Mathematik

Model Reduction for Parametric Multi-Scale Problems

Inaugural-Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften im Fachbereich
Mathematik und Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Felix Schindler, geb. Albrecht
aus Kirchheimbolanden
– 2015 –

Dekan:	Prof. Dr. Martin Stein
Erster Gutachter:	Prof. Dr. Mario Ohlberger
Zweiter Gutachter:	Prof. Axel Målqvist, PhD
Tag der mündlichen Prüfung:	23.10.2015
Tag der Promotion:	23.10.2015

To my beloved wife

Abstract

The focus of the present work are elliptic parametric multiscale problems, which inherit the computational challenges of both multiscale as well as parametric problems. We present the localized reduced basis multiscale method (LRBMS) for the efficient and accurate approximation of such problems. To achieve this goal, the LRBMS combines localization ideas from numerical multiscale methods with model reduction techniques from reduced basis methods.

We present a new reliable and localizable a posteriori error estimate in the context of the LRBMS, which allows to efficiently assess the discretization as well as the model reduction error during all stages of the computational process. Based on this estimate, we propose an adaptive online enrichment procedure to improve the quality of the resulting approximation by solving local corrector problems.

In addition, we present the design and implementation of a generic discretization and model reduction software framework, which is utilized to demonstrate the applicability of the LRBMS, in particular in the context of single-phase flow in porous media with complex heterogeneities.

Zusammenfassung

Die vorliegende Arbeit behandelt elliptische parametrische mehrskalen Problemen. Die numerische Behandlung solcher Probleme wird sowohl durch ihren parametrischen als auch durch ihren mehrskalen Charakter erschwert. In diesem Zusammenhang stellen wir die lokalisierte reduzierte Basis mehrskalen Methode (LRBMS) vor, die der exakten und effizienten Approximation solcher Probleme dient. Um dies zu erreichen kombinieren wir in der LRBMS Lokalisierungsansätze aus numerischen mehrskalen Methoden mit Ansätzen der Modellreduktion aus dem Bereich der reduzierte Basis Methoden.

Für die LRBMS stellen wir einen neuen zuverlässigen und lokalisierbaren a posteriori Fehlerschätzer vor. Dieser Fehlerschätzer ermöglicht es, sowohl den Diskretisierungsfehler als auch den Modellreduktionsfehler in allen Bereichen der numerischen Approximation effizient zu bewerten. Aufbauend auf diesem Fehlerschätzer schlagen wir ein adaptives online Anreicherungsverfahren vor, um die Approximationsgüte durch die Lösung lokaler Korrekturprobleme zu erhöhen.

Zusätzlich stellen wir ein Design und die Implementierung eines generischen Diskretisierungs- und Modellreduktions-Framework vor, um die Anwendbarkeit der LRBMS im Zusammenhang mit ein-Phasen Strömung in porösen Medien mit komplexen Heterogenitäten zu demonstrieren.

Acknowledgments

I would like to express my gratitude towards Mario Ohlberger for providing exceptional support and the best of all environments. I would also like to thank Christian Himpe, Stephan Rave, Anne Schindler, Michael Schindler and Barbara Verfürth for proofreading.

I acknowledge support of the German Research Foundation (DFG) under contract OH 98/4-2, and support of the German Federal Ministry of Education and Research (BMBF) under contract 05M13PMA.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Over the past decades, numerical approximations of solutions of *partial differential equations* (PDEs) have become increasingly important throughout almost all areas of natural sciences. At the same time, research efforts in the field of numerical analysis, computer science and approximation theory have led to adaptive algorithms to produce such approximations in an efficient and accurate manner: by now, advanced approximation techniques are available for a large variety of PDEs. However, there still exist problems where the computational cost of traditional algorithms easily exceeds the limits of available computing power.

This is, for instance, the case for multiscale problems, where the data functions constituting the PDE exhibit a high contrast or strong oscillations on small spatial or temporal scales. It is also the case for parametric problems, where the data functions depend on a low-dimensional input vector and where one is interested in many solutions for a large number of configurations. Multiscale problems arise in almost all contexts: it can be argued that they arise almost always, whenever real-world data is involved [EE2003a, p. 1]. Interest in parametric problems is equally widespread: they arise in the context of optimization, uncertainty quantification and decision making.

For each of these problems, there exist specialized algorithms to overcome the limitations of traditional approximation techniques: numerical multiscale methods are specifically tailored to allow for approximations in the context of multiscale problems, and model order reduction techniques are available to reduced the computational demand associated with parametric problems.

The focus of the present work are elliptic parametric multiscale problems, which inherit the computational challenges of both multiscale as well as parametric problems. We present the *localized reduced basis multiscale method* (LRBMS) for the efficient and accurate approximation of such problems. To achieve this goal, the LRBMS combines localization ideas from numerical multiscale methods with model reduction techniques from reduced basis methods.

This work is organized as follows: in Chapter 1 we introduce the problem setup and present existing approximation techniques. In particular, we discuss appropriate notions of accuracy and efficiency. Chapter 2 covers the LRBMS and presents novel a posteriori error estimation techniques, which are required for accurate and efficient approximations. In Chapter 3 we present the software framework, which was utilized for the numerical experiments presented in Chapter 4, which demonstrate the applicability of the LRBMS.

This work builds on previously published work: parts of Chapter 2 have been published in [OS2014, OS2015], parts of Section 4.1 have been publish in [AHKO2012], parts of Section 4.2.1 have been published in [AO2013], parts of Sections 4.2.4 and 4.4 have been published in [OS2015], and parts of Section 4.3 have been published in [MRS2015].

1 Elliptic parametric multiscale problems

This chapter introduces elliptic *partial differential equations* (PDEs) and numerical approximation techniques. In particular, we consider the accuracy and efficiency of approximations in the context of multiscale problems and parametric problems. To set the stage, we consider the example of immiscible two-phase flow in porous media.

Immiscible two-phase flow in porous media We consider two-phase immiscible flow in a porous medium, which is of interest for instance in the context of oil production, ground water pollution or CO_2 sequestration; we mainly follow the notation of [CHM2006]. Let Ω denote a porous medium with porosity ϕ, κ the absolute permeability tensor, g the gravitational acceleration, z the depth and let ρ_α denote the density, q_α the mass flow, ν_α the viscosity and $\kappa_{r\alpha}$ the relative permeability of the wetting phase ($\alpha = w$) and the non-wetting phase ($\alpha = n$). We are looking for wetting and non-wetting pressures p_α, Darcy velocities u_α and saturations s_α, for $\alpha = w, n$, to fulfill *mass conservation* and *Darcy's law* within each fluid phase,

$$\frac{\partial(\phi\rho_\alpha s_\alpha)}{\partial t} = -\nabla\cdot(\rho_\alpha u_\alpha) + q_\alpha \qquad \text{and} \qquad u_\alpha = -\frac{\kappa_{r\alpha}}{\nu_\alpha}\kappa(\nabla p_\alpha - \rho_\alpha g \nabla z),$$

respectively and the *capillary pressure* relation (for a given capillary pressure p_c),

$$p_c(s_w) = p_n - p_w, \qquad \text{together with} \qquad 1 = s_w + s_n,$$

modeling that the two fluids together fill the void completely. There exists a unique solution to the above equations together with appropriate boundary and initial conditions for incompressible fluids (see [Che2001, Che2002]). It is convenient to consider the *global pressure* formulation of the above equations, for simplicity under the assumption of constant densities (originating from [Ant1972, CJ1986]). For $\alpha = w, n$, we define the *phase mobilities* $\lambda_\alpha := \kappa_{r\alpha}\nu_\alpha^{-1}$, the *total mobility* $\lambda := \lambda_w + \lambda_n$, the *fractional flow* functions $f_\alpha := \lambda_\alpha\lambda^{-1}$ and the *total velocity* $u := u_w + u_n$. With $s := s_w$ we define the *global pressure* $p := p_n + \int_{p_c(s)} f_w(p_c^{-1}(\xi))\,\mathrm{d}\xi$. The above six equations can be transformed into the following three equations for p, u and s:

$$-\nabla\cdot(\lambda\kappa\nabla p) = (\rho_w f_w + \rho_n f_n)g\nabla z - \tfrac{\partial\phi}{\partial t} + \tfrac{q_w}{\rho_w} + \tfrac{q_n}{\rho_n} \tag{1.0.1a}$$

$$u = -\lambda\kappa\big(\nabla p - (\rho_w f_w + \rho_n f_n)g\nabla z\big) \tag{1.0.1b}$$

$$\phi\tfrac{\partial s}{\partial t} + S\tfrac{\partial\phi}{\partial t} + \nabla\cdot\big(\lambda_n\kappa f_w \tfrac{dp_c}{ds}\nabla s\big) = \nabla\cdot\big(\lambda_n\kappa f_w\big((\rho_n - \rho_w)g\nabla z\big) - f_w u\big) + \tfrac{p_w}{\rho_w} \tag{1.0.1c}$$

A popular method for the time discretization of the above equations is the *implicit pressure, explicit saturation* (IMPES) scheme, first introduced by [SZC1959, SG1961].

Using IMPES, Equations 1.0.1 are decoupled by implicitely solving for the pressure (1.0.1a) and computing the velocity (1.0.1b) in each time step, using quantities from the previous time step, and an explicit solve for the saturation (1.0.1c) afterwards. Considering the whole simulation process, the pressure equation (1.0.1a) has to be solved in each time step, which is responsible for most of the computational effort.

The permeability field in (1.0.1a) can be highly heterogeneous and rapidly varying on small spatial scales, while the computational domain may be very large (see the experiments in chapter 4). Thus, (1.0.1a) can be interpreted as a *multiscale* problem.

On the other hand, since a different saturation enters in the total mobility λ in each time step, we can also view (1.0.1a) as a *parametric* problem, with a parameter dependent data function λ.

The present work deals exclusively with approximations of such elliptic problems. Accordingly, this chapter is organized as follows. In Section 1.1, we introduce the general definition of an elliptic problem and discuss grid-based numerical approximation techniques based on Finite Element methods. We establish the notion of *accuracy* and *efficiency*, where the former is associated with error control and the latter is associated with adaptive methods based on reliable and localizable a posteriori error estimates.

We present multiscale problems, which are associated with highly oscillating or heterogeneous data functions, in Section 1.2. Though particularly challenging to approximate, such problems arise in almost all areas of natural sciences. We give a brief overview of numerical methods which are specifically tailored to multiscale problems in Section 1.2.2 and discuss the extend, to which accuracy and efficiency can be obtained for such problems.

In Section 1.3, we consider elliptic problems which are parameterized by a low-dimensional input vector and discuss the applicability of model reduction techniques. Model order reduction allows for an increased efficiency in many circumstances, though at the price of giving up full control over accuracy. As a particular model reduction strategy, we present *reduced basis* (RB) methods.

Finally, a definition of elliptic parametric multiscale problems is given in Section 1.4. The numerical approximation of such problems is particularly interesting, since the computational demand of straightforward attempts easily exceeds available resources. We give a brief presentation of the *localized reduced basis multiscale method* (LRBMS), which is covered in detail in Chapters 2 and 4, for the efficient and accurate approximation of parametric multiscale problems (such as Equation 1.0.1a in the context of IMPES).

1.1 Elliptic problems and grid-based approximations

Let $\Omega \subset \mathbb{R}^d$, for $d = 1, 2, 3$, be a bounded connected domain with Lipschitz boundary, let $H^1(\Omega)$ denote the Sobolev space of weakly differentiable functions over Ω and $H^1_0(\Omega) \subset H^1(\Omega)$ its elements that vanish on the boundary of Ω in the sense of traces.

1.1.1 Elliptic problems

We consider a linear functional $l : H^1(\Omega) \to \mathbb{R}$ and a bilinear form $b : H^1(\Omega)\times H^1(\Omega) \to \mathbb{R}$ with the following properties: we require l to be continuous, that is $|l(q)|/\|q\|_{H^1(\Omega)} < \infty$ for all $q \in H^1(\Omega)$, and we require b to be *elliptic*, i.e., to be continuous and coercive over $H^1_0(\Omega)$, that is

$$|b(p,q)| \leq C_b \,\|p\|_{H^1(\Omega)}\,\|q\|_{H^1(\Omega)} \qquad \text{for all } p,q \in H^1(\Omega) \text{ and} \tag{1.1.1}$$

$$c_b\,\|q\|^2_{H^1_0(\Omega)} \leq |b(q,q)| \qquad \text{for all } \quad q \in H^1_0(\Omega), \tag{1.1.2}$$

respectively, with positive constants $c_b > 0$ and $C_b \geq c_b$. Owing to the Lax-Milgram Lemma there exists a unique solution of the following elliptic problem, presuming the above requirements (see, for instance [Cia1978, Theorem 1.3.1]).

Definition 1.1.1 (Elliptic problem)**.** *Given b and l as defined above, find $p \in H^1_0(\Omega)$, such that*

$$b(p,q) = l(q) \qquad \textit{for all } q \in H^1_0(\Omega).$$

The stationary heat equation may serve as a model problem.

Example 1.1.2 (Stationary heat equation)**.** *Let $f \in L^2(\Omega)$ be bounded, let $\lambda \in L^\infty(\Omega)$ be strictly positive and let $\kappa \in [L^\infty(\Omega)]^{d\times d}$ be symmetric and positive definite, such that $\lambda\kappa \in [L^\infty(\Omega)]^{d\times d}$. Here, the product $\lambda\kappa$ models the thermal conductivity of a medium that is cooled at the boundary and f models a collection of heat sinks and sources. We are looking for a temperature distribution $p \in H^1_0(\Omega)$ in its equilibrium, such that $-\nabla\cdot(\lambda\kappa\nabla p) = f$ in a weak sense in $H^1_0(\Omega)$.*

The stationary heat equation results from considering conservation laws involving the heat flux $u := -\lambda\kappa\nabla p$ (which will be an important quantity in the context of flow problems and error estimation further below). We can see that the above example is an elliptic problem by setting

$$b(p,q) := \int_\Omega \big(\lambda\kappa\nabla p\big)\cdot\nabla q \,\mathrm{d}x \qquad \text{and} \qquad l(q) := \int_\Omega f q \,\mathrm{d}x, \tag{1.1.3}$$

respectively. The continuity of both l and b and the coercivity of b follow from the properties of f, λ and κ.

For arbitrary domains and data functions, we cannot solve Problem 1.1.1 directly and therefore need to rely on efficient and accurate approximations of such problems (we shall detail the precise meaning of “accurate” and “efficient” further below). There exist many techniques to obtain approximate solutions, such as wavelet methods [Urb2009], spectral methods [GO1977], radial basis functions [Buh2000], or other grid-free methods. In this work, we consider grid-based discretizations, such as *Finite Volume* (FV) and continuous or *discontinuous Galerkin* (DG) approximations.

1.1.2 Grid-based numerical approximations with Finite Element methods

Generally speaking, we consider *Finite Element methods* (FEM), which are widely used and thoroughly studied in engineering as well as academic contexts (see for instance the classical work [Cia1978]). Any such discretization is based on a partition of the computational domain by a *grid* τ_h into a finite number of non-overlapping elements $t \subset \Omega$ of simple shape, such that $\overline{\cup_{t \in \tau_h} t} = \Omega$, and an approximation of the infinite-dimensional space $H^1(\Omega)$ by a finite-dimensional (sub-)space $Q_h^k(\tau_h)$ of some local polynomial order $k \in \mathbb{N}$.[1] In addition, a discretization provides approximations of the bilinear form b and the linear functional l, denoted by b_h and l_h, respectively (which often consist of replacing integrals by numerical quadratures). If carefully constructed, these discrete counterparts inherit the properties of b and l and, as a result, there also exists a unique solution to the following discrete problem.

Definition 1.1.3 (Discrete elliptic problem)**.** *Given $Q_h^k(\tau_h)$, b_h and l_h by a suitable discretization, find $p_h \in Q_h^k(\tau_h)$, such that*

$$b_h(p_h, q_h) = l_h(q_h) \qquad \textit{for all } q_h \in Q_h^k(\tau_h).$$

We call an approximation *accurate*, iff, for a prescribed tolerance $\Delta > 0$,

$$\|p - p_h\|_* \le \Delta \tag{1.1.4}$$

in a suitable norm $\|\cdot\|_*$; this will often be the $L^2(\Omega)$- or the $H^1(\Omega)$-(semi-)norm, or the norm induced by the bilinear form b. On the other hand, we call an approximation *efficient*, iff the computational cost of accurately computing p_h is optimal (which may depend on the circumstances or posed requirements, see further below). We know from a priori theory that FE methods have the potential to be accurate in the sense that the *discretization error* arising in (1.1.4) is bounded as in

$$\|p - p_h\|_{L^2(\Omega)} \lesssim h^{k+1} \qquad \text{or} \qquad \|p - p_h\|_{H^1(\Omega)} \lesssim h^k, \tag{1.1.5}$$

respectively, for a discretization of order k (under mild assumptions on the regularity of p and the grid, compare [Cia1978, Chapters 2 and 3, in particular Theorems 2.4.1, 3.2.2 and 3.2.4]). Here, h denotes a typical width of an element of τ_h (see also Section 2.1) and $\lesssim$ denotes a relationship defined as $a \lesssim b :\Leftrightarrow a \le C\, b$ with a constant $C > 0$. The a priori estimate (1.1.5) justifies the use of FE methods in general. However, while it ensures that any prescribed accuracy can be reached in theory, it does not tell us how to choose τ_h (and thus $Q_h^k(\tau_h)$, b_h and l_h) to reach a desired accuracy in practice. We could, for instance, construct an algorithm that uniformly refines the grid τ_h and iteratively computes approximate solutions associated with each refinement. But we would have no

[1]We call the discretization a *conforming* one, if $Q_h^k(\tau_h) \subset H^1(\Omega)$, else a *non-conforming* one (compare Section 3.1.1.1). For ease of notation we presume a conforming discretization throughout this chapter. This allows us, for instance, to compare the approximate with the true solution, as in (1.1.4), in a meaningful way. We also restrict ourselves to approximation spaces with uniform local polynomial degree k, in contrast to locally varying polynomial degree.

rigorous means to decide when to stop the algorithm and, more importantly, no means to assess the quality of the computed approximations. In general, no *a priori*[2] information can guide the construction of an accurate discretization, let alone an efficient one.

This is the domain of *a posteriori*[3] analysis (for an overview see [Ver1996, Ver2013]). If successfully applied, it yields an estimate on the discretization error,

$$\|p - p_h\|_* \leq \eta_h(p_h),$$

for an already computed approximation $p_h \in Q_h^k(\tau_h)$. Given such an a posteriori error estimate η_h we can assess the quality of the computed approximation without knowledge of the true solution p. Therefore, we could enhance the above mentioned algorithm by computing such an estimate in each iteration of the algorithm and to derive an appropriate stopping criterion: namely, to stop the algorithm, once $\eta_h(p_h) \leq \Delta$. Such an algorithm would therefore yield an accurate approximation in the sense of (1.1.4) for any prescribed tolerance (we refer to [Ver1996, Ver2013] for an overview). Most likely, however, the computational cost of computing this accurate approximation would not be optimal (and the discretization thus not efficient), in the sense that the uniformly refined grid would be too fine in parts of the domain.

Adaptive Finite Element methods (AFEM) try to minimize the computational cost of the approximation by refining only relevant parts of the grid, for instance those parts that are associated with the largest estimated error.[4] To identify these parts of the grid they utilize *localized* a posteriori error estimates, which are composed of local error indicators η_h^t for each grid element $t \in \tau_h$, such that

$$\eta_h(p_h)^2 \lesssim \sum_{t \in \tau_h} \eta_h^t(p_h)^2. \tag{1.1.6}$$

There exist many classes of problems where such an iterative and adaptive *solve* $\rightarrow$ *estimate* $\rightarrow$ *mark* $\rightarrow$ *refine* procedure was shown to produce a series of approximations that converge to the desired solution. In particular, such an adaptive algorithm produces an accurate approximation for any prescribed tolerance in finite time. We refer to [BV1984], where first results for one space dimension were established, to [Doe1996], where the well-known *Dörfler*-marking strategy was proposed, and to [MNS2002], where a refinement strategy was proposed which removes the need for any a priori knowledge or user input.

If the a posteriori estimate serves additionally as a lower bound,

$$\eta_h(p_h) \lesssim \|p - p_h\|_* \leq \eta_h(p_h), \tag{1.1.7}$$

[2] "a priori" as in: before computing the approximation.

[3] "a posteriori" as in: after computing the approximation.

[4] As we do not consider approximation spaces with varying local polynomial degree, we only discuss the h-adaptive FEM. A similar notion of adaptivity is realized by locally adapting the polynomial degree of the approximation space (p-adaptive FEM [BSK1981]), or both the local polynomial degree and the grid (hp-adaptive FEM [BD1981]). For an overview we refer to [Ngu2010] and the references therein.

it induces a norm equivalent to $\|\cdot\|_*$. Such *reliable* estimates can be used to ensure that the computational cost of an adaptive algorithm is optimal (for instance by an additional adaptive coarsening of the grid, as proposed in [BDD2004]).

To conclude, adaptive FE discretizations based on reliable and localizable a posteriori error estimates are the de-facto standard to accurately and efficiently compute approximations of Problem 1.1.1 for arbitrary domains and data functions (at least within the classes of approximation algorithms considered here).

To summarize: we introduced elliptic problems and briefly mentioned grid-based approximation techniques (which are covered in detail in Section 3.1.1.1). We established the notion of *accuracy*, as full control over the approximation error for any prescribed tolerance, and the notion of *efficiency*, which is related to the computational cost of computing such accurate approximations. As a key ingredient for either concept, we briefly introduced *a posteriori* error estimation techniques, yielding reliable and localizable estimates on the discretization error. Such estimates η_h are employed to steer fully adaptive algorithms, yielding accurate as well as efficient approximations.

1.2 Multiscale problems and numerical multiscale methods

The numerical treatment of elliptic problems, such as Problem 1.1.1, becomes increasingly difficult with the increasing complexity of the data functions involved. This is for instance the case, if the data functions exhibit a high contrast or vary on several spatial scales: on a *coarse scale* associated with $|\Omega| = \mathcal{O}(1)$ and on a *fine scale* associated with a multiscale parameter $0 < \varepsilon \ll |\Omega|$.

1.2.1 Elliptic multiscale problems

In order to give a more rigorous description, we revisit the a priori results on the approximation quality of grid-based discretizations. In particular, we study the role of the data functions in the hidden constant in (1.1.5), where we stated the following a priori bound on the discretization error:

$$\|p - p_h\|_{H^1(\Omega)} \leq C\, h^k. \tag{1.2.1}$$

To investigate the effect of the data functions on the constant $C > 0$ in the above bound let us recall that we obtain the bound by first using Céa's Lemma [Cia1978, Theorem 2.4.1], which bounds the discretization error (on the left hand side) by the best-approximation error of the Finite Element space,

$$\|p - p_h\|_{H^1(\Omega)} \leq C_b/c_b \inf_{q_h \in Q_h^k(\tau_h)} \|p - q_h\|_{H^1(\Omega)}, \tag{1.2.2}$$

where c_b and C_b denote the coercivity and continuity constants of the bilinear form b (Equations 1.1.1 and 1.1.2). We further estimate the approximation error by employing local interpolation properties of the Finite Element space $Q_h^k(\tau_h)$, yielding local bounds of the right hand side in Equation 1.2.2 (under mild assumptions on the data functions, the grid and the regularity of the solution, see [Cia1978, Chapter 3, in particular Sections 3.1 and 3.2]), ultimately yielding

$$\|p - p_h\|_{H^1(\Omega)} \lesssim C_b/c_b\, |p|_{H^{k+1}(\Omega)}\, h^k, \tag{1.2.3}$$

with the relationship $\lesssim$ from the previous section. We obtain the desired result (1.2.1) since p is bounded (compare the proof of the Lax-Milgram Lemma [Cia1978, Theorem 1.3.1]).

As long as the bilinear form b and the linear functional l (and their discrete counterparts) fulfill the requirements stated in the previous section, the above analysis is valid and states the potential accuracy of Finite Element approximations (compare the discussion in the previous section). From (1.2.3) we know that we can reach any prescribed accuracy if h is chosen small enough, since the constants C_b/c_b and $|p|_{H^{k+1}(\Omega)}$ do not depend on h. They do, however, depend on the data functions that constitute the elliptic problem and we highlight two scenarios where this might be problematic in practical computations:

(*i*) Given a bilinear form b as in (1.1.3), we can think of the coercivity and continuity constants c_b and C_b as the minimum and maximum eigenvalues of the inducing data functions, $\lambda\kappa$. If these data functions exhibit a *high contrast* (compare the Spe10 data functions in Chapter 4), namely if C_b is of order $|\Omega| = \mathcal{O}(1)$, while c_b is of order $\varepsilon > 0$, for a very small $\varepsilon \ll |\Omega|$, we actually have $C_b/c_b = \mathcal{O}(\varepsilon^{-1})$ in (1.2.3).

(*ii*) There also exist many scenarios where the data functions exhibit strong *oscillations* on a fine scale $0 < \varepsilon \ll |\Omega|$ and the corresponding solution p shows similar oscillations (think of $\sin(x/\varepsilon)$). Consequently, the derivatives of p are of order ε^{-l} for some $l \geq 1$, and we have $|p|_{H^{k+1}(\Omega)} = \mathcal{O}(\varepsilon^{-l})$ in (1.2.3).

Neither of the above situations is troublesome in theory and does not impact the existence and uniqueness of the solution p. They do, however, impact the feasibility of grid-based approximation techniques, as we obtain the following estimate on the discretization error (in the presence of multiscale phenomena) from (1.2.3):

$$\|p - p_h\|_{H^1(\Omega)} \lesssim \frac{h^k}{\varepsilon^l} \overset{!}{<} \Delta, \tag{1.2.4}$$

for some $l \geq 1$ (a similar result holds for other norms). We have to interpret this result in the following way:

In the context of multiscale phenomena, grid-based approximations can only be accurate (in the sense of Section 1.1) if the grid is fine enough, $h < \mathcal{O}(\varepsilon)$.

We are now in the position to define elliptic multiscale problems.

Definition 1.2.1 (Elliptic multiscale problem)**.** *With the notation from Definition 1.1.1, let l be continuous and let $\varepsilon > 0$ denote a multiscale parameter, such that $\varepsilon \ll |\Omega|$. Let further $b_\varepsilon : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ denote an elliptic bilinear form in the sense of* (1.1.1) *and* (1.1.2)*. We call the problem of finding $p_\varepsilon \in H^1_0(\Omega)$, such that*

$$b_\varepsilon(p_\varepsilon, q) = l(q) \qquad \text{for all } q \in H^1_0(\Omega), \tag{1.2.5}$$

an elliptic multiscale *problem, if b_ε exhibits* high contrast *or strong* oscillations *(in the sense of* (*i*) *and* (*ii*) *above).*

The numerical approximation of Problem 1.2.1 is a computationally challenging task. In particular, it is not clear whether the notions of accuracy and efficiency established in Section 1.1 are suitable for the assessment of approximations of multiscale problems. While (1.2.4) states that traditional grid-based discretization methods can be accurate (in the sense that the discretization error can be controlled up to any prescribed tolerance Δ by decreasing h), it also states that the grid is therefore required to resolve all features associated with the fine scale ε.

Since the multiscale Problem 1.2.1 is an elliptic problem in the sense of Definition 1.1.1, one can apply traditional (adaptive) Finite Element methods. However, the level of accuracy that can possible be obtained is bounded by the available computing power. While this is also true for smooth elliptic problems, it is particularly troublesome in the context of multiscale problems, as detailed above (recall that the size of the approximation space scales with the size of the grid, which in turn is proportional to ε^{-l}).

Remark 1.2.2 (Efficiency of traditional (A)FE methods)**.** *A typical implementation of a Finite Element method (compare Section 3.1.1.1) requires $\mathcal{O}(N)$ operations in order to assemble a linear system to solve the discrete elliptic Problem 1.1.3, with $N := \dim Q_h^k(\tau_h)$. Similarly, the requirements on the system's memory scale with N. Solving the resulting linear system requires $\mathcal{O}(N^2)$ operations using standard iterative solvers.*

For the remainder of this section, we present several techniques for the approximation of such computationally challenging problems, and discuss their accuracy and efficiency. We first mention two classical approaches, which are not necessarily related to multiscale problems: domain decomposition and multi-grid methods. Both rely on at least two partitions of the computational domain: a *fine* grid τ_h that accurately resolves all features of the PDE (as discussed above) and a *coarse* grid $\mathcal{T}_H$, such that $|\mathcal{T}_H| \ll |\tau_h|$. Often, the two grids are nested partitions of Ω and one refers to the elements $T \in \mathcal{T}_H$ of the coarse grid as subdomains (compare Section 2.1).

The idea of *domain decomposition* methods (DD) is to split the computational domain into several subdomains $T \in \mathcal{T}_H$ and to consider the elliptic Problem 1.1.1 on each subdomain separately. These local problems are then independently discretized with local fine grids τ_h^T covering the subdomain T (possibly including overlap) and local approximation spaces $Q_h^k(\tau_h^T)$, with $N^T := \dim Q_h^k(\tau_h^T)$. To obtain the solution of the original problem, these local problems are iteratively solved and coupled by using the neighboring solution as boundary values (we refer to [QV1999, TW2005] for an overview). While DD methods still require $\mathcal{O}(\sum_{T\in\mathcal{T}_H} N^T) \approx \mathcal{O}(N)$ operations to assemble the problem (and the memory requirements also scale with N), they only require $\mathcal{O}(\sum_{T\in\mathcal{T}_H} {N^T}^2)$ operations to solve the problem (since one solves a number of local systems instead of one global system). In addition, the local systems can be assembled and solved in parallel (to a certain degree).

Multi-grid methods, on the other hand, rely on the observation that the error associated with the fine grid τ_h (for instance during an iterative solving procedure) behaves much smoother on a coarser grid $\mathcal{T}_H$. Using restriction and prolongation operators with respect to the two grids and smoothing operators, it is possible to greatly speed up the iterative solution process. There exist many variants of multi-grid methods, in particular *algebraic multi-grid* (AMG) methods (we refer to [Wes1992] and [Sha2003] for an overview). Such AMG methods construct the aforementioned operators solely based on algebraic information assembled with respect to τ_h without actually requiring a coarse grid $\mathcal{T}_H$. While the memory requirements of multi-grid or AMG methods still scale with N and while they also require $\mathcal{O}(N)$ operations to assemble the problem, they are capable of solving the problem in $\mathcal{O}(N^l)$ operations, for $1 \leq l \leq 2$ (where $l = 1$ is usually only obtained for non-algebraic multi-grid methods).

Remark 1.2.3 (Improved efficiency)**.** *Both domain decomposition and multi-grid methods are well-established and powerful means to reduced the computational cost of solving the discrete problem. However, even in the best of circumstances, the associated computational cost still scales linearly with the size of the approximation space, N.*

To conclude, the size of the approximation spaces (and thus the computational demands) of traditional FE methods (with or without DD or AMG methods) scales with $N \approx \varepsilon^{-l}$ for some $l \geq 1$, which poses severe restrictions on the usefulness of these methods in the context of multiscale phenomena. The computational cost of approximating elliptic multiscale problems arbitrarily accurate (i.e., for arbitrary small $\Delta \approx \varepsilon$) thus easily exceeds available computing resources, despite the ever-growing power of state-of-the-art super-computers. As a consequence, the minimum possible accuracy is bounded by the available computational power.

Luckily, many multiscale problems exhibit a separation of scales and the solution of such problems can be written as an asymptotic expansion

$$p_\varepsilon(x) = p_0(x) + \varepsilon p_1(x, x/\varepsilon) + \dots, \tag{1.2.6}$$

where the leading term p_0 represents the coarse (or average) behavior of the solution and p_1 represents fine-scale features of the solution. In many applications, it is not required to have explicit knowledge about the full multiscale solution but one is rather interested in its coarse behavior p_0. It would thus be sufficient to have access to a coarse approximation p_H of p_0, associated with a coarse grid $\mathcal{T}_H$ of moderate size $|\mathcal{T}_H| \ll |\tau_h|$, such that

$$\|p_0 - p_H\|_* \lesssim H,$$

in some suitable norm, where $H > 0$ denotes the width of a typical element of $\mathcal{T}_H$. Equivalently speaking: one is satisfied to reach a coarse accuracy $\Delta \approx H$.

It is, however, particularly troublesome in the context of multiscale problems, that the straightforward approach of computing such a coarse approximation by merely using a coarser grid, cannot succeed (as it is well known that standard approximation techniques on coarse grids do not lead to an appropriate coarse approximation of the solution, compare [Ohl2005, Section 6.1]).

For some classes of multiscale problems, *homogenization* theory provides a means to study rapidly oscillating PDEs and their convergence behavior for $\varepsilon \to 0$, for instance by means of Γ-convergence [Gio1975, Gio1983], G-convergence [Spa1968], H-convergence [Tar1976, MT1997], by the energy method of Tartar [Tar1976], or by two-scale convergence [All1992, Ngu1989]; for an overview we refer to [Hor1997]. If the multiscale nature of the PDE allows for it, it is even possible to obtain an explicit form of the homogenized coarse scale equation for p_0, which is independent of ε and can thus be approximated with a computational cost which is also independent of ε. This can for instance be the case if the multiscale nature is periodic or stochastic.

For the more general case, an ever-growing class of numerical multiscale methods has emerged over the past decades,

> *to compute a coarse approximation of the solution of elliptic multiscale problems as efficiently as possible.*

A discussion of these methods is the focus of the remainder of this section.

1.2.2 Numerical multiscale methods

The idea of numerical multiscale methods is to make use of a possible separation of scales in the underlying problem to approximate p_0 (and sometimes also p_1 in view of Equation 1.2.6) in circumstances where an explicit form of a coarse equation for p_0 is not given. For many of these methods, convergence and a priori estimates can be given for periodic or stochastic problems, while numerical evidence of their successful application is available for the more general case. Some of these methods can be cast into a unified framework, a brief presentation of which is given (in the discrete setting), based on [Mal2011, HO2012, Ohl2012, HOS2014].

Given a fine and a coarse grid, τ_h and $\mathcal{T}_H$, and an approximation space $Q_h^k(\tau_h)$ as above, the two scales of the underlying problem are reflected in a possible decomposition of the approximation space into a coarse and a fine space, $Q_h^k(\tau_h) = Q_H^{\mathrm{c}}(\mathcal{T}_H) \oplus Q_h^{\mathrm{f}}(\tau_h)$, associated with the two grids. The coarse space $Q_H^{\mathrm{c}}(\mathcal{T}_H) \subset Q_h^k(\tau_h)$ is supposed to capture the coarse behavior of the solution with few Degrees of Freedom, $\dim Q_H^{\mathrm{c}}(\mathcal{T}_H) = \mathcal{O}(|\mathcal{T}_H|)$, while the fine space $Q_h^{\mathrm{f}}(\tau_h) \subset Q_h^k(\tau_h)$ arises as the kernel of a coarse projection operator $\Pi_H : Q_h^k(\tau_h) \to Q_H^{\mathrm{c}}(\mathcal{T}_H)$.

Consequently, we can decompose each function $p_h \in Q_h^k(\tau_h)$ into a coarse part $p_H \in Q_H^{\mathrm{c}}(\mathcal{T}_H)$ and its fine-scale correction $p_h^{\mathrm{f}} \in Q_h^{\mathrm{f}}(\tau_h)$: $p_h = p_H + p_h^{\mathrm{f}}$. We associate this decomposition with a *correction* operator $\mathcal{Q} : Q_H^{\mathrm{c}}(\mathcal{T}_H) \to Q_h^{\mathrm{f}}(\tau_h)$ and a *reconstruction* operator $\mathcal{R} : Q_H^{\mathrm{c}}(\mathcal{T}_H) \to Q_h^k(\tau_h)$. Given a coarse function $p_H \in Q_H^{\mathrm{c}}(\mathcal{T}_H)$, the first computes the fine scale correction $p_h^{\mathrm{f}} = \mathcal{Q}(p_H)$, while the latter reconstructs the full multiscale function $p_h = \mathcal{R}(p_H) = p_H + \mathcal{Q}(p_H)$.

Inserting this decomposition into (1.2.5) yields a coupled problem for $p_H \in Q_H(\mathcal{T}_H)$, such that

$$b_\varepsilon(\mathcal{R}(p_H), q_H) = l(q_H) \qquad \text{for all } q_H \in Q_H(\mathcal{T}_H), \tag{1.2.7a}$$

$$b_\varepsilon(\mathcal{Q}(p_H), q_h^{\mathrm{f}}) = l(q_h^{\mathrm{f}}) \qquad \text{for all } q_h^{\mathrm{f}} \in Q_h^{\mathrm{f}}(\tau_h). \tag{1.2.7b}$$

It is coupled through the reconstruction operator $\mathcal{R}$, which in turn is determined by the correction operator $\mathcal{Q}$. However, problem (1.2.7) can be decoupled by a careful choice of the correction operator (usually by means of localization) to actually obtain a low-dimensional problem for the coarse approximation p_H.

Several numerical multiscale methods can be derived from the above framework by specifying the coarse projection operator, the reconstruction and the correction operator. We briefly discuss the most common methods but refer to [Mal2011, HO2012, Ohl2012, HOS2014] for a detailed discussion, in particular regarding the unified framework above.

We mention two approaches that allow to obtain a coarse approximation p_H with a computational complexity independent of ε under certain circumstances. The first is the *variational multiscale method* (VMM), which was proposed in [Hug1995, HFMQ1998]. The VMM was the first method to be based on the abstract splitting of the approximation spaces into coarse and fine contributions, as mentioned above. The space of fine contributions is determined analytically by means of bubble or Greens functions to decouple (1.2.7).

Second, the *two-scale Finite Element method* for homogenization problems presented by [MS2002, SM2002] is based on a two-scale formulation of the problem and yields an ε-independent computational scheme for the approximation of p_H, if the solution of the problem exhibits a certain two-scale regularity.

In addition, there exist several numerical multiscale methods which aim at dealing with more general situations, however at the price of involving ε-dependent computations. They are based on the idea to construct the coarse approximation space by solutions of local problems (associated with the elements of the coarse grid) posed on the fine grid. Since artificial boundary conditions have to be provided for these local problems, they are often posed and solved on slightly larger domains (often called “oversampling” or “overlap”) and restricted afterwards to minimize the effects of the boundary conditions.

In this spirit, an adaptive variant of the VMM was proposed by means of energy-based a posteriori error estimates [LM2007, LM2009] and duality-based estimates [LM2005, LM2009a]. Similarly, a fully adaptive variant based on a discontinuous Galerkin formulation was proposed [EGMP2013] and complemented with a posterior error estimates [EGM2013].

In case of the *multiscale Finite Element Method* (MsFEM), the coarse space $Q_H(\mathcal{T}_H)$ is spanned by globally continuous shape functions obtained by solving local fine-scale problems. The MsFEM was first introduced in [HW1997], applied to the linear elliptic [HW1997, HWC1999, EH2009] as well as the nonlinear elliptic [EHG2004] setting, to two-phase flow in porous media [EH2007], stochastic porous media flow [AE2008] and uncertainty quantification [DEH2008]. There exist several variants of the MsFEM with and without overlap, and we refer to [HP2013] for an overview. Convergence of the MsFEM in the general homogenization setting was shown in [EP2003, EP2004, EP2005, Hen2012, SV2011] and a priori estimates in periodic or stochastic scenarios were established in [HW1997, HWC1999, EHW2000, EHG2004, CS2008]. Recently, also a posteriori estimates were established in [HOS2014] which allow to adaptively control the resolution of the fine and coarse grids as well as the size of the overlap.

Finally, the *multiscale Finite Volume* method (MSFV) is based on a piecewise constant non-conforming approximation space $Q^0_h(\tau_h)$. It was introduced in [JLT2003, JLT2004, JLT2006] and successfully applied to flow in porous media [HBHJ2008, HJ2009] with a focus on a physically meaningful reconstruction of the flux u mentioned in the previous section (see also [ALKK2009, NB2008] for an overview). An adaptive iterative variant was proposed in [HJ2011].

Remark 1.2.4 (Efficiency of multiscale methods)**.** *Numerical multiscale methods such as the adaptive VMM, the MsFEM and the adaptive MSFV yield low-dimensional problems for the approximation of the coarse solution behavior, with little to no structural assumptions on the multiscale nature of the problem. However, these methods assume that the fine scale ε can be resolved by a fine grid τ_h and the assembly of the low-dimensional problem still requires $\mathcal{O}(N)$ computations.*

It is not clear whether these multiscale methods are more efficient (regarding their computational complexity) then the classical approaches discussed above (such as DD or AMG methods), if applied to a single multiscale problem. In addition, the localization which is required for the decoupling of (1.2.7) requires the coarse grid to resolve certain features of the PDE, further limiting the usefulness of these methods (for instance in the presence of global conductivity channels, see [AEJ2008]). We mention two methods which each overcomes one these limitations.

As an extension of the adaptive VMM, a new multiscale method based on *localized orthogonal decomposition* techniques (LOD) was introduced in [MP2014]. The idea of the LOD is to construct a coarse approximation space $Q_H(\mathcal{T}_H)$ with very high approximation qualities (with respect to both scales), based on a Clement-type quasi-interpolation operator. The method was extended to multiscale boundary conditions in [HM2014] and to the semi-linear setting in [HMP2014]. While the LOD also requires computations on a fine grid τ_h, it does not make any structural assumptions on the multiscale nature of the problem and is at the same time capable of dealing with high-conductivity channels connecting domain boundaries.

Lastly we mention the *heterogeneous multiscale method* (HMM), which was originally proposed in [EE2003, EE2003a, EE2005] including a priori analysis. While the HMM can be cast into the abstract framework presented above, it fundamentally differs from other methods regarding its computational requirements. Within the framework of the HMM, one also solves a low-dimensional problem for the coarse behavior of the solution. However, whenever the evaluation of a multiscale data function in a quadrature point on the coarse grid is required, a local fine grid is constructed around the quadrature point and a local fine-scale problem is solved to determine the effective value of the data function. The HMM was shown to correspond to a direct approximation of the two-scale homogenized problem in the classical homogenization setting by [Ohl2005]. First a posteriori error estimates were derived in [Ohl2005] based on this identification and later on complemented by [HO2009, HO2010, AN2009, AN2011]. While the HMM also requires fine-grid computations, it does so only in a small area proportional to ε.

To conclude, the HMM is the only numerical multiscale method which approximates the coarse behavior of a multiscale solution with a computational complexity independent of ε (to the best of our knowledge).

To summarize: we introduced a specific class of elliptic problems, which arise in the context of highly oscillating or heterogeneous data functions and discussed the limitations of traditional approximation techniques in this context. In particular, we discussed the impact of limited available computing power on the level of accuracy which can be obtained. We presented problems which exhibit scale separation and thus allow for an approximation of the coarse solution behavior with a computational complexity that does not depend on the multiscale features, and discussed homogenization techniques as well as numerical multiscale methods, which can allow for such a coarse approximation in an efficient manner.

1.3 Parametric problems and model order reduction

Consider elliptic problems, where all data functions and the domain involved may depend on a parameter vector $\boldsymbol{\mu} \in \mathbb{R}^\rho$, for some $\rho \in \mathbb{N}$. In this scenario, we are not interested in a single solution, but in many solutions associated with a set of admissible parameters $\mathcal{P} \subseteq \mathbb{R}^\rho$ (called the *parameter space*).

