

pyMOR – GENERIC ALGORITHMS AND INTERFACES FOR MODEL ORDER REDUCTION*

RENÉ MILK[†], STEPHAN RAVE[†], AND FELIX SCHINDLER[†]

Abstract. Reduced basis methods are projection-based model order reduction techniques for reducing the computational complexity of solving parametrized partial differential equation problems. In this work we discuss the design of **pyMOR**, a freely available software library of model order reduction algorithms, in particular reduced basis methods, implemented with the **Python** programming language. As its main design feature, all reduction algorithms in **pyMOR** are implemented generically via operations on well-defined vector array, operator, and discretization interface classes. This allows for an easy integration with existing open-source high-performance partial differential equation solvers without adding any model reduction specific code to these solvers. Besides an in-depth discussion of **pyMOR**'s design philosophy and architecture, we present several benchmark results and numerical examples showing the feasibility of our approach.

Key words. model order reduction, reduced basis method, empirical interpolation, scientific computing, software, Python

AMS subject classifications. 35-04, 35J20, 35L03, 65-04, 65N30, 65Y05, 68N01

DOI. 10.1137/15M1026614

1. Introduction. Over the past years, model order reduction methods have become an important part of many numerical simulation workflows for handling large-scale application problems. Reduced basis (RB) methods are a popular family of such reduction techniques, applicable to parametrized high-dimensional models described by partial differential equations (PDEs). The main ingredient of RB methods is a Galerkin projection of the differential equation onto a problem-adapted reduced subspace generated from solution snapshots of a high-dimensional approximation of the problem for certain well-chosen sampling parameters. While the high-dimensional approximation using standard discretization techniques (such as finite element methods) often yields discrete function spaces with millions of degrees of freedoms, the reduced spaces generated by RB methods typically are of order 100 or smaller, while still retaining the same approximation quality for the problem at hand as the high-dimensional space. In practice, model order reduction by RB approximation can lead to speedups of up to several orders of magnitude. By now, a large body of literature has emerged which theoretically proves and practically demonstrates the applicability of the RB approach to a large variety of application problems (see, e.g., the recent monographs [18, 30], the tutorial [15], and the references therein).

Despite the popularity of RB methods, few software implementations have been discussed in the literature so far. We are only aware of publications discussing **rbMIT** [28], RB modules for **libMesh** [21], and **feel++** [10], as well as the combination of **RBmatlab** with **dune-rb** [12]. However, these approaches are either too simplistic to be applied to large-scale problems (**rbMIT**) or offer limited code reusability by being

*Received by the editors June 17, 2015; accepted for publication (in revised form) April 5, 2016; published electronically October 27, 2016. This work has been supported by the German Federal Ministry of Education and Research (BMBF) under contract 05M13PMA and by the German Research Foundation (DFG) within the SPP 1648 “Software for Exascale Computing” program.

<http://www.siam.org/journals/sisc/38-5/M102661.html>

[†]Institute for Computational and Applied Mathematics, University of Münster, Einsteinstrasse 62, 48149 Münster, Germany (rene.milk@uni-muenster.de, stephan.rave@uni-muenster.de, felix.schindler@uni-muenster.de).

tied to a specific PDE solver ecosystem. While `RBmatlab` defines interfaces for the integration with `dune-rb` which would allow us to reuse its algorithms in conjunction with other PDE solvers implementing the same interfaces, code reusability is still limited by the fact that `RBmatlab` requires all parts of the reduction algorithm involving high-dimensional data to be implemented by the PDE solver.

In this paper we discuss the design of `pyMOR`, an open-source, `Python`-based model reduction library which is being developed as part of the MULTIBAT research project (cf. [25] for a brief overview on `pyMOR` in the context of MULTIBAT). Since one of the goals of said project is to use RB techniques for the reduction of microscale battery models implemented independently by another group participating in the project, an easy integration with third-party PDE solvers is a central design goal for `pyMOR`.

To allow easy integration with external solvers, `pyMOR` is built around a small set of interface classes for interaction with the solver. Unlike the approach taken by `RBmatlab`, `pyMOR`'s interface classes are designed for lower level communication by directly representing the high-dimensional vector and operator objects inside the solver. This allows us to implement model order reduction schemes completely within `pyMOR` as generic algorithms operating on these interface classes. A new external solver is integrated simply by making the solver's data structures available via a public interface to which `pyMOR` can connect to. No model reduction specific code has to be added to the solver. `pyMOR`'s interface design not only facilitates collaboration between researchers by decoupling PDE solver and RB algorithm development, it also fosters the evaluation and adoption of new RB techniques, since algorithms implemented with `pyMOR` can be tested more easily with problems which have not been considered by the original author. By now, `pyMOR` was used successfully in [7, 26, 8, 24].

A design approach similar to `pyMOR` has been taken by the `modred` [5] package, a software library implementing modal decomposition algorithms which operate on generic vector objects provided by an external source. Due to the algorithmic requirements of RB methods, however, our approach goes further by also allowing operations on the solver's system matrices or (nonlinear) operators, resulting in a deeper integration between the two software components.

Recently, `RBniCS` was introduced [1], a `Python`-based RB library built on top of the `FEniCS` [22] PDE solver library. Similar to `pyMOR`, `RBniCS` allows easy development of RB applications in `Python` while leveraging `FEniCS` as a high-performance solver. However, in contrast to `pyMOR`, `RBniCS` is tied to the `FEniCS` ecosystem and cannot be integrated with other PDE solvers. `redbKIT`, which has been developed as companion software to the recent monograph [30], follows a design similar to `rbMIT`. Apart from RB libraries, RB methods are used nowadays in several specialized simulation softwares such as `NiftySim` [19] or the commercial code `Akselos`.¹ However, these implementations are restricted to their specific application domain.

This paper is organized as follows: Section 2 contains a brief introduction to the RB methodology and provides the mathematical background needed to follow the technical discussions in the subsequent sections. We discuss `pyMOR`'s design from a bird's eye perspective and compare it to other design approaches in section 3. In section 4, we cover `pyMOR`'s interface classes in more detail, discuss important implementational aspects, and give a basic example how external solvers can be integrated with `pyMOR`. Moreover, we discuss the parallelization of `pyMOR`'s reduction algorithms. In section 5, we finally present technical benchmarks and a more advanced numerical example which demonstrate the performance of our software design and its applica-

¹<http://www.akselos.com>

bility to large-scale problems.

2. The reduced basis method. In this section we give a very short introduction to the RB methodology that will hopefully give the reader sufficient background to understand the discussion of our software design and our numerical experiments. We mainly focus on the basic class of linear, coercive, affinely decomposed problems for which the fundamental ideas of the approach can be most clearly described. A few of the many extensions of the methodology are discussed in section 2.3. For a more detailed introduction to RB methods, we refer to [15, 18, 30].

2.1. Linear, coercive, affinely decomposed problems. We consider parametrized linear, strongly elliptic problems in weak form. More abstractly, we search for solutions $u_\mu \in V$ in some Hilbert space V satisfying

$$(2.1) \quad B_\mu(u_\mu, \varphi) = F(\varphi) \quad \forall \varphi \in V.$$

Here, for each parameter μ contained in some parameter space $\mu \in \mathcal{P}$, B_μ is a continuous, coercive bilinear form and F is a continuous linear functional on V . Due to the Lax–Milgram theorem, (2.1) admits a unique solution for every μ . Moreover, we assume that B_μ is *affinely decomposed*, i.e., there are $Q \in \mathbb{N}$ bilinear forms $B_1, \dots, B_Q : V \times V \rightarrow \mathbb{R}$, and mappings $\theta_1, \dots, \theta_Q : \mathcal{P} \rightarrow \mathbb{R}$, such that

$$(2.2) \quad B_\mu = \sum_{q=1}^Q \theta_q(\mu) B_q \quad \forall \mu \in \mathcal{P}.$$

Our goal is now to be able to quickly find approximations u_μ for arbitrary $\mu \in \mathcal{P}$, assuming we are allowed to compute a few selected u_μ during a preceding *offline* phase. The latter is, in practice, achieved by assuming that (2.1) is already the result of an appropriate high-dimensional discretization of the original analytical equation, which, however, can only be solved with large computational effort.

The basic idea of the RB method is to first find appropriate $\mu_1, \dots, \mu_N \in \mathcal{P}$ during the offline phase such that $V_N := \text{span}\{u_{\mu_1}, \dots, u_{\mu_N}\}$ is a good approximation space for the so-called solution manifold $\mathcal{M} := \{u_\mu \mid \mu \in \mathcal{P}\}$. Then, an approximation $u_{N,\mu} \in V_N$ of u_μ is obtained during the *online* phase by Galerkin projection of (2.1), i.e., $u_{N,\mu}$ satisfies

$$(2.3) \quad B_\mu(u_{N,\mu}, \varphi) = F(\varphi) \quad \forall \varphi \in V_N.$$

Again, (2.3) has a unique solution according to the Lax–Milgram theorem. Assuming, moreover, we have bounds

$$(2.4) \quad \alpha_\mu \|\varphi\|^2 \leq B_\mu(\varphi, \varphi) \quad \forall \varphi \in V, \quad \|B_\mu\| \leq \gamma_\mu \quad \forall \mu \in \mathcal{P},$$

Cea’s lemma yields the a priori quasi-best approximation bound

$$(2.5) \quad \|u_\mu - u_{N,\mu}\| \leq \frac{\gamma_\mu}{\alpha_\mu} \inf_{\varphi \in V_N} \|u_\mu - \varphi\|.$$

To solve (2.3) numerically, we choose a basis b_1, \dots, b_N of V_N and let $\underline{u}_{N,\mu} \in \mathbb{R}^N$ be the coefficient vector of $u_{N,\mu}$ with respect to this basis, i.e.,

$$(2.6) \quad u_{N,\mu} = \sum_{n=1}^N \underline{u}_{N,\mu,n} b_n.$$

With $\underline{B}_{N,\mu,i,j} := B_\mu(b_j, b_i)$, $\underline{F}_{N,i} := F(b_i)$, $\underline{u}_{N,\mu}$ is then the solution of the linear equation system $\underline{B}_{N,\mu} \cdot \underline{u}_{N,\mu} = \underline{F}_N$. Due to (2.2), we, moreover, have $\underline{B}_{N,\mu} = \sum_{q=1}^Q \theta_q(\mu) \underline{B}_{q,N}$, where $\underline{B}_{q,N}$ denote the matrices of B_q . Thus, (2.3) can be assembled and solved with $\mathcal{O}(QN^2 + N^3)$ operations, independently of $\dim V$. With typical basis sizes of $N \approx 100$, (2.3) can then be solved in less than a millisecond on current hardware.

