Westfälische
Wilhelms-Universität
Münster

# dune-pyMor

## Model Order Reduction with Python and Dune

living.knowledge
WWU Münster

F. Albrecht, R. Milk,
M. Ohlberger, S. Rave

DUNE User Meeting
September 25, 2013

# pyMor

- Software package for Model Order Reduction, in particular Reduced Basis (RB) Method.
- Completely written in Python.

# pyMor

- Software package for Model Order Reduction, in particular Reduced Basis (RB) Method.
- Completely written in Python.
- Joint with Felix Albrecht and Rene Milk.
- Ca. 10k lines of code.
- BSD-License.
- `https://github.com/pyMor`



Stephan Rave (stephan.rave@wwu.de)

# pyMor

- Software package for Model Order Reduction, in particular Reduced Basis (RB) Method.
- Completely written in Python.
- Joint with Felix Albrecht and Rene Milk.
- Ca. 10k lines of code.
- BSD-License.
- `https://github.com/pyMor`
- 0.2 release coming soon!

# Why Python?

# Why Python?

- Agile software development.

# Why Python?

- Agile software development.
- Model reduction algorithms not performance-critical (in contrast to high dimensional solvers).
- `NumPy` delivers `MATLAB`™-like performance for matrix operations.

# Why Python?

- Agile software development.
- Model reduction algorithms not performance-critical (in contrast to high dimensional solvers).
- `NumPy` delivers `MATLAB`™-like performance for matrix operations.
- Plays nicely with other programming languages.
- 34.653 python packages on PyPI (Python Package Index).
- Open source.

# Why Python?

- Agile software development.
- Model reduction algorithms not performance-critical (in contrast to high dimensional solvers).
- `NumPy` delivers `MATLAB`™-like performance for matrix operations.
- Plays nicely with other programming languages.
- 34.653 python packages on PyPI (Python Package Index).
- Open source.
- It’s a great language!

# Outline

1. The Reduced Basis Method.
2. Design of pyMor and dune-pyMor.
3. dune-pyMor in action.

# Reduced Basis Method
## in a Nutshell

### Discrete Problem

For given parameter $\mu \in \mathcal{P}$, find $u_{\mu,h} \in V_h$ satisfying

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,h}, v_h) = F(v_h) \qquad \forall v_h \in V_h. \tag{$*$}$$

# Reduced Basis Method

## in a Nutshell

### Discrete Problem

For given parameter $\mu \in \mathcal{P}$, find $u_{\mu,h} \in V_h$ satisfying

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,h}, v_h) = F(v_h) \qquad \forall v_h \in V_h. \qquad (*)$$

- Assume that solving $(*)$ is very expensive.

# Reduced Basis Method

## in a Nutshell

### Discrete Problem

For given parameter $\mu \in \mathcal{P}$, find $u_{\mu,h} \in V_h$ satisfying

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,h}, v_h) = F(v_h) \qquad \forall v_h \in V_h. \qquad (*)$$

- Assume that solving $(*)$ is very expensive.
- Need to solve for many $\mu \in \mathcal{P}$.

# Reduced Basis Method

## in a Nutshell

### Discrete Problem

For given parameter $\mu \in \mathcal{P}$, find $u_{\mu,h} \in V_h$ satisfying

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,h}, v_h) = F(v_h) \qquad \forall v_h \in V_h. \tag{$*$}$$

- Assume that solving $(*)$ is very expensive.
- Need to solve for many $\mu \in \mathcal{P}$.
- Offline-Phase: Use some fancy algorithm to
  - compute snapshots $\mathcal{S} := \{u_{\mu_s,h} \,|\, s = 1, \ldots, S\}$
  - determine $V_N \subseteq \mathrm{span}(\mathcal{S})$ with $N = \dim V_N \ll \dim V_h$.

# Reduced Basis Method

## in a Nutshell

### Discrete Problem

For given parameter $\mu \in \mathcal{P}$, find $u_{\mu,h} \in V_h$ satisfying

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,h}, v_h) = F(v_h) \qquad \forall v_h \in V_h. \qquad (*)$$

- Assume that solving $(*)$ is very expensive.
- Need to solve for many $\mu \in \mathcal{P}$.
- Offline-Phase: Use some fancy algorithm to
  - compute snapshots $\mathcal{S} := \{u_{\mu_s,h} \,|\, s = 1, \dots, S\}$
  - determine $V_N \subseteq \operatorname{span}(\mathcal{S})$ with $N = \dim V_N \ll \dim V_h$.
- Online-Phase: For new $\mu$, solve $(*)$ restricted to $V_N$.