1.3.1 Elliptic parametric problems

For convenience, we consider a common space for all solutions to all parameters and presume that parametric domains are transformed to a reference domain and that an according geometry transformation is encoded in the bilinear form (compare [MMPR2001, Dro2009, DHO2009]). For simplicity, we do not consider a parameter dependent right hand side or parametric boundary values and refer to [PR2006] for the treatment of such problems. Hence, we consider a nonparametric linear functional l as in Section 1.1 and a parametric bilinear form $b : \mathcal{P} \to [H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}]$, such that for all parameters $\boldsymbol{\mu} \in \mathcal{P}$ the resulting nonparametric bilinear form $b(\cdot, \cdot; \boldsymbol{\mu}) : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ is continuous and coercive in the sense of (1.1.1) and (1.1.2). Thus, there exists a unique solution of the following problem for each $\boldsymbol{\mu} \in \mathcal{P}$.

Definition 1.3.1 (Elliptic parametric problem)**.** *Given $\mathcal{P}$ and b as defined above and l as defined in Section 1.1, find $p(\boldsymbol{\mu}) \in H^1_0(\Omega)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that*

$$b\big(p(\boldsymbol{\mu}), q; \boldsymbol{\mu}\big) = l(q) \qquad\qquad \textit{for all } q \in H^1_0(\Omega). \tag{1.3.1}$$

An example for a parametric problem is, for instance, given by a parametric variant of Example 1.1.2, where the thermal conductivity is locally controlled by a parameter component.

Example 1.3.2 (Thermal block problem)**.** *Let Ω be partitioned into $\varXi \in \mathbb{N}$ non-overlapping subdomains $\Omega_\xi \subset \Omega$, such that $\overline{\cup_{\xi=0}^{\varXi-1} \Omega_\xi} = \Omega$. Let f and κ be as in Example 1.1.2 and let λ be given piecewise constant in each subdomain: $\lambda : \mathcal{P} \to L^\infty(\Omega)$, $\boldsymbol{\mu} \mapsto \lambda(\cdot; \boldsymbol{\mu})$, with $\lambda(x; \boldsymbol{\mu}) := \boldsymbol{\mu}_\xi$ if $x \in \Omega_\xi$ and $\lambda(x; \boldsymbol{\mu}) := 0$ else, for $0 \leq \xi < \varXi$. For $\boldsymbol{\mu} \in \mathcal{P} := \{\boldsymbol{\mu} \in \mathbb{R}^\varXi \,|\, 0 < \boldsymbol{\mu}_{\min} \leq \boldsymbol{\mu} \leq \boldsymbol{\mu}_{\max}\}$, for some fixed $\boldsymbol{\mu}_{\min}, \boldsymbol{\mu}_{\max} \in \mathbb{R}^\varXi$, find $p(\boldsymbol{\mu}) \in H^1_0(\Omega)$, such that $-\nabla \cdot \big(\lambda(\boldsymbol{\mu})\kappa \nabla p(\boldsymbol{\mu})\big) = f$ in the weak sense in $H^1_0(\Omega)$.*

In this example, each parameter $\boldsymbol{\mu} \in \mathcal{P}$ corresponds to a specific thermal conductivity $\lambda(\boldsymbol{\mu})\kappa$ and yields a nonparametric elliptic problem as in Example 1.1.2. We can see that this example is an elliptic parametric problem by setting

$$b(p, q; \boldsymbol{\mu}) := \int_\Omega \big(\lambda\kappa\nabla p(\boldsymbol{\mu})\big) \cdot \nabla q \,\mathrm{d}x \tag{1.3.2}$$

in Problem 1.3.1. The continuity and coercivity of b for any $\boldsymbol{\mu} \in \mathcal{P}$ follow from the properties of λ.

We are interested in accurate and efficient approximations of elliptic parametric problems, such as the thermal block problem, for many parameters. To be more precise: given the input/output map

$$IO : \mathcal{P} \to H_0^1(\Omega), \qquad \boldsymbol{\mu} \mapsto p(\boldsymbol{\mu}), \text{ where } p(\boldsymbol{\mu}) \text{ is the solution of (1.3.1)}, \tag{1.3.3}$$

we are interested in an efficient evaluation of an accurate approximation of IO (since we do not have direct access to the solutions p, compare Section 1.1).

To properly define a discrete input/output map IO_h as an approximation of IO, we need to carefully examine the possible range of such a map (as an approximation of the range of IO, which is $H_0^1(\Omega)$). Each approximation of a solution of Problem 1.3.1 for a single parameter $\boldsymbol{\mu} \in \mathcal{P}$ is in general associated with a different grid $\tau_h(\boldsymbol{\mu})$ and a corresponding approximation space $Q_h^k\big(\tau_h(\boldsymbol{\mu})\big)$ of order k, which are required to ensure the accuracy of that specific approximation (compare AFEM in Section 1.1). We thus denote by $Q_h^k\big(\tau_h(\mathcal{P})\big)$ a common space for all approximations, where $\tau_h(\mathcal{P})$ is defined as the coarsest grid, such that $Q_h^k\big(\tau_h(\boldsymbol{\mu})\big) \subset Q_h^k\big(\tau_h(\mathcal{P})\big)$ for all $\boldsymbol{\mu} \in \mathcal{P}$.

Remark 1.3.3. *Though easily defined, the construction of $Q_h^k\big(\tau_h(\mathcal{P})\big)$ is not feasible in practice. Apart from the computational challenge of constructing $Q_h^k\big(\tau_h(\mathcal{P})\big)$ for a large number of parameters, it is usually the case that $\mathcal{P}$ is not finite.*

Nevertheless, the common approximation space $Q_h^k\big(\tau_h(\mathcal{P})\big)$ is required for the definition of a suitable approximation of IO. Therefore, we presume that a suitable discretization of Problem 1.3.1 is available, yielding discrete counterparts of b and l, denoted by b_h and l_h, respectively (compare Section 1.1), such that there exists a unique solution of the following problem for each $\boldsymbol{\mu} \in \mathcal{P}$.

Definition 1.3.4 (Discrete elliptic parametric problem)**.** *Given $Q_h^k\big(\tau_h(\mathcal{P})\big)$, b_h and l_h by a suitable discretization, find $p_h(\boldsymbol{\mu}) \in Q_h^k\big(\tau_h(\mathcal{P})\big)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that*

$$b_h\big(p_h(\boldsymbol{\mu}), q_h; \boldsymbol{\mu}\big) = l_h(q_h) \qquad\qquad \textit{for all } q_h \in Q_h^k\big(\tau_h(\mathcal{P})\big). \tag{1.3.4}$$

With this discrete problem as an approximation of Problem 1.3.1, we define the discrete input/output map

$$IO_h : \mathcal{P} \to Q_h^k\big(\tau_h(\mathcal{P})\big), \qquad \boldsymbol{\mu} \mapsto p_h(\boldsymbol{\mu}), \text{ where } p_h(\boldsymbol{\mu}) \text{ is the solution of (1.3.4)},$$

as an approximation of IO in (1.3.3). By construction, the evaluation of $IO_h(\boldsymbol{\mu})$ yields an accurate approximation of the evaluation of $IO(\boldsymbol{\mu})$ for any parameter $\boldsymbol{\mu} \in \mathcal{P}$ (since the approximation space $Q_h^k\big(\tau_h(\mathcal{P})\big)$ was chosen accordingly). Despite the infeasibility of an actual construction of $Q_h^k\big(\tau_h(\mathcal{P})\big)$ (see Remark 1.3.3), we can formulate a practical algorithm to evaluate IO_h arbitrarily accurate (see Algorithm 1.3.5), in the sense that for any prescribed tolerance $\Delta > 0$,

$$\|IO(\boldsymbol{\mu}) - IO_h(\boldsymbol{\mu})\|_* \leq \Delta \qquad\qquad \text{for all } \boldsymbol{\mu} \in \mathcal{P}_{\text{int.}}, \tag{1.3.5}$$

in a suitable norm $\|\cdot\|_*$. Recall that since, $IO(\boldsymbol{\mu}) = p(\boldsymbol{\mu})$ and $IO_h(\boldsymbol{\mu}) = p_h(\boldsymbol{\mu})$, the above difference corresponds to the discretization error from Section 1.1.

We therefore collect all parameters for which we would like to evaluate IO_h in a finite set of *parameters of interest*, $\mathcal{P}_{\text{int.}} \subset \mathcal{P}$. Additionally, we presume we are given a reliable and localizable a posteriori error estimate on the discretization error, as in Equations 1.1.7 and 1.1.6. By construction, the following algorithm yields accurate approximations of IO in the sense of (1.3.5). If we consider each parameter $\boldsymbol{\mu} \in \mathcal{P}_{\text{int.}}$ separately, this algorithm is also efficient in the sense of Section 1.1.

Algorithm 1.3.5 AFEM for elliptic parametric problems.

```
Input: $\mathcal{P}_{\text{int.}} \subseteq \mathcal{P}$, $\Delta > 0$, $\eta_h$
Output: $p_h(\boldsymbol{\mu})$, for all $\boldsymbol{\mu} \in \mathcal{P}_{\text{int.}}$, such that $\|p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\|_* \leq \Delta$
  for $\boldsymbol{\mu} \in \mathcal{P}$ do
    Use adaptive Finite Element methods (compare Section 1.1) to construct a grid
    $\tau_h(\boldsymbol{\mu})$ and to compute an approximate solution $p_h(\boldsymbol{\mu}) \in Q_h^k(\tau_h(\boldsymbol{\mu}))$, such
    that $\|p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\|_* \leq \eta_h(p_h(\boldsymbol{\mu}); \boldsymbol{\mu})$.
  end for
```

There exist circumstances, however, where this notion of efficiency is not suitable. We present two such scenarios in the context of parametric problems, following the overview of [RHP2008].

In *real-time* contexts, one is interested in very quick evaluations of the input/output map IO_h, or rather in a quantity derived from its output (e.g., some functional acting on the solution associated with a parameter). In order to achieve this near real-time evaluation one is willing to spend a considerable amount of computing power in advance. An example for such a scenario is the use of low-end devices (such as tablet computers, smartphones or embedded devices) to quickly access results in the context of decision making while extensive numerical simulations have been prepared in advance, possibly on a high performance cluster.

In *many-query* contexts, one is interested in an evaluation of the input/output map IO_h (or a derived quantity) for a very large number of parameters, that is overall less costly than Algorithm 1.3.5. An example for such a scenario is an optimization or Monte-Carlo procedure (for instance in the context of uncertainty quantification) which requires many subsequent solutions of Problem 1.3.4.

Both scenarios induce a different meaning of "efficiency". In a real-time context one is interested in *online* efficiency, namely to split the evaluation of IO_h into a computationally expensive "offline" part and a computationally very cheap "online" part. In a many-query context, on the other hand, one is interested in *overall* efficiency, namely to evaluate IO_h for a large number of parameters, with an overall computational cost that is lower than the cost of the procedure in Algorithm 1.3.5. Both notions of efficiency are the motivation for *model order reduction* (MOR).

The idea of model order reduction for parametric problems is to make use of some regularity of the input/output map IO_h (in other words: to exploit "opportunities" [RHP2008, p. 230] encoded in Problem 1.3.4) in order to achieve either online or overall efficiency. There exists a large class of established model reduction approaches, for instance POD- or snapshot-based [Sir1987, BHL1993, WP2002, LV2014, HO2014] and interpolation-based [GAB2008, AF2011, BBBG2011] approaches, and we refer to [BBH+2015] for an overview.

1.3.2 Model order reduction with reduced basis methods

We consider model reduction by projection-based *reduced basis* (RB) methods, which can be applied in real-time as well as many-query contexts and allow for an elegant mathematical setting. RB methods were first introduced several decades ago [FM1971, ASB1978, FR1983, Por1985] but have only gained the interest of a larger community in the past 15 years. They have been applied to a large variety of problems and we refer to [PR2006, QRM2011, RHP2008] for an overview. Beside the Lagrangian RB ansatz which we consider, there also exist Taylor, Hermit or least-squares RB methods (compare [Por1985, RHP2008]).

Consider the range of all possible solutions of Problem 1.3.4, or equivalently, the image of IO_h:

$$IO_h(\mathcal{P}) = \big\{p_h(\boldsymbol{\mu}) \in Q_h^k\big(\tau_h(\mathcal{P})\big), \text{ solution of (1.3.4)} \,\big|\, \boldsymbol{\mu} \in \mathcal{P}\big\}. \tag{1.3.6}$$

As we shall argue below, there exist many circumstances, where $IO_h(\mathcal{P})$ is of considerably lower dimension than $Q_h^k\big(\tau_h(\mathcal{P})\big)$. The idea of RB methods is to find a low-dimensional *reduced space* $Q_{\text{red}} \subset Q_h^k\big(\tau_h(\mathcal{P})\big)$, which is a good approximation of $IO_h(\mathcal{P})$.

We postpone the question of how to find a “good” reduced space until Section 1.3.2.2 and continue with the definition of the reduced problem to establish the notation, which is required for the discussion. For the time being, we can think of the reduced space as being spanned by a *reduced basis* ϕ_{red}, which is composed of solutions of Problem 1.3.4 for selected parameters.

Presuming we are given a reduced space $Q_{\text{red}} \subset Q_h^k\big(\tau_h(\mathcal{P})\big)$, the reduced problem is given by a Galerkin projection of Problem 1.3.4 onto the reduced space; existence and uniqueness of solutions of the following problem follow directly from the existence and uniqueness of solutions of Problem 1.3.4, since Q_{red} is a subspace.

Definition 1.3.6 (Reduced elliptic parametric problem)**.** *Given a reduced space $Q_{\text{red}} \subset Q_h^k\big(\tau_h(\boldsymbol{\mu})\big)$, we define the reduced bilinear form $b_{\text{red}} : Q_{\text{red}} \times Q_{\text{red}} \to \mathbb{R}$ and reduced linear functional $l_{\text{red}} : Q_{\text{red}} \to \mathbb{R}$ by restrictions of their discrete counterparts,*

$$b_{\text{red}} := b_h|_{Q_{\text{red}},Q_{\text{red}}} \qquad \textit{and} \qquad l_{\text{red}} := l_h|_{Q_{\text{red}}},$$

respectively, with b_h and l_h from Definition 1.3.4. Find $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that

$$b_{\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}), q_{\text{red}}; \boldsymbol{\mu}\big) = l_{\text{red}}(q_{\text{red}}) \qquad \textit{for all } q_{\text{red}} \in Q_{\text{red}}. \tag{1.3.7}$$

While the dimension of the original approximation space, $N := \dim Q_h^k\big(\tau_h(\mathcal{P})\big)$, scales with the size of the grid, the dimension of the reduced space, $n := \dim Q_{\text{red}}$, only depends on the parameterization of the problem and we can expect $n \ll N$ (see Section 1.3.2.2).

Thus, we can expect to solve the algebraic problem corresponding to (1.3.7) for any $\boldsymbol{\mu} \in \mathcal{P}$ much quicker than the one corresponding to (1.3.4). However, we also need to be able to assemble this algebraic problem for any $\boldsymbol{\mu} \in \mathcal{P}$ quickly, in order to achieve any kind of online or overall efficiency.

1.3.2.1 Offline/online decomposition

This kind of efficiency can be achieved in the context of RB methods by precomputing the restriction of b_h and l_h to the reduced space. This, in turn, is possible if the parameterization of Problem 1.3.1 allows for it, to be more precise: if all parameter dependent quantities allow for an affine parameter decomposition.

Definition 1.3.7 (Affine parameter dependence)**.** *We call* $b : \mathcal{P} \to [H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}]$ affinely decomposable *with respect to* $\mathcal{P}$*, iff there exist* $\varXi \in \mathbb{N}$ *nonparametric components* $b_\xi : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ *and coefficients* $\theta_\xi : \mathcal{P} \to \mathbb{R}$*, such that*

$$b(p, q; \boldsymbol{\mu}) = \sum_{\xi=0}^{\varXi-1} \theta_\xi(\boldsymbol{\mu})\, b_\xi(p, q) \qquad \textit{for all } p, q \in H^1(\Omega) \textit{ and all } \boldsymbol{\mu} \in \mathcal{P}. \tag{1.3.8}$$

The affine parameter dependence of the bilinear form usually follows from a similar decomposition of the data functions, for instance, if b is given as in (1.3.2).

Example 1.3.8 (Affine decomposition of the Thermal Block problem)**.** *With the notation and assumptions from Example 1.3.2, we define* $\lambda_\xi(x) := \chi_{\Omega_\xi}(x)$ *and* $\theta_\xi(\boldsymbol{\mu}) := \boldsymbol{\mu}_\xi$*, respectively, for all* $x \in \Omega$*, all* $\boldsymbol{\mu} \in \mathcal{P}$ *and all* $0 \leq \xi < \varXi$*, where* χ_ω *denotes the indicator function for any* $\omega \subset \Omega$*. It then holds that* $\lambda(x; \boldsymbol{\mu}) = \sum_{\xi=0}^{\varXi-1} \theta_\xi(\boldsymbol{\mu})\, \lambda_\xi(x)$*.*

If such an affine decomposition of the data functions is not available, one can replace the data functions by arbitrary close approximations using Empirical Interpolation, introduced in [BMNP2004].

We also presume that the discrete bilinear form b_h is affinely decomposable in the sense of Definition 1.3.7 as well, with components $b_{\xi,h} : Q_h^k\big(\tau_h(\boldsymbol{\mu})\big) \times Q_h^k\big(\tau_h(\boldsymbol{\mu})\big) \to \mathbb{R}$, such that:

$$b_h(p_h, q_h; \boldsymbol{\mu}) = \sum_{\xi=0}^{\varXi-1} \theta_\xi(\boldsymbol{\mu})\, b_{\xi,h}(p_h, q_h) \quad \text{for all } p_h, q_h \in Q_h^k(\tau_h) \text{ and all } \boldsymbol{\mu} \in \mathcal{P}. \tag{1.3.9}$$

For many problems, this decomposition is directly given by a discretization of the components of b. If such a decomposition is not available, for instance in the context of nonlinear PDEs, an efficient evaluation of the involved operators can still be achieved using Empirical Operator Interpolation [Dro2012, DHO2012] (or equivalently the discrete Empirical Interpolation method [CS2010]). These nonlinear model reduction techniques (sometimes also referred to as “hyper reduction” techniques) are an active field of research but do not lie within the scope of this work; we refer to [Dro2012, CFCA2013, WSH2014].

The affine decomposition of the discrete bilinear form is a crucial ingredient for an efficient evaluation of the quantities involved in (1.3.7), since the reduced bilinear form inherits this decomposition,

$$b_{\text{red}}(p_{\text{red}}, q_{\text{red}}; \boldsymbol{\mu}) := \sum_{\xi=0}^{\varXi-1} \theta_\xi(\boldsymbol{\mu})\, b_{\xi,\text{red}}(p_{\text{red}}, q_{\text{red}}), \quad \text{for all } p_{\text{red}}, q_{\text{red}} \in Q_{\text{red}} \text{ and } \boldsymbol{\mu} \in \mathcal{P}.$$

with the coefficients from (1.3.9) and the reduced components $b_{\xi,\text{red}} := b_{\xi,h}|_{Q_{\text{red}},Q_{\text{red}}}$. This affine decomposition of the reduced bilinear form allows for a decomposition of the computational process into an offline and an online part, since the projection of the high-dimensional components onto the reduced space can be precomputed.

We therefore denote the matrix and vector representation of $b_{\xi,h}$ and l_h with respect to the basis of $Q_h^k(\tau_h)$ by $\underline{b_{\xi,h}} \in \mathbb{R}^{N\times N}$, for $0 \leq \xi < \varXi$, and $\underline{l_h} \in \mathbb{R}^N$, respectively.[5] Let further $\Pi_{\text{red}} : Q_h^k(\tau_h(\boldsymbol{\mu})) \to Q_{\text{red}}$ denote the L^2-orthogonal projection onto Q_{red} and $\underline{\Pi_{\text{red}}} \in \mathbb{R}^{n\times N}$ its matrix representation (each row of $\underline{\Pi_{\text{red}}}$ corresponds to one element of the reduced basis ϕ_{red}).

We then obtain the matrix representation of the reduced components $b_{\xi,\text{red}}$ for all $0 \leq \xi < \varXi$, as well as the vector representation of l_{red}, with respect to the reduced basis,

$$\underline{b_{\xi,\text{red}}} := \underline{\Pi_{\text{red}}}\; \underline{b_{\xi,h}}\; \underline{\Pi_{\text{red}}}^{\perp} \in \mathbb{R}^{n\times n} \qquad \text{and} \qquad \underline{l_{\text{red}}} := \underline{l_h}\; \underline{\Pi_{\text{red}}}^{\perp} \in \mathbb{R}^{n}, \tag{1.3.10}$$

respectively.[6] The algebraic problem corresponding to (1.3.7) then reads: find $\underline{p_{\text{red}}(\boldsymbol{\mu})} \in \mathbb{R}^n$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that

$$\underline{b_{\text{red}}(\boldsymbol{\mu})}\;\underline{p_{\text{red}}(\boldsymbol{\mu})} = \underline{l_{\text{red}}}, \qquad \text{with} \qquad \underline{b_{\text{red}}(\boldsymbol{\mu})} := \sum_{\xi=0}^{\varXi-1} \theta_\xi(\boldsymbol{\mu})\, \underline{b_{\xi,\text{red}}} \quad \in \mathbb{R}^{n\times n}. \tag{1.3.11}$$

Remark 1.3.9 (Offline/online decomposition). *In the* offline *part of the computation, given a reduced basis, we compute $\underline{b_{\xi,\text{red}}}$ and $\underline{l_{\text{red}}}$ as in* (1.3.10) *with a computational complexity of $\mathcal{O}(\varXi n^2 N)$. The reduced basis can be obtained with a computational complexity of $\mathcal{O}(N^2)$ if spanned by high-dimensional solutions (compare the greedy algorithm 1.3.10 and Remark 1.2.2).*
In the online *part of the computation, we evaluate the $\varXi$ scalar coefficients θ_ξ and form the linear combination of the reduced component matrices as in* (1.3.11)*, with a computational complexity of $\mathcal{O}(\varXi n^2)$. The reduced dense algebraic system can then be solve with a computational complexity of $\mathcal{O}(n^3)$ using a direct solver.*

This offline/online decomposition of the computational process allows for online- as well as overall efficient computations, since solving the sparse algebraic system corresponding to the high-dimensional problem (1.3.4) requires $\mathcal{O}(N^l)$ operations, for $1 \leq l \leq 2$ (compare the discussion on multi-grid methods in Section 1.2).

[5]The same methodology can be applied to matrix-free operators and functionals as well.
[6]For a matrix $\underline{A} \in \mathbb{R}^{N\times M}$ we denote its transposed by $\underline{A}^{\perp} \in \mathbb{R}^{M\times N}$.

Since we assume that n is much smaller than N (say 100 compared to 10^6), solving the reduced system is much quicker than solving the high-dimensional system, resulting in online efficient computations. If additionally the computational demand of the offline part is taken into account, overall efficiency can be obtained if a large enough number of solutions are required.

1.3.2.2 Basis generation

The quality of the reduced space Q_{red} has a big influence on the success of the model reduction. It should be sufficiently rich to yield an accurate approximation of the image of IO_h (compare Equation 1.3.6). On the other hand, it should be of low dimension to yield a small reduced system (compare Equation 1.3.11).

The potential of a set of functions X to be approximated by a finite-dimensional subspace $X_n \subset \mathcal{X}$ of a Hilbert space $\mathcal{X}$ is measured by the *Kolmogorov n-width*,

$$d_n(X) = \inf_{\substack{X_n \subset \mathcal{X} \\ \dim X_n = n}} \sup_{x \in X} \inf_{y \in X_n} \|x - y\|_{\mathcal{X}},$$

(see [Pin1985, Definition 1.1]), going back to early studies of Kolmogorov [Kol1936]. It can be shown that spaces X_n exist, for which this infimum is attained (and finite) [Pin1985, Theorem II.2.2], but in general it is impossible to find such spaces.

In our setting, $\mathcal{X}$ is given by $Q_h^k\big(\tau_h(\mathcal{P})\big)$ and we are looking for n-dimensional reduced spaces Q_{red} to approximate $X = IO_h(\mathcal{P})$. If the Kolmogorov n-width $d_n\big(IO_h(\mathcal{P})\big)$ would decay exponentially fast for growing n, and if we knew how to construct appropriate n-dimensional subspaces, we would obtain the best possible reduced space.

Great effort has been invested into the task of specifying the n-width of specific sets, such as periodic functions over an interval or the range of linear operators (see for instance [Kol1936, Pin1985]). Explicitly computing the n-width of solution spaces for arbitrary parameterized PDEs a priori, however, can be considered (close to) impossible. However, recent work [CD2015] shows that for affinely decomposed problems such as those considered here, the n-width of the solution manifold actually does decay exponentially fast.

Thus, $IO_h(\mathcal{P})$ can be well approximated by low-dimensional spaces and we are left with the question of how to construct these. In the context of reduced basis methods, one employs the greedy algorithm to iteratively build a series of nested spaces $X_0 \subset X_1 \subset X_2 \cdots \subset X_n \subset \mathcal{X}$ to approximate a set X by using the hitherto worst approximated element of X (see Algorithm 1.3.10). greedy algorithms are well-known tools from approximation theory [Tem2008], and were first introduced in the context of RB methods in [VPP2003] and the references therein. The greedy algorithm searches for the worst approximated element over a set of training parameters $\mathcal{P}_{\text{train}}$ (using `estimate`), and enriches the basis ϕ_n of the approximation space X_n using this element (modeled by `extend`).

We shall present several variants of the greedy algorithm throughout this work, which are each given by specific choices of `init`, `estimate` and `extend`. The performance of the greedy algorithm and the approximation quality of the resulting reduced spaces

Algorithm 1.3.10 The greedy algorithm.

```
Input: $\mathcal{P}_{\text{train}} \subseteq \mathcal{P}$, $\Delta_{\text{red}} > 0$, $n_{\max} \in \mathbb{N}$, init, estimate, extend
Output: $\phi_n$, such that $X_n = \operatorname{span}(\phi_n)$
  $\phi_0 \leftarrow$ init($\mathcal{P}_{\text{train}}$), $n \leftarrow 0$
  while $\max_{\boldsymbol{\mu} \in \mathcal{P}_{\text{train}}}$estimate($\phi_n, \boldsymbol{\mu}$)$> \Delta_{\text{red}}$ and $n < n_{\max}$ do
    $\boldsymbol{\mu}^* \leftarrow \arg\max_{\boldsymbol{\mu} \in \mathcal{P}_{\text{train}}}$estimate($\phi_n, \boldsymbol{\mu}$)
    $\phi_{n+1} \leftarrow$extend($\phi_n, \boldsymbol{\mu}^*$), $n \leftarrow n + 1$
  end while
  return $\phi_n$
```

can be analyzed, if `estimate` is given by a reliable a posteriori error estimate on the approximation error, `init` is given by $\emptyset$ and `extend` is given by $\phi_n \cup IO(\boldsymbol{\mu}^*)$. Then, this algorithm is a weak greedy algorithm in the sense of [BCD+2011]. The worst best-approximation error of the resulting reduced space can be bounded

$$\sup_{\boldsymbol{\mu} \in \mathcal{P}_{\text{train}}} \inf_{q_{\text{red}} \in Q_{\text{red}}} \|IO(\boldsymbol{\mu}) - q_{\text{red}}\| \lesssim M e^{-cn^{\alpha(\alpha+1)^{-1}}},$$

if the Kolmogorov n-width of $IO_h(\mathcal{P}_{\text{train}})$ decays exponentially fast in the sense, that there exist $M, a, \alpha > 0$, such that

$$d_n\big(IO_h(\mathcal{P}_{\text{train}})\big) \leq M e^{-an^{\alpha}},$$

where $c > 0$ only depends on a and α [BCD+2011, Theorem 3.2].

Thus, the greedy algorithms yields nearly-optimal reduced spaces, if the set of training parameters is a good representative of $\mathcal{P}$ and if the Kolmogorov n-width of the solution manifold $IO_h(\mathcal{P}_{\text{train}})$ decays exponentially fast, which is the case for the class of problems studied throughout this work. The set of training parameters is usually given by an equidistant partition of the parameter space or a random selection of parameters, but far more complex choices are conceivable and used in practice (see for instance [HDO2011, MS2013]).

1.3.2.3 Accuracy vs. efficiency

As discussed in Section 1.3.2.1, reduced basis methods have the potential to be vastly superior to the Algorithm 1.3.5 in terms of efficiency. This, however, comes at a price in terms of accuracy.

By employing model order reduction, we introduced an additional level of approximation on top of the one already given by the underlying discretization. In the context of RB methods, our original notion of accuracy (compare Equation 1.1.4) reads: for any prescribed tolerance $\Delta > 0$,

$$\underbrace{\|p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\|_*}_{\textit{discretization error}} + \underbrace{\|p_h(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_*}_{\textit{model reduction error}} \leq \Delta \tag{1.3.12}$$

in a suitable norm $\|\cdot\|_*$ for any parameter of interest $\boldsymbol{\mu} \in \mathcal{P}_{\text{int.}}$.

However, due to the offline/online decomposition of the computational process, this level of accuracy cannot be attained by traditional RB methods. To elaborate this point, we present the traditionally used variant of the greedy algorithm, which presumes the existence of an "appropriate" discretization.

Definition 1.3.11 (Discrete weak greedy algorithm)**.** *Let $\mathcal{P}_{\text{train}} \subset \mathcal{P}$ be a finite set, let $p_h(\boldsymbol{\mu})$ denote the solution of the discrete Problem 1.3.4 for a given discretization and let $p_{\text{red}}(\boldsymbol{\mu})$ denote the solution of the reduced Problem 1.3.6, for $\boldsymbol{\mu} \in \mathcal{P}$. Let further $\eta_{\text{red}} : \mathcal{P} \to [Q_h^k\big(\tau_h(\mathcal{P})\big) \to \mathbb{R}]$ denote a reliable a-posterior estimate on the model reduction error, i.e.,*

$$\eta_{\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big) \lesssim \|p_h(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \eta_{\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big)$$

which is offline/online decomposable in the sense of Section 1.3.2.1. The discrete weak greedy algorithm *for the construction of a reduced basis ϕ_{red} spanning a reduced space space Q_{red} to approximate $IO_h(\mathcal{P}_{\text{train}})$ is then given by Algorithm 1.3.10 with*

$$\begin{aligned}
&\texttt{estimate(}\phi_{\text{red}}^{(n)}, \boldsymbol{\mu}\texttt{)} := \eta_{\text{red}}\big(p_{\text{red}}^{(n)}(\boldsymbol{\mu}); \boldsymbol{\mu}\big),\\
&\texttt{init(}\mathcal{P}_{\text{train}}\texttt{)} \qquad := p_h\big(\arg\max_{\boldsymbol{\mu}\in\mathcal{P}_{\text{train}}} \texttt{estimate(}\{0\}, \boldsymbol{\mu}\texttt{)}\big) \text{ and}\\
&\texttt{extend(}\phi_{\text{red}}^{(n)}, \boldsymbol{\mu}^*\texttt{)} := \texttt{ONB(}\phi_{\text{red}}^{(n)} \cup p_h(\boldsymbol{\mu}^*)\texttt{)},
\end{aligned}$$

where `ONB` *denotes an orthonormalization procedure for improved numerical stability, for instance a stabilized Gram Schmidt procedure (see also Section 3.2.4.1).*

The reduced scheme resulting from this greedy algorithm ensures that the model reduction error is below the prescribed greedy tolerance for all parameters contained in the training set

$$\|p_h(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \Delta_{\text{red}} \qquad \text{for all } \boldsymbol{\mu} \in \mathcal{P}_{\text{train}},$$

but it does not give such a guarantee for any untrained parameter $\boldsymbol{\mu} \in \mathcal{P}_{\text{int.}} \backslash \mathcal{P}_{\text{train}}$. While we can assess the model reduction error for any parameter efficiently during the online phase by means of the a posteriori error estimate,

$$\|p_h(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \eta_{\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big) \qquad \text{for all } \boldsymbol{\mu} \in \mathcal{P},$$

we have no means to improve the approximation quality of the reduced space without resorting to high-dimensional computations involving $Q_h(\tau_h)$. Since the high-dimensional approximation space was prescribed a priori, we have in general no means to assess the discretization error during the online phase. Thus, instead of (1.3.12) we have

$$\underbrace{\|p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\|_*}_{\leq ?} + \|p_h(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \begin{cases} ? + \Delta_{\text{red}} & \text{for } \boldsymbol{\mu} \in \mathcal{P}_{\text{train}},\\ ? + \eta_{\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big) & \text{for } \boldsymbol{\mu} \in \mathcal{P}_{\text{int.}} \backslash \mathcal{P}_{\text{train}}. \end{cases} \tag{1.3.13}$$

During the online phase of traditional RB methods, one is thus left with neither a guarantee on nor an assessment of the magnitude of the full approximation error (namely, the discretization and the model reduction error).

An improvement of this unfortunate situation has only recently been introduced by [Yan2014], by incorporating an AFEM procedure into the greedy algorithm to simultaneously generate $Q_h^k\big(\tau_h(\mathcal{P}_{\text{train}})\big)$ and Q_{red} in a fully adaptive manner.

Definition 1.3.12 (Spatio-parameter greedy algorithm)**.** *With the notation of Definition 1.3.11, let $\eta_{h,\text{red}} : \mathcal{P} \to [Q_h^k\big(\tau_h(\mathcal{P})\big) \to \mathbb{R}]$ denote an a posteriori estimate of the full approximation error, i.e.,*

$$\|p(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \le \eta_{h,\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big) \qquad \textit{for all } \boldsymbol{\mu} \in \mathcal{P},$$

which is offline/online decomposable in the sense of Section 1.3.2.1. Let further η_h denote a reliable and localizable a posteriori estimate of the discretization error and $\Delta > 0$ a tolerance. The spatio-parameter greedy algorithm *for the simultaneous construction of an accurate discretization space $Q_h^k\big(\tau_h(\mathcal{P}_{\text{train}})\big)$ and the reduced basis ϕ_{red} spanning a reduced space Q_{red} to approximate $IO_h(\mathcal{P}_{\text{train}})$ is then given by Algorithm 1.3.10 with*

$$\begin{aligned}
&\texttt{estimate(}\phi_{\text{red}}^{(n)}, \boldsymbol{\mu}\texttt{)} := \eta_{h,\text{red}}\big(p_{\text{red}}^{(n)}(\boldsymbol{\mu}); \boldsymbol{\mu}\big),\\
&\texttt{init(}\mathcal{P}_{\text{train}}\texttt{)} \qquad := \texttt{AFEM(}Q_h^k\big(\tau_h(\emptyset)\big), \arg\max_{\boldsymbol{\mu}\in\mathcal{P}_{\text{train}}} \texttt{estimate(}\{0\}, \boldsymbol{\mu}\texttt{)}, \Delta\texttt{)} \quad \textit{and}\\
&\texttt{extend(}\phi_{\text{red}}^{(n)}, \boldsymbol{\mu}^*\texttt{)} \;\; := \texttt{ONB(}\phi_{\text{red}}^{(n)} \cup \texttt{AFEM(}Q_h^k\big(\tau_h(\mathcal{P}_{\text{train}})\big)^{(n-1)}, \boldsymbol{\mu}^*, \Delta\texttt{))},
\end{aligned}$$

where $\texttt{AFEM(}Q_h^k(\tau_h), \boldsymbol{\mu}^*, \Delta\texttt{)}$ denotes an AFEM procedure which adaptively refines the given grid τ_h using η_h, and produces an accurate approximation $p_h(\boldsymbol{\mu}^*) \in Q_h^k\big(\tau_h(\boldsymbol{\mu}^*)\big)$, such that $\eta_h(\boldsymbol{\mu}^*) \le \Delta$. In `init`*, $\tau_h(\emptyset)$ denotes an arbitrary coarse initial grid.*

The spatio-parameter greedy algorithm constructs sequences of nested discretization spaces $Q_h^k\big(\tau_h(\mathcal{P}_{\text{train}})\big)^{(n)}$ and nested reduced basis spaces $Q_{\text{red}}^{(n)}$, such that $Q_{\text{red}}^{(n)} \subset Q_h^k\big(\tau_h(\mathcal{P}_{\text{train}})\big)^{(n)}$. Since it does not permit a coarsening of the grid during `AFEM`, all quantities from the previous iteration can be prolonged without any loss of accuracy.

Compared to the traditionally used discrete weak greedy algorithm, the spatio-parameter greedy algorithm guarantees the accuracy of the resulting reduced approximation over the whole training set and allows for an efficient estimation of the full approximation error. Instead of (1.3.13), we thus have

$$\|p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\|_* + \|p_h(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \le \begin{cases} \Delta & \text{for } \boldsymbol{\mu} \in \mathcal{P}_{\text{train}},\\ \eta_{h,\text{red}}(\boldsymbol{\mu}) & \text{for } \boldsymbol{\mu} \in \mathcal{P}_{\text{int.}} \backslash \mathcal{P}_{\text{train}}. \end{cases}$$

during the online phase.

Remark 1.3.13. *The spatio-parameter greedy algorithm yields efficient approximations of elliptic parametric problems that are accurate over the whole set of training parameters and furthermore allows to quantify the full approximation error for any parameter in an* *online-efficient manner. It is worth noting, that we require access to an offline/online decomposable a posteriori error estimate of the full error, which might not be as readily available as an estimate of the model reduction error. It is also worth noting, that the above greedy algorithm can be seen as the first feasible algorithm to approximate* $IO(\mathcal{P}_{train})$*, in contrast to* $IO_h(\mathcal{P}_{train})$*.*

We present such an error estimate in Section 2.3.2, which was developed independently of (and, to the best of our knowledge, earlier than) [Yan2014].

While the spatio-parameter greedy algorithm is fully adaptive during the offline part of the computation and we can efficiently estimate the error online, we have no means to improve the reduced approximation, except by breaking the offline/online decomposition of the computational process and thereby giving up efficiency.

In addition, there exist many problems where the available computing power is limited and where the greedy algorithm can only carry out a limited number of high-dimensional solution snapshots, resulting in a reduced space with insufficient approximation qualities. The latter is true, for instance, in the context of parametric multiscale problems (see Section 1.4), where one can only afford to evaluate IO_h for very few parameters.

An efficient enrichment of the reduced space in the online part of the computational process can only be achieved by a careful incorporation of additional adaptive strategies. Such “online-adaptive” model reduction strategies are an active field of research, see [Car2015, ZF2015]. The *localized reduced basis multiscale method* (LRBMS), which is the focus of the remainder of this work, can be considered the first method of this kind.

To summarize: we introduced elliptic problems that are parameterized by a low-dimensional input vector, and presented the reduced basis method as a particular model reduction technique for such problems. In particular, we discussed the trade-of between the improved efficiency of the resulting reduced scheme and the reduced level of accuracy of such schemes.

1.4 Parametric multiscale problems and combined approaches

This section treats elliptic *parametric multiscale* problems, which are a combination of the problems presented in the previous two sections.

1.4.1 Elliptic parametric multiscale problems

As in Sections 1.2 and 1.3, we consider data functions which exhibit multiscale features associated with a fine scale $\varepsilon \ll |\Omega|$ and also depend on a parameter vector $\boldsymbol{\mu} \in \mathcal{P}$: for fixed $\varepsilon > 0$, we consider a parametric bilinear form $b_\varepsilon : \mathcal{P} \to [H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}]$, such that $b_\varepsilon(\cdot, \cdot; \boldsymbol{\mu})$ is continuous and coercive over $H^1_0(\Omega)$ in the sense of (1.1.1) and (1.1.2) for all $\boldsymbol{\mu} \in \mathcal{P}$, and exhibits high-contrast or strong oscillations as in Definition 1.2.1. We pose the same assumptions on the linear functional l as in the previous sections and, for simplicity, do not consider any parameter or multiscale dependency in the forces and boundary values. Thus, for each parameter there exists a unique solution of the following problem.

Definition 1.4.1 (Elliptic parametric multiscale problem)**.** *For a fixed multiscale parameter $\varepsilon > 0$, a parameter space $\mathcal{P} \subset \mathbb{R}^p$, l as given in Definition 1.1.1 and b_ε as given above, find $p_\varepsilon(\boldsymbol{\mu}) \in H^1_0(\Omega)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that*

$$b_\varepsilon\big(p_\varepsilon(\boldsymbol{\mu}), q; \boldsymbol{\mu}\big) = l(q) \qquad \textit{for all } q \in H^1_0(\Omega). \tag{1.4.1}$$

An example for a parametric multiscale problem is given by the pressure Equation 1.0.1a in the context of two-phase flow in porous media.

Example 1.4.2 (Two-phase flow in porous media)**.** *Let the collection of forces $f \in L^2(\Omega)$ be bounded, let the parametric total mobility $\lambda : \mathcal{P} \to L^\infty(\Omega)$ be strictly positive for all $\boldsymbol{\mu} \in \mathcal{P}$ and let the multiscale permeability tensor $\kappa_\varepsilon \in [L^\infty(\Omega)]^{d\times d}$ be symmetric and positive definite, such that $\lambda(\boldsymbol{\mu})\kappa_\varepsilon \in [L^\infty(\Omega)]^{d\times d}$ for all $\boldsymbol{\mu} \in \mathcal{P}$. For $\boldsymbol{\mu} \in \mathcal{P}$, find a global pressure $p_\varepsilon(\boldsymbol{\mu}) \in H^1_0(\Omega)$, such that $-\nabla\cdot\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla p_\varepsilon(\boldsymbol{\mu})\big) = f$ in the weak sense in $H^1_0(\Omega)$.*

We can see that this is an example for an elliptic parametric multiscale problem by setting

$$b_\varepsilon(p, q; \boldsymbol{\mu}) := \int_\Omega \big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla p\big)\cdot\nabla q \,\mathrm{d}x \qquad \text{and} \qquad l(q) := \int_\Omega f q \,\mathrm{d}x \tag{1.4.2}$$

in Definition 1.4.1.

We are interested in accurate and efficient approximations of elliptic parametric multiscale problem, such as the one given in Example 1.4.2. Problem 1.4.1 is mainly an elliptic parametric problem in the sense of the previous section. As such, we can employ the reduced basis method to *efficiently* compute approximations of Problem 1.4.1 for a large number of parameters $\boldsymbol{\mu} \in \mathcal{P}$. The *accuracy* of any reduced basis approximation is related to the approximation quality of the reduced space, which in turn is related to the number of available solutions of Equation 1.4.1 (since the reduced space is spanned by solutions of (1.4.1) for selected parameters, see the greedy Algorithm 1.3.11).

However, the problem of solving (1.4.1) for a fixed parameter is a multiscale problem in the sense of Section 1.2 and thus suffers from the limitations discussed there. In particular, we cannot expect to approximate the solution of multiscale problems arbitrarily accurate. Moreover, already the cost of approximating only the coarse behavior of the solution may exceed the available computing power. This is particularly troublesome in the context of parametric multiscale problems, where we require many such approximations.

On the other hand, the parametric nature of Problem 1.4.1 also simplifies matters. Similar to traditional reduced basis methods, we can afford to spend a considerable amount of computing power in the offline phase to prepare a reduced space which incorporates as much multiscale information as possible. Many of the techniques used in numerical multiscale methods, such as the preparation of multiscale basis functions enriched with fine-scale information, may seem computationally too costly in the context of nonparametric multiscale problems (compare Remark 1.2.4). In the parametric case, however, these preparations can lead to efficient approximations, since we are interested in solving (1.4.1) for many parameters.

