor.functions.interfaces.FunctionInterface

Methods
ThermalBlockDiffusionFunction: fromFunction
FunctionInterface: __call__
ImmutableInterface: generate_sid, unlock
BasicInterface: disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__
Parametric: build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes
ThermalBlockDiffusionFunction: dim_domain, shape_range
ImmutableInterface: add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface: locked, logger, logging_disabled, name, uid
Parametric: parameter_local_type, parameter_space, parameter_type, parametric

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

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

\[- [d(x, \mu) \ u(x, \mu)] = f(x, \mu)\]

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 \(\mu_{ij} \).
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 `(μ_min, μ_max)`. Each `Parameter` component `μ_ij` is allowed to lie in the interval `[μ_min, μ_max]`.
- **rhs** The `Function` `f(x, μ)`.

**Methods**

<table>
<thead>
<tr>
<th><code>ImmutableInterface</code></th>
<th><code>generate_sid</code>, <code>unlock</code>, <code>with_</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BasicInterface</code></td>
<td><code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>lock</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th><code>EllipticProblem</code></th>
<th><code>advection_functionals</code>, <code>advection_functions</code>, <code>diffusion_functionals</code>, <code>diffusion_functions</code>, <code>dirichlet_data</code>, <code>domain</code>, <code>neumann_data</code>, <code>reaction_functionals</code>, <code>reaction_functions</code>, <code>rhs</code>, <code>robin_data</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>arguments</code>, <code>sid</code>, <code>sid_ignore</code>, <code>with_arguments</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></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

```python
class pymor.core.backports.abstractclassmethod(callable_method):
    Bases: classmethod
    A decorator indicating abstract classmethods.
    Similar to abstractmethod.
```

5.1. `pymor` package
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>

```python
class pymor.core.backports.abstractstaticmethod(callable_method)

Bases: staticmethod
```

A decorator indicating abstract staticmethods. Similar to abstractmethod.

Usage:

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

‘abstractstaticmethod’ is deprecated. Use ‘staticmethod’ with ‘abstractmethod’ instead.

Methods

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

**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` in case of a persistent `CacheRegion`, else the instance’s `uid`,
2. the method’s `__name__`,
3. the `state id` of the arguments,
4. the `state id` of pyMOR’s global `defaults`.

Note 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 \texttt{CacheRegion}. Currently two backends are provided for memory-based and disk-based caching (\texttt{MemoryRegion} and \texttt{SQLiteRegion}). The available regions are stored in the module level \texttt{cache_regions} dict. The user can add additional regions (e.g. multiple disk cache regions) as required. \texttt{CacheableInterface.cache_region} specifies a key of the \texttt{cache_regions} dict to select a cache region which should be used by the instance. (Setting \texttt{cache_region} to 'None' or 'none' disables caching.)

By default, a ‘memory’, a ‘disk’ and a ‘persistent’ cache region are configured. The paths and maximum sizes of the disk regions, as well as the maximum number of keys of the memory cache region can be configured via the \texttt{pymor.core.cache.default_regions.disk_path}, \texttt{pymor.core.cache.default_regions.disk_max_size}, \texttt{pymor.core.cache.default_regions.persistent_path}, \texttt{pymor.core.cache.default_regions.persistent_max_size} and \texttt{pymor.core.cache.default_regions.memory_max_keys} defaults.

There two ways to disable and enable caching in pyMOR:

1. Calling \texttt{disable_caching} (\texttt{enable_caching}), to disable (enable) caching globally.
2. Calling \texttt{CacheableInterface.disable_caching} (\texttt{CacheableInterface.enable_caching}) to disable (enable) caching for a given instance.

Caching of a method is only active if caching has been enabled both globally (enabled by default) and on instance level. For debugging purposes, it is moreover possible to set the environment variable \texttt{PYMOR_CACHE_DISABLE=1} which overrides any call to \texttt{enable_caching}.

A cache region can be emptied using \texttt{CacheRegion.clear}. The function \texttt{clear_caches} clears each cache region registered in \texttt{cache_regions}.

\begin{verbatim}
class pymor.core.cache.CacheRegion
    Bases: object

    Base class for all pyMOR cache regions.

    Methods
    \begin{verbatim}
    clear, get, set
    \end{verbatim}

    Attributes
    \begin{verbatim}
    persistent
    \end{verbatim}

    persistent
    If True, cache entries are kept between multiple program runs.

    clear()
    Clear the entire cache region.

    get(key)
    Return cache entry for given key.

    Parameters
    key  The key for the cache entry.

    Returns
\end{verbatim}
\end{verbatim}

5.1. pymor package 59
in case the key has been found in the cache region.

(False, None) in case the key is not present in the cache region.

**set**(key, value)
Set cache entry for key to given value.
This method is usually called only once for any given key (with the exception of issues due to concurrency).

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

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

    Methods
    ┌──────────────────────────────────────────────┐
    │ CacheableInterface.cached_method_call,    │
    │ disable_caching, enable_caching           │
    │ ImmutableInterface.generate_sid, unlock,  │
    │ with_                                        │
    │ BasicInterface.disable_logging,           │
    │ enable_logging, has_interface_name,       │
    │ implementor_names, implementors, lock,     │
    │ __setattr__                               │
    └──────────────────────────────────────────────┘

    Attributes
    ┌──────────────────────────────────────────────┐
    │ CacheableInterface.cache_region, sid_ignore│
    │ ImmutableInterface.add_with_arguments, sid,│
    │ with_arguments                               │
    │ BasicInterface.locked, logger, logging_disabled, name, uid │
    └──────────────────────────────────────────────┘

    cache_region
    Name of the CacheRegion to use. Must correspond to a key in the cache_regions dict. If None or 'none', caching is disabled.

    cached_method_call(method, *args, **kwargs)
    Call a given method and cache the return value.
    This method can be used as an alternative to the cached decorator.

    Parameters
    method The method that is to be called. This has to be a method of self.
    args Positional arguments for method.
    kwargs Keyword arguments for method

    Returns
    The (possibly cached) return value of method(*args, **kwargs).

    disable_caching()
    Disable caching for this instance.
```
**enable_caching**(region)

Enable caching for this instance.

When setting the object's cache region to a *persistent* CacheRegion, the object's **state id** will be computed.

**Parameters**

- **region** Name of the CacheRegion to use. Must correspond to a key in the cache_regions dict. If None or 'none', caching is disabled.

```python
class pymor.core.cache.MemoryRegion(max_keys)
Bases: pymor.core.cache.CacheRegion

Methods

MemoryRegion clear, get, set

Attributes

MemoryRegion NO_VALUE
CacheRegion persistent
```

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

Methods

SQLiteRegion clear, get, housekeeping, set

Attributes

CacheRegion persistent
```

```python
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

5.1. pymor package
__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.cleanup_non_persistent_regions()

pymor.core.cache.clear_caches()
Clear all cache regions.

pymor.core.cache.default_regions (disk_path='/tmp/pymor.cache.docs',
disk_max_size=1073741824, persist-
tent_path='/tmp/pymor.persistent.cache.docs', persist-
tent_max_size=1073741824, memory_max_keys=1000)

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.core.decorators.DecoratorBase (func)
Bases: object
A base for all decorators that does the common automagic

Methods

DecoratorBase __get__

__get__(obj, ownerClass=None)
Return a wrapper that binds self as a method of obj (!)

class pymor.core.decorators.DecoratorWithArgsBase
Bases: object
A base for all decorators with args that sadly can do little common automagic

Methods

DecoratorWithArgsBase mark
class pymor.core.decorators.Deprecated

Bases: pymor.core.decorators.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.

Methods

DecoratorBase __get__

pymor.core.decorators.fixup_docstring(doc)

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 a user-modifiable default value use the defaults decorator. As an additional feature, if None is passed 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 a configuration file named pymor_defaults.py in the current working directory. If found, the file is loaded via load_defaults_from_file. However, as a security precaution, the 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. 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.
An exemption from this rule are defaults which are listed in the sid_ignore argument of the defaults decorator. Such defaults will not enter the state id calculation. This allows the user to change defaults related to input/output, e.g. logging, without breaking caching. Before marking defaults as ignored in your own code, however, make sure to double check that these defaults will not affect the result of any mathematical algorithm.

class pymor.core.defaults.DefaultContainer
    Bases: object

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

    Methods
    DefaultContainer get, import_all, keys, update

    Attributes
    DefaultContainer sid

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 in case no value has been provided by the caller of the function. Moreover, if None is passed as a value for a default argument, the argument is set to its default value, as well. If no value has been specified using set_defaults or load_defaults_from_file, the default value provided in the function signature is used.

    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.

    Note that the defaults decorator can also be used in 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).

    sid_ignore List of strings naming the defaults in args which should not enter state id calculation (because they do not affect the outcome of any computation). Such defaults will typically be IO related. Use with extreme caution!

    qualname If a method of a class is decorated, the fully qualified name of the method has to 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 and for cache key generation.
pymor.core.defaults.load_defaults_from_file(filename='./pymor_defaults.py')

Loads default values defined in configuration file.

Suitable 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 generally 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 after 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)

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 generally only be changed/loaded 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).

pymor.core.defaults.write_defaults_to_file(filename='./pymor_defaults.py', packages=('pymor',))

Write the currently set default values 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.

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

```
Exception __new__
```

Attributes

```
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

```
Exception __new__
```

Attributes

```
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.
class pymor.core.exceptions.GmshError
    Bases: exceptions.Exception
    Is raised when a Gmsh related error occurs.

class pymor.core.exceptions.ImageCollectionError(op)
    Bases: exceptions.Exception
    Is raised when a pymor.algorithms.image.estimate_image fails for given operator.

class pymor.core.exceptions.InversionError
    Bases: exceptions.Exception
    Is raised if an operator inversion algorithm fails.
class `pymor.core.exceptions.NewtonError`

Bases: `exceptions.Exception`

Is raised if the Newton algorithm fails to converge.

**Methods**

```
Exception __new__
```

**Attributes**

```
BaseException args, message
```

class `pymor.core.exceptions.SIDGenerationError`

Bases: `exceptions.Exception`

Is raised when generate_sid fails.

**Methods**

```
Exception __new__
```

**Attributes**

```
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`. Thus 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`. 

68 Chapter 5. API Documentation
5. `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 uuid4 which is newly created for each pyMOR run and a counter which is increased for any object that requests an uid.

6. If not set by the user to another value, `BasicInterface.name` is set to the name of the object’s class.

`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. A unique state id for the instance can be calculated by calling `generate_sid` and is then stored as the object’s `sid` attribute. The state id is obtained by deterministically serializing the object’s state and then computing a checksum of the resulting byte stream.

3. `ImmutableInterface.sid_ignore` can be set to a set of attribute names which should be excluded from state id calculation.

4. `ImmutableInterface.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.) `ImmutableInterface.with_arguments` holds the set of allowed arguments.

   `ImmutableInterface` provides a default implementation of `with_` which works by creating a new instance, passing the arguments of `with_` to `__init__`. The missing `__init__` arguments are taken from instance attributes of the same name.

---

```python
class pymor.core.interfaces.BasicInterface
    Bases: object
    Base class for most classes in pyMOR.

    Methods
    BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, __setattr__

    Attributes
    BasicInterface locked, logger, logging_disabled, name, uid

    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 set to the class name.
```

---

5.1. pymor package 69
uid
A unique id for each instance. The uid is obtained by using \texttt{UID} and is unique for all pyMOR objects ever created.

\texttt{__setattr__}(\texttt{key}, \texttt{value})
depending on \_locked state I delegate the setattr call to object or raise an Exception

\texttt{disable\_logging}(\texttt{doit=True})
Disable logging output for this instance.

\texttt{enable\_logging}(\texttt{doit=True})
Enable logging output for this instance.

classmethod \texttt{has\_interface\_name}()
True if the class name ends with \texttt{Interface}. Used for introspection.

classmethod \texttt{implementor\_names}(\texttt{descend=False})
For convenience I return a list of my implementor names instead of class objects

classmethod \texttt{implementors}(\texttt{descend=False})
I return a, potentially empty, list of my subclass-objects. If \texttt{descend} is \texttt{True}, I traverse my entire subclass hierarchy and return a flattened list.

\texttt{lock}(\texttt{doit=True})
Make the instance immutable.

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

\texttt{unlock}()
Make the instance mutable again, after it has been locked using \texttt{lock}.

class \texttt{pymor.core.interfaces.ImmutableInterface}
\texttt{Bases: pymor.core.interfaces.BasicInterface}
Base class for immutable objects in pyMOR.

Instances of \texttt{ImmutableInterface} are immutable in the sense that they are \texttt{locked} after \texttt{__init__} returns.

\textbf{Warning: } For instances of \texttt{ImmutableInterface}, the result of member function calls should be completely determined by the function’s arguments together with the object’s state id 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.

Also note that mutable private attributes will cause false cache misses when these attributes enter state id calculation. If your implementation uses such attributes, you should therefore add their names to the \texttt{sid\_ignore} set.

\textbf{Methods}

| \texttt{Immuta} \texttt{bleInterface.generate\_sid, unlock, \_with|} |
| \texttt{BasicInterface.disable\_logging, enable\_logging, has\_interface\_name,} |
| \texttt{implementor\_names, implementors, lock, \_setattr__} |
Attributes

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

**add_with_arguments**
Set of additional arguments for with_. (See `with_arguments`.)

**sid**
The objects state id. Only available after `generate_sid` has been called.

**sid_ignore**
Set of attributes not to include in state id calculation.

**with_arguments**
Set of allowed keyword arguments for with_. This is the union of the argument names of `__init__` and the names specified via `add_with_arguments`.

**__metaclass__**
alias of `ImmutableMeta`

**generate_sid** *(debug=False)*
Generate a unique state id for the given object.

The generated state id is stored in the object’s sid attribute.

**Parameters**

**debug** If True, produce some debugging output.

**Returns**

The generated state id.

**unlock** ()
Make the instance mutable.

**Warning:** Unlocking an instance of `ImmutableInterface` will result in the deletion of its state id. However, this will not delete the state ids of objects referencing it. You really should not unlock an object unless you really know what you are doing.

**with_** (**kwargs**)
Returns a copy with changed attributes.

The default implementation is to create a new class instance with the given keyword arguments as arguments for `__init__`. Missing arguments are obtained form instance attributes with the same name.

**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.

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

Metaclass for ImmutableInterface.

Methods

<table>
<thead>
<tr>
<th>ABCMeta</th>
<th>register, <strong>instancecheck</strong>, <strong>subclasscheck</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>mro, <strong>subclasses</strong></td>
</tr>
</tbody>
</table>
```

```python
class pymor.core.interfaces.UID
Bases: object

Provides unique, quickly computed ids by combining a session UUID4 with a counter.

Attributes

<table>
<thead>
<tr>
<th>UID</th>
<th>counter, prefix, uid</th>
</tr>
</thead>
</table>

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

Methods

```python
<table>
<thead>
<tr>
<th>UberMeta</th>
<th><strong>new</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCMeta</td>
<td>register, <strong>instancecheck</strong>, <strong>subclasscheck</strong></td>
</tr>
<tr>
<td>type</td>
<td>mro, <strong>subclasses</strong></td>
</tr>
</tbody>
</table>

static __new__(classname, bases, classdict)

I copy 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 environment variable to 1.

```python
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

```python
<table>
<thead>
<tr>
<th>classmethod</th>
<th><strong>new</strong></th>
</tr>
</thead>
</table>
```
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.generate_sid(obj, debug=False)
Generate a unique state id for the current state of the given object.

Parameters

- **obj**  The object for which to compute the state sid.
- **debug**  If True, produce some debug output.

Returns

The generated state id.

logger module

This module contains pyMOR’s logging facilities.

pyMOR’s logging facilities are based on the logging module of the Python standard library. To obtain a new logger object use getLogger. Logging can be configured via the set_log_format and set_log_levels methods.

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

<table>
<thead>
<tr>
<th>ColoredFormatter</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatter</td>
<td>converter, formatException, formatTime, usesTime</td>
</tr>
</tbody>
</table>

class pymor.core.logger.DummyLogger
Bases: object

Methods

5.1. pymor package
class pymor.core.logger.LogIndenter (logger, doit)
Bases: object

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_format (max_hierarchy_level=1, indent_blocks=True, block_timings=False)
Set log levels for pyMOR’s logging facility.

Parameters
max_hierarchy_level The number of components of the loggers name which are printed. (The first component is always stripped, the last component always preserved.)
indent_blocks If True, indent log messages inside a code block started with with logger.block(...).
block_timings If True, measure the duration of a code block started with with logger.block(...).

Defaults
max_hierarchy_level, indent_blocks, block_timings (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. In particular, these methods will use `dumps_function` to serialize function objects which cannot be pickled by Python’s standard methods. Note, however, pickling such methods should be avoided since the implementation of `dumps_function` uses non-portable implementation details of CPython to achieve its goals.

```python
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.dump(obj, file, protocol=None)

pymor.core.pickle.dumps(obj, protocol=None)

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.load(file)

pymor.core.pickle.loads(str)

pymor.core.pickle.loads_function(s)
    Restores a function serialized with `dumps_function`.
```
class pymor.discretizations.basic.DiscretizationBase(operators, functionals, vector_operators, products=None, estimator=None, visualizer=None, cache_region=None, name=None)

Bases: pymor.discretizations.interfaces.DiscretizationInterface

Base class for Discretizations providing some common functionality.

Methods

- estimate, visualize
- solve
- cached_method_call, disable_caching, enable_caching
- generate_sid, unlock, with_
- disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__
- build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

- sid_ignore
- functionals, linear, operators, products, solution_space, vector_operators
- cache_region
- add_with_arguments, sid, with_arguments
- locked, logger, logging_disabled, name, uid
- parameter_local_type, parameter_space, parameter_type, parametric

estimate(U, mu=None)

Estimate the discretization error for a given solution.

The discretization error could be the error w.r.t. the analytical solution of the given problem or the model reduction error w.r.t. a corresponding high-dimensional Discretization.

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.

*kwargs* See docstring of self.visualizer.visualize.

class pymor.discretizations.basic.InstationaryDiscretization(*T, initial_data=None, operator=None, 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=None, 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 final 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* The time-stepper to be used by solve.
**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 *CacheRegion* to use.

**name** Name of the discretization.

---

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstationaryDiscretization</td>
<td><code>with_...</code></td>
</tr>
<tr>
<td>DiscretizationBase</td>
<td><code>estimate, visualize</code></td>
</tr>
<tr>
<td>DiscretizationInterface</td>
<td></td>
</tr>
<tr>
<td>CacheableInterface</td>
<td><code>cached_method_call, disable_caching, enable_caching</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>generate_sid, unlock</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__</code></td>
</tr>
<tr>
<td>Parametric</td>
<td><code>build_parameter_type, local_parameter, parse_parameter, strip_parameter</code></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstationaryDiscretization</td>
<td><code>initial_data, mass, operator, rhs, T, time_stepper</code></td>
</tr>
<tr>
<td>DiscretizationBase</td>
<td><code>sid_ignore</code></td>
</tr>
<tr>
<td>DiscretizationInterface</td>
<td><code>functionals, linear, operators, products, solution_space, vector_operators</code></td>
</tr>
<tr>
<td>CacheableInterface</td>
<td><code>cache_region</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>add_with_arguments, sid, with_arguments</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>locked, logger, logging_disabled, name, uid</code></td>
</tr>
<tr>
<td>Parametric</td>
<td><code>parameter_local_type, parameter_space, parameter_type, parametric</code></td>
</tr>
</tbody>
</table>

**T** The final time T.

**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 create a new class instance with the given keyword arguments as arguments for __init__. Missing arguments are obtained form instance attributes with the same name.

**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 (operator=None, 
              rhs=None, products=None, operators=None, functionals=None, vector_operators=None, 
              parameter_space=None, estimator=None, visualizer=None, cache_region=None, 
              name=None)

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

Generic class for discretizations of stationary problems.
This class describes discrete problems given by the equation:

\[ L(u(\mu), \mu) = F(\mu) \]

with a linear functional F and a (possibly non-linear) operator L.

**Parameters**

**operator** The Operator L.

**rhs** The Functional F.

**products** A dict of inner 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 CacheRegion to use.

name  Name of the discretization.

Methods

<table>
<thead>
<tr>
<th>StationaryDiscretization</th>
<th>estimate, visualize</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiscretizationBase</td>
<td></td>
</tr>
<tr>
<td>DiscretizationInterface</td>
<td>cached_method_call, disable_caching, enable_caching</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>getattr</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>StationaryDiscretization</th>
<th>operator, rhs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiscretizationBase</td>
<td>sid_ignore</td>
</tr>
<tr>
<td>DiscretizationInterface</td>
<td>functionals, linear, operators, products, solution_space, vector_operators</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>cache_region</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<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 create a new class instance with the given keyword arguments as arguments for __init__. Missing arguments are obtained from instance attributes with the same name.
**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

```python
class pymor.discretizations.interfaces.DiscretizationInterface

Interface for discretization objects.

A discretization object defines a discrete problem via its class and the Operators it contains. Furthermore, discretizations can be solved for a given Parameter resulting in a solution VectorArray.

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discretization:</td>
</tr>
<tr>
<td>distribute, solve, visualize</td>
</tr>
<tr>
<td>CacheableInterface:</td>
</tr>
<tr>
<td>reached_method_call, disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface:</td>
</tr>
<tr>
<td>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface:</td>
</tr>
<tr>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric:</td>
</tr>
<tr>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiscretizationInterface:</td>
</tr>
<tr>
<td>functionals, linear, operators, products, solution_space, vector_operators</td>
</tr>
<tr>
<td>CacheableInterface:</td>
</tr>
<tr>
<td>cache_region, sid_ignore</td>
</tr>
<tr>
<td>ImmutableInterface:</td>
</tr>
<tr>
<td>add_with_arguments, sid, with_arguments</td>
</tr>
<tr>
<td>BasicInterface:</td>
</tr>
<tr>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric:</td>
</tr>
<tr>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>
```

**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 (see `pymor.reductors.basic.reduce_generic_rb` for a usage example).
**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.

The discretization error could be the error w.r.t. the analytical solution of the given problem or the model reduction error w.r.t. a corresponding high-dimensional Discretization.

**Parameters**

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

**Returns**
The estimated error.

**solve (mu=None, **kwargs)**
Solve the discrete problem for the Parameter `mu`.

The result will be cached in case caching has been activated for the given discretization.

**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.
class pymor.discretizations.mpi.MPIDiscretization(obj_id, operators, functionals, vector_operators, products=None, pickle_subtypes=True, array_type=<class 'pymor.vectorarrays.mpi.MPIVectorArray'>):

Bases: pymor.discretizations.basic.DiscretizationBase

Wrapper class for MPI distributed Discretizations.

Given a single-rank implementation of a Discretization, this wrapper class uses the event loop from pymor.tools.mpi to allow an MPI distributed usage of the Discretization. The underlying implementation needs to be MPI aware. In particular, the discretization's solve method has to perform an MPI parallel solve of the discretization.

Note that this class is not intended to be instantiated directly. Instead, you should use mpi_wrap_discretization.

Parameters

obj_id  ObjectId of the local Discretization on each rank.

operators  Dictionary of all Operators contained in the discretization, wrapped for use on rank 0. Use mpi_wrap_discretization to automatically wrap all operators of a given MPI-aware Discretization.

functionals  See operators.

vector_operators  See operators.

products  See operators.

pickle_subtypes  If pickle_subtypes is False, a unique identifier is computed for each local solution_space subtype, which is then transferred to rank 0 instead of the true subtype. This allows to use MPIVectorArray, MPIOperator, MPIDiscretization even when the local subtypes are not picklable.

array_type  This class will be used to wrap the local VectorArrays returned by solve on each rank into an MPI distributed VectorArray managed from rank 0. By default, MPIVectorArray will be used, other options are MPIVectorArrayAutoComm and MPIVectorArrayNoComm.

Methods

<table>
<thead>
<tr>
<th>Discretization</th>
<th>estimate, visualize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discretization</td>
<td>interface</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>...method_call, disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>...generate_sid, unlock, with__</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes
**DiscretizationBase**: `sid_ignore`

**DiscretizationInterface**: `functional, linear, operators, products, solution_space, vector_operators`

**CacheableInterface**: `cache_region`

**ImmutableInterface**: `add_with_arguments, sid, with_arguments`

**BasicInterface**: `locked, logger, logging_disabled, name, uid`

**Parametric**: `parameter_local_type, parameter_space, parameter_type, parametric`

---

**class** `pymor.discretizations.mpi.MPIVisualizer(d_obj_id)`

**Bases**: `pymor.core.interfaces ImmutableInterface`

---

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPIVisualizer</code></td>
</tr>
<tr>
<td><code>visualize</code></td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
</tr>
<tr>
<td><code>generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
</tr>
<tr>
<td><code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>lock</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

---

**Attributes**

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

---

**pymor.discretizations.mpi.mpi_wrap_discretization** *(local_discretizations, use_with=False, with_apply2=False, pickle_subtypes=True, array_type=<class 'pymor.vectorarrays.mpi.MPIVectorArray'>)*

Wrap MPI distributed local `Discretizations` to a global `Discretization` on rank 0.

Given MPI distributed local `Discretizations` referred to by the `ObjectId local_discretizations`, return a new `Discretization` which manages these distributed discretizations from rank 0. This is done by first wrapping all `Operators` of the `Discretization` using `mpi_wrap_operator`.

Alternatively, `local_discretizations` can be a callable (with no arguments) which is then called on each rank to instantiate the local `Discretizations`.

When `use_with` is `False`, an `MPIDiscretization` is instantiated with the wrapped operators. A call to `solve` will then use an MPI parallel call to the `solve` methods of the wrapped local `Discretizations` to obtain the solution. This is usually what you want when the actual solve is performed by an implementation in the external solver.

When `use_with` is `True`, `with_` is called on the local `Discretization` on rank 0, to obtain a new `Discretization` with the wrapped MPI `Operators`. This is mainly useful when the local discretizations are generic `Discretizations` as in `pymor.discretizations.basic` and `solve` is implemented directly in pyMOR via operations on the contained `Operators`.

---

**Parameters**
local_discretizations  

ObjectId of the local Discretizations on each rank or a callable generating the Discretizations.

use_with  See above.

with_apply2  See MPIOperator.

pickle_subtypes  See MPIOperator.

array_type  See MPIOperator.

---

pymor.discretizers package

Submodules

advection module

---

pymor.discretizers.advection.discretize_nonlinear_instationary_advection_fv (analytical_problem, diameter=None, nt=100, num_flux='lax_friedrichs', lxf_lambda=1.0, eo_gausspoints=5, eo_intervals=1, num_values=None, domain_discretizer=None, grid=None, boundary_info=None)

Discretizes an InstationaryAdvectionProblem using the finite volume method. Explicit Euler time-stepping is used for time discretization.

---

Parameters

analytical_problem  The InstationaryAdvectionProblem to discretize.

diameter  If not None, diameter is passed as an argument 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_oshser', 'simplified_engquist_oshser' (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 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`.

---

**disk module**

---

```python
disk module

pymor.discretizers.disk.discretize_instationary_from_disk

Load a linear affinely decomposed `InstationaryDiscretization` from file.

Similarly to `discretize_stationary_from_disk`, the discretization is specified via an ini.-file of the following form

```system-matrices```
L_1.mat: l_1(μ_1,...,μ_n)
L_2.mat: l_2(μ_1,...,μ_n)
...

```rhs-vectors```
F_1.mat: f_1(μ_1,...,μ_n)
F_2.mat: f_2(μ_1,...,μ_n)
...

```mass-matrix```
D.mat

```initial-solution```
u0: u0.mat

```parameter```
μ_1: a_1,b_1
...
```
Parameters

- **parameter_file** Path to the `.ini` parameter file.
- **T** End-time of desired solution. If `None`, the value specified in the parameter file is used.
- **steps** Number of time steps to. If `None`, the value specified in the parameter file is used.
- **u0** Initial solution. If `None` the initial solution is obtained from parameter file.
- **time_stepper** The desired time stepper to use. If `None`, implicit Euler time stepping is used.

Returns

- **discretization** The `InstationaryDiscretization` that has been generated.

```
pymor.discretizers.disk.discretize_stationary_from_disk(parameter_file)
```

Load a linear affinely decomposed `StationaryDiscretization` from file.

The discretization is defined via an `.ini`-style file as follows:

```
[system-matrices]
L_1.mat: l_1(\mu_1, \ldots, \mu_n)
L_2.mat: l_2(\mu_1, \ldots, \mu_n)
...

[rhs-vectors]
F_1.mat: f_1(\mu_1, \ldots, \mu_n)
F_2.mat: f_2(\mu_1, \ldots, \mu_n)
...

[parameter]
\mu_1: a_1, b_1
\mu_2: a_2, b_2
...
\mu_n: a_n, b_n

[products]
Prod1: P_1.mat
Prod2: P_2.mat
...
```

Here, `L_1.mat`, `L_2.mat`, ..., `F_1.mat`, `F_2.mat`, ... are files containing matrices `L_1`, `L_2`, ... and vectors `F_1.mat`, `F_2.mat`, ... which correspond to the affine components of the operator and right-hand matrices.
side functional. The respective coefficient functionals, are given via the string expressions \( l_1(\mu) \), \( l_2(\mu) \), ..., \( f_1(\mu) \) in the (scalar-valued) Parameter components \( w_1, \ldots, w_n \). The allowed lower and upper bounds \( a_i, b_i \) for the component \( \mu_i \) are specified in the [parameters] section. The resulting operator and right-hand side are then of the form

\[
L(\mu) = l_1(\mu) \cdot L_1 + l_2(\mu) \cdot L_2 + \ldots \\
F(\mu) = f_1(\mu) \cdot F_1 + f_2(\mu) \cdot L_2 + \ldots
\]

In the [products] section, an optional list of inner products \( \text{Prod}_1, \text{Prod}_2, \ldots \) with corresponding matrices \( P_1, P_2, \ldots \) can be specified.

Example:

```
[system-matrices]
matrix1.mat: 1.
matrix2.mat: 1. - theta**2

[rhs-vectors]
rhs.mat: 1.

[parameter]
theta: 0, 0.5

[products]
h1: h1.mat
l2: mass.mat
```

Parameters

- **parameter_file**  Path to the parameter file.

Returns

- **discretization**  The StationaryDiscretization that has been generated.

elliptic module

```
pymor.discretizers.elliptic.discretize_elliptic_cg(analytical_problem, diameter=None, domain_discretizer=None, grid=None, boundary_info=None)
```

Discretizes an EllipticProblem using finite elements.

Parameters

- **analytical_problem**  The EllipticProblem to discretize.
- **diameter**  If not None, diameter is passed as an argument to the domain_discretizer.
- **domain_discretizer**  Discretizer to be used for discretizing the analytical domain. This has to be a function \( \text{domain_discretizer}(\text{domain_description}, \text{diameter}, \ldots) \). If None, \( \text{discretize_domain_default} \) 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(analytical_problem, diameter=None, domain_discretizer=None, grid=None, boundary_info=None)

Discretizes an EllipticProblem using the finite volume method.

Parameters

analytical_problem  The EllipticProblem to discretize.
diameter  If not None, diameter is passed as an argument 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 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.

parabolic module
Discretizes an `ParabolicProblem` using finite elements.

**Parameters**

- **analytical_problem**: The `ParabolicProblem` to discretize.
- **diameter**: If not None, diameter is passed as an argument to the `domain_discretizer`. If None, `discretize_domain_default` is used.
- **domain_discretizer**: Discretizer to be used for discretizing the analytical domain. This has to be a function `domain_discretizer(domain_description, diameter, ...)`. 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.
- **num_values**: The number of returned vectors of the solution trajectory. If None, each intermediate vector that is calculated is returned.
- **time-stepper**: The `time-stepper` to be used by `solve`.
- **nt**: If `time-stepper` is not specified, the number of time steps for implicit Euler time stepping.

**Returns**

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

Discretizes an `ParabolicProblem` using the finite volume method.

**Parameters**

- **analytical_problem**: The `ParabolicProblem` to discretize.
- **diameter**: If not None, diameter is passed to the `domain_discretizer`. If None, `discretize_domain_default` is used.
**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.

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

**time_stepper** The `time-stepper` to be used by `solve`.

**nt** If `time_stepper` is not specified, the number of time steps for implicit Euler time stepping.

**Returns**

**discretization** The `Discretization` that has been generated.

**data** Dictionary with the following entries:

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

---

**pymor.domaindescriptions package**

**Submodules**

**basic module**

**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.

**Methods**

<table>
<thead>
<tr>
<th>CircleDomain</th>
<th><strong>repr</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <em>setattr</em>_</td>
</tr>
</tbody>
</table>

**Attributes**

---

5.1. pymor package
pyMOR, Release 0.4.2

<table>
<thead>
<tr>
<th>CircleDomain</th>
<th>dim, domain, width</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainDescriptionInterface</td>
<td>boundary_types, has_dirichlet, has_neumann, has_robin</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

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

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.
- **top** The **BoundaryType** of the top edge.
- **bottom** The **BoundaryType** of the bottom edge.

### Methods

```python
CylindricalDomain repr
ImmutableInterface generate_sid, unlock, with_
BasicInterface disable_logging, enable_logging, has_interface_name,
implementor_names, implementors, lock, __setattr__
```

### Attributes

```python
CylindricalDomain bottom, diameter, dim, domain, height, lower_left, top,
upper_right, volume, width
DomainDescriptionInterface boundary_types, has_dirichlet, has_neumann, has_robin
ImmutableInterface add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface locked, logger, logging_disabled, name, uid
```

```python
domain
top
bottom
___repr__() <==> repr(x)
```
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

LineDomain __repr__
ImmutableInterface generate_sid, unlock, with_
BasicInterface disable_logging, enable_logging, has_interface_name,
        implementor_names, implementors, lock, __setattr__

Attributes

LineDomain dim, domain, left, right, width
DomainDescriptionInterface boundary_types, has_dirichlet, has_neumann,
        has_robin
ImmutableInterface add_with_arguments, sid, sid_ignore,
        with_arguments
BasicInterface locked, logger, logging_disabled, name, uid

domain
left
right
__repr__(x) <==> repr(x)

class pymor.domaindescriptions.basic.RectDomain(domain=((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

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RectDomain</code></td>
<td><code>__repr__</code>, <code>ImmutableInterface.generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td><code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>,</td>
</tr>
<tr>
<td></td>
<td><code>implementor_names</code>, <code>implementors</code>, <code>lock</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RectDomain</code></td>
<td><code>bottom</code>, <code>diameter</code>, <code>dim</code>, <code>domain</code>, <code>height</code>, <code>left</code>, <code>lower_left</code>,</td>
</tr>
<tr>
<td></td>
<td><code>right</code>, <code>top</code>, <code>upper_right</code>, <code>volume</code>, <code>width</code></td>
</tr>
<tr>
<td><code>DomainDescription</code></td>
<td><code>boundary_types</code>, <code>has_dirichlet</code>, <code>has_neumann</code>, <code>has_robin</code></td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>add_with_arguments</code>, <code>sid</code>, <code>sid_ignore</code>, <code>with_arguments</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></td>
</tr>
</tbody>
</table>

```python
domain
left
right
top
bottom
__repr__(x) <==> repr(x)
```

### class

```python
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

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TorusDomain</code></td>
<td><code>__repr__</code>, <code>ImmutableInterface.generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td><code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>,</td>
</tr>
<tr>
<td></td>
<td><code>implementor_names</code>, <code>implementors</code>, <code>lock</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

### Attributes
boundarytypes module

```python
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

    BoundaryType.register_type, __eq__, __hash__, __ne__, __repr__, __str__
    ImmutableInterface.generate_sid, unlock, with_
    BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

    Attributes

    BoundaryType.types
    ImmutableInterface.add_with_arguments, sid, sid_ignore, with_arguments
    BasicInterface.locked, logger, logging_disabled, name, uid

    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__ () <= 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
    ImmutableInterface: generate_sid, unlock, with_
    BasicInterface: disable_logging, enable_logging, has_interface_name,
                    implementor_names, implementors, lock, __setattr__

    Attributes
    DomainDescriptionInterface: boundary_types, dim, has_dirichlet, has_neumann,
                              has_robin
    ImmutableInterface: add_with_arguments, sid, sid_ignore,
                       with_arguments
    BasicInterface: locked, logger, logging_disabled, name, uid

    dim
        The dimension of the domain

    boundary_types
        Set of BoundaryTypes the domain has.

class pymor.domaindescriptions.polygonal.CircularSectorDomain (angle, radius,
                                                             arc=BoundaryType('dirichlet'),
                                                             radii=BoundaryType('dirichlet'),
                                                             num_points=100)
    Bases: pymor.domaindescriptions.polygonal.PolygonalDomain

    Describes a circular sector domain of variable radius.

    Parameters
    angle  The angle between 0 and 2*pi of the circular sector.
    radius  The radius of the circular sector.
    arc    The BoundaryType of the arc.
    radii  The BoundaryType of the two radii.
**num_points**  The number of points of the polygonal chain approximating the circular boundary.

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CircularSectorDomain</code></td>
<td><code>__repr__</code>, <code>ImmutableInterface.generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>PolygonalDomain</code></td>
<td><code>ImmutableInterface.generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>DomainDescriptionInterface</code></td>
<td><code>BasicInterface.disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>lock</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CircularSectorDomain</code></td>
<td><code>angle</code>, <code>arc</code>, <code>num_points</code>, <code>radii</code>, <code>radius</code></td>
</tr>
<tr>
<td><code>PolygonalDomain</code></td>
<td><code>dim</code>, <code>holes</code>, <code>points</code></td>
</tr>
<tr>
<td><code>DomainDescriptionInterface</code></td>
<td><code>boundary_types</code>, <code>has_dirichlet</code>, <code>has_neumann</code>, <code>has_robin</code></td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>add_with_arguments</code>, <code>sid</code>, <code>sid_ignore</code>, <code>with_arguments</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></td>
</tr>
</tbody>
</table>

- **angle**
- **radius**
- **arc**
- **radii**
- **num_points**

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

### Class