The coefficients $\underline{u}_{N,\mu}$ usually are not of interest themselves. However, if the reduced basis is available, we can *reconstruct* the reduced solution $u_{N,\mu}$ as an element of the function space V using (2.6). Moreover, one is often only interested in certain quantities of interest which are derived from the solution u_μ . If these quantities are linear functionals of u_μ , we can again precompute their evaluations on b_1, \dots, b_N to arrive at a reduced model which can produce these quantities completely independent of $\dim V$.

Finally, we have the following standard residual-based a posteriori bound for the reduction error available

$$(2.7) \quad \frac{\alpha_\mu}{\gamma_\mu} \cdot \frac{\|\mathcal{R}_\mu(u_{N,\mu})\|_{-1}}{\alpha_\mu} \leq \|u_\mu - u_{N,\mu}\| \leq \frac{\|\mathcal{R}_\mu(u_{N,\mu})\|_{-1}}{\alpha_\mu},$$

where the residual $\mathcal{R}_\mu : V \rightarrow V'$ is defined by $\mathcal{R}_\mu[u_{N,\mu}](\varphi) := F(\varphi) - B_\mu(u_{N,\mu}, \varphi)$. Since we assume V to be finite-dimensional, $\|\mathcal{R}_\mu(u_{N,\mu})\|_{-1}$ can be computed as the norm of the Riesz representative of $\mathcal{R}_\mu(u_{N,\mu})$, given by application of the inverse inner product matrix to the discrete residual vector. Using (2.2), this computation can be reduced to $\mathcal{O}(Q^2N^2)$ online operations. For cases where a stability estimate α_μ is not known a priori, several algorithms have been developed to efficiently compute α_μ during the online phase (cf. [30, section 3.7] and references therein).

2.2. Basis construction. As visible from (2.5), the reduced solution $u_{N,\mu}$ of (2.3) is a quasi-optimal approximation of u_μ within the given reduced space V_N . We are therefore interested in finding spaces V_N minimizing the maximum best approximation error for u_μ over all $\mu \in \mathcal{P}$. A lower bound for what can be achieved is given by the Kolmogorov N -width of the solution manifold \mathcal{M} , defined by

$$(2.8) \quad d_N(\mathcal{M}) := \inf_{\substack{V_N \subseteq V \\ \dim V_N \leq N}} \sup_{u \in \mathcal{M}} \inf_{v \in V_N} \|u - v\|.$$

For the problem class we consider, it is known that the N -widths fall at least subexponentially fast, i.e., there are constants $M, a, \alpha > 0$ such that (s.t.)

$$(2.9) \quad d_N(\mathcal{M}) \leq M e^{-aN^\alpha}.$$

Thus, good approximation spaces V_N do exist. However, it is usually impossible to find spaces for which this lower bound $d_N(\mathcal{M})$ is attained.

Reduced basis methods often employ greedy algorithms in order to obtain nearly best approximation spaces V_N . A standard approach is the error estimator controlled greedy search defined in Algorithm 1. This algorithm assumes the existence of methods **RBSolve** and **ErrEst** for solving the reduced problem and estimating the reduction error given the required reduced model data \mathcal{RD} which is available through the **Reduce** method. In each iteration of the algorithm, the maximum reduction error over a finite training set $\mathcal{S}_{train} \subseteq \mathcal{P}$ of parameters is estimated and a high-dimensional solution snapshot u_{μ^*} for the parameter μ^* maximizing the error estimates is computed. This snapshot is then used to extend the reduced space V_N .

Algorithm 1: Greedy basis generation with error estimator.

Input: training set $\mathcal{S}_{train} \subseteq \mathcal{P}$, tolerance ε , max. dimension N_{max}
Output: reduced spaces V_1, \dots, V_N
 $N \leftarrow 0$, $V_0 \leftarrow \{0\}$, $\mathcal{RD} \leftarrow \text{Reduce}(V_N)$
while $N < N_{max}$ **and** $\max_{\mu \in \mathcal{S}_{train}} \text{ErrEst}(\text{RBSolve}(\mu, \mathcal{RD}), \mu, \mathcal{RD}) > \varepsilon$ **do**
 $N \leftarrow N + 1$
 1 $\mu^* \leftarrow \arg\max_{\mu \in \mathcal{S}_{train}} \text{ErrEst}(\text{RBSolve}(\mu, \mathcal{RD}), \mu, \mathcal{RD})$
 2 $u_{\mu^*} \leftarrow \text{Solve}(\mu^*)$
 3 $V_N \leftarrow \text{span}(V_{N-1} \cup \{u_{\mu^*}\})$, $\mathcal{RD} \leftarrow \text{Reduce}(V_N)$
end

It is important to note that in Algorithm 1 only lines 2 and 3 of the main loop involve high-dimensional operations, and that these operations are performed only once per extension step. This allows us to afford large training sets \mathcal{S}_{train} , densely sampling the parameter space \mathcal{P} .

If the error estimator used in Algorithm 1 is rigorous and effective (2.7), this algorithm is a weak greedy algorithm in the sense of [6]. As a consequence [11], (2.9) carries over to the resulting spaces V_N in the sense that there are $M', a' > 0$ only depending on M, a , and $\sup_{\mu \in \mathcal{P}} \gamma_\mu / \alpha_\mu$, s.t.

$$(2.10) \quad \sup_{\mu \in \mathcal{S}_{train}} \inf_{v \in V_N} \|u_\mu - v\| \leq M' e^{-a' N^\alpha}.$$

In conclusion, if we assume that our training set \mathcal{S}_{train} is chosen dense enough, we will observe a (sub-)exponentially fast decay of the maximum reduction error, leading to large computational savings for many application problems.

2.3. Extensions. The presented basic RB methodology can be extended in many directions, making it suitable for a wide range of application problems. We briefly cover three extensions which are relevant for our numerical examples.

2.3.1. Proper orthogonal decomposition. As an alternative to a greedy construction of V_N (Algorithm 1), we can perform a “proper orthogonal decomposition” (POD) of a given training set of solution snapshots $\{u_{\mu_1}, \dots, u_{\mu_K}\} \subset V$ to obtain a set of orthogonal basis vectors which describe the “main directions” in V by which the snapshot set is characterized. More precisely, let $\Phi : \mathbb{R}^K \rightarrow V$ be the linear mapping sending the k th canonical basis vector of \mathbb{R}^K to u_{μ_k} . Then the k th POD mode of the training set is defined to be the k th left-singular vector of the singular value decomposition of Φ . The spaces $V_{N,pod}$ spanned by these vectors are l^2 -optimal in the sense that

$$(2.11) \quad \sum_{k=1}^K \inf_{v \in V_{N,pod}} \|u_{\mu,k} - v\|^2 = \min_{\substack{V_N \subseteq V \\ \dim V_N \leq N}} \sum_{k=1}^K \inf_{v \in V_N} \|u_{\mu,k} - v\|^2.$$

POD is therefore a good option if this notion of optimality is desired (and not the l^∞ -optimality the greedy approach is designed for). However, POD quickly becomes prohibitively expensive when a large training set is needed to approximate the solution manifold, since the full solution snapshot has to be computed for each parameter from the training set.

2.3.2. Instationary problems. The RB methodology can be extended to instationary problems of the form

$$(2.12) \quad \begin{aligned} \langle \partial_t u_\mu(t), \varphi \rangle + B_\mu(u_\mu(t), \varphi) &= F(\varphi) \quad \forall \varphi \in V, \\ u_\mu(0) &= u_{0,\mu} \end{aligned}$$

in a straightforward way. The most common approach is to perform a Galerkin projection of (2.12) onto a reduced space $V_N \subseteq V$ to arrive at an ordinary differential equation system of dimension N (method of lines).

Note, however, that solution snapshots are whole trajectories now, so it is no longer obvious how to extend V_N after a parameter μ^* has been selected in Algorithm 1. A well-established approach with proven quasi-optimality is the POD-GREEDY algorithm [14], which performs a POD of the orthogonal projection error trajectory $u_{\mu^*}(t) - P_{V_N}(u_{\mu^*}(t))$, of which the first modes are then added to V_N .

2.3.3. Empirical interpolation. If B_μ does not satisfy (2.1) or even is nonlinear in the first variable, the standard RB approach is no longer efficient: while (2.3) is still posed on the low-dimensional space V_N , we cannot solve (2.3) online without resorting to high-dimensional computations which will deteriorate most savings in computation time.

A very generic approach to overcome this issue is empirical operator interpolation: given $x_1, \dots, x_M \in V$, $c_1, \dots, c_M \in V'$ such that the matrix $\underline{L}_M := (c_j(x_i))_{i,j=1}^M$ is nonsingular, we approximate B_μ by the interpolated form $\mathcal{I}_M(B_\mu)$ uniquely defined by $\mathcal{I}_M(B_\mu)(v, \cdot) \in \text{span}\{c_1, \dots, c_M\}$ and

$$(2.13) \quad \mathcal{I}_M(B_\mu)(v, x_i) = B_\mu(v, x_i), \quad i = 1, \dots, M$$

for all $v \in V$. One easily checks that

$$(2.14) \quad \mathcal{I}_M(B_\mu)(v, \cdot) = [c_1, \dots, c_M] \cdot \underline{L}_M^{-1} \cdot [B_\mu(v, x_1), \dots, B_\mu(v, x_M)]^T.$$

In many cases, for instance when the x_i are chosen from a finite element basis and B_μ is the finite element discretization of a partial differential operator, the evaluations $B_\mu(v, x_1), \dots, B_\mu(v, x_M)$ can be computed quickly and independently of $\dim V$ by knowing only $M' < C \cdot M$ degrees of freedom of v for a fixed constant C (e.g., depending on the stencil of the discretization).