1.4.2 The localized reduced basis multiscale method

Inspired by the localized formulation of numerical multiscale and domain decomposition methods (see Section 1.2), the *localized reduced basis multiscale method* (LRBMS) was introduced [AHKO2012, OS2014, OS2015] for the approximation of parametric multiscale problems, such as Problem 1.4.1. The idea of the LRBMS is to construct a spatially localized reduced basis on each subdomain of a coarse grid $\mathcal{T}_H$, in contrast to a single reduced basis associated with Ω. These local reduced bases can be prescribed a priori, or given as solutions of discrete problems on a fine grid τ_h^T (possibly including overlap) in each of the coarse subdomains $T \in \mathcal{T}_H$, or by global solution snapshots that are restricted to the individual subdomains. In particular, these solution snapshots can be obtained by any of the numerical multiscale methods discussed in Section 1.2. The local reduced spaces are coupled with a discontinuous Galerkin scheme along the faces of the coarse grid, yielding a discontinuous reduced basis space $Q_{\text{red}}(\mathcal{T}_H)$, associated with the coarse grid.

The localized nature of the LRBMS allows to reduce the computational cost of many aspects of the basis generation, compared to traditional reduced basis methods. Since the LRBMS also inherits the offline/online decomposition of the computational process from RB methods, the LRBMS has the potential to be much more efficient than traditional approaches (compare the experiments in Chapter 4).

While the development of the LRBMS was motivated by considering multiscale problems, the resulting methodology is not tied to the multiscale setting and can be seen as a general localized reduced basis scheme. In particular, the resulting reduced scheme is equivalent to that obtained from traditional RB methods (see the discussion of Definition 2.2.1) and most results which are available for RB methods also apply to the LRBMS (including results on error estimation and approximation quality).

In addition to the standard a posteriori estimates on the model reduction error, we also provide a fully offline/online decomposable a posteriori error estimate on the full error in the context of the LRBMS (see Section 2.3.2). This estimate is additionally localizable with respect to the coarse grid and allows for an efficient estimation of the spatial error distribution during the online part of the computational process.

While it is not possible to enrich the reduced space in the context of tradition RB methods, without resorting to high-dimensional computations involving the full high-dimensional discretization, the localized nature of the LRBMS in combination with the localized error estimate, allows for an online enrichment of the local reduced spaces, that only involves local high-dimensional computations (see Section 2.4.2).

Remark 1.4.3 (Accuracy and efficiency of the LRBMS)**.** *If used in conjunction with the discrete weak greedy algorithm 1.3.11, the LRBMS yields accurate reduced approximations for all parameters in the training set, similar to traditional RB methods. In addition, it allows to give an estimate on the full error (including the discretization error), during the online phase. The LRBMS could also be used together with the spatio-parameter greedy algorithm 1.3.12 to additionally ensure the approximation quality of the high-dimensional approximation space.*

During online computations, the LRBMS additionally allows for an enrichment of the reduced basis and thus yields reduced approximations, which are as accurate as the high-dimensional approximation for any *parameter. If combined with local grid adaptation, the LRBMS would also allow for an adaptation of the local approximation spaces during the online enrichment, yielding fully accurate reduced approximations (concerning the discretization as well as the model reduction error).*[7]

After an online enrichment of the basis, all quantities can again be offline/online decomposed with only local high-dimensional computations. Thus, the LRBMS still allows for an efficient computation of such accurate approximations.

To conclude, the LRBMS in its current state allows for efficient and accurate approximations of parametric multiscale problems. Due to its localized nature, the LRBMS additionally carries the potential for a level of accuracy that is far beyond the scope of traditional methods. The theoretical framework of the LRBMS, including the adaptive basis generation, several a posteriori error estimates and the adaptive online enrichment, is presented in Chapter 2 and numerical experiments are given in Chapter 4. The corresponding software framework is presented in Chapter 3.

For the remainder of this chapter, we discuss related methods that combine ideas from model reduction and domain decomposition or numerical multiscale techniques.

1.4.3 Combined approaches

We begin with methodologies that combine domain decomposition and reduced basis techniques to allow for an online adaptation of the computational domain, which are

[7]This is subject to future work.

not explicitly associated with multiscale problems. The idea of the *reduced basis element method* [MR2002, MR2004, LMR2007] is to prepare a reduced basis for each archetype of subdomain and to allow for a deformation and combination of several of these subdomains to form the actual computational domain (much in the spirit of traditional Finite Element methods, where shape functions for each reference element of the grid are considered). The local reduced bases are coupled with Lagrange multipliers and the shape and composition of the domain can be altered during the online computation.

Similarly, the idea of the *reduced basis hybrid method* [IQR2012, IQRV2014] is also to prepare a reduced basis for each archetype of subdomain and to combine these subdomains online. In contrast to the reduced basis element method, however, the Degrees of Freedom associated with the coupling faces of the subdomains are not fully eliminated. This results in a hybrid reduced basis/Finite Element scheme in the online phase. Both methods have been applied for slow flow in the context of computational fluid dynamics.

The *port reduced static condensation reduced basis element method* [HKP2013, EP2013, EP2013a, Sme2015] is an extension of the reduced basis element method and is also based on a set of local reduced bases which have been trained by prescribing different boundary values at the connecting faces of the subdomains (ports). These subdomains can also be combined and assembled online, where the resulting system is treated by static condensation to yield online efficiency. The method has been mainly applied in the context of linear elasticity and structural analysis.

In addition, there exist several approaches where reduced basis techniques have been employed to lower the computational cost of traditional numerical multiscale methods. For instance, in the context of homogenization, the reduced basis methods was employed to provide efficient access to the solutions of the required cell problems [Boy2008]. Similarly, the *reduced basis Finite Element heterogeneous multiscale method* combines the HMM multiscale method (see Section 1.2) on the coarse scale with reduced basis techniques on the fine-scale to provide efficient evaluations of the multiscale data functions [AB2013, AB2014].

However, none of these methods is tailored to the efficient approximation of parametric multiscale problems. Only recently, a combination of the reduced basis framework with localized orthogonal decomposition (compare Section 1.2) has been proposed in [AH2014] for that purpose.

Last, we mention the generalized MsFEM for the linear [EGH2013] as well as the nonlinear setting [EGLP2014] as an extension of the traditional MsFEM (compare Section 1.2). It is based on ideas from [AE2008] where limited global information was used to construct multiscale basis functions. While these basis functions were reused in subsequent computations, yielding efficient approximations of a collection of multiscale problems, the methodology is not proposed as a model reduction technique for the approximation of parametric multiscale problems.

2 The localized reduced basis multiscale method (LRBMS)

This chapter introduces the *localized reduced basis multiscale method* (LRBMS). As an overview, we briefly revisit the problem context and give a very brief overview of the LRBMS methodology. For further details we refer to the previous chapter, in particular Section 1.4, and the respective sections of this chapter.

We are interested in *accurate* and *efficient* approximations of elliptic parametric multiscale problems such as: for a given multiscale parameter $0 < \varepsilon \ll |\Omega|$ and a set of admissible parameters $\mathcal{P} \in \mathbb{R}^\rho$, find $p_\varepsilon(\boldsymbol{\mu}) \in H^1_0(\Omega)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that

$$b_\varepsilon(p_\varepsilon(\boldsymbol{\mu}), q; \boldsymbol{\mu}) = l(q) \qquad \text{for all } q \in H^1_0(\Omega), \tag{2.0.1}$$

where b_ε denotes an elliptic parametric bilinear form, l a continuous linear functional and Ω the spatial domain.

To obtain approximations of 2.0.1 we rely on grid-based discretizations and presume we are given a grid τ_h of Ω, which resolves the fine scale associated with ε, a Finite Element approximation space $Q^k_h(\tau_h)$ of order $k \in \mathbb{N}$ and approximations of b_ε and l, denoted by $b_{\varepsilon,h}$ and l_h: find $p_{\varepsilon,h}(\boldsymbol{\mu}) \in Q^k_h(\tau_h)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that

$$b_{\varepsilon,h}(p_{\varepsilon,h}(\boldsymbol{\mu}), q_h; \boldsymbol{\mu}) = l_h(q_h) \qquad \text{for all } q_h \in Q^k_h(\tau_h). \tag{2.0.2}$$

Possible discretizations include standard continuous or discontinuous Finite Element discretizations (compare Section 3.1.1.1), the multiscale discretizations from Section 1.2 or the one we propose in the next section.

We also presume the existence of a localizable and reliable a posteriori error estimate on the discretization error,

$$\|p_\varepsilon(\boldsymbol{\mu}) - p_{\varepsilon,h}(\boldsymbol{\mu})\|_* \leq \eta_h\big(p_{\varepsilon,h}(\boldsymbol{\mu}); \boldsymbol{\mu}\big)$$

in a suitable norm $\|\cdot\|_*$ to assure the *accuracy* of the approximation; we discuss a particular estimate in Section 2.3.2 below.

To achieve online- and overall-*efficiency*, we rely on a further model reduction of (2.0.2) with respect to $\boldsymbol{\mu}$ by reduced basis methods, namely the localized reduced basis multiscale method (LRBMS). Therefore, we presume the existence of a coarse grid $\mathcal{T}_H$ in addition to the fine grid and a tensor-type decomposition of the approximation space, $Q^k_h(\tau_h) = \oplus_{T \in \mathcal{T}_H} Q^{k,T}_h$, given local approximation spaces $Q^{k,T}_h$ on each subdomain of the coarse grid. This decomposition is directly given for the discretization proposed in the next section and can be obtained for other discretizations by a projection of $p_{\varepsilon,h}$ onto suitable local approximation spaces.

The main idea of the LRBMS is to build individual reduced spaces $Q_{\text{red}}^T(Q_h^{k,T})$ on each subdomain $T \in \mathcal{T}_H$, yielding a reduced space which inherits the discontinuous structure of the approximation space: $Q_{\text{red}}(\mathcal{T}_H) = \oplus_{T \in \mathcal{T}_H} Q_{\text{red}}^T$. We obtain the reduced scheme by a projection onto this reduced space: find $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}(\mathcal{T}_H)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that

$$b_{\text{red}}(p_{\text{red}}(\boldsymbol{\mu}), q_{\text{red}}; \boldsymbol{\mu}) = l_{\text{red}}(q_{\text{red}}) \qquad \text{for all } q_{\text{red}} \in Q_{\text{red}}(\mathcal{T}_H). \tag{2.0.3}$$

The reduced bilinear form and linear functional in (2.0.3) are always given by restrictions of the discrete counterparts in Section 2.1, which do not have to coincide with the ones from (2.0.2).

To control the model reduction error,

$$\|p_{\varepsilon,h}(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \eta_{\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big),$$

as well as the full error

$$\|p_\varepsilon(\boldsymbol{\mu}) - p_{\varepsilon,h}(\boldsymbol{\mu})\|_* + \|p_{\varepsilon,h}(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \eta_{h,\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big), \tag{2.0.4}$$

we propose suitable a posteriori error estimates in Section 2.3, which are used in the adaptive greedy basis generation (Section 2.4.1) and to ensure the *accuracy* of the overall approximation. The latter estimate is additionally localizable with respect to the coarse grid $\mathcal{T}_H$ while being fully offline/online decomposable.

We use the localized estimate (2.0.4) to adaptively enrich the local reduced bases during the online computation (Section 2.4.2). This is of particularly interest in the context of multiscale problems, where we cannot expect to be given enough computing power to construct a reduced space with sufficient approximation quality during the offline computation (compare Sections 1.2 and 1.4).

2.1 Detailed discretization

This section introduces the approximation space $Q^k(\tau_h)$, the discrete bilinear form $b_{\varepsilon,h}$ and the linear functional l_h, all of which we require twofold. First, we use them to discretize the elliptic parametric multiscale Problem 1.4.1 and to compute approximate solutions $p_{\varepsilon,h}$, as in (2.0.2). Second, we use them to define the reduced scheme, namely the reduced bilinear form $b_{\varepsilon,\text{red}}$ and the reduced linear functional l_{red}. We make this explicit distinction here since we want to allow the use of other discretizations for the generation of the solution snapshots, in particular those given by the methods discussed in Section 1.2, which are specifically tailored to the multiscale setting.

We require two nested partitions of Ω, a coarse one, $\mathcal{T}_H$, and a fine one, τ_h. Let τ_h be a grid of Ω with non-overlapping elements t of simple shape, such that $\overline{\cup_{t \in \tau_h}} = \Omega$. We call τ_h a *fine grid* if it resolves all features of the quantities involved in (2.0.1). We pose no further requirements on the *coarse grid* $\mathcal{T}_H$ here, apart from $\mathcal{T}_H$ being a partition of Ω and that each coarse element is made up of at least one fine element. We will pose further requirements on both grids in the context of error estimation further below, but these are not needed to define the discretization.

For simplicity, we can think of τ_h as a shape-regular simplicial triangulation without hanging nodes, and of the coarse elements $T \in \mathcal{T}_H$ as convex. We collect all fine faces in $\mathcal{F}_h$, all coarse faces in $\mathcal{F}_H$ and denote by $\mathcal{N}(t) \subset \tau_h$ and $\mathcal{N}(T) \subset \mathcal{T}_H$ the neighbors of $t \in \tau_h$ and $T \in \mathcal{T}_H$, respectively, and by h_* the diameter of any element $*$ of the sets τ_h, $\mathcal{T}_H$, $\mathcal{F}_h$ or $\mathcal{F}_H$. In addition, we define $h := \max_{t\in\tau_h} h_t$ and $H := \max_{T\in\mathcal{T}_H} H_T$. We collect in $\tau_h^T \subset \tau_h$ the fine elements of τ_h that cover the coarse element $T \in \mathcal{T}_H$ and in $\mathcal{F}_h^* \subset \mathcal{F}_h$ all faces that cover the set $*$, e.g., by $\mathcal{F}_h^t$ the faces of a fine element $t \in \tau_h$, by $\mathcal{F}_h^E$ the faces that cover a coarse face $E \in \mathcal{F}_H$ and so forth; the same notation is used for coarse faces $\mathcal{F}_H^* \subset \mathcal{F}_H$. In addition, we denote the set of all boundary faces by $\overline{\mathcal{F}}_h \subset \mathcal{F}_h$ and the set of all inner faces, that share two elements, by $\mathring{\mathcal{F}}_h \subset \mathcal{F}_h$, such that $\overline{\mathcal{F}}_h \cup \mathring{\mathcal{F}}_h = \mathcal{F}_h$ and $\overline{\mathcal{F}}_h \cap \mathring{\mathcal{F}}_h = \emptyset$. We also denote the set of fine faces which lie on the boundary of any coarse element $T \in \mathcal{T}_H$ by $\overline{\mathcal{F}}_h^T := \cup_{E\in\mathcal{F}_H^T} \mathcal{F}_h^E$ and by $\mathring{\mathcal{F}}_h^T := \mathcal{F}_h^T \backslash \overline{\mathcal{F}}_h^T$ the set of fine faces which lie in the interior of the coarse element. Finally, we assign a unit normal $n_e \in \mathbb{R}^d$ to each inner face $\partial t^- \cap \partial t^+ = e \in \mathring{\mathcal{F}}_h$, pointing from t^- to t^+, and denote the unit outward normal to $\partial\Omega$ by n_e for a boundary face $e = \partial t^- \cap \partial\Omega$, for $t^-, t^+ \in \tau_h$.

Given any two such grids τ_h and $\mathcal{T}_H$ we extend the domain of the bilinear form b_ε and the linear functional l to the broken Sobolev space $H^1(\tau_h)$ and localize them with respect to $\mathcal{T}_H$, where $H^1(\tau_h^*) := \big\{q \in L^2(\Omega) \,\big|\, q|_t \in H^1(t) \;\; \forall t \in \tau_h^*\big\}$ for any $\tau_h^* \subseteq \tau_h$. Throughout this chapter, we presume b_ε and l to be as in (1.4.2), but the methodology can be applied to any elliptic problem. Given the data functions as in Example 1.4.2, we define the local bilinear forms $b_\varepsilon^T : \mathcal{P} \to [H^1(\tau_h^T) \times H^1(\tau_h^T) \to \mathbb{R}]$ and the local linear functionals $l^T : H^1(\tau_h^T)$ by

$$b_\varepsilon^T(p, q; \boldsymbol{\mu}) := \int_T \big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla p\big) \cdot \nabla q \,\mathrm{d}x \qquad \text{and} \qquad l^T(q) := \int_T f q \,\mathrm{d}x, \tag{2.1.1}$$

respectively, for all $\boldsymbol{\mu} \in \mathcal{P}$ and all $p, q \in H^1(\tau_h^T)$. By defining $b_\varepsilon := \sum_{T\in\mathcal{T}_H} b_\varepsilon^T : \mathcal{P} \to [H^1(\tau_h) \times H^1(\tau_h) \to \mathbb{R}]$ and $l := \sum_{T\in\mathcal{T}_H} l^T : H^1(\tau_h) \to \mathbb{R}$ we obtain the same b_ε and l as in (1.4.2), if restricted to $H^1(\Omega) \subset H^1(\tau_h)$.

We discretize Problem 1.4.1 by allowing for a suitable discretization of at least first order inside each coarse element $T \in \mathcal{T}_H$ and by coupling those with a *Symmetric weighted Interior-Penalty discontinuous Galerkin* (SWIPDG) discretization along the coarse faces of $\mathcal{T}_H$. This ansatz can be either interpreted as an extension of the SWIPDG discretization introduced in [ESZ2009] on the coarse partition $\mathcal{T}_H$, where we further refine each coarse element and introduce an additional local discretization, or it can be interpreted as a domain-decomposition approach, where we use local discretizations, defined on subdomains given by the coarse partition, which are then coupled by the SWIPDG fluxes. In view of the latter, this ansatz shows some similarities to [BZ2006] but allows for a wider range of local discretizations. A similar ansatz for a multi-numerics discretization using a different coupling strategy was independently developed and recently introduced in [PVWW2013]. We present two particular choices for the local discretizations and continue with the definition of the overall DG discretization.

2.1.1 Local discretizations

The main idea of our discretizations scheme is to approximate the local bilinear forms b_ε^T and the local linear functionals l^T from (2.1.1), which are defined on the local subdomain triangulations τ_h^T, by discrete counterparts $\mathring{b}_{\varepsilon,h}^T$ and l_h^T, respectively, discretizing Problem 1.4.1 on T with homogeneous Neumann boundary values. We additionally choose local discrete ansatz spaces $Q_h^{k,T} \subset H^1(\tau_h^T)$, with local polynomial order $k \in \mathbb{N}$, $k \geq 1$, to complete the definition of the local discretizations. A natural choice for the local discretization is to use a standard continuous Galerkin (CG) discretization, which we obtain by setting $\mathring{b}_{\varepsilon,h}^T$ to b_ε^T, l_h^T to l^T and $Q_h^{k,T}$ to

$$S_h^k(\tau_h^T) := \big\{ q \in C^0(T) \;\big|\; q|_t \in \mathbb{P}_k(t) \quad \forall t \in \tau_h^T \big\} \quad \subset \quad H^1(T),$$

where $\mathbb{P}_k(\omega)$ denotes the set of polynomials on $\omega \subseteq \Omega$ with total degree at most $k \in \mathbb{N}$. Another choice is to use a discontinuous space for $Q_h^{k,T}$, given by

$$Q_h^k(\tau_h^T) := \big\{ q \in L^2(T) \;\big|\; q|_t \in \mathbb{P}_k(t) \quad \forall t \in \tau_h^T \big\} \quad \subset \quad H^1(\tau_h^T),$$

to set l_h^T to l^T and to choose $\mathring{b}_{\varepsilon,h}^T$ from a family of DG discretizations. Therefore, we introduce the technicalities needed to state a common framework for the non-symmetric, the incomplete, the symmetric and the symmetric weighted interior-penalty (IP) DG discretization (henceforth denoted by NIPDG, IIPDG, SIPDG and SWIPDG, respectively, see [ESZ2009] and the references therein), following [ESV2010, Sect. 2.3].

For a function $q \in H^1(\tau_h)$, which is double-valued on interior faces, we denote its jump on an inner face $e \in \mathring{\mathcal{F}}_h$ by $[\![q]\!]_e := q^- - q^+$ with $q^\pm := q|_{t^\pm}$, recalling that $e = t^- \cap t^+$ for $t^\pm \in \tau_h$. We also assign weights ω_e^-, $\omega_e^+ > 0$ to each inner face, such that $\omega_e^- + \omega_e^+ = 1$, and denote the weighted average of q by $\{\!\{q\}\!\}_e := \omega_e^- q^- + \omega_e^+ q^+$. On a boundary face $e \in \overline{\mathcal{F}}_h$, we set $\omega_e^- = 1$, $\omega_e^+ = 0$, $[\![q]\!]_e := q$ and $\{\!\{q\}\!\}_e := q$. With these definitions, we define the local discrete bilinear form $\mathring{b}_{\varepsilon,h}^T : \mathcal{P} \to [H^1(\tau_h^T) \times H^1(\tau_h^T) \to \mathbb{R}]$ for $\vartheta \in \{-1, 0, 1\}$ by

$$\mathring{b}_{\varepsilon,h}^T(p, q; \boldsymbol{\mu}) := b_\varepsilon^T(p, q; \boldsymbol{\mu}) + \sum_{e \in \mathring{\mathcal{F}}_h^T} \Big(\vartheta b_c^e(q, p; \boldsymbol{\mu}) + b_c^e(p, q; \boldsymbol{\mu}) + b_p^e(q, p; \boldsymbol{\mu}) \Big) \tag{2.1.2}$$

on $T \in \mathcal{T}_H$, with its coupling and penalty parts b_c^e and b_p^e, respectively, defined by

$$b_c^e(p, q; \boldsymbol{\mu}) := \int_e - \{\!\{(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla p) \cdot n_e\}\!\}_e \, [\![q]\!]_e \, \mathrm{d}s \quad \text{and} \quad b_p^e(p, q; \boldsymbol{\mu}) := \int_e \sigma_e(\boldsymbol{\mu}) \, [\![p]\!]_e \, [\![q]\!]_e \, \mathrm{d}s, \tag{2.1.3}$$

for all $\boldsymbol{\mu} \in \mathcal{P}$, all $p, q \in H^1(\tau_h)$ and all $e \in \mathcal{F}_h$. The parametric positive penalty function $\sigma_e : \mathcal{P} \to \mathbb{R}$ is given by $\sigma_e(\boldsymbol{\mu}) := \sigma h_e^{-1} \{\!\{\lambda(\boldsymbol{\mu})\}\!\}_e \, \sigma_\varepsilon^e$, where $\sigma \geq 1$ denotes a user-dependent parameter and the locally adaptive weight is given by $\sigma_\varepsilon^e := \delta_e^+ \delta_e^- (\delta_e^+ + \delta_e^-)^{-1}$ for an interior face $e \in \mathring{\mathcal{F}}_h$ and by $\sigma_\varepsilon^e := \delta_e^-$ on a boundary face $e \in \overline{\mathcal{F}}_h$, respectively, with $\delta_e^\pm := n_e \kappa_\varepsilon^\pm n_e$. Using the weights $\omega_e^\pm := 1/2$, we obtain the NIPDG bilinear form for $\vartheta = -1$, the IIPDG bilinear form for $\vartheta = 0$ and the SIPDG bilinear form for $\vartheta = 1$. We obtain the SWIPDG bilinear form for $\vartheta = 1$ by using locally adaptive weights $\omega_e^- := \delta_e^+(\delta_e^+ + \delta_e^-)^{-1}$ and $\omega_e^+ := \delta_e^-(\delta_e^+ + \delta_e^-)^{-1}$. From now on, we assume that $\mathring{b}_{\varepsilon,h}^T$ is of the form (2.1.2), since this is the most general case (which also covers a CG discretization, where all face terms vanish due to the nature of $S_h^k(\tau_h^T)$).

2.1.2 Global coupling

Now given suitable local discretizations on the coarse elements $T \in \mathcal{T}_H$ we couple those along the coarse faces $E \in \mathcal{F}_H$ using a SWIPDG discretization again and define the bilinear form $b_{\varepsilon,h} : \mathcal{P} \to [H^1(\tau_h) \times H^1(\tau_h) \to \mathbb{R}]$ by

$$b_{\varepsilon,h}(p, q; \boldsymbol{\mu}) := \sum_{T \in \mathcal{T}_H} \mathring{b}_{\varepsilon,h}^T(p|_T, q|_T; \boldsymbol{\mu}) + \sum_{E \in \mathcal{F}_H} b_{\varepsilon,h}^E(p, q; \boldsymbol{\mu}), \tag{2.1.4}$$

where we use the SWIPDG variants of $\omega_e^\pm$ to define the coupling bilinear form $b_{\varepsilon,h}^E : \mathcal{P} \to [H^1(\tau_h) \times H^1(\tau_h) \to \mathbb{R}]$ by

$$b_{\varepsilon,h}^E(p, q; \boldsymbol{\mu}) := \sum_{e \in \mathcal{F}_h^E} \Big(b_c^e(q, p; \boldsymbol{\mu}) + b_c^e(p, q; \boldsymbol{\mu}) + b_p^e(q, p; \boldsymbol{\mu}) \Big)$$

for all $E \in \mathcal{F}_H$, all $\boldsymbol{\mu} \in \mathcal{P}$ and all $p, q \in H^1(\tau_h)$. We also define the discrete linear functional $l_h : H^1(\tau_h) \to \mathbb{R}$ by

$$l_h(q) := \sum_{T \in \mathcal{T}_H} l_h^T(q), \quad \text{for all } q \in H^1(\tau_h). \tag{2.1.5}$$

Finally, we define the global approximation space $Q_h^k(\tau_h) \subset H^1(\tau_h)$ for $k \geq 1$ by

$$Q_h^k(\tau_h) := \big\{ q \in H^1(\tau_h) \ \big| \ q|_T \in Q_h^{k,T} \quad \forall T \in \mathcal{T}_H \big\}, \tag{2.1.6}$$

with $Q_h^{k,T}$ either being the local CG space $S_h^k(\tau_h^T)$ or the local DG space $Q_h^k(\tau_h^T)$. Given the data functions from Example 1.4.2, l_h is continuous and, if σ is chosen large enough, $b_{\varepsilon,h}$ is continuous and coercive with respect to a DG norm over Q_h^k (e.g., given by the semi H^1 norm combined with a DG jump norm) and there exists a unique solution to the following problem.

Definition 2.1.1 (Discrete elliptic parametric multiscale problem). *With the notation from Definition 1.4.1 and $Q_h^k(\tau_h)$, $b_{\varepsilon,h}$ and l_h given as above, find $p_{\varepsilon,h}(\boldsymbol{\mu}) \in Q_h^k(\tau_h)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that*

$$b_{\varepsilon,h}\big(p_{\varepsilon,h}(\boldsymbol{\mu}), q_h; \boldsymbol{\mu}\big) = l_h(q_h) \qquad \textit{for all } q_h \in Q_h^k(\tau_h). \tag{2.1.7}$$

Depending on the choice of the coarse grid and the local discretizations we can recover several discretizations from (2.1.7). Choosing $\mathcal{T}_H = \Omega$ and $Q_h^{k,T} = S_h^k(\tau_h^T)$ yields a standard CG discretization (except for the treatment of the boundary values), for instance. Choosing $Q_h^{k,T} = Q_h^k(\tau_h^T)$ for any choice of $\mathcal{T}_H$ or $\mathcal{T}_H = \tau_h$ for any choice of $Q_h^{k,T}$, on the other hand, results in the standard SWIPDG discretization proposed in [ESZ2009, ESV2010]. Note that the local discretizations as well as the polynomial degree k do not have to coincide on each coarse element (or even inside a coarse element when using a local DG space). It is thus possible to balance the computational effort by choosing local CG or k-adaptive DG discretizations.

This puts our discretization close to the multi-numerics discretization proposed in [PVWW2013], where the latter allows for an even wider range of local discretizations while coupling along the coarse faces using Mortar methods. Our discretization is also closely related to the adaptive discontinuous Galerkin multiscale method [EGMP2013, EGM2013]. Concerning the choice of the user dependent penalty factor, we found an automated choice of σ depending on the polynomial degree k, as proposed in [ER2007], to work very well.

Remark 2.1.2 (Alternate form of the global bilinear form)**.** *We have given the global bilinear form $b_{\varepsilon,h}$ in primal formulation in* (2.1.4)*, due to our definition of the coupling bilinear forms $b_{\varepsilon,h}^E$. It is worth noting that there also exists an equivalent form of $b_{\varepsilon,h}$ which is more suitable for the reduction process. This local formulation is induced by a decomposition of $b_{\varepsilon,h}^E$, analogous to local DG methods [CKSS2002]: we can decompose $b_{\varepsilon,h}^E$ into several contributions associated with the coarse elements adjacent to E. We can then rearrange these contributions to obtain local bilinear forms $b_{\varepsilon,h}^T : \mathcal{P} \to [H^1(\tau_h) \times H^1(\tau_h) \to \mathbb{R}]$, given by*

$$b_{\varepsilon,h}^T(p_h^T, q_h^T; \boldsymbol{\mu}) := \mathring{b}_{\varepsilon,h}^T(p_h^T, q_h^T; \boldsymbol{\mu}) + \sum_{E \in \mathcal{F}_H^T} b_{\varepsilon,h}^E(p_h^T, q_h^T; \boldsymbol{\mu}),$$

for all $p_h^T, q_h^T \in Q_h^T$ and $\boldsymbol{\mu} \in \mathcal{P}$, and coupling bilinear forms $b_{\varepsilon,h}^{T,S} : \mathcal{P} \to [H^1(\tau_h^T) \times H^1(\tau_h^S) \to \mathbb{R}]$ for each coarse element $T \in \mathcal{T}_H$ and all of its neighbors $S \in \mathcal{N}(T)$, given by

$$b_{\varepsilon,h}^{T,S}(p_h^T, q_h^S; \boldsymbol{\mu}) := \sum_{E \in \mathcal{F}_H^T \cap \mathcal{F}_H^S} b_{\varepsilon,h}^E(p_h^T, q_h^S; \boldsymbol{\mu}),$$

for all $p_h^T \in Q_h^T$, $q_h^S \in Q_h^S$ and all $\boldsymbol{\mu} \in \mathcal{P}$, such that

$$b_{\varepsilon,h}(p_h, q_h; \boldsymbol{\mu}) = \sum_{T \in \mathcal{T}_H} \Big[b_{\varepsilon,h}^T(p_h|_T, q_h|_T; \boldsymbol{\mu}) + \sum_{S \in \mathcal{N}(T)} b_{\varepsilon,h}^{T,S}(p_h|_T, q_h|_S; \boldsymbol{\mu}) \Big], \qquad (2.1.8)$$

for all $p_h, q_h \in Q_h^k(\tau_h)$ and all $\boldsymbol{\mu} \in \mathcal{P}$.

2.2 Reduced discretization

Once we are given a reduced space Q_{red}, the reduced scheme follows directly from the discrete Problem 2.1.1 by Galerkin projection of all quantities onto the reduced space just like for standard RB methods, as detailed in Section 1.3. Nevertheless, we explicitly state a definition of the reduced discretization here and discuss some of its aspects, in particular those related to the offline/online decomposition of the computational process. For the rest of this section we presume we are given a reduced space $Q_{\text{red}} \subset Q_h^k(\tau_h)$. The reduced basis spanning Q_{red} is usually made up of solutions of (2.1.7) for selected parameters. This does not always have to be the case, however, as the reduced basis functions can also be given a priori (see Section 2.4.1 below) or by another approximation of (2.0.1) entirely (for instance by one of the multiscale methods discussed in Section 1.2). We postpone any discussion regarding the generation of the reduced basis to Section 2.4.1 and presume we are given elements of $Q_h^k(\tau_h)$, which span Q_{red}.

The reduced space inherits the discontinuous structure of $Q_h^k(\tau_h)$, namely its decomposition with respect to the coarse grid, and we explicitly denote the reduced space by $Q_{\text{red}}(\mathcal{T}_H)$ from here on. Following (2.1.6) we assume we are given local reduced basis spaces $Q_{\text{red}}^T \subset Q_h^{k,T}$ for all $T \in \mathcal{T}_H$, such that

$$\begin{aligned} Q_{\text{red}}(\mathcal{T}_H) &= \oplus_{T\in\mathcal{T}_H} Q_{\text{red}}^T \\ &= \left\{ q_h \in Q_h^k(\tau_h) \,\middle|\, q_h|_T \in Q_{\text{red}}^T \;\; \forall\, T \in \mathcal{T}_H \right\} \quad \subset Q_h^k(\tau_h). \end{aligned} \tag{2.2.1}$$

In the same manner, the reduced bilinear form and linear functional inherit the structure of their discrete counterparts, (2.1.8) and (2.1.5), where we prefer the alternate form of the bilinear form as discussed in Remark 2.1.2. We thus define reduced local bilinear forms $b_{\text{red}}^T : \mathcal{P} \to [Q_{\text{red}}^T \times Q_{\text{red}}^T \to \mathbb{R}]$, reduced local functionals $l_{\text{red}}^T : Q_{\text{red}}^T \to \mathbb{R}$ and reduced coupling bilinear forms $b_{\text{red}}^{T,S} : \mathcal{P} \to [Q_{\text{red}}^T \times Q_{\text{red}}^S \to \mathbb{R}]$ for all coarse elements $T \in \mathcal{T}_H$ and respective neighbors $S \in \mathcal{N}(T)$ by restrictions of their discrete counterparts to the corresponding local reduced spaces,

$$b_{\text{red}}^T := b_{\varepsilon,h}^T\big|_{Q_{\text{red}}^T, Q_{\text{red}}^T}, \qquad b_{\text{red}}^{T,S} := b_{\varepsilon,h}^{T,S}\Big|_{Q_{\text{red}}^T, Q_{\text{red}}^S} \qquad \text{and} \qquad l_{\text{red}}^T := l_h^T\big|_{Q_{\text{red}}^T},$$

respectively. The reduced bilinear form $b_{\text{red}} : \mathcal{P} \to [Q_{\text{red}}(\mathcal{T}_H) \times Q_{\text{red}}(\mathcal{T}_H) \to \mathbb{R}]$ and reduced linear functional $l_{\text{red}} : Q_{\text{red}}(\mathcal{T}_H) \to \mathbb{R}$ are then given by

$$b_{\text{red}}(p_{\text{red}}, q_{\text{red}}; \boldsymbol{\mu}) := \sum_{T\in\mathcal{T}_H} \Big[b_{\text{red}}^T(p_{\text{red}}|_T, q_{\text{red}}|_T; \boldsymbol{\mu}) + \sum_{S\in\mathcal{N}(T)} b_{\text{red}}^{T,S}(p_{\text{red}}|_T, q_{\text{red}}|_S; \boldsymbol{\mu}) \Big] \tag{2.2.2}$$

and

$$l_{\text{red}}(q_{\text{red}}) := \sum_{T\in\mathcal{T}_H} l_{\text{red}}^T(q_{\text{red}}|_T), \tag{2.2.3}$$

respectively, for all $p, q \in Q_{\text{red}}$. Since b_{red} and l_{red} inherit continuity and coercivity from their discrete counterparts, there exists a unique solution of the following problem.

Definition 2.2.1 (Reduced elliptic parametric multiscale problem)**.** *For a given parameter space $\mathcal{P}$, a reduced space $Q_{\text{red}}(\mathcal{T}_H) \subset Q_h^k(\tau_h)$ and b_{red} and l_{red} given as above, find $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}(\mathcal{T}_H)$ for $\boldsymbol{\mu} \in \mathcal{P}$, such that*

$$b_{\text{red}}\big(p_{\text{red}}(\boldsymbol{\mu}), q_{\text{red}}; \boldsymbol{\mu}\big) = l_{\text{red}}(q_{\text{red}}) \qquad \textit{for all } q_{\text{red}} \in Q_{\text{red}}(\mathcal{T}_H). \tag{2.2.4}$$

There are several things worth noting, regarding the above reduced scheme. First of all it is equivalent to a reduced scheme obtained by a standard RB method (see Section 1.3) in the following way: we can define a reduced space $Q_{\text{red}} \subset Q_h^k(\tau_h)$ by collecting all local reduced basis functions of all local reduced spaces Q_{red}^T for all $T \in \mathcal{T}_H$ and extending them by 0 outside of T. It is, however, more useful to consider the local reduced spaces, local reduced bilinear forms and local reduced linear functionals separately, as indicated by (2.2.1), (2.2.2) and (2.2.3). This local view allows us to carry out the offline/online decomposition for all subdomains independently, and in parallel, and is crucial for the online-adaptive basis enrichment (see Section 2.4.2).

2.2.1 Offline/online decomposition

The offline/online decomposition of the computational process in the context of the LRBMS is very close to that of standard RB methods, with some techniques borrowed from DG methods. Given the definitions in the previous subsection, the detailed bilinear form $b_{\varepsilon,h}$ is affinely decomposable in the sense of (1.3.8) if such a decomposition holds for λ (compare Example 1.3.8), which we presume from here on. This decomposition is also present in the detailed local and coupling bilinear forms and carries over to their respective reduced variants. With the notation from Definition 1.3.7 we thus presume that the local bilinear forms $b_{\varepsilon,h}^T$ and the coupling bilinear forms $b_{\varepsilon,h}^{T,S}$ are affinely decomposable in the sense of (1.3.8) and we denote their respective nonparametric components by $b_{\varepsilon,\xi,h}^T : Q_h^{k,T} \times Q_h^{k,T} \to \mathbb{R}$ and $b_{\varepsilon,\xi,h}^{T,S} : Q_h^{k,T} \times Q_h^{k,S} \to \mathbb{R}$, for all $0 \leq \xi < \varXi$, all subdomains $T \in \mathcal{T}_H$ and each respective neighbor $S \in \mathcal{N}(T)$. As usual, we define the reduced counterparts of these bilinear forms by restrictions to the respective local reduced spaces, for all $T \in \mathcal{T}_H$, $S \in \mathcal{N}(T)$ and $0 \leq \xi < \varXi$:

$$b_{\xi,\text{red}}^T := b_{\varepsilon,\xi,h}^T\big|_{Q_{\text{red}}^T, Q_{\text{red}}^T} \qquad\text{and}\qquad b_{\xi,\text{red}}^{T,S} := b_{\varepsilon,\xi,h}^{T,S}\Big|_{Q_{\text{red}}^T, Q_{\text{red}}^S}.$$

Given these nonparametric reduced local bilinear forms $b_{\xi,\text{red}}^T : Q_{\text{red}}^T \times Q_{\text{red}}^T \to \mathbb{R}$ and non-parametric reduced coupling bilinear forms $b_{\xi,\text{red}}^{T,S} : Q_{\text{red}}^T \times Q_{\text{red}}^S \to \mathbb{R}$ we define the non-parametric components of the reduced bilinear forms $b_{\xi,\text{red}} : Q_{\text{red}} \times Q_{\text{red}} \to \mathbb{R}$ by

$$b_{\xi,\text{red}}(p_{\text{red}}, q_{\text{red}}; \boldsymbol{\mu}) := \sum_{T \in \mathcal{T}_H} \Big[b_{\xi,\text{red}}^T(p_{\text{red}}|_T, q_{\text{red}}|_T; \boldsymbol{\mu}) + \sum_{S \in \mathcal{N}(T)} b_{\xi,\text{red}}^{T,S}(p_{\text{red}}|_T, q_{\text{red}}|_S; \boldsymbol{\mu}) \Big]$$

for all $p_{\text{red}}, q_{\text{red}} \in Q_{\text{red}}$, all $\boldsymbol{\mu} \in \mathcal{P}$ and all $0 \leq \xi < \varXi$, analogously to (2.2.2). The affine decomposition of the reduced bilinear form b_{red} then reads

$$b_{\text{red}}(p_{\text{red}}, q_{\text{red}}; \boldsymbol{\mu}) = \sum_{\xi=0}^{\varXi-1} \theta_\xi(\boldsymbol{\mu})\, b_{\xi,\text{red}}(p_{\text{red}}, q_{\text{red}}), \tag{2.2.5}$$

for all $p_{\text{red}}, q_{\text{red}} \in Q_{\text{red}}$ and all $\boldsymbol{\mu} \in \mathcal{P}$, with the coefficients θ_ξ of the corresponding decomposition of λ. Given the local structure of the reduced space (2.2.1) we presume that for each local reduced space Q^T_{red}, we are given a local reduced basis ϕ^T_{red} spanning Q^T_{red}, for all subdomains $T \in \mathcal{T}_H$. We denote for all $T \in \mathcal{T}_H$ by $\Pi^T_{\text{red}} : Q^{k,T}_h \to Q^T_{\text{red}}$ the L^2-orthogonal projection and by $\underline{\Pi^T_{\text{red}}} \in \mathbb{R}^{n^T \times N^T}$ its matrix representation with respect to the basis of $Q^{k,T}_h$, where $n^T := \dim Q^T_{\text{red}}$ and $N^T := \dim Q^{k,T}_h$. Each row of $\underline{\Pi^T_{\text{red}}}$ thus corresponds to the basis representation of one element of the local reduced basis ϕ^T_{red} with respect to $Q^{k,T}_h$. We also denote the matrix and vector representations of the local bilinear forms $b^T_{\varepsilon,\xi,h}$, the coupling bilinear forms $b^{T,S}_{\varepsilon,\xi,h}$ and the local functionals l^T_h with respect to the bases of $Q^{k,T}_h$ and $Q^{k,S}_h$ by

$$\underline{l^T_h} \in \mathbb{R}^{N^T}, \qquad \underline{b^T_{\varepsilon,\xi,h}} \in \mathbb{R}^{N^T \times \mathbb{R}^T} \qquad \text{and} \qquad \underline{b^{T,S}_{\varepsilon,\xi,h}} \in \mathbb{R}^{N^S \times N^T},$$

respectively, for all $0 \leq \xi < \Xi$, all $T \in \mathcal{T}_H$ and all $S \in \mathcal{N}(T)$.

With this notation, we split the assembly of the algebraic problem corresponding to Problem 2.2.1 into an offline and an online part as follows: we apply standard techniques locally on each subdomain and combine those with a DG-like mapping $\iota : \mathcal{T}_H \times \mathbb{N} \to \mathbb{N}$ to assemble the reduced component matrices.[1]

Algorithm 2.2.2 Local assembly of reduced component matrices and vectors (LRBMS).

```
Input: $\mathcal{T}_H$, $\iota$, $\underline{l^T_{\text{red}}}$ and $\underline{b^T_{\xi,\text{red}}}$ and $\underline{b^{T,S}_{\xi,\text{red}}}$ for all $0 \leq \xi < \Xi$
Output: $\underline{l_{\text{red}}}$ and $\underline{b_{\xi,\text{red}}}$ for all $0 \leq \xi < \Xi$
  initialize $\underline{l_{\text{red}}} \in \mathbb{R}^n$ and $\underline{b_{\xi,\text{red}}} \in \mathbb{R}^{n \times n}$ for all $0 \leq \xi < \Xi$ with zero entries
  for all $T \in \mathcal{T}_H$ do
      for all $0 \leq i \leq n^T$ do
          $\big(\underline{l_{\text{red}}}\big)_{\iota(T,i)} \leftarrow \big(\underline{l_{\text{red}}}\big)_{\iota(T,i)} + \big(\underline{l^T_{\text{red}}}\big)_i$
          for all $0 \leq j \leq n^T$ and all $0 \leq \xi < \Xi$ do
              $\big(\underline{b_{\xi,\text{red}}}\big)_{\iota(T,i),\iota(T,j)} \leftarrow \big(\underline{b_{\xi,\text{red}}}\big)_{\iota(T,i),\iota(T,j)} + \big(\underline{b^T_{\xi,\text{red}}}\big)_{i,j}$
          end for
      end for
      for all $S \in \mathcal{N}(T)$ do
          for all $0 \leq i \leq n^S$ and all $0 \leq j \leq n^T$ and all $0 \leq \xi < \Xi$ do
              $\big(\underline{b_{\xi,\text{red}}}\big)_{\iota(T,i),\iota(S,j)} \leftarrow \big(\underline{b_{\xi,\text{red}}}\big)_{\iota(T,i),\iota(S,j)} + \big(\underline{b^{T,S}_{\xi,\text{red}}}\big)_{i,j}$
          end for
      end for
  end for
  return $\underline{l_{\text{red}}}$ and $\underline{b_{\xi,\text{red}}}$ for all $0 \leq \xi < \Xi$
```

[1] Given a consecutive numbering of the subdomains, which we denote by $i_T \in \{0, |\mathcal{T}_H|-1\}$ for all $T \in \mathcal{T}_H$, we define the mapping $\iota(T,\cdot) : \{0, n^T - 1\} \to \{0, |\mathcal{T}_H| - 1\}$ for each subdomain by $i \mapsto \iota(T,i) := \sum_{\{S \in \mathcal{T}_H \,|\, i_S < i_T\}} n^S + i$, which corresponds to a standard DG DoF mapping (see also Section 3.1.1.1).