```python
class pymor.domaindescriptions.polygonal.DiscDomain(radius, boundary=BoundaryType('dirichlet'), num_points=100)
```

**Bases**: `pymor.domaindescriptions.polygonal.PolygonalDomain`

Describes a disc domain of variable radius.

### Parameters

- **radius**  The radius of the disc.
- **boundary**  The `BoundaryType` of the boundary.
- **num_points**  The number of points of the polygonal chain approximating the boundary.

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DiscDomain</code></td>
<td><code>__repr__</code>, <code>ImmutableInterface.generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td><code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>lock</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>
Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiscDomain</td>
<td>boundary, num_points, radius</td>
</tr>
<tr>
<td>PolygonalDomain</td>
<td>dim, holes, points</td>
</tr>
<tr>
<td>DomainDescriptionInterface</td>
<td>boundary_types, has_dirichlet, has_neumann, has_robin</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

radius
boundary
num_points

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

class pymor.domaindescriptions.polygonal.PolygonalDomain(points, boundary_types, holes=[],)
Bases: pymor.domaindescriptions.interfaces.DomainDescriptionInterface

Describes a domain with a polygonal boundary and polygonal holes inside the domain.

Parameters

- **points** List of points \([x_0, x_1]\) that describe the polygonal chain that bounds the domain.
- **boundary_types** Either a dictionary \(\{\text{boundary_type: } [i_0, ...], \text{boundary_type: } [j_0, ...], ...\}\) with \(i_0, ...\) being the ids of boundary segments for a given \text{BoundaryType} \(0\) is the line connecting point \(0\) to \(1\), \(1\) is the line connecting point \(1\) to \(2\) etc.), or a function that returns the \text{BoundaryType} for a given coordinate.
- **holes** List of lists of points that describe the polygonal chains that bound the holes inside the domain.

Methods

- `PolygonalDomain.__repr__`
- `ImmutableInterface.generate_sid, unlock, with_
- `BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__`

Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolygonalDomain</td>
<td>dim, holes, points</td>
</tr>
<tr>
<td>DomainDescriptionInterface</td>
<td>boundary_types, has_dirichlet, has_neumann, has_robin</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

points
boundary_types
holes

__repr__() <=> repr(x)

pymor.domaindiscretizers package

Submodules

default module

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

Mesh a DomainDescription using an appropriate 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></td>
</tr>
<tr>
<td>CircleDomain</td>
<td>OnedGrid</td>
<td>X</td>
</tr>
<tr>
<td>PolygonalDomain</td>
<td>GmshGrid</td>
<td></td>
</tr>
</tbody>
</table>

Parameters

domain_description  A DomainDescription of the domain to mesh.
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.

gmsh module

pymor.domaindiscretizers.gmsh.discretize_gmsh(domain_description=None, geo_file=None, geo_file_path=None, msh_file_path=None, mesh_algorithm='del2d', clscale=1.0, options='', refinement_steps=0)

Mesh a DomainDescription or an already existing Gmsh GEO-file using the Gmsh mesher.
Parameters

**domain_description** A *DomainDescription* of the *PolygonalDomain* or *RectDomain* to discretize. Has to be `None` when `geo_file` is given.

**geo_file** File handle of the Gmsh Geo-file to discretize. Has to be `None` when `domain_description` is given.

**geo_file_path** Path of the created Gmsh GEO-file. When meshing a *PolygonalDomain* or *RectDomain* and `geo_file_path` is `None`, a temporary file will be created. If `geo_file` is specified, this is ignored and the path to `geo_file` will be used.

**msh_file_path** Path of the created Gmsh MSH-file. If `None`, a temporary file will be created.

**mesh_algorithm** The mesh generation algorithm to use (meshadapt, del2d, front2d).

**clscale** Mesh element size scaling factor.

**options** Other options to control the meshing procedure of Gmsh. See [http://geuz.org/gmsh/doc/texinfo/gmsh.html#Command-line-options](http://geuz.org/gmsh/doc/texinfo/gmsh.html#Command-line-options) for all available options.

**refinement_steps** Number of refinement steps to do after the initial meshing.

Returns

**grid** The generated *GmshGrid*.

**boundary_info** The generated *GmshBoundaryInfo*.

**pymor.functions package**

**Submodules**

**basic module**

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

Bases: `pymor.functions.basic.FunctionBase`

A constant *Function*

\[
f: \mathbb{R}^d \to \mathbb{R}^{\text{shape}(c)}, \quad f(x) = c
\]

Parameters

**value** The constant `c`.

**dim_domain** The dimension `d`.

**name** The name of the function.

Methods
class pymor.functions.basic.ExpressionFunction(expression,  
dim_domain=1,  
shape_range=(),  
parameter_type=None,  
name=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. 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**
ExpressionFunction

GenericFunction

FunctionBase

FunctionInterface

ImmutableInterface

BasicInterface

Parametric

Attributes

ExpressionFunction

FunctionInterface

ImmutableInterface

BasicInterface

Parametric

\_\_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

FunctionInterface

ImmutableInterface

BasicInterface

Parametric

Attributes

FunctionInterface

ImmutableInterface

BasicInterface

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__()  
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 pickled if the function it is wrapping can be pickled (cf. dumps_function). For this reason, it is usually preferable to use ExpressionFunction instead of GenericFunction.

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__
FunctionInterface: __init__
ImmutableInterface: generate_sid, unlock, with_
BasicInterface: disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__
Parametric: build_parameter_type, local_parameter, parse_parameter, strip_parameter

---

Attributes

---

5.1. pymor package
**FunctionInterface**

- `dim_domain`, `shape_range`

**ImmutableInterface**

- `add_with_arguments`, `sid`, `sid_ignore`, `with_arguments`

**BasicInterface**

- `locked`, `logger`, `logging_disabled`, `name`, `uid`

**Parametric**

- `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

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

### evaluate(x, mu=None)

Evaluate the function for given argument x and Parameter mu.

### LincombFunction

**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 either as scalars or as *ParameterFunctionals*.

**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**

- `evaluate`, `evaluate_coefficients`
- `__add__`, `__mul__`, `__neg__`, `__radd__`, `__rmul__`, `__sub__`
- `__call__`
- `__setattr__`

**Attributes**

- `coefficients`, `functions`
- `dim_domain`, `shape_range`
- `add_with_arguments`, `sid`, `sid_ignore`, `with_arguments`
- `locked`, `logger`, `logging_disabled`, `name`, `uid`
- `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

```python
functions
coefficients
evaluate(x, mu=None)
```

Evaluate the function for given argument x and Parameter mu.
**evaluate_coefficients**(*mu*)
Compute the linear coefficients for a given `Parameter mu`.

### bitmap module

**class** `pymor.functions.bitmap.BitmapFunction`(*filename*, `bounding_box=[[0.0, 0.0], [1.0, 1.0]]`, `range=[0.0, 1.0]`)
Bases: `pymor.functions.basic.FunctionBase`

Define a 2D `Function` via a grayscale image.

**Parameters**
- `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**

| `evaluate` | `__add__`, `__mul__`, `__neg__`, `__radd__`, `__rmul__`, `__sub__` |
| `__call__` | `generate_sid`, `unlock`, `with_` |
| `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__` |
| `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter` |

**Attributes**

| `dim_domain`, `shape_range` | `add_with_arguments`, `sid`, `sid_ignore`, `with_arguments` |
| `locked`, `logger`, `logging_disabled`, `name`, `uid` |
| `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric` |

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

### interfaces module

**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

5.1. pymor package
The returned values are NumPy arrays of arbitrary (but fixed) shape. Note that NumPy distinguishes between one-dimensional arrays of length 1 (with shape \((1,)\)) and zero-dimensional scalar arrays (with shape \((\)\)). In pyMOR, we usually expect scalar-valued functions to have \(\text{shape\_range} == ()\).

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 \(x.\text{ndim} == k\), then

\[
f(x, \mu)[i_0, i_1, ..., i(k-2)] == f(x[i_0, i_1, ..., i(k-2)], \mu).
\]

In particular, \(f(x, \mu).\text{shape} == x.\text{shape}[:-1] + \text{shape\_range}\).

**Methods**

<table>
<thead>
<tr>
<th>FunctionInterface</th>
<th>evaluate, <strong>call</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</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\_**(\(x, mu=None\))

Shorthand for **evaluate**.

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

Evaluate the function for given argument \(x\) and Parameter \(mu\).

**pymor.grids package**

**Submodules**

- _unstructured module_

**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
- check_boundary_types, dirichlet_boundaries, dirichlet_mask,
  neumann_boundaries, neumann_mask, no_boundary_type_mask,
  robin_boundaries, robin_mask, unique_boundary_type_mask
- CacheableInterface
  method_call, disable_caching, enable_caching
- ImmutableInterface
  e_sid, unlock, with_
- BasicInterface
  addable_logging, enable_logging, has_interface_name,
  implementor_names, implementors, lock, __setattr__

Attributes

- BoundaryInfoInterface
  boundary_types, cache_region, has_dirichlet,
  has_neumann, has_robin
- CacheableInterface
  sid_ignore
- ImmutableInterface
  add_with_arguments, sid, with_arguments
- BasicInterface
  locked, logger, logging_disabled, name, uid

mask (boundary_type, codim)

retvall[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).

Methods
### class pymor.grids.boundaryinfos.EmptyBoundaryInfo(grid)

*Bases:* pymor.grids.interfaces.BoundaryInfoInterface

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

### Methods

**EmptyBoundaryInfo**

BoundaryInfoInterface: check_boundary_types, dirichlet_boundaries, dirichlet_mask, neumann_boundaries, neumann_mask, no_boundary_type_mask, robin_boundaries, robin_mask, unique_boundary_type_mask

CacheableInterface: cached_method_call, disable_caching, enable_caching

ImmutableInterface: sid, unlock, with_

BasicInterface: disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

### Attributes

**BoundaryInfoInterface** boundary_types, cache_region, has_dirichlet, has_neumann, has_robin

**CacheableInterface** sid

**ImmutableInterface** add_with_arguments, sid, with_arguments

**BasicInterface** locked, logger, logging_disabled, name, uid

**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.grids.interfaces.BoundaryInfoInterface*

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

### Methods

**EmptyBoundaryInfo**

BoundaryInfoInterface: check_boundary_types, dirichlet_boundaries, dirichlet_mask, neumann_boundaries, neumann_mask, no_boundary_type_mask, robin_boundaries, robin_mask, unique_boundary_type_mask

CacheableInterface: cached_method_call, disable_caching, enable_caching

ImmutableInterface: sid, unlock, with_

BasicInterface: disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

### Attributes

**BoundaryInfoInterface** boundary_types, cache_region, has_dirichlet, has_neumann, has_robin

**CacheableInterface** sid

**ImmutableInterface** add_with_arguments, sid, with_arguments

**BasicInterface** locked, logger, logging_disabled, name, uid

**mask** (boundary_type, codim)

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

---

108 Chapter 5. API Documentation
class pymor.grids.boundaryinfos.SubGridBoundaryInfo(subgrid, grid, grid_boundary_info, new_boundary_type=None)

Bases: pymor.grids.interfaces.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

SubGridBoundaryInfo

BoundaryInfoInterface

boundary_types, dirichlet_boundaries, dirichlet_mask, neumann_boundaries, neumann_mask, no_boundary_type_mask, robin_boundaries, robin_mask, unique_boundary_type_mask

CacheableInterface

method_call, disable_caching, enable_caching

ImmutableInterface

disable_logging, enable_logging

BasicInterface

add_with_arguments, sid, with_arguments

Attributes

BoundaryInfoInterface

boundary_types, cache_region, has_dirichlet, has_neumann, has_robin

CacheableInterface

sid_ignore

ImmutableInterface

has_interface_name, implementor_names, implementors, lock, __setattr__

BasicInterface

locked, logger, logging_disabled, name, uid

mask (boundary_type, codim)

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

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.ReferenceElementDefaultImplementations
    Bases: object
    Provides default implementations for ReferenceElements.

gmsh module

class pymor.grids.gmsh.GmshBoundaryInfo(grid, sections)
    Bases: pymor.grids.interfaces.BoundaryInfoInterface
    BoundaryInfo for a GmshGrid.

Parameters

grid  The corresponding GmshGrid.
sections  Parsed sections of the MSH-file as returned by load_gmsh.
mask (boundary_type, codim)

\[ \text{retval}[i] = \begin{cases} 
\text{True} & \text{if the codim-codim entity of global index } i \text{ is associated to the } \text{BoundaryType} \\
\text{boundary_type} 
\end{cases} \]

class `pymor.grids.gmsh.GmshGrid` *(sections)*

Bases: `pymor.grids.unstructured.UnstructuredTriangleGrid`

An `UnstructuredTriangleGrid` built from an existing Gmsh MSH-file.

Parameters

- **sections**  Parsed sections of the MSH-file as returned by `load_gmsh`.

Methods

- `__str__`
- `embeddings`, `size`, `subentities`, `visualize`
- `bounding_box`, `centers`, `diameters`, `integration_elements`, `jacobian_inverse_transposed`, `quadrature_points`, `unit_outer_normals`, `volumes`, `volumes_inverse`
- `boundaries`, `boundary_mask`, `neighbours`, `superentities`, `superentity_indices`
- `cached_method_call`, `disable_caching`, `enable_caching`
- `generate_sid`, `unlock`, `with_`
- `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`

Attributes
**UnstructuredTriangleGrid**
- `dim`, `dim_outer`, `reference_element`

**ConformalTopologicalGridInterface**
- `cache_region`

**CacheableInterface**
- `sidIgnore`

**ImmutableInterface**
- `add_with_arguments`, `sid`, `with_arguments`

**BasicInterface**
- `locked`, `logger`, `logging_disabled`, `name`, `uid`

```python
__str__() == str(x)
```

```python
pymor.grids.gmsh.load_gmsh(gmsh_file)
```

Parse a Gmsh file and create a corresponding `GmshGrid` and `GmshBoundaryInfo`.

**Parameters**

- `gmsh_file`  File handle of the Gmsh MSH-file.

**Returns**

- `grid`  The generated `GmshGrid`.
- `boundary_info`  The generated `GmshBoundaryInfo`.

**interfaces module**

```python
class pymor.grids.interfaces.AffineGridInterface
```

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**

- `bounding_box`, `centers`, `diameters`, `embeddings`, `integration_elements`, `jacobian_inverse_transposed`, `quadrature_points`, `reference_element`, `subentities`, `unit_outer_normals`, `volumes`, `volumes_inverse`
- `boundaries`, `boundary_mask`, `neighbours`, `size`, `superentities`, `superentity_indices`
- `cached_method_call`, `disable_caching`, `enable_caching`
- `generate_sid`, `unlock`, `with_`
- `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`
Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods/Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AffineGridInterface</td>
<td>dim_outer</td>
</tr>
<tr>
<td>ConformalTopologicalGridInterface</td>
<td>cache_region, dim</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>sid_ignore</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>add_with_arguments, sid, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name,</td>
</tr>
<tr>
<td></td>
<td>uid</td>
</tr>
</tbody>
</table>

**dim_outer**
The dimension of the space into which the grid is embedded.

**bounding_box()**
returns a \((2, \text{dim}\_\text{outer})\)-shaped array containing lower/upper bounding box coordinates.

**centers**(codim)
`retval[e]` is the barycenter of the codim-codim entity with global index e.

**diameters**(codim)
`retval[e]` is the diameter of the codim-codim entity with global index e.

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

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

**quadrature_points**(codim, order=None, npoints=None, quadrature_type='default')
`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 order or has npoints integration points. To integrate a function \(f\) over \(e\) one has to form

```python
np.dot(f(quadrature_points(codim, order)[e]), reference_element(codim).
→quadrature(order)[1]) *  
integration_elements(codim)[e]). # NOQA
```

**reference_element**(codim)
The `ReferenceElement` of the codim-codim entities.

**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.

The ordering of `subentities(0, subentity_codim)[e]` has to correspond, w.r.t. the embedding of e, to the local ordering inside the reference element.

For \(\text{codim} > 0\), we provide a default implementation by calculating the subentities of e as follows:

1. Find the \(\text{codim}-1\) parent entity \(e_0\) of e with minimal global index
2. Lookup the local indices of the subentities of e inside \(e_0\) using the reference element.
3. Map these local indices to global indices using `subentities(codim - 1, subentity_codim)`.

This procedure assures that `subentities(codim, subentity_codim)[e]` has the right ordering w.r.t. the embedding determined by `e_0`, which agrees with what is returned by `embeddings(codim)`.

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

```python
volumes(codim)
retval[e] is the (dim-codim)-dimensional volume of the codim-codim entity with global index `e`.
```

```python
volumes_inverse(codim)
retval[e] = 1 / volumes(codim)[e].
```

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

`AffineGrid` with an additional `orthogonal_centers` method.

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orthogonal_centers()</code></td>
</tr>
</tbody>
</table>
| 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.)
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

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>check_boundary_types</code></td>
</tr>
<tr>
<td><code>dirichlet_boundaries</code></td>
</tr>
<tr>
<td><code>dirichlet_mask</code></td>
</tr>
<tr>
<td><code>mask</code></td>
</tr>
<tr>
<td><code>neumann_boundaries</code></td>
</tr>
<tr>
<td><code>neumann_mask</code></td>
</tr>
<tr>
<td><code>no_boundary_type_mask</code></td>
</tr>
<tr>
<td><code>robin_boundaries</code></td>
</tr>
<tr>
<td><code>robin_mask</code></td>
</tr>
<tr>
<td><code>unique_boundary_type_mask</code></td>
</tr>
</tbody>
</table>

Cacheable Interface

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cached_method_call</code></td>
</tr>
<tr>
<td><code>disable_caching</code></td>
</tr>
<tr>
<td><code>enable_caching</code></td>
</tr>
</tbody>
</table>

Immutable Interface

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>generate_sid</code></td>
</tr>
<tr>
<td><code>unlock</code></td>
</tr>
<tr>
<td><code>with_</code></td>
</tr>
</tbody>
</table>

BasicInterface

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>disable_logging</code></td>
</tr>
<tr>
<td><code>enable_logging</code></td>
</tr>
<tr>
<td><code>has_interface_name</code></td>
</tr>
<tr>
<td><code>implementor_names</code></td>
</tr>
<tr>
<td><code>implementors</code></td>
</tr>
<tr>
<td><code>lock</code></td>
</tr>
<tr>
<td><code>__setattr__</code></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boundary_types</code></td>
</tr>
<tr>
<td><code>cache_region</code></td>
</tr>
<tr>
<td><code>has_dirichlet</code></td>
</tr>
<tr>
<td><code>has_neumann</code></td>
</tr>
<tr>
<td><code>has_robin</code></td>
</tr>
</tbody>
</table>

CacheableInterface

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sid_ignore</code></td>
</tr>
</tbody>
</table>

ImmutableInterface

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_with_arguments</code></td>
</tr>
<tr>
<td><code>sid</code></td>
</tr>
<tr>
<td><code>with_arguments</code></td>
</tr>
</tbody>
</table>

BasicInterface

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>locked</code></td>
</tr>
<tr>
<td><code>logger</code></td>
</tr>
<tr>
<td><code>logging_disabled</code></td>
</tr>
<tr>
<td><code>name</code></td>
</tr>
<tr>
<td><code>uid</code></td>
</tr>
</tbody>
</table>

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

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
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]`.

---

```python
class pymor.grids.interfaces.ReferenceElementInterface

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

**ReferenceElementInterface**

- `center`, `mapped_diameter`, `quadrature`, `quadrature_info`, `quadrature_types`, `size`, `sub_reference_element`, `subentities`, `subentity_embedding`, `unit_outer_normals`, `__call__`

**CacheableInterface**

- `cached_method_call`, `disable_caching`, `enable_caching`

**ImmutableInterface**

- `generate_sid`, `unlock`, `with_`

**BasicInterface**

- `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`

---

## Attributes

**ReferenceElementInterface**

- `cache_region`, `dim`, `volume`

**CacheableInterface**

- `sid_ignore`

**ImmutableInterface**

- `add_with_arguments`, `sid`, `with_arguments`

**BasicInterface**

- `locked`, `logger`, `logging_disabled`, `name`, `uid`

---

**dim**

The dimension of the reference element

**volume**

The volume of the reference element

**__call__**(codim)

Returns the reference element of the codim-codim subentities.

**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.

---

5.1. pymor package
**size** (codim)
Number of subentities of codimension codim.

**sub_reference_element** (codim)
Returns the reference element 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-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 \(\text{subentity_codim} > 1\)' , 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** ()
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 \((\text{left}, \text{right})\) containing the left and right boundary of the domain.

**num_intervals** The number of codim-0 entities.

**Methods**

<table>
<thead>
<tr>
<th>OnedGrid</th>
<th>bounding_box, embeddings, orthogonal_centers, size, subentities, visualize, <strong>reduce</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>AffineGridInterface</td>
<td>centers, diameters, integration_elements, jacobian_inverse_transposed, quadrature_points, unit_outer_normals, volumes, volumes_inverse</td>
</tr>
<tr>
<td>ConformalTopologicalGrid</td>
<td>left, right, dictionary_mask, neighbours, superentities, superentity_indices</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>cached_method_call, disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>
OnedGrid

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

Overrides

__reduce__()

helper for pickle

__str__ () <==> str(x)

bounding_box()

returns a (2, dim_outer)-shaped array containing lower/upper bounding box coordinates.

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 retalve[e] to retalve[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.

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.

The ordering of subentities(0, subentity_codim)[e] has to correspond, w.r.t. the embedding of e, to the local ordering inside the reference element.

For codim > 0, we provide a default implementation by calculating the subentities of e as follows:

1. Find the codim-1 parent entity e_0 of e with minimal global index
2. Lookup the local indices of the subentities of e inside e_0 using the reference element.
3. Map these local indices to global indices using subentities(codim - 1, subentity_codim).

This procedure assures that subentities(codim, subentity_codim)[e] has the right ordering w.r.t. the embedding determined by e_0, which agrees with what is returned by embeddings(codim)

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

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

Parameters

U NumPy array of the data to visualize. If U.dim == 2 and len(U) > 1, the data is visualized as a time series of plots. Alternatively, a tuple of NumPy arrays 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`

---

**rect module**

---

```python
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 $(n_0, n_1)$ 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>RectGrid</th>
<th>bounding_box, embeddings, global_toStructured, orthogonalCenters, size, structuredToGlobal, subentities, vertex_coordinates, visualize, <strong>reduce</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>AffineGridInterface</td>
<td>centers, diameters, integration_elements, jacobian_inverse_transposed, quadrature_points, unit_outer_normals, volumes, volumes_inverse</td>
</tr>
<tr>
<td>ConformalTopologicalGridInterface</td>
<td>boundaries, interface_mask, neighbours, superentities, superentity_indices</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>cached_method_call, disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

---

**Attributes**
__reduce__()  
helper for pickle

__str__(x)

bounding_box()
returns a (2, dim_outer)-shaped array containing lower/upper bounding box coordinates.

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 a NumPy 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.

The ordering of subentities(0, subentity_codim)[e] has to correspond, w.r.t. the embedding of e, to the local ordering inside the reference element.

For codim > 0, we provide a default implementation by calculating the subentities of e as follows:
1. Find the codim-1 parent entity e_0 of e with minimal global index
2. Lookup the local indices of the subentities of e inside e_0 using the reference element.
3. Map these local indices to global indices using subentities(codim - 1, subentity_codim).
This procedure assures that \texttt{subentities(codim, subentity\_codim)[e]} has the right ordering w.r.t. the embedding determined by \texttt{e\_0}, which agrees with what is returned by \texttt{embeddings(codim)}.

\textbf{vertex\_coordinates}(\textit{dim})

Returns an array of the \textit{x\_dim} coordinates of the grid vertices.

I.e.

\begin{verbatim}
centers(2)[structured\_to\_global(2)[i, j]] == np.array([vertex\_coordinates(0)[i], vertex\_coordinates(1)[j]])
\end{verbatim}

\textbf{visualize}(\textit{U, codim=2, **kwargs})

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

\textbf{Parameters}

\textit{U} NumPy array of the data to visualize. If \texttt{U.dim == 2 and len(U) > 1}, the data is visualized as a time series of plots. Alternatively, a tuple of NumPy arrays can be provided, in which case a subplot is created for each entry of the tuple. The lengths of all arrays have to agree.

\textit{codim} The codimension of the entities the data in \textit{U} is attached to (either 0 or 2).

\textit{kwargs} See \texttt{visualize\_patch}

\textbf{referenceelements module}

\textbf{class pymor.grids.referenceelements.Line}

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

\textbf{Methods}

Line

\begin{verbatim}
center, mapped\_diameter, quadrature, quadrature\_info, size, sub\_reference\_element, subentities, subentity\_embedding, unit\_outer\_normals
quadrature\_types, \_call_, \_cache_, \_disable\_caching, \_enable\_caching
\end{verbatim}

ReferenceElementInterface

\begin{verbatim}
cache\_region
\end{verbatim}

CacheableInterface

\begin{verbatim}
disable\_method\_call, disable\_caching, enable\_caching
\end{verbatim}

ImmutableInterface

\begin{verbatim}
generate\_sid, unlock, with_
\end{verbatim}

BasicInterface

\begin{verbatim}
disable\_logging, enable\_logging, has\_interface\_name, implementor\_names, implementors, lock, \_setattr_
\end{verbatim}

\textbf{Attributes}

Line

\begin{verbatim}
dim, volume
\end{verbatim}

ReferenceElementInterface

\begin{verbatim}
cache\_region
\end{verbatim}

CacheableInterface

\begin{verbatim}
sid\_ignore
\end{verbatim}

ImmutableInterface

\begin{verbatim}
disable\_with\_arguments, sid, with\_arguments
\end{verbatim}

BasicInterface

\begin{verbatim}
locked, logger, logging\_disabled, name, uid
\end{verbatim}

\textbf{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 element 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 `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

Methods

<table>
<thead>
<tr>
<th>method</th>
<th>docstring</th>
</tr>
</thead>
<tbody>
<tr>
<td>center, mapped_diameter, quadrature, quadrature_info, size, sub_reference_element, subentities, subentity_embedding, unit_outer_normals</td>
<td></td>
</tr>
</tbody>
</table>

Attributes

5.1. pymor package
**Point**
- `dim, volume`

**ReferenceElementInterface**
- `cache_region`

**CacheableInterface**
- `sid_ignore`

**ImmutableInterface**
- `add_with_arguments, sid, with_arguments`

**BasicInterface**
- `locked, logger, logging_disabled, name, uid`

**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 element 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-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.Square**
- Bases: `pymor.gridsinterfaces.ReferenceElementInterface`

---

124 Chapter 5. API Documentation
**Attributes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>center, mapped_diameter, quadrature, quadrature_info, size, sub_reference_element, subentities, subentity_embedding, unit_outer_normals</td>
</tr>
<tr>
<td>ReferenceElementInterface</td>
<td>quadrature_types, <strong>call</strong></td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>cached_method_call, disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>referente_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></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 element 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 `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.Triangle
Bases: pymor.grids.interfaces.ReferenceElementInterface

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>center</td>
<td>Coordinates of the barycenter.</td>
</tr>
<tr>
<td>mapped_diameter</td>
<td>The diameter of the reference element after transforming it with the matrix $A$ (vectorized).</td>
</tr>
<tr>
<td>quadrature</td>
<td>Returns tuple $(P, W)$ where $P$ is an array of quadrature points with corresponding weights $W$. The quadrature is of order <code>order</code> or has <code>npoints</code> integration points.</td>
</tr>
<tr>
<td>quadrature_info</td>
<td>Returns a tuple of dicts $(O, N)$ where $O[\text{quadrature_type}]$ is a list of orders which are implemented for <code>quadrature_type</code> and $N[\text{quadrature_type}]$ is a list of the corresponding numbers of integration points.</td>
</tr>
<tr>
<td>size</td>
<td>Number of subentities of codimension <code>codim</code>.</td>
</tr>
<tr>
<td>sub_reference_element</td>
<td>Returns the reference element of the codim-codim subentities.</td>
</tr>
<tr>
<td>subentities</td>
<td><code>subentities(c,sc)[i,j]</code> 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.</td>
</tr>
<tr>
<td>subentity_embedding</td>
<td>Returns a tuple $(A, B)$ which defines the embedding of the codim-subentity_codim subentities into the reference element. For <code>subentity_codim &gt; 1</code>, the embedding is by default given recursively via <code>subentity_embedding(subentity_codim - 1)</code> and <code>sub_reference_element(subentity_codim - 1).subentity_embedding(1)</code> choosing always the superentity with smallest index.</td>
</tr>
</tbody>
</table>
unit_outer_normals()

    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:

    bounding_box, centers, diameters, integration_elements, jacobian_inverse_transposed, quadrature_points,
    unit_outer_normals, volumes, volumes_inverse

ConformalTopologicalGridInterface:

    boundaries, boundary_mask, neighbours, superentities, superentity_indices

CacheableInterface:

    cached_method_call, disable_caching, enable_caching

ImmutableInterface:

    generate_sid, unlock, with_

BasicInterface:

    disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

Attributes

SubGrid:

    parent_grid, reference_element

AffineGridInterface:

    dim_outer

ConformalTopologicalGridInterface:

    cache_region, dim

CacheableInterface:

    sid_ignore

ImmutableInterface:

    add_with_arguments, sid, with_arguments

BasicInterface:

    locked, logger, logging_disabled, name, uid

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 (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.

5.1. pymor package
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.

**indices_from_parent_indices** (ind, codim)
Maps a NumPy array of indicies of codim-codim entities of the parent grid to indicies of the subgrid.

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

**parent_indices** (codim)
retval[e] is the index of the e-th codim-codim entity in the parent grid.

**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.

The ordering of subentities(0, subentity_codim)[e] has to correspond, w.r.t. the embedding of e, to the local ordering inside the reference element.

For codim > 0, we provide a default implementation by calculating the subentities of e as follows:

1. Find the codim-1 parent entity e_0 of e with minimal global index
2. Lookup the local indices of the subentities of e inside e_0 using the reference element.
3. Map these local indices to global indices using subentities(codim - 1, subentity_codim).

This procedures assures that subentities(codim, subentity_codim)[e] has the right ordering w.r.t. the embedding determined by e_0, which agrees with what is returned by embeddings(codim)

**toria module**

**class** pymor.grids.tria.TriaGrid (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 triangular grid on a rectangular domain.

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

```
6---------10----------7---------11---------8
| \
| 22 10 18 | 23 11 19 |
| \ / | \ / |
| 3 14 11 6 4 15 12 7 5 |
| / \ | / \ |
| 14 2 26 | 15 3 27 |
| / \ | / \ |
| 3---------8---------4---------9---------5
| \\n| /|
```
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>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriaGrid</td>
<td>bounding_box, embeddings, orthogonal_centers, size, subentities, visualize, <strong>reduce</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>AffineGridInterface</td>
<td>centers, diameters, integration_elements, jacobian_inverse_transposed, quadrature_points, unit_outer_normals, volumes, volumes_inverse</td>
</tr>
<tr>
<td>ConformalTopologicalGridInterface</td>
<td>entity_indices, reference_mask, neighbours, superentities, superentity_indices</td>
</tr>
<tr>
<td>CacheableInterface</td>
<td>cached_method_call, disable_caching, enable_caching</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

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

**__reduce__**()

helper for pickle

**__str__**() \<=> \(str(x)\)

**bounding_box**()

returns a \((2, \text{dim}\_outer)\)-shaped array containing lower/upper bounding box coordinates.

**embeddings**\((\text{codim}=0)\)

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 codim.

subentities (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 $e$.

The ordering of $\text{subentities}(0, \text{subentity_codim})[e]$ has to correspond, w.r.t. the embedding of $e$, to the local ordering inside the reference element.

For $\text{codim} > 0$, we provide a default implementation by calculating the subentities of $e$ as follows:

1. Find the codim-1 parent entity $e_0$ of $e$ with minimal global index
2. Lookup the local indices of the subentities of $e$ inside $e_0$ using the reference element.
3. Map these local indices to global indices using $\text{subentities}(\text{codim} - 1, \text{subentity_codim})$.

This procedures assures that $\text{subentities}(\text{codim}, \text{subentity_codim})[e]$ has the right ordering w.r.t. the embedding determined by $e_0$, which agrees with what is returned by $\text{embeddings}(\text{codim})$.

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

**Parameters**

- $U$ NumPy array of the data to visualize. If $U$.dim == 2 and len($U$) > 1, the data is visualized as a time series of plots. Alternatively, a tuple of NumPy arrays 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**

**unstructured module**

**class** pymor.grids.unstructured.UnstructuredTriangleGrid (vertices, faces)
Bases: pymor.grids.interfaces.AffineGridInterface

A generic unstructured, triangular grid.

**Parameters**

- vertices A (num_vertices, 2)-shaped NumPy array containing the coordinates of all vertices in the grid. The row numbers in the array will be the global indices of the given vertices (codim 2 entities).
- faces A (num_faces, 3)-shaped NumPy array containing the global indices of the vertices which define a given triangle in the grid. The row numbers in the array will be the global indices of the given triangles (codim 0 entities).

**Methods**
**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 \(e\).

The ordering of \(subentities(0, subentity_codim)[e]\) has to correspond, w.r.t. the embedding of \(e\), to the local ordering inside the reference element.

For \(codim > 0\), we provide a default implementation by calculating the subentities of \(e\) as follows:

1. Find the codim-1 parent entity \(e_0\) of \(e\) with minimal global index
2. Lookup the local indices of the subentities of \(e\) inside \(e_0\) using the reference element.
3. Map these local indices to global indices using \(subentities(codim - 1, subentity_codim)\).

This procedures assures that \(subentities(codim, subentity_codim)[e]\) has the right ordering w.r.t. the embedding determined by \(e_0\), which agrees with what is returned by \(embeddings(codim)\).

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

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

**Parameters**
**U** NumPy array of the data to visualize. If `U.dim == 2 and len(U) > 1`, the data is visualized as a time series of plots. Alternatively, a tuple of NumPy arrays 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`

---

**pymor.gui package**

**Submodules**

**fenics module**

**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`.

---

**class pymor.gui.gl.ColorBarWidget**

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

Bases: QGLWidget

Methods

- `ColorBarWidget` initializeGL, paintEvent, resizeGL, set

---

**class pymor.gui.gl.GLPatchWidget**

```python
class pymor.gui.gl.GLPatchWidget(parent, grid, vmin=None, vmax=None, bounding_box=[0, 0], codim=2)
```

Bases: QGLWidget

Methods

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

---

**pymor.gui.gl.compile_vertex_shader**

```python
pymor.gui.gl.compile_vertex_shader(source)
```

Compile a vertex shader from source.

---

**pymor.gui.gl.link_shader_program**

```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)
```

```python
class pymor.gui.matplotlib.MatplotlibPatchWidget (parent, grid, bounding_box=None, vmin=None, vmax=None, codim=2, dpi=100)
```

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.

```python
class pymor.gui.qt.Matplotlib1DVisualizer (grid, codim=1, 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`.
- `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**
visualize($U$, discretization, title=None, legend=None, block=None)

Visualize the provided data.

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 several plots are made into the same axes. The lengths of all arrays have to agree.

- **discretization** Filled in by 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
**Attributes**

- `BasicInterface
  locked, logger, logging_disabled, name, uid`

**visualize**(*U*, `discretization`, `title=``None`, `legend=``None`, `separate_colorbars=``False`, `rescale_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.

- **rescale_colorbars** If `True`, rescale colorbars to data in each frame.

- **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**

- `PlotMainWindow
  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.stop_gui_processes**()
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 \( \text{len}(U) > 1 \), the data is visualized as a time series of plots. Alternatively, a tuple of VectorArrays can be provided, in which case 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, rescale_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 \( \text{len}(U) > 1 \), 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.
- **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.
- **rescale_colorbars**  If True, rescale colorbars to data in each frame.
- **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.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

OperatorBase: apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, jacobian, pairwise_apply2, projected, __add__, __mul__, __radd__, __str__
OperatorBase: apply, assemble_lincomb, restricted
ImmutableInterface: generate_sid, unlock, with_
BasicInterface: disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__
Parametric: build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

OperatorInterface: linear, range, solver_options, source
ImmutableInterface: add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface: locked, logger, logging_disabled, name, uid
Parametric: parameter_local_type, parameter_space, parameter_type, parametric

__add__(other)
Sum of two operators.

__mul__(other)
Product of operator by a scalar.

__radd__(other)
Sum of two operators.

__str__() <=> str(x)
apply2 ($V, U, U\_ind=None, V\_ind=None, mu=None, product=None$)

Treat the operator as a 2-form by calculating $(V, \text{op}(U))$.

In general $\text{op}.\text{apply2}(V, U)$ is equivalent to:

```
\text{product}.\text{apply2}(V, \text{op}.\text{apply}(U)).
```

In case no $\text{product}$ has been specified, $\text{op}.\text{apply2}(V, U)$ is equivalent to:

```
V.\dot{(}\text{op}.\text{apply}(U)).
```

In the latter case, assuming that $\text{op}$ is a linear operator given by multiplication with a matrix $M$, then:

```
\text{op}.\text{apply2}(V, U) = V^T \cdot M \cdot U.
```

Parameters

$V$  
* VectorArray of the left arguments $V$.

$U$  
* VectorArray of the right arguments $U$.

$V\_ind$  
The indices of the vectors in $V$ to which the operator shall be applied (see VectorArray documentation for further details).

$U\_ind$  
The indices of the vectors in $U$ to which the operator shall be applied (see VectorArray documentation for further details).

$mu$  
The Parameter for which to evaluate the operator.

$\text{product}$  
The inner product used in the expression $(V, \text{op}(U))$ given as an Operator. If None, the euclidean product is chosen.

Returns

A NumPy array with shape $(\text{len}(V\_\text{ind}), \text{len}(U\_\text{ind}))$ containing the 2-form evaluations.

apply\_adjoint ($U, ind=None, mu=None, source\_product=None, range\_product=None$)

Apply the adjoint operator.

For a linear operator $\text{op}$ the adjoint $\text{op}^*$ of $\text{op}$ is given by:

```
(\text{op}^*(v), u)_s = (v, \text{op}(u))_r,
```

where $(\ , \ )_s$ and $(\ , \ )_r$ denote the inner products on the source and range space of $\text{op}$. If $\text{op}$ and the two products are given by the matrices $M$, $P_-$ and $P_+$, then:

```
\text{op}^*(v) = P_+^*(-1) \cdot M^T \cdot P_- \cdot v,
```

with $M^T$ denoting the transposed of $M$. Thus, if $(\ , \ )_s$ and $(\ , \ )_r$ are the Euclidean inner products, $\text{op}^*v$ is simply given by multiplication of the matrix of $\text{op}$ 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).
The Parameter for which to apply the adjoint operator.

The inner product Operator on the source space. If None, the Euclidean product is chosen.

The inner product Operator on the range space. If None, the Euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.

apply_inverse(V, ind=None, mu=None, least_squares=False)
Apply the inverse operator.

Parameters

V VectorArray of vectors to which the inverse operator is applied.

ind The indices of the vectors in V to which the inverse operator shall be applied (see VectorArray documentation for further details).

mu The Parameter for which to evaluate the inverse operator.

least_squares If True, solve the least squares problem:

\[ u = \text{argmin} \| \text{op}(u) - v \|_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

Returns

VectorArray of the inverse operator evaluations.

Raises

InversionError The operator could not be inverted.

apply_inverse_adjoint(U, ind=None, mu=None, source_product=None, range_product=None, least_squares=False)
Apply the inverse adjoint operator.

Parameters

U VectorArray of vectors to which the inverse adjoint operator is applied.

ind The indices of the vectors in U to which the inverse adjoint operator shall be applied (see the VectorArray documentation for further details).

mu The Parameter for which to evaluate the inverse adjoint operator.
source_product  See apply_adjoint.
range_product  See apply_adjoint.
least_squares  If True, solve the least squares problem:

\[ v = \text{argmin} \| \text{op}^*(v) - u \|_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

Returns

VectorArray of the inverse adjoint 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.

The result of the method 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 as a new *Operator*.

**Parameters**

**U**  Length 1 *VectorArray* containing the vector for which to compute the Jacobian.

**mu**  The *Parameter* for which to compute the Jacobian.

**Returns**

Linear *Operator* representing the Jacobian.

### pairwise_apply2 (*V, U, U_ind=None, V_ind=None, mu=None, product=None*)

Treat the operator as a 2-form by calculating \((V, \text{op}(U))\).

Same as *OperatorInterface.apply2*, except that vectors from \(V\) and \(U\) are applied in pairs.

**Parameters**

**V**  *VectorArray* of the left arguments \(V\).

**U**  *VectorArray* of the right arguments \(U\).

**V_ind**  The indices of the vectors in \(V\) to which the operator shall be applied (see *VectorArray* documentation for further details).

**U_ind**  The indices of the vectors in \(U\) to which the operator shall be applied (see *VectorArray* documentation for further details).

**mu**  The *Parameter* for which to evaluate the operator.

**product**  The inner product used in the expression \((V, \text{op}(U))\) given as an *Operator*. If None, the euclidean product is chosen.

**Returns**

A *NumPy array* with shape `(len(V_ind),) == (len(U_ind),)` containing the 2-form evaluations.