Thus, if we approximate B_μ by $\mathcal{I}_M(B_\mu)$ in (2.3) and substitute (2.14), we obtain

$$(2.15) \quad [c_1(\varphi), \dots, c_M(\varphi)] \cdot \underline{L}_M^{-1} \cdot [B_\mu(u_{N,\mu}, x_1), \dots, B_\mu(u_{N,\mu}, x_M)]^T = F(\varphi)$$

for all $\varphi \in V_N$, which can be solved $\dim V$ -independently by precomputing $c_j(b_n)$ and only storing the coefficients of b_n which are required for the evaluation of $B_\mu(v, x_m)$.

The *collateral* interpolation basis c_1, \dots, c_M , along with the interpolation points x_1, \dots, x_M , is obtained during the offline phase from a greedy search (EI-GREEDY, [17]) or POD (DEIM, [9]) on an appropriate training set of evaluations of B_μ .

3. Design of reduced basis software. In this section we discuss the software design issues which arise when implementing RB schemes and cover typical design approaches. We then present the approach taken by pyMOR and compare it to these standard designs.

3.1. Required high-dimensional operations. The RB method is by nature a very generic model reduction framework which can be applied to a wide range of problems. It is an important feature of RB methods that existing high-dimensional discretizations can be used as they are in order to derive a reduced order model: any discretization, no matter if, e.g., continuous finite elements, discontinuous Galerkin, or meshless, can be used as long as the high-dimensional problem is of an appropriate form, such as (2.1) or (2.12).

However, while this is true for the mathematical formulation, the code implementing the high-dimensional discretization nearly always has to be adapted for model order reduction.

While the main purpose of the solver already is the *computation of solution snapshots* u_μ , it is often not known a priori (cf. Algorithm 1) for which parameters μ the solution is required. Thus, either the high-dimensional solver has to stay initialized in memory during the whole offline phase, or the solver has to be reinitialized (mesh generation, matrix assembly, etc.) for each new solution snapshot.

To ensure the numerical stability of the reduced model, the new solution u_μ has to be orthonormalized with respect to (w.r.t.) to the existing reduced basis, e.g., using a stabilized *Gram-Schmidt algorithm*, before it can be added to the reduced basis. For instationary problems (2.12), the solution trajectory has to be *orthogonally projected* onto the current reduced space V_N and a *POD* of the projection errors has to be computed.

The *assembly of reduced system matrices* for (2.3) requires the evaluation of B_μ and F for each (combination of) basis vector(s) of V_N . Moreover, the affine decomposition (2.2) needs to be taken into account for offline/online decomposition. The *assembly of the reduction error estimator* requires the computation of *Riesz representatives* for all residual parts (2.2) and all inner products between them. An alternative approach with higher numerical stability [7] requires the *computation of an orthonormal basis* for the span of the Riesz representatives and the computation of the coefficients of the representatives with respect to the basis. For the reduction of nonaffinely decomposed or nonlinear problems using empirical operator interpolation (section 2.3.3), B_μ has to be evaluated on appropriate u_μ , the collateral basis c_i , and interpolation points x_i (2.14) have to be determined using the EI-GREEDY or DEIM algorithm and the corresponding reduced data has to be computed.

Finally, if access to the reduced solutions as elements of V is desired (e.g., for visualization), the solver needs to be invoked to perform the *reconstruction* (2.6).

3.2. Required low-dimensional operations. In order to *solve the reduced problem* (2.3), the same type of algorithms (linear solvers, time-stepping, Newton algorithm) are required as for the solution of (2.1). However, the involved data structures will be different: dense instead of sparse matrices, no or shared memory parallelization instead of distributed memory parallelization.

For reduced problems involving empirical operator interpolation, the *restricted evaluation* of B_μ according to (2.15), is required.

Lastly, the solution of (2.3) is already required in the offline phase during *greedy basis generation* which, in the case of adaptive variants [16] of Algorithm 1, may require substantial implementation work in itself.

3.3. Classical design approaches. Reduced basis implementations can be mainly categorized by how they realize the interaction between high- and low-dimensional operations. All implementations we are aware of fall into one of the following three categories:

Approach 1: Separate software. The whole RB scheme is implemented as a single software package which is able to read and process high-dimensional data produced by some external or self-written high-dimensional solver. All high-dimensional operations (section 3.1) within the offline phase are carried out by the RB software, with the possible exemption of the solution snapshot computation which may be performed by the solver. A typical example is the `rbMIT` [28] package for MATLAB.

The main benefit of this approach is its simplicity: a single self-contained software package can be developed and maintained by experts for model order reduction in a programming language ecosystem of their choice. Interfacing external high-dimensional solvers is simple, only export of system matrices and (solution) vectors has to be implemented on the solver side.

Consequently, the RB software has to be able to work with the high-dimensional data produced by the solver. For example, the given sparse matrix format has to be supported and an adequate linear solver for the computation of Riesz representatives needs to be available. While this may be easily possible for small to medium sized discretizations using the tools provided by numerics packages such as MATLAB, the limitations of this approach become obvious when we think of large-scale memory distributed problems solved on computer clusters where, for instance, a single system matrix for the whole problem is never assembled. By design, this approach also cannot be used in conjunction with matrix-free solvers.

Handling of empirical interpolation is problematic, as well: either, for each model the exact same B_μ (with correct ordering of degrees of freedom) has to be reimplemented, or B_μ has to be evaluated by the external solver. This, however, does not seem to fit the paradigm of this approach very well.

Approach 2: Inside high-dimensional solver. The complete reduction process is carried out by the high-dimensional solver which has been extended by an RB module. Examples are the RB implementations of the `libMesh` [21] and `feel++` [10] PDE solver packages.

This approach offers the tightest possible integration between the high-dimensional model and the RB code, allowing maximum performance and easy development of advanced reduction techniques which might require special operations on the high-dimensional data. Also, there cannot be any interoperability issues between different versions of the model reduction and the solver code.

As a downside, implemented algorithms can only be used within the chosen software ecosystem. Given the large number of available PDE solver libraries, the reusability of the code vastly diminishes and ultimately hinders collaboration. Moreover, the implementor is required to have a good understanding of the inner workings of the PDE solver library, which is typically written in a system language such as C++. Many researchers working on model order reduction do not have such technical knowledge, however. Consequently, many new methods are often only evaluated for ad hoc implementations of academic “toy problems”.

Approach 3: Separate low- and high-dimensional operations. As a compromise between the aforementioned approaches, all low-dimensional operations are implemented in a dedicated reduced basis software which communicates over well-defined interfaces with the PDE solver carrying out the high-dimensional operations (section 3.1). This approach has been pursued by the integration of `RBmatlab` with the `dune-rb` module of the DUNE numerics environment [12].

This approach shares many benefits of the previous two approaches. The main RB logic can be developed independently from the solver in a programming language of choice and can be reused with any external solver implementing the necessary

interfaces. All high-dimensional operations are performed by optimized solver code. Memory distributed or matrix-free implementations can be easily utilized.

However, extending the PDE solver to perform all high-dimensional model reduction operations (section 3.1) still requires substantial work. A change of the reduction algorithm will usually also require modification of the RB code in the solver. This can be a major issue when both components are developed by separate teams, in particular for research projects where these components are under constant change.

3.4. Design of `pyMOR`. Our design is based on the central observation that all high-dimensional operations in RB schemes (section 3.1) can be expressed using only a small set of basic operations on *operators* and (collections of) *vectors* which are independent from the concrete reduction scheme in use. This lead us to the following basic design paradigm for `pyMOR`:

1. Define model reduction agnostic interfaces for the mathematical objects involved with RB methods and related schemes.
2. Generically implement all algorithms through operations provided by these interfaces.

As with design approach 1, `pyMOR` offers all model reduction code in a single self-contained software package. Since `pyMOR` provides its own basic discretization toolkit, it can be used completely on its own to easily test new reduction algorithms. The data types provided by the toolkit can also be used to import high-dimensional data from a disk which has been produced by an external solver. This allows the use of `pyMOR`'s algorithms in a workflow similar to design approach 1. Once a new model reduction algorithm has been developed, `pyMOR`'s interfaces allow us to easily apply the exact same algorithm to high-fidelity models implemented in a high-performance solver running on a large computer cluster.

Integrating a new external solver will usually require additional work on the solver side. However, since `pyMOR`'s interfaces work on a lower model reduction agnostic level compared to design approach 3, these modifications can be made mostly independently from the development of new model reduction algorithms. Also, our approach offers higher code reusability and maintainability since the complete reduction algorithm can be implemented in a single software library. This is particularly important when the model reduction library shall be used in conjunction with different PDE solver ecosystems.

`pyMOR`'s interfaces correspond to data structures which are already present in most PDE solver designs: operators and vectors. Thus, implementing `pyMOR`'s interfaces is basically equivalent to exposing the solver's internal data structures via a public application programming interface (API). Refactoring a PDE solver to offer such an API generally empowers the user to easily use and extend the solver in new ways which have not been envisioned by the developers. For example, `pyMOR` implements time-stepping algorithms which can be used to easily manufacture instationary discretizations out of stationary discretizations (see section 5.2). While time-stepping schemes are clearly not a focus for `pyMOR`'s development, a software library of advanced time-stepping algorithms could use such an API to allow easy testing of these algorithms with a given discretization, after which a selected algorithm might be implemented in the solver for maximum performance. Moreover, a public API also allows interactive control of the solver, especially when used in conjunction with dynamic languages such as `Python`. Such interactive sessions can be a powerful tool for debugging, allowing us to inspect and modify the solvers state in ways not possible with a classical debugger.