# Reduced Basis Method
## in a Nutshell

### Reduced Problem

For given parameter $\mu \in \mathcal{P}$, find $u_{\mu,N} \in V_N$ satisfying

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,N}, v_N) = F(v_N) \qquad \forall v_N \in V_N. \qquad (**)$$

- Assume that solving $(*)$ is very expensive.
- Need to solve for many $\mu \in \mathcal{P}$.
- Offline-Phase: Use some fancy algorithm to
  - compute snapshots $\mathcal{S} := \{u_{\mu_s,h} \,|\, s = 1, \dots, S\}$
  - determine $V_N \subseteq \operatorname{span}(\mathcal{S})$ with $N = \dim V_N \ll \dim V_h$.
- Online-Phase: For new $\mu$, solve $(*)$ restricted to $V_N$.

# Reduced Basis Method

in a Nutshell

## Reduced Problem

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,N}, v_N) = F(v_N) \qquad \forall v_N \in V_N. \qquad (**)$$

- Let $b_1, \dots, b_N$ be a basis of $V_N$ and define

$$\underline{B}_k = [B_k(b_j, b_i)]_{i,j=1}^{N} \qquad \underline{F} = [F(b_i)]_{i=1}^{N}.$$

# Reduced Basis Method

## in a Nutshell

### Reduced Problem

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,N}, v_N) = F(v_N) \qquad \forall v_N \in V_N. \qquad (**)$$

- Let $b_1, \ldots, b_N$ be a basis of $V_N$ and define

$$\underline{B}_k = [B_k(b_j, b_i)]_{i,j=1}^{N} \qquad \underline{F} = [F(b_i)]_{i=1}^{N}.$$

- In coordinates, $(**)$ becomes

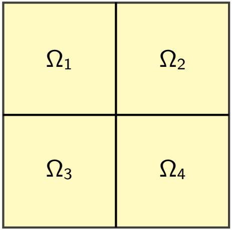
$$\sum_{k=1}^{K} \theta(\mu) \underline{B}_k \cdot \underline{u}_{\mu,N} = \underline{F}$$

with reconstruction equation

$$u_{\mu,N} = \sum_{i=1}^{N} b_i \cdot \underline{u}_{\mu,N,i}.$$

# Reduced Basis Method

## Example




$$\Omega = \bigcup_{k=1}^{4} \Omega_k, \ \mathcal{P} = [\alpha, 1]^4, \ \alpha > 0$$

$$a_\mu(x) = \sum_{k=1}^{4} \mu_k \cdot \chi_{\Omega_k}(x), \qquad x \in \Omega, \mu \in \mathcal{P}$$

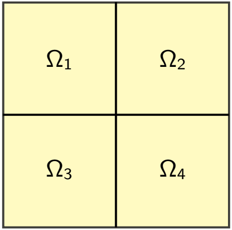
### Thermal-Block Problem

For $f \in L^2(\Omega)$ and $\mu \in \mathcal{P}$, find $u_\mu \in H_0^1(\Omega)$ s.t.

$$-\nabla \cdot (a_\mu \nabla u_\mu) = f$$

# Reduced Basis Method

## Example




$$\Omega = \bigcup_{k=1}^{4} \Omega_k, \ \mathcal{P} = [\alpha, 1]^4, \ \alpha > 0$$

$$a_\mu(x) = \sum_{k=1}^{4} \mu_k \cdot \chi_{\Omega_k}(x), \qquad x \in \Omega, \mu \in \mathcal{P}$$

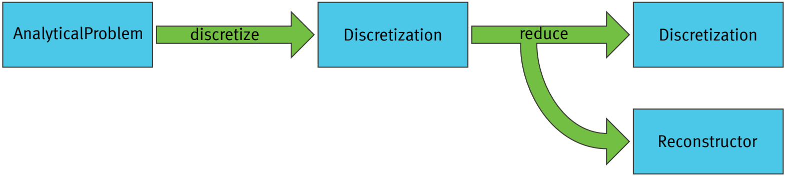
### Thermal-Block Problem

For $f \in L^2(\Omega)$ and $\mu \in \mathcal{P}$, find $u_\mu \in H_0^1(\Omega)$ s.t.