### projected (*range_basis, source_basis, product=None, name=None*)

Project the operator to subspaces of the source and range space.

Given an inner product \((\cdot, \cdot)\), source vectors \(b_1, \ldots, b_N\) and range vectors \(c_1, \ldots, c_M\), the projection \(\text{op\_proj}\) of op is defined by
\[ \text{op}_i(j) = (c_i, \text{op}(b_j)) \]

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 \( \text{op}_i \) is the coordinate representation w.r.t. the \( b_i/c_i \) bases of the restriction of \( \text{op} \) to \( \text{span}(b_i) \) concatenated with the orthogonal projection onto \( \text{span}(c_i) \).

From another point of view, if \( \text{op} \) is viewed as a bilinear form (see \text{apply2}) and \( (, ) \) is the Euclidean inner product, then \( \text{op}_i \) 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 \text{NumpyMatrixOperator} will again be a \text{NumpyMatrixOperator}, only a generic \text{ProjectedOperator} can be returned in general.

A default implementation is provided in \text{OperatorBase}.

**Parameters**

- *range_basis* The vectors \( c_1, \ldots, c_M \) as a \text{VectorArray}. If None, no projection in the range space is performed.
- *source_basis* The vectors \( b_1, \ldots, b_N \) as a \text{VectorArray} or None. If None, no restriction of the source space is performed.
- *product* An \text{Operator} representing the inner product. If None, the Euclidean inner product is chosen.
- *name* Name of the projected operator.

**Returns**

The projected \text{Operator} \( \text{op}_i \).

**class** \text{pymor.operators.basicProjectedOperator} (\text{operator}, \text{range Basis}, \text{source Basis}, \text{product}=None, \text{solver_options}=None, \text{name}=None)

\text{Bases: pymor.operators.basic OperatorBase}

Generic \text{Operator} representing the projection of an \text{Operator} to a subspace.

This operator is implemented as the concatenation of the linear combination with \text{source Basis}, application of the original \text{operator} and projection onto \text{range Basis}. As such, this operator can be used to obtain a reduced basis projection of any given \text{operator}. However, no offline/online decomposition is performed, so this operator is mainly useful for testing before implementing offline/online decomposition for a specific application.

This operator is instantiated in the default implementation of \text{projected} in \text{OperatorBase} for parametric or nonlinear operators.

**Parameters**

- *operator* The \text{Operator} to project.
- *source Basis* See \text{projected}.
- *range Basis* See \text{projected}.
**product** See projected.

**name** Name of the projected operator.

### Methods

- `ProjectedOperator apply`, `assemble`, `jacobian`, `projected_to_subbasis`
- `OperatorBase apply2`, `apply_adjoint`, `apply_inverse`, `apply_inverse_adjoint`, `as_vector`, `pairwise_apply2`, `projected`, `__add__`, `__mul__`, `__radd__`, `__str__`
- `OperatorInterface assemble_lincomb`, `restricted`
- `ImmutableInterface gen_id`, `unlock`, `with__`
- `BasicInterface disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`, `__add__`, `__mul__`, `__radd__`, `__str__`
- `Parametric build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

### Attributes

- `ProjectedOperator linear`
- `OperatorInterface range`, `solver_options`, `source`
- `ImmutableInterface add_with_arguments`, `sid`, `sid_ignore`, `with_arguments`
- `BasicInterface locked`, `logger`, `logging_disabled`, `name`, `uid`
- `Parametric parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

**apply** \((U, \text{ind=}\text{None, \text{mu=}\text{None}})\)

Apply the operator to a VectorArray.

**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** \((\text{mu=}\text{None})\)

Assemble the operator for a given parameter.

The result of the method 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 as a new Operator.

Parameters
U  Length 1 VectorArray containing the vector for which to compute the Jacobian.
mu  The Parameter for which to compute the Jacobian.

Returns
Linear Operator representing the Jacobian.

projected_to_subbasis(dim_range=None, dim_source=None, name=None)
See NumpyMatrixOperator.projected_to_subbasis.

block module

class pymor.operators.block.BlockDiagonalOperator(blocks)
Bases: pymor.operators.block.BlockOperator
Block diagonal Operator of arbitrary Operators.
This is a specialization of BlockOperator for the block diagonal case.

Methods

Attributes
apply_inverse \( (V, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{least_squares}=\text{False}) \)

Apply the inverse operator.

---

**Parameters**

\( V \)  VectorArray of vectors to which the inverse operator is applied.

\( \text{ind} \) The indices of the vectors in \( V \) to which the inverse operator shall be applied (see VectorArray documentation for further details).

\( \text{mu} \) The Parameter for which to evaluate the inverse operator.

**least_squares** If True, solve the least squares problem:

\[ u = \arg\min ||\text{op}(u) - v||_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

---

**Returns**

VectorArray of the inverse operator evaluations.

---

**Raises**

InversionError  The operator could not be inverted.

---

apply_inverse_adjoint \( (U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None}, \text{least_squares}=\text{False}) \)

Apply the inverse adjoint operator.

---

**Parameters**

\( U \)  VectorArray of vectors to which the inverse adjoint operator is applied.

\( \text{ind} \) The indices of the vectors in \( U \) to which the inverse adjoint operator shall be applied (see the VectorArray documentation for further details).

\( \text{mu} \) The Parameter for which to evaluate the inverse adjoint operator.

**source_product** See apply_adjoint.

**range_product** See apply_adjoint.

**least_squares** If True, solve the least squares problem:

\[ v = \arg\min ||\text{op}^*(v) - u||_2. \]
Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate `solver_options` are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

**Returns**

`VectorArray` of the inverse adjoint operator evaluations.

**Raises**

`InversionError` The operator could not be inverted.

### class `pymor.operators.block.BlockOperator` blocks

A sparse matrix of arbitrary `Operators`.

This operator can be applied to `BlockVectorArrays` of an appropriate subtype.

**Parameters**

`blocks` Two-dimensional NumPy array where each entry is an `Operator` or None.

**Methods**

| BlockOperator | apply, apply_adjoint, hstack, vstack |
| OperatorBase | apply2, apply_inverse, apply_inverse_adjoint, as_vector, assemble, jacobian, pairwise_apply2, projected, __add__, __mul__, __radd__, __str__ |
| OperatorInterface | assemble, lincomb, restricted |
| ImmutableInterface | add_with_arguments, sid, sid_ignore, with_arguments |
| BasicInterface | disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__ |
| Parametric | build_parameter_type, local_parameter, parse_parameter, strip_parameter |

**Attributes**

| OperatorInterface | linear, range, solver_options, source |
| ImmutableInterface | add_with_arguments, sid, sid_ignore, with_arguments |
| BasicInterface | locked, logger, logging_disabled, name, uid |
| Parametric | parameter_local_type, parameter_space, parameter_type, parametric |

**apply** *(U, ind=None, mu=None)*

Apply the operator to a `VectorArray`.

**Parameters**


**apply_adjoint** *(U, ind=None, mu=None, source_product=None, range_product=None)*

Apply the adjoint operator.

For a linear operator $\mathbf{op}$ the adjoint $\mathbf{op}^*$ of $\mathbf{op}$ is given by:

$$(\mathbf{op}^*(\mathbf{v}), \mathbf{u})_s = (\mathbf{v}, \mathbf{op}(\mathbf{u}))_r,$$

where $(\ , \ )_s$ and $(\ , \ )_r$ denote the inner products on the source and range space of $\mathbf{op}$. If $\mathbf{op}$ and the two products are given by the matrices $M, P_s$ and $P_r$, then:

$$\mathbf{op}^*(\mathbf{v}) = P_s^{-1} \cdot M^T \cdot P_r \cdot \mathbf{v},$$

with $M^T$ denoting the transposed of $M$. Thus, if $(\ , \ )_s$ and $(\ , \ )_r$ are the Euclidean inner products, $\mathbf{op}^*\mathbf{v}$ is simply given by multiplication of the matrix of $\mathbf{op}$ with $\mathbf{v}$ 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 inner product *Operator* on the source space. If *None*, the Euclidean product is chosen.
- **range_product** The inner product *Operator* on the range space. If *None*, the Euclidean product is chosen.

**Returns**

*VectorArray* of the adjoint operator evaluations.

**classmethod hstack** *(operators)*

Horizontal stacking of *Operators*.

**Parameters**

- **operators** A tuple, list, array, or iterable of *Operators*.

**classmethod vstack** *(operators)*

Vertical stacking of *Operators*.
Parameters

operators  A tuple, list, array, or iterable of Operators.

cg module

This module provides some operators for continuous finite element discretizations.

class pyom.operators.cg.AdvectionOperatorP1(grid, boundary_info, advection_function=None, advection_constant=None, dirichlet_clear_columns=False, dirichlet_clear_diag=False, solver_options=None, name=None)

Bases: pyom.operators.numpy.NumpyMatrixBasedOperator

Linear advection Operator for linear finite elements.

The operator is of the form

\[(Lu)(x) = c \quad [ v(x) \ u(x) ]\]

The function \(v\) is vector-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.

advection_function  The Function \(v(x)\) with shape_range = (grid.dim_outer, ). If None, constant one is assumed.

advection_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. Otherwise they are set to one.

name  Name of the operator.

Methods
### Class Description

**AdvectionOperatorP1**

```python
class pymor.operators.cg.AdvectionOperatorP1(grid, boundary_info, advection_function=None, advection_constant=None, dirichlet_clear_columns=False, dirichlet_clear_diag=False, solver_options=None, name=None)
```

**Bases:** `pymor.operators.numpy.NumpyMatrixBasedOperator`

Linear advection operator for bilinear finite elements.

The operator is of the form

\[(Lu)(x) = c \begin{bmatrix} v(x) & u(x) \end{bmatrix}\]

The function \(v\) has to be vector-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.
- **advection_function** The `Function` \(v(x)\) with shape_range = `(grid.dim_outer,)`. If None, constant one is assumed.
- **advection_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. 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, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>AdvectionOperatorQ1</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperatorP1</td>
<td></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

class pymor.operators.cg.DiffusionOperatorP1(grid, boundary_info, diffusion_function=None, diffusion_constant=None, dirichlet_clear_columns=False, dirichlet_clear_diag=False, solver_options=None, 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 == () 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. 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, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>DiffusionOperatorP1</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

### Class

**class pymor.operators.cg.DiffusionOperatorQ1**

```python
class pymor.operators.cg.DiffusionOperatorQ1 (grid, boundary_info, diffusion_function=None, diffusion_constant=None, dirichlet_clear_columns=False, dirichlet_clear_diag=False, solver_options=None, name=None)
```

**Bases:** `pymor.operators.numpy.NumpyMatrixBasedOperator`

**Diffusion Operator** for bilinear finite elements.

The operator is of the form

\[(Lu)(x) = c \begin{bmatrix} d(x) & u(x) \end{bmatrix}\]

The function c 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 == ()` or `shape_range = (grid.\(_{\text{dim}_{\text{outer}}}, grid._{\text{dim}_{\text{outer}}}\))`. 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. 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, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</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`

Vector-like Lagrange interpolation `Operator` for continuous finite element spaces.

### Parameters

- **grid** The `Grid` on which to interpolate.
- **function** The `Function` to interpolate.

### Methods
class pymor.operators.cg.L2ProductFunctionalP1(grid, function, boundary_info=None, dirichlet_data=None, neumann_data=None, robin_data=None, order=2, solver_options=None, name=None)

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator

Linear finite element Functional representing the inner product with an L2-Function.

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, Robin boundaries by providing a robin_data tuple.

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 inner 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.

robin_data Tuple of two Functions providing the Robin parameter and boundary values, see RobinBoundaryOperator. If None, constant-zero for both functions is assumed.

order Order of the Gauss quadrature to use for numerical integration.
pyMOR, Release 0.4.2

**name** The name of the functional.

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix</td>
</tr>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>L2ProductFunctionalP1</td>
<td>range, sparse</td>
</tr>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**class** `pymor.operators.cg.L2ProductFunctionalQ1(grid, function, boundary_info=None, dirichlet_data=None, neumann_data=None, robin_data=None, order=2, name=None)`

**Bases**: `pymor.operators.numpy.NumpyMatrixBasedOperator`

Bilinear finite element *Functional* representing the inner product with an L2-Function.

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, Robin boundaries by providing a `robin_data` tuple.

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** The *Function* with which to take the inner 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.
**robin_data** Tuple of two *Functions* providing the Robin parameter and boundary values, see `RobinBoundaryOperator`. If `None`, constant-zero for both functions is assumed.

**order** Order of the Gauss quadrature to use for numerical integration.

**name** The name of the functional.

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix</td>
</tr>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>L2ProductFunctionalQ1</td>
<td>range, sparse</td>
</tr>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

```python
class pymor.operators.cg.L2ProductP1(grid, boundary_info, dirichlet_clear_rows=True, dirichlet_clear_columns=False, dirichlet_clear_diag=False, coefficient_function=None, solver_options=None, name=None)

Bases: pymor.operators.numpy.NumpyMatrixBasedOperator
```

Operator representing the L2-product between linear finite element functions.

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.

**dirichlet_clear_columns** If `True`, set columns of the system matrix corresponding to Dirichlet boundary DOFs to zero.
**dirichlet_clear_diag** If True, also set diagonal entries corresponding to Dirichlet boundary DOFs to zero. Otherwise, if either `dirichlet_clear_rows` or `dirichlet_clear_columns` is True, the diagonal entries are set to one.

**coefficient_function** Coefficient Function for product with `shape_range == ()`. If None, constant one is assumed.

**name** The name of the product.

---

### Methods

<table>
<thead>
<tr>
<th>Bases/Interface</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>apply_inverse_adjoint</code>, <code>jacobian</code>, <code>pairwise_apply2</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_lincomb</code>, <code>restricted</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>lock</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>Bases/Interface</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2ProductP1</td>
<td><code>sparse</code></td>
</tr>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td><code>linear</code></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td><code>range</code>, <code>solver_options</code>, <code>source</code></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td><code>add_with_arguments</code>, <code>sid</code>, <code>sid_ignore</code>, <code>with_arguments</code></td>
</tr>
<tr>
<td>BasicInterface</td>
<td><code>locked</code>, <code>logger</code>, <code>logging_disabled</code>, <code>name</code>, <code>uid</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`, `coefficient_function=None`, `solver_options=None`, `name=None`)  

**Bases:** `pymor.operators.numpy.NumpyMatrixBasedOperator`  

**Operator** representing the L2-product between bilinear finite element functions.  
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.
- **dirichlet_clear_columns** If True, set columns of the system matrix corresponding to Dirichlet boundary DOFs to zero.
**dirichlet_clear_diag**  If `True`, also set diagonal entries corresponding to Dirichlet boundary DOFs to zero. Otherwise, if either `dirichlet_clear_rows` or `dirichlet_clear_columns` is `True`, the diagonal entries are set to one.

**coefficient_function**  Coefficient `Function` for product with `shape_range == ()`. If `None`, constant one is assumed.

**name**  The name of the product.

---

### 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, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>L2ProductQ1</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

---

**class**  `pymor.operators.cg.RobinBoundaryOperator(grid, boundary_info, robin_data=None, order=2, solver_options=None, name=None)`

Bases: `pymor.operators.numpy.NumpyMatrixBasedOperator`

Robin boundary `Operator` for linear finite elements.

The operator represents the contribution of Robin boundary conditions to the stiffness matrix, where the boundary condition is supposed to be given in the form

\[-[d(x) u(x)] n(x) = c(x) (u(x) - g(x))\]

\(d\) and \(n\) are the diffusion function (see `DiffusionOperatorP1`) and the unit outer normal in \(x\), while \(c\) is the (scalar) Robin parameter function and \(g\) is the (also scalar) Robin boundary value function.

**Parameters**

**grid**  The `Grid` over which to assemble the operator.

**boundary_info**  `BoundaryInfo` for the treatment of Dirichlet boundary conditions.
robin_data Tuple providing two Functions that represent the Robin parameter and boundary value function. If None, the resulting operator is zero.

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, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>RobinBoundaryOperator</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>linear</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

constructions module

Module containing some constructions to obtain new operators from old ones.

class pymor.operators.constructions.AdjointOperator (operator, source_product=None, range_product=None, name=None, with_apply_inverse=True, solver_options=None)

Bases: pymor.operators.basic.OperatorBase

Represents the adjoint of a given linear Operator.

See apply_adjoint.

Parameters

operator The Operator of which the adjoint is formed.

source_product If not None, inner product Operator for the source VectorSpace w.r.t. which to take the adjoint.

range_product If not None, inner product Operator for the range VectorSpace w.r.t. which to take the adjoint.
name  If not None, name of the operator.

with_apply_inverse  If True, provide own apply_inverse and apply_inverse_adjoint implementations by calling these methods on the given operator. (Is set to False in the default implementation of and apply_inverse_adjoint.)

solver_options  When with_apply_inverse is False, the solver_options to use for the apply_inverse default implementation.

---

Methods

<table>
<thead>
<tr>
<th>Operator</th>
<th>Methods</th>
<th>Methods</th>
<th>Methods</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdjointOperator</td>
<td>apply, apply_adjoint, apply_inverse, apply_inverse_adjoint, projected</td>
<td>OperatorBase</td>
<td>apply2, as_vector, assemble, jacobian, pairwise_apply2, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
<td>OperatorInterface</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with_</td>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
<td>Parametric</td>
</tr>
</tbody>
</table>

---

Attributes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Attributes</th>
<th>Attributes</th>
<th>Attributes</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdjointOperator</td>
<td>linear</td>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
<td>ImmutableInterface</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
<td></td>
</tr>
</tbody>
</table>

---

apply \((U, \text{ind}=\text{None}, \mu=\text{None})\)

Apply the operator to a VectorArray.

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 \(\mathbf{op}\) the adjoint \(\mathbf{op}^*\) of \(\mathbf{op}\) is given by:
\[(\text{op}^*(v), u)_s = (v, \text{op}(u))_r,\]

where \(( , )_s\) and \(( , )_r\) denote the inner products on the source and range space of \(\text{op}\). If \(\text{op}\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[\text{op}^*(v) = P_s^{-1} \times M^T \times P_r \times v,\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \(( , )_s\) and \(( , )_r\) are the Euclidean inner products, \(\text{op}^*v\) is simply given by multiplication of the matrix of \(\text{op}\) 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 inner product `Operator` on the source space. If `None`, the Euclidean product is chosen.
- \(\text{range\_product}\) The inner product `Operator` on the range space. If `None`, the Euclidean product is chosen.

**Returns**

`VectorArray` of the adjoint operator evaluations.

**apply\_inverse** \((V, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{least\_squares}=\text{False})\)

Apply the inverse operator.

**Parameters**

- \(V\) `VectorArray` of vectors to which the inverse operator is applied.
- \(\text{ind}\) The indices of the vectors in \(V\) to which the inverse operator shall be applied (see `VectorArray` documentation for further details).
- \(\text{mu}\) The `Parameter` for which to evaluate the inverse operator.
- \(\text{least\_squares}\) If `True`, solve the least squares problem:

\[u = \arg\min \| \text{op}(u) - v \|_2.\]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate `solver\_options` are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

**Returns**

`VectorArray` of the inverse operator evaluations.
Raises
InversionError The operator could not be inverted.

apply_inverse_adjoint (U, ind=None, mu=None, source_product=None, range_product=None, least_squares=False)
Apply the inverse adjoint operator.

Parameters
U VectorArray of vectors to which the inverse adjoint operator is applied.
ind The indices of the vectors in U to which the inverse adjoint operator shall be applied (see the VectorArray documentation for further details).
mu The Parameter for which to evaluate the inverse adjoint operator.
source_product See apply_adjoint.
range_product See apply_adjoint.
least_squares If True, solve the least squares problem:

\[ v = \arg\min ||op^*(v) - u||_2.\]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

Returns
VectorArray of the inverse adjoint operator evaluations.

Raises
InversionError The operator could not be inverted.

projected (range_basis, source_basis, product=None, name=None)
Project the operator to subspaces of the source and range space.

Given an inner product ( , ), source vectors b_1, ..., b_N and range vectors c_1, ..., c_M, the projection op_proj of op is defined by

\[
[ \text{op proj}(e_j) ]_i = ( c_i, \text{op}(b_j) )
\]

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 op_proj is the coordinate representation w.r.t. the b_i/c_i bases of the restriction of op to span(b_i) concatenated with the orthogonal projection onto span(c_i).

From another point of view, if op is viewed as a bilinear form (see apply2) and ( , ) is the Euclidean inner product, then op_proj represents the matrix of the bilinear form restricted span(b_i) / span(c_i) (w.r.t. the b_i/c_i bases).

5.1. pymor package
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 ProjectedOperator can be returned in general.

A default implementation is provided in OperatorBase.

Parameters

range_basis The vectors $c_1, \ldots, c_M$ as a VectorArray. If None, no projection in the range space is performed.

source_basis The vectors $b_1, \ldots, b_N$ as a VectorArray or None. If None, no restriction of the source space is performed.

product An Operator representing the inner product. If None, the Euclidean inner product is chosen.

name Name of the projected operator.

Returns

The projected Operator $\text{op}_\text{proj}$.

class pymor.operators.constructions.ComponentProjection(components, source, name=None)

Bases: pymor.operators.basic.OperatorBase

Operator representing the projection of a VectorArray 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_restricted

OperatorBase apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint,
as_vector, assemble, jacobian, pairwise_apply2, projected, __add__,
__mul__, __radd__, __str__

ImmutableComponentProjection lincomb

BasicInterface__eq__, __hash__, __str__, __repr__, disable_logging, enable_logging, has_interface_name,
implementor_names, implementors, lock, __setattr__

Parametric__build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes
**apply** $(U, \text{ind}=\text{None}, \text{mu}=\text{None})$

Apply the operator to a VectorArray.

**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.

**restricted** $(\text{dofs})$

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.

**Parameters**

- $\text{dofs}$ 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.
Operator representing the concatenation of two Operators.

Parameters

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>Concatenation</th>
<th>apply, apply_adjoint, jacobian, projected, restricted</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_inverse, apply_inverse_adjoint, as_vector, assemble, pairwise_apply2, <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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

Attributes

| OperatorInterface | linear, range, solver_options, source |
| ImmutableInterface | add_with_arguments, sid, sid_ignore, with_arguments |
| BasicInterface | locked, logger, logging_disabled, name, uid |
| Parametric | parameter_local_type, parameter_space, parameter_type, parametric |

apply \((U, \text{ind}=\text{None}, \text{mu}=\text{None})\)

Apply the operator to a VectorArray.

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 \(\text{op}\) the adjoint \(\text{op}^*\) of \(\text{op}\) is given by:

\[
(\text{op}^*(v), u)_s = (v, \text{op}(u))_r,
\]
where $(\cdot,\cdot)_s$ and $(\cdot,\cdot)_r$ denote the inner products on the source and range space of $\text{op}$. If $\text{op}$ and the two products are given by the matrices $M, P_s$ and $P_r$, then:

$$\text{op}^*(v) = P_s^{-1} \cdot M^T \cdot P_r \cdot v,$$

with $M^T$ denoting the transposed of $M$. Thus, if $(\cdot,\cdot)_s$ and $(\cdot,\cdot)_r$ are the Euclidean inner products, $\text{op}^*v$ is simply given by multiplication of the matrix of $\text{op}$ 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 inner product `Operator` on the source space. If `None`, the Euclidean product is chosen.
- **range_product** The inner product `Operator` on the range space. If `None`, the Euclidean product is chosen.

### Returns

`VectorArray` of the adjoint operator evaluations.

### jacobian

- **(U, mu=None)**
  Return the operator’s Jacobian as a new `Operator`.

### Parameters

- **U** Length 1 `VectorArray` containing the vector for which to compute the Jacobian.
- **mu** The `Parameter` for which to compute the Jacobian.

### Returns

Linear `Operator` representing the Jacobian.

### projected

- **(range_basis, source_basis, product=None, name=None)**
  Project the operator to subspaces of the source and range space.

  Given an inner product $(\cdot,\cdot)$, source vectors $b_1, \ldots, b_N$ and range vectors $c_1, \ldots, c_M$, the projection $\text{op}_\text{proj}$ of $\text{op}$ is defined by

  $$\left[ \text{op}_\text{proj}(e_j) \right]_i = (c_i, \text{op}(b_j))$$

  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 $\text{op}_\text{proj}$ is the coordinate representation w.r.t. the $b_i/c_i$ bases of the restriction of $\text{op}$ to $\text{span}(b_1)$ concatenated with the orthogonal projection onto $\text{span}(c_i)$.  

### 5.1. pymor package

165
From another point of view, if \( \text{op} \) is viewed as a bilinear form (see \text{apply2} ) and \( (, ) \) is the Euclidean inner product, then \( \text{op\_proj} \) 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 \text{NumpyMatrixOperator} will again be a \text{NumpyMatrixOperator}, only a generic \text{ProjectedOperator} can be returned in general.

A default implementation is provided in \text{OperatorBase}.

**Parameters**

- **range\_basis** The vectors \( c_1, \ldots, c_M \) as a \text{VectorArray}. If \text{None}, no projection in the range space is performed.
- **source\_basis** The vectors \( b_1, \ldots, b_N \) as a \text{VectorArray} or \text{None}. If \text{None}, no restriction of the source space is performed.
- **product** An \text{Operator} representing the inner product. If \text{None}, the Euclidean inner product is chosen.
- **name** Name of the projected operator.

**Returns**

- The projected \text{Operator} \( \text{op\_proj} \).

**restricted**(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 \text{VectorArray} \( U \) in \text{self.source} the following is true:

\[
\text{self.apply}(U, \text{mu}) . \text{components}(\text{dofs}) == \text{restricted\_op}.\text{apply}(\text{NumpyVectorArray}(U.\text{components}(\text{source\_dofs})), \text{mu})
\]

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.

**Parameters**

- **dofs** One-dimensional \text{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 \text{NumPy array} of source degrees of freedom as defined above.
class pymor.operators.constructions.ConstantOperator(value, source, 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.
name Name of the operator.

Methods

apply, jacobian, projected, projected_to_subbasis, restricted

apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint,
as_vector, assemble, pairwise_apply2, __add__, __mul__, __radd__,
__str__

OperatorInterface assemble_lincomb

ImmutableInterface add_with_arguments, sid, sid_ignore, with_

BasicInterface enable_logging, has_interface_name,
implementor_names, implementors, lock, __setattr__

Parametric build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes

ConstantOperator linear

OperatorInterface range, solver_options, source

ImmutableInterface add_with_arguments, sid, sid_ignore, with_arguments

BasicInterface locked, logger, logging_disabled, name, uid

Parametric parameter_local_type, parameter_space, parameter_type,
parametric

apply (U, ind=None, mu=None)

Apply the operator to a VectorArray.

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 as a new Operator.
Parameters

- **U** Length 1 `VectorArray` containing the vector for which to compute the Jacobian.
- **mu** The `Parameter` for which to compute the Jacobian.

Returns

A `LinearOperator` representing the Jacobian.

The `projected` method projects the operator to subspaces of the source and range space. Given an inner product `( , )`, source vectors $b_1, \ldots, b_N$ and range vectors $c_1, \ldots, c_M$, the projection $\text{op}_\text{proj}$ of $\text{op}$ is defined by

$$[ \text{op}_\text{proj}(e_j) ]_i = ( c_i, \text{op}(b_j) )$$

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 $\text{op}_\text{proj}$ is the coordinate representation w.r.t. the $b_i/c_i$ bases of the restriction of $\text{op}$ to $\text{span}(b_i)$ concatenated with the orthogonal projection onto $\text{span}(c_i)$.

From another point of view, if $\text{op}$ is viewed as a bilinear form (see `apply2`) and $( , )$ is the Euclidean inner product, then $\text{op}_\text{proj}$ 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 `ProjectedOperator` can be returned in general.

A default implementation is provided in `OperatorBase`.

The `restricted` method restricts 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 `source_dofs` such that for any `VectorArray U` in `self.source` the following is true:
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.

**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.

---

```python
self.apply(U, mu).components(dofs)
== restricted_op.apply(NumpyVectorArray(U.components(source_dofs)), mu))
```

---

### Class pymor.operators.constructions.FixedParameterOperator

Bases: pymor.operators.basic.OperatorBase

Makes an Operator Parameter-independent by setting a fixed Parameter.

**Parameters**

- **operator**  
  The Operator to wrap.

- **mu**  
  The fixed Parameter that will be fed to the apply method of operator.

**Methods**

```plaintext
FixedParameterOperator.apply, apply_adjoint, apply_inverse, apply_inverse_adjoint,
jacobian, restricted
OperatorBase.apply2, as_vector, assemble, pairwise_apply2.projected,
__add__, __mul__, __radd__, __str__
OperatorInterface.assemble_lincomb
ImmutableInterface.generate_sid, unlock, with_
BasicInterface.disable_logging, enable_logging, has_interface_name,
implementor_names, implementors, lock, __setattr__
Parametric.build_parameter_type, local_parameter, parse_parameter,
strip_parameter
```
apply \((U, \text{ind}=\text{None}, \text{mu}=\text{None})\)

Apply the operator to a VectorArray.

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 \((U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None})\)

Apply the adjoint operator.

For a linear operator \(\text{op}\) the adjoint \(\text{op}^*\) of \(\text{op}\) is given by:

\[
(\text{op}^*(v), u)_s = (v, \text{op}(u))_r,
\]

where \((\ , \ )_s\) and \((\ , \ )_r\) denote the inner products on the source and range space of \(\text{op}\). If \(\text{op}\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[
\text{op}^*(v) = P_s^{-1} \cdot M^T \cdot P_r \cdot v,
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \((\ , \ )_s\) and \((\ , \ )_r\) are the Euclidean inner products, \(\text{op}^*v\) is simply given by multiplication of the matrix of \(\text{op}\) 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 inner product Operator on the source space. If None, the Euclidean product is chosen.

range_product  The inner product Operator on the range space. If None, the Euclidean product is chosen.

Returns
apply_inverse(V, ind=None, mu=None, least_squares=False)

Apply the inverse operator.

Parameters

V : VectorArray of vectors to which the inverse operator is applied.
ind : The indices of the vectors in V to which the inverse operator shall be applied (see VectorArray documentation for further details).
mu : The Parameter for which to evaluate the inverse operator.
least_squares : If True, solve the least squares problem:

\[ \text{argmin} \| \text{op}(u) - v \|_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

Returns

VectorArray of the inverse operator evaluations.

Raises

InversionError : The operator could not be inverted.

apply_inverse_adjoint(U, ind=None, mu=None, source_product=None, range_product=None, least_squares=False)

Apply the inverse adjoint operator.

Parameters

U : VectorArray of vectors to which the inverse adjoint operator is applied.
ind : The indices of the vectors in U to which the inverse adjoint operator shall be applied (see the VectorArray documentation for further details).
mu : The Parameter for which to evaluate the inverse adjoint operator.
source_product : See apply_adjoint.
range_product : See apply_adjoint.
least_squares : If True, solve the least squares problem:

\[ \text{argmin} \| \text{op}^*(v) - u \|_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.
Returns

VectorArray of the inverse adjoint operator evaluations.

Raises

InversionError  The operator could not be inverted.

jacobian (U, mu=None)

Return the operator's Jacobian as a new Operator.

Parameters

U  Length 1 VectorArray containing the vector for which to compute the Jacobian.
mu  The Parameter for which to compute the Jacobian.

Returns

Linear Operator representing the Jacobian.

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.

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.IdentityOperator(space, name=None)

Bases: pymor.operators.basic.OperatorBase

The identity Operator.

In other words:

\[
op\text{.apply}(U) = U
\]

Parameters

space  The VectorSpace the operator acts on.

name  Name of the operator.

Methods

IdentityOperator
apply, apply_adjoint, apply_inverse, apply_inverse_adjoint,
assemble, assemble_lincomb, restricted

OperatorBase
apply2, as_vector, jacobian, pairwise_apply2, projected, __add__,
__mul__, __radd__, __str__

ImmutableInterface
generate_sid, unlock, with_

BasicInterface
disable_logging, enable_logging, has_interface_name,
implementor_names, implementors, lock, __setattr__

Parametric
build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes

IdentityOperator
linear

OperatorInterface
range, solver_options, source

ImmutableInterface
add_with_arguments, sid, sid_ignore, with_arguments

BasicInterface
locked, logger, logging_disabled, name, uid

Parametric
parameter_local_type, parameter_space, parameter_type,
parametric

apply(U, ind=None, mu=None)

Apply the operator to a VectorArray.

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 \( \text{op} \) the adjoint \( \text{op}^* \) of \( \text{op} \) is given by:

\[
(\text{op}^*(v), u)_s = (v, \text{op}(u))_r,
\]

where \( (\ , \ )_s \) and \( (\ , \ )_r \) denote the inner products on the source and range space of \( \text{op} \). If \( \text{op} \) and the two products are given by the matrices \( M, P_s \) and \( P_r \), then:

\[
\text{op}^*(v) = P_s^{-1} * M^T * P_r * v,
\]

with \( M^T \) denoting the transposed of \( M \). Thus, if \( (\ , \ )_s \) and \( (\ , \ )_r \) are the Euclidean inner products, \( \text{op}^*v \) is simply given by multiplication of the matrix of \( \text{op} \) 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 inner product *Operator* on the source space. If *None*, the Euclidean product is chosen.
- **range_product** The inner product *Operator* on the range space. If *None*, the Euclidean product is chosen.

**Returns**

*VectorArray* of the adjoint operator evaluations.

**apply_inverse** *(V, ind=None, mu=None, least_squares=False)*

Apply the inverse operator.

**Parameters**

- **V** *VectorArray* of vectors to which the inverse operator is applied.
- **ind** The indices of the vectors in \( V \) to which the inverse operator shall be applied (see *VectorArray* documentation for further details).
- **mu** The *Parameter* for which to evaluate the inverse operator.
- **least_squares** If *True*, solve the least squares problem:

\[
u = \arg\min ||\text{op}(u) - v||_2.
\]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate *solver_options* are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

**Returns**
VectorArray of the inverse operator evaluations.

Raises

InversionError The operator could not be inverted.

apply_inverse_adjoint (U, ind=None, mu=None, source_product=None, range_product=None, least_squares=False)
Apply the inverse adjoint operator.

Parameters

U VectorArray of vectors to which the inverse adjoint operator is applied.
ind The indices of the vectors in U to which the inverse adjoint operator shall be applied (see the VectorArray documentation for further details).
mu The Parameter for which to evaluate the inverse adjoint operator.
source_product See apply_adjoint.
range_product See apply_adjoint.
least_squares If True, solve the least squares problem:

\[ v = \text{argmin}_{v} ||\text{op}^*(v) - u||_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

Returns

VectorArray of the inverse adjoint operator evaluations.

Raises

InversionError The operator could not be inverted.

assemble (mu=None)
Assemble the operator for a given parameter.

The result of the method 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.
assemble_lincomb(operators, coefficients, solver_options=None, 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.

solver_options solver_options for the assembled operator.

name Name of the assembled operator.

**Returns**

The assembled Operator if assembly is possible, otherwise None.

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.

**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.InducedNorm(product, raise_negative, tol, name)

Bases: pymor.core.interfaces.ImmutableInterface, pymor.parameters.base.Parametric

Instantiated by induced_norm. Do not use directly.

Methods

InducedNorm __call__
ImmutableInterface generate_sid, unlock, with_
BasicInterface disable_logging, enable_logging, has_interface_name,
implementor_names, implementors, lock, __setattr__
Parametric build_parameter_type, local_parameter, parse_parameter,
strip_parameter

Attributes

ImmutableInterface add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface locked, logger, logging_disabled, name, uid
Parametric parameter_local_type, parameter_space, parameter_type, parametric

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

class pymor.operators.constructions.LincombOperator(operators, coefficients, solver_options=None, name=None)

Bases: pymor.operators.basic.OperatorBase

Linear combination of arbitrary Operators.

This Operator represents a (possibly Parameter dependent) linear combination of a given list of 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
### apply

Apply the operator to a `VectorArray`.

#### 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

Treat the operator as a 2-form by calculating `(V, op(U))`.

In general `op.apply2(V, U)` is equivalent to:

\[
\text{product.apply2}(V, \text{op.apply}(U)).
\]

In case no `product` has been specified, `op.apply2(V, U)` is equivalent to:

\[
V\cdot\text{op.apply}(U).
\]

In the latter case, assuming that `op` is a linear operator given by multiplication with a matrix `M`, then:

\[
\text{op.apply2}(V, U) = V^\top M U.
\]

#### Parameters
V  VectorArray of the left arguments V.
U  VectorArray of the right right arguments U.

V_ind  The indices of the vectors in V to which the operator shall be applied (see VectorArray documentation for further details).
U_ind  The indices of the vectors in U to which the operator shall be applied (see VectorArray documentation for further details).

mu  The Parameter for which to evaluate the operator.

product  The inner product used in the expression \((V, \, \, op(U))\) given as an Operator. If None, the euclidean product is chosen.

---

Returns
A NumPy array with shape \((\text{len}(V\_\text{ind}), \, \text{len}(U\_\text{ind}))\) containing the 2-form 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 \(op\) the adjoint \(op^*\) of \(op\) is given by:

\[
\langle op^*(v), \, u \rangle_s = \langle v, \, op(u) \rangle_r,
\]

where \(\langle \, , \rangle_s\) and \(\langle \, , \rangle_r\) denote the inner products on the source and range space of \(op\). If \(op\) and the two products are given by the matrices \(M, \, P_s\) and \(P_r\), then:

\[
\text{op}^*(v) = P_s^{-1} \times M^T \times P_r \times v,
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \(\langle \, , \rangle_s\) and \(\langle \, , \rangle_r\) are the Euclidean inner products, \(op^*v\) is simply given by multiplication of the matrix of \(op\) 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 inner product Operator on the source space. If None, the Euclidean product is chosen.

range_product  The inner 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

\[
\text{self.as_vector}(\mu)\cdot \text{dot}(U) = \text{self.apply}(U, \mu)
\]

holds, whereas in the case of a vector-like operator we have

\[
\text{self.as_vector}(\mu) = \text{self.apply}(\text{NumpyVectorArray}(1), \mu).
\]

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** \((\text{mu}=\text{None})\)

Assemble the operator for a given parameter.

The result of the method 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 for a given Parameter.

Parameters

- **mu**  Parameter for which to compute the linear coefficients.

Returns

- List of linear coefficients.

**jacobian** \((U, \text{mu}=\text{None})\)

Return the operator’s Jacobian as a new Operator.
Parameters

- **U** Length 1 *VectorArray* containing the vector for which to compute the Jacobian.
- **mu** The *Parameter* for which to compute the Jacobian.

Returns

Linear *Operator* representing the Jacobian.

**pairwise_apply2** (*V*, *U*, *U_ind=None*, *V_ind=None*, *mu=None*, *product=None*)

Treat the operator as a 2-form by calculating *(V, op(U))*.

Same as *OperatorInterface.apply2* except that vectors from *V* and *U* are applied in pairs.

Parameters

- **V** *VectorArray* of the left arguments *V*.
- **U** *VectorArray* of the right right arguments *U*.
- **V_ind** The indices of the vectors in *V* to which the operator shall be applied (see *VectorArray* documentation for further details).
- **U_ind** The indices of the vectors in *U* to which the operator shall be applied (see *VectorArray* documentation for further details).
- **mu** The *Parameter* for which to evaluate the operator.
- **product** The inner product used in the expression *(V, op(U))* given as an *Operator*. If *None*, the euclidean product is chosen.

Returns

A *NumPy array* with shape (len(*V_ind*),) == (len(*U_ind*),) containing the 2-form evaluations.

**projected** (*range_basis*, *source_basis*, *product=None*, *name=None*)

Project the operator to subspaces of the source and range space.