Note that design approach 3 and `pyMOR`'s design strongly differ in the view on the relationship between the model reduction software and the high-dimensional solver: while approach 3 advocates a strong separation between low- and high-dimensional operations, we think of both components as strongly intertwined. For example, with `pyMOR` it is easily possible to perform preliminary analyses of reduced models for which offline/online decomposition is not yet available.

While design approaches 2 and 3 allow us to perform all high-dimensional reduction operations with the maximum efficiency the PDE solver has to offer, it is clear that `pyMOR`'s interface design comes at a price of suboptimal performance: every call of an interface method incurs a certain overhead, compiler optimizations may be hindered, and the restriction to the available interface methods may prevent implementations which optimally exploit the given data structures. To assess the possible loss in performance, we have conducted several performance benchmarks (section 5.1) which show that, while a certain overhead exists, it becomes negligible for large problems.

4. Model reduction with `pyMOR`. In this section we present `pyMOR`'s interfaces in more detail (section 4.1) and discuss how the integration of external solvers via `pyMOR`'s interfaces can be technically realized (section 4.2). A detailed example for the reduction of models implemented with the `FEniCS`, `deal.II` and `DUNE` solver libraries is presented in section 4.3. Finally, we cover the parallelization of `pyMOR`'s reduction algorithms in section 4.4.

4.1. `pyMOR`'s interface classes. From a bird's eye perspective, `pyMOR` can be seen as a collection of generic algorithms operating on `VectorArray`, `Operator`, and `Discretization` objects which implement the interface methods that are defined by the abstract `VectorArrayInterface`, `OperatorInterface`, and `Discretization` base classes.² To integrate `pyMOR` with an external high-dimensional solver, wrapper classes for these types representing the corresponding high-dimensional data inside the solver have to be implemented (cf. Figure 1).

`VectorArrays` are ordered collections of vectors of same dimension, on which standard linear algebra operations such as linear combination (`lincomb`) or inner products (`dot`, `gramian`) can be performed. Selected degrees of freedom can be extracted (`components`), as required for empirical interpolation. All `VectorArray` methods can either act on single vectors or the whole array. Choosing arrays of vectors as elementary objects in `pyMOR` (as opposed to single vectors) not only allows us to express many model reduction operations in a concise manner, it also allows implementations which optimally exploit the vectorized structure of the instructions for performance. An example of such an implementation is `pyMOR`'s `NumpyVectorArray` (cf. section 5.1). Lists of single vector objects inside external solvers can be easily managed using the `ListVectorArray` class.

`Operators`, which in `pyMOR` represent parametric matrices, bilinear forms, inner products, functionals, or nonlinear operators, can be applied to `VectorArrays`, yielding a new `VectorArray` of results (`apply`). Access to (linear) solvers is provided via `apply_inverse`, the Jacobian of an `Operator` is obtained as a new `Operator` via the `jacobian` method. Affinely decomposed (2.2) operators are represented as `LincombOperators` which hold lists of the `Operators` B_q and the `ParameterFunctionals` θ_q . `LincombOperators` can contain other `LincombOperators` as summands,

²The full documentation of all interface methods is shipped with `pyMOR` and is available online at <http://docs.pyMor.org/>.

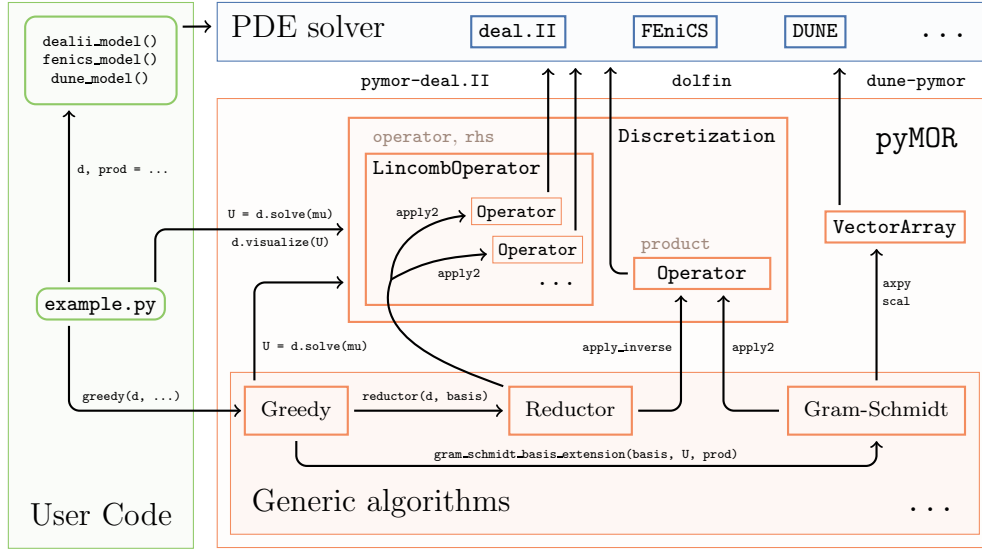


FIG. 1. Schematic view of a typical pyMOR application displaying the interaction between the user code, external PDE solvers, and several parts of pyMOR (see section 4.3).

allowing an easy construction of arbitrarily nested affine decompositions, which are automatically handled by pyMOR's reduction algorithms.

Finally, **Discretizations** act as structured containers for **Operators**. For instance, `StationaryDiscretization` has an `operator` and `rhs` attribute. Moreover, **Discretizations** can be solved for a given parameter, returning a solution `VectorArray` (`solve`) which might be visualized using the `visualize` method. In `StationaryDiscretization`, `solve` for a given parameter `mu` is implemented generically by calling `self.operator.apply_inverse(self.rhs.as_vector(mu), mu=mu)`. Other **Discretizations** might call optimized solution algorithms of an external solver.

It is an important property of pyMOR's interfaces that each method either returns low-dimensional data or new `VectorArray`, `Operator`, or `Discretization` objects. This ensures that no high-dimensional data ever has to be communicated between the external solver and pyMOR and that no code for handling the solver-specific high-dimensional data structures has to be added to pyMOR.

Note that not only the high-dimensional model but also the reduced low-dimensional model is represented by `VectorArrays`, `Operators`, and `Discretizations` implemented inside pyMOR. This allows us to use all algorithms in pyMOR with both high- and low-dimensional objects. For instance, the reduced model could be interpreted again as the high-dimensional model for an additional reduction step.

4.2. Implementation. pyMOR is implemented with Python, a managed, dynamically typed programming language, which is easy to pick up (even for inexperienced programmers), allowing quick and easy prototyping algorithms.

In contrast to MATLAB, Python does not have copy-on-write semantics for assignment which, while allowing more precise control over data, often raises the issue of object ownership. To alleviate the novice user from having to care too much about ownership, pyMOR enforces immutability on all **Discretizations** and **Operators**. In combination with Python's dynamic memory management, this makes the question

LISTING 1
Main script of `example.py` (simplified).

```

1  from pymor.basic import *
2  from functools import partial

3  d, prod = pymor_model()

4  coerc_est = ExpressionParameterFunctional('min(diffusion)', d.parameter_type)
5  reducer = partial(reduce_stationary_coercive,
6                  error_product=prod, coercivity_estimator=coerc_est)
7  rd, rc = reduce_naive(d, reducer, 10)

8  res = reduction_error_analysis(rd, d, rc,
9                              error_norms=[induced_norm(prod)], random_seed=77)
10 print(res['summary'])

```

of ownership irrelevant for these objects. For `VectorArrays`, we have implemented shallow-copy/deep-copy-on-write semantics.

While `Python` is designed as a general purpose language, the `NumPy` [27] package offers a multidimensional array class with similar feature set and performance as `MATLAB` matrices. Additional numerical algorithms and sparse matrix types can be found in the `SciPy` [20] package. Both packages are used extensively in `pyMOR` for all low-dimensional data structures as well as for `pyMOR`'s builtin discretization toolbox.

`pyMOR`'s interfaces do not make any assumption on how the communication between `pyMOR` and the external solver is implemented by the interfacing classes, and many communication patterns are conceivable, such as disk-based communication via job and output files or network-based communication via a standard protocol such as `xml-rpc` or custom protocols. Being a long-running general purpose scripting language, `Python` is ideally suited for `pyMOR`'s interface-based approach, offering a large selection of extension packages for handling virtually any established input-output protocol.

However, in the spirit of the tight coupling between `pyMOR` and the external solver as promoted by our design, we favor, whenever possible, to integrate the solver by recompiling it as a `Python` extension module, giving `Python` direct access to the solver's data structures. This design not only delivers maximum performance as no communication overhead is present, it also allows us to directly manipulate the solvers state from within `Python` beyond the operations available via `pyMOR`'s interfaces. This allows easy exploration of new ideas and offers the user an interactive `Python` debugging shell with direct access to the solver's memory. All external solvers in the following examples have been integrated with this approach, showing its feasibility, even for `MPI`-distributed solvers running on high-performance computing clusters. This approach has also been taken for the integration of `pyMOR` with the `BEST` battery simulation code as part of the `MULTIBAT` project [25].

4.3. `pyMOR` by example. In the following, we consider a basic example of how four different high-dimensional models of the form (2.1), implemented with `pyMOR`'s discretization toolkit, `FEniCS` [22], `deal.II` [2], and the `DUNE` numerics environment [4, 3], can be all reduced with `pyMOR` using identical reduction algorithms.

Listing 1 contains the typical workflow in a `pyMOR` application. A `Discretization` holding the high-dimensional model is obtained first by calling the `pymor_model` method. In addition, `pymor_model` returns an `Operator` representing the inner prod-

LISTING 2
Methods used in Listing 1.

```

1 def pymor_model():
2     problem = ThermalBlockProblem(num_blocks=(2, 2))
3     d, _ = discretize_elliptic_cg(problem, diameter=1./100.)
4     return d, d.h1_0_semi_product

5 def reduce_naive(d, reductor, rb_size):
6     training_set = d.parameter_space.sample_randomly(rb_size)

7     basis = d.operator.source.empty()
8     for mu in training_set:
9         basis.append(d.solve(mu))

10    rd, rc, _ = reductor(d, basis)
11    return rd, rc

```

uct on the space V . Next, a **reductor** for performing the actual RB projection is selected. Here, we choose the generic **reduce_stationary_coercive** method, which will also assemble the error estimator (2.7) according to [7]. A **ParameterFunctional** which assigns to each $\mu \in \mathcal{P}$ the coercivity estimate α_μ is provided. The reduced basis of size 10 and the resulting reduced model are then created using the **reduce_naive** method, which returns the **Discretization** holding the reduced model (**rd**) along with a **Reconstructor** object which is able to perform the high-dimensional reconstruction (2.6). Finally, the quality of the reduced model is evaluated using **pyMOR**'s **reduction_error_analysis** method, which computes the model reduction error and the error estimator effectivity for random parameters.