Offline, for all subdomains $T \in \mathcal{T}_H$ and neighbors $S \in \mathcal{N}(T)$ we assemble the matrix and vector representations of $b_{\xi,\mathrm{red}}^T$, $b_{\xi,\mathrm{red}}^{T,S}$ and l_{red}^T with respect to the local reduced bases, by

$$\begin{aligned}
\underline{l_{\mathrm{red}}^T} &:= & \underline{l_h^T}\,\underline{\Pi_{\mathrm{red}}^T}^{\perp} && \in \mathbb{R}^{n^T} \\
\underline{b_{\xi,\mathrm{red}}^T} &:= \underline{\Pi_{\mathrm{red}}^T}\,& \underline{b_{\varepsilon,\xi,h}^T}\,\underline{\Pi_{\mathrm{red}}^T}^{\perp} && \in \mathbb{R}^{n^T \times n^T} \quad \text{and} \\
\underline{b_{\xi,\mathrm{red}}^{T,S}} &:= \underline{\Pi_{\mathrm{red}}^S}\,& \underline{b_{\varepsilon,\xi,h}^{T,S}}\,\underline{\Pi_{\mathrm{red}}^T}^{\perp} && \in \mathbb{R}^{n^S \times n^T},
\end{aligned}$$

respectively, for all $0 \leq \xi < \Xi$.[2] We then assemble the matrix and vector representations of $b_{\xi,\mathrm{red}}$ and l_{red}, denoted by $\underline{b_{\xi,\mathrm{red}}} \in \mathbb{R}^{n \times n}$ for all $0 \leq \xi < \Xi$ and $\underline{l_{\mathrm{red}}} \in \mathbb{R}^n$, respectively with $n := \sum_{T \in \mathcal{T}_H} n^T = \dim Q_{\mathrm{red}}(\mathcal{T}_H)$, using Algorithm 2.2.2.

Online, given any $\boldsymbol{\mu} \in \mathcal{P}$ we proceed just like in the standard RB setting described in Section 1.3.2.1 and assemble the reduced system matrix $\underline{b_{\mathrm{red}}(\boldsymbol{\mu})} \in \mathbb{R}^{n \times n}$ by a linear combination of all component matrices:

$$\underline{b_{\mathrm{red}}(\boldsymbol{\mu})} := \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\,\underline{b_{\xi,\mathrm{red}}}.$$

The reduced linear algebraic problem corresponding to (2.2.4) then reads: find $\underline{p_{\mathrm{red}}(\boldsymbol{\mu})} \in \mathbb{R}^n$, such that $\underline{b_{\mathrm{red}}(\boldsymbol{\mu})}\,\underline{p_{\mathrm{red}}(\boldsymbol{\mu})} = \underline{l_{\mathrm{red}}}$.

There are several things worth noting regarding the offline as well as the online part of the computation, compared to standard RB methods (which correspond to $|\mathcal{T}_H| = 1$). Offline we can carry out the projection of the local matrices and vectors and coupling matrices in parallel, where we only require access to those values of the neighboring local reduced basis functions which lie on the subdomain boundaries. Thus we only have to communicate data associated with one layer of fine grid elements touching the coarse subdomain boundaries (which corresponds to the communication pattern of standard FE methods). Online, the reduced linear system is roughly $|\mathcal{T}_H|$ times larger than that of standard RB methods. The reduced system matrix $\underline{b_{\mathrm{red}}(\boldsymbol{\mu})}$, however, is sparse and we can use standard iterative solvers, if needed.

[2]Note that $\underline{\Pi_{\mathrm{red}}^T}^{\perp}$ denotes the transpose of $\underline{\Pi_{\mathrm{red}}^T}$.

2.3 Error control

As motivated in the previous Chapter, a posteriori error estimates are crucial for all parts of the computational process. Denoting the weak solution of Problem 1.4.1 for a parameter $\boldsymbol{\mu} \in \mathcal{P}$ by $p_\varepsilon(\boldsymbol{\mu}) \in H_0^1(\Omega)$, the discrete solution of Problem 2.1.1 by $p_{\varepsilon,h}(\boldsymbol{\mu}) \in Q_h^k(\tau_h)$ and the reduced solution of Problem 2.2.1 by $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}(\mathcal{T}_H)$, we require

- an estimate on the discretization error,

$$\|p_\varepsilon(\boldsymbol{\mu}) - p_{\varepsilon,h}(\boldsymbol{\mu})\|_* \leq \eta_h(\boldsymbol{\mu}) \leq \Delta_h \tag{2.3.1}$$

  to assess the quality of the discrete solution. If the estimate is localizable, i.e., $\eta_h(\boldsymbol{\mu})^2 = \sum_{t\in\tau_h} \eta_h^t(\boldsymbol{\mu})^2$, we can also use it to steer an adaptive refinement of the computational grid to reach a prescribed tolerance $\Delta_h > 0$ (ensuring an *accurate* approximation) or to drive the spatio-parameter greedy algorithm 1.3.12. If the estimate is additionally rigorous, i.e., it is an upper as well as a lower bound on the error, we can additionally ensure that these adaptive approximations are also *efficient* (see Sections 1.1 and 1.3 for further details);

- an estimate on the model reduction error,

$$\|p_{\varepsilon,h}(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \eta_{\text{red}}(\boldsymbol{\mu}) \leq \Delta_{\text{red}} \tag{2.3.2}$$

  to drive the adaptive basis generation in the case that we are given a fixed approximation space $Q_h^k(\tau_h)$, for instance using the discrete weak greedy algorithm 1.3.11 with a prescribed tolerance $\Delta_{\text{red}} > 0$. This estimate should be reliable as well as offline/online decomposable, to allow the greedy algorithm to search over a large set of training parameters during the basis generation. It also allows us to efficiently assess the model reduction error during the online part of the computation;

- and an estimate on the full error,

$$\|p(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})\|_* \leq \eta_{h,\text{red}}(\boldsymbol{\mu}). \tag{2.3.3}$$

  If offline/online decomposable, such an estimate allows us to assess the full approximation error during the online part of the computation. If additionally localizable with respect to the coarse grid, i.e., $\eta_{h,\text{red}}(\boldsymbol{\mu})^2 = \sum_{T\in\mathcal{T}_H} \eta_{h,\text{red}}^T(\boldsymbol{\mu})^2$ with offline/online decomposable local indicators $\eta_{h,\text{red}}^T(\boldsymbol{\mu})$, we can assess the local error distribution efficiently during the online part of the computation.

For the discretization presented in the previous section, an estimate on the discretization error as in (2.3.1) is given in [ESV2010], if we choose the local approximation spaces as DG spaces, i.e., $Q_h^T = Q_h^k(\tau_h^T)$ or the coarse grid as the fine grid, $\mathcal{T}_H = \tau_h$. The estimate from [ESV2010] can be extended to the parameter-dependent case (analogously to the methodology presented below) and is localizable with respect to the fine grid τ_h.

We present an extension of this estimate to arbitrary local approximation spaces (given mild requirements on the shape of the subdomains), which is localizable with respect to the coarse grid $\mathcal{T}_H$, in Subsection 2.3.2. In the context of RB methods, an estimate on the model reduction error as in (2.3.2) is readily available (at least for linear problems) by residual type estimates and we present a variant of this standard estimate in the context of the LRBMS in Section 2.3.1. That estimate, however, is not localizable and its offline/online computation depends on the number of subdomains.

The estimate we propose in Section 2.3.2 is also an estimate on the full error as in (2.3.3), given mild requirements on the local reduced spaces. Moreover it is fully offline/online decomposable and localizable, which is quite a novelty in the context of RB methods.

For our error analysis we require several norms and scalar products. We denote the product over a space $\mathbb{V}(\omega)$, for $\omega \subseteq \Omega$, by $(\cdot,\cdot)_{\mathbb{V},\omega}$ and omit ω if $\omega = \Omega$; the same holds for the induced norm, $\|\cdot\|_{\mathbb{V},\omega}^2 := (\cdot,\cdot)_{\mathbb{V},\omega}$. For the estimates (2.3.1) and (2.3.3), we use the parametric energy semi-norm $|||\cdot|||_{\cdot} : \mathcal{P} \to [H^1(\tau_h) \to \mathbb{R}]$, which is for $\overline{\boldsymbol{\mu}} \in \mathcal{P}$ defined as

$$|||\cdot|||_{\overline{\boldsymbol{\mu}}}^2 := \sum_{T \in \mathcal{T}_H} |||\cdot|||_{\overline{\boldsymbol{\mu}},T}^2, \qquad |||\cdot|||_{\overline{\boldsymbol{\mu}},T}^2 := \sum_{t \in \tau_h^T} |||\cdot|||_{\overline{\boldsymbol{\mu}},t}^2, \qquad |||\cdot|||_{\overline{\boldsymbol{\mu}},t}^2 := b_\varepsilon^t(\cdot,\cdot;\overline{\boldsymbol{\mu}}), \tag{2.3.4}$$

with $b_\varepsilon^t(p,q;\boldsymbol{\mu}) := \int_t \big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla p\big) \cdot \nabla q \,\mathrm{d}x$ for all $t \in \tau_h$, all $p, q \in H^1(\tau_h)$ and all $\boldsymbol{\mu} \in \mathcal{P}$. Note that $|||\cdot|||_{\cdot}$ is a norm only on $H_0^1(\Omega)$. For the estimate (2.3.2), on the other hand, we define the discrete energy scalar product $(((\cdot,\cdot)))_{h,\cdot} : \mathcal{P} \to [Q_h^k(\tau_h) \times Q_h^k(\tau_h) \to \mathbb{R}]$ and use the discrete energy norm $|||\cdot|||_{h,\cdot} : \mathcal{P} \to [Q_h^k(\tau_h) \to \mathbb{R}]$, for $\overline{\boldsymbol{\mu}} \in \mathcal{P}$ defined as

$$(((p_h, q_h)))_{h,\overline{\boldsymbol{\mu}}} := b_{\varepsilon,h}(p_h, q_h; \overline{\boldsymbol{\mu}}) \qquad \text{and} \qquad |||q_h|||_{h,\overline{\boldsymbol{\mu}}}^2 := (((q_h, q_h)))_{h,\overline{\boldsymbol{\mu}}} \tag{2.3.5}$$

respectively, for all $p_h, q_h \in Q_h^k(\tau_h)$, with $b_{\varepsilon,h}$ given from Section 2.1.

In order to compare these norms for different parameters we use the affine decomposition of λ (compare Example 1.3.8), $\lambda(x;\boldsymbol{\mu}) = \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\, \lambda_\xi(x)$, where we presume from here on that the coefficients θ_ξ are positive and that the components λ_ξ are non-negative, for all $0 \leq \xi < \Xi$. We can then compare λ for two parameters by

$$\alpha(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})\, \lambda(x;\overline{\boldsymbol{\mu}}) \leq \lambda(x;\boldsymbol{\mu}) \leq \gamma(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})\, \lambda(x;\overline{\boldsymbol{\mu}}) \qquad \text{for all } x \in \Omega, \tag{2.3.6}$$

with the positive equivalent functions $\alpha, \gamma : \mathcal{P} \times \mathcal{P} \to \mathbb{R}$ given by

$$\alpha(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}}) := \min_{\xi=0}^{\Xi-1} \frac{\theta_\xi(\boldsymbol{\mu})}{\theta_\xi(\overline{\boldsymbol{\mu}})} \qquad \text{and} \qquad \gamma(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}}) := \max_{\xi=0}^{\Xi-1} \frac{\theta_\xi(\boldsymbol{\mu})}{\theta_\xi(\overline{\boldsymbol{\mu}})}, \tag{2.3.7}$$

for all $\boldsymbol{\mu}, \overline{\boldsymbol{\mu}} \in \mathcal{P}$. Since this decomposition carries over to b_ε and $b_{\varepsilon,h}$, we can also compare the above energy semi-norm for any two parameters $\boldsymbol{\mu}, \overline{\boldsymbol{\mu}} \in \mathcal{P}$ by

$$\sqrt{\alpha(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})}\; |||\cdot|||_{\overline{\boldsymbol{\mu}}} \;\leq\; |||\cdot|||_{\boldsymbol{\mu}} \;\leq\; \sqrt{\gamma(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})}\; |||\cdot|||_{\overline{\boldsymbol{\mu}}}, \tag{2.3.8}$$

the same holds for the local energy semi-norms in (2.3.4) and the discrete energy norm in (2.3.5).

2.3.1 Residual based error control of the model reduction error

Residual-based error estimation is a well established technique within the RB community to estimate the *model reduction error* $e_{\mathrm{red}}(\boldsymbol{\mu}) := p_{\varepsilon,h}(\boldsymbol{\mu}) - p_{\mathrm{red}}(\boldsymbol{\mu})$, see for instance [PR2006, QRM2011, RHP2008]. This technique is particularly suited for the RB context since the solution we compare to, $p_{\varepsilon,h}(\boldsymbol{\mu})$, is already a discrete quantity (which makes the computation of the norm of the Riesz-representative feasible). It is worth noting that the concept of residual-based error estimation has already been used in the context of Finite Element methods to estimate the discretization error for a long time (see for instance [Ver1996, Ver2013] and the references therein).

We estimate the error in the discrete energy norm $|||\cdot|||_{h,\overline{\boldsymbol{\mu}}}$ (see Equation 2.3.5), which enables us to easily compute all constants involved and to obtain a fully computable upper bound on the error during the online part of the computation. One can also estimate the error in any equivalent norm and obtain computable bounds using the so-called *min-θ* approach [PR2006, QRM2011, RHP2008] and Successive constraints methods [HRSP2007].

Given $b_{\varepsilon,h}$ and l_h from the Section 2.1, we define the *residual* $r_{\varepsilon,h} : \mathcal{P} \to [Q_h^k(\tau_h) \to Q_h^k(\tau_h)']$ for all $p_h, q_h \in Q_h^k(\tau_h)$ and all $\boldsymbol{\mu} \in \mathcal{P}$ by

$$r_{\varepsilon,h}[p_h; \boldsymbol{\mu}](q_h) := b_{\varepsilon,h}(p_h, q_h; \boldsymbol{\mu}) - l_h(q_h).$$

Residual-based error estimation is based on the following error identity for the reduced solution $p_{\mathrm{red}}(\boldsymbol{\mu}) \in Q_{\mathrm{red}}(\mathcal{T}_H)$ for all $q_h \in Q_h^k(\tau_h)$ and all $\boldsymbol{\mu} \in \mathcal{P}$:

$$r_{\varepsilon,h}[p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}](q_h) = (((e_{\mathrm{red}}(\boldsymbol{\mu}), q_h)))_{h,\boldsymbol{\mu}}. \tag{2.3.9}$$

This error identity allows us to bound the norm of the model reduction error by the norm of the Riesz-representative of the above residual. We therefore denote the Riesz-representative of $r_{\varepsilon,h}[p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]$ with respect to the discrete energy scalar product given by (2.3.5) for a fixed parameter $\overline{\boldsymbol{\mu}} \in \mathcal{P}$ by $q_{p_{\mathrm{red}}(\boldsymbol{\mu});\overline{\boldsymbol{\mu}}} \in Q_h^k(\tau_h)$, such that

$$\Big(\Big(\Big(q_{p_{\mathrm{red}}(\boldsymbol{\mu});\overline{\boldsymbol{\mu}}}, q_h\Big)\Big)\Big)_{h,\overline{\boldsymbol{\mu}}} = r_{\varepsilon,h}[p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}](q_h) \qquad \text{for all } q_h \in Q_h^k(\tau_h). \tag{2.3.10}$$

We then have the following a posteriori error estimate.

Theorem 2.3.1 (Residual-based a posteriori error estimate). *Let $p_{\varepsilon,h}(\boldsymbol{\mu}) \in Q_h^k(\tau_h)$ denote the discrete solution of Problem 2.1.1 and $p_{\mathrm{red}}(\boldsymbol{\mu}) \in Q_{\mathrm{red}}(\mathcal{T}_H)$ the reduced solution of Problem 2.2.1 for a parameter $\boldsymbol{\mu} \in \mathcal{P}$. Let further $q_{p_{\mathrm{red}}(\boldsymbol{\mu});\overline{\boldsymbol{\mu}}} \in Q_h^k(\tau_h)$, for a fixed parameter $\overline{\boldsymbol{\mu}} \in \mathcal{P}$, denote the Riesz-representative of the residual for $p_{\mathrm{red}}(\boldsymbol{\mu})$, as defined above. It then holds that*

$$\frac{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}{\sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}\,\eta_{\mathrm{red}}(\boldsymbol{\mu}) \leq |||p_{\varepsilon,h}(\boldsymbol{\mu}) - p_{\mathrm{red}}(\boldsymbol{\mu})|||_{h,\overline{\boldsymbol{\mu}}} \leq \eta_{\mathrm{red}}(\boldsymbol{\mu})$$

with $|||\cdot|||_{h,\overline{\boldsymbol{\mu}}}$ given by (2.3.5), *α and γ given by* (2.3.7) *and*

$$\eta_{\mathrm{red}}(\boldsymbol{\mu}) := \frac{1}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}\,\Big|\Big|\Big|q_{p_{\mathrm{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\Big|\Big|\Big|_{h,\overline{\boldsymbol{\mu}}}.$$

Proof. We obtain the upper bound by

$$\begin{aligned}\alpha(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})\, |||p_{\varepsilon,h}(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})|||_{h,\overline{\boldsymbol{\mu}}}^2 &\leq |||e_{\text{red}}(\boldsymbol{\mu})|||_{h,\boldsymbol{\mu}}^2 \\ &= (((e_{\text{red}}(\boldsymbol{\mu}), e_{\text{red}}(\boldsymbol{\mu}))))_{h,\boldsymbol{\mu}} \\ &= r_{\varepsilon,h}[p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\big(e_{\text{red}}(\boldsymbol{\mu})\big) \\ &= \big(\big(\big(q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}, e_{\text{red}}(\boldsymbol{\mu})\big)\big)\big)_{h,\overline{\boldsymbol{\mu}}} \\ &\leq \big|\big|\big|q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\big|\big|\big|_{h,\overline{\boldsymbol{\mu}}} |||e_{\text{red}}(\boldsymbol{\mu})|||_{h,\overline{\boldsymbol{\mu}}},\end{aligned}$$

where we used the norm equivalence (2.3.8) and the definition of e_h in the first inequality, the definition of the discrete energy norm (2.3.5) in the first equality, the error identity (2.3.9) in the second equality, the definition of the Riesz-representative (2.3.10) in the third equality and the Cauchy-Schwarz inequality in the last inequality. We obtain the lower bound, on the other hand, by

$$\begin{aligned}\big|\big|\big|q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\big|\big|\big|_{h,\overline{\boldsymbol{\mu}}}^2 &\leq \big(\big(\big(q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}, q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\big)\big)\big)_{h,\overline{\boldsymbol{\mu}}} \\ &= r_{\varepsilon,h}[p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\big(q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\big) \\ &= \big(\big(\big(e_{\text{red}}(\boldsymbol{\mu}), q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\big)\big)\big)_{h,\boldsymbol{\mu}} \\ &\leq |||e_{\text{red}}(\boldsymbol{\mu})|||_{h,\boldsymbol{\mu}} \big|\big|\big|q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\big|\big|\big|_{h,\boldsymbol{\mu}} \\ &\leq \gamma(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})\, |||e_{\text{red}}(\boldsymbol{\mu})|||_{h,\overline{\boldsymbol{\mu}}} \big|\big|\big|q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}\big|\big|\big|_{h,\overline{\boldsymbol{\mu}}},\end{aligned}$$

using the same arguments. □

As already noted, this kind of estimate is a common one in the context of RB methods. It is reliable, fully offline/online decomposable and gives a fully online computable upper bound (at least in the variant we proposed here). This estimate, however, relies on the error identity (2.3.9), which is a global property. Thus, we cannot except to immediately get a localized estimate using this approach, though there exist recent contributions towards localization [Sme2015]. Additionally, the offline/online decomposition of the computation of $|||q_{p_{\text{red}}(\boldsymbol{\mu});\boldsymbol{\mu}}|||_{h,\overline{\boldsymbol{\mu}}}$ requires as many inversions of the global scalar product as there are elements of the global reduced basis, which in turn scales with the size of the coarse grid, as noted earlier. Again, this global interpretation of the reduced basis is not appropriate in the context of the LRBMS and leads to extensive offline computations, compare the experiments in Section 4.1.

Next, we present a completely different approach to error estimation, which is based on local flux reconstruction. This approach yields a reliable estimate on the discretization as well as on the full error. The resulting estimate is also offline/online decomposable and additionally localizable with respect to the coarse grid.

2.3.2 Localized error control of the discretization and the full error

Our error analysis is a generalization of the ansatz presented in [ESV2010] to provide an estimate on the discretization error for the solution $p_{\varepsilon,h}(\boldsymbol{\mu}) \in Q_h^k(\tau_h)$ of the discrete

Problem 2.1.1 as well as an estimate on the full error for the solution $p_{\mathrm{red}}(\boldsymbol{\mu}) \in Q_{\mathrm{red}}(\mathcal{T}_H)$ of the reduced Problem 2.2.1. The main idea of the a posteriori error estimate presented in [Voh2007, ESV2010] is to observe that the approximate discrete diffusive flux $-\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla_h p_{\varepsilon,h}(\boldsymbol{\mu})$ is nonconforming while its exact counterpart belongs to $H_{\mathrm{div}}(\Omega) \subset [L^2(\Omega)]^d$, which denotes the space of vector valued functions the divergence of which lies in $L^2(\Omega)$. The idea of [Voh2007, ESV2010] is to reconstruct the discrete diffusive flux in a conforming Raviart-Thomas-Nédélec space $V_h^l(\tau_h) \subset H_{\mathrm{div}}(\Omega)$ and compare it to the nonconforming one. Their error analysis relies on a local conservation property of the reconstructed flux on the fine grid τ_h to prove estimates local to the fine grid.

We transfer this concept to the discretization from Section 2.1 and prove estimates local to the coarse grid that are valid for the discrete as well as the reduced approximation. We obtain mild requirements for the coarse triangulation and the local approximation spaces, namely that a local Poincaré inequality holds on each coarse element and that the constant function $\mathbb{1}$ is present in the local approximation spaces. The latter is obvious for traditional discretizations and can be easily achieved for the LRBMS approximation. The estimates are fully offline/online decomposable and can thus be efficiently used for model reduction in the context of the LRBMS.

From here on, we presume the fine grid τ_h to be a simplicial one (and thus call it a triangulation from here on) and to fulfill the requirements stated in [ESV2010, Sect. 2.1], namely shape-regularity and the absence of hanging nodes; an extension to more general triangulations is possible analogously to [ESV2010, A.1]. We also presume the subdomains $T \in \mathcal{T}_H$ to be shaped convex.

We begin by stating an *abstract energy norm estimate* (see [ESV2010, Lemma 4.1]) that splits the difference between the weak solution $p_\varepsilon \in H_0^1(\Omega)$ of problem (1.3.1) and any function $p_h \in H^1(\tau_h)$ into two contributions. This abstract estimate does not depend on the discretization and thus leaves the choice of s and v open. Note that we formulate the following Lemma with different parameters for the energy norm and the weak solution. The price we have to pay for this flexibility is the additional constants involving $\alpha(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})$ and $\gamma(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})$, that vanish if $\overline{\boldsymbol{\mu}}$ and $\boldsymbol{\mu}$ coincide.

Lemma 2.3.2 (Abstract energy norm estimate)**.** *Let $p_\varepsilon(\boldsymbol{\mu}) \in H_0^1(\Omega)$ be the weak solution of Problem 1.4.1 for $\boldsymbol{\mu} \in \mathcal{P}$ and let $p_h \in H^1(\tau_h)$ and $\overline{\boldsymbol{\mu}} \in \mathcal{P}$ be arbitrary. Then*

$$\begin{aligned} |||p_\varepsilon(\boldsymbol{\mu}) - p_h|||_{\overline{\boldsymbol{\mu}}} &\leq \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}} \Big(\sqrt{\gamma(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})} \inf_{s \in H_0^1(\Omega)} |||p_h - s|||_{\overline{\boldsymbol{\mu}}} \\ &\qquad + \inf_{v \in H_{\mathrm{div}}(\Omega)} \Big\{ \sup_{\substack{\varphi \in H_0^1(\Omega) \\ |||\varphi|||_{\boldsymbol{\mu}} = 1}} \big\{ \big(f - \nabla\cdot v, \varphi\big)_{L^2} - \big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \cdot \nabla_h p_h + v, \nabla\varphi\big)_{L^2} \big\} \Big\} \Big) \\ &\leq \tfrac{\sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}\ 2\ |||p_\varepsilon(\boldsymbol{\mu}) - p_h|||_{\overline{\boldsymbol{\mu}}}. \end{aligned} \tag{2.3.11}$$

Proof. We mainly follow the proof of [ESV2010, Lemma 4.1] while accounting for the parameter dependency of the energy norm and the weak solution. It holds for arbitrary

$\boldsymbol{\mu} \in \mathcal{P}$, $p \in H_0^1(\Omega)$ and $p_h \in H^1(\tau_h)$, that

$$|||p - p_h|||_{\boldsymbol{\mu}} \leq \inf_{s \in H_0^1(\Omega)} |||p_h - s|||_{\boldsymbol{\mu}} + \sup_{\substack{\varphi \in H_0^1(\Omega) \\ |||\varphi|||_{\boldsymbol{\mu}} = 1}} b_\varepsilon(p - p_h, \varphi; \boldsymbol{\mu}) \tag{2.3.12}$$

(see [Voh2007, Lemma 7.1]) and for the weak solution $p_\varepsilon(\boldsymbol{\mu}) \in H_0^1(\Omega)$ of Problem (1.4.1), that

$$\begin{aligned} b_\varepsilon(p_\varepsilon(\boldsymbol{\mu}) - p_h, \varphi; \boldsymbol{\mu}) &= \big(f, \varphi\big)_{L^2} - \big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla_h p_h, \nabla\varphi\big)_{L^2}, \\ &= \big(f - \nabla\cdot v, \varphi\big)_{L^2} - \big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \cdot \nabla_h p_h + v, \nabla\varphi\big)_{L^2} \end{aligned} \tag{2.3.13}$$

for all $\varphi \in H_0^1(\Omega)$ and all $v \in H_{\text{div}}(\Omega)$, where we used the definition of b_ε in the first equality and the fact that $(v, \nabla\varphi)_{L^2} = -(\nabla\cdot v, \varphi)_{L^2}$ due to Green's Theorem and $\varphi \in H_0^1(\Omega)$ in the second one. Inserting (2.3.13) into (2.3.12) with $p = p_\varepsilon(\boldsymbol{\mu})$ and using the norm equivalence (2.3.8) then yields the first inequality in (2.3.11).

To obtain the second inequality we choose $s = p_\varepsilon(\boldsymbol{\mu})$ and $v = -\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla p_\varepsilon(\boldsymbol{\mu})$ in the right hand side of (2.3.11) which eliminates the two infimums and leaves us with two terms yet to be estimated arising inside the supremum. Using Green's Theorem and the definition of b_ε we observe the vanishing of the first term. We estimate the second term as

$$\begin{aligned} &\big|\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla_h p_h - \lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla p_\varepsilon(\boldsymbol{\mu}), \nabla\varphi\big)_{L^2}\big| \\ &\qquad = \big|\big((\lambda(\boldsymbol{\mu})\kappa_\varepsilon)^{1/2} \nabla_h(p_h - p_\varepsilon(\boldsymbol{\mu})), (\lambda(\boldsymbol{\mu})\kappa_\varepsilon)^{1/2}\nabla\varphi\big)_{L^2}\big| \\ &\qquad \leq \Big\|(\lambda(\boldsymbol{\mu})\kappa_\varepsilon)^{1/2} \nabla_h(p_h - p_\varepsilon(\boldsymbol{\mu}))\Big\|_{L^2} \Big\|(\lambda(\boldsymbol{\mu})\kappa_\varepsilon)^{1/2}\nabla\varphi\Big\|_{L^2} \\ &\qquad = |||p_h - p_\varepsilon(\boldsymbol{\mu})|||_{\boldsymbol{\mu}} |||\varphi|||_{\boldsymbol{\mu}}, \end{aligned}$$

where we used the Cauchy-Schwarz inequality and the definition of the energy norm. We finally obtain the second inequality of (2.3.11) from the bound above by observing that the supremum vanishes (due to $|||\varphi|||_{\boldsymbol{\mu}} = 1$) and by using the norm equivalence (2.3.8) again. □

The following theorem states the main localization result and gives an indication on how to proceed with the choice of v: it allows to localize the estimate of the above Lemma, if v fulfills a local conservation property. It is still an abstract estimate in the sense that it does not use any information of the discretization and does not yet fully prescribe s and v. Given our assumptions on the data functions λ and κ_ε, we denote by $0 < c_\varepsilon^t(\boldsymbol{\mu})$ and $c_\varepsilon^t(\boldsymbol{\mu}) \leq C_\varepsilon^t(\boldsymbol{\mu})$ the smallest and largest eigenvalue of $\lambda(\boldsymbol{\mu})\kappa_\varepsilon|_t$, respectively, for any $\boldsymbol{\mu} \in \mathcal{P}$ and additionally define $c_\varepsilon^t := \inf_{\boldsymbol{\mu} \in \mathcal{P}} c_\varepsilon^t(\boldsymbol{\mu})$ and $c_\varepsilon^t \leq C_\varepsilon^t := \sum_{\boldsymbol{\mu} \in \mathcal{P}} C_\varepsilon^t(\boldsymbol{\mu})$, for all $t \in \tau_h$.

Theorem 2.3.3 (Locally computable abstract energy norm estimate)**.** *Let $p_\varepsilon(\boldsymbol{\mu}) \in H_0^1(\Omega)$ be the weak solution of Problem 1.4.1 for $\boldsymbol{\mu} \in \mathcal{P}$, let $s \in H_0^1(\Omega)$ and $p_h \in H^1(\tau_h)$ be arbitrary, let $v \in H_{\text{div}}(\Omega)$ fulfill the* local conservation property

$$(\nabla \cdot v, \mathbb{1})_{L^2,T} = (f, \mathbb{1})_{L^2,T}$$

and let $C_P^T > 0$ *denote the constant from the Poincaré inequality,*

$$\big\|\varphi - \Pi_0^T\varphi\big\|_{L^2,T}^2 \le C_P^T h_T^2 \|\nabla\varphi\|_{L^2,T}^2 \qquad \text{for all } \varphi \in H^1(T), \tag{2.3.14}$$

on all $T \in \mathcal{T}_H$*, where* Π_l^ω *denotes the* L^2*-orthogonal projection onto* $\mathbb{P}_l(\omega)$ *for* $l \in \mathbb{N}$ *and* $\omega \subseteq \Omega$*. It then holds for arbitrary* $\overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}} \in \mathcal{P}$*, that*

$$|||p_\varepsilon(\boldsymbol{\mu}) - p_h|||_{\overline{\boldsymbol{\mu}}} \le \tilde{\eta}(p_h, s, v; \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}),$$

with the abstract global estimator $\tilde{\eta}(p_h, s, v; \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})$ *defined as*

$$\tilde{\eta}(p_h, s, v; \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) := \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}\Bigg[\sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}\Big(\sum_{T\in\mathcal{T}_H} \tilde{\eta}_{\mathrm{nc}}^T(p_h, s; \overline{\boldsymbol{\mu}})^2\Big)^{1/2} + \Big(\sum_{T\in\mathcal{T}_H} \tilde{\eta}_{\mathrm{r}}^T(v)^2\Big)^{1/2}$$
$$+ \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}}\Big(\sum_{T\in\mathcal{T}_H} \tilde{\eta}_{\mathrm{df}}^T(p_h, v; \hat{\boldsymbol{\mu}})^2\Big)^{1/2}\Bigg]$$

and the local nonconformity estimator *defined as*

$$\tilde{\eta}_{\mathrm{nc}}^T(p_h, s; \overline{\boldsymbol{\mu}}) := |||p_h - s|||_{\overline{\boldsymbol{\mu}},T},$$

the local residual estimator *defined as*

$$\tilde{\eta}_{\mathrm{r}}^T(v) := (C_P^T / c_\varepsilon^T)^{1/2} h_T \|f - \nabla\cdot v\|_{L^2,T}$$

and the local diffusive flux estimator *defined as*

$$\tilde{\eta}_{\mathrm{df}}^T(p_h, v; \hat{\boldsymbol{\mu}}) := \Big\|\big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2}\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla_h p_h + v\big)\Big\|_{L^2,T}$$

for all coarse elements $T \in \mathcal{T}_H$*, where* $c_\varepsilon^T := \inf_{t\in\tau_h^T} c_\varepsilon^t$*.*

Proof. We loosely follow the proof of [ESV2010, Theorem 3.1] while accounting for the parameter dependency and the coarse triangulation. Fixing an arbitrary $s \in H_0^1(\Omega)$ in (2.3.11) and localizing with respect to the coarse triangulation yields

$$|||p_\varepsilon(\boldsymbol{\mu}) - p_h|||_{\overline{\boldsymbol{\mu}}} \le \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}\Big(\sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}\sqrt{\sum_{T\in\mathcal{T}_H} |||p_h - s|||_{\overline{\boldsymbol{\mu}},T}^2} \tag{2.3.15}$$
$$+ \sup_{\substack{\varphi\in H_0^1(\Omega)\\ |||\varphi|||_{\boldsymbol{\mu}}=1}} \Big\{\sum_{T\in\mathcal{T}_H}\Big(\underbrace{\big(f - \nabla\cdot v, \varphi\big)_{L^2,T}}_{:=(i)} - \underbrace{\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\cdot\nabla_h p_h + v, \nabla\varphi\big)_{L^2,T}}_{:=(ii)}\Big)\Big\}\Big)$$

which leaves us with two local terms we will estimate separately.

(*i*) Since $(f - \nabla\cdot v, \Pi_0^T\varphi)_{L^2,T} = 0$ due to the local conservation property of v we can estimate the first term as

$$\big|\big(f - \nabla\cdot v, \varphi\big)_{L^2,T}\big| \le \|f - \nabla\cdot v\|_{L^2,T}\big\|\varphi - \Pi_0^T\varphi\big\|_{L^2,T}$$
$$\le \sqrt{C_P^T} h_T \Big(\max_{t\in\tau_h^T}\frac{1}{c_\varepsilon^t}\Big)^{1/2}\|f - \nabla\cdot v\|_{L^2,T}|||\varphi|||_{\boldsymbol{\mu},T},$$

where we used the Cauchy-Schwarz inequality, the Poincaré inequality and the equivalence of the local norms, (2.3.8), on all $t \in \tau_h^T$.

(*ii*) We estimate the second term as

$$\begin{aligned}\big|\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla_h p_h + v, \nabla\varphi\big)_{L^2,T}\big| &\leq \Big\|\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\big)^{-1/2}\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla_h p_h + v\big)\Big\|_{L^2,T} |||\varphi|||_{\boldsymbol{\mu},T}\\ &\leq \sqrt{\alpha(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}^{-1}\Big\|\big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2}\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla_h p_h + v\big)\Big\|_{L^2,T} |||\varphi|||_{\boldsymbol{\mu},T}\end{aligned}$$

using the Cauchy-Schwarz inequality, the definition of the local energy semi-norms (2.3.4) and the parameter equivalence (2.3.6).

Inserting the last two inequalities in (2.3.15) and using the Cauchy-Schwarz inequality and the definition of the local estimators and of c_ε^T yields

$$\begin{aligned}|||p_\varepsilon(\boldsymbol{\mu}) - p_h|||_{\overline{\boldsymbol{\mu}}} \leq \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}\Big(&\sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}\Big(\sum_{T\in\mathcal{T}_H}\tilde{\eta}_{\mathrm{nc}}^T(p_h, s; \overline{\boldsymbol{\mu}})^2\Big)^{1/2}\\ &+ \sup_{\substack{\varphi\in H_0^1(\Omega)\\ |||\varphi|||_{\boldsymbol{\mu}}=1}} \underbrace{\Big[\sum_{T\in\mathcal{T}_H}\big(\tilde{\eta}_{\mathrm{r}}^T(v) + \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}}\,\tilde{\eta}_{\mathrm{df}}^T(p_h, v; \hat{\boldsymbol{\mu}})\big)|||\varphi|||_{\boldsymbol{\mu},T}\Big]^2}_{(iii)}\Big).\end{aligned}$$

Using the Cauchy-Schwarz inequality again we can further estimate (*iii*) as

$$(iii) \leq \Bigg[\Big(\sum_{T\in\mathcal{T}_H}\tilde{\eta}_{\mathrm{r}}^T(v)^2\Big)^{1/2} + \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}}\Big(\sum_{T\in\mathcal{T}_H}\tilde{\eta}_{\mathrm{df}}^T(p_h, v; \hat{\boldsymbol{\mu}})^2\Big)^{1/2}\Bigg]|||\varphi|||_{\boldsymbol{\mu}}.$$

The previous two inequalities combined give the final result, since the supremum vanishes due to $|||\varphi|||_{\boldsymbol{\mu}} = 1$. □

Remark 2.3.4 (Properties of the locally computable abstract energy norm estimate)**.** *In contrast to the estimator proposed in [ESV2010] the above estimate is local with respect to $\mathcal{T}_H$, not τ_h. Choosing $\mathcal{T}_H = \tau_h$ we obtain nearly the same estimate as the one in [ESV2010] for the pure diffusion case (apart from a slightly less favorable summation). In general, however, we can only expect $\tilde{\eta}_{\mathrm{r}}$ to be super-convergent as in [ESV2007] if we refine $\mathcal{T}_H$ along with τ_h (see the experiments in Section 4.2.4.1), thus keeping the ratio H/h fixed.*

What is left now in order to turn the abstract estimate of Theorem 2.3.3 into a fully computable one is to specify s and v, given a discrete solution $p_{\varepsilon,h}(\boldsymbol{\mu})$. We will do so in the following paragraphs, finally using the knowledge that $p_{\varepsilon,h}(\boldsymbol{\mu})$ was computed using the discretization from Section 2.1.

2.3.2.1 Oswald interpolation

The form of the nonconformity estimator in Theorem 2.3.3 already indicates how to choose s: it should be close to $p_{\varepsilon,h}(\boldsymbol{\mu})$, in order to minimize $\tilde{\eta}_{\mathrm{nc}}$, and it should be computable with reasonable effort. Both requirements are met by the Oswald interpolation operator, that goes back to [KP2003] (in the context of a posteriori error estimates; see also [ESV2010, Section 2.5] and the references therein). Given any possibly discontinuous function $q_h \in Q_h^k(\mathcal{T}_H)$ we define the *Oswald interpolation operator* $I_{\mathrm{os}} : Q_h^k(\mathcal{T}_H) \to Q_h^k(\mathcal{T}_H) \cap H_0^1(\Omega)$ by prescribing its values on the Lagrange nodes ν of the triangulation: we set $I_{\mathrm{os}}[q_h](\nu) := q_h^t(\nu)$ inside any $t \in \tau_h$ and

$$I_{\mathrm{os}}[q_h](\nu) := \tfrac{1}{|\tau_h^\nu|} \sum_{t \in \tau_h^\nu} q_h^t(\nu) \quad \text{for all inner nodes of } \tau_h \text{ and} \quad I_{\mathrm{os}}[q_h](\nu) := 0 \qquad (2.3.16)$$

for all boundary nodes of τ_h, where $\tau_h^\nu \subset \tau_h$ denotes the set of all simplices of the fine triangulation which share ν as a node. Given a nonconforming approximation $p_{\varepsilon,h}(\boldsymbol{\mu})$, we choose $s = I_{\mathrm{os}}\big[p_{\varepsilon,h}(\boldsymbol{\mu})\big]$.

We continue with the specification of v, which is a bit more involved. The only formal requirement we have is for v to fulfill the local conservation property on each coarse element, but the diffusive flux estimator already gives a good hint on the specific form of v (namely, to be close to $-\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla_h p_{\varepsilon,h}(\boldsymbol{\mu})$). A particular choice is given by the element-wise diffusive flux reconstruction (with respect to the fine triangulation) that was proposed in [ESV2010], which fulfills the local conservation property on the coarse elements if properly defined with respect to the discretization from Section 2.1.

2.3.2.2 Diffusive flux reconstruction

We reconstruct a conforming diffusive flux approximation $u_{\varepsilon,h}(\boldsymbol{\mu}) \in H_{\mathrm{div}}(\Omega)$ of the nonconforming discrete diffusive flux $-\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla_h p_{\varepsilon,h}(\boldsymbol{\mu}) \notin H_{\mathrm{div}}(\Omega)$ in a conforming discrete subspace $V_h^l(\tau_h) \subset H_{\mathrm{div}}(\Omega)$, namely the *Raviart-Thomas-Nédélec* space of vector valued functions (see [ESV2010] and the references therein), defined for $k-1 \leq l \leq k$ by

$$V_h^l(\tau_h) := \big\{ v \in H_{\mathrm{div}}(\Omega) \,\big|\, v|_t \in [\mathbb{P}_l(t)]^d + \boldsymbol{x}\mathbb{P}_l(t) \quad \forall t \in \tau_h \big\}.$$

See [ESV2010, Section 2.4] and the references therein for a detailed discussion of the role of the polynomial degree l, the properties of elements of $V_h^l(\tau_h)$ and the origin of the use of diffusive flux reconstructions in the context of error estimation in general. We define the parametric *diffusive flux reconstruction* operator $R_h^l : \mathcal{P} \to [Q_h^k(\mathcal{T}_H) \to V_h^l(\tau_h)]$, $\boldsymbol{\mu} \mapsto \big[q_h \mapsto R_h^l[q_h;\boldsymbol{\mu}]\big]$, by locally specifying $R_h^l[q_h;\boldsymbol{\mu}] \in V_h^l(\tau_h)$, such that

$$\big(R_h^l[q_h;\boldsymbol{\mu}]\cdot n_e, q\big)_{L^2,e} = b_c^e(q_h,q;\boldsymbol{\mu}) + b_p^e(q_h,q;\boldsymbol{\mu}) \qquad \text{for all } q \in \mathbb{P}_l(e) \qquad (2.3.17\text{a})$$

and all $e \in \mathcal{F}_h^t$ and

$$\big(R_h^l[q_h;\boldsymbol{\mu}], \nabla q\big)_{L^2,t} = -b^t(q_h,q;\boldsymbol{\mu}) - \vartheta \sum_{e \in \mathcal{F}_h^t} b_c^e(q,q_h;\boldsymbol{\mu}) \quad \text{for all } \nabla q \in [\mathbb{P}_{l-1}(t)]^d \qquad (2.3.17\text{b})$$

with $q \in \mathbb{P}_l(t)$ for all $t \in \tau_h$, where ϑ is given by the local discretization inside each coarse element and by $\vartheta = 1$ on all fine faces that lie on a coarse face. This reconstruction of the diffusive flux is sensible for the discrete solution as well as the reduced solution, since the reconstructions of either fulfill the requirements of Theorem 2.3.3.