Given an inner product *( , )*, source vectors b_1, ..., b_N and range vectors c_1, ..., c_M, the projection op_proj of op is defined by

\[
\{ \text{op_proj}(e_j) \}_i = ( c_i, \text{op}(b_j) )
\]

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 op_proj is the coordinate representation w.r.t. the b_i/c_i bases of the restriction of op to span(b_i) concatenated with the orthogonal projection onto span(c_i).

From another point of view, if op is viewed as a bilinear form (see apply2) and *( , )* is the Euclidean inner product, then op_proj represents the matrix of the bilinear form restricted span(b_i) / spanc(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 `ProjectedOperator` can be returned in general.

A default implementation is provided in `OperatorBase`.

### Parameters

- **range_basis** The vectors $c_1, \ldots, c_M$ as a `VectorArray`. If `None`, no projection in the range space is performed.
- **source_basis** The vectors $b_1, \ldots, b_N$ as a `VectorArray` or `None`. If `None`, no restriction of the source space is performed.
- **product** An `Operator` representing the inner product. If `None`, the Euclidean inner product is chosen.
- **name** Name of the projected operator.

### Returns

The projected `Operator` $\text{op}_\text{proj}$.

### projected_to_subbasis

Function `projected_to_subbasis(dim_range=None, dim_source=None, name=None)`


### class pymor.operators.constructions.SelectionOperator(operators, parameter_functional, boundaries, name=None)

Bases: `pymor.operators.basic.OperatorBase`  

An `Operator` selected from a list of `Operators`. `operators[i]` is used if `parameter_functional(mu)` is less or equal than `boundaries[i]` and greater than `boundaries[i-1]`:

```
-\infty ------- boundaries[i] ------- boundaries[i+1] ------- \infty
    |                             |
--- operators[i] ---|---- operators[i+1] ----|---- operators[i+2]
    |                             |
```

### Parameters

- **operators** List of `Operators` from which one `Operator` is selected based on the given `Parameter`.  
- **parameter_functional** The `ParameterFunctional` used for the selection of one `Operator`.  
- **boundaries** The interval boundaries as defined above.  
- **name** Name of the operator.
apply \((U, \text{ind}=\text{None}, \mu=\text{None})\)

Apply the operator to a VectorArray.

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 \((U, \text{ind}=\text{None}, \mu=\text{None}, \text{source\_product}=\text{None}, \text{range\_product}=\text{None})\)

Apply the adjoint operator.

For a linear operator \(\text{op}\) the adjoint \(\text{op}^*\) of \(\text{op}\) is given by:

\[
(\text{op}^*(v), u)_s = (v, \text{op}(u))_r,
\]

where \(( , )_s\) and \(( , )_r\) denote the inner products on the source and range space of \(\text{op}\). If \(\text{op}\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[
\text{op}^*(v) = P_s^{-1} \times M^T \times P_r \times v,
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \(( , )_s\) and \(( , )_r\) are the Euclidean inner products, \(\text{op}^*v\) is simply given by multiplication of the matrix of \(\text{op}\) 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 inner product Operator on the source space. If None, the Euclidean product is chosen.

range_product  The inner 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

```
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.

The result of the method 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.

**projected** *(range_basis, source_basis, product=None, name=None)*

Project the operator to subspaces of the source and range space.

Given an inner product ( , ), source vectors b_1, ..., b_N and range vectors c_1, ..., c_M, the projection op_proj of op is defined by

\[
[ \text{op_proj}(e_j) ]_i = ( c_i, \text{op}(b_j) )
\]

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 op_proj is the coordinate representation w.r.t. the b_i/c_i bases of the restriction of op to span(b_i) concatenated with the orthogonal projection onto span(c_i).

From another point of view, if op is viewed as a bilinear form (see apply2) and ( , ) is the Euclidean inner product, then op_proj 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 ProjectedOperator can be returned in general.

A default implementation is provided in OperatorBase.

**Parameters**

- **range_basis** The vectors c_1, ..., c_M as a VectorArray. If None, no projection in the range space is performed.
- **source_basis** The vectors b_1, ..., b_N as a VectorArray or None. If None, no restriction of the source space is performed.
- **product** An Operator representing the inner product. If None, the Euclidean inner product is chosen.
- **name** Name of the projected operator.

**Returns**

The projected Operator op_proj.

**class** *pymor.operators.constructions.VectorArrayOperator*(array, transposed=False, name=None)*

Bases: pymor.operators.basic.OperatorBase

Wraps a VectorArray as an Operator.

If transposed is 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 inner products of the argument with the vectors in the given array.

**Parameters**
array  The `VectorArray` which is to be treated as an operator.

transposed  See description above.

name  The name of the operator.

Methods

| VectorArrayOperator           | apply, apply_adjoint, apply_inverse_adjoint, as_vector, 
|                              | assemble_lincomb, restricted |
| OperatorBase                 | apply2, apply_inverse, assemble, jacobian, pairwise_apply2, 
|                              | projected, __add__, __mul__, __radd__, __str__ |
| ImmutableInterface           | assemble_sid, unlock, with, 
| BasicInterface               | disable_logging, enable_logging, has_interface_name, 
|                              | implementor_names, implementors, lock, __setattr__ |
| Parametric                   | build_parameter_type, local_parameter, parse_parameter, 
|                              | strip_parameter |

Attributes

| VectorArrayOperator         | linear |
| OperatorInterface          | range, solver_options, source |
| ImmutableInterface         | add_with_arguments, sid, sid_ignore, with_arguments |
| BasicInterface             | locked, logger, logging_disabled, name, uid |
| Parametric                 | parameter_local_type, parameter_space, parameter_type, 
|                            | parametric |

**apply** (*U*, *ind=None*, *mu=None*)

Apply the operator to a `VectorArray`.

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 `op` the adjoint `op^*` of `op` is given by:

\[(op^*(v), u)_s = (v, op(u))_r,\]

where `( , )_s` and `( , )_r` denote the inner products on the source and range space of `op`. If `op` and the two products are given by the matrices `M`, `P_s` and `P_r`, then:
\[ \text{op}^*(v) = P_s^{-1} \cdot M^T \cdot P_r \cdot v, \]

with \( M^T \) denoting the transposed of \( M \). Thus, if \((\ ,\ )_s\) and \((\ ,\ )_r\) are the Euclidean inner products, \( \text{op}^*v \) is simply given by multiplication of the matrix of \( \text{op} \) with \( v \) from the left.

### Parameters

- **U** \( \text{VectorArray} \) 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 \( \text{VectorArray} \) documentation for further details).
- **mu** The \( \text{Parameter} \) for which to apply the adjoint operator.
- **source_product** The inner product \( \text{Operator} \) on the source space. If \( \text{None} \), the Euclidean product is chosen.
- **range_product** The inner product \( \text{Operator} \) on the range space. If \( \text{None} \), the Euclidean product is chosen.

### Returns

\( \text{VectorArray} \) of the adjoint operator evaluations.

### apply_inverse_adjoint

\[ \text{apply_inverse_adjoint}(U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None}, \text{least_squares}=\text{False}) \]

Apply the inverse adjoint operator.

#### Parameters

- **U** \( \text{VectorArray} \) of vectors to which the inverse adjoint operator is applied.
- **ind** The indices of the vectors in \( U \) to which the inverse adjoint operator shall be applied (see the \( \text{VectorArray} \) documentation for further details).
- **mu** The \( \text{Parameter} \) for which to evaluate the inverse adjoint operator.
- **source_product** See apply_adjoint.
- **range_product** See apply_adjoint.
- **least_squares** If \( \text{True} \), solve the least squares problem:

\[ v = \arg\min ||\text{op}^*(v) - u||_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate \( \text{solver_options} \) are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

#### Returns

\( \text{VectorArray} \) of the inverse adjoint 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_lincomb** *(operators, coefficients, solver_options=None, 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.
- **solver_options**  *solver_options* for the assembled operator.
- **name**  Name of the assembled operator.

**Returns**

- The assembled Operator if assembly is possible, otherwise *None*.

---

**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.

**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.VectorFunctional(vector, product=None, name=None)**

- **Bases:** `pymor.operators.constructions.VectorArrayOperator`

Wrap a vector as a linear Functional.

Given a vector `v` of dimension `d`, this class represents the functional

\[
 f: \mathbb{R}^d \rightarrow \mathbb{R}^1 \\
 u \mapsto (u, v)
\]

where \((\cdot, \cdot)\) denotes the inner product given by `product`.

In particular, if `product` is None

\[
\text{VectorFunctional(vector).as_vector()} = \text{vector}.
\]

If `product` is not none, we obtain

\[
\text{VectorFunctional(vector).as_vector()} = \text{product.apply(vector)}.
\]

**Parameters**

- **vector** `VectorArray` of length 1 containing the vector `v`.
- **product** `Operator` representing the scalar product to use.
- **name** Name of the operator.

**Methods**
class pymor.operators.constructions.VectorOperator(vector, name=None)

Wrap a vector as a vector-like Operator.

Given a vector v of dimension d, this class represents the operator

\[
\text{op: } \mathbb{R}^1 \rightarrow \mathbb{R}^d \\
\text{x \rightarrow xv}
\]

In particular:

VectorOperator(vector).as_vector() == vector

Parameters

vector VectorArray of length 1 containing the vector v.

name Name of the operator.

Methods
class pymor.operators.constructions.ZeroOperator(source, range, name=None)
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

ZeroOperator  apply, apply_inverse, apply_inverse_adjoint, assemble_lincomb, project, restricted

OperatorBase  apply2, apply_adjoint, as_vector, assemble, jacobian, pairwise_apply2, __add__, __mul__, __radd__, __str__

ImmutableInterface  generate_sid, unlock, with_

BasicInterface  disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __getattr__

Parametric  build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

ZeroOperator  linear

OperatorInterface  range, solver_options, source

ImmutableInterface  add_with_arguments, sid, sid_ignore, with_arguments

BasicInterface  locked, logger, logging_disabled, name, uid

Parametric  parameter_local_type, parameter_space, parameter_type, parametric

apply (U, ind=None, mu=None)

Apply the operator to a VectorArray.

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_inverse (V, ind=None, mu=None, least_squares=False)
Apply the inverse operator.

Parameters
V  VectorArray of vectors to which the inverse operator is applied.
ind  The indices of the vectors in V to which the inverse operator shall be applied (see VectorArray documentation for further details).
mu  The Parameter for which to evaluate the inverse operator.
least_squares  If True, solve the least squares problem:

\[ u = \text{argmin} \| \text{op}(u) - v \|_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

Returns
VectorArray of the inverse operator evaluations.

Raises
InversionError  The operator could not be inverted.

apply_inverse_adjoint (U, ind=None, mu=None, source_product=None, range_product=None, least_squares=False)
Apply the inverse adjoint operator.

Parameters
U  VectorArray of vectors to which the inverse adjoint operator is applied.
ind  The indices of the vectors in U to which the inverse adjoint operator shall be applied (see the VectorArray documentation for further details).
mu  The Parameter for which to evaluate the inverse adjoint operator.
source_product  See apply_adjoint.
range_product  See apply_adjoint.
least_squares  If True, solve the least squares problem:
\[ v = \text{argmin} \| \text{op}^*(v) - u \|_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

Returns

`VectorArray` of the inverse adjoint operator evaluations.

Raises

`InversionError` The operator could not be inverted.

assemble_lincomb (operators, coefficients, solver_options=None, 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.
- `solver_options` solver_options for the assembled operator.
- `name` Name of the assembled operator.

Returns

The assembled `Operator` if assembly is possible, otherwise None.

projected (range_basis, source_basis, product=None, name=None)

Project the operator to subspaces of the source and range space.

Given an inner product `( , )`, source vectors \( b_1, \ldots, b_N \) and range vectors \( c_1, \ldots, c_M \), the projection \( \text{op}_\text{proj} \) of \( \text{op} \) is defined by

\[
[ \text{op}_\text{proj}(e_j) ]_i = ( c_i, \text{op}(b_j) )
\]

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 \( \text{op}_\text{proj} \) is the coordinate representation w.r.t. the \( b_i/c_i \) bases of the restriction of \( \text{op} \) to \( \text{span}(b_i) \) concatenated with the orthogonal projection onto \( \text{span}(c_i) \).
From another point of view, if \( \text{op} \) is viewed as a bilinear form (see `apply2`) and \((\ ,\ )\) is the Euclidean inner product, then \( \text{op} \_\text{proj} \) 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 \textit{NumpyMatrixOperator} will again be a \textit{NumpyMatrixOperator}, only a generic \textit{ProjectedOperator} can be returned in general.

A default implementation is provided in \textit{OperatorBase}.

### Parameters

- **range\_basis**: The vectors \( c_1, \ldots, c_M \) as a \textit{VectorArray}. If None, no projection in the range space is performed.
- **source\_basis**: The vectors \( b_1, \ldots, b_N \) as a \textit{VectorArray} or None. If None, no restriction of the source space is performed.
- **product**: An \textit{Operator} representing the inner product. If None, the Euclidean inner product is chosen.
- **name**: Name of the projected operator.

### Returns

The projected \textit{Operator} \( \text{op} \_\text{proj} \).

### restricted\((dofs)\)

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 \textit{VectorArray} \( U \) in \( \text{self.source} \) 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 \textit{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.

### Parameters

- **dofs**: One-dimensional \textit{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 \textit{NumpyVectorSpace} (\text{len}(\text{source\_dofs})) as source and \textit{NumpyVectorSpace} (\text{len}(\text{dofs})) as range.
- **source\_dofs**: One-dimensional \textit{NumPy array} of source degrees of freedom as defined above.
pymor.operators.constructions.

### induced_norm

#### Obtain induced norm of an inner product.

The norm of the vectors in a `VectorArray` `U` is calculated by calling

```python
product.pairwise_apply2(U, U, mu=mu).
```

In addition, negative norm squares of absolute value smaller than `tol` are clipped to `0`. If `raise_negative` is `True`, a `ValueError` exception is raised if there are negative norm squares of absolute value larger than `tol`.

#### Parameters

- **product**: The inner product `Operator` for which the norm is to be calculated.
- **raise_negative**: If `True`, raise an exception if calculated norm is negative.
- **tol**: See above.

#### Returns

- **norm**: A function `norm(U, mu=None)` taking a `VectorArray` `U` as input together with the `Parameter` `mu` which is passed to the product.

#### Defaults

- `raise_negative`, `tol` (see `pymor.core.defaults`)

#### ei module

class pymor.operators.ei.

### EmpiricalInterpolatedOperator

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 let `b_1, ..., b_M` in `R^(L.range.dim)` be collateral basis vectors. If moreover `ψ_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

| i | L_EI(U, μ) = b_i λ_i such that |
|   | i=1 |
|   | ψ_j(c_i)(L_EI(U, μ)) = ψ_j(c_i)(L(U, μ)) for i=0,...,M |

Since the original operator only has to be evaluated at the given interpolation DOFs, `EmpiricalInterpolatedOperator` calls `restricted` to obtain a restricted version of the operator which is 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.
- **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 $\psi_{(c_i)(b_j)} = 0$ for $i < j$, which means that the interpolation matrix is triangular.
- **name** Name of the operator.

**Methods**

- `apply`, `apply2`, `apply_inverse`, `apply_inverse_adjoint`, `as_vector`, `assemble`, `pairwise_apply2`, `__add__`, `__mul__`, `__radd__`, `__str__`
- `assemble_lincomb`, `restricted`
- `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`
- `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

**Attributes**

- `linear`, `range`, `solver_options`, `source`
- `add_with_arguments`, `sid`, `sid_ignore`, `with_arguments`
- `locked`, `logger`, `logging_disabled`, `name`, `uid`
- `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

**apply** ($U$, ind=None, mu=None)

Apply the operator to a `VectorArray`.

**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 as a new *Operator*.

**Parameters**

- **U** Length 1 *VectorArray* containing the vector for which to compute the Jacobian.
- **mu** The *Parameter* for which to compute the Jacobian.

**Returns**

Linear *Operator* representing the Jacobian.

---

**projected** *(range_basis, source_basis, product=None, name=None)*

Project the operator to subspaces of the source and range space.

Given an inner product *( , )* , source vectors $b_1, \ldots, b_N$ and range vectors $c_1, \ldots, c_M$ , the projection $\text{op}_\text{proj}$ of $\text{op}$ is defined by

$$[ \text{op}_\text{proj}(e_j) ]_i = ( c_i, \text{op}(b_j) )$$

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 $\text{op}_\text{proj}$ is the coordinate representation w.r.t. the $b_i/c_i$ bases of the restriction of $\text{op}$ to $\text{span}(b_i)$ concatenated with the orthogonal projection onto $\text{span}(c_i)$.

From another point of view, if $\text{op}$ is viewed as a bilinear form (see *apply2*) and *( , )* is the Euclidean inner product, then $\text{op}_\text{proj}$ 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 *ProjectedOperator* can be returned in general.

A default implementation is provided in *OperatorBase*.

**Parameters**

- **range_basis** The vectors $c_1, \ldots, c_M$ as a *VectorArray*. If None, no projection in the range space is performed.
- **source_basis** The vectors $b_1, \ldots, b_N$ as a *VectorArray* or None. If None, no restriction of the source space is performed.
- **product** An *Operator* representing the inner product. If None, the Euclidean inner product is chosen.
- **name** Name of the projected operator.

**Returns**

The projected *Operator* $\text{op}_\text{proj}$.
class pymor.operators.eiProjectedEmpiricalInterpolatedOperator(restricted_operator, interpolation_matrix, source_basis_dofs, projected_collateral_basis, triangular, solver_options=None, name=None)

Bases: pymor.operators.basic.OperatorBase

A projected EmpiricalInterpolatedOperator.

Not intended to be used directly. Instead use projected.

Methods

<table>
<thead>
<tr>
<th>ProjectedEmpiricalInterpolatedOperator</th>
<th>apply, projected_to_subbasis, apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, pairwise_apply2, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>linear, range, solver_options, source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</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 to a VectorArray.

**Parameters**

<table>
<thead>
<tr>
<th>U VectorArray of vectors to which the operator is applied.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ind The indices of the vectors in U to which the operator shall be applied (see the VectorArray documentation for further details).</td>
</tr>
<tr>
<td>mu The Parameter for which to evaluate the operator.</td>
</tr>
</tbody>
</table>

**Returns**

VectorArray of the operator evaluations.
jacobian (U, mu=None)
Return the operator's Jacobian as a new Operator.

Parameters
U  Length 1 VectorArray containing the vector for which to compute the Jacobian.
mu  The Parameter for which to compute the Jacobian.

Returns
Linear Operator representing the Jacobian.

fenics module

fv module

This module provides some operators for finite volume discretizations.

class pymor.operators.fv.DiffusionOperator (grid, boundary_info, diffusion_function=None, diffusion_constant=None, solver_options=None, name=None)
Bases: pymor.operators.numpy.NumpyMatrixBasedOperator
Finite Volume Diffusion Operator.
The operator is of the form

\((Lu)(x) = c \begin{bmatrix} d(x) & u(x) \end{bmatrix}\)

Parameters
grid  The Grid over which to assemble the operator.
boundary_info  BoundaryInfo for the treatment of Dirichlet boundary conditions.
diffusion_function  The scalar-valued Function \(d(x)\). If None, constant one is assumed.
diffusion_constant  The constant \(c\). If None, \(c\) is set to one.
name  Name of the operator.

Methods
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:

    \[
    F(U_{\text{in}}, U_{\text{out}}, \text{normal}, \text{vol}) = \text{vol} \times \left( c^+(U_{\text{in}}, \text{normal}) + c^-(U_{\text{out}}, \text{normal}) \right)
    \]

    \[
    c^+(U_{\text{in}}, \text{normal}) = f(0) \text{normal} + \max(f'(s) \text{normal}, 0) \, ds \quad \text{s=0}
    \]

    \[
    c^-(U_{\text{out}}, \text{normal}) = \min(f'(s) \text{normal}, 0) \, ds \quad \text{s=0}
    \]

**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**
class `pymor.operators.fv.L2Product` (grid, solver_options=None, name=None)

**Bases:** `pymor.operators.numpy.NumpyMatrixBasedOperator`  

**Operator** representing the L2-product between finite volume functions.

**Parameters**

- `grid`  The `Grid` for which to assemble the product.
- `name`  The name of the product.

**Methods**

- `apply`, `apply_adjoint`, `apply_inverse`, `as_vector`, `assemble`, `export_matrix`
- `apply2`, `apply_inverse_adjoint`, `jacobian`, `pairwise_apply2`, `projected`, `__add__`, `__mul__`, `__radd__`, `__radd__`, `__str__`
- `assemble_lincomb`, `restricted`
- `generate_sid`, `unlock`, `with_`
- `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`
- `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

**Attributes**

- `sparse`  True
- `linear`  True
- `range`, `solver_options`, `source`
- `add_with_arguments`, `sid`, `sid_ignore`, `with_arguments`
- `locked`, `logger`, `logging_disabled`, `name`, `uid`
- `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 inner 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 inner 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

NumpyMatrixBasedOperator: apply, apply_adjoint, apply_inverse, as_vector, assemble, export_matrix
OperatorBase: apply2, apply_inverse_adjoint, jacobian, pairwise_apply2, projected, __add__, __mul__, __radd__, __str__
OperatorInterface: assemble_lincomb, restricted
ImmutableInterface: generate_sid, unlock, with_
BasicInterface: disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__
Parametric: build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

L2ProductFunctional: range, sparse
NumpyMatrixBasedOperator: linear
OperatorInterface: solver_options, source
ImmutableInterface: add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface: locked, logger, logging_disabled, name, uid
Parametric: parameter_local_type, parameter_space, parameter_type, parametric
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:

\[
F(U_{in}, U_{out}, \text{normal}, \text{vol}) = \text{vol} \times \left[ \text{normal}(f(U_{in}) + f(U_{out}))/2 + (U_{in} - U_{\rightarrow out})/(2*\lambda) \right]
\]

Parameters

**flux**  *Function* defining the analytical flux \( f \).

**lxf_lambda**  The stabilization parameter \( \lambda \).

Methods

LaxFriedrichsFlux.evaluate_stage1, evaluate_stage2
ImmutableInterface.generate_sid, unlock, with_
BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__
Parametric.build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

ImmutableInterface.add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface.locked, logger, logging_disabled, name, uid
Parametric.parameter_local_type, parameter_space, parameter_type, parametric

class pymor.operators.fv.LinearAdvectionLaxFriedrichs(grid, boundary_info, velocity_field, lxf_lambda=1.0, solver_options=None, 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**

<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, apply_inverse_adjoint, jacobian, pairwise_apply2, 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>generate_sid, unlock, with__</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>NumpyMatrixBasedOperator</th>
<th>linear, sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parameter_type, parametric</td>
</tr>
</tbody>
</table>

---

**class** pymor.operators.fv.NonlinearAdvectionOperator(grid, boundary_info, numerical_flux, dirichlet_data=None, solver_options=None, name=None)

Bases: pymor.operators.basic.OperatorBase

Nonlinear finite volume advection Operator.

The operator is of the form

\[ L(u, \mu)(x) = f(u(x), \mu) \]

---

**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

<table>
<thead>
<tr>
<th>NonlinearAdvectionOperator</th>
<th>apply, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, pairwise_apply2, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, pairwise_apply2, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>assemble_lincomb, generate_sid, unlock, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, with_arguments, locked_with_arguments, sid, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>NonlinearAdvectionOperator</th>
<th>linear, sid_ignore</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</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 to a `VectorArray`.

**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 as a new `Operator`.

**Parameters**

*U* Length 1 `VectorArray` containing the vector for which to compute the Jacobian.

*mu* The `Parameter` for which to compute the Jacobian.

**Returns**
Linear Operator representing the Jacobian.

**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.

**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.

**with_(**kwargs)**

Returns a copy with changed attributes.

The default implementation is to create a new class instance with the given keyword arguments as arguments for **init**. Missing arguments are obtained form instance attributes with the same name.

**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.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 \( F(U_{\text{inner}}, U_{\text{outer}}, \text{unit}_{\text{outer}}_{\text{normal}}, \text{edge}_\text{volume}, \mu) \).

The flux evaluation is vectorized and happens in two stages:
1. `evaluate_stage1` receives a NumPy array `U` of all values which appear as `U_inner` or `U_outer` for all edges the flux shall be evaluated at and returns a tuple of NumPy arrays each of the same length as `U`.

2. `evaluate_stage2` 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_stage1`, the `l`-th entry `D_l` of the `stage1_data` tuple has the shape `(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_stage1`, we have
   
   \[ D_l[k, 0] = R_l[i], \quad D_l[k, 1] = R_l[j]. \]

`evaluate_stage2` returns a NumPy array of the flux evaluations for each edge.

### Methods

<table>
<thead>
<tr>
<th>Base Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>generate_sid</code>, <code>unlock</code>, <code>with_</code></td>
</tr>
<tr>
<td><code>BasicInterface</code></td>
<td><code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>lock</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>Base Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ImmutableInterface</code></td>
<td><code>add_with_arguments</code>, <code>sid</code>, <code>sid_ignore</code>, <code>with_arguments</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></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>

### Class

```
class pymor.operators.fv.SimplifiedEngquistOsherFlux(flux, flux_derivative):
    # Engquist-Osher numerical flux. Simplified Implementation for special case.
    # For the definition of the Engquist-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 \( f' \).
```

### Methods
**Instantiate a** *NonlinearAdvectionOperator* using *EngquistOsherFlux*.

**Instantiate a** *NonlinearAdvectionOperator* using *LaxFriedrichsFlux*.

**Instantiate a** *NonlinearAdvectionOperator* using *SimplifiedEngquistOsherFlux*. 

---

**Attributes**

<table>
<thead>
<tr>
<th>ImmutableInterface</th>
<th>add_with_arguments, sid, sid_ignore, with_arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>
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 as its range VectorSpace.

Methods

<table>
<thead>
<tr>
<th>OperatorInterface</th>
<th>apply, apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, assemble_lincomb, jacobian, pairwise_apply2, projected, restricted, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>build_parameter_type, local_parameter, parse_parameter, projected, restricted, unlock, with <strong>setattr</strong></td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, projected, restricted, unlock, with <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>OperatorInterface</th>
<th>linear, range, solver_options, source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

solver_options

If not None, a dict which can contain the following keys:

- **inverse** solver options used for apply_inverse
- **inverse_adjoint** solver options used for apply_inverse_adjoint
- **jacobian** solver options for the operators returned by jacobian (has no effect for linear operators)

If solver_options is None or a dict entry is missing or None, default options are used. The interpretation of the given solver options is up to the operator at hand. In general, values in solver_options should either be strings (indicating a solver type) or dicts of options, usually with an entry 'type' which specifies the solver type to use and further items which configure this solver.

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 to a VectorArray.

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, U_ind=None, V_ind=None, mu=None, product=None)
Treat the operator as a 2-form by calculating (V, op(U)).
In general op.apply2(V, U) is equivalent to:
product.apply2(V, op.apply(U)).
In case no product has been specified, op.apply2(V, U) is equivalent to:
V.dot(op.apply(U)).
In the latter case, assuming that op is a linear operator given by multiplication with a matrix M, then:

\[
\text{op.apply2}(V, U) = V^T \cdot M \cdot U.
\]

Parameters

V VectorArray of the left arguments V.
U VectorArray of the right arguments U.
V_ind The indices of the vectors in V to which the operator shall be applied (see VectorArray documentation for further details).
U_ind The indices of the vectors in U to which the operator shall be applied (see VectorArray documentation for further details).
mu The Parameter for which to evaluate the operator.
product The inner product used in the expression \((V, \text{op}(U))\) given as an Operator. If None, the euclidean product is chosen.
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 \(op\) the adjoint \(op^*\) of \(op\) is given by:

\[
(op^*(v), u)_s = (v, op(u))_r,
\]

where \((\ , \ , \ )_s\) and \((\ , \ , \ )_r\) denote the inner products on the source and range space of \(op\). If \(op\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[
op^*(v) = P_s^{-1} \cdot M^T \cdot P_r \cdot v,
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \((\ , \ , \ )_s\) and \((\ , \ , \ )_r\) are the Euclidean inner products, \(op^*v\) is simply given by multiplication of the matrix of \(op\) 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 inner product Operator on the source space. If \(\text{None}\), the Euclidean product is chosen.

\(\text{range\_product}\) The inner product Operator on the range space. If \(\text{None}\), the Euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.

apply_inverse \((V, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{least\_squares}=\text{False})\)

Apply the inverse operator.

Parameters

\(V\) VectorArray of vectors to which the inverse operator is applied.

\(\text{ind}\) The indices of the vectors in \(V\) to which the inverse operator shall be applied (see VectorArray documentation for further details).

\(\text{mu}\) The Parameter for which to evaluate the inverse operator.

\(\text{least\_squares}\) If True, solve the least squares problem:

\[
u = \text{argmin} \ ||op(u) - v||_2.\]
Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

Returns
VectorArray of the inverse operator evaluations.

Raises
InversionError The operator could not be inverted.

apply_inverse_adjoint \((U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None}, \text{least_squares}=\text{False})\)
Apply the inverse adjoint operator.

Parameters

\(U\) VectorArray of vectors to which the inverse adjoint operator is applied.

\(\text{ind}\) The indices of the vectors in \(U\) to which the inverse adjoint operator shall be applied (see the VectorArray documentation for further details).

\(\text{mu}\) The Parameter for which to evaluate the inverse adjoint operator.

\(\text{source_product}\) See apply_adjoint.

\(\text{range_product}\) See apply_adjoint.

\(\text{least_squares}\) If True, solve the least squares problem:

\[
\mathbf{v} = \text{argmin} \| \mathbf{op}^*(\mathbf{v}) - \mathbf{u} \|_2.
\]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

Returns
VectorArray of the inverse adjoint operator evaluations.

Raises
InversionError The operator could not be inverted.

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 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

\[
\text{self.as_vector}(\mu) = \text{self.apply}(\text{NumpyVectorArray}(1), \mu).
\]

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 (mu=None)
Assemble the operator for a given parameter.

The result of the method 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, solver_options=None, 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.
solver_options solver_options for the assembled operator.
namer Name of the assembled operator.
Returns
The assembled Operator if assembly is possible, otherwise None.

jacobian \((U, \mu=None)\)
Return the operator’s Jacobian as a new Operator.

Parameters
U Length 1 `VectorArray` containing the vector for which to compute the Jacobian.
\(\mu\) The `Parameter` for which to compute the Jacobian.

Returns
Linear `Operator` representing the Jacobian.

pairwise_apply2 \((V, U, U\_ind=None, V\_ind=None, \mu=None, \text{product=None})\)
Treat the operator as a 2-form by calculating \((V, \text{op}(U))\).
Same as `OperatorInterface.apply2`, except that vectors from \(V\) and \(U\) are applied in pairs.

Parameters
V `VectorArray` of the left arguments \(V\).
U `VectorArray` of the right arguments \(U\).
V\_ind The indices of the vectors in \(V\) to which the operator shall be applied (see `VectorArray` documentation for further details).
U\_ind The indices of the vectors in \(U\) to which the operator shall be applied (see `VectorArray` documentation for further details).
\(\mu\) The `Parameter` for which to evaluate the operator.
product The inner product used in the expression \((V, \text{op}(U))\) given as an `Operator`. If None, the euclidean product is chosen.

Returns
A NumPy array with shape `(len(V\_ind),) == (len(U\_ind),)` containing the 2-form evaluations.

projected \((\text{range\_basis}, \text{source\_basis}, \text{product=None}, \text{name=None})\)
Project the operator to subspaces of the source and range space.

Given an inner product \((\ , \ )\), source vectors \(b_1, \ldots, b_N\) and range vectors \(c_1, \ldots, c_M\), the projection \(\text{op\_proj}\) of \text{op} is defined by

\[
[\text{op\_proj}(e_j)]_i = (c_i, \text{op}(b_j))
\]
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 \(\text{op\_proj}\) is the coordinate representation w.r.t. the \(b_i/c_i\) bases of the restriction of \(\text{op}\) to \(\text{span}(b_i)\) concatenated with the orthogonal projection onto \(\text{span}(c_i)\).

From another point of view, if \(\text{op}\) is viewed as a bilinear form (see \(\text{apply2}\)) and \((\quad,\quad)\) is the Euclidean inner product, then \(\text{op\_proj}\) 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 \(\text{NumpyMatrixOperator}\) will again be a \(\text{NumpyMatrixOperator}\), only a generic \(\text{ProjectedOperator}\) can be returned in general.

A default implementation is provided in \(\text{OperatorBase}\).

**Parameters**

- **range_basis** The vectors \(c_1, \ldots, c_M\) as a VectorArray. If None, no projection in the range space is performed.
- **source_basis** The vectors \(b_1, \ldots, b_N\) as a VectorArray or None. If None, no restriction of the source space is performed.
- **product** An Operator representing the inner product. If None, the Euclidean inner product is chosen.
- **name** Name of the projected operator.

**Returns**

The projected Operator \(\text{op\_proj}\).

**restricted(** \(dofs\)**)**

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 \(self.source\) 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 \(\text{source\_dofs}\) will be needed to evaluate the restricted operator which can make its evaluation very fast compared to evaluating the original 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 \(\text{NumpyVectorSpace}(\text{len}(\text{source\_dofs}))\) as \(\text{source}\) and \(\text{NumpyVectorSpace}(\text{len}(\text{dofs}))\) as \(\text{range}\).
- **source\_dofs** One-dimensional NumPy array of source degrees of freedom as defined above.
### mpi module

**class** pymor.operators.mpi.MPIOperator

```python
def __init__(self, obj_id, functional=False, vector=False, with_apply2=False, pickle_subtypes=True, array_type=<class 'pymor.vectorarrays.mpi.MPIVectorArray'>)
```

**Bases:** pymor.operators.basic.OperatorBase

**MPI distributed Operator.**

Given a single-rank implementation of an Operator, this wrapper class uses the event loop from pymor.tools.mpi to allow an MPI distributed usage of the Operator.

Instances of MPIOperator can be used on rank 0 like any other (non-distributed) Operator.

Note, however, that the underlying Operator implementation needs to be MPI aware. For instance, the operator’s apply method has to perform the necessary MPI communication to obtain all DOFs hosted on other MPI ranks which are required for the local operator evaluation.

Instead of instantiating MPIOperator directly, it is usually preferable to use mpi_wrap_operator instead.

**Parameters**

- **obj_id** ObjectId of the local Operators on each rank.
- **functional** Set to True if the operator represents a Functional. As required for functionals in pyMOR, this will set the range of the operator to NumpyVectorSpace(1) instead of MPIVectorArray-based space.
- **vector** Set to True if the operator represents a (parametric) vector. As required for vector-like Operators in pyMOR, this will set the source of the operator to NumpyVectorSpace(1) instead of an MPIVectorArray-based space.
- **with_apply2** Set to True if the operator implementation has its own MPI aware implementation of apply2 and pairwise_apply2. Otherwise, the default implementations using apply and dot will be used.
- **pickle_subtypes** If pickle_subtypes is False, a unique identifier is computed for each local source/range subtype, which is then transferred to rank 0 instead of the true subtype. This allows the usage of MPIVectorArray even when the local subtypes are not picklable.
- **array_type** This class will be used to wrap the local VectorArrays returned by the local operators into an MPI distributed VectorArray managed from rank 0. By default, MPIVectorArray will be used, other options are MPIVectorArrayAutoComm and MPIVectorArrayNoComm.

**Methods**
apply \((U, \text{ind}=\text{None}, \text{mu}=\text{None})\)

Apply the operator to a VectorArray.

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 \((V, U, U_{\text{ind}}=\text{None}, V_{\text{ind}}=\text{None}, \text{mu}=\text{None}, \text{product}=\text{None})\)

Treat the operator as a 2-form by calculating \((V, \text{op}(U))\).

In general \(\text{op}.\text{apply2}(V, U)\) is equivalent to:

\[
\text{product}.\text{apply2}(V, \text{op}.\text{apply}(U)).
\]

In case no product has been specified, \(\text{op}.\text{apply2}(V, U)\) is equivalent to:

\[
V.\text{dot}(\text{op}.\text{apply}(U)).
\]

In the latter case, assuming that \(\text{op}\) is a linear operator given by multiplication with a matrix \(M\), then:

\[
\text{op}.\text{apply2}(V, U) = V^T M U.
\]

Parameters

- **V** VectorArray of the left arguments V.
- **U** VectorArray of the right arguments U.
V_ind  The indices of the vectors in V to which the operator shall be applied (see VectorArray documentation for further details).

U_ind  The indices of the vectors in U to which the operator shall be applied (see VectorArray documentation for further details).

mu  The Parameter for which to evaluate the operator.

product  The inner product used in the expression \((V, op(U))\) given as an Operator. If None, the euclidean product is chosen.

**Returns**

A NumPy array with shape \((\text{len}(V\_ind), \text{len}(U\_ind))\) containing the 2-form evaluations.

**apply_adjoint** \((U, \text{ind=None, mu=None, source\_product=None, range\_product=None})\)

Apply the adjoint operator.

For a linear operator \(op\) the adjoint \(op^*\) of \(op\) is given by:

\[
(op^*(v), u)_s = (v, op(u))_r,
\]

where \((v, u)_s\) and \((v, u)_r\) denote the inner products on the source and range space of \(op\). If \(op\) and the two products are given by the matrices \(M, P_s, P_r\), then:

\[
op^*(v) = P_s^(-1) * M^T * P_r * v,
\]

with \(M^T\) denoting the transposed of \(M\). Thus, if \((v, u)_s\) and \((v, u)_r\) are the Euclidean inner products, \(op^*v\) is simply given by multiplication of the matrix of \(op\) 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 inner product Operator on the source space. If None, the Euclidean product is chosen.
- **range_product** The inner product Operator on the range space. If None, the Euclidean product is chosen.

**Returns**

VectorArray of the adjoint operator evaluations.

**apply_inverse** \((V, \text{ind=None, mu=None, least\_squares=False})\)

Apply the inverse operator.

**Parameters**

- **V** VectorArray of vectors to which the inverse operator is applied.
ind  The indices of the vectors in V to which the inverse operator shall be applied (see VectorArray documentation for further details).

mu  The Parameter for which to evaluate the inverse operator.

least_squares  If True, solve the least squares problem:

$$
    u = \text{argmin} \; ||\text{op}(u) - v||_2.
$$

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

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

$$
    \text{self.as_vector(mu)}.\text{dot}(U) = \text{self.apply}(U, \mu)
$$

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 (mu=None)

Assemble the operator for a given parameter.

The result of the method 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, solver_options=None, 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.
- **solver_options**  `solver_options` for the assembled operator.
- **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 as a new `Operator`.

Parameters

- **U**  Length 1 `VectorArray` containing the vector for which to compute the Jacobian.
- **mu**  The `Parameter` for which to compute the Jacobian.

Returns

Linear `Operator` representing the Jacobian.

**pairwise_apply2**(V, U, U_ind=None, V_ind=None, mu=None, product=None)

Treat the operator as a 2-form by calculating `(V, op(U))`.

Same as `OperatorInterface.apply2`, except that vectors from `V` and `U` are applied in pairs.

Parameters

- **V**  `VectorArray` of the left arguments `V`. 
A NumPy array with shape (len(V_ind),) == (len(U_ind),) containing the 2-form 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:

```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.

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.

pymor.operators.mpi.mpi_wrap_operator(obj_id, functional=False, vector=False, with_apply2=False, pickle_subtypes=True, array_type=<class 'pymor.vectorarrays.mpi.MPIVectorArray'>)

Wrap MPI distributed local Operators to a global Operator on rank 0.