Listing 2 contains the **pymor_model** and **reduce_naive** methods used in Listing 1. The former uses **pyMOR**'s built-in discretization toolkit, first instantiating a problem description class (line 2) and then discretizing the problem using first order continuous finite elements (line 3). We consider here a classic 2×2 “thermal block” test problem of finding solutions for the stationary diffusion equation

$$(4.1) \quad \begin{aligned} -\nabla \cdot (a_\mu(x) \nabla u_\mu(x)) &= 1, & x \in \Omega &:= [0, 1]^2, \\ u_\mu(x) &= 0, & x \in \partial\Omega, \end{aligned}$$

with diffusion coefficient a_μ given by the linear combination of indicator functions

$$(4.2) \quad a_\mu(x) := \sum_{i=0}^m \sum_{j=0}^n \mu_{i,j} \cdot \chi_{[i/m, (i+1)/m] \times [j/n, (j+1)/n]}(x),$$

$m = n = 2$, for parameters $\mu \in \mathcal{P} := [0, 1]^{2 \times 2}$. We can think of this as computing the heat distribution in a square material composed of 2×2 blocks of different thermal conductivity $\mu_{i,j}$ while being uniformly heated and its temperature at the boundary kept constant at a value of 0.

The **reduce_naive** method simply computes the reduced basis by selecting a set of random parameters (line 6) for which the high-dimensional solution is computed and appended to the **basis VectorArray** (lines 7–9).

Listings 1 and 2 already form a complete **pyMOR** application. However, to obtain good results, more advanced basis generation techniques should be used instead of **reduce_naive**. Listing 3 contains three alternatives using POD, the basic greedy algorithm (Algorithm 1) and an extended adaptive version according to

LISTING 3

Further reduction methods in example.py.

```

1  def reduce_pod(d, reductor, product, snapshots, rb_size):
2      training_set = d.parameter_space.sample_uniformly(snapshots)

3      snapshots = d.operator.source.empty()
4      for mu in training_set:
5          snapshots.append(d.solve(mu))
6      basis, singular_values = pod(snapshots, modes=rb_size, product=product)

7      rd, rc, _ = reductor(d, basis)
8      return rd, rc

9  def reduce_greedy(d, reductor, product, snapshots, rb_size):
10     training_set = d.parameter_space.sample_uniformly(snapshots)
11     ext_alg = partial(gram_schmidt_basis_extension, product=product)
12     result = greedy(d, reductor, training_set,
13                    extension_algorithm=ext_alg, max_extensions=rb_size,
14                    pool=new_parallel_pool())
15     return result['reduced_discretization'], result['reconstructor']

16 def reduce_adaptive_greedy(d, reductor, product, validation_mus, rb_size):
17     ext_alg = partial(gram_schmidt_basis_extension, product=product)
18     result = adaptive_greedy(d, reductor, validation_mus=-validation_mus,
19                             extension_algorithm=ext_alg, max_extensions=rb_size,
20                             pool=new_parallel_pool())
21     return result['reduced_discretization'], result['reconstructor']

```

[16]. `reduce_pod` extends `reduce_naive` by computing a POD of the given solution snapshots (line 6). The `reduce_greedy` method mainly defers all work to pyMOR’s `greedy` implementation (lines 12–14) which has to be called with a training set of parameters (line 10) and a method for extending the existing reduced basis by the new solution snapshot. For this stationary problem, we can simply choose `gram_schmidt_basis_extension` which orthonormalizes the new solution snapshot w.r.t. the old basis using a stabilized Gram–Schmidt process including reorthonormalization for improved numerical accuracy. Similarly, `reduce_adaptive_greedy` defers all work to pyMOR’s `adaptive_greedy` method. Both algorithms are provided a new default worker pool (`new_parallel_pool`) for automatic parallelization of the reduction error estimation (line 1 of Algorithm 1).

All four reduction methods are included in the `example.py` script provided in the supplementary material, which contains a slightly extended version of Listing 1 as the main function. In addition to `pymor_model`, three further high-dimensional models are available (cf. Listing 4).

`fenics_model` uses the `dolfin` [23] Python module of the FEniCS project to discretize a 4×3 “thermal block” problem (4.1), (4.2) ($m = 4, n = 3$) using higher order finite elements. The resulting matrix objects are then wrapped by pyMOR’s `FenicsMatrixOperator` and `FenicsVector` classes which translate pyMOR interface calls into appropriate operations on the underlying FEniCS data structures (lines 8, 10, 11). The affine decomposition (2.2) of the problem is encoded by defining appropriate `ParameterFunctionals` and then constructing an operator representing the decomposition (lines 6, 7, 9). Finally, an appropriate parameter space is chosen, and everything is wrapped up in a generic `StationaryDiscretizaion` object (lines 12–14). Figure 2 (left) shows model reduction errors and timings for `fenics_model`

LISTING 4
Further models in `example.py` (shortened).

```

1 def fenics_model():
2     ... # FEniCS code to setup discrete function space,
3     V, matrices, rhs, h1_mat = ... # as well as system, inner product and rhs matrices

4     from pymor.operators.fenics import FenicsMatrixOperator
5     from pymor.vectorarrays.fenics import FenicsVector
6     coeffs = [ProjectionParameterFunctional('diffusion', (4, 3), (3 - y - 1, x))
7               for x in range(4) for y in range(3)]
8     ops = [FenicsMatrixOperator(m, V, V) for m in matrices]
9     op = LincombOperator(ops, coeffs)
10    rhs = VectorFunctional(ListVectorArray([FenicsVector(rhs, V)]))
11    h1_product = FenicsMatrixOperator(h1_mat, V, V, name='h1_0_semi')

12    param_space = CubicParameterSpace(op.parameter_type, 0.1, 1.)
13    d = StationaryDiscretization(op, rhs, products={'h1_0_semi': h1_product},
14                                parameter_space=param_space)
15    return d, d.h1_0_semi_product

16 def dealii_model():
17     from dealii_example import ElasticityExample
18     example = ElasticityExample(refine_steps=9)

19     from pydealii.pymor.operator import DealIIMatrixOperator
20     from pydealii.pymor.vectorarray import DealIIVector
21     coeffs = [ProjectionParameterFunctional('mu', tuple()),
22               ProjectionParameterFunctional('lambda', tuple())]
23     ops = [DealIIMatrixOperator(example.mu_mat()),
24            DealIIMatrixOperator(example.lambda_mat())]
25     op = LincombOperator(ops, coeffs)
26     rhs = VectorFunctional(ListVectorArray([DealIIVector(example.rhs())]))

27    param_space = CubicParameterSpace(op.parameter_type, 1., 10.)
28    d = StationaryDiscretization(op, rhs, products={"energy": ops[0]},
29                                parameter_space=param_space)
30    return d, d.energy_product

31 def dune_model():
32     from spe10 import examples, wrapper
33     example = examples['aluconformgrid']['pdelab']['ist1']('60 220 85', True)
34     d = wrapper[example.discretization()]

35    param_ranges = {'blockade': (1e-4, 1), 'sink': (0, 1)}
36    param_space = CubicParameterSpace(op.parameter_type, ranges=param_ranges)
37    d = d.with_(parameter_space=param_space)
38    return d, d.energy_0_product.assemble(1e-4)

```

(second order finite elements, 361.201 degrees of freedom) reduced by `reduce_greedy` using the training sets $\mathcal{S}_{train,1} := \{0.1, 1\}^{4 \times 3}$ and $\mathcal{S}_{train,2} := \{0.1, 0.55, 1\}^{4 \times 3}$. We observe that choosing a too small training set leads to *overfitting*: while the solution is very well approximated for parameters in the training set, the approximation quality is not maintained for parameters not contained in the training set. The error $|u_\mu - u_{N,\mu}|$ for the final basis size ($\mathcal{S}_{train,2}$) is shown in Figure 2 (right).

In `discretize_dealii`, we consider the two-dimensional linear elasticity problem

$$(4.3) \quad -\nabla \lambda (\nabla \cdot u_{\mu,\lambda}) - (\nabla \cdot \mu \nabla) u_{\mu,\lambda} - \nabla \cdot \mu (\nabla u_{\mu,\lambda})^T = f \quad \text{in } \Omega = [0, 1]^2$$

with Lamé parameters $\mu, \lambda \in [1, 10]$ and homogeneous Dirichlet boundary conditions

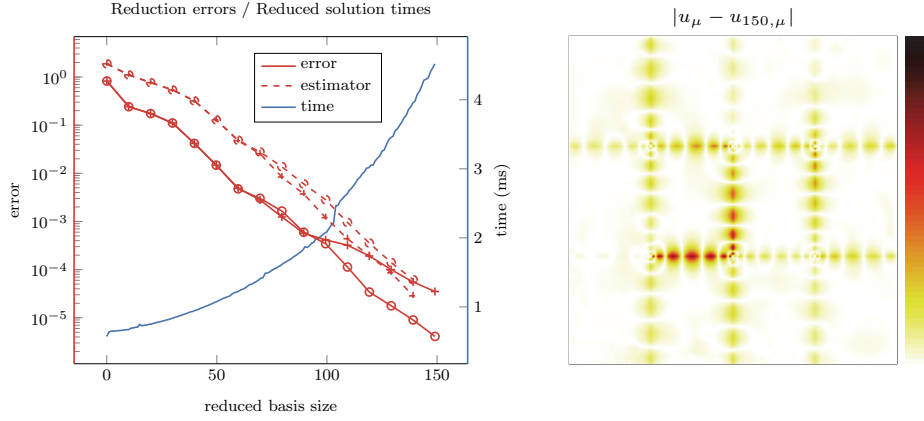


FIG. 2. Error analysis of the RB approximation of `fenics_model` using `reduce_greedy`. Left: Maximum H^1 -model order reduction error (red, solid) on a test set of 1000 parameters and maximum estimated error (red, dashed) on the training set in dependence on the reduced basis ($S_{train,1} := \{0.1, 1\}^{4 \times 3}$; \circ , $S_{train,2} := \{0.1, 0.55, 1\}^{4 \times 3}$; $+$); average time for the computation of a reduced solution including error estimation in dependence on the reduced basis size (blue). Right: Plot of the absolute difference (dark: 0, light: $6.03 \cdot 10^{-7}$) between the detailed and the reduced solution $|u_\mu - u_{150,\mu}|$ (for $S_{train,2}$) for the worst approximated parameter of the test set. (Figure appears in color online only.)