Lemma 2.3.5 (Local conservativity). *Let $\mathbb{1} \in Q^T_{\text{red}} \subset Q^{k,T}_h$ for all $T \in \mathcal{T}_H$ and let $p_{\varepsilon,h}(\boldsymbol{\mu}) \in Q^k_h(\mathcal{T}_H)$ and $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}(\mathcal{T}_H)$ be the discrete and reduced solution of Problems 2.1.1 and 2.2.1 for a parameter $\boldsymbol{\mu} \in \mathcal{P}$, respectively, and let $u_{\varepsilon,h}(\boldsymbol{\mu}) := R^l_h[p_{\varepsilon,h}(\boldsymbol{\mu}); \boldsymbol{\mu}] \in V^l_h(\tau_h)$ and $u_{\text{red}}(\boldsymbol{\mu}) := R^l_h[p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}] \in V^l_h(\tau_h)$ denote their respective diffusive flux reconstructions. It then holds that $u_{\varepsilon,h}(\boldsymbol{\mu})$ and $u_{\text{red}}(\boldsymbol{\mu})$ fulfill the local conservation property of Theorem 2.3.3, i.e.,*

$$\big(\nabla\cdot u_{\varepsilon,h}(\boldsymbol{\mu}), \mathbb{1}\big)_{L^2,T} = \big(f, \mathbb{1}\big)_{L^2,T} = \big(\nabla\cdot u_{\text{red}}(\boldsymbol{\mu}), \mathbb{1}\big)_{L^2,T} \qquad \textit{for all } T \in \mathcal{T}_H.$$

Proof. We follow the ideas of [ESV2010, Lemma 2.1] while accounting for the coarse triangulation. Let $\mathbb{1}^T \in Q^k_h(\mathcal{T}_H)$ be an indicator for $T \in \mathcal{T}_H$, such that $\mathbb{1}^T\big|_T = \mathbb{1} \in Q^{k,T}_h$ and zero everywhere else. It then holds, that

$$\begin{aligned}\big(\nabla\cdot u_{\varepsilon,h}(\boldsymbol{\mu}), \mathbb{1}\big)_{L^2,T} &= \sum_{t\in\tau^T_h} \big[\big(u_{\varepsilon,h}(\boldsymbol{\mu})\cdot n, \mathbb{1}\big)_{L^2,\partial t} - \big(u_{\varepsilon,h}(\boldsymbol{\mu}), \nabla\mathbb{1}\big)\big)_{L^2,t}\big]\\ &= b_{\varepsilon,h}(p_{\varepsilon,h}(\boldsymbol{\mu}), \mathbb{1}^T; \boldsymbol{\mu}) = \big(f, \mathbb{1}\big)_{L^2,T},\end{aligned}$$

for all $T \in \mathcal{T}_H$, where we used Green's Theorem in the first equality, the definition of the diffusive flux reconstruction, (2.3.17), and the definition of $\mathbb{1}^T$ and $b_{\varepsilon,h}$ in the second and the fact, that $\mathbb{1} \in Q^{k,T}_h$ and $p_{\varepsilon,h}$ solves (2.1.7) in the third. The very same arguments hold for u_{red} and p_{red} solving (2.2.4). ☐

Inserting the Oswald interpolation for s and the diffusive flux reconstruction for v in Theorem 2.3.3 then yields a locally computable energy estimate for the discrete as well as the reduced solution. Though mathematically identical we explicitly distinguish between η_h and $\eta_{h,\text{red}}$, since the latter, restricted to $Q_{\text{red}}(\mathcal{T}_H)$, is offline/online decomposable.

Corollary 2.3.6 (Locally computable energy norm estimate). *Let $p_\varepsilon(\boldsymbol{\mu}) \in H^1_0(\Omega)$ be the weak solution of Problem 1.4.1, let $p_{\varepsilon,h}(\boldsymbol{\mu}) \in Q^1_h(\mathcal{T}_H)$ be the discrete solution of Problem 2.1.1, let $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}(\mathcal{T}_H)$ be the reduced solution of Problem 2.2.1 and let R^l_h denote the diffusive flux reconstruction operator. Let the assumptions of Theorem 2.3.3 and Lemma 2.3.5 be fulfilled and let $\overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}} \in \mathcal{P}$ be arbitrary. It then holds, that*

$$\begin{aligned}|||p_\varepsilon(\boldsymbol{\mu}) - p_{\varepsilon,h}(\boldsymbol{\mu})|||_{\overline{\boldsymbol{\mu}}} &\leq \eta_h(p_{\varepsilon,h}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}),\\ |||p_\varepsilon(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})|||_{\overline{\boldsymbol{\mu}}} &\leq \eta_{h,\text{red}}(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}).\end{aligned}$$

with

$$\begin{aligned}\eta_*(\cdot; \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) := \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}}\Bigg[\ &\sqrt{\gamma(\boldsymbol{\mu}, \overline{\boldsymbol{\mu}})}\Big(\sum_{T\in\mathcal{T}_H} \eta^T_{\text{nc}}(\cdot; \overline{\boldsymbol{\mu}})^2\Big)^{1/2} + \Big(\sum_{T\in\mathcal{T}_H} \eta^T_{\text{r}}(\cdot; \boldsymbol{\mu})^2\Big)^{1/2}\\ &+ \tfrac{1}{\sqrt{\alpha(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}}\Big(\sum_{T\in\mathcal{T}_H} \eta^T_{\text{df}}(\cdot; \boldsymbol{\mu}, \hat{\boldsymbol{\mu}})^2\Big)^{1/2}\Bigg]\end{aligned}$$

for $\eta_* = \eta_h = \eta_{h,\mathrm{red}}$ and

$$\eta_{\mathrm{nc}}^T(\cdot;\overline{\boldsymbol{\mu}}) := \tilde{\eta}_{\mathrm{nc}}^T(\cdot, I_{os}[\cdot];\overline{\boldsymbol{\mu}}), \quad \eta_{\mathrm{r}}^T(\cdot;\boldsymbol{\mu}) := \tilde{\eta}_{\mathrm{r}}^T(R_h^l[\cdot;\boldsymbol{\mu}]), \quad \eta_{\mathrm{df}}^T(\cdot;\boldsymbol{\mu},\hat{\boldsymbol{\mu}}) := \tilde{\eta}_{\mathrm{df}}^T(\cdot, R_h^l[\cdot;\boldsymbol{\mu}];\hat{\boldsymbol{\mu}}).$$

2.3.2.3 Local efficiency

The global efficiency of the abstract estimate was already shown in Lemma 2.3.2 (again note, that $\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}}) = \alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}}) = 1$ if $\boldsymbol{\mu}$ and $\overline{\boldsymbol{\mu}}$ coincide), see [ESV2007, Remarks 4.2 and 4.3] for a discussion. We also state a local efficiency of the local estimates from Corollary 2.3.6 using the form of the discretization, the Oswald interpolation and the diffusive flux reconstruction. Therefore, we further localize our estimates with respect to the fine triangulation and apply the ideas of [ES2008, ESV2007, ESV2010]. We denote by $\lesssim$ a proportionality relation between to quantities a and b in the sense that $a \lesssim b :\Longleftrightarrow a \leq Cb$, where the positive constant C only depends on the space dimension, the polynomial degree k, the polynomial degree of f, the shape-regularity of τ_h and the DG parameters σ and ϑ. We additionally denote the set of all fine elements that touch $T \in \mathcal{T}_H$ by $\tilde{\tau}_h^T := \{t \in \tau_h \,|\, t \cap T \neq \emptyset\}$, the set of all fine faces that touch T by $\tilde{\mathcal{F}}_h^T := \{e \in \mathcal{F}_h \,|\, \exists t \in \tau_h^T : e \cap t \neq \emptyset\}$ and the weighted jump seminorms $[\![\![\cdot]\!]\!]_{\cdot,\mathcal{F}} : \mathcal{P} \to [H^1(\tau_h) \to \mathbb{R}]$, $\boldsymbol{\mu} \mapsto [q \mapsto [\![\![q]\!]\!]_{\boldsymbol{\mu},\mathcal{F}}]$ and $[\![\![\cdot]\!]\!]_{p,\cdot,\mathcal{F}} : \mathcal{P} \to [H^1(\tau_h) \to \mathbb{R}]$, $\boldsymbol{\mu} \mapsto [q \mapsto [\![\![q]\!]\!]_{p,\boldsymbol{\mu},\mathcal{F}}]$ for any subset $\mathcal{F} \subset \mathcal{F}_h$, all $\boldsymbol{\mu} \in \mathcal{P}$ and $q \in H^1(\tau_h)$ by

$$[\![\![q]\!]\!]_{\boldsymbol{\mu},\mathcal{F}} := \Big(\sum_{e\in\mathcal{F}} \big\| [\![(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla q)\cdot n_e]\!]_e \big\|_{L^2,e}\Big)^{1/2} \quad \text{and} \quad [\![\![q]\!]\!]_{p,\boldsymbol{\mu},\mathcal{F}} := \Big(\sum_{e\in\mathcal{F}} b_p^e(q,q;\boldsymbol{\mu})\Big)^{1/2},$$

respectively. We also denote the set of all fine elements that touch $t \in \tau_h$ by $\tilde{\tau}_h^t$ and the set of all fine faces that touch t by $\tilde{\mathcal{F}}_h^t$ and define

$$\begin{aligned}
\tilde{c}_\varepsilon^T &:= \min_{t\in\tilde{\tau}_h^T} c_\varepsilon^t, & \overline{h_T} &:= \max_{t\in\tau_h^T} h_t, & \overline{\omega}^T &:= \big(\max_{e\in\mathcal{F}_h^T} {\omega_e^+}^2\big)^{1/2},\\
C_\varepsilon^T &:= \max_{t\in\tau_h} C_\varepsilon^t, & \underline{h_T} &:= \min_{t\in\tau_h^T} h_t, & \underline{\overline{C}}_\varepsilon^T &:= \max_{t\in\tau_h^T} \big(\big(\max_{s\in t\cup\mathcal{N}(t)} \tfrac{C_\varepsilon^s}{c_\varepsilon^s}\big)^2\big)
\end{aligned}$$

for all $T \in \mathcal{T}_H$.

Theorem 2.3.7 (Local efficiency of the locally computable energy norm estimate)**.** *With the notation and assumptions from Corollary 2.3.6, let f be polynomial and* $\max_{t\in\tau_h} h_t \leq 1$. *It then holds for $p_* = p_{\varepsilon,h}, p_{\mathrm{red}}$, that*

$$\begin{aligned}
\eta_{\mathrm{nc}}^T(p_*(\boldsymbol{\mu});\overline{\boldsymbol{\mu}}) &\lesssim \big(C_\varepsilon^T/\tilde{c}_\varepsilon^T\big)^{1/2}\ [\![\![p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})]\!]\!]_{p,\overline{\boldsymbol{\mu}},\tilde{\mathcal{F}}_h^T},\\
\eta_{\mathrm{r}}^T(p_*(\boldsymbol{\mu});\boldsymbol{\mu}) &\lesssim \sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}(C_p^T/c_\varepsilon^T)^{1/2} h_T \Big[\quad C_\varepsilon^T\ \underline{h_T}^{-1} |\!|\!| p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu}) |\!|\!|_{\overline{\boldsymbol{\mu}},T}\\
&\qquad + \ \overline{\omega}^T\ \overline{h_T}\quad [\![\![p_*(\boldsymbol{\mu})]\!]\!]_{\overline{\boldsymbol{\mu}},\mathcal{F}_h^T}\\
&\qquad + \sigma^{1/2}\ \underline{h_T}^{-1} [\![\![p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})]\!]\!]_{p,\overline{\boldsymbol{\mu}},\mathcal{F}_h^T}\Big]\\
\eta_{\mathrm{df}}^T(p_*(\boldsymbol{\mu});\boldsymbol{\mu},\hat{\boldsymbol{\mu}}) &\lesssim \sqrt{\gamma(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}\sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}\ {\underline{\overline{C}}_\varepsilon^T}^{1/2}\Big(\quad |\!|\!| p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu}) |\!|\!|_{\overline{\boldsymbol{\mu}},T}\\
&\qquad + [\![\![p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})]\!]\!]_{p,\overline{\boldsymbol{\mu}},\mathcal{F}_h^T}\Big)
\end{aligned}$$

for all coarse elements $T \in \mathcal{T}_H$.

Proof. We estimate each local estimator separately. It holds for the nonconformity estimator, that

$$\begin{aligned}\eta_{\mathrm{nc}}^T(p_*(\boldsymbol{\mu}); \overline{\boldsymbol{\mu}}) &= \Big(\sum_{t \in \tau_h^T} |||p_*(\boldsymbol{\mu}) - I_{\mathrm{os}}[p_*(\boldsymbol{\mu})]|||^2_{\overline{\boldsymbol{\mu}},t} \Big)^{1/2} \\ &\lesssim \Big(\sum_{t \in \tau_h^T} C_\varepsilon^t \min_{\tilde{t} \in \tilde{\tau}_h^t} (c_\varepsilon^{\tilde{t}})^{-1} [\![p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})]\!]^2_{p,\overline{\boldsymbol{\mu}},\tilde{\mathcal{F}}_h^t} \Big)^{1/2} \\ &\leq \big(C_\varepsilon^T / \tilde{c}_\varepsilon^T \big)^{1/2} [\![p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})]\!]_{p,\overline{\boldsymbol{\mu}},\tilde{\mathcal{F}}_h^T},\end{aligned}$$

where we use the definition of $\eta_{\mathrm{nc}}^T[p_*(\boldsymbol{\mu})]$ and τ_h^T in the equality and the arguments of [ESV2010, Proof of Theorem 3.2] in the first and the definition of C_ε^T, $\tilde{c}_\varepsilon^T$ and $\tilde{\mathcal{F}}_h^T$ in the second inequality.

It holds for the residual estimator that

$$\begin{aligned}\eta_{\mathrm{r}}^T(p_*(\boldsymbol{\mu}); \boldsymbol{\mu}) \leq \big(C_p^T / c_\varepsilon^T \big)^{1/2} h_T \Big(&\underbrace{\big\| f - \nabla \cdot \big(\lambda(\boldsymbol{\mu}) \kappa_\varepsilon \nabla_h p_*(\boldsymbol{\mu}) \big) \big\|_{L^2,T}}_{:=(i)} \\ &+ \underbrace{\big\| \nabla \cdot \big(\lambda(\boldsymbol{\mu}) \kappa_\varepsilon \nabla_h p_*(\boldsymbol{\mu}) + u_*(\boldsymbol{\mu}) \big) \big\|_{L^2,T}}_{:=(ii)} \Big),\end{aligned}$$

with u_* as in Lemma 2.3.5, where we used the definition of $\eta_{\mathrm{r}}^T[u_*(\boldsymbol{\mu})]$ and the triangle inequality, which leaves us with two terms we will estimate separately.

(*i*) The first term can be estimated as follows, using the definition of τ_h^T and the arguments of [ES2008, Proposition 3.3] in the first and the definition of C_ε^T and τ_h^T and the fact that $\max_{t \in \tau_h} h_t \leq 1$ in the second inequality:

$$(i) \lesssim \Big(\sum_{t \in \tau_h^T} C_\varepsilon^t h_t^{-2} |||p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})|||^2_{\boldsymbol{\mu},t} \Big)^{-1/2} \leq C_\varepsilon^T \underline{h_T}^{-1} |||p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})|||_{\boldsymbol{\mu},T}$$

(*ii*) The second term can be estimated as

$$\begin{aligned}(ii) &\lesssim \Big[\sum_{t \in \tau_h} \Big(h_t^{1/2} \sum_{e \in \mathcal{F}_h^t} \omega_e^+ \big\| [\![\big(\lambda(\boldsymbol{\mu}) \kappa_\varepsilon \nabla_h p_*(\boldsymbol{\mu}) \big) \cdot n_e]\!]_e \big\|_{L^2,e} \\ &\qquad + \sigma^{1/2} h_t^{-1} [\![p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})]\!]_{p,\boldsymbol{\mu},\mathcal{F}_h^t} \Big)^2 \Big]^{1/2} \\ &\lesssim \overline{\omega}^T \, \overline{h_T} \, [\![p_*(\boldsymbol{\mu})]\!]_{\boldsymbol{\mu},\mathcal{F}_h^T} + \underline{h_T}^{-1} \sigma^{1/2} \, [\![p_\varepsilon(\boldsymbol{\mu}) - p_*(\boldsymbol{\mu})]\!]_{p,\boldsymbol{\mu},\mathcal{F}_h^T},\end{aligned}$$

using the definition of τ_h^T and the arguments of [ESV2010, Proof of Theorem 3.2] in the first and Young's inequality and the definition of $\overline{\omega}^T$, $\overline{h^T}$, $\underline{h_T}$ and τ_h^T in the second inequality.

Applying the norm equivalence (2.3.8) yields the desired result for the residual estimator.

Finally, it holds for the diffusive flux estimator, that

$$\begin{aligned}\eta_{\mathrm{df}}^T(p_*(\boldsymbol{\mu});\boldsymbol{\mu},\overline{\boldsymbol{\mu}}) &\leq \sqrt{\gamma(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}\Big(\sum_{t\in\tau_h^T}\Big\|\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\big)^{-1/2}\big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon\nabla p_*(\boldsymbol{\mu})+u_*(\boldsymbol{\mu})\big)\Big\|_{L^2,t}^2\Big)^{1/2}\\ &\lesssim \sqrt{\gamma(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}\Big[\sum_{t\in\tau_h^T}\big(\max_{s\in t\cup\mathcal{N}(t)}\tfrac{C_\varepsilon^s}{c_\varepsilon^s}\big)^2\Big(|||p_\varepsilon(\boldsymbol{\mu})-p_*(\boldsymbol{\mu})|||_{\boldsymbol{\mu},t}\\ &\qquad\qquad + [\![p_\varepsilon(\boldsymbol{\mu})-p_*(\boldsymbol{\mu})]\!]_{p,\boldsymbol{\mu},\mathcal{F}_h^t}\Big)^2\Big]^{1/2}\\ &\lesssim \sqrt{\gamma(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}\;\underline{\overline{C}}_\varepsilon^{T^{1/2}}\Big(|||p_\varepsilon(\boldsymbol{\mu})-p_*(\boldsymbol{\mu})|||_{\boldsymbol{\mu},T}+[\![p_\varepsilon(\boldsymbol{\mu})-p_*(\boldsymbol{\mu})]\!]_{p,\boldsymbol{\mu},\mathcal{F}_h^T}\Big),\end{aligned}$$

where we used the definition of $\eta_{\mathrm{df}}^T[p_*(\boldsymbol{\mu})]$ and τ_h^T and the parameter equivalence from (2.3.8) in the first, [ESV2007, Lemma 4.12] in the second and the definition of $\underline{\overline{C}}_\varepsilon^T$ and $\mathcal{F}_h^T$ in the third inequality. Applying the norm equivalence (2.3.8) again finally yields the desired result for the diffusive flux estimator. $\square$

2.3.2.4 Localized offline/online decomposition

We give a localized offline/online decomposition of the local error indicators and of the global estimator, as defined in Corollary 2.3.6. Here, localized is to be understood in the sense that in order to compute the indicators for a subdomain we only require quantities associated with this subdomain and its neighbors. Therefore, for each local reduced basis space Q_{red}^T, we denote the corresponding reduced basis by $\phi_{\mathrm{red}}^T = \{\varphi_0^T,\dots,\varphi_{n^T-1}^T\}$, with $n^T = \dim Q_{\mathrm{red}}^T \in \mathbb{N}$ for all subdomains $T\in\mathcal{T}_H$. We also define $q_{\mathrm{red}}^T := q_{\mathrm{red}}|_T \in Q_{\mathrm{red}}^T$ for any function $q_{\mathrm{red}}\in Q_{\mathrm{red}}(\mathcal{T}_H)$ (and implicitly understand q_{red}^T to be extended by 0 outside of T, if required by the context). For the reduced solution $p_{\mathrm{red}}(\boldsymbol{\mu})\in Q_{\mathrm{red}}(\mathcal{T}_H)$, we thus get $p_{\mathrm{red}}(\boldsymbol{\mu}) = \sum_{T\in\mathcal{T}_H} p_{\mathrm{red}}^T(\boldsymbol{\mu})$ with $p_{\mathrm{red}}^T(\boldsymbol{\mu}) = \sum_{i=0}^{n^T-1} p_i^T(\boldsymbol{\mu})\varphi_i^T$. We denote the vector of Degrees of Freedom (DoF) of $p_{\mathrm{red}}^T(\boldsymbol{\mu})\in Q_{\mathrm{red}}^T$ by $\underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})}\in\mathbb{R}^{n^T}$, with $\big(\underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})}\big)_i = p_i^T(\boldsymbol{\mu})$.

First of all, however, we need to revisit the definition of the diffusive flux reconstruction operator (2.3.17). Since that operator is defined locally with respect to the fine grid, we can reformulate it with respect to each subdomain and its neighbors, analogously to (2.1.8). Additionally, it inherits the affine decomposition of the bilinear forms. This does not only make it possible to decompose the evaluation of the local indicators as defined in Corollary 2.3.6 into an offline and an online part, given prescribed $\overline{\boldsymbol{\mu}},\hat{\boldsymbol{\mu}}\in\mathcal{P}$. It also allows us to evaluate those indicators efficiently for any online choice of $\overline{\boldsymbol{\mu}}$. This, in turn, allows us to efficiently estimate the full error of a reduced solution in the parameter dependent energy norm $|||\cdot|||_{\overline{\boldsymbol{\mu}}}$ online, for any $\overline{\boldsymbol{\mu}}\in\mathcal{P}$.

For each subdomain $T\in T$, we therefore define the local component diffusive flux reconstruction operators $\mathring{R}_{h,\xi}^{l,T}: Q_h^T \to V_h^l(\tau_h^T)$ for all $0\leq\xi<\Xi$ by locally specifying $\mathring{R}_{h,\xi}^{l,T}[q_h]$, for $q_h\in Q_h^k(\tau_h)$, on all $e\in\mathring{\mathcal{F}}_h^T$, such that

$$\big(\mathring{R}_{h,\xi}^{l,T}[q_h]\cdot n_e, q\big)_{L^2,e} = b_{c,\xi}^e(q_h,q) + b_{p,\xi}^e(q_h,q) \qquad\qquad \text{for all } q\in\mathbb{P}_l(e)$$

and on all $t \in \tau_h^T$, such that

$$\big(\mathring{R}_{h,\xi}^{l,T}[q_h], \nabla q\big)_{L^2,t} = -b_{\varepsilon,\xi}^t(q_h, q) - \vartheta \sum_{e \in \mathcal{F}_h^t \cap \mathring{\mathcal{F}}_h^T} b_{c,\xi}^e(q, q_h) \qquad \text{for all } \nabla q \in [\mathbb{P}_{l-1}(t)]^d$$

with $q \in \mathbb{P}_l(t)$, where ϑ is given as in (2.3.17) and $b_{\varepsilon,\xi}^t$, $b_{c,\xi}^e$ and $b_{p,\xi}^e$ denote the components of the affine decomposition of the local, coupling and penalty bilinear forms b_ε^t, b_c^e and b_p^e from (2.3.4) and (2.1.3), respectively (which we do not give explicitly here). Additionally for all coarse faces $E \in \mathcal{F}_H$, we define the face component diffusive flux reconstruction operator $R_{h,\xi}^{l,E} : Q_h^k(\tau_h) \to V_h^l(\tau_h)$ for all $0 \leq \xi < \Xi$, by locally specifying $R_{h,\xi}^{l,E}[q_h]$, such that

$$\big(R_{h,\xi}^{l,E}[q_h] \cdot n_e, q\big)_{L^2,e} = b_{c,\xi}^e(q_h, q) + b_{p,\xi}^e(q_h, q) \qquad \text{for all } q \in \mathbb{P}_l(e)$$

on all $e \in \mathcal{F}_h^E$ and

$$\big(R_{h,\xi}^{l,E}[q_h], \nabla q\big)_{L^2,t} = -\sum_{e \in \mathcal{F}_h^t \cap \mathcal{F}_h^E} b_{c,\xi}^e(q, q_h) \qquad \text{for all } \nabla q \in [\mathbb{P}_{l-1}(t)]^d$$

with $q \in \mathbb{P}_l(t)$ on all $t \in \tau_h^E$, where $\tau_h^E \subset \tau_h$ denotes those fine elements, a face of which lies in $\mathcal{F}_h^E$. With these definitions we can rewrite the diffusive flux reconstruction operator from (2.3.17) as

$$R_h^l[q_h; \boldsymbol{\mu}] = \sum_{T \in \mathcal{T}_H} \Big[R_h^{l,T}[q_h|_T; \boldsymbol{\mu}] + \sum_{S \in \mathcal{N}(T)} R_h^{l,T,S}[q_h|_S; \boldsymbol{\mu}]\Big],$$

with

$$R_h^{l,T}[q_h|_T; \boldsymbol{\mu}] := \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu}) R_{h,\xi}^{l,T}[q_h|_T] \quad \text{and} \quad R_h^{l,T,S}[q_h|_T; \boldsymbol{\mu}] := \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu}) R_{h,\xi}^{l,T,S}[q_h|_T],$$

where θ_ξ denote the coefficients from the affine decomposition of λ and

$$\begin{aligned} R_{h,\xi}^{l,T}[q_h|_T] &:= \mathring{R}_{h,\xi}^{l,T}[q_h|_T] + \sum_{E \in \mathcal{F}_H^T} R_{h,\xi}^{l,E}[q_h|_T] \qquad \text{and} \\ R_{h,\xi}^{l,T,S}[q_h|_T] &= \sum_{E \in \mathcal{F}_H^T} R_{h,\xi}^{l,E}[q_h|_S], \end{aligned}$$

for all $q_h \in Q_h^k(\tau_h)$ and all $\boldsymbol{\mu} \in \mathcal{P}$.

Using this reformulation of the diffusive flux operator and the fact that the Oswald interpolation operator I_{os} from (2.3.16) is linear, we proceed with the actual offline/online decomposition of the local error indicators and the a posteriori error estimator from Corollary 2.3.6.

The nonconformity estimator η_{nc}^T. Using the definitions of η_{nc}^T and the energy norm (2.3.4), we obtain for the nonconformity estimator

$$\begin{aligned}
\eta_{\text{nc}}^T\big(p_{\text{red}}(\boldsymbol{\mu}); \overline{\boldsymbol{\mu}}\big)^2 &= |||p_{\text{red}}(\boldsymbol{\mu}) - I_{\text{os}}[p_{\text{red}}(\boldsymbol{\mu})]|||_{\overline{\boldsymbol{\mu}},T}^2 \\
&= b_\varepsilon^T\big(p_{\text{red}}(\boldsymbol{\mu}), p_{\text{red}}(\boldsymbol{\mu}); \overline{\boldsymbol{\mu}}\big) - 2b_\varepsilon^T\big(p_{\text{red}}(\boldsymbol{\mu}), I_{\text{os}}[p_{\text{red}}(\boldsymbol{\mu})]; \overline{\boldsymbol{\mu}}\big) \\
&\qquad + \ b_\varepsilon^T\big(I_{\text{os}}[p_{\text{red}}(\boldsymbol{\mu})], I_{\text{os}}[p_{\text{red}}(\boldsymbol{\mu})]; \overline{\boldsymbol{\mu}}\big). \qquad (2.3.18)
\end{aligned}$$

We define the component matrices $\underline{A_\xi^T}, \underline{B_\xi^T}, \underline{C_\xi^T} \in \mathbb{R}^{n^T \times n^T}$ for all $0 \leq \xi < \varXi$ and the parametric matrices $\underline{A^T(\overline{\boldsymbol{\mu}})}, \underline{B^T(\overline{\boldsymbol{\mu}})}, \underline{C^T(\overline{\boldsymbol{\mu}})} \in \mathbb{R}^{n^T \times n^T}$ for all $\overline{\boldsymbol{\mu}} \in \mathcal{P}$ by

$$\begin{aligned}
\big(\underline{A_\xi^T}\big)_{i,j} &:= b_{\varepsilon,\xi}^T(\varphi_i^T, \varphi_j^T), & \underline{A^T(\overline{\boldsymbol{\mu}})} &:= \sum_{\xi=0}^{\varXi-1} \theta_\xi(\overline{\boldsymbol{\mu}})\underline{A_\xi^T}, \\
\big(\underline{B_\xi^T}\big)_{i,j} &:= b_{\varepsilon,\xi}^T\big(\varphi_i^T, I_{\text{os}}[\varphi_j^T]\big), & \underline{B^T(\overline{\boldsymbol{\mu}})} &:= \sum_{\xi=0}^{\varXi-1} \theta_\xi(\overline{\boldsymbol{\mu}})\underline{B_\xi^T}, \\
\big(\underline{C_\xi^T}\big)_{i,j} &:= b_{\varepsilon,\xi}^T\big(I_{\text{os}}[\varphi_i^T], I_{\text{os}}[\varphi_j^T]\big) \quad \text{and} & \underline{C^T(\overline{\boldsymbol{\mu}})} &:= \sum_{\xi=0}^{\varXi-1} \theta_\xi(\overline{\boldsymbol{\mu}})\underline{C_\xi^T},
\end{aligned}$$

respectively, for all $\varphi_i^T, \varphi_j^T \in \phi_{\text{red}}^T$ and all subdomains $T \in \mathcal{T}_H$. We thus obtain the following decomposition of the evaluation of η_{nc}^T from (2.3.18), for variable $\overline{\boldsymbol{\mu}} \in \mathcal{P}$ and all $T \in \mathcal{T}_H$:

$$\eta_{\text{nc}}^T\big(p_{\text{red}}(\boldsymbol{\mu}); \overline{\boldsymbol{\mu}}\big) = \left(\underline{p_{\text{red}}^T(\boldsymbol{\mu})}\,\Big(\underline{A^T(\overline{\boldsymbol{\mu}})} - 2\underline{B^T(\overline{\boldsymbol{\mu}})} + \underline{C^T(\overline{\boldsymbol{\mu}})}\Big)\,\underline{p_{\text{red}}^T(\boldsymbol{\mu})}\right)^{1/2}.$$

The residual estimator η_{r}^T. Similarly, using the definitions of η_{r}^T and the L^2 norm, we obtain for the residual estimator

$$\begin{aligned}
\eta_{\text{r}}^T\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big)^2 &= \big(C_P^T/c_\varepsilon^T\big)\Big\|f - \nabla\cdot R_h^l[p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\Big\|_{L^2,T}^2 \\
&= \big(C_P^T/c_\varepsilon^T\big)\Big(\big\|f\big\|_{L^2,T}^2 - 2\,\underbrace{\big(f, \nabla\cdot R_h^l[p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\big)_{L^2,T}}_{=:(i)} \qquad (2.3.19) \\
&\qquad + \quad \underbrace{\big\|\nabla\cdot R_h^l[p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\big\|_{L^2,T}^2}_{=:(ii)}\Big),
\end{aligned}$$

which leaves us with two terms we consider separately. We make use of the reformulation of the diffusive flux operator from above and the fact that

$$R_h^l[q_h; \boldsymbol{\mu}]\Big|_T = R_h^{l,T}[q_h^T; \boldsymbol{\mu}] + \sum_{S\in\mathcal{N}(T)} R_h^{l,T,S}[q_h^S; \boldsymbol{\mu}],$$

which also holds for all components of the reformulated diffusive flux operator.

(i) We define the component vectors $\underline{d_\xi^T} \in \mathbb{R}^{n^T}$ and $\underline{d_\xi^{T,S}} \in \mathbb{R}^{n^S}$ for all $0 \leq \xi < \Xi$ and the parametric vectors $\underline{d^T(\boldsymbol{\mu})} \in \mathbb{R}^{n^T}$ and $\underline{d^{T,S}(\boldsymbol{\mu})} \in \mathbb{R}^{n^S}$ for all $\boldsymbol{\mu} \in \mathcal{P}$ by

$$\begin{aligned}
\big(\underline{d_\xi^T}\big)_i &:= \big(f, \nabla\cdot R_{h,\xi}^{l,T}[\varphi_i^T]\big)_{L^2,T}, & \underline{d^T(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\underline{d_\xi^T},\\
\big(\underline{d_\xi^{T,S}}\big)_i &:= \big(f, \nabla\cdot R_{h,\xi}^{l,T,S}[\varphi_i^S]\big)_{L^2,T} \quad\text{and} & \underline{d^{T,S}(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\underline{d_\xi^{T,S}},
\end{aligned}$$

respectively, for all $\varphi_i^T \in \phi_{\text{red}}^T$, $\varphi_i^S \in \phi_{\text{red}}^S$ and all subdomains $T \in \mathcal{T}_H$ and their respective neighbors $S \in \mathcal{N}(T)$. We thus obtain

$$(i) = \underline{p_{\text{red}}^T(\boldsymbol{\mu})}\cdot\underline{d^T(\boldsymbol{\mu})} + \sum_{S\in\mathcal{N}(T)} \underline{p_{\text{red}}^S(\boldsymbol{\mu})}\cdot\underline{d^{T,S}(\boldsymbol{\mu})}. \tag{2.3.20}$$

(ii) Using the definition of the L^2 norm, we obtain

$$\begin{aligned}
(ii) = \quad & \big\|\nabla\cdot R_h^{l,T}[p_{\text{red}}^T(\boldsymbol{\mu});\boldsymbol{\mu}]\big\|_{L^2,T}^2\\
+ \quad & 2\sum_{S\in\mathcal{N}(T)} \Big(\nabla\cdot R_h^{l,T}[p_{\text{red}}^T(\boldsymbol{\mu});\boldsymbol{\mu}], \nabla\cdot R_h^{l,T,S}[p_{\text{red}}^S(\boldsymbol{\mu});\boldsymbol{\mu}]\Big)_{L^2,T}\\
+ & \sum_{S\in\mathcal{N}(T)}\sum_{S'\in\mathcal{N}(T)} \Big(\nabla\cdot R_h^{l,T,S}[p_{\text{red}}^S(\boldsymbol{\mu});\boldsymbol{\mu}], \nabla\cdot R_h^{l,T,S'}[p_{\text{red}}^{S'}(\boldsymbol{\mu});\boldsymbol{\mu}]\Big)_{L^2,T}.
\end{aligned} \tag{2.3.21}$$

We define the component matrices $\underline{E_{\xi,\xi'}^T} \in \mathbb{R}^{n^T\times n^T}$, $\underline{E_{\xi,\xi'}^{T,S}} \in \mathbb{R}^{n^T\times n^S}$ and $\underline{E_{\xi,\xi'}^{S,T,S'}} \in \mathbb{R}^{n^S\times n^{S'}}$ for all $0 \leq \xi,\xi' < \Xi$ and the parametric matrices $\underline{E^T(\boldsymbol{\mu})} \in \mathbb{R}^{n^T\times n^T}$, $\underline{E^{T,S}(\boldsymbol{\mu})} \in \mathbb{R}^{n^T\times n^S}$ and $\underline{E^{S,T,S'}(\boldsymbol{\mu})} \in \mathbb{R}^{n^S\times n^{S'}}$ for all $\boldsymbol{\mu} \in \mathcal{P}$ by

$$\begin{aligned}
\big(\underline{E_{\xi,\xi'}^T}\big)_{i,j} &:= \Big(\nabla\cdot R_{h,\xi}^{l,T}[\varphi_i^T], \nabla\cdot R_{h,\xi'}^{l,T}[\varphi_j^T]\Big)_{L^2,T}, & \underline{E^T(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\underline{E_\xi^T},\\
\big(\underline{E_{\xi,\xi'}^{T,S}}\big)_{i,j} &:= \Big(\nabla\cdot R_{h,\xi}^{l,T}[\varphi_i^T], \nabla\cdot R_{h,\xi'}^{l,T,S}[\varphi_j^S]\Big)_{L^2,T}, & \underline{E^{T,S}(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\underline{E_\xi^{T,S}},\\
\big(\underline{E_{\xi,\xi'}^{S,T,S'}}\big) &:= \Big(\nabla\cdot R_{h,\xi}^{l,T,S}[\varphi_i^S], \nabla\cdot R_{h,\xi'}^{l,T,S'}[\varphi_j^{S'}]\Big)_{L^2,T}, & \underline{E^{S,T,S'}(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\underline{E_\xi^{S,T,S'}},
\end{aligned}$$

respectively for all $\varphi_i^T \in \phi_{\text{red}}^T$, $\varphi_i^S, \varphi_j^S \in \phi_{\text{red}}^S$, $\varphi_j^{S'} \in \phi_{\text{red}}^{S'}$ and all subdomains $T \in \mathcal{T}_H$

and their respective neighbors $S, S' \in \mathcal{N}(T)$. We thus obtain from (2.3.21):

$$\begin{aligned}(ii) = \quad & \underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})}\ \underline{E^T(\boldsymbol{\mu})}\ \underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})} \\ + \quad & 2 \sum_{S \in \mathcal{N}(T)} \underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})}\ \underline{E^{T,S}(\boldsymbol{\mu})}\ \underline{p_{\mathrm{red}}^S(\boldsymbol{\mu})} \\ + & \sum_{S \in \mathcal{N}(T)} \sum_{S' \in \mathcal{N}(T)} \underline{p_{\mathrm{red}}^S(\boldsymbol{\mu})}\ \underline{E^{S,T,S'}(\boldsymbol{\mu})}\ \underline{p_{\mathrm{red}}^{S'}(\boldsymbol{\mu})}.\end{aligned} \tag{2.3.22}$$

Inserting (2.3.20) and (2.3.22) into (2.3.19) finally yields the following decomposition for the evaluation of the residual estimator η_{r}^T, for all $\boldsymbol{\mu} \in \mathcal{P}$ and all $T \in \mathcal{T}_H$:

$$\begin{aligned}\eta_{\mathrm{r}}^T\big(p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}\big) = \frac{1}{\sqrt{C_P^T / c_\varepsilon^T}} \Bigg(& \|f\|_{L^2,T}^2 + \Big(\underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})}\ \underline{E^T(\boldsymbol{\mu})} - 2\,\underline{d^T(\boldsymbol{\mu})}\Big) \cdot \underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})} \\ & + 2 \sum_{S \in \mathcal{N}(T)} \Big(\underline{p_{\mathrm{red}}^T(\boldsymbol{\mu})}\ \underline{E^{T,S}(\boldsymbol{\mu})} - \underline{d^{T,S}(\boldsymbol{\mu})}\Big) \cdot \underline{p_{\mathrm{red}}^S(\boldsymbol{\mu})} \\ & + \sum_{S \in \mathcal{N}(T)} \sum_{S' \in \mathcal{N}(T)} \underline{p_{\mathrm{red}}^S(\boldsymbol{\mu})}\ \underline{E^{S,T,S'}(\boldsymbol{\mu})}\ \underline{p_{\mathrm{red}}^{S'}(\boldsymbol{\mu})} \Bigg)^{1/2}.\end{aligned}$$

The diffusive flux estimator η_{df}^T. Similarly, using the definitions of η_{df}^T and the L^2 norm, we obtain for the diffusive flux estimator

$$\begin{aligned}\eta_{\mathrm{df}}^T\big(p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \hat{\boldsymbol{\mu}}\big)^2 &= \Big\| \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} \Big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla_h p_{\mathrm{red}}(\boldsymbol{\mu}) + R_h^l[p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\Big)\Big\|_{L^2,T}^2 \\ &= \underbrace{\Big\| \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} \big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla_h p_{\mathrm{red}}(\boldsymbol{\mu})\big)\Big\|_{L^2,T}^2}_{=:(i)} \\ &\quad + 2 \underbrace{\Big(\lambda(\boldsymbol{\mu})\kappa_\varepsilon \nabla_h p_{\mathrm{red}}(\boldsymbol{\mu}), \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1} R_h^l[p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\Big)_{L^2,T}}_{=:(ii)} \\ &\quad + \underbrace{\Big\| \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} R_h^l[p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}]\Big\|_{L^2,T}^2}_{=:(iii)},\end{aligned} \tag{2.3.23}$$

which leaves us with three terms we consider separately.