$$\sum_{k=1}^{4} \mu_k \int_{\Omega_k} \nabla u_\mu \cdot \nabla v = \int_\Omega f \cdot v \qquad \forall v \in H_0^1(\Omega)$$

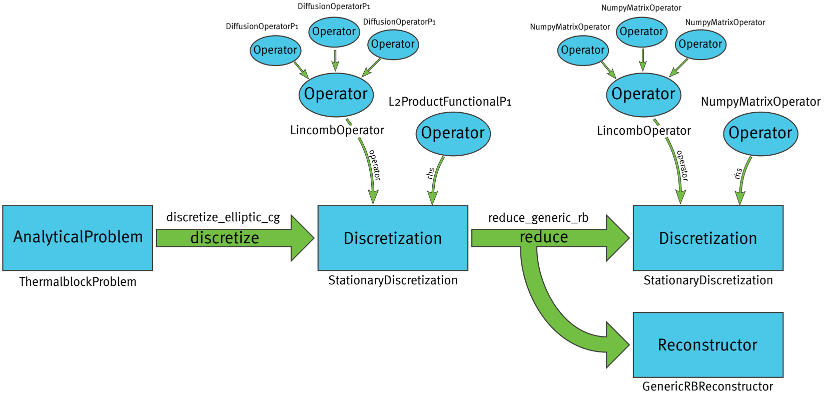
# Architecture of pyMor

## Workflow



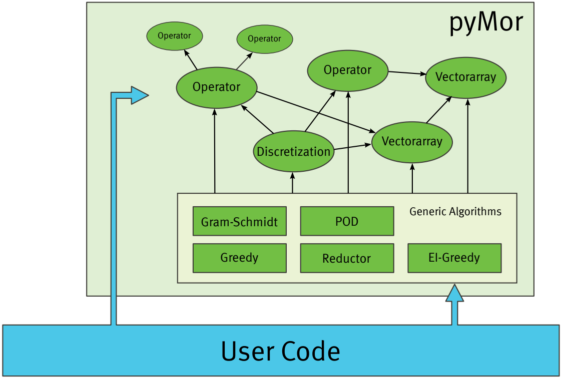
# Architecture of pyMor

## Workflow



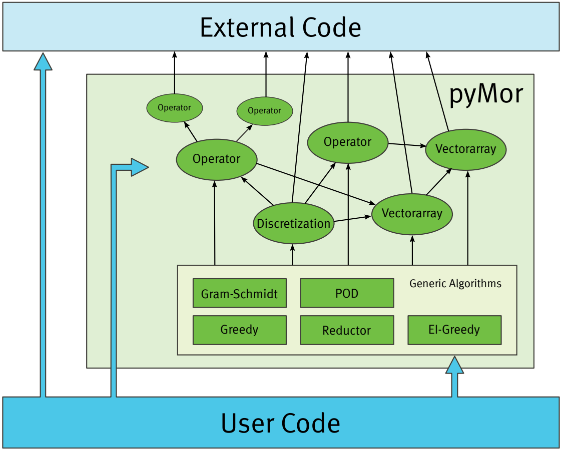
# Architecture of pyMor

## Interfaces



# Architecture of pyMor

## Interfaces



# Accessing External Code from pyMor

## as a Python Extension Module

# Accessing External Code from pyMor

## as a Python Extension Module

- Implement `Operators`, `VectorArrays`, `Discretizations`.

# Accessing External Code from pyMor

## as a Python Extension Module

- Implement `Operators`, `VectorArrays`, `Discretizations`.
- Create Python module (e.g. using `PyBindGen`).

# Accessing External Code from pyMor

## as a Python Extension Module

- Implement `Operators`, `VectorArrays`, `Discretizations`.
- Create Python module (e.g. using `PyBindGen`).
- Write Python wrapper classes implementing the full `pyMor` interfaces.

# DUNE-Bindings with dune-pyMor

# DUNE-Bindings with dune-pyMor

- Implement `Operators`, `Vectors`, `Discretizations` by deriving from the corresponding `dune-pyMor` interface classes.

# DUNE-Bindings with dune-pyMor

- Implement `Operators`, `Vectors`, `Discretizations` by deriving from the corresponding `dune-pyMor` interface classes.
  - `Dune::DynamicMatrix` and `Eigen` based operators already provided.
  - Bindings for LA backend of `PDELab` in development.