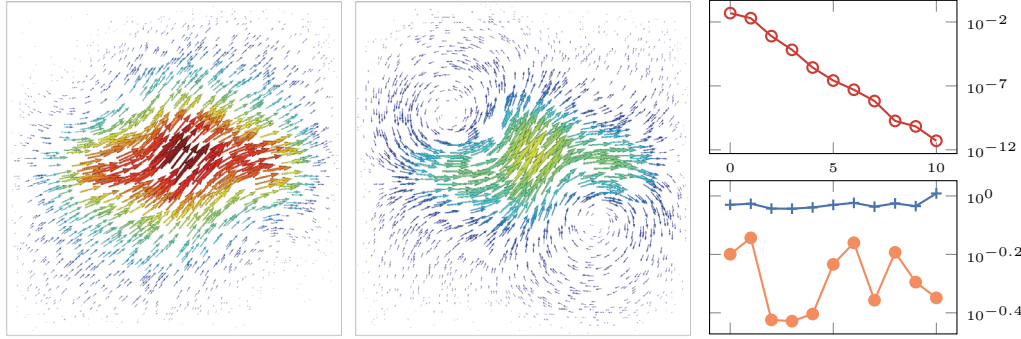


FIG. 3. Sample solutions and error analysis of the RB approximation of `dealii_model` using `adaptive_greedy`. Plot of displacement field $u_{\mu,\lambda}$ for $(\mu, \lambda) = (1, 1)$ (left) and $(\mu, \lambda) = (1, 10)$ (middle) with color representing magnitude (blue = 0, red = 0.033). Right: Plot of maximum energy model reduction error (top, \circ) and error estimator effectivity (bottom, min: \bullet , max: $+$) over a test set of ten random parameters against the size of the reduced basis ($[0, 10]$). (Figure appears in color online only.)

from the tutorial documentation for the `deal.II`³ [2] C++ solver library (cf. Figure 3, left, middle). The tutorial program was refactored into the `ElasticityExample` class and `Python` bindings for this class were added (line 17). Moreover, we implemented `Python` bindings for the `deal.II` `SparseMatrix` and `Vector` classes, as well as corresponding `pyMOR` wrapper classes. These are used to make the `deal.II` matrices and right-hand side vector provided by `ElasticExample` available as `pyMOR` `Operators` (lines 23, 24, 26). The `ElasticityExample` calculates a solution for the displacement field $u_{\mu,\lambda}$ by discretizing (4.3) with first order continuous Galerkin finite elements (65.536 degrees of freedom) and solves the resulting linear system with

³<https://dealii.org/8.1.0/doxygen/deal.II/step.8.html>

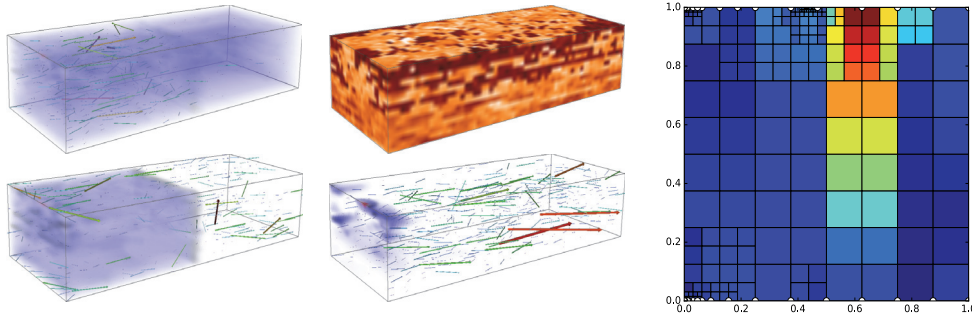


FIG. 4. Data function, sample solutions, and the adaptively refined set of training samples of `dune_model` using `adaptive_greedy`. Top center: Logarithmic plot of the Frobenius norm of the anisotropic SPE10 model2 permeability tensor (dark: $6.65 \cdot 10^{-8}$, light: $2 \cdot 10^5$). Left and bottom center: Volume plot of the pressure u_μ and vector plot of the reconstructed Darcy velocity $-\kappa_\mu \nabla u_\mu$ (colored and scaled by magnitude) for several parameters (blue: weak, red: strong): $\mu = (10^{-4}, 0)$ bottom left, $\mu = (10^{-4}, 1)$ top left, and $\mu = (1, 0)$ bottom center. The first parameter component models the existence of a blockade in the middle (enabled: 10^{-4} , disabled: 1) and the second parameter component acts as a switch for a sink near the blockade. Right: Plot of the adaptively refined training set for a reduced basis of size 50 (colored contribution of the local error indicators, blue: low, red: high) and selected training parameters (white circles). Note the strong influence of the first parameter component (which influences the operator), as opposed to the second parameter component (which only influences the right-hand side). (Figure appears in color online only.)

a conjugate gradient method. The Python bindings and `pyMOR` wrappers are available in the `pyMOR-deal.II` package. Figure 3 (right) shows model reduction errors and estimator effectivities for `dealii_model` reduced by `adaptive_greedy` for a reduced basis of size 10.

`dune_model` uses the DUNE numerics environment to discretize a multiscale single phase flow problem with the highly heterogeneous and anisotropic SPE10 model2 permeability field,⁴ with inflow boundary conditions at $x_1 = 0$, a fixed pressure at $x_1 = 5$ and no flow at the other boundaries:

$$(4.4) \quad -\nabla \cdot (\kappa_\mu \nabla u_\mu) = f_\mu \quad \text{in } \Omega := [0, 2] \times [0, 5] \times [0, 1].$$

The parameter μ allows us to toggle the presence of a blockade at $x_1 = 2.5$ and a sink near the blockade (cf. Figure 4, left and middle). Problem (4.4) is discretized with `dune-gdt` [31] using first order continuous finite elements on a tetrahedral grid with $6.7 \cdot 10^6$ elements ($1.2 \cdot 10^6$ degrees of freedom). The integration with `pyMOR` is based on `dune-pymor` which automatically wraps the resulting discretization objects defined in the `dune-hdd` module as `pyMOR Discretizations` (line 34) taking the affine decomposition of the problem into account. Only the parameter space is additionally chosen (lines 36–37). Figure 4 shows the adaptively refined training set and selected training parameters for a reduced basis of size 50, which yields a maximum absolute model reduction energy error of $5.5 \cdot 10^{-10}$ over a set of 100 randomly chosen test parameters.

Again, let us note that all four high-dimensional models can be reduced with the exact same model reduction code, ranging from the trivial `reduce_naive` algorithm to the sophisticated adaptive snapshot selection in `reduce_adaptive_greedy`. In fact,

⁴<http://www.spe.org/web/csp/datasets/set02.htm>

the `example.py` script provided in the supplementary material allows us to choose any of the 16 possible combinations between PDE solver and reduction method.

4.4. Parallelism in pyMOR. In many application areas of reduced order modeling, not only the efficiency of the reduced model but also the required time for generating the model needs to be taken into account. Thus, RB software should be able to perform offline computations with good computational efficiency. For modern computing architectures this requires parallelization of algorithms.

In a greedy basis generation algorithm (Algorithm 1), the main computational work is made up by three types of operations: 1. reduction error estimation on \mathcal{S}_{train} , 2. computation of the solution snapshot u_{μ^*} , and 3. reduced basis extension and reduction of the high-dimensional model.

For the parallelization of the high-dimensional operations in steps 2 and 3, pyMOR relies on already existing high-performance parallelization of the external solver. Since pyMOR's interfaces require no communication of any high-dimensional data and are completely implementation agnostic, memory distributed vector data can be handled via the `VectorArray` interface as efficiently as any nondistributed data.

To ease the integration of MPI distributed codes, pyMOR offers tools which allow us to automatically support MPI parallel use of the solver when pyMOR bindings for the sequential case already exist. For instance, to parallelize `fenics_model` in Listing 4, one would simply execute

```
d = mpi_wrap_discretization(lambda: fenics_model()[0],
                             use_with=True, pickle_subtypes=False)
prod = d.h1_0_semi_product
```

When the script is executed with `mpirun`, an event loop is launched on all MPI ranks except for rank 0 which executes the main script. `mpi_wrap_discretization` then instructs each rank to execute the function given as the first argument to obtain a local `Discretization` object. The `Discretization` returned by `mpi_wrap_discretization` and all contained `Operators` will then use the event loop to issue MPI distributed operations on the rank-local objects when corresponding interface methods are called. In section 5.2, we consider another example where pyMOR's MPI wrappers are used.