(i) We define the component matrices $\underline{F_{\xi,\xi'}^T}$ for all $0 \leq \xi, \xi' < \varXi$ and the parametric matrices $\underline{F^T(\boldsymbol{\mu})}$ for all $\boldsymbol{\mu} \in \mathcal{P}$ by

$$\big(\underline{F_{\xi,\xi'}^T}\big)_{i,j} := \Big(\big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1} \big(\lambda_\xi \kappa_\varepsilon \nabla_h \varphi_i^T\big), \lambda_{\xi'} \kappa_\varepsilon \nabla_h \varphi_j^T\Big)_{L^2,T}$$

and

$$\underline{F^T(\boldsymbol{\mu})} := \sum_{\xi=0}^{\Xi-1} \sum_{\xi'=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu}) \theta_{\xi'}(\boldsymbol{\mu}) \underline{F^T_{\xi,\xi'}},$$

respectively, for all $\varphi_i^T, \varphi_j^T \in \phi_{\text{red}}^T$ and all subdomains $T \in \mathcal{T}_H$. We thus obtain

$$(i) = \underline{p_{\text{red}}^T(\boldsymbol{\mu})}\ \underline{F^T(\boldsymbol{\mu})}\ \underline{p_{\text{red}}^T(\boldsymbol{\mu})}. \tag{2.3.24}$$

(ii) We define the component matrices $\underline{G^T_{\xi,\xi'}} \in \mathbb{R}^{n^T \times n^T}$ and $\underline{G^{T,S}_{\xi,\xi'}} \in \mathbb{R}^{n^T \times n^S}$ for all $0 \leq \xi, \xi' < \Xi$ and the parametric matrices $\underline{G^T(\boldsymbol{\mu})} \in \mathbb{R}^{n^T \times n^T}$ and $\underline{G^{T,S}(\boldsymbol{\mu})} \in \mathbb{R}^{n^T \times n^S}$ for all $\boldsymbol{\mu} \in \mathcal{P}$ by

$$\begin{aligned}
\big(\underline{G^T_{\xi,\xi'}}\big)_{i,j} &:= \Big(\lambda_{\xi'} \kappa_\varepsilon \nabla_h \varphi_i^T, \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1} R^T_{h,\xi'}[\varphi_j^T]\Big)_{L^2,T},\\
\big(\underline{G^{T,S}_{\xi,\xi'}}\big)_{i,j} &:= \Big(\lambda_{\xi'} \kappa_\varepsilon \nabla_h \varphi_i^T, \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1} R^{T,S}_{h,\xi'}[\varphi_j^S]\Big)_{L^2,T}
\end{aligned}$$

and

$$\begin{aligned}
\underline{G^T(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1} \sum_{\xi'=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu}) \theta_{\xi'}(\boldsymbol{\mu}) \underline{G^T_{\xi,\xi'}},\\
\underline{G^{T,S}(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1} \sum_{\xi'=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu}) \theta_{\xi'}(\boldsymbol{\mu}) \underline{G^{T,S}_{\xi,\xi'}},
\end{aligned}$$

respectively, for all $\varphi_i^T, \varphi_j^T \in \phi_{\text{red}}^T$, $\varphi_j^S \in \phi_{\text{red}}^S$ and all subdomains $T \in \mathcal{T}_H$ and their respective neighbors $S \in \mathcal{N}(T)$. We thus obtain

$$(ii) = \underline{p_{\text{red}}^T(\boldsymbol{\mu})}\ \underline{G^T(\boldsymbol{\mu})}\ \underline{p_{\text{red}}^T(\boldsymbol{\mu})} \ + \sum_{S \in \mathcal{N}(T)} \underline{p_{\text{red}}^T(\boldsymbol{\mu})}\ \underline{G^{T,S}(\boldsymbol{\mu})}\ \underline{p_{\text{red}}^S(\boldsymbol{\mu})}. \tag{2.3.25}$$

(iii) We define the component matrices $\underline{H^T_{\xi,\xi'}} \in \mathbb{R}^{n^T \times n^T}$, $\underline{H^{T,S}_{\xi,\xi'}} \in \mathbb{R}^{n^T \times n^S}$ and $\underline{H^{S,T,S'}_{\xi,\xi'}} \in \mathbb{R}^{n^S \times n^{S'}}$ for all $0 \leq \xi < \Xi$ and the parametric matrices $\underline{H^T(\boldsymbol{\mu})} \in \mathbb{R}^{n^T \times n^T}$, $\underline{H^{T,S}(\boldsymbol{\mu})} \in \mathbb{R}^{n^T \times n^S}$ and $\underline{H^{S,T,S'}(\boldsymbol{\mu})} \in \mathbb{R}^{n^S \times n^{S'}}$ for all $\boldsymbol{\mu} \in \mathcal{P}$ by

$$\begin{aligned}
\big(\underline{H^T_{\xi,\xi'}}\big)_{i,j} &:= \Big(\big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} R^{l,T}_{h,\xi}[\varphi_i^T], \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} R^{l,T}_{h,\xi'}[\varphi_j^T]\Big)_{L^2,T},\\
\big(\underline{H^{T,S}_{\xi,\xi'}}\big)_{i,j} &:= \Big(\big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} R^{l,T}_{h,\xi}[\varphi_i^T], \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} R^{l,T,S}_{h,\xi'}[\varphi_j^S]\Big)_{L^2,T},\\
\big(\underline{H^{S,T,S'}_{\xi,\xi'}}\big)_{i,j} &:= \Big(\big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} R^{l,T,S}_{h,\xi}[\varphi_i^S], \big(\lambda(\hat{\boldsymbol{\mu}})\kappa_\varepsilon\big)^{-1/2} R^{l,T,S'}_{h,\xi'}[\varphi_j^{S'}]\Big)_{L^2,T},
\end{aligned}$$

and

$$\begin{aligned}
\underline{H^T(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1}\sum_{\xi'=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\theta_{\xi'}(\boldsymbol{\mu})\underline{H^T_{\xi,\xi'}},\\
\underline{H^{T,S}(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1}\sum_{\xi'=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\theta_{\xi'}(\boldsymbol{\mu})\underline{H^{T,S}_{\xi,\xi'}},\\
\underline{H^{S,T,S'}(\boldsymbol{\mu})} &:= \sum_{\xi=0}^{\Xi-1}\sum_{\xi'=0}^{\Xi-1} \theta_\xi(\boldsymbol{\mu})\theta_{\xi'}(\boldsymbol{\mu})\underline{H^{S,T,S'}_{\xi,\xi'}},
\end{aligned}$$

respectively, for all $\varphi_i^T, \varphi_j^T \in \phi^T_{\text{red}}$, $\varphi_i^S, \varphi_j^S \in \phi^S_{\text{red}}$, $\varphi_j^{S'} \in \phi^{S'}_{\text{red}}$ and all subdomains $T \in \mathcal{T}_H$ and their respective neighbors $S, S' \in \mathcal{N}(T)$. We thus obtain

$$\begin{aligned}
(iii) = {} & \underline{p^T_{\text{red}}(\boldsymbol{\mu})}\ \underline{H^T(\boldsymbol{\mu})}\ \underline{p^T_{\text{red}}(\boldsymbol{\mu})}\\
&+ \sum_{S\in\mathcal{N}(T)} \underline{p^T_{\text{red}}(\boldsymbol{\mu})}\ \underline{H^{T,S}(\boldsymbol{\mu})}\ \underline{p^S_{\text{red}}(\boldsymbol{\mu})}\\
&+ \sum_{S\in\mathcal{N}(T)}\sum_{S'\in\mathcal{N}(T)} \underline{p^S_{\text{red}}(\boldsymbol{\mu})}\ \underline{H^{S,T,S'}(\boldsymbol{\mu})}\ \underline{p^{S'}_{\text{red}}(\boldsymbol{\mu})}.
\end{aligned} \tag{2.3.26}$$

Inserting (2.3.24), (2.3.25) and (2.3.26) into (2.3.23) finally yields the following decomposition for the evaluation of the diffusive flux estimator η^T_{df}, for all $\boldsymbol{\mu} \in \mathcal{P}$ and all $T \in \mathcal{T}_H$:

$$\begin{aligned}
\eta^T_{\text{df}}\big(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \hat{\boldsymbol{\mu}}\big) = \Bigg(& \underline{p^T_{\text{red}}(\boldsymbol{\mu})}\Big(\underline{F^T(\boldsymbol{\mu})} + \underline{G^T(\boldsymbol{\mu})} + \underline{H^T(\boldsymbol{\mu})}\Big)\underline{p^T_{\text{red}}(\boldsymbol{\mu})}\\
&+ 2\sum_{S\in\mathcal{N}(T)} \underline{p^T_{\text{red}}(\boldsymbol{\mu})}\Big(\underline{G^{T,S}(\boldsymbol{\mu})} + \underline{H^{T,S}(\boldsymbol{\mu})}\Big)\underline{p^S_{\text{red}}(\boldsymbol{\mu})}\\
&+ \sum_{S\in\mathcal{N}(T)}\sum_{S'\in\mathcal{N}(T)} \underline{p^S_{\text{red}}(\boldsymbol{\mu})}\ \underline{H^{S,T,S'}(\boldsymbol{\mu})}\ \underline{p^{S'}_{\text{red}}(\boldsymbol{\mu})}\Bigg)^{1/2}.
\end{aligned}$$

Given these decompositions of the local indicators, the corresponding offline/online decomposition of the estimator η_{red} is straightforward. Note that the computational complexity of computing the estimators during the online phase scales quadratically with the number of components of the affine decomposition of λ, Ξ.

Now that we know how to compute high-dimensional snapshots and how to estimate the error of the discrete as well as the reduced solution, the definition of the LRBMS is nearly complete. For the remainder of this chapter, we therefore discuss how to generate the local reduced bases offline and how to adaptively enrich them online.

2.4 Adaptivity

We have already identified several challenges in the context of parametric multiscale problems, where traditional RB methods reach their limits (see Section 1.4). These challenges are related to the fact that the dimension of the approximation space scales with ε^{-l} for some $l \geq 1$, which results in a very expensive offline part of the computational process. We address this issue by localizing many aspects of the offline part, for instance by a combination of the greedy algorithm using localized snapshots and an a priori choice of local reduced basis functions (see Section 2.4.1).

In addition, we have already discussed another shortcoming of traditional RB methods with respect to the training character of the greedy algorithm (see Section 1.3): we can only expect the greedy algorithm to produce a good reduced approximation space (in the sense of the Kolmogorov n-width), if we allow it to search over a large set of training parameters, $\mathcal{P}_{\text{train}} \subset \mathcal{P}$, and if we allow it to add a certain amount of solution snapshots to the reduced basis. In general, we cannot say anything about the approximation quality of the generated reduced space for untrained parameters, $\mathcal{P} \backslash \mathcal{P}_{\text{train}}$. While we can assess the approximation error online, we have no means to improve the approximation quality of the reduced basis online, without dropping back to high-dimensional computations.

The latter is particularly troublesome in the context of parametric multiscale problems, where a single solution snapshot is already very costly and where we usually do not have the computational power to generate an acceptable reduced space offline. We address this issue by an adaptive enrichment of the local reduced bases during the online part of the computational process (see Section 2.4.2). While we allow for high-dimensional computations online (and thus break the traditional offline/online decomposition), we only require computations that are local to a subdomain.

2.4.1 Offline basis generation

As discussed in Section 1.3, there exist several variants of the greedy algorithm, depending on the circumstances and requirements. In many multiscale scenarios, the fine grid τ_h, and thus the high-dimensional approximation space $Q_h^k(\tau_h)$, is prescribed a priori (see for instance the experiment in Section 4.1.2). A natural choice for these kind of problems is the discrete weak greedy algorithm 1.3.11, which we adapt to the LRBMS setting. We do so by using the residual-based a-posterior estimate on the model reduction error from Theorem 2.3.1, η_{red}, and by restricting the snapshots to each subdomain:

$$\texttt{extend}(\phi_{\text{red}}^{(n)}, \boldsymbol{\mu}^*) := \cup_{T \in \mathcal{T}_H} \texttt{ONB}(\phi_{\text{red}}^{T(n)} \cup p_{\varepsilon,h}(\boldsymbol{\mu}^*)|_T), \quad \text{with } \phi_{\text{red}}^{(n)} = \cup_{T \in \mathcal{T}_H} \phi_{\text{red}}^{T(n)}.$$

This variant of the greedy algorithm produces a reduced space with better approximation properties than that of standard RB methods while requiring less global snapshots (see the experiments in section 4.1). In addition, the local orthonormalization procedures operate on much smaller quantities and can be carried out in parallel. This step of the offline part is thus much cheaper than the global orthonormalization step of the traditional discrete weak greedy algorithm. On the other hand, the offline part suffers from the bad scaling of the offline/online decomposition of the residual-based error estimator with respect to the coarse grid, which undermines our efforts of reducing the overall offline cost.

We addressed this drawback by using the localized a posteriori error estimate, $\eta_{h,\text{red}}$, from Corollary 2.3.6. We could use this estimate on the full error, together with the localized estimate on the discretization error from [ESV2010], to drive the spatio-parameter greedy Algorithm 1.3.12, which adaptively refines the discrete space $Q_h^k(\tau_h)$ along with the reduced space $Q_\text{red}(\mathcal{T}_H)$. However, since the idea of the spatio-parameter greedy algorithm has only recently been proposed and since there still remain some practical aspects to be considered in the context of the LRBMS, we postpone this ansatz for future work. We thus use the estimate on the full error in the above context where a fixed approximation space $Q_h^k(\tau_h)$ is given, which requires some careful considerations: since the full error of a reduced solution can never be lower than the discretization error induced by the prescribed approximation space $Q_h^k(\tau_h)$, say $\Delta_h > 0$, we cannot expect a greedy algorithm to reach any tolerance below that, $\Delta_\text{red} < \Delta_h$ (see also Definition 2.4.1 below).

In general we do not have any information about the solution manifold a priori and it thus does not make sense to initialize the reduced basis with problem independent functions a-priori. The most natural choice therefore is to initialize the reduced basis with the largest element of the solution manifold (see the variants of the greedy algorithms in Section 1.3). In the context of the LRBMS, however, we are not interested in generating one reduced basis with global support, but instead many local reduced bases, associated with the subdomains of the coarse grid. Inspired by numerical multiscale methods (see Section 1.2) and the interpretation of the LRBMS as a generalized DG method with respect to the coarse grid, we initialize each local reduced basis with DG shape functions of order up to k_H, for some $k_H \in \mathbb{N}$. We thus ensure that any LRBMS approximation is at least as good as a standard DG method of order k_H on the coarse grid. In addition, the coarse behavior of the reduced solutions will be captured by the DoFs associated with these basis functions, while all subsequent local reduced basis functions can be interpreted as local fine scale corrections (see also the discussion of numerical multiscale methods in Section 1.2.2).

Definition 2.4.1 (Discrete weak greedy algorithm in the context of the LRBMS)**.** *Let $\mathcal{P}_\text{train} \subset \mathcal{P}$ be a finite set and let $p_{\varepsilon,h}(\boldsymbol{\mu})$ and $p_\text{red}(\boldsymbol{\mu})$ denote the solutions of the discrete Problem 2.1.1 and the reduced Problem 2.2.1, respectively, for $\boldsymbol{\mu} \in \mathcal{P}$. Let further $\eta_h(\cdot;\cdot,\overline{\boldsymbol{\mu}},\hat{\boldsymbol{\mu}})$ and $\eta_{h,\text{red}}(\cdot;\cdot,\overline{\boldsymbol{\mu}},\hat{\boldsymbol{\mu}})$ denote the localizable a-posterior estimates on the full error from Corollary 2.3.6 for some fixed $\overline{\boldsymbol{\mu}},\hat{\boldsymbol{\mu}} \in \mathcal{P}$ and let $k_H \in \mathbb{N}$. With the maximum estimated discretization error denoted by $\Delta_h := \max_{\boldsymbol{\mu}\in\mathcal{P}_\text{train}} \eta_h\big(p_{\varepsilon,h}(\boldsymbol{\mu});\boldsymbol{\mu},\overline{\boldsymbol{\mu}},\hat{\boldsymbol{\mu}}\big)$, the* discrete weak greedy algorithm *for the construction of a local reduced basis ϕ_red^T spanning a local reduced space space Q_red^T on each subdomain $T \in \mathcal{T}_H$, to approximate $IO_h(\mathcal{P}_\text{train})$ with an accuracy $\Delta_\text{red} > \Delta_h$, is then given by Algorithm 1.3.10 with*

$$\texttt{init(}\mathcal{P}_\text{train}\texttt{)} := \cup_{T\in\mathcal{T}_H}\big\{\textit{DG shapefunctions of order up to } k_H \textit{ w.r.t } T\big\}$$

$$\texttt{estimate(}\phi_\text{red}^{(n)},\boldsymbol{\mu}\texttt{)} := \eta_{h,\text{red}}\big(p_\text{red}^{(n)}(\boldsymbol{\mu});\boldsymbol{\mu},\overline{\boldsymbol{\mu}},\hat{\boldsymbol{\mu}}\big),$$

$$\texttt{extend}(\phi_{\text{red}}^{(n)}, \boldsymbol{\mu}^*) \;:= \cup_{T\in\mathcal{T}_H} \texttt{ONB}(\phi_{\text{red}}^{T(n)} \cup p_{\varepsilon,h}(\boldsymbol{\mu}^*)|_T), \textit{ with } \phi_{\text{red}}^{(n)} = \cup_{T\in\mathcal{T}_H} \phi_{\text{red}}^{T(n)},$$

where `ONB` *denotes an orthonormalization procedure for improved numerical stability, for instance a stabilized Gram Schmidt procedure (see also Section 3.2.4.1).*

In order to bring down the computational cost of the offline part we can use the above greedy algorithm while allowing only for a very limited amount of global solution snapshots. This will result in a relatively small reduced space that may well be insufficient for most parameters (even most training parameters).

We are thus now in the situation already outlined in the beginning of this chapter:

> *We are not given sufficient resources to generate an appropriate reduced basis offline and we would thus like to adaptively improve the reduced basis online, while maintaining as much of the offline/online decomposition of the computational process as possible.*

2.4.2 Online basis enrichment

Online, given any $\boldsymbol{\mu} \in \mathcal{P}$, we compute a reduced solution $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}(\mathcal{T}_H)$ and efficiently assess its quality using the error estimator $\eta_{h,\text{red}}$. With traditional RB methods, we have no means to improve the quality of the reduced solution, if the estimated error was too large. The localized nature of the LRBMS, however, allows us to improve the quality of the reduced solution during the online phase by carrying out an intermediate local enrichment step in the SEMR ($\underline{\text{s}}$olve $\to$ $\underline{\text{e}}$stimate $\to$ $\underline{\text{m}}$ark $\to$ $\underline{\text{r}}$efine) spirit of adaptive FE methods (see Section 1.1). Instead of enlarging the approximation space by local grid adaptation, however, we enrich the local reduced bases by solving local corrector problems on selected subdomains, motivated by domain decomposition and numerical multiscale methods (the procedure is summarized in Algorithm 2.4.2): we first compute local error indicators $\eta_{h,\text{red}}^T(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})$ for all $T \in \mathcal{T}_H$, such that $\eta_{h,\text{red}}(\cdot; \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})^2 \leq \sum_{T\in\mathcal{T}_H} \eta_{h,\text{red}}^T(\cdot; \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})^2$, defined as

$$\eta_{h,\text{red}}^T(\cdot; \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})^2 := \frac{3}{\sqrt{\alpha(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}} \Big[\sqrt{\gamma(\boldsymbol{\mu},\overline{\boldsymbol{\mu}})}\, \eta_{\text{nc}}^T(\cdot; \overline{\boldsymbol{\mu}})^2 + \eta_{\text{r}}^T(\cdot; \boldsymbol{\mu})^2 + \frac{1}{\sqrt{\alpha(\boldsymbol{\mu},\hat{\boldsymbol{\mu}})}}\, \eta_{\text{df}}^T(\cdot; \boldsymbol{\mu}, \hat{\boldsymbol{\mu}})^2 \Big], \tag{2.4.1}$$

and mark subdomains $\tilde{\mathcal{T}}_H \subseteq \mathcal{T}_H$ for enrichment, given a marking strategy `MARK`. We propose and discuss several such strategies in Section 4.4. On each marked subdomain $T \in \tilde{\mathcal{T}}_H$ we solve the local corrector problem

$$b_{\varepsilon,h}^{T_\delta}(p_{\varepsilon,h}^{T_\delta}(\boldsymbol{\mu}), q_h; \boldsymbol{\mu}) = l_h^{T_\delta}(q_h) \qquad \text{for all } q_h \in Q_h^k(\tau_h^{T_\delta}) \tag{2.4.2}$$

on an overlapping domain $T_\delta \supset T$ with the insufficient reduced solution $p_{\text{red}}(\boldsymbol{\mu})$ as dirichlet boundary values on ∂T_δ to obtain an updated detailed solution $p_h^{T_\delta}(\boldsymbol{\mu}) \in Q_h^k(\tau_h^{T_\delta})$. Here $Q_h^k(\tau_h^{T_\delta})$, $b_{\varepsilon,h}^{T_\delta}$ and $l_h^{T_\delta}$ are extensions of $Q_h^k(\tau_h^T)$, $b_{\varepsilon,h}^T$ and l^T, respectively, to the overlapping domain T_δ while additionally encoding $p_{\text{red}}(\boldsymbol{\mu})$ as dirichlet boundary values.

We then extend each marked local reduced basis by performing an orthonormalization procedure on $p_{\varepsilon,h}^{T_\delta}(\boldsymbol{\mu})\big|_T$ with respect to the existing local reduced basis and update all reduced quantities with respect to the newly added basis vector. Finally, we compute an updated reduced solution using the updated coarse reduced space and estimate the error again. We repeat this procedure until the estimated error falls below the prescribed tolerance Δ_{red} or until the prescribed maximum number of iterations, $n_{\text{ext}} \in \mathbb{N}$, is reached.

Algorithm 2.4.2 Adaptive basis enrichment in the intermediate local enrichment phase.

```
Input: MARK, ONB, $\{\phi_{\text{red}}^T\}_{T\in\mathcal{T}_H}$, $p_{\text{red}}(\boldsymbol{\mu})$, $\boldsymbol{\mu}$, $\Delta_{\text{red}} > \Delta_h$, $n_{\text{ext}} \in \mathbb{N}$
Output: Updated reduced solution and local reduced bases.
  $\phi_{\text{red}}^{T\,(0)} \leftarrow \phi_{\text{red}}^T$, $\forall T \in \mathcal{T}_H$, $n \leftarrow 0$
  while $\eta_{h,\text{red}}(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) > \Delta_{\text{red}}$ and $n < n_{\text{ext}}$ do
    for all $T \in \mathcal{T}_H$ do
      Compute local error indicator $\eta_{h,\text{red}}^T(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})$ according to (2.4.1).
    end for
    $\tilde{\mathcal{T}}_H \leftarrow$ MARK$(\mathcal{T}_H)$
    for all $T \in \tilde{\mathcal{T}}_H$ do
      Solve (2.4.2) for $p_{\varepsilon,h}^{T_\delta}(\boldsymbol{\mu}) \in Q_h^k(\tau_h^{T_\delta})$.
      $\phi_{\text{red}}^{T\,(n+1)} \leftarrow$ ONB$\big(\{\phi_{\text{red}}^{T\,(n)}, p_{\varepsilon,h}^{T_\delta}(\boldsymbol{\mu})\big|_T\}\big)$
    end for
    $Q_{\text{red}}(\mathcal{T}_H)^{(n+1)} \leftarrow \bigoplus_{T\in\tilde{\mathcal{T}}_H} \text{span}\big(\phi_{\text{red}}^{T\,(n+1)}\big) \oplus \bigoplus_{T\in\mathcal{T}_H\setminus\tilde{\mathcal{T}}_H} \text{span}\big(\phi_{\text{red}}^{T\,(n)}\big)$
    Update all reduced quantities w.r.t $Q_{\text{red}}(\mathcal{T}_H)^{(n+1)}$.
    Solve (2.2.4) for $p_{\text{red}}(\boldsymbol{\mu}) \in Q_{\text{red}}(\mathcal{T}_H)^{(n+1)}$.
    $n \leftarrow n + 1$
  end while
  return $p_{\text{red}}(\boldsymbol{\mu})$, $\{\phi_{\text{red}}^{T\,(n)}\}_{T\in\mathcal{T}_H}$
```

There are several things worth noting about our modification of the online part of the computational process. First of all, if the approximation quality of the reduced space is sufficient for all parameters of interest, we do not interrupt the online phase at all; it is then completely analogous to the online phase of traditional RB methods. The computation of the local error indicators in Algorithm 2.4.2 can be efficiently offline/online decomposed (see above); the computational complexity of `MARK` only depends on the number of subdomains, $|\mathcal{T}_H|$. Once a set of subdomains has been marked, the enrichment can be carried out in parallel with respect to the marked subdomains, without any communication. For the update of the reduced quantities only local information and the information on one layer of neighboring fine grid cells is needed.

The LRBMS method with the proposed basis generation Algorithm 2.4.1 and the adaptive online enrichment strategy from Algorithm 2.4.2 is now suitable for a far wider range of circumstances than standard RB methods or the initially published variants of the LRBMS method [AHKO2012]. As mentioned before it can now be applied if the computational power available for the offline phase is limited by time- or resource constraints. It can also be applied if the set of training parameters given to the greedy Algorithm was insufficiently chosen or even if online a solution to a parameter is requested that is outside of the original bounds of the parameter space. In general, the online adaptive LRBMS method can be applied whenever the basis that was generated during the offline phase turns out to not be sufficient for what is required during the online phase.

Remark 2.4.3. *Our choice of the greedy Algorithm 2.4.1 and the adaptive online enrichment Algorithm 2.4.2 covers a wide range of scenarios. Disabling the online enrichment (by setting $n_{ext} = 0$) and choosing any suitable Δ_{red} and n_{red} yields the standard discrete weak greedy basis generation. Setting $n_{\text{red}} = 0$ and $k_H = 1$, on the other hand, disables the greedy procedure and merely initializes the reduced bases with the coarse DG basis of order one. This is of particular interest in situations where the computation of solutions of the detailed problem during the greedy procedure might be too costly. In that setup nearly all work is done in the adaptive online enrichment phase.*

Many other variants of Algorithm 2.4.2 are possible, e.g. other local boundary conditions, other marking strategies `MARK` or orthonormalization algorithms `ONB` or other stopping criteria; one could also limit the number of intermediate snapshots added to the local bases. Depending on these choices the resulting method is then close to existing DD methods (i.e., a DD method with overlapping subdomains, see [QV1999]) or numerical multiscale methods (i.e., the adaptive iterative multiscale finite volume method, see [HJ2011] or Section 1.2).

3 Software concepts and implementations

In this chapter we present and discuss the discretization and the model reduction framework, which were utilized to conduct the experiments in Chapter 4. Scientific software plays a crucial role in research and is vital for a practical confirmation of theoretical results as well as for an experimental justification of algorithms, which have not yet been fully studied theoretically. For each of the two frameworks, we give a thorough mathematical description of the underlying theoretical requirements (Sections 3.1.1 and 3.2.1), which naturally lead to the abstract design principles for each framework (Sections 3.1.2 and 3.2.3). We also discuss existing software frameworks (Sections 3.1.3 and 3.2.2) and present our own implementation of either framework in detail (Sections 3.1.4 and 3.2.4).

3.1 Discretization framework

An implementation of a discretization is at the heart of many numerical algorithms, be it the adaptive grid-based discretizations discussed in Section 1.1, the numerical multiscale methods discussed in Section 1.2 or the model reduction techniques discussed in Section 1.4.

However, the implementation of a discretization framework as a library of building blocks for many different circumstances is not a trivial task. We present a realization of a discretization framework, the purpose of which is to provide such a library for rapid prototyping of new discretization schemes as well as for well performing discretizations for highly complex real-world problems.

This section is organized as follows: we examine the different circumstances, where a discretization framework is required, and define all relevant mathematical concepts in Section 3.1.1. Based on these requirements, we derive abstract design principles for a designated discretization framework in Section 3.1.2 and discuss possible available candidates in Section 3.1.3. Since none of those met our needs, we present a new discretization framework in Section 3.1.4, which is centered around the **DUNE** module `dune-gdt`.

3.1.1 Mathematical foundation and theoretical requirements

We recall what we require (mathematically speaking) of a discretization framework.

- Most prominently: given data functions, we require an approximation of the solution of a partial differential equation (PDE). This approximation involves a partition of the computational domain, an approximation of integrals, a discrete representation of the solution and the solution of linear systems, among other ingredients (see Section 3.1.1.1).

- For a given solution we require an assessment on the error introduced by the approximation, either by comparison with a known exact solution, a more detailed approximation or by means of a posteriori error estimation (see Section 3.1.1.2).
- We also require direct access to discrete counterparts of operators, bilinear forms, products and functionals. This is especially the case in the context of model reduction (see Section 3.2).

We shall examine these different scenarios and requirements in the following sections and derive key concepts along the way.

3.1.1.1 Approximating the solution of a partial differential equation

Following Section 1.1, we consider an elliptic problem as in Example 1.1.2, but allow for more general types of boundary values. Let therefore $\Omega \subset \mathbb{R}^d$, for $d = 1, 2, 3$, denote a bounded connected domain with polygonal boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$, which is separated into a Dirichlet boundary Γ_D and a Neumann boundary Γ_N, such that $\Gamma_D \cap \Gamma_N = \emptyset$ and $\Gamma_D \neq \emptyset$. Let additionally $H^1(\Omega)$ denote the Sobolev space of weakly differentiable functions over Ω and $H^1_{\Gamma_D}(\Omega) \subset H^1(\Omega)$ its elements that vanish on the Dirichlet boundary Γ_D in the sense of traces.

Given a force $f \in L^2(\Omega)$, a diffusion factor $\lambda \in L^\infty(\Omega)$, a diffusion tensor $\kappa \in [L^\infty(\Omega)]^{d\times d}$, Dirichlet boundary values $g_D \in L^2(\Gamma_D)$ and Neumann boundary values $g_N \in L^2(\Gamma_N)$, we are looking for a pressure $p \in H^1(\Omega)$, such that

$$\begin{aligned} -\nabla\cdot(\lambda\kappa\nabla p) &= f && \text{in } \Omega,\\ p &= g_D && \text{on } \Gamma_D \text{ and}\\ (\lambda\kappa\nabla p)\cdot n &= g_N && \text{on } \Gamma_N \end{aligned}$$

in a weak sense in $H^1_{\Gamma_D}(\Omega)$. Equivalently put (in a variational setting), we are looking for $p \in H^1(\Omega)$ with $p = g_D$ on Γ_D, such that

$$b(q,p) = l_f(q) + l_{g_N}(q) \qquad \text{for all } q \in H^1_{\Gamma_D}(\Omega), \tag{3.1.1}$$

with the bilinear form $b : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ and the linear functionals $l_f : L^2(\Omega) \to \mathbb{R}$ and $l_N : L^2(\Gamma_N) \to \mathbb{R}$ defined as

$$b(q,p) := \int_\Omega (\lambda\kappa\nabla q)\cdot\nabla p \,\mathrm{d}x, \quad l_f(q) := \int_\Omega f q \,\mathrm{d}x \quad \text{and} \quad l_{g_N}(q) := \int_{\Gamma_N} g_N q \,\mathrm{d}x, \tag{3.1.2}$$

respectively.

The classes of discretizations we consider, namely *Finite Element methods* (FEM), are based on a suitable partition of the computational domain Ω by a grid τ_h and a Galerkin projection of (3.1.1) onto a discrete function space associated with this grid.[1] To proceed further, however, we need to differentiate between a conforming approximation, where the approximating space is a subspace (e.g., $S^k_h(\tau_h) \subset H^1(\Omega)$), or a nonconforming one, where this is not the case (e.g., $Q^k_h(\tau_h) \subset L^2(\Omega)$ with $Q^k_h(\tau_h) \not\subset H^1(\Omega)$).

[1]See Section 2.1 for a definition of a grid.

A conforming approximation: continuous Galerkin FEM. We consider the operator $B : H^1(\Omega) \to H^{-1}(\Omega)$, $p \mapsto [q \mapsto B[p](q) := b(q,p)]$ corresponding to the bilinear form b, and express the solution p in terms of its homogeneous part $p_0 \in H^1_{\Gamma_D}(\Omega)$ and a Dirichlet shift $g_{h,D} \in H^1(\Omega)$. Problem (3.1.1) is then equivalent to: find $p_0 \in H^1_{\Gamma_D}(\Omega)$, such that

$$B[p_0] = l_f + l_{g_N} - B[g_{h,D}] \qquad \text{in } H^1_{\Gamma_D}(\Omega), \tag{3.1.3}$$

where $g_{h,D}$ is any representation of g_D, such that $g_{h,D}|_{\Gamma_D} = g_D$ in the sense of traces (we will specify the choice of $g_{h,D}$ further below). The weak solution is then given by $p := p_0 + g_{h,D}$.

In order to discretize (3.1.3), we introduce two approximations by substituting the ansatz space by a finite dimensional subspace and the operator B and the two functionals l_f and l_{g_N} by discrete variants; both approximations are induced by the choice of the grid τ_h. (We think of grids as introduced in [BBD+2008] and use the notation of Chapter 2, that is: a grid is denoted by τ_h, its elements are denoted by $t \in \tau_h$ and its faces are denoted by $e \in \mathcal{F}_h$.) To begin with, presuming we are given a grid, let $S^k_h(\tau_h) \subset H^1(\Omega)$ denote the continuous Lagrange space of globally continuous and piecewise polynomial functions of order $k \in \mathbb{N}$, $k \geq 1$,

$$S^k_h(\tau_h) := \big\{ q \in C^0(\Omega) \;\big|\; q|_t \in \mathbb{P}_k(t) \quad \forall t \in \tau_h \big\}, \tag{3.1.4}$$

where $\mathbb{P}_k(\omega)$ denotes the set of polynomials of order at most k for any set $\omega \subset \Omega$. (For ease of notation we shall frequently drop the explicit polynomial order, where appropriate, and denote by $\mathbb{P}(\omega)$ the set of polynomials of at most *some* fixed finite order, which does not have to be the same at each occurrence.) The Galerkin projection of (3.1.3) onto S^k_h then simply reads: find $p_{h,0} \in S^k_h(\tau_h) \cap H^1_{\Gamma_D}(\Omega)$, such that

$$B[p_{h,0}] = l_f + l_{g_N} - B[g_{h,D}] \qquad \text{in } S^k_h(\tau_h) \cap H^1_{\Gamma_D}(\Omega). \tag{3.1.5}$$

Before we proceed any further, let us consider the relationship between the elliptic bilinear form b and the elliptic operator B.

Remark 3.1.1 (On operators and two-forms)**.** *We think of an operator L as a mapping between Hilbert-spaces V and W, that is: $L : V \to W'$. We can also interpret each operator as a two-form acting on V and W:*

$$(\cdot,\cdot)_L : W \times V \to \mathbb{R}, \qquad (w,v)_L := L[v](w) \qquad \textit{for all } v \in V, w \in W.$$

If L is linear, the corresponding two-form is a bilinear form.

It is thus equivalent to consider (3.1.5), which is posed in operator notation, or the following problem, which is expressed using the bilinear form: find $p_{h,0} \in S^k_h(\tau_h) \cap H^1_{\Gamma_D}(\Omega)$, such that

$$b(q_h, p_{h,0}) = l_f(q_h) + l_{g_N}(q_h) - b(q_h, g_{h,D}) \qquad \text{for all } q_h \in S^k_h(\tau_h) \cap H^1_{\Gamma_D}(\Omega).$$

To proceed with the approximation of the PDE we can consider either problem but mainly prefer operator notation in this section. It will only be important to clearly separate operators and two-forms in the context of the implementation, compare Section 3.1.4.2.

Now that we have approximated H^1 by the discrete (finite dimensional) space $S_h^k(\tau_h)$, we require an approximation of the operator and the functionals by discrete counterparts $B_h \approx B$, $l_{h,f} \approx l_f$ and $l_{h,g_N} \approx l_{g_N}$, since B, l_f and l_{g_N} act on $H^1(\Omega)$ and involve integrals which need to be approximated. We obtain these discrete counterparts by a localization of the integrals with respect to τ_h and by numerical quadratures to approximate the local integrals on each element of the grid. In order to define the correct discrete operator, we start by localizing B:

$$B[\psi](\varphi) = \sum_{t \in \tau_h} \int_t (\lambda \kappa \nabla \psi) \cdot \nabla \varphi \, \mathrm{d}x. \tag{3.1.6}$$

The next step would be to approximate the above local integrals by a numerical quadrature on each grid element $t \in \tau_h$. However, providing quadratures for arbitrary elements $t \in \tau_h$ is not an easy task (the same holds for the local shape functions that are required for the local basis functions, as we will see below). Most discretization frameworks thus presume that each element of the grid, $t \in \tau_h$, is defined in terms of a reference element $\hat{t}$ and a bijective map $\varPhi^t : \hat{t} \to t$ and we follow this assumption.[2] Using this map and the definition $\Lambda_t^2 := |\det(\nabla {\varPhi^t}^{\perp} \nabla \varPhi^t)|$ we transform the integral in (3.1.6) and obtain

$$\int_t \big(\lambda \kappa \nabla \psi\big) \cdot \nabla \varphi \, \mathrm{d}x = \int_{\hat{t}} \Lambda_t \big((\lambda \circ \varPhi^t)(\kappa \circ \varPhi^t)(\nabla \psi \circ \varPhi^t)\big) \cdot (\nabla \varphi \circ \varPhi^t) \, \mathrm{d}x. \tag{3.1.7}$$

Definition 3.1.2 (Local function, local derivative)**.** *Let τ_h be a grid of Ω and let $\varPhi^t : \hat{t} \to t$ denote the reference map for all $t \in \tau_h$. Let further $\hat{\mathbb{P}}(t) := \{\varphi^t = \varphi \circ \varPhi^t \mid \varphi \in \mathbb{P}(t)\}$ denote the set of localized polynomials for $t \in \tau_h$. Given a function $\varphi : t \to \mathbb{R}$, for $t \in \tau_h$, we call the function $\varphi^t := \varphi \circ \varPhi^t : \hat{t} \to \mathbb{R}$ a* local function*, iff there exists a finite polynomial degree (which may depend on t), such that $\varphi^t \in \hat{\mathbb{P}}(t)$. Given a derivative operator D acting on φ, we define the* local derivative *of φ^t by $D_t \varphi^t := D\varphi \circ \varPhi^t$. In particular, for a local function $\varphi^t \in \hat{\mathbb{P}}(t)$, we define its* local gradient *by $\nabla_t \varphi^t := \nabla \varphi \circ \varPhi^t \in [\hat{\mathbb{P}}(t)]^d$.*[3]

Remark 3.1.3. *Note that the local gradient of a local function, $\nabla_t \varphi^t$, does not coincide with the gradient of a local function, $\nabla \varphi^t$. Using the chain rule we obtain for the latter*

$$\nabla \varphi^t = \nabla(\varphi \circ \varPhi^t) = (\nabla \varphi \circ \varPhi^t) \nabla \varPhi^t,$$

while the following holds for the former:

$$\nabla_t \varphi^t = \nabla \varphi \circ \varPhi^t = \nabla(\varphi \circ \varPhi^t)(\nabla \varPhi^t)^{-1} = \nabla \varphi^t (\nabla \varPhi^t)^{-1}.$$

[2]The reference element $\hat{t}$ is usually given by the unit cube or simplex on which quadratures and shape functions can be constructed.

[3]Note that the concept of localization naturally holds for vector- and matrix-valued functions as well. For ease of notation, however, we only discuss the scalar case in this chapter (unless noted otherwise).

Nevertheless, the local gradient of a local function is a useful concept, as it allows to rewrite (3.1.7) in a more intuitive way, preserving the structure of the integrand:

$$\int_t \big(\lambda\kappa\nabla\psi\big)\cdot\nabla\varphi\,\mathrm{d}x = \int_{\hat{t}} \Lambda_t \underbrace{\big(\lambda^t\kappa^t\nabla_t\psi^t\big)\cdot\nabla_t\varphi^t}_{=:(*)}\,\mathrm{d}x. \tag{3.1.8}$$

We transform the local integrals required for the approximation of the linear functional l_f from (3.1.2) in a similar manner:

$$\int_t f\psi\,\mathrm{d}x = \int_{\hat{t}} \Lambda_t \underbrace{f^t\psi^t}_{=:(**)}\,\mathrm{d}x. \tag{3.1.9}$$

In the above integrals, we implicitly assumed the existence of local functions λ^t, κ^t and f^t, given the data functions λ, κ and f. We formalize this assumption by demanding all functions involved to be localizable from here on.

Definition 3.1.4 (Localizable function). *Let τ_h be a grid of Ω and let $q : \Omega \to \mathbb{R}$ be a function. We call q* localizable *with respect to τ_h, iff it can be expressed as a local function on all elements of the grid. To be more precise, we call q localizable, iff $q \in Q(\tau_h)$, where*

$$Q(\tau_h) := \big\{q : \Omega \to \mathbb{R} \;\big|\; \forall t \in \tau_h \;\; \exists k(t) \in \mathbb{N}, \text{ such that } q^t = q|_t \circ \Phi^t \in \hat{\mathbb{P}}_{k(t)}(t)\big\}$$

denotes the vector space of functions which are localizable with respect to τ_h.

By requiring all data functions involved to be localizable, we can always transform local integrals as in (3.1.8) and (3.1.9) and obtain polynomial integrands.

Remark 3.1.5 (Local functions and quadratures). *While only considering locally polynomial data functions might seem restrictive, note that by using numerical quadratures we can only integrate polynomials exactly. In practice we are thus bound by the quadrature of largest available order, anyway. In addition, non-polynomial data functions such as $q(x) = \sin(x)$ can still be approximated arbitrarily well (given appropriate quadratures) by specifying large enough local polynomial degrees. The problem of quadrature errors is thus shifted towards the specification of appropriate local polynomial degrees of the data functions.*

In order to properly define the discrete operator and functionals we return to the investigation of the transformed local integrals (3.1.8) and (3.1.9). The respective integrand is composed of the transformation Λ_t and a local evaluation ($*$ and $**$, respectively), motivated by the form of b and l_f in (3.1.2). The question of finding an appropriate numerical quadrature in order to approximate the integrals over the reference element $\hat{t}$, however, does only depend on the polynomial degree of the integrand, not on its actual form. Since we are seeking an approximation of (3.1.8) and (3.1.9) by a (local) discrete operator and functional, say

$$B_h^t[\varphi^t](\psi^t) \approx \int_{\hat{t}} \Lambda_t \underbrace{\big(\lambda^t\kappa^t\nabla_t\psi^t\big)\cdot\nabla_t\varphi^t}_{=(*)}\,\mathrm{d}x \qquad \text{and} \qquad l_{h,f}^t(\psi^t) \approx \int_{\hat{t}} \Lambda_t \underbrace{f^t\psi^t}_{=(**)}\,\mathrm{d}x,$$

the main difference between the two local evaluations $*$ and $**$ is the number of local basis functions they depend on.

Definition 3.1.6 (Local binary volume evaluation and local volume integral operator)**.** *Let* τ_h *be a grid and let* $\Upsilon_2^t : \hat{\mathbb{P}}(t) \times \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)$ *denote a* local binary volume evaluation.[4] *We define the* local volume integral operator $\Sigma_h^t\big(\Upsilon_2^t\big) : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)'$ *by*

$$\Sigma_h^t\big(\Upsilon_2^t\big)[\varphi^t](\psi^t) := \sum_{n=0}^{N(\Upsilon_2^t)-1} w_n^t \Lambda_t(x_n^t) \Upsilon_2^t\big(\psi^t, \varphi^t\big)(x_n^t),$$

where w_n^t *and* x_n^t*, for all* $0 \leq n < N(\Upsilon_2^t) \in \mathbb{N}$*, denote the quadrature weights and points, respectively, in order to integrate* $\Lambda_t \Upsilon_2^t\big(\psi^t, \varphi^t\big)$ *exactly.*

We finally obtain the local elliptic operator B_h^t to approximate the integral in (3.1.8) by combining the abstract local volume integral operator Σ_h^t with the appropriate local evaluation.

Example 3.1.7 (Local elliptic evaluation and local elliptic operator)**.** *Let* τ_h *be a grid and let* $\lambda \in Q(\tau_h)$ *and* $\kappa \in [Q(\tau_h)]^{d\times d}$ *be localizable with respect to* τ_h*. We define the* local elliptic evaluation $\Upsilon_{\text{ell.}}^t : Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \to \big[\hat{\mathbb{P}}(t) \times \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)\big]$ *and the* local elliptic operator $B_h^t : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)'$ *by*

$$\Upsilon_{\text{ell.}}^t[\lambda, \kappa]\big(\psi^t, \varphi^t\big) := \big(\lambda^t \kappa^t \nabla_t \psi^t\big) \cdot \nabla_t \varphi^t \qquad \text{and} \qquad B_h^t := \Sigma_h^t\big(\Upsilon_{\text{ell.}}^t[\lambda, \kappa]\big),$$

respectively, where Σ_h^t *denotes the local volume operator from Definition 3.1.6. Note that* $\Upsilon_{\text{ell.}}^t[\lambda, \kappa]$ *is a local binary volume evaluation in the sense of Definition 3.1.6.*

Given the above definition of an elliptic operator, some remarks are in order.

Remark 3.1.8 (On integrals vs. integrands or operators vs. evaluations)**.** *The explicit distinction between the operator integrating over a certain domain and the evaluation modeling the integrand may seem over-engineered at first and may seem to come at a price of overly complex notation. However, it gives us an increased flexibility and may even simplify the notation, since we can focus our efforts on specifying meaningful local evaluations. This shall become apparent in the next paragraphs, where we require exactly the same evaluation for a non-conforming approximation and for error estimation. In addition, the concept of local evaluations allows us to formalize many quantities arising in discretization schemes. In the context of Finite Volume approximations, for instance, we usually do not carry out a quadrature but compute numerical fluxes on faces (which are appropriately modeled by local evaluations).*

We continue with the approximation of the integral in (3.1.9) for the linear functional l_f.

[4]As noted earlier, we restrict ourselves to scalar functions in order to simplify the notation, all concepts hold for vector- and matrix-valued local functions as well.

Definition 3.1.9 (Local unary volume evaluation and local volume integral functional)**.** *Let τ_h be a grid and let $\Upsilon_1^t : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)$ denote a* local unary volume evaluation*. We define the* local volume integral functional $\Sigma_h^t\big(\Upsilon_1^t\big) : \hat{\mathbb{P}}(t) \to \mathbb{R}$ *by*

$$\Sigma_h^t\big(\Upsilon_1^t\big)(\psi^t) := \sum_{n=0}^{N(\Upsilon_1^t)-1} w_n^t \Lambda_t(x_n^t) \Upsilon_1^t\big(\psi^t\big)(x_n^t),$$

where w_n^t and x_n^t, for all $0 \leq n < N(\Upsilon_1^t) \in \mathbb{N}$, denote the quadrature weights and points, respectively, in order to integrate $\Lambda_t \Upsilon_1^t\big(\psi^t\big)$ exactly.