Given MPI distributed local Operators referred to by the ObjectId obj_id, return a new Operator which manages these distributed operators from rank 0. This is done by instantiating MPIOperator. Additionally, the structure of the wrapped operators is preserved. E.g. LincombOperators will be wrapped as a LincombOperator of MPIOperators.
**Parameters**

See :class:`MPIOperator`.

**Returns**

The wrapped :class:`Operator`.

---

**numpy module**

This module provides the following NumPy based Operators:

- :class:`NumpyMatrixOperator` wraps a 2D NumPy array as an :class:`Operator`.
- :class:`NumpyMatrixBasedOperator` should be used as base class for all :class:`Operator` which assemble into a :class:`NumpyMatrixOperator`.
- :class:`NumpyGenericOperator` wraps an arbitrary Python function between NumPy arrays as an :class:`Operator`.

---

**class** `pymor.operators.numpy.NumpyGenericOperator(mapping, dim_source=1, dim_range=1, linear=False, parameter_type=None, solver_options=None, name=None)`

Wraps an arbitrary Python function between NumPy arrays as a :class:`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:

  ```python
  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 :class:`ParameterType` of the :class:`Parameters` the mapping accepts.
- **name** Name of the operator.

---

**Methods**
apply \( U, \text{ind}=\text{None}, \mu=\text{None} \)

Apply the operator to a VectorArray.

**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.
NumpyMatrixBasedOperator

- `apply`, `apply_adjoint`, `apply_inverse`, `as_vector`, `assemble`, `export_matrix`
- `__add__`, `__mul__`, `__radd__`, `__str__`
- `apply2`, `apply_inverse_adjoint`, `jacobian`, `pairwise_apply2`, `projected`, `restricted`, `__add__`, `__mul__`, `__str__`
- `assemble`, `assemble_lincomb`, `generate_sid`, `unlock`, `with_`
- `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`
- `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

### Attributes

<table>
<thead>
<tr>
<th>NumpyMatrixBasedOperator</th>
<th>linear, sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</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 to a VectorArray.

**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 op the adjoint op^* of op is given by:

\[(op^*(v), u)_s = (v, op(u))_r,\]

where ( , )_s and ( , )_r denote the inner products on the source and range space of op. If op and the two products are given by the matrices M, P_s and P_r, then:

\[op^*(v) = P_s^{-1} \times M^T \times P_r \times v,\]

with M^T denoting the transposed of M. Thus, if ( , )_s and ( , )_r are the Euclidean inner products, op^*v is simply given by multiplication of the matrix of op 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 inner product Operator on the source space. If None, the Euclidean product is chosen.

\( \text{range\_product} \) The inner product Operator on the range space. If None, the Euclidean product is chosen.

Returns

VectorArray of the adjoint operator evaluations.

\text{apply\_inverse}(V, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{least\_squares}=\text{False})

Apply the inverse operator.

Parameters

\( V \) VectorArray of vectors to which the inverse operator is applied.

\( \text{ind} \) The indices of the vectors in \( V \) to which the inverse operator shall be applied (see VectorArray documentation for further details).

\( \text{mu} \) The Parameter for which to evaluate the inverse operator.

\( \text{least\_squares} \) If True, solve the least squares problem:

\[
u = \arg \min ||\text{op}(u) - v||_2.
\]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

Returns

VectorArray of the inverse operator evaluations.

Raises

InversionError The operator could not be inverted.

\text{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 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

```python
self.as_vector(mu).dot(U) == self.apply(U, mu)
```

holds, whereas in the case of a vector-like operator we have

```python
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:

assemble (mu=None)

Assembles the operator for a given *Parameter*.

Parameters

**mu**  The *Parameter* for which to assemble the operator.

Returns

The assembled parameter independent *Operator*.

.. _export_matrix:

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:

class pymor.operators.numpy.NumpyMatrixOperator (matrix, solver_options=None, 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

<table>
<thead>
<tr>
<th>NumpyMatrixOperator</th>
<th>apply, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, assemble_lincomb, from_file, projected_to_subbasis</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>apply2, jacobian, pairwise_apply2, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorBase</td>
<td>apply2, jacobian, pairwise_apply2, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>restricted</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with__</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <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>NumpyMatrixBasedOperator</th>
<th>linear, sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sidIgnore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
<tr>
<td>Parametric</td>
<td>parameter_local_type, parameter_space, parameter_type, parametric</td>
</tr>
</tbody>
</table>

**apply** \((U, \text{ind}=\text{None}, \text{mu}=\text{None})\)

Apply the operator to a VectorArray.

**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** \((U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None})\)

Apply the adjoint operator.

For a linear operator \(\text{op}\) the adjoint \(\text{op}^*\) of \(\text{op}\) is given by:

\[
(\text{op}^*(v), u)_s = (v, \text{op}(u))_r,
\]

where \(( , )_s\) and \(( , )_r\) denote the inner products on the source and range space of \(\text{op}\). If \(\text{op}\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:
\[ \text{op}^*(v) = P_s^{-1} \cdot M^T \cdot P_r \cdot v, \]

with \( M^T \) denoting the transposed of \( M \). Thus, if \((\cdot, \cdot)_s\) and \((\cdot, \cdot)_r\) are the Euclidean inner products, \( \text{op}^*v \) is simply given by multiplication of the matrix of \( \text{op} \) 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 inner product Operator on the source space. If None, the Euclidean product is chosen.
- **range_product** The inner product Operator on the range space. If None, the Euclidean product is chosen.

**Returns**

VectorArray of the adjoint operator evaluations.

**apply_inverse** \((V, \text{ind=None, mu=None, least_squares=False})\)

Apply the inverse operator.

**Parameters**

- **V** VectorArray of vectors to which the inverse operator is applied.
- **ind** The indices of the vectors in \( V \) to which the inverse operator shall be applied (see VectorArray documentation for further details).
- **mu** The Parameter for which to evaluate the inverse operator.
- **least_squares** If True, solve the least squares problem:

\[ u = \arg\min ||\text{op}(u) - v||_2. \]

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate solver_options are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

**Returns**

VectorArray of the inverse operator evaluations.

**Raises**

InversionError The operator could not be inverted.
apply_inverse_adjoint \((U, \text{ind}=\text{None}, \text{mu}=\text{None}, \text{source_product}=\text{None}, \text{range_product}=\text{None}, \text{least_squares}=\text{False})\)

Apply the inverse adjoint operator.

**Parameters**

- **U** `VectorArray` of vectors to which the inverse adjoint operator is applied.
- **ind** The indices of the vectors in U to which the inverse adjoint operator shall be applied (see the `VectorArray` documentation for further details).
- **mu** The `Parameter` for which to evaluate the inverse adjoint operator.
- **source_product** See `apply_adjoint`.
- **range_product** See `apply_adjoint`.
- **least_squares** If True, solve the least squares problem:

  \[
  v = \arg\min ||\text{op}^*(v) - u||_2. 
  \]

  Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate `solver_options` are set for the operator, most operator implementations will choose a least squares solver by default which may be undesirable.

**Returns**

- `VectorArray` of the inverse adjoint operator evaluations.

**Raises**

- `InversionError` The operator could not be inverted.

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 `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

\[
\text{self.as_vector(mu) dot (U)} == \text{self.apply(U, mu)}
\]

holds, whereas in the case of a vector-like operator we have

\[
\text{self.as_vector(mu)} == \text{self.apply(NumpyVectorArray(1), mu)}.
\]
**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** (:math:`\mu=None`)
Assembles the operator for a given Parameter.

**Parameters**

- **mu** The Parameter for which to assemble the operator.

**Returns**
The assembled parameter independent Operator.

**assemble_lincomb** (:math:`\text{operators, coefficients, solver_options=None, 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.
- **solver_options** solver_options for the assembled operator.
- **name** Name of the assembled operator.

**Returns**
The assembled Operator if assembly is possible, otherwise None.

**projected_to_subbasis** (:math:`\text{dim_range=None, dim_source=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.

\[
\text{op.projected}(r\_basis, s\_basis, \text{prod}).\text{projected_to_subbasis}(\text{dim\_range}, \text{dim\_source})
\]

should be the same as

\[
\text{op.projected}(r\_basis.copy(range(\text{dim\_range})), s\_basis.copy(range(\text{dim\_source})), \text{prod})
\]

For a NumpyMatrixOperator this amounts to extracting the upper-left (dim_range, dim_source) corner of the matrix it wraps.
Parameters

**dim_range**  Dimension of the range subbasis.

**dim_source**  Dimension of the source subbasis.

Returns

The projected Operator.

---

```python
pymor.operators.numpy._apply_inverse(matrix, V, options=None)
```

Solve linear equation system.

Applies the inverse of `matrix` to the row vectors in `V`.

See `dense_options` for documentation of all possible options for sparse matrices.

See `sparse_options` for documentation of all possible options for sparse matrices.

This method is called by `NumpyMatrixOperator.apply_inverse` and usually should not be used directly.

Parameters

- **matrix**  The left-hand side of the linear equation systems to solve as a NumPy matrix.
- **V** 2-dimensional NumPy array containing as row vectors the right-hand sides of the linear equation systems to solve.
- **options**  The solver options to use. (See `_options`.)

Returns

NumPy array of the solution vectors.

---

```python
pymor.operators.numpy._options(matrix=None, sparse=None)
```

Returns possible `solver_options` (with default values) for a given NumPy matrix.

See `dense_options` for documentation of all possible options for dense matrices.

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

---

5.1. pymor package
A tuple of all possible `solver_options`.

```python
pymor.operators.numpy.dense_options(
    default_solver='solve',
    default_least_squares_solver='least_squares_lstsq',
    least_squares_lstsq_rcond=-1.0
)
```

Returns possible `solver_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 possible values for `solver_options`.

**Defaults**

- `default_solver`, `default_least_squares_solver`, `least_squares_lstsq_rcond` (see `pymor.core.defaults`)

```python
pymor.operators.numpy.matrix_astype_nocopy(matrix, dtype)
```
pymor.operators.numpy.sparse_options (default_solver='spsolve',
  default_least_squares_solver='least_squares_lsmr',
  bicgstab_tol=1e-15, bicgstab_maxiter=None,
  spilu_drop_tol=0.0001, spilu_fill_factor=10,
  spilu_drop_rule='basic,area',
  spilu_permc_spec='COLAMD',
  spsolve_permc_spec='COLAMD',
  spsolve_keep_factorization=True, lgmres_tol=1e-05,
  lgmres_maxiter=1000, lgmres_inner_m=39, lgmres_outer_k=3,
  least_squares_lsmr_atol=1e-06,
  least_squares_lsmr_btol=1e-06, least_squares_lsmr_conlim=10000000.0,
  least_squares_lsmr_maxiter=None,
  least_squares_lsmr_show=False, least_squares_lsmr_maxiter=None,
  least_squares_lsmr_tol=1e-05, pyamg_tol=1e-05,
  pyamg_maxiter=400, pyamg_verb=False,
  pyamg_rs_strength=('classical', {'theta': 0.25}),
  pyamg_rs_CF='RS', pyamg_rs_presmoother=('gauss_seidel',
  {'sweep': 'symmetric'}),
  pyamg_rs_postsmoother=('gauss_seidel', {'sweep': 'symmetric'}),
  pyamg_rs_max_levels=10, pyamg_rs_max_coarse=500,
  pyamg_rs_cycle='V', pyamg_rs_accel=None,
  pyamg_rs_tol=1e-05, pyamg_rs_maxiter=100,
  pyamg_sa_symmetry='hermitian',
  pyamg_sa_strength='symmetric',
  pyamg_sa_aggregate='standard',
  pyamg_sa_smooth=('jacobi', {'omega': 1.3333333333333333}),
  pyamg_sa_presmoother=('block_gauss_seidel',
  {'sweep': 'symmetric'}),
  pyamg_sa_postsmoother=('block_gauss_seidel',
  {'sweep': 'symmetric'}),
  pyamg_sa_improve_candidates=[('block_gauss_seidel',
  {'sweep': 'symmetric', 'iterations': 4})],
  pyamg_sa_max_levels=10, pyamg_sa_max_coarse=500,
  pyamg_sa_diagonal_dominance=False,
  pyamg_sa_coarse_solver='pinv2',
  pyamg_sa_cycle='V', pyamg_sa_accel=None,
  pyamg_sa_tol=1e-05, pyamg_sa_maxiter=100)

Returns possible solver_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,
default_least_squares_solver Default solver to use for least squares problems (least_squares_lsmr,
least_squares_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.
pyamgverb 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_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_max_coarse 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 solver_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_lsq_chol, least_squares_lsq_roCond, least_squares_lsq_conlim, least_squares_lsq_iter_lim, least_squares_lsq_rCond, pyamg_tol, pyamg_maxiter, pyamg_vertex, 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_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)
This module contains a base class for implementing WorkerPoolInterface.

### class pymor.parallel.basic.RemoteObject(pool, remote_id, uid=None)
Bases: pymor.parallel.interfaces.RemoteObjectInterface

#### Methods

- RemoteObjectInterface: remove

#### Attributes

- RemoteObjectInterface: removed

### class pymor.parallel.basic.WorkerPoolBase
Bases: pymor.parallel.basic.WorkerPoolDefaultImplementations, pymor.parallel.interfaces.WorkerPoolInterface

#### Methods

- WorkerPoolBase: apply, apply_only, map, push
- WorkerPoolDefaultImplementations: implementor_strong, scatter_list
- WorkerPoolInterface: __len__
- BasicInterface: disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, __setattr__

#### Attributes

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

### apply (function, *args, **kwargs)

Apply function in parallel on each worker.

This calls function on each worker in parallel, passing args as positional and kwargs as keyword arguments. Keyword arguments which are RemoteObjects are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

#### Parameters

- function: The function to execute on each worker.
- args: The positional arguments for function.
kwargs The keyword arguments for function.

Returns
List of return values of the function executions, ordered by worker number (from 0 to len(pool) - 1).

apply_only (function, worker, *args, **kwargs)
Apply function on a single worker.
This calls function on on the worker with number worker, passing args as positional and kwargs as keyword arguments. Keyword arguments which are RemoteObjects are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

Parameters
function The function to execute.
worker The worker on which to execute the function. (Number between 0 and len(pool) - 1.)
args The positional arguments for function.
kwargs The keyword arguments for function.

Returns
Return value of the function execution.

map (function, *args, **kwargs)
Parallel version of the builtin map function.
Each positional argument (after function) must be a sequence of same length n. map calls function in parallel on each of these n positional argument combinations, always passing kwargs as keyword arguments. Keyword arguments which are RemoteObjects are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

Parameters
function The function to execute on each worker.
args The sequences of positional arguments for function.
kwargs The keyword arguments for function.

Returns
List of return values of the function executions, ordered by the sequence of positional arguments.
`push(obj)`
Push a copy of `obj` to all workers of the pool.

A `RemoteObject` is returned as a handle to the pushed object. This object can be used as a keyword argument to `apply`, `apply_only`, `map` and will then be transparently mapped to the respective copy of the pushed object on the worker.

`Immutable` objects will be pushed only once. If the same `immutable` object is pushed a second time, the returned `RemoteObject` will refer to the already transferred copy. It is therefore safe to use `push` to ensure that a given `immutable` object is available on the worker. No unnecessary copies will be created.

**Parameters**

`obj` The object to push to all workers.

**Returns**

A `RemoteObject` referring to the pushed data.

```python
class pymor.parallel.basic.WorkerPoolDefaultImplementations
    Bases: object

    Methods
    WorkerPoolDefaultImplementations scatter_array, scatter_list
```

**default module**

```python
pymor.parallel.default.new_parallel_pool(ipython_num_engines=None,
    ipython_profile=None, allow_mpi=True)
```

Creates a new default `WorkerPool`.

If `ipython_num_engines` or `ipython_profile` is provided as an argument or set as a `default`, an `IPythonPool WorkerPool` will be created using the given parameters via the `ipcluster` script.

Otherwise, when `allow_mpi` is `True` and an MPI parallel run is detected, an `MPIPool WorkerPool` will be created.

Otherwise, a sequential run is assumed and `pymor.parallel.dummy.dummy_pool` is returned.

**Defaults**

`ipython_num_engines`, `ipython_profile`, `allow_mpi` (see `pymor.core.defaults`)
dummy module

class pymor.parallel.dummy.DummyPool
   Bases: pymor.parallel.interfaces.WorkerPoolInterface

Methods

<table>
<thead>
<tr>
<th>DummyPool</th>
<th>apply, apply_only, map, push, scatter_array, scatter_list, <strong>len</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name,</td>
</tr>
<tr>
<td></td>
<td>implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

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

__len__()  
The number of workers in the pool.

apply (function, *args, **kwargs)
   Apply function in parallel on each worker.
   This calls function on each worker in parallel, passing args as positional and kwargs as keyword arguments. Keyword arguments which are RemoteObjects are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

Parameters

| function | The function to execute on each worker. |
| args | The positional arguments for function. |
| kwargs | The keyword arguments for function. |

Returns

List of return values of the function executions, ordered by worker number (from 0 to len(pool) - 1).

apply_only (function, worker, *args, **kwargs)
   Apply function on a single worker.
   This calls function on on the worker with number worker, passing args as positional and kwargs as keyword arguments. Keyword arguments which are RemoteObjects are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

Parameters

| function | The function to execute. |
**worker**  The worker on which to execute the function. (Number between 0 and \( \text{len(pool)} - 1 \).)

**args**  The positional arguments for `function`.

**kwargs**  The keyword arguments for `function`.

---

**Returns**

Return value of the function execution.

---

```py
map(function, *args, **kwargs)
```

Parallel version of the builtin `map` function.

Each positional argument (after `function`) must be a sequence of same length \( n \). `map` calls `function` in parallel on each of these \( n \) positional argument combinations, always passing `kwargs` as keyword arguments. Keyword arguments which are `RemoteObjects` are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are `immutable` objects that have already been pushed to the workers will not be transmitted again. (`Immutable` objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

**Parameters**

- **function**  The function to execute on each worker.
- **args**  The sequences of positional arguments for `function`.
- **kwargs**  The keyword arguments for `function`.

---

**Returns**

List of return values of the function executions, ordered by the sequence of positional arguments.

---

```py
push(obj)
```

Push a copy of `obj` to all workers of the pool.

A `RemoteObject` is returned as a handle to the pushed object. This object can be used as a keyword argument to `apply`, `apply_only`, `map` and will then be transparently mapped to the respective copy of the pushed object on the worker.

`Immutable` objects will be pushed only once. If the same `immutable` object is pushed a second time, the returned `RemoteObject` will refer to the already transferred copy. It is therefore safe to use `push` to ensure that a given `immutable` object is available on the worker. No unnecessary copies will be created.

**Parameters**

- **obj**  The object to push to all workers.

---

**Returns**

A `RemoteObject` referring to the pushed data.

---

```py
scatter_array(U, copy=True)
```

Distribute `VectorArray` evenly among the workers.
On each worker a VectorArray is created holding an (up to rounding) equal amount of vectors of U. The returned RemoteObject therefore refers to different data on each of the workers.

**Parameters**

- **U** The VectorArray to distribute.
- **copy** If False, U will be emptied during distribution of the vectors.

**Returns**

A RemoteObject referring to the scattered data.

### scatter_list(l)

Distribute list of objects evenly among the workers.

On each worker a list is created holding an (up to rounding) equal amount of objects of l. The returned RemoteObject therefore refers to different data on each of the workers.

**Parameters**

- **l** The list (sequence) of objects to distribute.

**Returns**

A RemoteObject referring to the scattered data.

---

```python
class pymor.parallel.dummy.DummyRemoteObject(obj)
Bases: pymor.parallel.interfaces.RemoteObjectInterface

Methods

- RemoteObjectInterface remove

Attributes

- RemoteObjectInterface removed
```

---

**interfaces module**

```python
class pymor.parallel.interfaces.RemoteObjectInterface
Bases: object

Handle to remote data on the workers of a WorkerPool.

See documentation of WorkerPoolInterface for usage of these handles in conjunction with apply, scatter_array, scatter_list.
```

---

5.1. pymor package
Remote objects can be used as a context manager: when leaving the context, the remote object’s `remove` method is called to ensure proper cleanup of remote resources.

### Methods

| RemoteObjectInterface | remove |

### Attributes

| RemoteObjectInterface | removed |

#### removed

True, after `remove` has been called.

#### remove()

Remove the remote object from the workers.

Remove the object this handle refers to from all workers. Note that the object will only be destroyed if no other object on the worker holds a reference to that object. Moreover, immutable objects will only be destroyed if `remove` has been called on all `RemoteObjects` which refer to the object (see push).

class pymor.parallel.interfaces.WorkerPoolInterface

**Bases:** pymor.core.interfaces.BasicInterface

Interface for parallel worker pools.

*WorkerPools* allow to easily parallelize algorithms which involve no or little communication between the workers at runtime. The interface methods give the user simple means to distribute data to workers (*push*, *scatter_array*, *scatter_list*) and execute functions on the distributed data in parallel (*apply*), collecting the return values from each function call. A single worker can be instructed to execute a function using the *WorkerPoolInterface.apply_only* method. Finally, a parallelized *map* function is available, which automatically scatters the data among the workers.

All operations are performed synchronously.

### Methods

| WorkerPoolInterface | apply, apply_only, map, push, scatter_array, scatter_list, __len__ |

| BasicInterface | disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, __setattr__ |

### Attributes

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

#### __len__()

The number of workers in the pool.

#### apply(function, *args, **kwargs)

Apply function in parallel on each worker.

This calls `function` on each worker in parallel, passing `args` as positional and `kwargs` as keyword arguments. Keyword arguments which are `RemoteObjects` are automatically mapped to the respective
object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

**Parameters**

- **function**  The function to execute on each worker.
- **args**  The positional arguments for function.
- **kwargs**  The keyword arguments for function.

**Returns**

List of return values of the function executions, ordered by worker number (from 0 to \( \text{len(pool)} - 1 \)).

**apply_only** (function, worker, *args, **kwargs)

Apply function on a single worker.

This calls function on the worker with number worker, passing args as positional and kwargs as keyword arguments. Keyword arguments which are RemoteObjects are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

**Parameters**

- **function**  The function to execute.
- **worker**  The worker on which to execute the function. (Number between 0 and \( \text{len(pool)} - 1 \)).
- **args**  The positional arguments for function.
- **kwargs**  The keyword arguments for function.

**Returns**

Return value of the function execution.

**map** (function, *args, **kwargs)

Parallel version of the builtin map function.

Each positional argument (after function) must be a sequence of same length n. map calls function in parallel on each of these n positional argument combinations, always passing kwargs as keyword arguments. Keyword arguments which are RemoteObjects are automatically mapped to the respective object on the worker. Moreover, keyword arguments which are immutable objects that have already been pushed to the workers will not be transmitted again. (Immutable objects which have not been pushed before will be transmitted and the remote copy will be destroyed after function execution.)

**Parameters**

- **function**  The function to execute on each worker.
- **args**  The sequences of positional arguments for function.
**kwargs**  The keyword arguments for *function*.

**Returns**
List of return values of the function executions, ordered by the sequence of positional arguments.

**push** (*obj*)
Push a copy of *obj* to all workers of the pool.

A *RemoteObject* is returned as a handle to the pushed object. This object can be used as a keyword argument to *apply*, *apply_only*, *map* and will then be transparently mapped to the respective copy of the pushed object on the worker.

*Immutable* objects will be pushed only once. If the same *immutable* object is pushed a second time, the returned *RemoteObject* will refer to the already transferred copy. It is therefore safe to use push to ensure that a given *immutable* object is available on the worker. No unnecessary copies will be created.

**Parameters**
- **obj**  The object to push to all workers.

**Returns**
A *RemoteObject* referring to the pushed data.

**scatter_array** (*U*, *copy=True*)
Distribute *VectorArray* evenly among the workers.

On each worker a *VectorArray* is created holding an (up to rounding) equal amount of vectors of *U*. The returned *RemoteObject* therefore refers to different data on each of the workers.

**Parameters**
- **U**  The *VectorArray* to distribute.
- **copy**  If *False*, *U* will be emptied during distribution of the vectors.

**Returns**
A *RemoteObject* referring to the scattered data.

**scatter_list** (*l*)
Distribute list of objects evenly among the workers.

On each worker a list is created holding an (up to rounding) equal amount of objects of *l*. The returned *RemoteObject* therefore refers to different data on each of the workers.

**Parameters**
- **l**  The list (sequence) of objects to distribute.
Returns

A `RemoteObject` referring to the scattered data.

**ipython module**

```python
class pymor.parallel.ipython.IPythonPool(num_engines=None, **kwargs):
    Bases: pymor.parallel.basic.WorkerPoolBase

    WorkerPool based on the IPython parallel computing features.

    Parameters

    num_engines  Number of IPython engines to use. If `None`, all available engines are used.
    kwargs  Keyword arguments used to instantiate the IPython cluster client.

    Methods

    IPythonPool  __len__
    WorkerPoolBase  apply, apply_only, map, push
    WorkerPoolDefaultImplementations  scatter_array, scatter_list
    BasicInterface  disable_logging, enable_logging, has_interface_name,
    .............................
    implementor_names, implementors, lock, unlock, __setattr__

    Attributes

    BasicInterface  locked, logger, logging_disabled, name, uid

    __len__()  The number of workers in the pool.
```

```python
class pymor.parallel.ipython.RemoteId:
    Bases: int

    Methods

    int  bit_length, conjugate, __new__, __trunc__

    Attributes

    int  denominator, imag, numerator, real
```

### 5.1. pymor package
class `pymor.parallel.ipython.new_ipcluster_pool` (profile=None, cluster_id=None, num_engines=None, ipython_dir=None, min_wait=1, timeout=60)

Bases: `pymor.core.interfaces.BasicInterface`

Create a new IPython parallel cluster and connect to it.

This context manager can be used to create an `IPythonPool WorkerPool`. When entering the context a new IPython cluster is created using the `ipcluster` script and an `IPythonPool` is instantiated for the newly created cluster. When leaving the context the cluster is shut down.

Parameters

- `profile` Passed as `--profile` parameter to the `ipcluster` script.
- `cluster_id` Passed as `--cluster-id` parameter to the `ipcluster` script.
- `num_engines` Passed as `--n` parameter to the `ipcluster` script.
- `ipython_dir` Passed as `--ipython-dir` parameter to the `ipcluster` script.
- `min_wait` Wait at least this many seconds before trying to connect to the new cluster.
- `timeout` Wait at most this many seconds for all Ipython cluster engines to become available.

Methods

- `enable_logging`, `disable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `unlock`, `__setattr__`

Attributes

- `locked`, `logger`, `logging_disabled`, `name`, `uid`

manager module

class `pymor.parallel.manager.RemoteObjectManager` 

Bases: `pymor.core.interfaces.BasicInterface`

A simple context manager to keep track of `RemoteObjects`.

When leaving this context, all `RemoteObjects` that have been managed by this object will be removed.

Methods

- `manage`, `remove_objects`

Attributes

- `locked`, `logger`, `logging_disabled`, `name`, `uid`
manage(remote_object)
Add a RemoteObject to the list of managed objects.

Parameters
remote_object  The object to add to the list.

Returns
remote_object

remove_objects()
Call remove for all managed objects.

mpi module

class pymor.parallel.mpi.MPIPool
Bases: pymor.parallel.basic.WorkerPoolBase

WorkerPool based pyMOR’s MPI event loop.

Methods

<table>
<thead>
<tr>
<th></th>
<th><strong>len</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkerPoolBase</td>
<td>apply, apply_only, map, push</td>
</tr>
<tr>
<td>WorkerPoolDefaultImplementor</td>
<td>implementor_names, scatter_list</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <em>setattr</em>_</td>
</tr>
</tbody>
</table>

_Attributes

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

__len__() 
The number of workers in the pool.

pymor.parameters package

Submodules

base module

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 the dict of the shapes of the parameter components, i.e.
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 be a superset 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, e.g., an 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, when given a Parameter, parse_parameter ensures the Parameter has an appropriate ParameterType. 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.
• We override __str__ to ensure alphanumerical ordering of the keys and 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`.

**allclose (mu)**

Compare two `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.

**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`.

---

5.1. pymor package 249
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**

- `t` 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**

- `ParameterType.sid`

---

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 argument `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_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. (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 empty.

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->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 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.

If mu is not already a Parameter, 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 **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** pymor.parameters.functionals.ExpressionParameterFunctional *(expression, parameter_type, name=None)*

Bases: 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 functions class attribute.

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

**Parameters**
- **expression** A Python expression in the parameter components of the given parameter_type.
- **parameter_type** The ParameterType of the Parameters the functional expects.

**Methods**
ExpressionParameterFunctional.__repr__

GenericParameterFunctional.__reduce__

ParameterFunctionalInterface.__call__

ImmutableInterface.generate_sid, unlock, with_

BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

Parametric.build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes

ExpressionParameterFunctional.functions

ImmutableInterface.add_with_arguments, sid, sid_ignore, with_arguments

BasicInterface.locked, logger, logging_disabled, name, uid

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 pickled if the function it is wrapping can be pickled. For this reason, it is usually preferable to use ExpressionParameterFunctional instead of GenericParameterFunctional.

Parameters

parameter_type The ParameterType of the Parameters the functional expects.
mapping The function to wrap. The function has signature mapping(mu).
name The name of the functional.

Methods

GenericParameterFunctional.disable_logging

ParameterFunctionalInterface.locked

ImmutableInterface.generate_sid, unlock, with_

BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

Parametric.build_parameter_type, local_parameter, parse_parameter, strip_parameter

Attributes
evaluate \( (\mu=\text{None}) \)
Evaluate the functional for the given Parameter \( \mu \).

class pymor.parameters.functionals.ProductParameterFunctional (factors, name=\text{None})
Bases: pymor.parameters.interfaces.ParameterFunctionalInterface
Forms the product of a list of ParameterFunctionals or numbers.

Parameters
factors A list of ParameterFunctionals or numbers.
name Name of the functional.

Methods

Attributes

evaluate \( (\mu=\text{None}) \)
Evaluate the functional for the given Parameter \( \mu \).

class pymor.parameters.functionals.ProjectionParameterFunctional (component_name, component_shape, coordinates=(), name=\text{None})
Bases: pymor.parameters.interfaces.ParameterFunctionalInterface
ParameterFunctional returning a component of the given parameter.
For given parameter \( \mu \), this functional evaluates to
Parameters

**component_name**  The name of the parameter component to return.

**component_shape**  The shape of the parameter component.

**coordinates**  See above.

**name**  Name of the functional.

Methods

- **ProjectionParameterFunctional**
  - `evaluate`

- **ParameterFunctionalInterface**
  - `__call__`

- **ImmutableInterface**
  - `generate_sid`, `unlock`, `with_`

- **BasicInterface**
  - `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`

- **Parametric**
  - `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`

Attributes

- **ImmutableInterface**
  - `add_with_arguments`, `sid`, `sid_ignore`, `with_arguments`

- **BasicInterface**
  - `locked`, `logger`, `logging_disabled`, `name`, `uid`

- **Parametric**
  - `parameter_local_type`, `parameter_space`, `parameter_type`, `parametric`

**evaluate** *(mu=None)*

Evaluate the functional for the given `Parameter` `mu`.

**interfaces module**

class pymor.parameters.interfaces.ParameterFunctionalInterface

Bases: `pymor.core.interfaces.ImmutableInterface`, `pymor.parameters.base.Parametric`

Interface for `Parameter` functionals.

A parameter functional is simply a function mapping a `Parameter` to a number.

Methods

- **ParameterFunctionalInterface**
  - `evaluate`

- **ImmutableInterface**
  - `generate_sid`, `unlock`, `with_`

- **BasicInterface**
  - `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `__setattr__`

- **Parametric**
  - `build_parameter_type`, `local_parameter`, `parse_parameter`, `strip_parameter`
Attributes

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

__call__(...) \leftrightarrow \chi(\ldots)

evaluate(mu=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>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterSpaceInterface</td>
<td>contains(mu)</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterSpaceInterface</td>
<td>parameter_type</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

parameter_type

    ParameterType of the space.

contains(mu)

    True if mu is contained in the space.

spaces module

class pymor.parameters.spaces.CubicParameterSpace (parameter_type, minimum=None, maximum=None, ranges=None)

    Bases: pymor.parameters.interfaces.ParameterSpaceInterface

    Simple ParameterSpace where each summand is an n-cube.

Parameters

    parameter_type  The ParameterType of the space.

    minimum         The minimum for each matrix entry of each Parameter component. Must be None if ranges is specified.
maximum  The maximum for each matrix entry of each Parameter component. Must be None if ranges
is specified.

ranges  dict whose keys agree with parameter_type and whose values are tuples (min, max) specifying the
minimum and maximum of each matrix entry of corresponding Parameter component. Must be None
if minimum and maximum are specified.

Methods

<table>
<thead>
<tr>
<th>CubicParameterSpace methods:</th>
<th>parse_parameter, sample_randomly, sample_uniformly,</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>repr</strong>, <strong>str</strong></td>
<td></td>
</tr>
<tr>
<td>ImmutableInterface methods:</td>
<td>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface methods:</td>
<td>disable_logging, enable_logging, has_interface_name,</td>
</tr>
<tr>
<td></td>
<td>implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

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

__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)
   Randomly sample Parameters from the space.

   Warning: When neither random_state nor seed are specified, repeated calls to this method will
   return the same sequence of parameters!

Parameters

count  None or number of random parameters (see below).

random_state  RandomState to use for sampling. If None, a new random state is generated using
   seed as random seed.

seed  Random seed to use. If None, the default random seed is used.

Returns

If count is None, an inexhaustible iterator returning random Parameters. Otherwise a list of count
random Parameters.

sample_uniformly (counts)
   Uniformly sample Parameters from the space.

5.1. pymor package
class pymor.playground.core.network_cache.NetworkFileSystemRegion(server_path, secret='')

Methods

NetworkFileSystemRegion clear, get, set

Attributes

NetworkFileSystemRegion persistent

class pymor.playground.core.network_cache.NetworkFileSystemRegionServer(addr, path, secret=None)

Bases: pymor.core.interfaces.BasicInterface

Methods

NetworkFileSystemRegionServer.serve_forever

BasicInterface disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, __setattr__

Attributes

BasicInterface locked, logger, logging_disabled, name, uid
pyMOR, Release 0.4.2

pymor.playground.discretizers.numpylistvectorarray.convert_to_numpy_list_vector_array

Use NumpyListVectorArrayMatrixOperator instead of NumpyMatrixOperator.

This simple function converts linear, affinely decomposed discretizations to use NumpyListVectorArrayMatrixOperator instead of NumpyMatrixOperator.

pymor.playground.functions package

Submodules

expression_function module

class pymor.playground.functions.expression_function.ExpressionFunction(expressions, variables='x y z')

Bases: pymor.core.interfaces.BasicInterface

Methods

| __call__ | BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, __setattr__ |

Attributes

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

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumpyListVectorArray</td>
<td>apply, apply_adjoint, apply_inverse, as_vector, assemble_lincomb</td>
</tr>
<tr>
<td>NumpyMatrixOperator</td>
<td>apply_inverse_adjoint, assemble, from_file, projected_to_subbasis</td>
</tr>
<tr>
<td>NumpyMatrixBasedOperator</td>
<td>export_matrix</td>
</tr>
<tr>
<td>OperatorBase</td>
<td>apply2, jacobian, pairwise_apply2, projected, <strong>add</strong>, <strong>mul</strong>, <strong>radd</strong>, <strong>str</strong></td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>restricted</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>Parameter</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>NumpyMatrixBasedOperator</td>
<td>linear, sparse</td>
</tr>
<tr>
<td>OperatorInterface</td>
<td>range, solver_options, source</td>
</tr>
<tr>
<td>ImmutableInterface</td>
<td>add_with_arguments, sid, sid_ignore, with_arguments</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</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 to a VectorArray.

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 \(op\) the adjoint \(op^*\) of \(op\) is given by:

\[
(op^*(v), u)_s = (v, op(u))_r,
\]

where \(( , )_s\) and \(( , )_r\) denote the inner products on the source and range space of \(op\). If \(op\) and the two products are given by the matrices \(M, P_s\) and \(P_r\), then:

\[
op^*(v) = P_s^\dagger \times M^T \times P_r \times v,
\]
with $M^T$ denoting the transposed of $M$. Thus, if $(\cdot, \cdot)_s$ and $(\cdot, \cdot)_r$ are the Euclidean inner products, $op^*v$ is simply given by multiplication of the matrix of $op$ 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 inner product `Operator` on the source space. If `None`, the Euclidean product is chosen.
- **range_product** The inner product `Operator` on the range space. If `None`, the Euclidean product is chosen.

**Returns**

`VectorArray` of the adjoint operator evaluations.

**apply_inverse** ($V$, **ind=None**, **mu=None**, **least_squares=False**)  
Apply the inverse operator.

**Parameters**

- **V** `VectorArray` of vectors to which the inverse operator is applied.
- **ind** The indices of the vectors in $V$ to which the inverse operator shall be applied (see `VectorArray` documentation for further details).
- **mu** The `Parameter` for which to evaluate the inverse operator.
- **least_squares** If `True`, solve the least squares problem:

$$u = \text{argmin} \; ||op(u) - v||_2.$$  

Since for an invertible operator the least squares solution agrees with the result of the application of the inverse operator, setting this option should, in general, have no effect on the result for those operators. However, note that when no appropriate `solver_options` are set for the operator, most implementations will choose a least squares solver by default which may be undesirable.

**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.\n
In the case of a functional, the identity\n\[
\text{self.as_vector}(\mu)\cdot\text{dot}(U) = \text{self.apply}(U, \mu)
\]\nholds, whereas in the case of a vector-like operator we have\n\[
\text{self.as_vector}(\mu) = \text{self.apply}(\text{NumpyVectorArray}(1), \mu).
\]

**Parameters**\n\mu\ The\ Parameter\ for\ which\ to\ return\ the\ vector\ representation.\n
**Returns**\n\V\ VectorArray\ of\ length\ 1\ containing\ the\ vector\ representation.\ \V\ belongs\ to\ self.source\ for\ functionals\ and\ to\ self.range\ for\ vector-like\ operators.\n
**assemble_lincomb**\n(\text{operators},\ \text{coefficients},\ \text{solver_options}=None,\ \text{name}=None)\n
Try\ to\ assemble\ a\ linear\ combination\ of\ the\ given\ operators.\n
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.\n
**Parameters**\n\text{operators}\ List\ of\ Operators\ whose\ linear\ combination\ is\ formed.\n\text{coefficients}\ List\ of\ the\ corresponding\ linear\ coefficients.\n\text{solver_options}\ solver_options\ for\ the\ assembled\ operator.\n\text{name}\ Name\ of\ the\ assembled\ operator.\n
**Returns**\nThe\ assembled\ \text{Operator}\ if\ assembly\ is\ possible,\ otherwise\ None.\n
**pymor.playground.vectorarrays package**\n
**Submodules**\n
**disk module**
class pymor.playground.vectorarrays.disk.DiskVectorArray(vectors, subtype=())
Bases: pymor.vectorarrays.interfaces.VectorArrayInterface

VectorArray implementation via a list of vectors stored in temporary files.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiskVectorArray, append, axpy, components, copy, destroy, dot, get, gramian, l1_norm, l2_norm, lincomb, make_array, pairwise_dot, remove, replace, scal, <strong>len</strong>, <strong>str</strong></td>
<td></td>
</tr>
<tr>
<td>VectorArray, check_ind_unique, empty, len_ind, len_ind_unique, sup_norm, zeros, <strong>add</strong>, <strong>iadd</strong>, <strong>imul</strong>, <strong>isub</strong>, <strong>mul</strong>, <strong>neg</strong>, <strong>radd</strong>, <strong>sub</strong></td>
<td></td>
</tr>
<tr>
<td>BasicInterface, disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiskVectorArray</td>
<td>data, dim, subtype</td>
</tr>
<tr>
<td>VectorArray</td>
<td>space</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

__len__()  
The number of vectors in the array.

__str__() == str(x)

amax(ind=None)  
The maximum absolute value of the vectors contained in the array.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ind</td>
<td>Indices of the vectors whose maximum absolute value is to be calculated (see class documentation).</td>
</tr>
</tbody>
</table>

**Returns**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_ind</td>
<td>NumPy array containing for each vector an index at which the maximum is attained.</td>
</tr>
<tr>
<td>max_val</td>
<td>NumPy array containing for each vector the maximum absolute value of its components.</td>
</tr>
</tbody>
</table>

append(other, o_ind=None, remove_from_other=False)  
Append vectors to the array.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>other</td>
<td>A VectorArray containing the vectors to be appended.</td>
</tr>
<tr>
<td>o_ind</td>
<td>Indices of the vectors that are to be appended (see class documentation).</td>
</tr>
<tr>
<td>remove_from_other</td>
<td>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.</td>
</tr>
</tbody>
</table>

axpy(alpha, x, ind=None, x_ind=None)  
BLAS AXPY operation.
This method forms the sum

\[
\text{self}[\text{ind}] = \alpha \times \text{x}[\text{x_ind}] + \text{self}[\text{ind}]
\]

If the length of \(\text{x}[\text{x_ind}]\) is 1, the same \(\text{x}\) vector is used for all vectors in \(\text{self}\). Otherwise, the lengths of \(\text{self}[\text{ind}]\) and \(\text{x}[\text{x_ind}]\) have to agree. If \(\alpha\) is a scalar, each \(\text{x}\) vector is multiplied with the same factor \(\alpha\). Otherwise, \(\alpha\) has to be a one-dimensional NumPy array of the same length as \(\text{self}[\text{ind}]\) containing the coefficients for each \(\text{x}\) vector.