For the parallelization of step 1 and similar embarrassingly parallel tasks which require little to no communication, pyMOR provides an abstraction layer for existing Python parallelization solutions based on a simple worker pool concept (`pymor.parallel`). For instance, line 1 in Algorithm 1 could be parallelized by executing

```
numpy.argmax(pool.map(lambda mu, rd: rd.estimate(rd.solve(mu), mu),
                    training_set, rd=reduced_discretization))
```

The function and all arguments are automatically serialized and distributed to the workers of the pool, ensuring that immutable data is only communicated once.

pyMOR currently provides a worker pool implementation based on the `IPython` [29] toolkit, which allows easy parallel computation with large collections of heterogeneous compute nodes, and an MPI-based implementation using pyMOR's event loop which can seamlessly be used in conjunction with external solvers using the same event loop. The MPI-based worker pool was also used for the experiment in Figure 2, employing 16 cores of the compute server described in section 5.1. For the larger training set $\mathcal{S}_{train,2}$ with 531.441 parameters, the total offline computation time was 11 hours, of which 5 hours were spent on error estimation. Performing the error estimation without parallelization would have required an additional 37 hours of computation

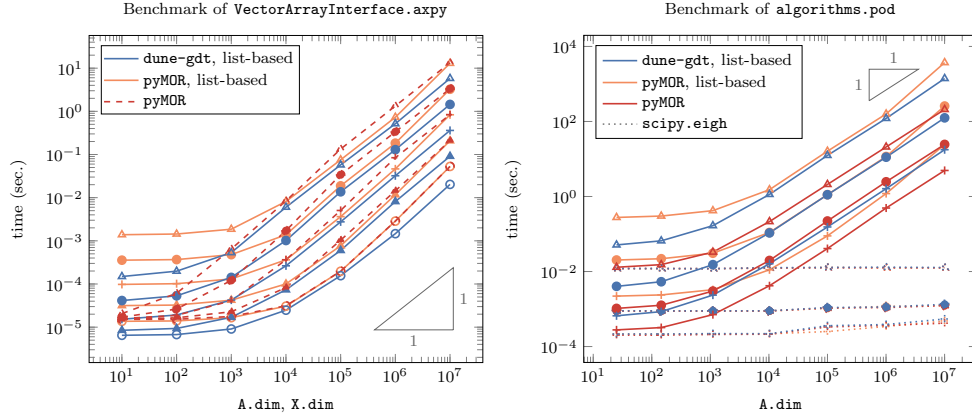


FIG. 5. Log/log plot of the measured execution time of `A.axy` (left) and the POD algorithm (right) for different implementations and several lengths of the vector array (`len(A)=1`: \circ , `len(A)=4`: \triangle , `len(A)=16`: $+$, `len(A)=64`: \bullet , `len(A)=256`: \triangle). Left: Comparison of `A.axy(X)` with `len(X)=len(A)` for several implementations of `A`. Right: Comparison of `pod(A=A, modes=10, product=h1_0, orthonormalize=False, check=False)` for several implementations of `A` and the `h1_0` product (solid) and the respective time spent in `scipy.eigh` (dotted).

time.

5. Performance evaluation. In this section we evaluate the applicability and performance of `pyMOR`'s design approach by considering technical benchmarks of some of `pyMOR`'s interfaces, as well as a challenging nonlinear large-scale model order reduction problem.

5.1. Benchmarks. The main goal of this section is to compare the performance of `pyMOR`'s `VectorArray` and `Operator` interfaces when used to access external high-dimensional solver data structures to native implementations of these classes. Moreover, we want to investigate possible performance benefits for vectorized `VectorArray` implementations.

As native implementations within `pyMOR`, we consider `NumpyVectorArray`, which allows vectorized operations on vectors by internally holding an appropriately sized two-dimensional NumPy array, as well as `ListVectorArray` maintaining a Python list data structure holding vector objects implemented as one-dimensional NumPy arrays. The inner product in the `pod` benchmark is implemented with `NumpyMatrixOperators` holding sparse SciPy matrices coming from `pyMOR`'s own discretization toolbox.

The external solver code is based on the DUNE numerics environment [4, 3], centered around the discretization toolbox `dune-gdt` [31] (cf. section 4.3) and is compiled as a Python extension module as described in section 4.2.

Our benchmarks were executed on a dual socket compute server equipped with two Intel Xeon E5-2698 v3 CPUs with 16 cores running at 2.30GHz each and 256GB of memory available. All benchmarks were performed as single-threaded processes.

Vector array benchmark. We consider the `axy` method of the `VectorArray-Interface`, which performs a vectorized BLAS-conforming `axy` operation, i.e., pairwise in-place addition of the vectors in the array with vectors of a second array multiplied by a scalar factor.

As we observe in Figure 5 (left), both `ListVectorArray`-based implementations show about the same performance for sufficiently large array dimensions. In fact,

the DUNE-based implementation (`dune-gdt`, list based) actually performs better than the NumPy-based implementation (`pyMOR`, list based), showing the tight integration between `pyMOR` and the external solver. The `NumpyVectorArray` implementation, on the other hand, cannot benefit from vectorization for larger array dimensions.⁵

POD benchmark. As detailed previously (e.g., section 3.1), the Gram–Schmidt and POD algorithms are important tools in the context of model reduction. The implementation of a numerically stable Gram–Schmidt or POD algorithm might not be completely straightforward and it is an important benefit that `pyMOR`’s interface design allows us to provide tried and tested implementations of these algorithms which can automatically be used with any external solver integrated with `pyMOR`.

`pyMOR`’s POD algorithm mainly consist of three steps with different complexities: 1. computation of a Gramian matrix w.r.t. the given inner product (`product.apply2`), 2. computation of the eigenvalue decomposition of the Gramian (using the SciPy `eigh` method), and 3. mapping right-singular vectors to left-singular vectors (by calling `lincomb` on the original `VectorArray`). Note that the computational cost for steps 1 and 3 depends on the `VectorArray` and `Operator` implementation, scaling linearly with the array dimension and quadratically (resp., linearly) with the array length. The computational cost for step 2 is independent of the space dimension, and only increases with the number of given vectors.

As the inner product for both benchmarks and all implementations we have chosen the full H^1 -product matrix stemming from a first order continuous finite element discretization over the same structured triangular grid on the unit square.

As we observe again in Figure 5 (right) both `ListVectorArray`-based implementation show roughly equal performance. However, the vectorized `pyMOR` implementation is able to clearly outperform both of the other implementations thanks to the fact that the computationally dominant steps 1 and 3 of the algorithm can be expressed idiomatically via a single interface call. This shows that `VectorArray` implementations can indeed greatly benefit from `pyMOR`’s vectorized interface design. `NumpyVectorArray`, for instance, does so by calling NumPy’s `dot` method which is able to defer the task to highly optimized BLAS implementations. We expect similar performance benefits for external high-dimensional solvers, when consecutive-in-memory arrays of vectors are available as native data structures inside theses solvers.

5.2. A large, nonlinear problem. To evaluate `pyMOR`’s ability to handle large-scale problems, we consider a three-dimensional version of the Burgers-type problem already discussed in [13], i.e., we solve the scalar conservation law

$$(5.1) \quad \begin{aligned} \partial_t u_\mu(t, x) + \nabla_x \cdot (\mathbf{v} \cdot u(t, x)^\mu) &= 0, & x \in [0, 2] \times [0, 1]^2, & \quad t \in [0, 0.3], \\ u_\mu(0, x) &= \frac{1}{2}(1 + \sin(2\pi x_1) \sin(2\pi x_2) \sin(2\pi x_3)) \end{aligned}$$

for exponents $\mu \in [1, 2]$, periodic boundary conditions, and constant transport direction $\mathbf{v} = (1, 1, 1)$ (cf. Figure 6). The problem was discretized using a finite volume scheme on a $480 \times 240 \times 240$ voxel grid with 27.6 million degrees of freedom.

To keep the implementation, which is available in the supplementary material, as simple as possible, we chose a basic Lax–Friedrichs numerical flux with explicit Euler time discretization using 600 equidistant time steps. Our MPI-parallel code

⁵We assume that this is due to the fact that NumPy offers no native `axpy` operations, such that a large temporary array has to be created holding all to be added and scaled vectors. It is planned to improve performance of `NumpyVectorArray.axpy` by directly calling out to BLAS (cf. <https://github.com/pymor/pymor/issues/73>).

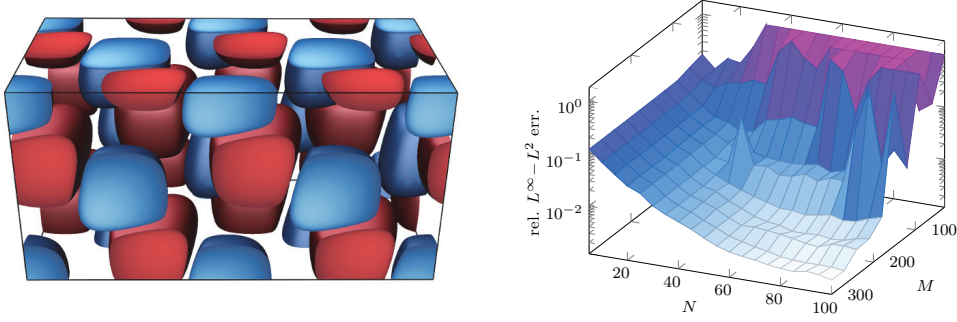


FIG. 6. Left: Plot of the solution of (5.1) for $\mu = 2$ at final time $t = 0.3$. Only values in the intervals $[0, 0.4]$ (blue) and $[0.6, 1]$ (red) are displayed. Right: Model reduction errors for RB approximation of (5.1). Maximum $L^\infty - L^2$ -error on a test set of ten random parameters for different reduced basis sizes N and numbers of interpolation points M . (Figure appears in color online only.)

TABLE 1

Time in seconds needed for the solution of (5.1) for a single parameter, using standalone DUNE code in comparison to DUNE code with time-stepping in pyMOR (best of three runs).

MPI ranks	1	2	3	6	12	24	48	96	192
DUNE	16858	8532	5726	2959	1526	773	396	203	107
pyMOR	17683	8940	6050	3124	1604	815	417	213	110
Overhead	4.9%	4.8%	5.7%	5.6%	5.1%	5.4%	5.3%	4.9%	2.8%

only depends on the DUNE grid interface and includes handwritten pyMOR bindings utilizing the MPI helper classes and event loop covered in section 4.4.