Note that we use the same notation for the local volume operator and functional, Σ_h^t, to indicate their similarity (since the sole purpose of either is to approximate the integral over $\hat{t}$). It will be clear from the context to which of the two we are referring to (for instance by the number of arguments).

In the same spirit as above, we obtain the local L^2 functional $l_{h,f}^t$ to approximate the integral in (3.1.9) by combining the abstract local volume integral functional Σ_h^t with the appropriate local evaluation.

Example 3.1.10 (Local product evaluation and local L^2-volume functional)**.** *Let τ_h be a grid and let $f \in Q(\tau_h)$ be localizable with respect to τ_h. We define the* local product evaluation $\Upsilon_{\text{prod.}}^t : Q(\tau_h) \to \big[\hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)\big]$ *and the* local L^2-volume functional $l_{h,f}^t :$ $\hat{\mathbb{P}}(t) \to \mathbb{R}$ *by*

$$\Upsilon_{\text{prod.}}^t[f]\big(\psi^t\big) := f^t \psi^t \qquad \text{and} \qquad l_{h,f}^t := \Sigma_h^t\big(\Upsilon_{\text{prod.}}^t[f]\big),$$

respectively, where Σ_h^t denotes the local volume integral functional from Definition 3.1.9. Note that $\Upsilon_{\text{prod.}}^t[f]$ is a local unary volume evaluation in the sense of Definition 3.1.9.

The approximation of the original PDE is now nearly complete: we are still missing the approximation of the Neumann L^2 functional l_{g_N} from (3.1.2). We start again by localizing the boundary integral with respect to τ_h, where we use the fact that every boundary face is uniquely associated with an element of the grid, $e = t \cap \Gamma_N$:

$$l_{g_N}(q) = \sum_{e \in \overline{\mathcal{F}}_{h,N}} \int_e g_N \, q \, \mathrm{d}x = \sum_{t \in \tau_h} \sum_{e \in \overline{\mathcal{F}}_{h,N}^t} \int_e g_N \, q \, \mathrm{d}x. \tag{3.1.10}$$

The latter form is more suitable in our context, since we approximate the linear functional in terms of a local discrete functional acting on local functions, which in turn are associated with elements of the grid, not faces.

We proceed with the approximation of the above face integral in the same manner as we did with the volume integrals: by transformation onto a reference face and numerical quadrature. Similarly to the elements of the grid, each face $e \in \mathcal{F}_h$ is given by a reference face $\hat{e}$ and a bijective map $\Phi^e : \hat{e} \to e$. In addition we require another map in order to formulate a quadrature on $\hat{e}$ which allows us to evaluate a local function on $\hat{t}$: for each element $t \in \tau_h$ of the grid that shares the face $e \subset \partial t$, there exists a bijective embedding of the respective reference face into the respective reference element, which we denote by $\hat{\Phi}^t_e : \hat{e} \to \hat{t}$. This allows us to transform the integral in (3.1.10) to the respective reference face and to formulate it in terms of local functions,

$$\int_e g_N \, q \, \mathrm{d}x = \int_{\hat{e}} \Lambda_e \left(\tilde{g}^t_N \circ \hat{\Phi}^t_e\right)\left(q^t \circ \hat{\Phi}^t_e\right) \mathrm{d}x = \int_{\hat{e}} \Lambda_e \, \tilde{g}^t_{N\,e} \, q^t_e, \tag{3.1.11}$$

where $\Lambda_e^2 := |\nabla \Phi^{e\perp} \cdot \nabla \Phi^e|$, $t \in \tau_h$ such that $e = t \cap \Gamma_N$ and where $\tilde{g}_N \in Q(\tau_h)$ is an extension of g_N, such that $\tilde{g}_N|_{\Gamma_N} = g_N$ in the sense of traces (which is only evaluated on Γ_N).

Remark 3.1.11 (Local functions and face evaluations)**.** *In order to preserve the structure of integrands associated with face integrals, we denote a local function $q^t \in \hat{\mathbb{P}}(t)$, for $t \in \tau_h$, which can be evaluated on the reference face $\hat{e}$ of $e \in \mathcal{F}_h$, such that $e \subset \partial t$, by $q^t_e := q^t \circ \hat{\Phi}^t_e \in \hat{\mathbb{P}}(e)$, with $\hat{\mathbb{P}}(e) := \{\varphi \circ \Phi^e \,|\, \varphi \in \mathbb{P}(e)\}$. We apply the same notation for the local derivative as in Definition 3.1.17, e.g., we denote the local gradient of a local function which can be evaluated on a reference face by $\nabla_t q^t_e := \left(\nabla_t q^t\right) \circ \hat{\Phi}^t_e$.*

Similar to volume integrals, we approximate the above integral in terms of a local evaluation and a local functional.

Definition 3.1.12 (Local unary face evaluation and local boundary integral functional)**.** *Let τ_h be a grid and let $\Upsilon^e_1[\Gamma_N] : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(e)$ denote a* local unary face evaluation*, for a boundary face $e \in \overline{\mathcal{F}}_{h,N}$, with $t \in \tau_h$, such that $e = t \cap \Gamma_N$. We define the* local boundary integral functional $\Sigma^e_h\big(\Upsilon^e_1[\Gamma_N]\big) : \hat{\mathbb{P}}(t) \to \mathbb{R}$ *by*

$$\Sigma^e_h\big(\Upsilon^e_1[\Gamma_N]\big)(\psi^t) := \sum_{n=0}^{N(\Upsilon^e_1)-1} w^e_n \Lambda_e(x^e_n) \Upsilon^e_1[\Gamma_N]\big(\psi^t\big)(x^e_n),$$

where w^e_n and x^e_n, for all $0 \leq n < N(\Upsilon^e_1) \in \mathbb{N}$, denote the quadrature weights and points, respectively, in order to integrate $\Lambda_e \Upsilon^e_1[\Gamma_N]\big(\psi^t\big)$ exactly.

We finally obtain the local L^2-boundary functional l^e_{h,g_N} to approximate the integral in (3.1.11) by combining the abstract local boundary integral functional Σ^e_h with the appropriate local evaluation.

Example 3.1.13 (Local product evaluation and local L^2-boundary functional)**.** *Let τ_h be a grid and let $\tilde{g}_N \in Q(\tau_h)$ be localizable with respect to τ_h, such that $\tilde{g}_N|_{\Gamma_N} = g_N$ in the sense of traces. We define the* local product evaluation $\Upsilon^e_{\text{prod.}}[\Gamma_N, \cdot] : Q(\tau_h) \to \big[\hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(e)\big]$ *for a boundary face $e \in \overline{\mathcal{F}}_{h,N}$, with $t \in \tau_h$ such that $e = t \cap \Gamma_N$, and the* local L^2-boundary functional $l^e_{h,g_N} : \hat{\mathbb{P}}(t) \to \mathbb{R}$ *by*

$$\Upsilon^e_{\text{prod.}}[\Gamma_N, \tilde{g}_N]\big(\psi^t\big) := \tilde{g}_{N\,e}^{\,t} \, \psi^t_e \qquad \text{and} \qquad l^e_{h,g_N} := \Sigma^e_h\big(\Upsilon^e_{\text{prod.}}[\Gamma_N, g_N]\big),$$

respectively, where Σ^e_h denotes the local boundary integral functional from Definition 3.1.12. Note that $\Upsilon^e_{\text{prod.}}[\Gamma_N, g_N]$ is a local unary face evaluation in the sense of Definition 3.1.12.

This completes the approximation of the local operator and functionals which constitute their respective global counterparts.

Definition 3.1.14 (Discrete elliptic operator, discrete L^2-functionals)**.** *With B_h^t, $l_{h,f}$ and l_{h,g_N} as in Examples 3.1.7, 3.1.10 and 3.1.13 we define the* discrete elliptic operator *$B_h : Q(\tau_h) \to Q(\tau_h)'$ and the* discrete L^2-functionals *$l_{h,f} : Q(\tau_h) \to \mathbb{R}$ and $l_{h,g_N} : Q(\tau_h) \to \mathbb{R}$ for any $p, q \in Q(\tau_h)$ by*

$$\begin{aligned}
B_h[p](q) &:= \sum_{t\in\tau_h} B_h^t[p|_t \circ \Phi^t](q|_t \circ \Phi^t) = \sum_{t\in\tau_h} B_h^t[p^t](q^t),\\
l_{h,f}(q) &:= \sum_{t\in\tau_h} l_{h,f}^t(q|_t \circ \Phi^t) = \sum_{t\in\tau_h} l_{h,f}^t(q^t) \qquad \textit{and}\\
l_{h,g_N}(q) &:= \sum_{t\in\tau_h} \sum_{e\in\overline{\mathcal{F}}_{h,N}^t} l_{h,g_N}^e(q|_t \circ \Phi^t) = \sum_{t\in\tau_h} \sum_{e\in\overline{\mathcal{F}}_{h,N}^t} l_{h,g_N}^e(q^t),
\end{aligned}$$

respectively.

The discrete variant of (3.1.3) then reads: find $p_{h,0} \in S_h^1(\tau_h) \cap H_{\Gamma_D}^1(\Omega)$, such that

$$B_h[p_{h,0}](q_h) = l_{h,f}(q_h) + l_{h,g_N}(q_h) - B_h[g_{h,D}](q_h) \quad \text{for all } q_h \in S_h^1(\tau_h) \cap H_{\Gamma_D}^1(\Omega), \tag{3.1.12}$$

where we make use of the fact that $S_h^k(\tau_h) \subset Q(\tau_h)$.

The approximation of the original PDE is now formally complete. However, there remain some important technical details on how to compute all quantities arising in (3.1.12), namely how to form (and solve) the algebraic problem and how to choose $g_{h,D}$. We first consider the algebraic problem associated with (3.1.12).

Given a basis $\phi := \{\varphi_0, \dots, \varphi_{I-1}\}$ of $S_h^k(\tau_h)$, for some $I \in \mathbb{N}$, we represent the unknown solution in terms of its <u>D</u>egrees <u>o</u>f <u>F</u>reedom (DoFs) $p_i \in \mathbb{R}$: $p_{h,0} = \sum_{i=0}^{I-1} p_i\varphi_i$. Using this basis representation, (3.1.12) reads: find $p_j \in \mathbb{R}$, for $0 \le j < I$, such that

$$\sum_{j=0}^{I-1} p_j B_h[\varphi_j](\varphi_i) = l_{h,f}(\varphi_i) + l_{h,g_N}(\varphi_i) - B_h[g_{h,D}](\varphi_i) \quad \text{for all } 0 \le i < I, \tag{3.1.13}$$

under the constraint that $\varphi_i|_{\Gamma_D} = 0$ for all $0 \le i < I$.[5] We have thus arrived at a system of I linear equations to determine the I unknowns p_i. Given (3.1.13), we define the matrix representation of B_h and vector representations of $l_{h,f}$ and l_{h,g_N} with respect to the basis ϕ, namely $\underline{B_h} \in \mathbb{R}^{I\times I}$, $\underline{l_{h,f}} \in \mathbb{R}^I$ and $\underline{l_{h,g_N}} \in \mathbb{R}^I$, by

$$\left(\underline{B_h}\right)_{i,j} := B_h[\varphi_j](\varphi_i), \quad \left(\underline{l_{h,f}}\right)_i := l_{h,f}(\varphi_i) \quad \text{and} \quad \left(\underline{l_{h,g_N}}\right)_i := l_{h,g_N}(\varphi_i), \tag{3.1.14}$$

respectively, for all $0 \le i, j < I$. A naive way of computing the above matrices and vectors is given in Algorithm 3.1.15. This algorithm, however, is computationally not

[5]We will see further below how to enforce this constraint.

Algorithm 3.1.15 Naive assembly of global matrices and vectors.

```
Input: a global basis $\phi$ of $S_h^k(\tau_h)$
Output: $\underline{B_h}$, $\underline{l_{h,f}}$ and $\underline{l_{h,g_N}}$
  for all $0 \leq i < I$ do
      $\left(\underline{l_{h,f}}\right)_i \leftarrow l_{h,f}(\varphi_i)$
      $\left(\underline{l_{h,g_N}}\right)_i \leftarrow l_{h,g_N}(\varphi_i)$
      for all $0 \leq j < I$ do
          $\left(\underline{B_h}\right)_{i,j} \leftarrow B_h[\varphi_j](\varphi_i)$
      end for
  end for
```

efficient, since random access to containers is usually much cheaper than iterating over the elements of a grid. In addition, Algorithm 3.1.15 is inspired by the interpretation of ϕ as a *global* basis of $S_h^k(\tau_h)$, which is not very useful in the context of Finite Element methods. In the spirit of Finite Element methods (see [Cia1978, FEM3 on page 41]) we presume each basis function to have a local support of only very few elements of the grid. It would thus be a waste of resources to repeatedly iterate over all elements of the grid for each entry $(\underline{l_{h,f}})_i$, $(\underline{l_{h,g_N}})_i$ and $(\underline{B_h})_{i,j}$ to compute the underlying integrals. It shall be much more useful to consider the localization of the global basis ϕ with respect to the grid τ_h.

Definition 3.1.16 (Discrete function space)**.** *Let τ_h be a grid of Ω and $\mathbb{V}_h(\tau_h)$ be a vector-space of real-valued functions $v : \Omega \to \mathbb{R}^{r \times c}$, for $r, c \in \mathbb{N}$.*[6] *We call $\mathbb{V}_h(\tau_h)$ a* discrete function space *iff*

- *its basis $\phi = \{\varphi_0, \dots, \varphi_{I-1}\}$ is a finite set, for $I \in \mathbb{N}$; and*
- *for all $t \in \tau_h$, there exists a* local basis $\phi^t_{k(t)} = \{\varphi^t_0, \dots, \varphi^t_{I(t)-1}\}$ *of polynomial degree at most $k(t) \in \mathbb{N}$, where the size of the local basis, $I(t) \in \mathbb{N}$, only depends on the polynomial degree $k(t)$, the dimension d and the element t; and*
- *for all $t \in \tau_h$, there exists a* DoF map $\iota(t, \cdot) : \{0, \dots, I(t) - 1\} \hookrightarrow \{0, \dots, I - 1\}$, *such that for each local basis function $\varphi^t_i \in \phi^t_{k(t)}$ there exists exactly one global basis function $\varphi_{\iota(t,i)} \in \phi$, such that $\varphi^t_i = \varphi_{\iota(t,i)}\big|_t \circ \Phi^t$.*

Each local basis function is thus given as the localization of a global basis function and the relation of the two is determined by the DoF map ι. As a consequence, each element of a discrete function space is localizable.

Definition 3.1.17 (Discrete function, DoF vector)**.** *Let $\mathbb{V}_h(\tau_h)$ be a discrete function space. We call an element $v_h \in \mathbb{V}_h(\tau_h)$ a* discrete function*, represented by its* DoF vector $\underline{v_h} \in \mathbb{R}^I$*, such that $v_h(x) = \sum_{i=0}^{I-1} (\underline{v_h})_i \, \varphi_i(x)$ for all $x \in \Omega$. Each discrete function can be localized (in the sense of Definition 3.1.2) with respect to τ_h by means of a* local discrete function $v^t_h \in \hat{\mathbb{P}}(t)$*, represented by its* local DoF vector $\underline{v_h}^t \in \mathbb{R}^{I(t)}$*, such that $v^t_h(\hat{x}) = \sum_{i=0}^{I(t)-1} (\underline{v_h})_{\iota(t,i)} \, \varphi^t_i(\hat{x})$ for all $\hat{x} \in \hat{t}$.*

[6]We only consider real valued functions. The concepts introduced work for other fields as well, though.

Compared to Algorithm 3.1.15 it is computationally much more efficient to make use of the fact that $S_h^k(\tau_h)$ is a discrete function space and to consider all local basis functions on one grid element at a time and to add their respective contribution to the corresponding global matrix or vector entry (see Algorithm 3.1.18, where $\overline{\mathcal{F}}_{h,N}^t \subset \overline{\mathcal{F}}_h$ denotes the set of all faces of t that lie on the Neumann boundary Γ_N).

Algorithm 3.1.18 Local assembly of global matrices and vectors (CG FEM).

```
Input: a discrete function space $S_h^k(\tau_h)$
Output: $\underline{B_h}$, $\underline{l_{h,f}}$ and $\underline{l_{h,g_N}}$
  initialize $\underline{B_h}$, $\underline{l_{h,f}}$ and $\underline{l_{h,g_N}}$ with zero entries
  for all $t \in \tau_h$ do                                        ▷ compute volume integrals
      for all $0 \leq i < I(t)$ do
          $\big(\underline{l_{h,f}}\big)_{\iota(t,i)} \leftarrow \big(\underline{l_{h,f}}\big)_{\iota(t,i)} + l_{h,f}^t(\varphi_i^t)$
          for all $0 \leq j < I(t)$ do
              $\big(\underline{B_h}\big)_{\iota(t,i),\iota(t,j)} \leftarrow \big(\underline{B_h}\big)_{\iota(t,i),\iota(t,j)} + B_h^t[\varphi_j^t](\varphi_i^t)$
          end for
      end for
      for all $e \in \overline{\mathcal{F}}_{h,N}^t$ do                          ▷ compute Neumann face integrals
          for all $0 \leq i < I(t)$ do
              $\big(\underline{l_{h,g_N}}\big)_{\iota(t,i)} \leftarrow \big(\underline{l_{h,g_N}}\big)_{\iota(t,i)} + l_{h,g_N}^e(\varphi_i^t)$
          end for
      end for
  end for
```

Last, we specify how to choose $g_{h,D} \in H^1(\Omega)$ and how to enforce the Dirichlet constraints, in order to form the algebraic problem. For both, we make use of the fact that $S_h^k(\tau_h)$ is a Lagrangian type discrete function space.

Definition 3.1.19 (Continuous Lagrange discrete function space)**.** *Let $S_h^k(\tau_h)$ be a discrete function space fulfilling* (3.1.4) *with $k \in \mathbb{N}$, $k \geq 1$. We call $S_h^k(\tau_h)$ a* continuous Lagrange discrete function space *of order k, iff, for all $t \in \tau_h$ there exists a finite set of* Lagrange points *$\{\hat{\nu}_0^t, \dots, \hat{\nu}_{I(t)-1}^t\} \subset \hat{t}$, such that $\varphi_i^t(\hat{\nu}_j^t) = \delta_{ij}$ for all $0 \leq i, j, < I(t)$, where δ_{ij} denotes the Kronecker symbol.*

Thus, each element of a Lagrange discrete function space is uniquely defined by specifying its values on all Lagrange points. The Lagrange points of the piecewise linear continuous Lagrange discrete function space $S_h^1(\tau_h)$ are given by the vertices of each element $t \in \tau_h$, for instance.

Given the definition of a Lagrange discrete function space, the choice of $g_{h,D}$ as a projection of the Dirichlet boundary values g_D is straightforward and we choose $g_{h,D} := \Pi_{S_h^k(\tau_h)}^{\Gamma_D}[g_D]$ in (3.1.3).

Definition 3.1.20 (Dirichlet projection)**.** *Let $S_h^k(\tau_h)$ be a continuous Lagrange discrete function space. We define the* Dirichlet projection *$\Pi_{S_h^k(\tau_h)}^{\Gamma_D} : L^2(\Gamma_D) \to S_h^k(\tau_h)$, $g \mapsto \Pi_{S_h^k(\tau_h)}^{\Gamma_D}[g]$, by specifying the values of the local discrete function of its image, $\Pi_{S_h^k(\tau_h)}^{\Gamma_D}[g]^t$,* *at each Lagrange point $\hat{\nu}$ on each element $t \in \tau_h$:*

$$\Pi^{\Gamma_D}_{S^k_h(\tau_h)}[g]^t(\hat{\nu}) := \begin{cases} (g \circ \Phi^t)(\hat{\nu}), & \textit{if } \Phi^t(\hat{\nu}) \in \Gamma_D, \\ 0, & \textit{else.} \end{cases}$$

Finally, we enforce the constraint $p_{h,0} \in S^k_h(\tau_h) \cap H^1_{\Gamma_D}(\Omega)$, required by (3.1.12), on the algebraic level, again making use of the structure of $S^k_h(\tau_h)$.

Definition 3.1.21 (Dirichlet constraints)**.** *Let $S^k_h(\tau_h)$ be a Lagrange discrete function space, let $\underline{B} \in \mathbb{R}^{I \times I}$ and $\underline{l} \in \mathbb{R}^I$ and let $p_h \in S^k_h(\tau_h)$ be a discrete function with DoF vector $\underline{p} \in \mathbb{R}^I$. We call the problem: find $\underline{p} \in \mathbb{R}^I$, such that*

$$\underline{B}|_D \, \underline{p} = \underline{l}|_D,$$

Dirichlet constrained*, denoted by $\cdot|_D$, iff for all elements $t \in \tau_h$*

$$\big(\underline{B}|_D\big)_{\iota(t,i),\iota(t,j)} = \delta_{ij} \qquad \textit{and} \qquad \big(\underline{l}|_D\big)_{\iota(t,i)} = 0$$

for all $\{0 \leq i < I(t) \,|\, \hat{\nu}^t_i \in \Gamma_D\}$ and all $0 \leq j < I(t)$.

The algebraic problem that corresponds to (3.1.12) then reads: find $\underline{p_{h,0}} \in \mathbb{R}^I$, such that

$$\underline{B_h}\big|_D \, \underline{p_{h,0}} = \big(\underline{l_{h,f}} + \underline{l_{h,g_N}} - \underline{B_h}\, \underline{g_{h,D}}\big)\Big|_D, \tag{3.1.15}$$

where $\underline{p_{h,0}}$ is the DoF vector of the constrained solution $p_{h,0} \in S^k_h(\tau_h)$ of (3.1.12). The DoF vector of the unconstrained solution $p_h \in S^k_h(\Omega)$ can be obtained by $\underline{p_h} := \underline{p_{h,0}} + \underline{g_{h,D}}$.

This finalizes the continuous Galerkin Finite Element approximation of (3.1.1).

Next we consider a nonconforming approximation by discontinuous Galerkin methods. In contrast to a conforming approximation, discontinuous Galerkin methods enforce the continuity of the solution along the faces of the grid only implicitly. While this comes at the expense of a considerably higher number of Degrees of Freedom, these methods have other desirable properties (such as local conservation properties or less communication effort in parallel environments, at least for higher orders).

A nonconforming approximation: discontinuous Galerkin FEM. There exist a large variety of discontinuous Galerkin based Finite Element approximations for elliptic problems. We consider the *symmetric weighted Interior Penalty discontinuous Galerkin* (SWIPDG) method, introduced in [ESZ2009], which covers many existing methods (compare the discussion in Section 2.1).

Similar to continuous Galerkin Finite Element approximations, a discontinuous Galerkin approximation consists of two parts: the definition of an approximation space as well as the choice of appropriate discrete operators and functionals. Therefore, we directly consider (3.1.1) and choose a space of globally discontinuous and piecewise polynomial functions,

$$Q_h^k(\tau_h) := \{q \in L^2(\Omega) \mid q|_t \in \mathbb{P}_k(t), \ \ \forall t \in \tau_h\} \subset Q(\tau_h), \tag{3.1.16}$$

for $k \in \mathbb{N}$, to approximate $H^1(\Omega)$, with $Q_h^k(\tau_h) \not\subset H^1(\Omega)$.[7]

Definition 3.1.22 (Discontinuous Galerkin discrete function space)**.** *Let τ_h be a grid and $k \in \mathbb{N}$. We call $Q_h^k(\tau_h)$ a* discontinuous Galerkin discrete function space *iff it is a discrete function space in the sense of Definition 3.1.16 and fulfills* (3.1.16)*.*

Particular discontinuous Galerkin discrete function spaces are given by specifying the local shape functions which span $\mathbb{P}_k(t)$, possible choices include Legendre and Lagrange polynomials. We consider the latter and thus denote $Q_h^k(\tau_h)$ a *discontinuous Lagrange discrete function space* from here on.

Since $Q_h^k(\tau_h) \not\subset H^1(\Omega)$ we cannot obtain a discretization of (3.1.1) by a mere Galerkin projection onto $Q_h^k(\tau_h)$. Functions in $Q_h(\tau_h)$ are discontinuous across and thus double valued on the faces of the grid and we need to add additional continuity and penalty terms to the discrete operator and functionals. We thus approximate (3.1.1) by the following problem: for $k \geq 1$ find $p_h \in Q_h^k(\tau_h)$, such that

$$b_h(q_h, p_h) = l_f(q_h) + l_{g_N}(q_h) + l_{h,D}(q_h) \qquad \text{for all } q_h \in Q_h^k(\tau_h), \tag{3.1.17}$$

where l_f and l_{g_N} are given by (3.1.2). In order to specify the bilinear form b_h and the linear functional $l_{h,D}$ we recall from Section 2.1 of the previous chapter the *broken Sobolev space*

$$H^k(\tau_h) = \{q \in L^2(\Omega) \mid q|_t \in H^k(t) \ \ \forall t \in \tau_h\},$$

the *broken gradient* operator $\nabla_h : H^k(\tau_h) \to [H^{k-1}(\tau_h)]^d$,

$$(\nabla_h q)|_t = \nabla(q|_t) \qquad \text{for all } t \in \tau_h,$$

and the *jump* and *weighted average* of a two-valued function $q \in H^k(\tau_h)$ on an inner face $e \in \mathring{\mathcal{F}}_h$, with $e = t^- \cap t^+$,

$$[\![q]\!]_e = q|_{t^-} - q|_{t^+} \qquad \text{and} \qquad \{\!\{q\}\!\}_e = \omega_e^- q|_{t^-} + \omega_e^+ q|_{t^+}$$

and on a boundary face $e \in \overline{\mathcal{F}}_h$, $[\![q]\!]_e = q$ and $\{\!\{q\}\!\}_e = q$, respectively. The locally adaptive weights $\omega_e^\pm > 0$ are given by $\omega_e^- = \delta_e^+(\delta_e^+ + \delta_e^-)^{-1}$ and $\omega_e^+ = \delta_e^-(\delta_e^+ + \delta_e^-)^{-1}$, respectively, such that $\omega_e^- + \omega_e^+ = 1$, with $\delta_e^\pm = n_e \cdot \kappa|_{t^\pm} \cdot n_e$. We recall that each inner face $e \in \mathring{\mathcal{F}}_h$ is uniquely oriented with a unit normal $n_e \in \mathbb{R}^d$ pointing from t^- to t^+ (corresponding to an ordering of the elements of the grid which allows for a meaningful comparison in the sense of $t^- < t^+$).

[7]For simplicity we only consider the same polynomial degree k on all elements, all concepts hold for locally varying polynomial degrees as well.

With this notation we define the SWIPDG bilinear form $b_h : H^1(\tau_h) \times H^1(\tau_h) \to \mathbb{R}$ and the linear functional $l_{h,D} : H^1(\tau_h) \to \mathbb{R}$ in (3.1.17) by

$$\begin{aligned} b_h(q,p) := &\sum_{t\in\tau_h} \int_t (\lambda\kappa\nabla_h q)\cdot\nabla_h p \,\mathrm{d}x \qquad (3.1.18)\\ &+ \underbrace{\sum_{e\in\overset{\circ}{\mathcal{F}}_h\cup\overline{\mathcal{F}}_{h,D}} \int_e - \{\{(\lambda\kappa\nabla_h p)\cdot n_e\}\}_e [\![q]\!]_e - \{\{(\lambda\kappa\nabla_h q)\cdot n_e\}\}_e [\![p]\!]_e + \sigma_e [\![q]\!]_e [\![p]\!]_e \,\mathrm{d}x}_{=:(i)} \end{aligned}$$

and

$$l_{h,D}(q) := \sum_{e\in\overline{\mathcal{F}}_{h,D}} \int_e \big(\sigma_e q - (\lambda\kappa\nabla_h q)\cdot n_e\big) g_D \,\mathrm{d}x, \qquad (3.1.19)$$

respectively, with the locally adaptive penalty function given by $\sigma_e := \sigma h_e^{-1} \{\{\lambda\}\}_e \sigma_\kappa$ (with a locally adaptive penalty parameter $\sigma \geq 1$ as specified in [ER2007]) and the locally adaptive weight given by $\sigma_\kappa = \delta_e^+\delta_e^-(\delta_e^+ + \delta_e^-)^{-1}$ on an inner face and by $\sigma_\kappa = \delta_e^-$ on a boundary face.

In order to fully discretize Problem (3.1.17) we need to rephrase it in terms of local operators and local functionals acting on local functions. We have already done so for the linear functionals l_f and l_{g_N} in (3.1.17) and the part of b_h in (3.1.18) that is associated with the volume integral. We are left to specify how to approximate $l_{h,D}$ and the parts of b_h in (3.1.18) that are associated with the face integrals. We start with the former and recall from Definition 3.1.12 the local boundary integral functional Σ_h^e which approximates a face integral by a numerical quadrature, given an appropriate local unary face evaluation (which we obtain by transforming the face integral in (3.1.19) into an integral on the reference face).

Example 3.1.23 (Local SWIPDG boundary evaluation and local SWIPDG boundary functional)**.** *Let τ_h be a grid and let $\lambda \in Q(\tau_h)$, $\kappa \in [Q(\tau_h)]^{d\times d}$ and $\tilde{g}_D \in Q(\tau_h)$ be localizable with respect to τ_h, where $\tilde{g}_D$ is an extension of g_D, such that $\tilde{g}_D|_{\Gamma_D} = g_D$ in the sense of traces. For a boundary face $e \in \overline{\mathcal{F}}_{h,D}$, with $t \in \tau_h$, such that $e = t\cap\Gamma_D$, we define the* local SWIPDG boundary evaluation $\overline{\Upsilon}^e_{\mathrm{SWIP}}[\Gamma_D, \dots] : Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \times Q(\tau_h) \to \big[\hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(e)\big]$*, by*

$$\overline{\Upsilon}^e_{\mathrm{SWIP}}[\Gamma_D, \lambda, \kappa, \tilde{g}_D]\big(\psi^t\big) := \big(\sigma_e \psi_e^t - (\lambda_e^t\, \kappa_e^t\, \nabla_t \psi_e^t)\cdot n_e\big)\tilde{g}_{D}{}_e^t$$

and the local SWIPDG boundary functional $\overline{l}^e_{h,g_D} : \hat{\mathbb{P}}(t) \to \mathbb{R}$ *by*

$$\overline{l}^e_{h,g_D} := \Sigma_h^e\big(\,\overline{\Upsilon}^e_{\mathrm{SWIP}}[\Gamma_D, \lambda, \kappa, \tilde{g}_D]\big),$$

where Σ_h^e denotes the local boundary integral functional from Definition 3.1.12. Note that $\overline{\Upsilon}^e_{\mathrm{SWIP}}[\Gamma_D, \lambda, \kappa, \tilde{g}_D]$ is a local unary face evaluation in the sense of Definition 3.1.12.[8]

[8]Note that $\psi_e^t \in \hat{\mathbb{P}}(e)$ denotes a local function which can be evaluated on the reference face, compare Remark 3.1.11.

We continue with the approximation of the face integral in (3.1.18). In order to reformulate the bilinear form b_h in terms of local functions associated with the elements of the grid we examine the face integral and split it up, depending on the integration area and the local functions involved. We start by inserting the definition of $\{\!\{\cdot\}\!\}_e$ and $[\![\cdot]\!]_e$ in (3.1.18):

$$\begin{aligned}(i) = \sum_{e\in\overline{\mathcal{F}}_{h,D}} &\underbrace{\int_e -(\lambda\kappa\nabla p)\cdot n_e q - (\lambda\kappa\nabla q)\cdot n_e p + \sigma_e\, q\, p \,\mathrm{d}x}_{=:(ii)} \qquad (3.1.20)\\
+ \sum_{e\in\overset{\circ}{\mathcal{F}}_h} &\int_e -\big((\omega_e^-\,\lambda|_{t^-}\,\kappa|_{t^-}\nabla p|_{t^-} + \omega_e^+\,\lambda|_{t^+}\,\kappa|_{t^+}\nabla p|_{t^+})\cdot n_e\big)\big(q|_{t^-} - q|_{t^+}\big)\\
&\quad - \big((\omega_e^-\,\lambda|_{t^-}\,\kappa|_{t^-}\nabla q|_{t^-} + \omega_e^+\,\lambda|_{t^+}\,\kappa|_{t^+}\nabla q|_{t^+})\cdot n_e\big)\big(p|_{t^-} - p|_{t^+}\big)\\
&\underbrace{\quad +\sigma_e\big(q|_{t^-} - q|_{t^+}\big)\big(p|_{t^-} - p|_{t^+}\big)\,\mathrm{d}x.}_{=:(iii)}\end{aligned}$$

Next, we transform the integrals to the reference face and group the terms associated with an inner face by the adjacent grid elements, to which the ansatz and test functions p and q are restricted to. We start with the integral associated with boundary faces and obtain from (3.1.20), with $t \in \tau_h$, such that $e = t \cap \Gamma_D$:

$$(ii) = \int_{\hat{e}} \Lambda_e\big(\underbrace{-(\lambda_e^t\,\kappa_e^t\nabla_t p_e^t)\cdot n_e\, q_e^t - (\lambda_e^t\,\kappa_e^t\nabla_t q_e^t)\cdot n_e\, p_e^t + \sigma_e\, q_e^t\, p_e^t}_{=:\overline{\Upsilon}^e_{\mathrm{SWIP}}[\Gamma_D,\lambda,\kappa](q^t,p^t)}\big)\,\mathrm{d}x. \qquad (3.1.21)$$

We continue with the integral associated with inner faces and obtain from (3.1.20), with $t^\pm \in \tau_h$, such that $e = t^- \cap t^+$:

$$\begin{aligned}(iii) = \int_{\hat{e}} \Lambda_e\Big(&\underbrace{-\omega_e^-\big(\lambda_e^{t^-}\kappa_e^{t^-}\nabla_{t^-}p_e^{t^-}\big)\cdot n_e\, q_e^{t^-} - \omega_e^-\big(\lambda_e^{t^-}\kappa_e^{t^-}\nabla_{t^-}q_e^{t^-}\big)\cdot n_e\, p_e^{t^-} + \sigma_e\, q_e^{t^-}\, p_e^{t^-}}_{=:\overset{\circ}{\Upsilon}{}^{e--}_{\mathrm{SWIP}}[\lambda,\kappa](q^{t^-},p^{t^-})}\\
&\qquad (3.1.22)\\
&\underbrace{-\omega_e^+\big(\lambda_e^{t^+}\kappa_e^{t^+}\nabla_{t^+}p_e^{t^+}\big)\cdot n_e\, q_e^{t^-} + \omega_e^-\big(\lambda_e^{t^-}\kappa_e^{t^-}\nabla_{t^-}q_e^{t^-}\big)\cdot n_e\, p_e^{t^+} + \sigma_e\, q_e^{t^-}\, p_e^{t^+}}_{=:\overset{\circ}{\Upsilon}{}^{e-+}_{\mathrm{SWIP}}[\lambda,\kappa](q^{t^-},p^{t^+})}\\
&\underbrace{+\omega_e^-\big(\lambda_e^{t^-}\kappa_e^{t^-}\nabla_{t^-}p_e^{t^-}\big)\cdot n_e\, q_e^{t^+} - \omega_e^+\big(\lambda_e^{t^+}\kappa_e^{t^+}\nabla_{t^+}q_e^{t^+}\big)\cdot n_e\, p_e^{t^-} + \sigma_e\, q_e^{t^+}\, p_e^{t^-}}_{=:\overset{\circ}{\Upsilon}{}^{e+-}_{\mathrm{SWIP}}[\lambda,\kappa](q^{t^+},p^{t^-})}\\
&\underbrace{+\omega_e^+\big(\lambda_e^{t^+}\kappa_e^{t^+}\nabla_{t^+}p_e^{t^+}\big)\cdot n_e\, q_e^{t^+} + \omega_e^+\big(\lambda_e^{t^+}\kappa_e^{t^+}\nabla_{t^+}q_e^{t^+}\big)\cdot n_e\, p_e^{t^-} + \sigma_e\, q_e^{t^+}\, p_e^{t^+}}_{=:\overset{\circ}{\Upsilon}{}^{e++}_{\mathrm{SWIP}}[\lambda,\kappa](q^{t^+},p^{t^+})}\Big)\,\mathrm{d}x.\end{aligned}$$

We have thus derived the local evaluations and local operators required for the approximation of the face integrals of b_h.

Definition 3.1.24 (Local binary face evaluation and local boundary integral operator)**.** *Let τ_h be a grid and let $\Upsilon_2^e : \hat{\mathbb{P}}(t) \times \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(e)$ denote a* local binary face evaluation *for a boundary face $e \in \overline{\mathcal{F}}_{h,D}$, with $t \in \tau_h$ such that $e = t \cap \Gamma_D$. We define the* local boundary integral operator $\Sigma_h^e\big(\Upsilon_2^e\big) : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)'$ *by*

$$\Sigma_h^e\big(\Upsilon_2^e\big)[\varphi^t](\psi^t) := \sum_{n=0}^{N(\Upsilon_2^e)-1} w_n^e \Lambda_e(x_n^e)\Upsilon_2^e\big(\psi^t, \varphi^t\big)(x_n^e),$$

where w_n^e and x_n^e, for all $0 \le n < N(\Upsilon_2^e) \in \mathbb{N}$, denote the quadrature weights and points, respectively, in order to integrate $\Lambda_e \Upsilon_2^e\big(\psi^t, \varphi^t\big)$ exactly.

Example 3.1.25 (Local SWIPDG boundary evaluation and local SWIPDG boundary operator)**.** *Let τ_h be a grid and let $\lambda \in Q(\tau_h)$ and $\kappa \in [Q(\tau_h)]^{d\times d}$ be localizable with respect to τ_h. For a boundary face $e \in \overline{\mathcal{F}}_{h,D}$, such that $e = t \cap \Gamma_D$ for an element $t \in \tau_h$, we define the* local SWIPDG boundary evaluation $\overline{\Upsilon}^e_{\text{SWIP}}[\Gamma_D, \cdot, \cdot] : Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \to \big[\hat{\mathbb{P}}(t) \times \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(e)\big]$ *by* (3.1.21) *and the* local SWIPDG boundary operator $\overline{B}^e_{h,D} : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)'$ *by*

$$\overline{B}^e_{h,D} := \Sigma_h^e\big(\,\overline{\Upsilon}^e_{\text{SWIP}}[\Gamma_D, \lambda, \kappa]\big),$$

where Σ_h^e denotes the local boundary integral operator from Definition 3.1.24. Note that $\overline{\Upsilon}^e_{\text{SWIP}}[\Gamma_D, \lambda, \kappa]$ is a local binary face evaluation in the sense of Definition 3.1.24.

Definition 3.1.26 (Local quaternary face evaluation and local coupling integral operator)**.** *Let τ_h be a grid. For an inner face $e \in \mathring{\mathcal{F}}_h$, with $t^\pm \in \tau_h$ such that $e = t^- \cap t^+$, let Υ_4^e denote a* local quaternary face evaluation *given by*

$$\begin{aligned} &\Upsilon_4^{e^{--}} : \hat{\mathbb{P}}(t^-) \times \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(e), && && \Upsilon_4^{e^{-+}} : \hat{\mathbb{P}}(t^-) \times \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(e),\\ &\Upsilon_4^{e^{+-}} : \hat{\mathbb{P}}(t^+) \times \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(e) && \textit{and} && \Upsilon_4^{e^{++}} : \hat{\mathbb{P}}(t^+) \times \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(e). \end{aligned}$$

We define the local coupling integral operators

$$\begin{aligned} &\mathring{\Sigma}_h^{e^{--}}\big(\Upsilon_4^{e^{--}}\big) : \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(t^-)', && && \mathring{\Sigma}_h^{e^{-+}}\big(\Upsilon_4^{e^{-+}}\big) : \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(t^+)',\\ &\mathring{\Sigma}_h^{e^{+-}}\big(\Upsilon_4^{e^{+-}}\big) : \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(t^-)' && \textit{and} && \mathring{\Sigma}_h^{e^{++}}\big(\Upsilon_4^{e^{++}}\big) : \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(t^+)' \end{aligned}$$

for all combinations of $-$ and $+$ by

$$\mathring{\Sigma}_h^{e^{\pm\pm}}\big(\Upsilon_4^{e^{\pm\pm}}\big)[\varphi^{t^\pm}](\psi^{t^\pm}) := \sum_{n=0}^{N(\Upsilon_4^{e^{\pm\pm}})-1} w_n^e \Lambda_e(x_n^e)\Upsilon_4^{e^{\pm\pm}}\big(\psi^{t^\pm}, \varphi^{t^\pm}\big)(x_n^e)$$

where w_n^e and x_n^e, for all $0 \le n < N(\Upsilon_4^{e^{\pm\pm}}) \in \mathbb{N}$, denote the quadrature weights and points, respectively, in order to integrate $\Lambda_e \Upsilon_4^{e^{\pm\pm}}\big(\psi^{t^\pm}, \varphi^{t^\pm}\big)$ exactly.

Example 3.1.27 (Local SWIPDG coupling evaluations and local SWIPDG coupling operators)**.** *Let τ_h be a grid and let $\lambda \in Q(\tau_h)$ and $\kappa \in [Q(\tau_h)]^{d\times d}$ be localizable with respect to τ_h. For an inner face $e \in \mathring{\mathcal{F}}_h$, such that $e = t^- \cap t^+$ for elements $t^\pm \in \tau_h$, we define the* local SWIPDG coupling evaluations

$$\begin{aligned}
\mathring{\Upsilon}^{e--}_{\mathrm{SWIP}} &: Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \to \big[\hat{\mathbb{P}}(t^-) \times \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(e)\big],\\
\mathring{\Upsilon}^{e-+}_{\mathrm{SWIP}} &: Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \to \big[\hat{\mathbb{P}}(t^-) \times \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(e)\big],\\
\mathring{\Upsilon}^{e+-}_{\mathrm{SWIP}} &: Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \to \big[\hat{\mathbb{P}}(t^+) \times \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(e)\big] \qquad \textit{and}\\
\mathring{\Upsilon}^{e++}_{\mathrm{SWIP}} &: Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \to \big[\hat{\mathbb{P}}(t^+) \times \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(e)\big]
\end{aligned}$$

by (3.1.22)*. Note that these local evaluations form a local quaternary evaluation in the sense of Definition 3.1.26. We define the* local SWIPDG coupling operators

$$\begin{aligned}
\mathring{B}^{e--}_h &: \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(t^-)', & & \mathring{B}^{e-+}_h : \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(t^+)',\\
\mathring{B}^{e+-}_h &: \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(t^-)' & \textit{and} \quad & \mathring{B}^{e++}_h : \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(t^+)'
\end{aligned}$$

for all combinations of $-$ and $+$ by

$$\mathring{B}^{e\pm\pm}_h := \Sigma^{e\pm\pm}_h\big(\mathring{\Upsilon}^{e\pm\pm}_{\mathrm{SWIP}}[\lambda,\kappa]\big),$$

where $\Sigma^{e\pm\pm}_h$ denote the local coupling integral operators from Definition 3.1.26.

We are now in the position to define the discrete SWIPDG operators $\mathring{B}_h$ and $\overline{B}_h$ and the discrete SWIPDG functional $\bar{l}_{h,D}$ which are required to approximate b_h from (3.1.18) and $l_{h,D}$ from (3.1.19), respectively.