# DUNE-Bindings with dune-pyMor

- Implement `Operators`, `Vectors`, `Discretizations` by deriving from the corresponding `dune-pyMor` interface classes.
  - `Dune::DynamicMatrix` and `Eigen` based operators already provided.
  - Bindings for LA backend of `PDELab` in development.
- Utilize provided helper methods (`inject_operator`, ...) and build system to semi-automatically create Python module.

# DUNE-Bindings with dune-pyMor

- Implement `Operators`, `Vectors`, `Discretizations` by deriving from the corresponding `dune-pyMor` interface classes.
  - `Dune::DynamicMatrix` and `Eigen` based operators already provided.
  - Bindings for LA backend of `PDELab` in development.
- Utilize provided helper methods (`inject_operator, ...`) and build system to semi-automatically create Python module.
- In pyMor, simply call `dune.pymor.core.wrap_module` to obtain wrappers for all implemented `DUNE` classes.

# Live Demo

# Thank you for your attention!

pyMor – Model Order Reduction with Python
`https://github.com/pyMor`

dune-pyMor demo application
`https://github.com/pyMor/dune-hdd-demos`

# Interfaces
## VectorArrays

```
class VectorArrayInterface(BasicInterface):
    # array creation
    @classmethod
    def empty(cls, dim, reserve=0): pass
    @classmethod
    def zeros(cls, dim, count=1): pass

    # comparing arrays
    def __len__(self): pass
    @property
    def dim(self): pass
    def almost_equal(self, other, ind, o_ind, rtol, atol): pass

    # array manipulation
    def copy(self, ind): pass
    def append(self, other, o_ind, remove_from_other=False): pass
    def remove(self, ind): pass
    def replace(self, other, ind, o_ind, remove_from_other=False): pass
    # ...
```

# Interfaces
## VectorArrays

```python
#class VectorArrayInterface (continued)
    # linear algebra
    def scal(self, alpha, ind): pass
    def axpy(self, alpha, x, ind, x_ind): pass
    def dot(self, other, pairwise, ind, o_ind): pass
    def lincomb(self, coefficients, ind): pass
    def l2_norm(self, ind): pass

    # empirical interpolation
    def components(self, component_indices, ind): pass
    def amax(self, ind): pass
```

# Interfaces

## Operators

```
class OperatorInterface(ImmutableInterface, Parametric, Named):
    dim_source = None
    dim_range = None
    type_source = None
    type_range = None
    linear = False
    invert_options = None

    def apply(self, U, ind, mu): pass
    def apply2(self, V, U, U_ind, V_ind, mu, product, pairwise): pass
    def apply_inverse(self, U, ind, mu, options): pass

    @staticmethod
    def lincomb(operators, coefficients, ...): pass
```

# Interfaces

## RB-Projection in pyMor (more or less)

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,N}, v_N) = F(v_N) \qquad \forall v_N \in V_N.$$

```
def reduce_generic_rb(discretization, RB):
    projected_ops = {k: rb_project_operator(op, RB)
                     for k, op in discretization.operators.items()}
    rd = discretization.with_(operators=projected_operators)
    rc = GenericRBReconstructor(RB)
    return rd, rc

def rb_project_operator(op, RB):
    source_basis = RB
    range_basis = RB if op.dim_range = op.dim_source else None
    return op.projected(source_basis, range_basis)

class LincombOperatorBase(OperatorBase, LincombOperatorInterface):
    def projected(self, source_basis, range_basis):
        proj_ops = [op.projected(source_basis, range_basis)
                    for op in self.operators]
        return proj_ops[0].lincomb(proj_operators, self.coefficients)
```

# Interfaces

## RB-Projection in pyMor (more or less)

$$\sum_{k=1}^{K} \theta_k(\mu) B_k(u_{\mu,N}, v_N) = F(v_N) \qquad \forall v_N \in V_N.$$

```
class OperatorBase(OperatorInterface):
    def projected(self, source_basis, range_basis):
        assert self.linear and not self.parametric
        mat = self.apply2(source_basis, range_basis, pairwise=False)
        return NumpyMatrixOperator(mat)

class GenericRBReconstructor(ImmutableInterface):
    def __init__(self, RB):
        self.RB = RB

    def reconstruct(self, U):
        assert isinstance(U, NumpyVectorArray)
        return self.RB.lincomb(U.data)
```