To evaluate the performance of the integration with pyMOR, we compared the solution time for (5.1) using a standalone version of the solver to the time needed with time-stepping done by pyMOR's `explicit_euler` time-stepper (Table 1). We observe that the pyMOR integration shows very good performance with only small overhead compared to the native DUNE version. All computations were performed on 1 to 16 nodes of the University of Münster's PALMA computing cluster. Every node contains 48GB of main memory and two hexa-core Intel Xeon E5650 CPUs.

For the model order reduction we used the EI-GREEDY [17] algorithm to generate the interpolation data for the nonlinear space differential operator and a simple POD for the computation of the reduced basis. In both cases, solution trajectories for ten equidistant parameters were chosen as input which had each been compressed beforehand using additional PODs with a relative tolerance of 10^{-7} . Thanks to the parallelization of the high-dimensional discretization, the offline phase of the experiment could be completed in only 3.4 hours.

Figure 6 summarizes the approximation quality of the reduced model for different reduced basis sizes and numbers of interpolation points. In particular, we observe that for a reduced basis of size 80 and 300 interpolation points, we can achieve a relative $L^\infty - L^2$ model reduction error of $2.6 \cdot 10^{-3}$. A solution for this reduced model takes on average 2.8 seconds using a single processor core, resulting in a speedup of 38 in comparison to a solution of the high-dimensional model using all 192 cores or a speedup of 6000 in comparison to a single-core computation.

Code availability. pyMOR and all further code used for the production of the results in this work are available under open source licenses. The specific versions used here are included in the supplementary material. Current versions of pyMOR, including the wrapper classes for FEniCS, deal.II, and DUNE, as well as the code for the example in section 5.2, can be found at <http://www.pymor.org/>.

REFERENCES

- [1] F. BALLARIN, G. ROZZA, AND A. SARTORI, *RBniCS – reduced order modelling in FEniCS*, ScienceOpen Posters, (2015), doi:10.14293/P2199-8442.1.SOP-MATH.PUQ0WD.v1.
- [2] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *deal.II – a general purpose object oriented finite element library*, ACM Trans. Math. Software, 33 (2007), 24, doi:10.1145/1268776.1268779.
- [3] P. BASTIAN, M. BLATT, A. DEDNER, C. ENGWER, R. KLÖFKORN, R. KORNUBER, M. OHLBERGER, AND O. SANDER, *A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE*, Computing, 82 (2008), pp. 121–138, doi:10.1007/s00607-008-0004-9.
- [4] P. BASTIAN, M. BLATT, A. DEDNER, C. ENGWER, R. KLÖFKORN, M. OHLBERGER, AND O. SANDER, *A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework*, Computing, 82 (2008), pp. 103–119, doi:10.1007/s00607-008-0003-x.
- [5] B. A. BELSON, J. H. TU, AND C. W. ROWLEY, *Algorithm 945: Modred—a parallelized model reduction library*, ACM Trans. Math. Software, 40 (2014), 30, doi:10.1145/2616912.
- [6] P. BINEV, A. COHEN, W. DAHMEN, R. DEVORE, G. PETROVA, AND P. WOJTASZCZYK, *Convergence rates for greedy algorithms in reduced basis methods*, SIAM J. Math. Anal., 43 (2011), pp. 1457–1472, doi:10.1137/100795772.
- [7] A. BUHR, C. ENGWER, M. OHLBERGER, AND S. RAVE, *A numerically stable a posteriori error estimator for reduced basis approximations of elliptic equations*, in Proceedings of the 11th World Congress on Computational Mechanics, X. Oliver, E. Onate, and A. Huerta, eds., CIMNE, Barcelona, 2014, pp. 4094–4102.
- [8] A. BUHR, C. ENGWER, M. OHLBERGER, AND S. RAVE, *Arbilomod, a simulation technique designed for arbitrary local modifications*, arXiv e-prints, 2015, <http://arxiv.org/abs/1512.07840>.
- [9] S. CHATURANTABUT AND D. C. SORENSSEN, *Nonlinear model reduction via discrete empirical interpolation*, SIAM J. Sci. Comput., 32 (2010), pp. 2737–2764, doi:10.1137/090766498.
- [10] C. DAVERSIN, S. VEYS, C. TROPHIME, AND C. PRUD’HOMME, *A reduced basis framework: Application to large scale non-linear multi-physics problems*, ESAIM: Proceedings and Surveys, 43 (2013), pp. 225–254, doi:10.1051/proc/201343015.
- [11] R. DEVORE, G. PETROVA, AND P. WOJTASZCZYK, *Greedy algorithms for reduced bases in Banach spaces*, Constr. Approx., 37 (2013), pp. 455–466, doi:10.1007/s00365-013-9186-2.
- [12] M. DROHMANN, B. HAASDONK, S. KAULMANN, AND M. OHLBERGER, *A software framework for reduced basis methods using dune-rb and rbmatlab*, in Advances in DUNE, A. Dedner, B. Flemisch, and R. Klöforn, eds., Springer, Berlin, Heidelberg, 2012, pp. 77–88, doi:10.1007/978-3-642-28589-9_6.
- [13] M. DROHMANN, B. HAASDONK, AND M. OHLBERGER, *Reduced basis approximation for nonlinear parametrized evolution equations based on empirical operator interpolation*, SIAM J. Sci. Comput., 34 (2012), pp. A937–A969, doi:10.1137/10081157X.
- [14] B. HAASDONK, *Convergence rates of the pod-greedy method*, ESAIM Math. Model. Numer. Anal., 47 (2013), pp. 859–873, doi:10.1051/m2an/2012045.
- [15] B. HAASDONK, *Reduced basis methods for parametrized PDEs—A tutorial introduction for stationary and instationary problems*, chapter to appear in Model Reduction and Approximation: Theory and Algorithms, P. Benner, A. Cohen, M. Ohlberger, and K. Willcox, SIAM, Philadelphia.
- [16] B. HAASDONK, M. DIHLMANN, AND M. OHLBERGER, *A training set and multiple bases generation approach for parameterized model reduction based on adaptive grids in parameter space*, Math. Comput. Model. Dyn. Syst., 17 (2011), pp. 423–442, doi:10.1080/13873954.2011.547674.
- [17] B. HAASDONK, M. OHLBERGER, AND G. ROZZA, *A reduced basis method for evolution schemes with parameter-dependent explicit operators*, Electron. Trans. Numer. Anal., 32 (2008), pp. 145–161.

- [18] J. S. HESTHAVEN, G. ROZZA, AND B. STAMM, *Certified Reduced Basis Methods for Parametrized Partial Differential Equations*, SpringerBriefs in Mathematics, Springer, Cham, 2016, doi:10.1007/978-3-319-22470-1.
- [19] S. F. JOHNSEN, Z. A. TAYLOR, M. J. CLARKSON, J. HIPWELL, M. MODAT, B. EIBEN, L. HAN, Y. HU, T. MERTZANIDOU, D. J. HAWKES, S. OURSELIN, *Niftysim: A gpu-based nonlinear finite element package for simulation of soft tissue biomechanics*, Int. J. Comput. Assist. Radiol. Surg., 10 (2015), pp. 1077–1095, doi:10.1007/s11548-014-1118-5.
- [20] E. JONES, T. OLIPHANT, P. PETERSON, ET AL., *SciPy: Open Source Scientific Tools for Python*, <http://www.scipy.org/> (2001–2015).
- [21] D. J. KNEZEVIC AND J. W. PETERSON, *A high-performance parallel implementation of the certified reduced basis method*, Comput. Methods Appl. Mech. Engrg., 200 (2011), pp. 1455–1466, doi:10.1016/j.cma.2010.12.026.
- [22] A. LOGG, K.-A. MARDAL, G. N. WELLS, EDS., *Automated Solution of Differential Equations by the Finite Element Method*, Springer, Heidelberg, 2012, doi:10.1007/978-3-642-23099-8.
- [23] A. LOGG AND G. N. WELLS, *Dolfin: Automated finite element computing*, ACM Trans. Math. Software, 37 (2010), 20, doi:10.1145/1731022.1731030.
- [24] M. OHLBERGER, S. RAVE, AND F. SCHINDLER, *Model reduction for multiscale lithium-ion battery simulation*, in ENUMATH 2015, Ankara, Turkey, Lect. Notes Comput. Sci. Eng., Springer, 2016.
- [25] M. OHLBERGER, S. RAVE, S. SCHMIDT, AND S. ZHANG, *A model reduction framework for efficient simulation of li-ion batteries*, in Finite Volumes for Complex Applications VII—Elliptic, Parabolic and Hyperbolic Problems, J. Fuhrmann, M. Ohlberger, and C. Rohde, eds., Springer, Cham, 2014, pp. 695–702, doi:10.1007/978-3-319-05591-6_69.
- [26] M. OHLBERGER AND F. SCHINDLER, *Error control for the localized reduced basis multiscale method with adaptive on-line enrichment*, SIAM J. Sci. Comput., 37 (2015), pp. A2865–A2895, doi:10.1137/151003660.
- [27] T. E. OLIPHANT, *Python for scientific computing*, Comput. Sci. Eng., 9 (2007), pp. 10–20, doi:10.1109/MCSE.2007.58.
- [28] A. T. PATERA AND G. ROZZA, *Reduced basis approximation and a posteriori error estimation for parametrized partial differential equations, version 1.0*, copyright MIT 2006–2007, to appear in (tentative rubric) MIT Pappalardo Graduate Monographs in Mechanical Engineering.
- [29] F. PÉREZ AND B. E. GRANGER, *IPython: A system for interactive scientific computing*, Comput. Sci. Eng., 9 (2007), pp. 21–29, doi:10.1109/MCSE.2007.53.
- [30] A. QUARTERONI, A. MANZONI, AND F. NEGRI, *Reduced Basis Methods for Partial Differential Equations*, La Matematica per il 3+2, Springer, Cham, 2016.
- [31] F. SCHINDLER AND R. MILK, *dune-gdt*, (2015), doi:10.5281/zenodo.45465.