**Parameters**

- **alpha** The scalar coefficient or one-dimensional NumPy array of coefficients with which the vectors in \(\text{x}\) are multiplied.
- **x** A VectorArray containing the \(\text{x}\)-summands.
- **ind** Indices of the vectors of \(\text{self}\) that are to be added (see class documentation). Repeated indices are forbidden.
- **x_ind** Indices of the vectors in \(\text{x}\) that are to be added (see class documentation). Repeated indices are allowed.

**components** \((\text{component_indices}, \text{ind}=None)\)

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 retrieved (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** \((\text{ind}=None)\)

Returns a copy of a subarray.

All VectorArray implementations in pyMOR have copy-on-write semantics: if not specified otherwise by setting deep to True, the returned copy will hold a handle to the same array data as the original array, and a deep copy of the data will only be performed when one of the arrays is modified.

Note that for NumpyVectorArray, a deep copy is always performed when only some vectors in array are copied (i.e. \text{ind} is specified).

**Parameters**

- **ind** Indices of the vectors that are to be copied (see class documentation).
- **deep** Ensure that an actual copy of the array data is made (see above).

**Returns**
A copy of the VectorArray.

**dot** *(other, ind=None, o_ind=None)*

Returns the inner products between VectorArray elements.

**Parameters**

- **other** A VectorArray containing the second factors.
- **ind** Indices of the vectors whose inner products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in other whose inner products are to be taken (see class documentation).

**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 self.dot(self, ind=ind, o_ind=ind).

**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 \( \text{result}[i] \) 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 \( \text{result}[i] \) 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).
**Indices of the vectors which are linear combined (see class documentation).**

**Returns**

A :class:`VectorArray` `result` such that

\[
\text{result[i]} = \text{self[j]} \times \text{coefficients[i,j]}
\]

in case `coefficients` is of dimension 2, otherwise `len(result) == 1` and

\[
\text{result[0]} = \text{self[j]} \times \text{coefficients[j]}.
\]

**pairwise_dot** (*other*, *ind=None*, *o_ind=None*)

Returns the pairwise inner products between :class:`VectorArray` elements.

**Parameters**

*other*  A :class:`VectorArray` containing the second factors.

*ind*  Indices of the vectors whose inner products are to be taken (see class documentation).

*o_ind*  Indices of the vectors in `other` whose inner products are to be taken (see class documentation).

**Returns**

A NumPy array `result` such that

\[
\text{result[i]} = (\text{self[ind[i]]}, \text{other[o_ind[i]]}).
\]

**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 :class:`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

```python
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.

---

```python
pymor.playground.vectorarrays.disk.basedir(path='/tmp/pymor.diskarray.docs')
```

```python
pymor.playground.vectorarrays.disk.cleanup()
```

**mpi module**

```python
pymor.playground.vectorarrays.mpi.random_array(dims, length, seed)
```

```python
pymor.playground.vectorarrays.mpi.random_list_array(dims, length, seed)
```

**Submodules**

- **progressbar module**
- **pymor.reductors package**

**Submodules**

- **basic module**

```python
class pymor.reductors.basic.GenericRBReconstructor(RB)
    Bases: pymor.core.interfaces.BasicInterface
    Simple reconstructor forming linear combinations with a reduced basis.
```

**Methods**
GenericRBReconstructor

Attributes

reconstruct \((U)\)
Reconstruct high-dimensional vector from reduced vector U.

restricted_to_subbasis \((dim)\)
See projected_to_subbasis.

class pymor.reductors.basic.SubbasisReconstructor \((dim, \quad \text{dim}_{subbasis}, \quad \text{old}\_reconstructor=\text{None})\)
Bases: pymor.core.interfaces.BasicInterface
Returned by reduce_to_subbasis.

Methods

Attributes

reconstruct \((U)\)
Reconstruct high-dimensional vector from reduced vector U.

pymor.reductors.basic.reduce_generic_rb \((\text{discretization}, \quad RB, \quad \text{vector}\_product=\text{None}, \quad \text{disable}\_caching=\text{True}, \quad \text{extends}=\text{None})\)
Generic reduced basis reductor.

Replaces each Operator of the given Discretization with the Galerkin 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.

vector_product Inner 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* (ignored).

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 (empty).

```python
pymor.reductors.basic.reduce_to_subbasis(discretization, dim, reconstructor=None)
```

Further reduce a *Discretization* to the subbasis formed by the first `dim` basis vectors.

This is achieved by calling `projected_to_subbasis` for each operator of the given *Discretization*. Additionally, if a reconstructor for the *Discretization* is provided, its `restricted_to_subbasis` method is also called to obtain a reconstructor for the further reduced *Discretization*. Otherwise `SubbasisReconstructor` is used (which will be less efficient).

**Parameters**

- **discretization** The *Discretization* to further reduce.
- **dim** The dimension of the subbasis.
- **reconstructor** Reconstructor for `discretization` or `None`.

**Returns**

- **rd** The further reduced *Discretization*.
- **rc** Reconstructor for `rd`.

---

**coercive module**

```python
class pymor.reductors.coercive.ReduceCoerciveEstimator(residual, residual_range_dims, coercivity_estimator)
```

Bases: `pymor.core.interfaces.ImmutableInterface`

Instantiated by `reduce_coercive`.

Not to be used directly.

**Methods**

- `ReduceCoerciveEstimator.estimate, restricted_to_subbasis`
- `ImmutableInterface.generate_sid, unlock, with_`
- `BasicInterface.disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__`

5.1. pymor package
class pymor.reductors.coercive.ReduceCoerciveSimpleEstimator(estimator_matrix, coercivity_estimator)

Bases: pymor.core.interfaces.ImmutableInterface

Instantiated by reduce_coercive_simple.
Not to be used directly.

Methods

ReduceCoerciveSimpleEstimator
estimate, restricted_to_subbasis

ImmutableInterface
generate_sid, unlock, with_

BasicInterface
disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

Attributes

ImmutableInterface
add_with_arguments, sid, sid_ignore, with_arguments

BasicInterface
locked, logger, logging_disabled, name, uid

pymor.reductors.coercive.reduce_coercive(discretization, RB, error_product=None, coercivity_estimator=None, disable_caching=True, extends=None)

Reducer for StationaryDiscretizations with coercive linear operator.
This reducer uses reduce_generic_rb for the actual reduce basis projection. The only addition is an error estimator which evaluates the dual norm of the residual with respect to a given inner product. For the reduction of the residual we use reduce_residual for improved numerical stability [BEOR14].

Parameters

discretization The Discretization which is to be reduced.

RB VectorArray containing the reduced basis on which to project.

error_product Inner 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).

```python
def pymor.reductors.coercive.reduce_coercive_simple(discretization, RB, error_product=None, coercivity_estimator=None, disable_caching=True, extends=None):
    """Reductor for linear StationaryDiscretizations with affinely decomposed operator and rhs."
    """
    Note: The reductor reduce_coercive can be used for arbitrary coercive StationaryDiscretizations and offers an improved error estimator with better numerical stability.

    This reductor uses reduce_generic_rb for the actual reduced basis 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 Inner product Operator used to calculate the 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).
```

linear module

```python
def pymor.reductors.linear.reduce_stationary_affine_linear(*args, **kwargs):
    """DEPRECATED! Renamed to pymor.reductors.coercive.reduce_coercive_simple."
```

5.1. pymor package
parabolic module

class pymor.reductors.parabolic.ReduceParabolicEstimator (residual, residual_range_dims, initial_residual, initial_residual_range_dims, coercivity_estimator)

Bases: pymor.core.interfaces.ImmutableInterface

Instantiated by reduce_parabolic.

Not to be used directly.

Methods

ReduceParabolicEstimator estimate, restricted_to_subbasis
ImmutableInterface generate_sid, unlock, with_
BasicInterface disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, __setattr__

Attributes

ImmutableInterface add_with_arguments, sid, sid_ignore, with_arguments
BasicInterface locked, logger, logging_disabled, name, uid

pymor.reductors.parabolic.reduce_parabolic (discretization, RB, product=None, coercivity_estimator=None, disable_caching=True, extends=None)

Reducer for parabolic equations.

This reductor uses reduce_generic_rb for the actual RB-projection. The only addition is the assembly of an error estimator which bounds the discrete L2-in time / energy-in space error similar to [GP05], [HO08] as follows:

\[ \left( C_a^{-1}(\mu) \| e_N(\mu) \|^2 + \sum_{n=1}^N \Delta t \| e_n(\mu) \|^2 \right)^{1/2} \leq \left[ C_a^{-1}(\mu) \Delta t \sum_{n=1}^N \| \mathcal{R}^n(u_n(\mu), \mu) \|_{\mathcal{E},-1}^2 + C_a^{-1}(\mu) \| e_0 \|^2 \right]^{1/2} \]

Here, \( \| \cdot \| \) denotes the norm induced by the problem’s mass matrix (e.g. the L^2-norm) and \( \| \cdot \|_{\mathcal{E}} \) is an arbitrary energy norm w.r.t. which the space operator \( A(\mu) \) is coercive, and \( C_a(\mu) \) is a lower bound for its coercivity constant. Finally, \( \mathcal{R}^n \) denotes the implicit Euler timestepping residual for the (fixed) time step size \( \Delta t \),

\[ \mathcal{R}^n(u_n(\mu), \mu) := f - M u_n(\mu) - u_n-1(\mu) \Delta t - A(u_n(\mu), \mu), \]

where \( M \) denotes the mass operator and \( f \) the source term. The dual norm of the residual is computed using the numerically stable projection from [BEOR14].

Warning: The reduced basis \( \text{RB} \) is required to be orthonormal w.r.t. the given energy product. If not, the projection of the initial values will be computed incorrectly.

Parameters
**discretization** The `InstationaryDiscretization` which is to be reduced.

**RB** `VectorArray` containing the reduced basis on which to project.

**product** The energy inner product `Operator` w.r.t. the reduction error is estimated. RB must be to be orthonormal w.r.t. this product!

**coercivity_estimator** None or a `ParameterFunctional` returning a lower bound $C_a(\mu)$ for the coercivity constant of `discretization.operator` w.r.t. `product`.

**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).

---

**residual module**

---

**class** `pymor.reductors.residual.ImplicitEulerResidualOperator` *(operator, mass, functional, dt, name=None)*

Bases: `pymor.operators.basic.OperatorBase`

Returned by `reduce_implicit_euler_residual`.

**Methods**

<table>
<thead>
<tr>
<th>ImplicitEulerResidualOperator</th>
<th>apply, projected_to_subbasis</th>
</tr>
</thead>
<tbody>
<tr>
<td>OperatorBase</td>
<td>apply2, apply_adjoint, apply_inverse, apply_inverse_adjoint, as_vector, assemble, jacobian, pairwise_apply2, 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>generate_sid, unlock, with_</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, <strong>setattr</strong></td>
</tr>
<tr>
<td>Parametric</td>
<td>build_parameter_type, local_parameter, parse_parameter, strip_parameter</td>
</tr>
</tbody>
</table>

**Attributes**
apply \((U, U_{\text{old}}, \text{ind}, \text{ind}_{\text{old}}=\text{None}, \mu=\text{None})\)
Apply the operator to a VectorArray.

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.

class pymor.reductors.residual.NonProjectedImplicitEulerResidualOperator (operator, mass, functional, dt, product)
Bases: pymor.reductors.residual.ImplicitEulerResidualOperator
Returned by reduce_implicit_euler_residual.
Not to be used directly.

Methods

NonProjectedImplicitEulerResidualOperator
    apply
    projected_to_subbasis

OperatorBase
    apply2, apply_adjoint, apply_inverse,
    apply_inverse_adjoint, as_vector, assemble, jacobian,
    pairwise_apply2, projected, __add__, __mul__, __radd__,
    __str__

OperatorInterface
    assemble_lincomb, restricted

ImmutableInterface
    generate_sid, unlock, with_

BasicInterface
    disable_logging, enable_logging, has_interface_name,
    implementor_names, implementors, lock, __setattr__

Parametric
    build_parameter_type, local_parameter, parse_parameter,
    strip_parameter

Attributes
apply \((U, U\_old, ind=None, ind\_old=None, mu=None)\)

Apply the operator to a VectorArray.

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.NonProjectedReconstructor (product)

Bases: pymor.core.interfaces.ImmutableInterface

Returned by reduce_residual.

Not to be used directly.

Methods

- **reconstruct**
- **generate_sid**, **unlock**, **with_**

Attributes

- **add_with_arguments, sid, sid_ignore, with_arguments**
- **locked, logger, logging_disabled, name, uid**

class pymor.reductors.residual.NonProjectedResidualOperator (operator, rhs, rhs_is_functional, product)

Bases: pymor.reductors.residual.ResidualOperator

Returned by reduce_residual.

Not to be used directly.

Methods
Apply the operator to a `VectorArray`.

**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, rhs, rhs_is_functional=True, name=None)

Bases: pymor.operators.basic.OperatorBase

Returned by `reduce_residual`.

Methods
apply \((U, \text{ind}=\text{None}, \text{mu}=\text{None})\)

Apply the operator to a \textbf{VectorArray}.

**Parameters**

\*U* \textbf{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 \textbf{VectorArray} documenta-
tion for further details).

\*mu* The \textbf{Parameter} for which to evaluate the operator.

**Returns**

\textbf{VectorArray} of the operator evaluations.

pymor.reductors.residual.\textbf{reduce_implicit_euler_residual} \((\text{operator}, \text{mass}, \text{dt}, \text{func-
tional}=\text{None}, \text{RB}=\text{None}, \text{product}=\text{None}, \text{extends}=\text{None})\)

Reduced basis residual reductor with mass operator for implicit Euler timestepping.

Given an operator, mass and a functional, the concatenation of residual operator with the Riesz isomorphism is
given by:

\[ \text{riesz_residual.apply}(U, \text{U}_\text{old}, \text{mu}) = \text{product.apply_inverse(operator.apply}(U, \text{mu}) + 1/\text{dt} \times \text{mass.apply}(U, \text{mu}) - 1/\text{dt} \times \text{mass.apply}(\text{U}_\text{old}, \text{mu}) - \text{functional.as_vector}(	ext{mu})) \]

This reductor determines a low-dimensional subspace of the image of a reduced basis space under \text{riesz_residual} using \textbf{estimate_image_hierarchical}, computes an orthonormal basis
\textbf{residual_range} of this range space and then returns the Petrov-Galerkin projection.
of the `riesz_residual` operator. Given reduced basis coefficient vectors \( u \) and \( u_{\text{old}} \), the dual norm of the residual can then be computed as

\[
\text{projected} \_\text{riesz} \_\text{residual}.\text{apply}(u, u_{\text{old}}, \mu).\text{l2}_\text{norm}()
\]

Moreover, a `reconstructor` is provided such that

\[
\text{reconstructor}.\text{reconstruct}(\text{projected} \_\text{riesz} \_\text{residual}.\text{apply}(u, u_{\text{old}}, \mu))
\]

\[= \text{riesz} \_\text{residual}.\text{apply}(\text{RB}.\text{lincomb}(u), \text{RB}.\text{lincomb}(u_{\text{old}}), \mu)
\]

**Parameters**

- `operator` See definition of `riesz_residual`.
- `mass` The mass operator. See definition of `riesz_residual`.
- `dt` The time step size. 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` Inner 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).

**pymor.reductors.residual.reduce_residual**

Generic reduced basis residual reductor.

Given an operator and a right-hand side, the residual is given by:

\[
\text{residual}.\text{apply}(U, \mu) = \text{operator}.\text{apply}(U, \mu) - \text{rhs}.\text{as}_\text{vector}(\mu)
\]

When the rhs is a functional we are usually interested in the Riesz representative of the residual:

\[
\text{residual}.\text{apply}(\bar{U}, \mu)
\]

\[= \text{product}.\text{apply}_\text{inverse}(\text{operator}.\text{apply}(\bar{U}, \mu) - \text{rhs}.\text{as}_\text{vector}(\mu))
\]

Given a basis `RB` of a subspace of the source space of `operator`, this method uses `estimate_image_hierarchical` to determine a low-dimensional subspace containing the image of the
subspace under residual (resp. riesz_residual), computes an orthonormal basis residual_range
for this range space and then returns the Petrov-Galerkin projection

```python
projected_residual
    === residual.projected(range_basis=residual_range, source_basis=RB)
```

of the residual operator. Given an reduced basis coefficient vector u, w.r.t. RB, the (dual) norm of the residual
can then be computed as

```python
projected_residual.apply(u, mu).l2_norm()
```

Moreover, a reconstructor is provided such that

```python
reconstructor.reconstruct(projected_residual.apply(u, mu))
    == residual.apply(RB.lincomb(u), mu)
```

Parameters

- **operator**: See definition of residual.
- **rhs**: See definition of residual. If None, zero right-hand side is assumed.
- **rhs_is_functional**: Set this to True when rhs is a Functional.
- **RB**: VectorArray containing a basis of the reduced space onto which to project.
- **product**: Inner product Operator w.r.t. which to compute the Riesz representatives in case
  rhs_is_functional is True. When product is None, no Riesz representatives are computed
- **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_residual**: See above.
- **reconstructor**: See above.
- **reduction_data**: Additional data produced by the reduction process (compare the extends parameter).

### stationary module

```python
pymor.reductors.stationary.reduce_stationary_coercive(*args, **kwargs)
```

DEPRECATED! Renamed to pymor.reductors.coercive.reduce_coercive.

### pymor.tools package

#### Submodules

#### context module
class pymor.tools.context.NoContext
    Bases: object

counter module

class pymor.tools.counter.Counter(start=0)
    Bases: object

    Methods
    Counter inc

floatcmp module

def pymor.tools.floatcmp.float_cmp(x, y, rtol=1e-14, atol=1e-14):
    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)

    def 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

inverse module

pymor.tools.inverse.inv_transposed_two_by_two(A)
    Efficiently compute the tranposed inverses of a NumPy array of 2x2-matrices
    | retval[i1,...,ik,m,n] = numpy.linalg.inv(A[i1,...,ik,:,:]).

pymor.tools.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,:,:]).

io module

pymor.tools.io.load_matrix(path, key=None)

mpi module

This module provides helper methods to use pyMOR in parallel with MPI.

Executing this module will run event_loop on all MPI ranks except for rank 0 where either a given script is executed:
or an interactive session is started:

```
mpirun -n 16 python -m pymor.tools.mpi
```

When IPython is available, an IPython kernel is started which the user can connect to by calling:

```
ipython console --existing file_name_printed_by_ipython.json
```

(Starting the IPython console directly will not work properly with most MPI implementations.) When IPython is not available, the builtin Python REPL is started.

When `event_loop` is running on the MPI ranks, `call` can be used on rank 0 to execute the same Python function (given as first argument) simultaneously on all MPI ranks (including rank 0). Calling `quit` will exit `event_loop` on all MPI ranks.

Additionally, this module provides several helper methods which are intended to be used in conjunction with `call`: `mpi_info` will print a summary of all active MPI ranks, `run_code` will execute the given code string on all MPI ranks, `import_module` imports the module with the given path.

A simple object management is implemented with the `manage_object`, `get_object` and `remove_object` methods. It is the user’s responsibility to ensure that calls to `manage_object` are executed in the same order on all MPI ranks to ensure that the returned `ObjectId` refers to the same distributed object on all ranks. The functions `function_call`, `function_call_manage`, `method_call`, `method_call_manage` map instances `ObjectId` transparently to distributed objects. `function_call_manage` and `method_call_manage` will call `manage_object` on the return value and return the corresponding `ObjectId`. The functions `method_call` and `method_call_manage` are given an `ObjectId` and a string as first and second argument and execute the method named by the second argument on the object referred to by the first argument.

---

```python
class pymor.tools.mpi.ObjectId
    Bases: int

    A handle to an MPI distributed object.

    Methods
    int | bit_length, conjugate, __new__, __trunc__

    Attributes
    int | denominator, imag, numerator, real

pymor.tools.mpi.call(method, *args, **kwargs)
    Execute method on all MPI ranks.

    Assuming `event_loop` is running on all MPI ranks (except rank 0), this will execute `method` on all ranks (including rank 0) with positional arguments `args` and keyword arguments `kwargs`.

    Parameters
    method  The function to execute on all ranks (must be picklable).
```

---

Chapter 5. API Documentation
**args**  The positional arguments for `method`.

**kwargs**  The keyword arguments for `method`.

---

**Returns**

The return value of `method` on rank 0.

---

```python
pymor.tools.mpi.event_loop()
```

Launches an MPI-based event loop.

Events can be sent by either calling `call` on rank 0 to execute an arbitrary method on all ranks or by calling `quit` to exit the loop.

---

```python
pymor.tools.mpi.event_loop_settings(auto_launch=True)
```

Settings for pyMOR’s MPI event loop.

**Parameters**

- **auto_launch**  If True, automatically execute `event_loop` on all MPI ranks (except 0) when pyMOR is imported.

**Defaults**

- `auto_launch` (see `pymor.core.defaults`)

---

```python
pymor.tools.mpi.function_call(f, *args, **kwargs)
```

Execute the function `f` with given arguments.

Intended to be used in conjunction with `call`. Arguments of type `ObjectId` are transparently mapped to the object they refer to.

---

```python
pymor.tools.mpi.function_call_manage(f, *args, **kwargs)
```

Execute the function `f` and manage the return value.

Intended to be used in conjunction with `call`. The return value of `f` is managed by calling `manage_object` and the corresponding `ObjectId` is returned. Arguments of type `ObjectId` are transparently mapped to the object they refer to.

---

```python
pymor.tools.mpi.get_object(obj_id)
```

Return the object referred to by `obj_id`.

---

```python
pymor.tools.mpi.import_module(path)
```

Import the module named by `path`.

Intended to be used in conjunction with `call`.
pymor.tools.mpi.manage_object(obj)
  Keep track of obj and return an ObjectId handle.

pymor.tools.mpi.method_call(obj_id, name_, *args, **kwargs)
  Execute a method with given arguments.
  Intended to be used in conjunction with call. Arguments of type ObjectId are transparently mapped to the object they refer to.

  Parameters
  
  obj_id  The ObjectId of the object on which to call the method.
  
  name_  Name of the method to call.
  
  args  Positional arguments for the method.
  
  kwargs  Keyword arguments for the method.

pymor.tools.mpi.method_call_manage(obj_id, name_, *args, **kwargs)
  Execute a method with given arguments and manage the return value.
  Intended to be used in conjunction with call. The return value of the called method is managed by calling manage_object and the corresponding ObjectId is returned. Arguments of type ObjectId are transparently mapped to the object they refer to.

  Parameters
  
  obj_id  The ObjectId of the object on which to call the method.
  
  name_  Name of the method to call.
  
  args  Positional arguments for the method.
  
  kwargs  Keyword arguments for the method.

pymor.tools.mpi.mpi_info()
  Print some information on the MPI setup.
  Intended to be used in conjunction with call.

pymor.tools.mpi.quit()
  Exit the event loop on all MPI ranks.
  This will cause event_loop to terminate on all MPI ranks.

pymor.tools.mpi.remove_object(obj_id)
  Remove the object referred to by obj_id from the registry.
pymor.tools.mpi.run_code(code)
Execute the code string code.
Intended to be used in conjunction with call.

pprint module

pymor.tools.pprint.format_array(array, 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)

quadratures module

class pymor.tools.quadratures.GaussQuadratures
Bases: object
Gauss quadrature on the interval [0, 1]

Methods

GaussQuadratures
iter_quadrature, maxpoints, quadrature

Attributes

GaussQuadratures
a, order_map, orders, points, weights

classmethod iter_quadrature(order=None, npoints=None)
iterates over a quadrature tuple wise

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

5.1. pymor package
pyMOR, Release 0.4.2

random module

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

timing module

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:

```python
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

```python
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

**pymor.vectorarrays package**

**Submodules**

**block module**

```python
class pymor.vectorarrays.block.BlockVectorArray(blocks, copy=False)
```
Bases: `pymor.vectorarrays.interfaces.VectorArrayInterface`

*VectorArray* where each vector is a direct sum of sub-vectors.

Given a list of equal length `VectorArrays` `blocks`, this `VectorArray` represents the direct sums of the vectors contained in the arrays.

The *subtype* of the array will be the tuple

```
(blocks[0].space, blocks[1].space, ..., blocks[-1].space).
```
BlockVectorArray can be used in conjunction with BlockOperator.

Parameters
blocks The list of sub-arrays.
copy If True, copy all arrays contained in blocks.

Methods
BlockVectorArray: append, axpy, block, components, copy, dot, l1_norm, l2_norm, lincomb, make_array, pairwise_dot, remove, replace, scal, sup_norm, __len__
VectorArray: check_ind_unique, empty, gramian, len_ind, len_ind_unique, zeros, __add__, __iadd__, __imul__, __isub__, __mul__, __neg__, __radd__, __sub__
BasicInterface: checkable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, __setattr__

Attributes
BlockVectorArray: data, dim, num_blocks, subtype
VectorArrayInterface: space
BasicInterface: locked, logger, logging_disabled, name, uid

__len__() The number of vectors in the array.
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

```python
self[ind] = alpha*x[x_ind] + self[ind]
```

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 retrieved (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, deep=False)*

Returns a copy of a subarray.

All VectorArray implementations in pyMOR have copy-on-write semantics: if not specified otherwise by setting `deep` to True, the returned copy will hold a handle to the same array data as the original array, and a deep copy of the data will only be performed when one of the arrays is modified.

Note that for NumpyVectorArray, a deep copy is always performed when only some vectors in array are copied (i.e. `ind` is specified).

**Parameters**

- **ind**  Indices of the vectors that are to be copied (see class documentation).
**deep** Ensure that an actual copy of the array data is made (see above).

**dot** *(other, ind=None, o_ind=None)*

Returns the inner products between VectorArray elements.

**Parameters**

*other* A VectorArray containing the second factors.

*ind* Indices of the vectors whose inner products are to be taken (see class documentation).

*o_ind* Indices of the vectors in other whose inner products are to be taken (see class documentation).

**Returns**

A NumPy array result such that

\[
\text{result}[i, j] = (\text{self}[\text{ind}[i]], \text{other}[\text{o_ind}[j]])
\]

**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  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
result[0] = self[j] * coefficients[j].

pairwise_dot (other, ind=None, o_ind=None)
Returns the pairwise inner products between VectorArray elements.

Parameters
other  A VectorArray containing the second factors.
ind  Indices of the vectors whose inner products are to be taken (see class documentation).

o_ind  Indices of the vectors in other whose inner products are to be taken (see class documentation).

Returns
A NumPy array result such that
result[i] = ( self[ind[i]], other[o_ind[i]] ).

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 forbid-
den.
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

```python
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.

**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]]`.

**constructions module**

**pymor.vectorarrays.constructions.cat_arrays (vector_arrays)**

Return a new VectorArray which a concatenation of the arrays in `vector_arrays`.

**fenics module**

**interfaces module**

**class pymor.vectorarrays.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 an external PDE solver code), 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. As such, methods like \texttt{l2\_norm} or \texttt{gramian} will always return NumPy arrays.

An implementation of the interface via \texttt{NumPy arrays} is given by \texttt{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, \texttt{VectorArray} constructors should provide a \texttt{reserve} keyword argument which allows to specify to what size the array is assumed to grow.

Most methods provide \texttt{ind} and/or \texttt{o\_ind} arguments which are used to specify on which vectors the method is supposed to operate. If \texttt{ind(o\_ind)} is \texttt{None} the whole array is selected. Otherwise, \texttt{ind} can be a single index in \texttt{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

<table>
<thead>
<tr>
<th>VectorArrayInterface</th>
<th>data, dim, space, subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging, disabled, name, uid</td>
</tr>
</tbody>
</table>

#### \texttt{data}

Implementors can provide a \texttt{data} property which returns a NumPy array of shape \texttt{(len(v), v\_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 \texttt{NumpyVectorArray}, an actual view of the internally used NumPy array is returned, so changing it, will alter the \texttt{VectorArray}. Thus, you cannot assume to own the data returned to you, in general.

#### \texttt{dim}

The dimension of the vectors in the array.

#### \texttt{space}

\texttt{VectorSpace} array the array belongs to.

#### \texttt{subtype}

Can be any Python object (e.g. can be added) if they are instances of the same class and have equal subtypes. A valid subtype has to be provided to \texttt{make\_array} and the resulting array will be of that subtype. By default, the subtype of an array is simply \texttt{None}. For \texttt{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.

\texttt{__add\_(other)}

The pairwise sum of two \texttt{VectorArrays}.

\texttt{__iadd\_(other)}

In-place pairwise addition of \texttt{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.

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

self[ind] = alpha*x[x_ind] + self[ind]

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 admissable list of indices in the sense of the class documentation.

**check_ind_unique**(ind)
Check if ind is an admissable 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 retrieved (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, deep=False)
Returns a copy of a subarray.

All VectorArray implementations in pyMOR have copy-on-write semantics: if not specified otherwise by setting deep to True, the returned copy will hold a handle to the same array data as the original array, and a deep copy of the data will only be performed when one of the arrays is modified.

Note that for NumpyVectorArray, a deep copy is always performed when only some vectors in array are copied (i.e. ind is specified).

**Parameters**

**ind** Indices of the vectors that are to be copied (see class documentation).

**deep** Ensure that an actual copy of the array data is made (see above).

**Returns**

A copy of the VectorArray.
**dot** *(other, ind=None, o_ind=None)*

Returns the inner products between *VectorArray* elements.

**Parameters**

- **other** A *VectorArray* containing the second factors.
- **ind** Indices of the vectors whose inner products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in *other* whose inner products are to be taken (see class documentation).

**Returns**

A NumPy array result such that

\[ \text{result}[i, j] = (\text{self}[\text{ind}[i]], \text{other}[\text{o_ind}[j]]) \].

**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=None)*

Shorthand for *self.dot*(self, ind=ind, o_ind=ind).

**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]].
A NumPy array result such that result[i] contains the norm of self[ind[i]].

**len_ind**(ind)
Return the number of specified indices.

**len_ind_unique**(ind)
Return the number of specified unique indices.

**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

\[
\text{result}[i] = \text{self}[\text{ind}[i]] \times \text{coefficients}[i,j]
\]

in case coefficients is of dimension 2, otherwise len(result) == 1 and

\[
\text{result}[0] = \text{self}[\text{ind}[j]] \times \text{coefficients}[j].
\]

**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.

**pairwise_dot**(other, ind=None, o_ind=None)
Returns the pairwise inner products between VectorArray elements.

**Parameters**

other A VectorArray containing the second factors.

ind Indices of the vectors whose inner products are to be taken (see class documentation).

o_ind Indices of the vectors in other whose inner products are to be taken (see class documentation).

**Returns**

A NumPy array result such that

\[
\text{result}[i] = (\text{self}[\text{ind}[i]], \text{other}[\text{o_ind}[i]])
\]
**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

```python
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.

**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.vectorarrays.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

<table>
<thead>
<tr>
<th>VectorSpace</th>
<th>empty, zeros, <strong>contains</strong>, <strong>eq</strong>, <strong>ne</strong>, <strong>repr</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

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</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)

Two spaces are equal iff their types and subtypes agree.

__ne__(other)

Two spaces are equal iff their types and subtypes agree.

__repr__() \iff repre(x)

__str__() \iff repre(x)
**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.

---

### list module

**class** pymor.vectorarrays.list.*CopyOnWriteVector*

**Bases:** pymor.vectorarrays.list.VectorInterface

**Methods**

<table>
<thead>
<tr>
<th>CopyOnWriteVector</th>
<th>axpy, copy, from_instance, scal</th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorInterface</td>
<td>amax, components, dot, l1_norm, l2_norm, make_zeros, sup_norm</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>VectorInterface</th>
<th>dim, subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

**class** pymor.vectorarrays.list.*ListVectorArray* *(vectors, subtype=(), copy=True)*

**Bases:** pymor.vectorarrays.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*.  

---

300  Chapter 5. API Documentation
Parameters

**vectors**  List of **vectors** contained in the array.

**subtype**  If **vectors** is empty, the array’s **subtype** must be provided, as the subtype cannot be derived from **vectors** in this case.

**copy**  If True, make copies of the vectors in **vectors**.

Methods

<table>
<thead>
<tr>
<th>ListVectorArray</th>
<th>append, appendpy, components, copy, dot, gramian, l1_norm, l2_norm, lincomb, make_array, pairwise_dot, remove, replace, scal, sup_norm, len, str</th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorArrayInterface</td>
<td>check_ind_unique, empty, len_ind, len_ind_unique, zeros, add, iadd, imul, isub, mul, neg, radd, sub</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, setattr</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>ListVectorArray</th>
<th>data, dim, subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorArrayInterface</td>
<td>space</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

**len**

The number of vectors in the array.

**str**  \( \Rightarrow \) str(x)

**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.

```python
def axpy(alpha, x, ind=None, x_ind=None):
    # BLAS AXPY operation.
    # This method forms the sum
    # self[ind] = alpha*x[x_ind] + self[ind]
    #
    # 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.
    
    if len(x) == 1:
        self[ind] = alpha * x[x_ind] + self[ind]
    else:
        self[ind] = alpha * x[x_ind] + self[ind]

    return self
```

components (component_indices, ind=None)
Extract components of the vectors contained in the array.

```python
def 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 retrieved (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.
    
    result = np.zeros((len(ind), len(component_indices)), dtype=component_indices.dtype)
    for i in range(len(ind)):
        for j in range(len(component_indices)):
            result[i, j] = self[ind[i]][component_indices[j]]
    return result
```

copy (ind=None, deep=False)
Returns a copy of a subarray.

All VectorArray implementations in pyMOR have copy-on-write semantics: if not specified otherwise by setting deep to True, the returned copy will hold a handle to the same array data as the original array, and a deep copy of the data will only be performed when one of the arrays is modified.

Note that for NumpyVectorArray, a deep copy is always performed when only some vectors in array are copied (i.e. ind is specified).

```python
def copy(ind=None, deep=False):
    # Returns a copy of a subarray.
    #
    # Parameters
    # ind  Indices of the vectors whose components are to be retrieved (see class documentation).
    # deep  Whether to perform a deep copy.
    #
    if deep:
        return np.array(self[ind])
    else:
        return self[ind]
```
**ind** Indices of the vectors that are to be copied (see class documentation).

**deep** Ensure that an actual copy of the array data is made (see above).

**Returns**
A copy of the `VectorArray`.

**dot** *(other, ind=None, o_ind=None)*
Returns the inner products between `VectorArray` elements.

**Parameters**
- **other** A `VectorArray` containing the second factors.
- **ind** Indices of the vectors whose inner products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in `other` whose inner products are to be taken (see class documentation).

**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 `self.dot(self, ind=ind, o_ind=ind)`.

**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\( (\text{coefficients}, \text{ind}={\text{None}}) \)
Returns linear combinations of the vectors contained in the array.

Parameters

coefficients A NumPy array of dimension 1 or 2 containing the linear coefficients. \(\text{coefficients}.\text{shape}[-1]\) has to agree with \(\text{len(self)}\).
ind Indices of the vectors which are linear combined (see class documentation).

Returns

A VectorArray result such that
\[
\text{result}[i] = \text{self}[j] \times \text{coefficients}[i,j]
\]
in case coefficients is of dimension 2, otherwise \(\text{len(result)} == 1\) and
\[
\text{result}[0] = \text{self}[j] \times \text{coefficients}[j].
\]

pairwise_dot\( (\text{other}, \text{ind}={\text{None}}, \text{o\_ind}={\text{None}}) \)
Returns the pairwise inner products between VectorArray elements.

Parameters

other A VectorArray containing the second factors.
ind Indices of the vectors whose inner products are to be taken (see class documentation).
o_ind Indices of the vectors in other whose inner products are to be taken (see class documentation).

Returns

A NumPy array result such that
\[
\text{result}[i] = ( \text{self}[\text{ind}[i]], \text{other}[\text{o\_ind}[i]] ).
\]

remove\( (\text{ind}={\text{None}}) \)
Remove vectors from the array.

Parameters

ind Indices of the vectors that are to be removed (see class documentation).

replace\( (\text{other}, \text{ind}={\text{None}}, \text{o\_ind}={\text{None}}, \text{remove\_from\_other}={\text{False}}) \)
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.
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

```python
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.

**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]]`.

---

**class** `pymor.vectorarrays.list.NumpyVector` *(instance, dtype=None, copy=False, order=None, subok=False)*

Bases: `pymor.vectorarrays.list.CopyOnWriteVector`

Vector stored in a NumPy 1D-array.

**Methods**

- `amax`, `components`, `dot`, `from_instance`, `l1_norm`, `l2_norm`, `make_zeros`
- `axpy`, `copy`, `scal`
- `sup_norm`
- `disable_logging`, `enable_logging`, `has_interface_name`, `implementor_names`, `implementors`, `lock`, `unlock`, `__setattr__`

**Attributes**

---
class pymor.vectorarrays.list.VectorInterface

Bases: pymor.core.interfaces.BasicInterface

Interface for vectors.

This Interface is intended to be used in conjunction with ListVectorArray. All pyMOR algorithms operate on VectorArrays instead of single vectors! All methods of the interface have a direct counterpart in the VectorArray interface.

Methods

- amax
- axpy
- components
- copy
- dot
- l1_norm
- l2_norm
- make_zeros
- scal
- sup_norm

Attributes

- dim
- subtype

mpi module

Wrapper classes for building MPI distributed VectorArrays.

This module contains several wrapper classes which allow to transform single rank VectorArrays into MPI distributed VectorArrays which can be used on rank 0 like ordinary VectorArrays. Similar classes are provided for handling Vectors. The implementations are based on the event loop provided by pymor.tools.mpi.

class pymor.vectorarrays.mpi.MPIVector(cls, subtype, obj_id)

Bases: pymor.vectorarrays.list.VectorInterface

MPI distributed Vector.

Given a single-rank implementation of VectorInterface cls, this wrapper class uses the event loop from pymor.tools.mpi to build MPI distributed vector where on each MPI rank an instance of cls is used to manage the local data.

Instances of MPIVector can be used on rank 0 in conjunction with ListVectorArray like any other (non-distributed) Vector class.

Note, however, that the implementation of cls needs to be MPI aware. For instance, cls.dot must perform the needed MPI communication to sum up the local inner products and return the sum on rank 0.

Default implementations for all communication requiring interface methods are provided by MPIVectorAutoComm (also see MPIVectorNoComm).

Note that resource cleanup is handled by object.__del__. Please be aware of the peculiarities of destructors in Python!
The class of the VectorInterface implementation used on every MPI rank.

subtype tuple of the different subtypes of cls on the MPI ranks. Alternatively, the length of subtype may be 1, in which case the same subtype is assumed for all ranks.

obj_id ObjectId of the MPI distributed instances of cls wrapped by this object.

### Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIVector</td>
<td>amax, axpy, components, copy, dot, l1_norm, l2_norm, make_zeros, scal</td>
</tr>
<tr>
<td>VectorInterface</td>
<td>sup_norm</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIVector</td>
<td>dim, subtype</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

The class pymor.vectorarrays.mpi.MPIVectorArray(cls, subtype, obj_id)

Bases: pymor.vectorarrays.interfaces.VectorArrayInterface

MPI distributed VectorArray.

Given a single-rank VectorArray implementation cls, this wrapper class uses the event loop from pymor.tools.mpi to build MPI distributed vector arrays where on each MPI rank an instance of cls is used to manage the local data.

Instances of MPIVectorArray can be used on rank 0 like any other (non-distributed) VectorArray.

Note, however, that the implementation of cls needs to be MPI aware. For instance, cls.dot must perform the needed MPI communication to sum up the local inner products and return the sums on rank 0.

Default implementations for all communication requiring interface methods are provided by MPIVectorArrayAutoComm (also see MPIVectorArrayNoComm).

Note that resource cleanup is handled by object.__del__. Please be aware of the peculiarities of destructors in Python!

### Parameters

- **cls** The class of the VectorArray implementation used on every MPI rank.
- **subtype** tuple of the different subtypes of cls on the MPI ranks. Alternatively, the length of subtype may be 1, in which case the same subtype is assumed for all ranks.
- **obj_id** ObjectId of the MPI distributed instances of cls wrapped by this array.

### Methods
**Attributes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIVectorArray</td>
<td>dim, subtype</td>
</tr>
<tr>
<td>VectorArrayInterface</td>
<td>data, space</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

```python
__len__() -> int
```

The number of vectors in the array.

```python
amax(ind=None) -> (max_ind, max_val)
```

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.

```python
append(other, o_ind=None, remove_from_other=False)
```

Append vectors to the array.

**Parameters**

- **other** A :class:`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.

```python
axpy(alpha, x, ind=None, x_ind=None)
```

BLAS AXPY operation.

This method forms the sum

```
sel[ind] = alpha * x[x_ind] + self[ind]
```

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  

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 retrieved (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  

Returns a copy of a subarray.

All VectorArray implementations in pyMOR have copy-on-write semantics: if not specified otherwise by setting deep to True, the returned copy will hold a handle to the same array data as the original array, and a deep copy of the data will only be performed when one of the arrays is modified.

Note that for NumpyVectorArray, a deep copy is always performed when only some vectors in array are copied (i.e. ind is specified).

Parameters

ind  Indices of the vectors that are to be copied (see class documentation).

deep  Ensure that an actual copy of the array data is made (see above).

Returns

A copy of the VectorArray.

dot  

Returns the inner products between VectorArray elements.

Parameters
other A VectorArray containing the second factors.

ind Indices of the vectors whose inner products are to be taken (see class documentation).

o_ind Indices of the vectors in other whose inner products are to be taken (see class documentation).

Returns
A NumPy array result such that

\[
\text{result}[i, j] = (\text{self}[\text{ind}[i]], \text{other}[\text{o_ind}[j]])
\]

11_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

\[
\text{result}[i] = \text{self}[j] \times \text{coefficients}[i,j]
\]
in case coefficients is of dimension 2, otherwise len(result) == 1 and
result[0] = self[j] * coefficients[j].

---

**pairwise_dot** *(other, ind=None, o_ind=None)*

Returns the pairwise inner products between *VectorArray* elements.

**Parameters**

- **other** A *VectorArray* containing the second factors.
- **ind** Indices of the vectors whose inner products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in *other* whose inner products are to be taken (see class documentation).

**Returns**

A *NumPy array* `result` such that

\[ \text{result}[i] = (\text{self}[\text{ind}[i]], \text{other}[\text{o_ind}[i]]) \]

---

**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

\[ \text{self}[\text{ind}] = \alpha \times \text{self}[\text{ind}] \]

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.vectorarrays.mpi.MPIVectorArrayAutoComm(cls, subtype, obj_id)

Bases: pymor.vectorarrays.mpi.MPIVectorArray

MPI distributed VectorArray.

This is a subclass of MPIVectorArray which provides default implementations for all communication requiring interface methods for the case when the wrapped array is not MPI aware.

Note, however, that depending on the discretization these default implementations might lead to wrong results (for instance in the presence of shared DOFs).

Methods

```
<table>
<thead>
<tr>
<th>MPIVectorArrayAutoComm</th>
<th>amax, components, dot, l1_norm, l2_norm, pairwise_dot</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIVectorArray</td>
<td>append, axpy, copy, lincomb, make_array, remove, replace, scal, <strong>len</strong></td>
</tr>
<tr>
<td>VectorArrayInterface</td>
<td>check_ind_unique, empty, gramian, len_ind, len_ind_unique, sup_norm, zeros, <strong>add</strong>, <strong>iadd</strong>, <strong>imul</strong>, <strong>isub</strong>, <strong>mul</strong>, <strong>neg</strong>, <strong>radd</strong>, <strong>sub</strong></td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>
```

Attributes

```
<table>
<thead>
<tr>
<th>MPIVectorArrayAutoComm</th>
<th>dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIVectorArray</td>
<td>subtype</td>
</tr>
<tr>
<td>VectorArrayInterface</td>
<td>data, space</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>
```

**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.

**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 retrieved (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.

**dot** *(other, ind=None, o_ind=None)*

Returns the inner products between VectorArray elements.

**Parameters**

**other** A VectorArray containing the second factors.

**ind** Indices of the vectors whose inner products are to be taken (see class documentation).

**o_ind** Indices of the vectors in other whose inner products are to be taken (see class documentation).

**Returns**

A NumPy array result such that

\[
\text{result}[i, j] = (\text{self}[\text{ind}[i]], \text{other}[\text{o_ind}[j]])
\]

**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]].
**pairwise_dot** *(other, ind=None, o_ind=None)*

Returns the pairwise inner products between `VectorArray` elements.

**Parameters**

- **other** A `VectorArray` containing the second factors.
- **ind** Indices of the vectors whose inner products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in `other` whose inner products are to be taken (see class documentation).

**Returns**

A NumPy array result such that

```python
result[i] = (self[ind[i]], other[o_ind[i]])
```

---

**class** `pymor.vectorarrays.mpi.MPIVectorArrayNoComm` *(cls, subtype, obj_id)*

Bases: `pymor.vectorarrays.mpi.MPIVectorArray`

MPI distributed `VectorArray`.

This is a subclass of `MPIVectorArray` which overrides all communication requiring interface methods to raise `NotImplementedError`.

This is mainly useful as a security measure when wrapping arrays for which simply calling the respective method on the wrapped arrays would lead to wrong results and `MPIVectorArrayAutoComm` cannot be used either (for instance in the presence of shared DOFs).

**Methods**

<table>
<thead>
<tr>
<th>Base</th>
<th><code>amax</code>, <code>components</code>, <code>dot</code>, <code>l1_norm</code>, <code>l2_norm</code>, <code>pairwise_dot</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>append</code>, <code>axpy</code>, <code>copy</code>, <code>lincomb</code>, <code>make_array</code>, <code>remove</code>, <code>replace</code>, <code>scal</code>, <code>__len__</code></td>
</tr>
<tr>
<td></td>
<td><code>check_ind</code>, <code>check_ind_unique</code>, <code>empty</code>, <code>gramian</code>, <code>len</code>, <code>len_ind</code>, <code>len_ind_unique</code>, <code>sup_norm</code>, <code>zeros</code>, <code>__add__</code>, <code>__iadd__</code>, <code>__imul__</code>, <code>__isub__</code>, <code>__mul__</code>, <code>__neg__</code>, <code>__radd__</code>, <code>__rsub__</code></td>
</tr>
<tr>
<td></td>
<td><code>BasicInterface</code>, <code>disable_logging</code>, <code>enable_logging</code>, <code>has_interface_name</code>, <code>implementor_names</code>, <code>implementors</code>, <code>lock</code>, <code>unlock</code>, <code>__setattr__</code></td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Base</th>
<th><code>dim</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>subtype</code></td>
</tr>
<tr>
<td><code>VectorArrayInterface</code></td>
<td><code>data</code>, <code>space</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></td>
</tr>
</tbody>
</table>

**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.

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 retrieved (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.

dot (other, ind=None, o_ind=None)

Returns the inner products between VectorArray elements.

Parameters

other A VectorArray containing the second factors.
ind Indices of the vectors whose inner products are to be taken (see class documentation).
o_ind Indices of the vectors in other whose inner products are to be taken (see class documentation).

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]].

pairwise_dot(other, ind=None, o_ind=None)
Returns the pairwise inner products between VectorArray elements.

Parameters
other  A VectorArray containing the second factors.
ind  Indices of the vectors whose inner products are to be taken (see class documentation).
o_ind  Indices of the vectors in other whose inner products are to be taken (see class documentation).

Returns
A NumPy array result such that
result[i] = (self[ind[i]], other[o_ind[i]]).

class pymor.vectorarrays.mpi.MPIVectorAutoComm(cls, subtype, obj_id)
Bases: pymor.vectorarrays.mpi.MPIVector

MPI distributed Vector.

This is a subclass of MPIVector which provides default implementations for all communication requiring
interface methods for the case when the wrapped vector is not MPI aware.

Note, however, that depending on the discretization these default implementations might lead to wrong results
(for instance in the presence of shared DOFs).

Attributes
### pymor.package

5.1. pymor package

#### class pymor.vectorarrays.mpi.MPIVectorNoComm

Bases: object

MPI distributed Vector.

This is a subclass of MPIVector which overrides all communication requiring interface methods to raise `NotImplementedError`.

This is mainly useful as a security measure when wrapping vectors for which simply calling the respective method on the wrapped vectors would lead to wrong results and MPIVectorAutoComm cannot be used either (for instance in the presence of shared DOFs).

#### Methods

| MPIVectorNoComm | amax, components, dot, l1_norm, l2_norm, pairwise_dot |

#### Attributes

| MPIVectorNoComm | dim |

#### class pymor.vectorarrays.mpi.RegisteredSubtype

Bases: int

#### Methods

| int | bit_length, conjugate, __new__, __trunc__ |

#### Attributes

| int | denominator, imag, numerator, real |

nump module

#### class pymor.vectorarrays.numpy.NumpyVectorArray

Bases: pymor.vectorarrays.interfaces.VectorArrayInterface

VectorArray implementation via NumPy arrays.

This is the default VectorArray type used by all Operators in pyMOR’s discretization toolkit. Moreover, all reduced Operators are based on NumpyVectorArray.

Note that this class is just a thin wrapper around the underlying NumPy array. Thus, while operations like `axpy` or `dot` will be quite efficient, removing or appending vectors will be costly.
Methods

<table>
<thead>
<tr>
<th>NumpyVectorArray</th>
<th>amax, append, axpy, components, copy, dot, from_file, l1_norm, l2_norm, lincomb, make_array, pairwise_dot, remove, replace, scal, <strong>len</strong>, <strong>repr</strong>, <strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorArrayInterface</td>
<td>check_ind, check_ind_unique, empty, gramian, len_ind, len_ind_unique, sup_norm, zeros, <strong>add</strong>, <strong>iadd</strong>, <strong>imul</strong>, <strong>isub</strong>, <strong>mul</strong>, <strong>neg</strong>, <strong>radd</strong>, <strong>rsub</strong></td>
</tr>
<tr>
<td>BasicInterface</td>
<td>disable_logging, enable_logging, has_interface_name, implementor_names, implementors, lock, unlock, <strong>setattr</strong></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>NumpyVectorArray</th>
<th>data, dim, imag, real, subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorArrayInterface</td>
<td>space</td>
</tr>
<tr>
<td>BasicInterface</td>
<td>locked, logger, logging_disabled, name, uid</td>
</tr>
</tbody>
</table>

__len__()  
The number of vectors in the array.

__repr__() <==> repr(x)
__str__() <==> str(x)

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
\[ self[ind] = \alpha \times x[x\_ind] + self[ind] \]

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** (\( component\_indices, ind=\text{None} \))

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 retrieved (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** (\( ind=\text{None}, deep=\text{False} \))

Returns a copy of a subarray.

All VectorArray implementations in pyMOR have copy-on-write semantics: if not specified otherwise by setting deep to True, the returned copy will hold a handle to the same array data as the original array, and a deep copy of the data will only be performed when one of the arrays is modified.

Note that for NumpyVectorArray, a deep copy is always performed when only some vectors in array are copied (i.e. \( ind \) is specified).

**Parameters**

- \( ind \) Indices of the vectors that are to be copied (see class documentation).
- \( deep \) Ensure that an actual copy of the array data is made (see above).
A copy of the `VectorArray`.

**dot** *(other, ind=None, o_ind=None)*

Returns the inner products between `VectorArray` elements.

**Parameters**

- **other** A `VectorArray` containing the second factors.
- **ind** Indices of the vectors whose inner products are to be taken (see class documentation).
- **o_ind** Indices of the vectors in `other` whose inner products are to be taken (see class documentation).

**Returns**

A NumPy array result such that

\[
\text{result}[i, j] = (\text{self}[\text{ind}[i]], \text{other}[\text{o_ind}[j]]).
\]

**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** 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

$$result[0] = self[j] * coefficients[j].$$

`pairwise_dot (other, ind=None, o_ind=None)`

Returns the pairwise inner products between `VectorArray` elements.

Parameters

other  A `VectorArray` containing the second factors.

ind  Indices of the vectors whose inner products are to be taken (see class documentation).

o_ind  Indices of the vectors in other whose inner products are to be taken (see class documentation).

Returns

A NumPy array result such that

$$result[i] = (self[ind[i]], other[o_ind[i]]).$$

`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
```python
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.

---

**pymor.vectorarrays.numpy.NumpyVectorSpace(dim)**

Shorthand for `VectorSpace(NumpyVectorArray, dim)`.

### 5.1.2 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**
6.1 pymordemos package

6.1.1 Submodules

analyze_pickle module

Analyze pickled data demo.

Usage: 
```
analyze_pickle.py histogram [-detailed=DETAILED_DATA] [-error-norm=NORM] REDUCED_DATA SAMPLES 
analyze_pickle.py convergence [-detailed=DETAILED_DATA] [-error-norm=NORM] [-ndim=NDIM] REDUCED_DATA SAMPLES analyze_pickle.py (-h | --help)
```

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 parameter 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.

```python
def analyze_pickle_convergence(args):
```
pyMOR, Release 0.4.2

pymordemos.analyze_pickle.analyze_pickle_demo(args)

pymordemos.analyze_pickle.analyze_pickle_histogram(args)

**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_osher) [default: lax_friedrichs].
- `-h, --help` Show this message.
- `--vx=X_SPEED` Speed in x-direction [default: 1].
- `--vy=Y_SPEED` 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:

--cache-region=REGION  Name of cache region to use for caching solution snapshots (NONE, MEMORY, DISK, PERSISTENT) [default: DISK]
--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].
--ipython-engines=COUNT  If positive, the number of IPython cluster engines to use for parallel greedy search. If zero, no parallelization is performed. [default: 0]
--ipython-profile=PROFILE  IPython profile to use for parallelization.

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)

**elliptic2 module**

Simple demonstration of solving the Poisson equation in 2D using pyMOR’s builtin 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)

**elliptic_unstructured module**

Simple demonstration of solving the Poisson equation in 2D on a circular sector domain of radius 1 using an unstructured mesh.

Note that Gmsh (http://geuz.org/gmsh/) is required for meshing.

**Usage:** elliptic_unstructured.py [--fv] ANGLE NUM_POINTS CLSCALE

**Arguments:** ANGLE The angle of the circular sector.

NUM_POINTS The number of points that form the arc of the circular sector.

CLSCALE Mesh element size scaling factor.

**Options:**
-h, --help
  Show this message.
--fv
  Use finite volume discretization instead of finite elements.

pymordemos.elliptic_unstructured.elliptic_gmsh_demo(args)

pymordemos.elliptic_unstructured.polar(X)

parabolic module

Simple demonstration of solving the heat equation using pyMOR’s builtin discretizations.


Arguments: TOP The heat diffusion coefficient for the top bars.
Options:
  -h, --help
    Show this message.
  --fv
    Use finite volume discretization instead of finite elements.
  --r
    Use RectGrid instead of TriaGrid.
  --g=NI
    Use grid with NIxNI intervals [default: 100].
  --n=COUNT
    Number of time steps [default: 10].

parabolic_mor module

Reduced basis approximation of the heat equation.

Usage: parabolic_mor.py [options] BACKEND ALG SNAPSHOTS RBSIZE TEST

Arguments: BACKEND Discretization toolkit to use (pymor, fenics).

  ALG The model reduction algorithm to use (greedy, adaptive_greedy, pod).
  SNAPSHOTS greedy/pod: number of training set parameters adaptive_greedy: size of validation set.
  RBSIZE Size of the reduced basis.
  TEST Number of test parameters for reduction error estimation.

6.1. pymordemos package
**pyMOR, Release 0.4.2**

```python
pymordemos.parabolic_mor.reduce_adaptive_greedy(d, reductor, validation_mus, basis_size)
```

```python
pymordemos.parabolic_mor.reduce_greedy(d, reductor, snapshots, basis_size)
```

```python
pymordemos.parabolic_mor.reduce_pod(d, reductor, snapshots, basis_size)
```

**thermalblock module**

Thermalblock demo.

**Usage:** thermalblock.py [options] XBLOCKS YBLOCKS SNAPSHOTS RBSIZE thermalblock.py -h -help

**Arguments:**
- **XBLOCKS** Number of blocks in x direction.
- **YBLOCKS** Number of blocks in y direction.
- **SNAPSHOTS** naive: ignored
  - greedy/pod: Number of training set parameters per block (in total SNAPSHOTS^(XBLOCKS * YBLOCKS) parameters).
  - adaptive_greedy: size of validation set.
- **RBSIZE** Size of the reduced basis

**Options:**

- **--adaptive-greedy-rho=RHO**  See pymor.algorithms.adaptivegreedy [default: 1.1].
- **--adaptive-greedy-gamma=GAMMA**  See pymor.algorithms.adaptivegreedy [default: 0.2].
- **--adaptive-greedy-theta=THETA**  See pymor.algorithms.adaptivegreedy [default: 0.]
- **--alg=ALG**  The model reduction algorithm to use (naive, greedy, adaptive_greedy, pod) [default: greedy].
- **--cache-region=REGION**  Name of cache region to use for caching solution snapshots (none, memory, disk, persistent) [default: none].
- **--estimator-norm=NORM**  Norm (euclidean, h1) in which to calculate the residual [default: h1].
- **--extension-alg=ALG**  Basis extension algorithm (trivial, gram_schmidt, h1_gram_schmidt) to be used [default: h1_gram_schmidt].
- **--fenics**  Use FEniCS discretization.
- **--grid=NI**  Use grid with 4*NI*NI elements [default: 100].
- **-h, --help**  Show this message.
- **--ipython-engines=COUNT**  If positive, the number of IPython cluster engines to use for parallel greedy search. If zero, no parallelization is performed. [default: 0]
- **--ipython-profile=PROFILE**  IPython profile to use for parallelization.
- **--order=ORDER**  Polynomial order of the Lagrange finite elements to use in FEniCS discretization [default: 1].
--pickle=PREFIX  Pickle reduced discretization, as well as reconstructor and high-dimensional discretization to files with this prefix.
--plot-err  Plot error.
--plot-solutions  Plot some example solutions.
--plot-error-sequence  Plot reduction error vs. basis size.
--pod-product=PROD  Inner product w.r.t. with to compute the pod (euclidean, h1) [default: h1].
--reductor=RED  Reductor (error estimator) to choose (traditional, residual_basis) [default: residual_basis]
--test=COUNT  Use COUNT snapshots for stochastic error estimation [default: 10].
--greedy-without-estimator  Do not use error estimator for basis generation.

pymordemos.thermalblock.discretize_fenics(xblocks, yblocks, grid_num_intervals, element_order)

pymordemos.thermalblock.discretize_pymor(xblocks, yblocks, grid_num_intervals, use_list_vector_array)

pymordemos.thermalblock.parse_arguments(args)

pymordemos.thermalblock.reduce_adaptive_greedy(d, reductor, validation_mus, extension_alg_name, max_extensions, use_estimator, rho, gamma, theta, pool)

pymordemos.thermalblock.reduce_greedy(d, reductor, snapshots_per_block, extension_alg_name, max_extensions, use_estimator, pool)

pymordemos.thermalblock.reduce_naive(d, reductor, basis_size)

pymordemos.thermalblock.reduce_pod(d, reductor, snapshots_per_block, product_name, basis_size)

pymordemos.thermalblock.thermalblock_demo(args)
thermalblock_adaptive module

Modified thermalblock demo using adaptive greedy basis generation algorithm.

Usage: thermalblock_adaptive.py [options] RBSIZE
thermalblock_adaptive.py -h | --help

Arguments: RBSIZE Size of the reduced basis

Options:

- `h`, `--help` Show this message.
- `--estimator-norm=NORM` Norm (trivial, h1) in which to calculate the residual [default: h1].
- `--without-estimator` Do not use error estimator for basis generation.
- `--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].
- `--pickle=PREFIX` Pickle reduced discretizaion, as well as reconstructor and high-dimensional discretization to files with this prefix.
- `--plot-err` Plot error.
- `--plot-solutions` Plot some example solutions.
- `--plot-error-sequence` Plot reduction error vs. basis size.
- `--reductor=RED` Reductor (error estimator) to choose (traditional, residual_basis) [default: residual_basis]
- `--test=COUNT` Use COUNT snapshots for stochastic error estimation [default: 10].
- `--ipython-engines=COUNT` If positive, the number of IPython cluster engines to use for parallel greedy search. If zero, no parallelization is performed. [default: 0]
- `--ipython-profile=PROFILE` IPython profile to use for parallelization.
- `--cache-region=REGION` Name of cache region to use for caching solution snapshots (NONE, MEMORY, DISK, PERSISTENT) [default: NONE]
- `--visualize-refinement` Visualize the training set refinement indicators.
- `--validation-mus` Size of validation set. [default: 0]
- `--rho=VALUE` Maximum allowed ratio between error on validation set and on training set [default: 1.1].
- `--gamma=VALUE` Weight factor for age penalty term in refinement indicators [default: 0.2].
- `--theta=VALUE` Ratio of elements to refine [default: 0.].
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
```

---

**class** `pymordemos.thermalblock_gui.ParamRuler` *(parent, sim)*

`Bases: object`

`Methods`

```
ParamRuler enable
```

---

**class** `pymordemos.thermalblock_gui.RBGui` *(args)*

`Bases: QMainWindow`

---

6.1. pymordemos package
class pymordemos.thermalblock_gui.ReducedSim(args)
    Bases: pymordemos.thermalblock_gui.SimBase

Methods
ReducedSim solve

class pymordemos.thermalblock_gui.SimBase(args)
    Bases: object

class pymordemos.thermalblock_gui.SimPanel(parent, sim)
    Bases: object

Methods
SimPanel solve_update

thermalblock_simple module

Simplified version of the thermalblock demo.

Usage: thermalblock_simple.py [options] MODEL ALG SNAPSHOTS RBSIZE TEST

Arguments: MODEL High-dimensional model (pymor, fenics).
        ALG The model reduction algorithm to use (naive, greedy, adaptive_greedy, pod).
        SNAPSHOTS naive: ignored
        greedy/pod: Number of training_set parameters per block (in total SNAPSHOTS^(XBLOCKS * YBLOCKS) parameters).
        adaptive_greedy: size of validation set.
        RBSIZE Size of the reduced basis.
        TEST Number of parameters for stochastic error estimation.

pymordemos.thermalblock_simple.discretize_fenics()

pymordemos.thermalblock_simple.discretize_pymor()

pymordemos.thermalblock_simple.main()

pymordemos.thermalblock_simple.reduce_adaptive_greedy(d, reductor, validation_mus, basis_size)
pymordemos.thermalblock_simple.reduce_greedy \( (d, \text{reductor}, \text{snapshots}, \text{basis\_size}) \)

pymordemos.thermalblock_simple.reduce_naive \( (d, \text{reductor}, \text{basis\_size}) \)

pymordemos.thermalblock_simple.reduce_pod \( (d, \text{reductor}, \text{snapshots}, \text{basis\_size}) \)


## Symbols

<table>
<thead>
<tr>
<th>Method</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__add__()</code></td>
<td>102, 137, 209, 293</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>61, 106, 117, 177, 256, 259, 299</td>
</tr>
<tr>
<td><code>__contains__()</code></td>
<td>299</td>
</tr>
<tr>
<td><code>__eq__()</code></td>
<td>95, 299</td>
</tr>
<tr>
<td><code>__hash__()</code></td>
<td>95</td>
</tr>
<tr>
<td><code>__iadd__()</code></td>
<td>293</td>
</tr>
<tr>
<td><code>__imul__()</code></td>
<td>294</td>
</tr>
<tr>
<td><code>__isub__()</code></td>
<td>294</td>
</tr>
<tr>
<td><code>__len__()</code></td>
<td>239, 242, 245, 247, 263, 288, 294, 301, 304, 308, 318, 324, 328, 331, 339, 343, 347, 351, 355, 359, 363</td>
</tr>
<tr>
<td><code>__mul__()</code></td>
<td>102, 137, 210, 294</td>
</tr>
<tr>
<td><code>__ne__()</code></td>
<td>95, 299</td>
</tr>
<tr>
<td><code>__neg__()</code></td>
<td>103, 299</td>
</tr>
<tr>
<td><code>__new__()</code></td>
<td>72</td>
</tr>
<tr>
<td><code>__radd__()</code></td>
<td>103</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__add__()</code></td>
<td>102, 137, 209, 293</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>61, 106, 117, 177, 256, 259, 299</td>
</tr>
<tr>
<td><code>__contains__()</code></td>
<td>299</td>
</tr>
<tr>
<td><code>__eq__()</code></td>
<td>95, 299</td>
</tr>
<tr>
<td><code>__hash__()</code></td>
<td>95</td>
</tr>
<tr>
<td><code>__iadd__()</code></td>
<td>293</td>
</tr>
<tr>
<td><code>__imul__()</code></td>
<td>294</td>
</tr>
<tr>
<td><code>__isub__()</code></td>
<td>294</td>
</tr>
<tr>
<td><code>__len__()</code></td>
<td>239, 242, 245, 247, 263, 288, 294, 301, 304, 308, 318, 324, 328, 331, 339, 343, 347, 351, 355, 359, 363</td>
</tr>
<tr>
<td><code>__mul__()</code></td>
<td>102, 137, 210, 294</td>
</tr>
<tr>
<td><code>__ne__()</code></td>
<td>95, 299</td>
</tr>
<tr>
<td><code>__neg__()</code></td>
<td>103, 299</td>
</tr>
<tr>
<td><code>__new__()</code></td>
<td>72</td>
</tr>
<tr>
<td><code>__radd__()</code></td>
<td>103</td>
</tr>
<tr>
<td>Function</td>
<td>Class/Method</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>apply2()</td>
<td>pymor.operators.interfaces.OperatorInterface</td>
</tr>
<tr>
<td>apply2()</td>
<td>pymor.operators.mpi.MPIOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.basic.OperatorBase</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.block.BlockOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.constructions.AdjointOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.constructions.Concatenation</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.constructions.FixedParameterOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.constructions.IdentityOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.constructions.LincombOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.constructions.SelectionOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.constructions.VectorArrayOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.interfaces.OperatorInterface</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.mpi.MPIOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.numpy.NumpyMatrixBasedOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.operators.numpy.NumpyMatrixOperator</td>
</tr>
<tr>
<td>apply_adjoint()</td>
<td>pymor.playground.operators.numpy.NumpyListVectorArrayMatrixOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.playground.operators.numpy.NumpyListVectorArrayMatrixOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.basic.OperatorBase</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.block.BlockDiagonalOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.constructions.AdjointOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.constructions.Concatenation</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.constructions.FixedParameterOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.constructions.IdentityOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.constructions.LincombOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.constructions.SelectionOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.constructions.VectorArrayOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.interfaces.OperatorInterface</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.mpi.MPIOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.numpy.NumpyMatrixBasedOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.operators.numpy.NumpyMatrixOperator</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.parallel.basic.WorkerPoolBase</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.parallel.dummy.DummyPool</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.parallel.interfaces.WorkerPoolInterface</td>
</tr>
<tr>
<td>apply_inverse()</td>
<td>pymor.domaindescriptions.polygonal.CircularSectorDomain</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.basic.OperatorBase</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.block.BlockDiagonalOperator</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.constructions.AdjointOperator</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.constructions.Concatenation</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.constructions.IdentityOperator</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.constructions.LincombOperator</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.constructions.SelectionOperator</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.constructions.VectorArrayOperator</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.interfaces.OperatorInterface</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.mpi.MPIOperator</td>
</tr>
<tr>
<td>as_vector()</td>
<td>pymor.operators.numpy.NumpyMatrixOperator</td>
</tr>
</tbody>
</table>
as_vector() (pymor.playground.operators.numpy.NumpyListVectorArrayMatrixOperator method), 261
assemble() (pymor.operators.basic.OperatorBase method), 140
assemble() (pymor.operators.basicProjectedOperator method), 143
assemble() (pymor.operators.constructions.IdentityOperator method), 175
assemble() (pymor.operators.constructions.LinearCombinationOperator method), 180
assemble() (pymor.operators.constructions.SelectionOperator method), 184
assemble() (pymor.operators.interfaces.OperatorInterface method), 213
assemble() (pymor.operators.mpi.MPIOperator method), 219
assemble() (pymor.operators.numpy.NumpyMatrixBasedOperator method), 226
assemble() (pymor.operators.numpy.NumpyMatrixOperator method), 230
assemble_lincomb() (pymor.operators.constructions.IdentityOperator method), 176
assemble_lincomb() (pymor.operators.constructions.LinearCombinationOperator method), 188
assemble_lincomb() (pymor.operators.constructions.SelectionOperator method), 193
assemble_lincomb() (pymor.operators.interfaces.OperatorInterface method), 213
assemble_lincomb() (pymor.operators.mpi.MPIOperator method), 220
assemble_lincomb() (pymor.operators.numpy.NumpyMatrixOperator method), 230
assemble_lincomb() (pymor.playground.operators.numpy.NumpyListVectorArrayMatrixOperator method), 262
axpy() (pymor.playground.vectorarrays.disk.DiskVectorArray method), 263
axpy() (pymor.vectorarrays.block.BlockVectorArray method), 288
axpy() (pymor.vectorarrays.interfaces.VectorArrayInterface method), 294
axpy() (pymor.vectorarrays.list.ListVectorArray method), 302
axpy() (pymor.vectorarrays.mpi.MPIVectorArray method), 308
axpy() (pymor.vectorarrays.numpy.NumpyVectorArray method), 318

B
basedir() (in module pymor.playground.vectorarrays.disk), 267
BasicInterface (class in pymor.core.interfaces), 69
BitmapFunction (class in pymor.functions.bitmap), 105
block() (pymor.vectorarrays.block.BlockVectorArray method), 289
BlockDiagonalOperator (class in pymor.operators.block), 144
BlockOperator (class in pymor.operators.block), 146
BlockVectorArray (class in pymor.vectorarrays.block), 287
boundary (pymor.domain.descriptions.basic.CylindricalDomain attribute), 92
boundary (pymor.domain.descriptions.basic.RectangularDomain attribute), 94
boundary() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 116
boundary (pymor.domain.descriptions.polygons.DiscDomain attribute), 98
boundary_mask() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 116
boundary_types (pymor.domain.descriptions.interfaces.DomainDescriptionInterface attribute), 96
boundary_types (pymor.domain.descriptions.polygons.PolygonalDomain attribute), 98
boundary_types (pymor.grids.interfaces.BoundaryInfoInterface attribute), 115
BoundaryInfoFromIndicators (class in pymor.grids.boundaryinfos), 107
BoundaryInfoInterface (class in pymor.grids.interfaces), 114
BoundaryType (class in pymor.domain.descriptions.boundarytypes), 95
bounding_box() (pymor.grids.interfaces.AffineGridInterface method), 113
bounding_box() (pymor.grids.oned.OnedGrid method), 119
bounding_box() (pymor.grids.rect.RectGrid method), 129
bounding_box() (pymor.grids.tria.TriaGrid method), 129
build_parameter_type() (pymor.parameters.base.Parametric method), 251
Burgers2DBumpInitialData (class in pymor.analyticalproblems.burgers), 46
Burgers2DFlux (class in pymor.analyticalproblems.burgers), 46
Burgers2DFluxDerivative (class in pymor.analyticalproblems.burgers), 47
Burgers2DProblem (class in pymor.analyticalproblems.burgers), 47
Burgers2DSinInitialData (class in pymor.analyticalproblems.burgers), 48
burgers_demo() (in module pymordemos.burgers), 324
burgers_demo() (in module pymordemos.burgers_ei), 325
BurgersBumpInitialData (class in pymor.analyticalproblems.burgers), 49
BurgersFlux (class in pymor.analyticalproblems.burgers), 49
BurgersFluxDerivative (class in pymor.analyticalproblems.burgers), 50
BurgersProblem (class in pymor.analyticalproblems.burgers), 50
BurgersSinInitialData (class in pymor.analyticalproblems.burgers), 51
busywait() (in module pymor.tools.timing), 287
coefficients (pymor.functions.basic.LincombFunction attribute), 104
ColorBarWidget (class in pymor.gui.gl), 132
ColoredFormatter (class in pymor.core.logger), 73
compile_vertex_shader() (in module pymor.gui.gl), 132
ComponentProjection (class in pymor.operators.constructions), 162
components() (pymor.playground.vectorarrays.disk.DiskVectorArray method), 264
components() (pymor.vectorarrays.block.BlockVectorArray method), 289
components() (pymor.vectorarrays.interfaces.VectorArrayInterface method), 295
components() (pymor.vectorarrays.list.ListVectorArray method), 302
components() (pymor.vectorarrays.mpi.MPIVectorArray method), 309
components() (pymor.vectorarrays.mpi.MPIVectorArrayAutoComm method), 312
components() (pymor.vectorarrays.mpi.MPIVectorArrayNoComm method), 315
components() (pymor.vectorarrays.numpy.NumpyVectorArray method), 319
Concatenation (class in pymor.operators.constructions), 163
ConformalTopologicalGridDefaultImplementations (class in pymor.grids.defaultimpl), 110
ConformalTopologicalGridInterface (class in pymor.grids.interfaces), 115
ConstantFunction (class in pymor.functions.basic), 100
ConstantOperator (class in pymor.operators.constructions), 166
ConstError (class in pymor.core.exceptions), 66
contains() (pymor.parameters.interfaces.ParameterSpaceInterface method), 256
contains() (pymor.parameters.spaces.CubicParameterSpace method), 257
convert_to_numpy_list_vector_array() (in module pymor.playground.discretizers.numpylistvectorarray), 259
copy() (pymor.playground.vectorarrays.disk.DiskVectorArray method), 264
copy() (pymor.vectorarrays.block.BlockVectorArray method), 289
copy() (pymor.vectorarrays.interfaces.VectorArrayInterface method), 295
copy() (pymor.vectorarrays.list.ListVectorArray method), 302
copy() (pymor.vectorarrays.mpi.MPIVectorArray method), 309
copy() (pymor.vectorarrays.numpy.NumpyVectorArray method), 319
CopyOnWriteVector (class in pymor.vectorarrays.list), 300
Counter (class in pymor.tools.counter), 280
CubicParameterSpace (class in pymor.parameters.spaces), 256
CylindricalDomain (class in pymor.domaindescriptions.basic), 92
Data (pymor.vectorarrays.interfaces.VectorArrayInterface attribute), 293
DecoratorBase (class in pymor.core.decorators), 62
DecoratorWithArgsBase (class in pymor.core.decorators), 62
default_regions() (in module pymor.core.cache), 62
DefaultContainer (class in pymor.core.defaults), 64
defaul() (in module pymor.core.defaults), 64
defaults() (in module pymor.core.defaults), 64
defaults_sid() (in module pymor.core.defaults), 64
dim() (in module pymor.algorithms.ei), 28
dime() (module pymor.algorithms.ei), 28
dense_options() (in module pymor.operators.numpy), 232
Depreciated (class in pymor.core.decorators), 63
DetailedSim (class in pymordemos.thermalblock_gui), 331
diameters() (pymor.grids.interfaces.AffineGridInterface method), 113
diffusion_functionals (pymor.analyticalproblems.elliptic.EllipticProblem attribute), 53
diffusion_functionals (pymor.analyticalproblems.parabolic.ParabolicProblem attribute), 55
diffusion_functions (pymor.analyticalproblems.elliptic.EllipticProblem attribute), 53
diffusion_functions (pymor.analyticalproblems.parabolic.ParabolicProblem attribute), 55
DiffusionOperator (class in pymor.operators.fv), 199
DiffusionOperatorP1 (class in pymor.operators.cg), 150
DiffusionOperatorQ1 (class in pymor.operators.cg), 151
dim (pymor.domaindescriptions.interfaces.DomainDescriptionInterface method), 96
dim (pymor.grids.interfaces.ConformalTopologicalGridInterface attribute), 116
dim (pymor.grids.interfaces.ReferenceElementInterface attribute), 117
dim (pymor.vectorarrays.interfaces.VectorArrayInterface attribute), 293
dim_domain (pymor.functions.interfaces.FunctionInterface attribute), 106
dim_outer (pymor.grids.interfaces.AffineGridInterface attribute), 113
dirichlet_data (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 46
dirichlet_data (pymor.analyticalproblems.parabolic.ParabolicProblem attribute), 55
dirichlet_data (pymor.analyticalproblems.parabolic.ParabolicProblem attribute), 55
discrete_caching() (in module pymor.core.cache), 62
discrete_caching() (pymor.core.cache.CacheableInterface method), 60
discrete_logging() (pymor.core.interfaces.BasicInterface method), 70
DiscDomain (class in pymor.domaindescriptions.polygonal), 97
DiscretizationBase (class in pymor.discretizations.basic), 76
DiscretizationInterface (class in pymor.discretizations.interfaces), 81
discretize_domain_default() (in module pymor.domaindiscretizers.default), 99
discretize_elliptic() (in module pymor.discretizers.elliptic), 88
discretize_elliptic_fv() (in module pymor.discretizers.elliptic), 89
discretize_fenics() (in module pymor.fenics.parabolic_mor), 327
discretize_fenics() (in module pymor.fenics.thermalblock), 329
discretize_fenics() (in module pymor.fenics.thermalblock_simple), 332
discretize_gmsh() (in module pymor.domaindiscretizers.gmsh), 99
discretize_nonlinear_instationary_from_disk() (in module pymor.discretizers.disk), 86
discretize_instationary_from_disk() (in module pymor.discretizers.disk), 86
discretize_nonlinear_instationary_advection_fv() (in module pymor.discretizers.advection), 85
discretize_parabolic() (in module pymor.discretizers.parabolic), 90
discretize_parabolic_fv() (in module pymor.discretizers.parabolic), 90
discretize_pymor() (in module pymor.domaindiscretizers.pymor.thermalblock), 327
discretize_pymor() (in module pymor.domaindiscretizers.pymor.thermalblock), 329
discretize_pymor() (in module pymor.domaindiscretizers.pymor.thermalblock), 329
discretize_pymor() (in module pymor.domaindiscretizers.pymor.thermalblock), 329
discretize_stationary_from_disk() (in module pymor.discretizers.disk), 87
DiskVectorArray (class in pymor.playground.vectorarrays.disk), 263
domain (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 46
domain (pymor.analyticalproblems.elliptic.EllipticProblem attribute), 46
domain (pymor.analyticalproblems.parabolic.ParabolicProblem attribute), 53
domain (pymor.domaindescriptions.basic.CircleDomain attribute), 92
evaluate() (pymor.parameters.functionals.ProjectionParameterFunctional method), 255
evaluate() (pymor.parameters.interfaces.ParameterFunctionalInterface method), 256
evaluate_coefficients() (pymor.functions.basic.LincombFunction method), 104
evaluate_coefficients() (pymor.operators.constructions.LinearOperator method), 180
event_loop() (in module pymor.mpi), 283
event_loop_settings() (in module pymor.mpi), 283
explicit_euler() (in module pymor.algorithms.timestepping), 44
ExplicitEulerTimeStepper (class in pymor.algorithms.timestepping), 41
export_matrix() (pymor.operators.numpy.NumpyMatrixBasedOperator method), 226
ExpressionFunction (class in pymor.functions.basic), 101
ExpressionFunction (class in pymor.functions.basic.ExpressionFunction), 259
ExpressionParameterFunctional (class in pymor.parameters.functionals), 252
ExtensionError (class in pymor.core.exceptions), 66

F
FixedParameterOperator (class in pymor.operators.constructions), 169
fixup_docstring() (in module pymor.core.decorators), 63
flatten_grid() (in module pymor.grids.constructions), 109
float_cmp() (in module pymor.tools.floatcmp), 280
float_cmp_all() (in module pymor.tools.floatcmp), 280
flux_function (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 46
flux_function_derivative (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 46
format_array() (in module pymor.tools.prettyprint), 285
from_parameter_type() (py:meth:`pymor.parameters.base.Parameter`), 249
FrozenDict (class in pymor.tools.frozendict), 281
function_call() (in module pymor.mpi), 283
function_call_manage() (in module pymor.mpi), 283
functionals (pymor.discretizations.interfaces.DiscretizationInterface attribute), 81
FunctionBase (class in pymor.functions.basic), 102
FunctionInterface (class in pymor.functions.interfaces), 105
functions (pymor.functions.basic.LincombFunction attribute), 104
G
GaussQuadratures (class in pymor.tools.quadratures), 285
generate_sid() (in module pymor.core.interfaces), 73
generate_sid() (pymor.core.interfaces.ImmutableInterface method), 71
GenericFunction (class in pymor.functions.basic), 103
GenericParameterFunctional (class in pymor.parameters.functionals), 253
GenericRBReconstructor (class in pymor.reductors.basic), 267
get() (pymor.core.cache.CacheRegion method), 59
get_object() (in module pymor.mpi), 283
getLogger() (in module pymor.core.logger), 74
global_to_structured() (pymor.grids.rect.RectGrid method), 121
GLPatchWidget (class in pymor.gui.gl), 132
GmshBoundaryInfo (class in pymor.grids.gmsh), 110
GmshError (class in pymor.core.exceptions), 67
GmshGrid (class in pymor.grids.gmsh), 111
gram_schmidt() (class in pymor.algorithms.gram_schmidt), 35
gram_schmidt_basis_extension() (class in pymor.algorithms.basisextension), 26
gramian() (pymor.playground.vectorarrays.disk.DiskVectorArray method), 265
gramian() (pymor.vectorarrays.interfaces.VectorArrayInterface method), 296
gramian() (pymor.vectorarrays.list.ListVectorArray method), 303
greedy() (in module pymor.algorithms.greedy), 36
H
has_interface_name() (pymor.core.interfaces.BasicInterface method), 70
HelmholtzProblem (class in pymor.analyticalproblems.helmholtz), 53
holes (pymor.domaindescriptions.polygonal.PolygonalDomain attribute), 98
hstack() (pymor.operators.block.BlockOperator method), 147
I
IdentityOperator (class in pymor.operators.constructions), 172
ImageCollectionError (class in pymor.core.exceptions), 67
ImmutableInterface (class in pymor.core.interfaces), 70
ImmutableMeta (class in pymor.core.interfaces), 72
implementor_names() (pymor.core.interfaces.BasicInterface method), 70
implementors() (pymor.core.interfaces.BasicInterface method), 70
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>implicit_euler()</td>
<td>in module pyMor.algorithms.timestepping, 44</td>
</tr>
<tr>
<td>ImplicitEulerResidualOperator</td>
<td>(class in pyMor.reductors.residual), 273</td>
</tr>
<tr>
<td>ImplicitEulerTimeStepper</td>
<td>(class in pyMor.algorithms.timestepping), 42</td>
</tr>
<tr>
<td>import_module()</td>
<td>(in module pymor.tools.mpi), 283</td>
</tr>
<tr>
<td>indices_from_parent_indices()</td>
<td>(in module pymor.grids.subgrid.SubGrid), 128</td>
</tr>
<tr>
<td>induced_norm()</td>
<td>(in module pyMor.operators.constructions), 194</td>
</tr>
<tr>
<td>InducedNorm</td>
<td>(class in pymor.operators.constructions), 176</td>
</tr>
<tr>
<td>initial_data</td>
<td>(pymor.analyticalproblems.advection.InstationaryAdvectionProblem), 46</td>
</tr>
<tr>
<td>initial_data</td>
<td>(pymor.analyticalproblems.parabolic.ParabolicProblem), 55</td>
</tr>
<tr>
<td>initial_data</td>
<td>(pymor.discretizations.basic.InstationaryDiscretization), 77</td>
</tr>
<tr>
<td>InstationaryAdvectionProblem</td>
<td>(class in pymor.analyticalproblems.advection), 45</td>
</tr>
<tr>
<td>InstationaryDiscretization</td>
<td>(class in pymor.discretizations.basic), 77</td>
</tr>
<tr>
<td>integration_elements()</td>
<td>(in module pymor.grids.interfaces.AffineGridInterface), 113</td>
</tr>
<tr>
<td>interpolate_operators()</td>
<td>(in module pymor.algorithms.ei), 30</td>
</tr>
<tr>
<td>InterpolationOperator</td>
<td>(class in pymor.operators.cg), 152</td>
</tr>
<tr>
<td>inv_transposed_two_by_two()</td>
<td>(in module pymor.tools.inverse), 281</td>
</tr>
<tr>
<td>inv_two_by_two()</td>
<td>(in module pymor.tools.inverse), 281</td>
</tr>
<tr>
<td>InversionError</td>
<td>(class in pymor.core.exceptions), 67</td>
</tr>
<tr>
<td>IPythonPool</td>
<td>(class in pymor.parallel.ipython), 245</td>
</tr>
<tr>
<td>iter_quadrature()</td>
<td>(in module pymor.tools.quadratures), 285</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.basic.OperatorBase), 141</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.basicProjectedOperator), 144</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.constructions.Concatenation), 165</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.constructions.ConstantOperator), 167</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.constructions.FixedParameterOperator), 172</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.constructions.LinearOperator), 180</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.ei.EmpiricalInterpolatedOperator), 196</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.operators.Interface), 201</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.optimization.OptimizationProblem), 205</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.interfaces.OperatorInterface), 214</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.mpi.MPIOperator), 220</td>
</tr>
<tr>
<td>jacobian_inverse_transposed()</td>
<td>(in module pymor.grids.interfaces.AffineGridInterface), 113</td>
</tr>
<tr>
<td>jacobian_options()</td>
<td>(in module pymor.operators.constructions), 208</td>
</tr>
<tr>
<td>l1_norm()</td>
<td>(in module pymor.playground.vectorarrays.list.ListVectorArray), 265</td>
</tr>
<tr>
<td>l2_norm()</td>
<td>(in module pymor.playground.vectorarrays.list.ListVectorArray), 265</td>
</tr>
<tr>
<td>L2Product</td>
<td>(class in pymor.operators.fv), 201</td>
</tr>
<tr>
<td>L2ProductFunctional</td>
<td>(class in pymor.operators.fv), 201</td>
</tr>
<tr>
<td>L2ProductFunctionalP1</td>
<td>(class in pymor.operators.cg), 153</td>
</tr>
<tr>
<td>L2ProductFunctionalQ1</td>
<td>(class in pymor.operators.cg), 154</td>
</tr>
<tr>
<td>L2ProductQ1</td>
<td>(class in pymor.operators.cg), 155</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.fv.NonlinearAdvectionOperator), 205</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.interfaces.OperatorInterface), 214</td>
</tr>
<tr>
<td>jacobian()</td>
<td>(pymor.operators.mpi.MPIOperator), 220</td>
</tr>
</tbody>
</table>
Index

M
main() (in module pymordemos.parabolic_mor), 327
main() (in module pymordemos.thermalblock_simple), 332
make_array() (pymor.vectorarrays.interfaces.VectorArrayInterface class method), 297
manage() (pymor.parallel.manager.RemoteObjectManager method), 246
manage_object() (in module pymor.tools mpi), 284
map() (pymor.parallel.basic.WorkerPoolBase method), 237
map() (pymor.parallel.dummy.DummyPool method), 240
map() (pymor.parallel.interfaces.WorkerPoolInterface method), 243
mapped_diameter() (pymor.grids.interfaces.ReferenceElementInterface method), 117
mapped_diameter() (pymor.grids.referenceelements.Line method), 122
mapped_diameter() (pymor.grids.referenceelements.Point method), 124
mapped_diameter() (pymor.grids.referenceelements.Square method), 125
mapped_diameter() (pymor.grids.referenceelements.Triangle method), 126
mask() (pymor.grids.boundaryinfos.AllDirichletBoundaryInfo method), 107
mask() (pymor.grids.boundaryinfos.BoundaryInfoFromIndicators method), 108
mask() (pymor.grids.boundaryinfos.EmptyBoundaryInfo method), 108
mask() (pymor.grids.boundaryinfos.SubGridBoundaryInfo method), 109
mask() (pymor.grids.gmsh.GmshBoundaryInfo method), 111
mask() (pymor.grids.interfaces.BoundaryInfoInterface method), 115
mass (pymor.discretizations.basic.InstationaryDiscretization attribute), 79
Matplotlib1DVisualizer (class in pymor.gui.qt), 133
Matplotlib1DWidget (class in pymor.gui.matplotlib), 133
MatplotlibPatchWidget (class in pymor.gui.matplotlib), 133
matrix_astype_nocopy() (in module pymor.operators.numpy), 232
MemoryRegion (class in pymor.core.cache), 61
method_call() (in module pymor.tools mpi), 284
method_call_manage() (in module pymor.tools mpi), 284
Module (class in pymor.core.pickle), 75
mpi_info() (in module pymor.tools mpi), 284

346 Index
pairwise_apply2() (pymor.operators.mpi.MPIOperator method), 220
pairwise_dot() (pymor.playground.vectorarrays.disk.DiskVectorArray method), 266
pairwise_dot() (pymor.vectorarrays.block.BlockVectorArray method), 291
pairwise_dot() (pymor.vectorarrays.interfaces.VectorArrayInterface method), 297
pairwise_dot() (pymor.vectorarrays.list.ListVectorArray method), 304
pairwise_dot() (pymor.vectorarrays.mpi.MPIVectorArray method), 307
pairwise_dot() (pymor.vectorarrays.mpi.MPIVectorArrayAutoComm method), 311
pairwise_dot() (pymor.vectorarrays.mpi.MPIVectorArrayNoComm method), 314
pairwise_dot() (pymor.vectorarrays.mpi.MPIVectorArrayProjected method), 316
pairwise_dot() (pymor.vectorarrays.mpi.MPIVectorArrayUserChains method), 321
pairwise_dot() (pymor.vectorarrays.numpy.NumpyVectorArray method), 327
parabolic_demo() (in module pymordemos.parabolic), 327
ParabolicProblem (class in pymor.analyticalproblems.parabolic), 54
Parameter (class in pymor.parameters.base), 248
parameter_local_type (pymor.parameters.base.Parametric attribute), 250
parameter_space (pymor.parameters.base.Parametric attribute), 251
parameter_type (pymor.parameters.base.Parameter attribute), 249
parameter_type (pymor.parameters.base.Parametric attribute), 250
parameter_type (pymor.parameters.interfaces.ParameterSpaceInterface attribute), 256
ParameterFunctionalInterface (class in pymor.parameters.interfaces), 255
ParameterSpaceInterface (class in pymor.parameters.interfaces), 256
ParameterType (class in pymor.parameters.base), 249
Parametric (class in pymor.parameters.base), 250
parametric (pymor.parameters.base.Parametric attribute), 251
ParamRuler (class in pymordemos.thermalblock_gui), 331
parent_grid (pymor.grids.subgrid.SubGrid attribute), 127
parent_indices() (pymor.grids.subgrid.SubGrid method), 128
parse_arguments() (in module pymordemos.parabolic), 327
parse_parameter() (pymor.parameters.base.Parametric method), 252
PatchVisualizer (class in pymor.gui.qt), 134
persistent (pymor.core.cache.CacheRegion attribute), 59
PlotMainWindow (class in pymor.gui.qt), 135
pod() (in module pymor.algorithms.pod), 40
pod_basis_extension() (in module pymor.algorithms.basisextension), 27
Points (class in pymor.domains.polygonal.PolygonDomain attribute), 98
polar() (in module pymordemos.thermalblock_gui), 327
PolygonalDomain (class in pymor.domains.polygonal), 98
print_defaults() (in module pymor.core.defaults), 65
ProductParameterFunctional (class in pymor.parameters.functionals), 254
projected() (pymor.operators.basic.OperatorBase method), 141
projected() (pymor.operators.constructions.AdjointOperator method), 161
projected() (pymor.operators.constructions.Concatenation method), 165
projected() (pymor.operators.constructions.ConstantOperator method), 168
projected() (pymor.operators.constructions.LincombOperator method), 181
projected() (pymor.operators.constructions.SelectionOperator method), 185
projected() (pymor.operators.constructions.ZeroOperator method), 193
projected() (pymor.operators.ei.EmpiricalInterpolatedOperator method), 197
projected() (pymor.operators.interfaces.OperatorInterface), 214
projected_to_subbasis() (pymor.operators.basic.ProjectedOperator method), 144
projected_to_subbasis() (pymor.operators.constructions.LincombOperator method), 182
projected_to_subbasis() (pymor.operators.constructions.ZeroOperator method), 193
ProjectedEmpiricalInterpolatedOperator (class in pymor.parameters.ei), 197
ProjectedOperator (class in pymor.operators.basic), 142
ProjectionParameterFunctional (class in pymor.parameters.functionals), 254
push() (pymor.parallel.basic.WorkerPoolBase method), 237
push() (pymor.parallel.dummy.DummyPool method), 240
push() (pymor.parallel.interfaces.WorkerPoolInterface method), 244
pymor (module), 23
pymor.algorithms (module), 23
pymor.algorithms.adaptivegreedy (module), 23
Index

pyMor, Release 0.4.2

pyMor.algorithms.basic (module), 25 pyMor.algorithms.basicserialization (module), 26 pyMor.algorithms.ei (module), 28 pyMor.algorithms.error (module), 31 pyMor.algorithms.genericsolvers (module), 33 pyMor.algorithms.gram_schmidt (module), 35 pyMor.algorithms.greedy (module), 36 pyMor.algorithms.image (module), 37 pyMor.algorithms.newton (module), 39 pyMor.algorithms.pod (module), 40 pyMor.algorithms.timestepping (module), 41 pyMor.analyticalproblems (module), 44 pyMor.analyticalproblems.advection (module), 44 pyMor.analyticalproblems.burgers (module), 46 pyMor.analyticalproblems.elliptic (module), 51 pyMor.analyticalproblems.helmholtz (module), 53 pyMor.analyticalproblems.parabolic (module), 54 pyMor.analyticalproblems.thermalblock (module), 56 pyMor.basic (module), 322 pyMor.core (module), 57 pyMor.core.backends (module), 57 pyMor.core.cache (module), 58 pyMor.core.decorators (module), 62 pyMor.core.defaults (module), 63 pyMor.core.dynamic (module), 66 pyMor.core.exceptions (module), 66 pyMor.core.interfaces (module), 68 pyMor.core.logger (module), 73 pyMor.core.logger (module), 75 pyMor.discretizations (module), 76 pyMor.discretizations.basic (module), 76 pyMor.discretizations.interfaces (module), 81 pyMor.discretizations.mpi (module), 82 pyMor.discretizers (module), 85 pyMor.discretizers.advection (module), 85 pyMor.discretizers.disk (module), 86 pyMor.discretizers.elliptic (module), 88 pyMor.discretizers.parabolic (module), 89 pyMor.domaindescriptions (module), 91 pyMor.domaindescriptions.basic (module), 91 pyMor.domaindescriptions.boundarytypes (module), 95 pyMor.domaindescriptions.interfaces (module), 96 pyMor.domaindescriptions.polyginal (module), 96 pyMor.domaindiscretizers (module), 99 pyMor.domaindiscretizers.default (module), 99 pyMor.domaindiscretizers.gmsh (module), 99 pyMor.functions (module), 100 pyMor.functions.basic (module), 100 pyMor.functions.bitmap (module), 100 pyMor.functions.mpi (module), 105 pyMor.grids (module), 106 pyMor.grids._unstructured (module), 106 pyMor.grids.boundaryinfos (module), 106 pyMor.grids.constructions (module), 109 pyMor.grids.defaultimpl (module), 110 pyMor.grids.gmsh (module), 110 pyMor.grids.interfaces (module), 112 pyMor.grids.oned (module), 118 pyMor.grids.rect (module), 120 pyMor.grids.referenceelements (module), 122 pyMor.grids.subgrid (module), 127 pyMor.grids.tria (module), 128 pyMor.grids.unstructured (module), 130 pyMor.gui (module), 132 pyMor.gui.fenics (module), 132 pyMor.gui.qt (module), 133 pyMor.gui.matplotlib (module), 133 pyMor.gui.qt (module), 133 pyMor.operators (module), 137 pyMor.operators.basic (module), 137 pyMor.operators.block (module), 144 pyMor.operators.cg (module), 148 pyMor.operators.constructions (module), 158 pyMor.operators.ef (module), 195 pyMor.operators.fenics (module), 199 pyMor.operators.numpy (module), 199 pyMor.operators.interfaces (module), 209 pyMor.operators.mpi (module), 216 pyMor.operators.numpy (module), 222 pyMor.parallel (module), 236 pyMor.parallel.basic (module), 236 pyMor.parallel.default (module), 238 pyMor.parallel.dummy (module), 239 pyMor.parallel.interfaces (module), 241 pyMor.parallel.ipython (module), 245 pyMor.parallel.manager (module), 246 pyMor.parallel.mpi (module), 247 pyMor.parallel.mpi (module), 247 pyMor.parameters (module), 247 pyMor.parameters.base (module), 247 pyMor.parameters.functionals (module), 252 pyMor.parameters.interfaces (module), 255 pyMor.parameters.spaces (module), 256 pyMor.playground (module), 258 pyMor.playground.core (module), 258 pyMor.playground.core.network_cache (module), 258 pyMor.playground.discretizers (module), 258 pyMor.playground.discretizers.numpylistvectorarray (module), 258 pyMor.playground.functions (module), 259 pyMor.playground.functions.expression_function (module), 259 pyMor.playground.ops (module), 259 pyMor.playground.ops.numpy (module), 259 pyMor.playground.progressbar (module), 267 pyMor.playground.vectorarrays (module), 262 pyMor.playground.vectorarrays.disk (module), 262 pyMor.playground.vectorarrays.mpi (module), 267 pyMor.reductors (module), 267
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce_coercive_simple()</td>
<td>pymor.reductors.coercive</td>
<td>271</td>
</tr>
<tr>
<td>reduce_generic_rb()</td>
<td>pymor.reductors.basic</td>
<td>268</td>
</tr>
<tr>
<td>reduce_greedy()</td>
<td>pymor.reductormos.parabolic_mor</td>
<td>328</td>
</tr>
<tr>
<td>reduce_greedy()</td>
<td>pymor.reductormos.thermalblock</td>
<td>329</td>
</tr>
<tr>
<td>reduce_greedy()</td>
<td>pymor.reductormos.thermalblock_simple</td>
<td>332</td>
</tr>
<tr>
<td>reduce_implicit_euler_residual()</td>
<td>pymor.reductors.residual</td>
<td>277</td>
</tr>
<tr>
<td>reduce_naive()</td>
<td>pymor.reductormos.thermalblock</td>
<td>329</td>
</tr>
<tr>
<td>reduce_naive()</td>
<td>pymor.reductormos.thermalblock_simple</td>
<td>333</td>
</tr>
<tr>
<td>reduce_parabolic()</td>
<td>pymor.reductors.parabolic</td>
<td>272</td>
</tr>
<tr>
<td>reduce_pod()</td>
<td>pymor.reductormos.parabolic_mor</td>
<td>328</td>
</tr>
<tr>
<td>reduce_pod()</td>
<td>pymor.reductormos.thermalblock</td>
<td>329</td>
</tr>
<tr>
<td>reduce_pod()</td>
<td>pymor.reductormos.thermalblock_simple</td>
<td>333</td>
</tr>
<tr>
<td>reduce_residual()</td>
<td>pymor.reductors.residual</td>
<td>278</td>
</tr>
<tr>
<td>reduce_stationary_affine_linear()</td>
<td>pymor.reductors.stationary</td>
<td>271</td>
</tr>
<tr>
<td>reduce_stationary_coercive()</td>
<td>pymor.reductors.stationary</td>
<td>279</td>
</tr>
<tr>
<td>reduce_to_subbasis()</td>
<td>pymor.reductors.basic</td>
<td>269</td>
</tr>
<tr>
<td>ReduceCoerciveEstimator</td>
<td>pymor.reductors.coercive</td>
<td>269</td>
</tr>
<tr>
<td>ReduceCoerciveSimpleEstimator</td>
<td>pymor.reductors.coercive</td>
<td>270</td>
</tr>
<tr>
<td>ReducedSim</td>
<td>pymor.reductormos.thermalblock_gui</td>
<td>331</td>
</tr>
<tr>
<td>ReduceParabolicEstimator</td>
<td>pymor.reductors.parabolic</td>
<td>272</td>
</tr>
<tr>
<td>reduction_error_analysis()</td>
<td>pymor.algorithms.error</td>
<td>31</td>
</tr>
<tr>
<td>reference_element()</td>
<td>pymor.grids.interfaces.AffineGridInterface</td>
<td>113</td>
</tr>
<tr>
<td>ReferenceElementDefaultImplementations</td>
<td>pymor.grids.defaultimpl</td>
<td>110</td>
</tr>
<tr>
<td>ReferenceElementInterface</td>
<td>pymor.grids.interfaces</td>
<td>117</td>
</tr>
<tr>
<td>register_type()</td>
<td>pymor.domaindescriptions.boundarytypes.BoundaryType</td>
<td>96</td>
</tr>
<tr>
<td>RegisteredSubtype</td>
<td>pymor.vectorarrays.mpi</td>
<td>317</td>
</tr>
<tr>
<td>RemoteId</td>
<td>pymor.parallel.ipython</td>
<td>245</td>
</tr>
<tr>
<td>RemoteObject</td>
<td>pymor.parallel.basic</td>
<td>236</td>
</tr>
<tr>
<td>RemoteObjectManagerInterface</td>
<td>pymor.parallel.interfaces</td>
<td>241</td>
</tr>
<tr>
<td>RemoteObjectManager</td>
<td>pymor.parallel.manager</td>
<td>246</td>
</tr>
<tr>
<td>remove()</td>
<td>pymor.parallel.interfaces.RemoteObjectInterface</td>
<td>242</td>
</tr>
<tr>
<td>remove()</td>
<td>pymor.playground.vectorarrays.disk.DiskVectorArray</td>
<td>266</td>
</tr>
<tr>
<td>remove()</td>
<td>pymor.vectorarrays.block.BlockVectorArray</td>
<td>291</td>
</tr>
<tr>
<td>remove()</td>
<td>pymor.vectorarrays.interfaces.VectorArrayInterface</td>
<td>298</td>
</tr>
<tr>
<td>remove()</td>
<td>pymor.vectorarrays.list.ListVectorArray</td>
<td>304</td>
</tr>
<tr>
<td>remove()</td>
<td>pymor.vectorarrays.mpi.MPIVectorArray</td>
<td>311</td>
</tr>
<tr>
<td>remove()</td>
<td>pymor.vectorarrays.numpy.NumpyVectorArray</td>
<td>321</td>
</tr>
<tr>
<td>remove_object()</td>
<td>pymor.tools.mpi</td>
<td>284</td>
</tr>
<tr>
<td>remove_objects()</td>
<td>pymor.parallel.manager.RemoteObjectManager</td>
<td>247</td>
</tr>
<tr>
<td>removed</td>
<td>pymor.parallel.interfaces.RemoteObjectInterface</td>
<td>242</td>
</tr>
<tr>
<td>replace()</td>
<td>pymor.playground.vectorarrays.disk.DiskVectorArray</td>
<td>266</td>
</tr>
<tr>
<td>replace()</td>
<td>pymor.vectorarrays.block.BlockVectorArray</td>
<td>291</td>
</tr>
<tr>
<td>replace()</td>
<td>pymor.vectorarrays.interfaces.VectorArrayInterface</td>
<td>298</td>
</tr>
<tr>
<td>replace()</td>
<td>pymor.vectorarrays.list.ListVectorArray</td>
<td>304</td>
</tr>
<tr>
<td>replace()</td>
<td>pymor.vectorarrays.mpi.MPIVectorArray</td>
<td>311</td>
</tr>
<tr>
<td>replace()</td>
<td>pymor.vectorarrays.numpy.NumpyVectorArray</td>
<td>321</td>
</tr>
<tr>
<td>RestrictedOperator</td>
<td>pymor.vectorarrays.numpy</td>
<td>276</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.constructions.ComponentProjection</td>
<td>163</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.constructions.Concatenation</td>
<td>166</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.constructions.ConstantOperator</td>
<td>168</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.constructions.FixedParameter</td>
<td>172</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.constructions.IdentityOperator</td>
<td>176</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.constructions.VectorArrayOperator</td>
<td>188</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.fv.NonlinearAdvectionOperator</td>
<td>206</td>
</tr>
<tr>
<td>restricted()</td>
<td>pymor.operators.interfaces.OperatorInterface</td>
<td>215</td>
</tr>
</tbody>
</table>
restricted() (pymor.operators.mpi.MPIOperator method), 221
restricted_to_subbasis() (pymor.reductors.basic.GenericRBReconstructor method), 268
rhs (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 46
rhs (pymor.analyticalproblems.elliptic.EllipticProblem attribute), 53
rhs (pymor.discretizations.basic.InstationaryDiscretization attribute), 79
rhs (pymor.discretizations.basic.StationaryDiscretization attribute), 80
right (pymor.domaindescriptions.basic.LineDomain attribute), 93
right (pymor.domaindescriptions.basic.RectDomain attribute), 94
robin_data (pymor.analyticalproblems.elliptic.EllipticProblem attribute), 53
RobinBoundaryOperator (class in pymor.operators.cg), 157
run_code() (in module pymor.tools.mpi), 284
sample_randomly() (pymor.parameters.spaces.CubicParameterSpace method), 257
sample_uniformly() (pymor.parameters.spaces.CubicParameterSpace method), 257
scal() (pymor.playground.vectorarrays.disk.DiskVectorArray method), 266
scal() (pymor.vectorarrays.block.BlockVectorArray method), 291
scal() (pymor.vectorarrays.interfaces.VectorArrayInterface method), 298
scal() (pymor.vectorarrays.list.ListVectorArray method), 305
scal() (pymor.vectorarrays.mpi.MPIVectorArray method), 311
scal() (pymor.vectorarrays.numpy.NumpyVectorArray method), 321
scatter_array() (pymor.parallel.dummy.DummyPool method), 240
scatter_array() (pymor.parallel.interfaces.WorkerPoolInterface method), 244
scatter_list() (pymor.parallel.dummy.DummyPool method), 241
scatter_list() (pymor.parallel.interfaces.WorkerPoolInterface method), 244
SelectionOperator (class in pymor.operators.constructions), 182
set() (pymor.core.cache.CacheRegion method), 60
set.defaults() (in module pymor.core.defaults), 65
set_log_format() (in module pymor.core.logger), 74
set_log_levels() (in module pymor.core.logger), 74
shape_range (pymor.functions.interfaces.FunctionInterface method), 106
sid (pymor.core.interfaces.ImmutableInterface attribute), 71
sid (pymor.parameters.base.Parameter attribute), 249
sid_ignore (pymor.core.interfaces.ImmutableInterface attribute), 71
SIDGenerationError (class in pymor.core.exceptions), 68
SimBase (class in pymordemos.thermalblock_gui), 332
SimPanel (class in pymordemos.thermalblock_gui), 332
SimplifiedEngquistOsherFlux (class in pymor.operators.fv), 207
size() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 116
size() (pymor.grids.interfaces.ReferenceTopologicalGridInterface method), 117
size() (pymor.grids.oned.OnedGrid method), 119
size() (pymor.grids.rect.RectGrid method), 121
size() (pymor.grids.referenceelements.Line method), 123
size() (pymor.grids.referenceelements.Point method), 124
size() (pymor.grids.referenceelements.Square method), 125
size() (pymor.grids.referenceelements.Triangle method), 126
size() (pymor.grids.subgrid.SubGrid method), 128
size() (pymor.grids.tria.TriaGrid method), 129
size() (pymor.grids.unstructured.UnstructuredTriangleGrid method), 131
solution_space (pymor.discretizations.interfaces.DiscretizationInterface attribute), 81
solve() (pymor.algorithms.timestepping.ExplicitEulerTimeStepper method), 41
solve() (pymor.algorithms.timestepping.ImplicitEulerTimeStepper method), 43
solve() (pymor.algorithms.timestepping.TimeStepperInterface method), 44
solve() (pymor.discretizations.interfaces.DiscretizationInterface method), 82
solver_options (pymor.operators.interfaces.OperatorInterface attribute), 209
source (pymor.operators.interfaces.OperatorInterface attribute), 209
space (pymor.vectorarrays.interfaces.VectorArrayInterface attribute), 293
sparse (pymor.operators.numpy.NumpyMatrixBasedOperator attribute), 224
sparse_options() (in module pymor.operators.numpy), 232
SQLiteRegion (class in pymor.core.cache), 61
Square (class in pymor.grids.referenceelements), 124
Index

StationaryDiscretization (class in pymor.discretizations.basic), 79
stop_gui_processes() (in module pymor.gui.qt), 135
strip_parameter() (pymor.parameters.base.Parametric method), 252
structured_to_global() (pymor.grids.RectGrid method), 121
sub_reference_element() (pymor.grids.interfaces.ReferenceElementInterface method), 118
sub_reference_element() (pymor.grids.referenceelements.Line method), 123
sub_reference_element() (pymor.grids.referenceelements.Point method), 124
sub_reference_element() (pymor.grids.referenceelements.Square method), 125
sub_reference_element() (pymor.grids.referenceelements.Triangle method), 126
SubGrid (class in pymor.grids.subgrid), 127
SubGridBoundaryInfo (class in pymor.grids.boundaryinfos), 108
subtype (pymor.vectorarrays.interfaces.VectorArrayInterface attribute), 293
sup_norm() (pymor.vectorarrays.block.BlockVectorArray method), 292
sup_norm() (pymor.vectorarrays.interfaces.VectorArrayInterface method), 298
superentities() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 116
superentity_indices() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 117

SubbasisReconstructor (class in pymor.reductors.basic), 268
subentities() (pymor.grids.interfaces.AffineGridInterface method), 113
subentities() (pymor.grids.interfaces.ConformalTopologicalGridInterface method), 116
subentities() (pymor.grids.interfaces.ReferenceElementInterface method), 118
subentities() (pymor.grids.oned.OnedGrid method), 119
subentities() (pymor.grids.RectGrid method), 121
subentities() (pymor.grids.referenceelements.Line method), 123
subentities() (pymor.grids.referenceelements.Point method), 124
subentities() (pymor.grids.referenceelements.Square method), 125
subentities() (pymor.grids.referenceelements.Triangle method), 126

T (pymor.analyticalproblems.advection.InstationaryAdvectionProblem attribute), 46
T (pymor.analyticalproblems.parabolic.ParabolicProblem attribute), 56
T (pymor.discretizations.basic.InstationaryDiscretization attribute), 78
thermalblock_demo() (in module pymor.domains.thermalblock), 329
thermalblock_demo() (in module pymor.domains.thermalblock_adaptive), 330
ThermalBlockDiffusionFunction (class in pymor.analyticalproblems.thermalblock), 56
ThermalBlockProblem (class in pymor.analyticalproblems.thermalblock), 56
time_stepper (pymor.discretizations.basic.InstationaryDiscretization attribute), 79
TorusDomain (class in pymor.domaindescriptions.basic), 94
top (pymor.domaindescriptions.basic.CylindricalDomain attribute), 92
top (pymor.domaindescriptions.basic.RectDomain attribute), 94
TriaGrid (class in pymor.grids.tria), 128
Triangle (class in pymor.grids.referenceelements), 125
<table>
<thead>
<tr>
<th>Page</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>trivial_basis_extension()</td>
<td>(in module pymor.algorithms.basisextension)</td>
</tr>
<tr>
<td>299</td>
<td>type</td>
<td>(pymor.vectorarrays.interfaces.VectorSpace attribute)</td>
</tr>
<tr>
<td>95</td>
<td>types</td>
<td>(pymor.domaindescriptions.boundarytypes.BoundaryType attribute)</td>
</tr>
<tr>
<td>72</td>
<td>UberMeta</td>
<td>(class in pymor.core.interfaces)</td>
</tr>
<tr>
<td>72</td>
<td>UID</td>
<td>(class in pymor.core.interfaces)</td>
</tr>
<tr>
<td>70</td>
<td>uid</td>
<td>(pymor.core.interfaces.BasicInterface attribute)</td>
</tr>
<tr>
<td>115</td>
<td>unique_boundary_type_mask()</td>
<td>(pymor.grids.interfaces.BoundaryInfoInterface method)</td>
</tr>
<tr>
<td>114</td>
<td>unit_outer_normals()</td>
<td>(pymor.grids.interfaces.AffineGridInterface method)</td>
</tr>
<tr>
<td>118</td>
<td>unit_outer_normals()</td>
<td>(pymor.grids.interfaces.ReferenceElementInterface method)</td>
</tr>
<tr>
<td>123</td>
<td>unit_outer_normals()</td>
<td>(pymor.grids.referenceelements.Line method)</td>
</tr>
<tr>
<td>124</td>
<td>unit_outer_normals()</td>
<td>(pymor.grids.referenceelements.Point method)</td>
</tr>
<tr>
<td>125</td>
<td>unit_outer_normals()</td>
<td>(pymor.grids.referenceelements.Square method)</td>
</tr>
<tr>
<td>126</td>
<td>unit_outer_normals()</td>
<td>(pymor.grids.referenceelements.Triangle method)</td>
</tr>
<tr>
<td>70</td>
<td>unlock()</td>
<td>(pymor.core.interfaces.BasicInterface method)</td>
</tr>
<tr>
<td>71</td>
<td>unlock()</td>
<td>(pymor.core.interfaces.ImmutableInterface method)</td>
</tr>
<tr>
<td>130</td>
<td>UnstructuredTriangleGrid</td>
<td>(class in pymor.grids.unstructured)</td>
</tr>
<tr>
<td>27</td>
<td>visualize()</td>
<td>(pymor.discretizations.basic.DiscretizationBase method)</td>
</tr>
<tr>
<td>82</td>
<td>visualize()</td>
<td>(pymor.discretizations.interfaces.DiscretizationInterface method)</td>
</tr>
<tr>
<td>119</td>
<td>visualize()</td>
<td>(pymor.grids.oned.OnedGrid method)</td>
</tr>
<tr>
<td>122</td>
<td>visualize()</td>
<td>(pymor.grids.rect.RectGrid method)</td>
</tr>
<tr>
<td>130</td>
<td>visualize()</td>
<td>(pymor.grids.tria.TriaGrid method)</td>
</tr>
<tr>
<td>131</td>
<td>visualize()</td>
<td>(pymor.grids.unstructured.UnstructuredTriangleGrid method)</td>
</tr>
<tr>
<td>134</td>
<td>visualize()</td>
<td>(pymor.gui.qt.Matplotlib1DVisualizer method)</td>
</tr>
<tr>
<td>135</td>
<td>visualize()</td>
<td>(pymor.gui.qt.PatchVisualizer method)</td>
</tr>
<tr>
<td>136</td>
<td>visualize()</td>
<td>(pymor.gui.qt.PatchVisualizer method)</td>
</tr>
<tr>
<td>117</td>
<td>volume</td>
<td>(pymor.grids.interfaces.ReferenceElementInterface attribute)</td>
</tr>
<tr>
<td>114</td>
<td>volumes()</td>
<td>(pymor.grids.interfaces.AffineGridInterface method)</td>
</tr>
<tr>
<td>114</td>
<td>volumes_inverse()</td>
<td>(pymor.grids.interfaces.AffineGridInterface method)</td>
</tr>
<tr>
<td>147</td>
<td>vstack()</td>
<td>(pymor.operators.block.BlockOperator class method)</td>
</tr>
<tr>
<td>71</td>
<td>with()</td>
<td>(pymor.core.interfaces.ImmutableInterface method)</td>
</tr>
<tr>
<td>79</td>
<td>with()</td>
<td>(pymor.discretizations.basic.InstationaryDiscretization method)</td>
</tr>
<tr>
<td>80</td>
<td>with()</td>
<td>(pymor.discretizations.basic.StationaryDiscretization method)</td>
</tr>
<tr>
<td>206</td>
<td>with()</td>
<td>(pymor.operators.fv.NonlinearAdvectionOperator method)</td>
</tr>
<tr>
<td>71</td>
<td>with_arguments</td>
<td>(pymor.core.interfaces.ImmutableInterface attribute)</td>
</tr>
<tr>
<td>236</td>
<td>WorkerPoolBase</td>
<td>(class in pymor.parallel.basic)</td>
</tr>
<tr>
<td>238</td>
<td>WorkerPoolDefaultImplementations</td>
<td>(class in pymor.parallel.basic)</td>
</tr>
<tr>
<td>242</td>
<td>write_defaults_to_file()</td>
<td>(in module pymor.core.defaults)</td>
</tr>
<tr>
<td>287</td>
<td>write_vtk()</td>
<td>(in module pymor.tools.vtkio)</td>
</tr>
<tr>
<td>191</td>
<td>ZeroOperator</td>
<td>(class in pymor.operators.constructions)</td>
</tr>
<tr>
<td>191</td>
<td>zeros()</td>
<td>(pymor.vectorarrays.interfaces.VectorArrayInterface method)</td>
</tr>
<tr>
<td>300</td>
<td>zeros()</td>
<td>(pymor.vectorarrays.interfaces.VectorSpace method)</td>
</tr>
</tbody>
</table>