Definition 3.1.28 (SWIPDG operators and functional)**.** *With the notation from Examples 3.1.23 3.1.25 and 3.1.27 we define the* SWIDPG coupling operator $\mathring{B}_h : Q(\tau_h) \to Q(\tau_h)'$*, for inner faces $e \in \mathring{\mathcal{F}}_h$ with $e = t^- \cap t^+$ for $t^\pm \in \tau_h$, by*

$$\begin{aligned}
\mathring{B}_h[p](q) := \sum_{e\in\mathring{\mathcal{F}}_h} \Big(\quad & \mathring{B}^{e--}_h[p^{t^-}](q^{t^-}) \;+\; \mathring{B}^{e-+}_h[p^{t^-}](q^{t^+})\\
&+\; \mathring{B}^{e+-}_h[p^{t^+}](q^{t^-}) \;+\; \mathring{B}^{e++}_h[p^{t^+}](q^{t^+})\Big)
\end{aligned}$$

and the SWIPDG boundary operator $\overline{B}_{h,D} : Q(\tau_h) \to Q(\tau_h)'$ *and* SWIDG boundary functional $\bar{l}_{h,D} : Q(\tau_h) \to \mathbb{R}$*, for Dirichlet boundary faces $e \in \overline{\mathcal{F}}_{h,D}$ with $e = t \cap \Gamma_D$ for $t \in \tau_h$, by*

$$\overline{B}_{h,D}[p](q) := \sum_{e\in\overline{\mathcal{F}}_{h,D}} \overline{B}^e_{h,D}[p^t](q^t) \qquad \textit{and} \qquad \bar{l}_{h,D}(q) := \sum_{e\in\overline{\mathcal{F}}_{h,D}} \bar{l}^e_{h,D}(q^t),$$

respectively, for all $p, q \in Q(\tau_h)$.

Algorithm 3.1.29 Local assembly of global matrices and vectors (SWIPDG FEM).

```
Input: a discrete function space $Q_h^k(\tau_h)$
Output: $\underline{B_h}, \underline{\mathring{B}_h}, \underline{\overline{B}_{h,D}} \in \mathbb{R}^{I \times I}$ and $\underline{l_{h,f}}, \underline{l_{h,g_N}}, \underline{\bar{l}_{h,D}} \in \mathbb{R}^I$
  initialize $\underline{B_h}$, $\underline{\mathring{B}_h}$, $\underline{\overline{B}_{h,D}}$, $\underline{l_{h,f}}$, $\underline{l_{h,g_N}}$ and $\underline{\bar{l}_{h,D}}$ with zero entries
  for all $t \in \tau_h$ do
      for all $0 \leq i < I(t)$ do                                        ▷compute volume integrals
          $\big(\underline{l_{h,f}}\big)_{\iota(t,i)} \leftarrow \big(\underline{l_{h,f}}\big)_{\iota(t,i)} + l_{h,f}^t(\varphi_i^t)$
          for all $0 \leq j < I(t)$ do
              $\big(\underline{B_h}\big)_{\iota(t,i),\iota(t,j)} \leftarrow \big(\underline{B_h}\big)_{\iota(t,i),\iota(t,j)} + B_h^t[\varphi_j^t](\varphi_i^t)$
          end for
      end for
      for all $e \in \overline{\mathcal{F}}_{h,N}^t$ do                              ▷compute Neumann face integrals
          for all $0 \leq i < I(t)$ do
              $\big(\underline{l_{h,g_N}}\big)_{\iota(t,i)} \leftarrow \big(\underline{l_{h,g_N}}\big)_{\iota(t,i)} + l_{h,g_N}^e(\varphi_i^t)$
          end for
      end for
      for all $e \in \overline{\mathcal{F}}_{h,D}^t$ do                              ▷compute Dirichlet face integrals
          for all $0 \leq i < I(t)$ do
              $\Big(\underline{\bar{l}_{h,D}}\Big)_{\iota(t,i)} \leftarrow \Big(\underline{\bar{l}_{h,D}}\Big)_{\iota(t,i)} + \bar{l}_{h,D}^e(\varphi_i^t)$
              for all $0 \leq j < I(t)$ do
                  $\Big(\underline{\overline{B}_{h,D}}\Big)_{\iota(t,i),\iota(t,j)} \leftarrow \Big(\underline{\overline{B}_{h,D}}\Big)_{\iota(t,i),\iota(t,j)} + \overline{B}_{h,D}^e[\varphi_j^t](\varphi_i^t)$
              end for
          end for
      end for
      for all $e \in \mathring{\mathcal{F}}_h^t$ do with $t^- := t$ and $t^+ \in \tau_h$, such that $e = t^- \cap t^+$: ▷compute inner face integrals
          if $t^- < t^+$ then          ▷(visit each face only once, given an ordering of the elements)
              for all $0 \leq i < I(t^-)$ do
                  for all $0 \leq j < I(t^-)$ do
                      $\Big(\underline{\mathring{B}_h}\Big)_{\iota(t^-,i),\iota(t^-,j)} \leftarrow \Big(\underline{\mathring{B}_h}\Big)_{\iota(t^-,i),\iota(t^-,j)} + \mathring{B}_h^{e^{--}}[\varphi_j^{t^-}](\varphi_i^{t^-})$
                  end for
                  for all $0 \leq j < I(t^+)$ do
                      $\Big(\underline{\mathring{B}_h}\Big)_{\iota(t^-,i),\iota(t^+,j)} \leftarrow \Big(\underline{\mathring{B}_h}\Big)_{\iota(t^-,i),\iota(t^+,j)} + \mathring{B}_h^{e^{-+}}[\varphi_j^{t^-}](\varphi_i^{t^+})$
                  end for
              end for
              for all $0 \leq i < I(t^+)$ do
                  for all $0 \leq j < I(t^-)$ do
                      $\Big(\underline{\mathring{B}_h}\Big)_{\iota(t^+,i),\iota(t^-,j)} \leftarrow \Big(\underline{\mathring{B}_h}\Big)_{\iota(t^+,i),\iota(t^-,j)} + \mathring{B}_h^{e^{+-}}[\varphi_j^{t^+}](\varphi_i^{t^-})$
                  end for
                  for all $0 \leq j < I(t^+)$ do
                      $\Big(\underline{\mathring{B}_h}\Big)_{\iota(t^+,i),\iota(t^+,j)} \leftarrow \Big(\underline{\mathring{B}_h}\Big)_{\iota(t^+,i),\iota(t^+,j)} + \mathring{B}_h^{e^{++}}[\varphi_j^{t^+}](\varphi_i^{t^+})$
                  end for
              end for
          end if
      end for
  end for
```

With these definitions the fully discretized variant of (3.1.17) then reads: for $k \geq 1$ find $p_h \in Q_h^k(\tau_h)$, such that

$$B_h[p_h](q_h) + \mathring{B}_h[p_h](q_h) + \overline{B}_{h,D}[p_h](q_h) = l_{h,f}(q_h) + l_{h,g_N}(q_h) + \bar{l}_{h,D}(q_h), \qquad (3.1.23)$$

for all $q_h \in Q_h^k(\tau_h)$, where the elliptic operator B_h and the L^2 functionals $l_{h,f}$ and l_{h,g_N} are given by Definition 3.1.14.

Analogous to (3.1.14) we define the matrix and vector representation of the operators $\mathring{B}_h$ and $\overline{B}_{h,D}$ and the functional $\bar{l}_{h,D}$ with respect to the global basis $\phi = \{\varphi_0, \dots, \varphi_{I-1}\}$ of $Q_h^k(\tau_h)$, namely $\underline{\mathring{B}_h} \in \mathbb{R}^{I \times I}$, $\underline{\overline{B}_{h,D}} \in \mathbb{R}^{I \times I}$ and $\underline{\bar{l}_{h,D}} \in \mathbb{R}^I$, by

$$\big(\underline{\mathring{B}_h}\big)_{i,j} := \mathring{B}_h[\varphi_j](\varphi_i), \qquad \Big(\underline{\overline{B}_{h,D}}\Big)_{i,j} := \overline{B}_{h,D}[\varphi_j](\varphi_i) \qquad \text{and} \qquad \Big(\underline{\bar{l}_{h,D}}\Big)_i := \bar{l}_{h,D}(\varphi_i),$$

respectively, for all $0 \leq i, j \leq I - 1$.

Similar to the previous paragraph we assemble these containers in a localized fashion (see Algorithm 3.1.29) using the fact that $Q_h^k(\tau_h)$ provides a local basis. The algebraic problem then reads: find $\underline{p_h} \in \mathbb{R}^I$, such that

$$\Big(\underline{B_h} + \underline{\mathring{B}_h} + \underline{\overline{B}_{h,D}}\Big)\, \underline{p_h} = \underline{l_{h,f}} + \underline{l_{h,g_N}} + \underline{\bar{l}_{h,D}}, \qquad (3.1.24)$$

where $\underline{p_h} \in \mathbb{R}^I$ is the DoF vector of the solution $p_h \in Q_h^k(\tau_h)$ of (3.1.23).

This finalizes the discontinuous Galerkin Finite Element approximation of (3.1.1) using a SWIPDG discretization.

While approximating the solution of a PDE is surely one of the most important tasks we require of a discretization framework, we also need to assess the quality of the resulting approximation.

3.1.1.2 Error estimation

This section deals with the assessment of the approximation quality of a discretization by estimating the *discretization error* $e_h := p - p_h$ (where "estimating" is not to be solely understood in the sense of a posterior error estimation, see the discussion below). As usual, p denotes the exact solution of (3.1.1) and p_h the discrete solution of (3.1.12) or (3.1.23), associated with a grid τ_h and an approximation space $S_h^k(\tau_h)$ or $Q_h^k(\tau_h)$. We know from a priori theory (compare Section 1.1), that the discretization error is bounded as in

$$\|e_h\|_{L^2} \lesssim h^{k+1}, \qquad \|e_h\|_{H^1} \lesssim h^k \qquad \text{and} \qquad |\!|\!|e_h|\!|\!| \lesssim h^k,$$

where $h > 0$ denotes the typical width of an element of τ_h, $k > 0$ denotes the polynomial degree of the ansatz space, $|\!|\!|\cdot|\!|\!|$ denotes the energy norm induced by the bilinear form b and $\lesssim$ denotes a proportionality relationship (compare Sections 1.1 and 2.1). We thus know the rate by which the approximation should improve whenever we decrease h (for instance by refining the grid τ_h): namely k or $k+1$, depending on the norm in question.

There are several ways to estimate the norm of the discretization error, in particular: by a direct comparison of the approximate solution with either a known exact or a computed reference solution, or indirectly by an evaluation of a reliable a posteriori error estimate η_h (1.1.7); each of these options is presented in detail further below. Given any such (direct or indirect) estimate on the discretization error, we compute the *experimental order of convergence* (EOC) of this quantity and compare it to the expected rate, k or $k+1$.

We therefore consider a series of refined grids $\tau_h^{(0)}, \tau_h^{(1)}, \tau_h^{(2)}, \dots$ with respective grid widths $h^{(0)} > h^{(1)} > h^{(2)} > \dots > 0$. For each level $\nu = 0, 1, 2, \dots$ we compute an approximate solution $p_h^{(\nu)}$ and compute the estimate on the discretization error, say $\tilde{e}_h^{(\nu)}$, by one of the means presented below. We then compare its values on different grid refinements, to obtain the corresponding EOC (for all $\nu > 0$):

$$EOC_{\tilde{e}_h^{(\nu)}} := \frac{\ln\left(\frac{\tilde{e}_h^{(\nu)}}{\tilde{e}_h^{(\nu-1)}}\right)}{\ln\left(\frac{h^{(\nu)}}{h^{(\nu-1)}}\right)}.$$

If, for a series of refined grids, the EOC of the quantity in question is close to its theoretical order (k or $k+1$), we have good reason to trust our discretization framework.

In the next paragraphs, we discuss several means to actually compute an estimate on the discretization error, and derive further requirements for a discretization framework.

Comparison with a known solution. In some (rare) cases we actually do explicitly know the exact solution p of (3.1.1) for given data functions. Though usually of academic nature, these problems are important to easily test the approximation quality of our discretization framework. Since we presume all data functions as well as the exact solution $p \in Q(\tau_h)$ to be localizable with respect to τ_h and since the discrete solution p_h is an element of a discrete function space $\mathbb{V}_h(\tau_h) \subset Q(\tau_h)$, we can compare the two in $Q(\tau_h)$ in a meaningful way: $p - p_h = e_h \in Q_h(\tau_h)$.

The question of how to compute the norm of the discretization error thus simplifies to the question of how to compute the norm of a localizable function $e_h \in Q(\tau_h)$. Given a grid τ_h, we consider norms

$$\|\cdot\|_* : Q(\tau_h) \to \mathbb{R}, \qquad\qquad q \mapsto \|q\|_* := (q,q)_*^{1/2},$$

which are induced by a corresponding scalar product $(\cdot,\cdot)_* : Q(\tau_h) \times Q(\tau_h) \to \mathbb{R}$. While the energy scalar product is induced by the elliptic operator B from (3.1.3),

$$|||q|||^2 := (((q,q))) := B[q](q) = \int_\Omega \left(\lambda\kappa\nabla_h q\right) \cdot \nabla_h q \,\mathrm{d}x,$$

we express the L^2-product or the H^1-semiproduct directly by

$$(q,q)_{L^2(\Omega)} := \int_\Omega q^2 \,\mathrm{d}x \qquad \text{and} \qquad (q,q)_{H^1(\Omega)} := \int_\Omega \nabla q^2 \,\mathrm{d}x, \tag{3.1.25}$$

respectively. In order to actually compute the evaluation of these products, we need to numerically approximate the arising integrals. We have already done so for the elliptic operator inducing the energy product by localization with respect to the grid and by numerical quadratures, yielding

$$(\!(\!(q, p)\!)\!) = \sum_{t \in \tau_h} B_h^t[p^t](q^t),$$

with the local elliptic operator B_h^t from Example 3.1.7, where q^t denotes the local function of q with respect to $t \in \tau_h$ in the sense of Definition 3.1.2. We recall that the local elliptic operator is defined in terms of an abstract local volume integral operator Σ_h^t (approximating the integral) and a local elliptic evaluation (modeling the integrand).

We obtain approximations of the other products by defining the appropriate local evaluation and using existing local operators.

Example 3.1.30 (Local product evaluation and local L^2 product operator)**.** *Let τ_h be a grid and let $q \in Q(\tau_h)$ be localizable with respect to τ_h. We define the* local product evaluation $\Upsilon_{\text{prod.}}^t : Q(\tau_h) \to \big[\hat{\mathbb{P}}(t) \times \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)\big]$ *and the* local L^2 product operator $(\cdot,\cdot)_{L^2}^t : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)'$ *by*

$$\Upsilon_{\text{prod.}}^t[q]\big(\psi^t, \varphi^t\big) := q^t\,\psi^t\,\varphi^t \qquad \textit{and} \qquad (\cdot,\cdot)_{L^2}^t := \Sigma_h^t\big(\Upsilon_{\text{prod.}}^t[\mathbb{1}]\big)[\cdot](\cdot),$$

respectively, where Σ_h^t denotes the local volume integral operator from Definition 3.1.6 and $\mathbb{1} \in Q(\tau_h)$ denotes a function mapping to $1 \in \mathbb{R}$. Note that $\Upsilon_{\text{prod.}}^t[\mathbb{1}]$ is a local binary volume evaluation in the sense of Definition 3.1.6.[9]

Example 3.1.31 (Local H^1-semi product operator)**.** *With the notation from Example 3.1.7 we define the* local H^1-semi product operator $(\cdot,\cdot)_{H^1}^t : \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)'$ *by*

$$(\cdot,\cdot)_{L^2}^t := \Sigma_h^t\big(\Upsilon_{\text{ell.}}^t[\mathbb{1}, \mathbb{1}_d]\big)[\cdot](\cdot),$$

respectively, where Σ_h^t denotes the local volume integral operator from Definition 3.1.6 and $\mathbb{1}_d \in [Q(\tau_h)]^{d\times d}$ denotes a function mapping to the unit matrix in $\mathbb{R}^{d\times d}$.

Note that the above example demonstrates the usefulness of the concept of local evaluations, since we can easily express the localization of the integrand in (3.1.25) by an existing local evaluation:[10]

$$\Sigma_h^t\big(\Upsilon_{\text{ell.}}^t[\mathbb{1}, \mathbb{1}_d]\big)[\varphi^t](\psi^t) = \int_{\hat{t}} \Lambda_t \nabla_t \psi^t \cdot \nabla_t \varphi^t \,\mathrm{d}x = \int_t \nabla\psi \cdot \nabla\varphi \,\mathrm{d}x$$

With these local product operators, we obtain the discrete variants of (3.1.25).

[9]Note that we use the same notation for the unary as well as for the binary local product evaluation, $\Upsilon_{\text{prod.}}^t$ (compare Example 3.1.10); it is clear from the context which of the two is used. The above definition of the local binary product evaluation also allows for weighted L^2 products, which we require in the context of a posteriori error estimation.

[10]While this is a rather simple case of a reuse of local evaluations the benefit will be more apparent in the context of a posterior error estimation further below, we we reuse the local SWIPDG evaluations.

Definition 3.1.32 (Discrete L^2 and H^1-semi product operators)**.** *With the notation from Example 3.1.30 and 3.1.31 we define the* discrete L^2 and H^1-semi product operators $(\cdot,\cdot)_{h,L^2} : Q(\tau_h) \to Q(\tau_h)'$ *and* $(\cdot,\cdot)_{h,H^1} : Q(\tau_h) \to Q(\tau_h)'$ *by*

$$(q,p)_{h,L^2} := \sum_{t\in\tau_h} \big(q^t,p^t\big)^t_{L^2} \qquad \text{and} \qquad (q,p)_{h,H^1} := \sum_{t\in\tau_h} \big(q^t,p^t\big)^t_{H^1},$$

respectively, for $p,q \in Q(\tau_h)$*.*

We assemble the evaluation of these products in a localized fashion, analogously to Algorithm 3.1.18.

Algorithm 3.1.33 Local computation of global norms.

```
Input: a local product $(\cdot,\cdot)^t_*$, a localizable function $e_h \in Q(\tau_h)$
Output: $\|e_h\|_*$
  $\|e_h\|^2_* \leftarrow 0$
  for all $t \in \tau_h$ do
      $\|e_h\|^2_* \leftarrow \|e_h\|^2_* + \big(e^t_h, e^t_h\big)^t_*$
  end for
```

Comparison with a more detailed discrete solution. In general, we do not have access to the exact solution p of (3.1.1) to compare our approximate solution to. In order to still have a way to compute an approximate EOC on the first $\nu \in \mathbb{N}$ levels of the grid, we substitute the unknown exact solution by a discrete approximation on a more refined *reference grid*, say $p_h^{(\nu')} \in \mathbb{V}_h(\tau_h^{(\nu')})$, with $\nu' \in \mathbb{N}$, such that $\nu < \nu'$. Since we do not considering a coarsening of the grid, we obtain a set of nested discrete function spaces for each grid refinement, $\mathbb{V}_h(\tau_h^{(0)}) \subset \mathbb{V}_h(\tau_h^{(1)}) \subset \mathbb{V}_h(\tau_h^{(2)}) \subset \cdots \subset \mathbb{V}_h(\tau_h^{(\nu)}) \subset \cdots \subset \mathbb{V}_h(\tau_h^{(\nu')})$, such that we can compare all quantities in $\mathbb{V}_h(\tau_h^{(\nu')}) \subset Q(\tau_h^{(\nu')})$. (As usual, $\mathbb{V}_h$ denotes a discrete function space in the sense of Definition 3.1.16 and Q denotes the space of localizable functions in the sense of Definition 3.1.4.)

To this end, we *prolong* the discrete approximations $p_h^{(\nu)} \in \mathbb{V}_h(\tau_h^{(\nu)})$, for each level $\nu < \nu'$, onto the reference grid $\tau_h^{(\nu')}$ by means of a *prolongation operator*

$$\Pi^{\tau_h^{(\nu)}}_{\mathbb{V}_h(\tau_h^{(\nu')})} : Q(\tau_h^{(\nu)}) \to \mathbb{V}_h(\tau_h^{(\nu')}). \tag{3.1.26}$$

We present several prolongation operators further below, along with projection operators. Given a suitable prolongation operator for all $\nu < \nu'$, the question of how to obtain an approximation of the discretization error simplifies to the question of how to compute the norm of a discrete function

$$p_h^{(\nu')} - \Pi^{\tau_h^{(\nu)}}_{\mathbb{V}_h(\tau_h^{(\nu')})}[p_h^{(\nu)}] := e_h^{(\nu)} \quad \in \mathbb{V}_h(\tau_h^{(\nu')}).$$

Since $e_h^{(\nu)}$ is localizable with respect to $\tau_h^{(\nu')}$ we can employ the techniques of the previous paragraph to compute any norm of $e_h^{(\nu)}$ in a localized manner.

However, we can also make use of the fact, that $e_h^{(\nu)}$ is a discrete function in the sense of Definition 3.1.17, belonging to the discrete function space $\mathbb{V}_h(\tau_h^{(\nu')})$. Since the local polynomial degree of all discrete functions belonging to a discrete function space is fixed, we can precompute a matrix representation of the product in question with respect to this discrete function space, as detailed in the following algorithm, where we reuse the local product operators from the previous paragraph.

Algorithm 3.1.34 Local assembly of global product matrices.

```
Input: a local product $(\cdot,\cdot)_*^t$, a discrete function space $\mathbb{V}_h(\tau_h)$
Output: $\underline{(\cdot,\cdot)_*} \in \mathbb{R}^{I\times I}$ (the matrix representation of $(\cdot,\cdot)_*$)
  initialize $\underline{(\cdot,\cdot)_*}$ with zero entries
  for all $t \in \tau_h$ do
    for all $0 \leq i < I(t)$ do with $\varphi_i^t$ the $i$th element of the local basis $\phi_{k(t)}^t$ of $\mathbb{V}_h(\tau_h)$:
      for all $0 \leq j < I(t)$ do with $\varphi_j^t$ the $j$th element of the local basis $\phi_{k(t)}^t$ of $\mathbb{V}_h(\tau_h)$:
        $\big(\underline{(\cdot,\cdot)_*}\big)_{\iota(t,i),\iota(t,j)} \leftarrow \big(\underline{(\cdot,\cdot)_*}\big)_{\iota(t,i),\iota(t,j)} + \big(\varphi_j^t, \varphi_i^t\big)_*^t$
      end for
    end for
  end for
```

Given such a matrix representation $\underline{(\cdot,\cdot)_*} \in \mathbb{R}^{I\times I}$ of a product $(\cdot,\cdot)_* : \mathbb{V}_h(\tau_h) \times \mathbb{V}_h(\tau_h) \to \mathbb{R}$, we can compute the induced norm of any discrete function $e_h \in \mathbb{V}_h(\tau_h)$ (e.g., for the discretization errors in each refinement step) by a simple matrix vector multiplication using e_h's DoF vector $\underline{e_h} \in \mathbb{R}^I$:

$$\|e_h\|_* = \sqrt{\underline{e_h}^\perp \underline{(\cdot,\cdot)_*}\,\underline{e_h}}.$$

While the comparison with a more detailed discrete solution allows us to estimate the discretization error in cases where we do not have access to the exact solution, it is computationally not always feasible (since it requires a discrete approximation on a finer grid). In addition we do not obtain any guarantee on how well we approximated the evaluation of the norm of the discretization error (since we do not use the unknown exact solution to compare with). We usually obtain good results if the reference grid is much finer than the original one, but this may not be the case for finer approximations, e.g., where $\nu = \nu' - 1$. In many cases, a good way of computing an estimate on the discretization error is by means of a reliable a posteriori error estimate (if available).

A posteriori error estimation. There exist a large variety of a posteriori error estimators and we refer to [Ver1996, Ver2013] for an overview. We consider the localized a posteriori error estimate from Section 2.3.2, which (being formulated for the parametric case) is based on a similar estimate on the discretization error for the nonparametric case, introduced in [ESV2010]. The form of the latter is very similar to Corollary 2.3.6, except for the parameter dependency. Given a solution $p_h \in Q_h(\tau_h)$ of the discrete Problem (3.1.23), the estimate $\eta_h(p_h)$ consists of three components, *(i)* a local nonconformity estimator $\eta^t_{\mathrm{nc}}(p_h) = |||p_h - I_{OS}[p_h]|||_t$,

(ii) a local residual estimator $\eta^t_{\mathrm{r}}(p_h) = (C^t_P/c^t_\varepsilon)^{1/2} h_t \big\| f - \nabla\cdot R^l_h[p_h] \big\|_{L^2,t}$ and

(iii) a local diffusive flux estimator $\eta^t_{\mathrm{df}}(p_h) = \big\|(\lambda\kappa)^{-1/2}(\lambda\kappa\nabla_h p_h + R^l_h[p_h])\big\|_{L^2,t}$,

where I_{OS} denotes the Oswald interpolation operator from (2.3.16), R^l_h denotes the diffusive flux reconstruction from (2.3.17) and $C^t_P, c^t_\varepsilon \in \mathbb{R}$ denote positive constants.

We briefly discuss the ingredients we require in order to realize each of these local estimators.

(i) The Oswald interpolation operator I_{OS} actually coincides with the generalized Lagrange projection operator $\Pi_{S^k_h(\tau_h)}$ we present further below. To compute the nonconformity estimator we can thus use the local elliptic operator B^t_h from Example 3.1.7 as detailed in the previous paragraphs, applied to the local function of $p_h - \Pi_{S^k_h(\tau_h)}[p_h] \in Q(\tau_h)$.

(ii) Since the divergence of the image of the diffusive flux reconstruction operator R^l_h (the implementation of which we discuss below) is localizable with respect to the grid, $f - \nabla\cdot R^l_h[p_h] \in Q(\tau_h)$ is as well. And since $(C^t_P/c^t_\varepsilon)^{1/2} h_t\big|_t \in \mathbb{P}_0(t)$ is positive for all $t \in \tau_h$ we use the product evaluation $\Upsilon^t_{\mathrm{prod.}}[c]$ from Example 3.1.30 and the local volume operator from Definition 3.1.6 to compute the residual estimator, with $c \in Q(\tau_h)$ such that $c|_t = (C^t_P/c^t_\varepsilon)^{1/2} h_t$ for all $t \in \tau_h$.

(iii) To approximate the diffusive flux estimator we use the local volume operator from Definition 3.1.6 as well, but require a special local evaluation: given localizable $\lambda \in Q(\tau_h)$, $\kappa \in [Q(\tau_h)]^{d\times d}$ and $v \in [Q(\tau_h)]^d$, we define $\Upsilon^t_{\mathrm{df}} : Q(\tau_h) \times [Q(\tau_h)]^{d\times d} \times [Q(\tau_h)]^d \to \big[\hat{\mathbb{P}}(t) \times \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(t)\big]$ by

$$\Upsilon^t_{\mathrm{df}}[\lambda,\kappa,v](\psi^t,\varphi^t) := (\lambda^t\kappa^t)^{-1}\big(\lambda^t\kappa^t\nabla_t\psi^t + v^t\big)\cdot\big(\lambda^t\kappa^t\nabla_t\varphi^t + v^t\big).$$

Given the diffusive flux reconstruction $R^l_h[p_h]$ we then evaluate the local product operator $\Sigma^t_h\big(\Upsilon^t_{\mathrm{df}}\big[\lambda,\kappa,R^l_h[p_h]\big]\big)(p^t_h,p^t_h)$. Note that $(\lambda^t\kappa^t)^{-1}$ in the definition of Υ^t_{df} is in general not polynomial any more and Υ^t_{df} thus only yields an approximate local evaluation.

For the remainder of this paragraph we discuss the diffusive flux reconstruction $R^l_h[p_h] \in V^l_h(\tau_h)$ in a Raviart-Thomas-Nédélec space of order $l \in \mathbb{N}$:

$$V^l_h(\tau_h) := \big\{v \in H_{\mathrm{div}}(\Omega)\big|\, v|_t \in [\mathbb{P}_l(t)]^d + \boldsymbol{x}\mathbb{P}_l(t) \quad \forall t \in \tau_h\big\}. \tag{3.1.27}$$

The local DoFs of an element of V^l_h are associated with the faces of a grid element and the grid element itself, depending on the dimension d, the polynomial degree l and the shape of the grid element (for simplicity we restrict ourselves to triangles and orders 0 and 1; for a definition of $V^l_h(\tau_h)$ for higher orders, higher dimensions and other grids, see [BF1991, §*III*.3]).

Definition 3.1.35 (Raviart-Thomas-Nédélec discrete function space)**.** *Let τ_h be a simplicial triangulation of $\Omega \subset \mathbb{R}^2$ and let $V_h^l(\tau_h)$ be a discrete function space fulfilling* (3.1.27) *with $l = 0, 1$. We call $V_h^l(\tau_h)$ a* Raviart-Thomas-Nédélec discrete function space *of order l, iff, for all $t \in \tau_h$ there exist the following bijections: between the elements of the local basis*

- *and the faces of the element ($\phi_{l(t)}^t \leftrightarrow \mathcal{F}_h^t$) for $l = 0$;*
- *and the faces of the element and the element itself ($\phi_{l(t)}^t \leftrightarrow \{\mathcal{F}_h^t \cup t\}$) for $l = 1$.*

Similar to the parametric case, $R_h^l[p_h]$ is given analogously to (2.3.17) and the arising nonparametric local coupling and penalty bilinear forms b^t, b_c^e and b_p^e coincide with the individual terms in (3.1.18), ignoring $\boldsymbol{\mu}$. We already discretized those using the local SWIPDG evaluations in the previous paragraphs and the corresponding local operators and functionals. We briefly discuss how to obtain the local DoFs of the diffusive flux reconstruction $R_h^0[p]$ of lowest order, given a localizable function $p \in Q(\tau_h)$.

For $l = 0$, these local DoFs are associated with the faces of the grid only. We transform the integrals in (2.3.17a) to the reference face, for a boundary face $e \in \overline{\mathcal{F}}_{h,D}$ with $t \in \tau_h$ such that $e = t \cap \Gamma_D$:

$$\int_{\hat{e}} \Lambda_e \, \Upsilon_{\text{df}}^e\big[\Gamma_D\big]\big(\mathbb{1}^t, R_h^0[p]^t\big) \, \mathrm{d}x = \int_{\hat{e}} \Lambda_e \overline{\Upsilon}_{\text{SWIP}}^e\big[\Gamma_D, \lambda, \kappa\big]\big(\mathbb{1}^t, p^t\big) \, \mathrm{d}x, \tag{3.1.28}$$

where $\Upsilon_{\text{df}}^e\big[\Gamma_D\big]$ denotes the local evaluation from Example 3.1.36 below, $\overline{\Upsilon}_{\text{SWIP}}^e$ denotes the local SWIPDG boundary evaluation from Example 3.1.25 and where $\mathbb{1}^t \in \hat{\mathbb{P}}_0(t)$ denotes a local function mapping to $1 \in \mathbb{R}$, modeling the basis of $\mathbb{P}_0(e)$. Since $R_h^0[p] \in V_h^0(\tau_h)$ is a discrete function, we can express its local function in terms of the local basis $\phi_{k(t)}^t$ of $V_h^0(\tau_h)$ and the local DoF vector $\underline{R_h^0[p]}^t \in \mathbb{R}^{I(t)}$, with $I(t) := |\phi_{k(t)}^t| \in \mathbb{N}$, for all $t \in \tau_h$. Using this basis representation and the local boundary volume operator Σ_h^e from Definition 3.1.24 (to approximate the integral), we obtain from (3.1.28): find $\underline{R_h^0[p]}_i^t \in \mathbb{R}$, such that

$$\underline{R_h^0[p]}_i^t \; \Sigma_h^e\big(\Upsilon_{\text{df}}^e\big[\Gamma_D\big]\big)[\varphi_i^t](\mathbb{1}^t) = \Sigma_h^e\big(\, \overline{\Upsilon}_{\text{SWIP}}^e\big[\Gamma_D, \lambda, \kappa\big]\big)[p^t](\mathbb{1}^t), \tag{3.1.29}$$

where $i \in \{0, \dots, I(t) - 1\}$ denotes the index of the local DoF, which corresponds to the local basis function $\varphi_i^t \in \phi_{k(t)}^t$ that is associated with the face e (compare Definition 3.1.35).

We proceed with the integrals in (2.3.17b) in a similar manner. Since the normal component of $R_h^0[p]$ is continuous across faces of the grid, it suffices to consider only one of the local functions of $R_h^0[p]$ associated with the grid elements adjacent to a face (in order to fully specify the local DoFs associated with inner faces of the grid). Using the above basis representation we obtain from (2.3.17b), for an inner face $e \in \mathring{\mathcal{F}}_h$ with $t^\pm \in \tau_h$ such that $e = t^- \cap t^+$: find $\underline{R_h^0[p]}_i^{t^-} \in \mathbb{R}$, such that

$$\begin{aligned} \underline{R_h^0[p]}_i^{t^-} \; \Sigma_h^e\big(\Upsilon_{\text{df}}^{e^{--}}\big)[\varphi_i^{t^-}](\mathbb{1}^{t^-}) = \; & \Sigma_h^e\big(\, \mathring{\Upsilon}_{\text{SWIP}}^{e^{--}}\big[\lambda, \kappa\big]\big)[p^{t^-}](\mathbb{1}^{t^-}) \\ & + \Sigma_h^e\big(\, \mathring{\Upsilon}_{\text{SWIP}}^{e^{+-}}\big[\lambda, \kappa\big]\big)[p^{t^+}](\mathbb{1}^{t^-}), \end{aligned} \tag{3.1.30}$$

where the local evaluation $\Upsilon_{\text{df}}^{e^{--}}$ is given by the example below, $\mathring{\Upsilon}_{\text{SWIP}}^{e^{\pm-}}$ denote the local SWIPDG coupling evaluations from Example 3.1.27, Σ_h^e denotes the local coupling integral operator from Definition 3.1.26 and where $i \in \{0, \dots, I(t^-) - 1\}$ denotes the index of the local DoF, which corresponds to the local basis function $\varphi_i^{t^-} \in \phi_{k(t^-)}^{t^-}$ that is associated with the face e (compare Definition 3.1.35).

Example 3.1.36 (Local evaluation)**.** *Let τ_h be a grid. For a boundary face $e \in \overline{\mathcal{F}}_{h,D}$, such that $e = t \cap \Gamma_D$ for an element $t \in \tau_h$, we define the local boundary evaluation $\Upsilon_{\text{df}}^e\big[\Gamma_D\big] : [\hat{\mathbb{P}}(t)]^d \times \hat{\mathbb{P}}(t) \to \hat{\mathbb{P}}(e)\big]$ by*

$$\Upsilon_{\text{df}}^e\big[\Gamma_D\big]\big(\psi^t, \varphi^t\big) := \psi_e^t \cdot n_e\, \varphi_e^t.$$

For an inner face $e \in \mathring{\mathcal{F}}_h$, such that $e = t^- \cap t^+$ for elements $t^\pm \in \tau_h$, we define the local coupling evaluations

$$\Upsilon_{\text{df}}^{e^{--}} : [\hat{\mathbb{P}}(t^-)]^d \times \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(e), \qquad \Upsilon_{\text{df}}^{e^{-+}} : [\hat{\mathbb{P}}(t^-)]^d \times \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(e),$$
$$\Upsilon_{\text{df}}^{e^{+-}} : [\hat{\mathbb{P}}(t^+)]^d \times \hat{\mathbb{P}}(t^-) \to \hat{\mathbb{P}}(e) \quad \text{and} \quad \Upsilon_{\text{df}}^{e^{++}} : [\hat{\mathbb{P}}(t^+)]^d \times \hat{\mathbb{P}}(t^+) \to \hat{\mathbb{P}}(e)$$

by

$$\Upsilon_{\text{df}}^{e^{\pm\pm}}\big(\psi^{t^\pm}, \varphi^{t^\pm}\big) := \psi_e^{t^\pm} \cdot n_e\, \varphi_e^{t^\pm}$$

for all combinations of $-$ and $+$. Note that $\Upsilon_{\text{df}}^e\big[\Gamma_D\big]$ is a local binary face evaluation in the sense of Definition 3.1.24 and that the four local evaluations $\Upsilon_{\text{df}}^{e^{\pm\pm}}$ form a local quaternary evaluation in the sense of Definition 3.1.26.

Due to the nature of the Raviart-Thomas-Nédélec space we have completely specified the discrete function $R_h^0[p] \in V_h^0(\tau_h)$ by specifying its local DoFs for all faces of the grid, as summarized in Algorithm 3.1.37.

Algorithm 3.1.37 Local computation of the diffusive flux reconstruction.

```
Input: a Raviart-Thomas-Nédélec space $V_h^0(\tau_h)$, localizable $\lambda, p \in Q(\tau_h)$ and $\kappa \in [Q(\tau_h)]^{d\times d}$
Output: the diffusive flux reconstruction $R_h^0[p]$
  initialize $\underline{R_h^0[p]} \in \mathbb{R}^I$
  for all $t \in \tau_h$ do
    for all $e \in \overline{\mathcal{F}}_h^t$ do                                   ▷ compute boundary DoFs
      compute $\underline{R_h^0[p]}_i^t \in \mathbb{R}$ according to (3.1.29)
      $\big(\underline{R_h^0[p]}\big)_{\iota(i,t)} \leftarrow \underline{R_h^0[p]}_i^t$
    end for
    for all $e \in \mathring{\mathcal{F}}_h^t$ do with $t^- := t$ and $t^+ \in \tau_h$, such that $e = t^- \cap t^+$:   ▷ compute inner DoFs
      if $t^- < t^+$ then                                                  ▷ (visit each face only once)
        compute $\underline{R_h^0[p]}_i^{t^-} \in \mathbb{R}$ according to (3.1.30)
        $\big(\underline{R_h^0[p]}\big)_{\iota(i,t^-)} \leftarrow \underline{R_h^0[p]}_i^{t^-}$
      end if
    end for
  end for
```

3.1.1.3 Projections and prolongations.

We have already encountered a scenario which requires the prolongation of a discrete function: in order to determine the EOC of an error norm we require to prolong a discrete function $q_h \in \mathbb{V}_h^k(\tau_h^{(\nu)})$, associated with a coarse grid onto a discrete function space $\mathbb{V}_h^l(\tau_h^{(\nu')})$ associated with a finer grid, to compare it to a reference solution (Equation 3.1.26). A similar situation arises in the context of adaptive Finite Element methods (compare Section 1.1).

In general we require a prolongation operator whenever we are given a localizable function associated with a source grid τ_h^s that we wish to interpret in the context of a range grid τ_h^r. In the above cases the range grid is simply a refinement of the source grid, such that $Q(\tau_h^s) \subset \mathbb{V}_h(\tau_h^r)$. In the context of multiscale methods, however, we also allow for two grids associated with different physical domains: while the source grid could be a partition of the full domain Ω, the range grid could be a partition of only a subdomain $T \subset \Omega$, possibly including overlap (compare the nested fine and coarse grids in Section 2.1). In that case, the prolongation operator also carries out a restriction and we have $Q(\tau_h^s)|_T \subset \mathbb{V}_h(\tau_h^r)$.

We can actually think of a prolongation as two distinct operations:

(*i*) Given a function $q \in Q(\tau_h^s)$, which is localizable with respect to τ_h^s, we require a function $\Pi_{\tau_h^r}^{\tau_h^s}[q] \in Q(\tau_h^r)$, which is localizable with respect to τ_h^r.

(*ii*) Given a function $q \in Q(\tau_h^r)$, which is localizable with respect to τ_h^r, we require a projection onto a discrete function space $\mathbb{V}_h^k(\tau_h^r)$, which we denote by $\Pi_{\mathbb{V}_h^k(\tau_h^r)}[q]$.

Thus, we have split the problem of prolonging a function into two separate problems and the prolongation operator from (3.1.26) is actually given by:

$$\Pi_{\mathbb{V}_h(\tau_h^r)}^{\tau_h^s} : Q_h(\tau_h^s) \to \mathbb{V}_h(\tau_h^r), \qquad\qquad \Pi_{\mathbb{V}_h(\tau_h^r)}^{\tau_h^s} := \Pi_{\mathbb{V}_h^k(\tau_h^r)} \circ \Pi_{\tau_h^r}^{\tau_h^s}.$$

We define the *reinterpretation operator*, $\Pi_{\tau_h^r}^{\tau_h^s} : Q(\tau_h^s) \to Q(\tau_h^r)$, locally by defining the local functions of its image, for all elements of the range grid $t^r \in \tau_h^r$:

$$\Pi_{\tau_h^r}^{\tau_h^s}[q]^{t^r} \in \hat{\mathbb{P}}(t^r), \text{ such that} \qquad\qquad \Pi_{\tau_h^r}^{\tau_h^s}[q]^{t^r}(\hat{x}^r) := q^{t^s}\Big({\Phi^{t^s}}^{-1}\big(\Phi^{t^r}(\hat{x}^r)\big)\Big),$$

where $t^s \in \tau_h^s$ denotes the element of the source grid, such that $\Phi^{t^r}(\hat{x}^r) \in t^s$.

Remark 3.1.38 (The reinterpretation operator $\Pi_{\tau_h^r}^{\tau_h^s}$)**.** *There are several things worth noting regarding the reinterpretation of a localizable function with respect to a different grid. The role of the reinterpretation operator is merely to provide a local function on each element of the range grid τ_h^r, by redirecting the evaluation to an appropriate local function on the source grid τ_h^s. Formally, we do not require any relationship between the source grid and the range grid, apart from the fact that the physical domain covered by τ_h^r has to be included in the physical domain covered by τ_h^s. While this also holds* *for the case where the source grid is coarser than the range grid, the definition of the operator* $\Pi_{\tau_h^r}^{\tau_h^s}$ *is not meaningful in that case and we would require a* restriction *operator (which we do not consider here). While the operator is easily defined in theory, a well performing implementation of* $\Pi_{\tau_h^r}^{\tau_h^s}$ *can be quite involved in practice (since it needs to find an appropriate* $t^s \in \tau_h^s$ *for each* $t^r \in \tau_h^r$*).*

Given the close connection between prolongation and projection operators, we only need to define the projection of an arbitrary localizable function q onto a discrete function space $\mathbb{V}_h^k(\tau_h)$. The actual definition of the projection operator depends on its range: in case of a continuous Lagrange space we can carry out the projection in a localized manner.

Definition 3.1.39 ((Generalized) Lagrange projection)**.** *Let* $S_h^k(\tau_h)$ *be a continuous Lagrange discrete function space in the sense of Definition 3.1.19 and let* q *be localizable with respect to* τ_h*. For globally continuous* q*, we define the* Lagrange projection operator $\Pi_{S_h^k(\tau_h)} : C^0(\Omega) \cap Q(\tau_h) \to S_h^k(\tau_h)$ *by specifying the local DoF vector* $\underline{\Pi_{S_h^k(\tau_h)}[q]^t} \in \mathbb{R}^{I(t)}$ *of the local functions of its image* $\Pi_{S_h^k(\tau_h)}[q]$*, by*

$$\left(\underline{\Pi_{S_h^k(\tau_h)}[q]^t}\right)_i := q^t(\hat{\nu}_i^t), \qquad\qquad \textit{for all } 0 \leq i < I(t) \textit{ and all } t \in \tau_h,$$

where $\hat{\nu}_i^t \in \hat{t}$ *denotes a Lagrange point, for all* $0 \leq i < I(t)$*. For an arbitrary localizable function* $q \in Q(\tau_h)$ *we define the* generalized Lagrange projection operator $\Pi_{S_h^k(\tau_h)} : Q(\tau_h) \to S_h^k(\tau_h)$ *by specifying*

$$\left(\underline{\Pi_{S_h^k(\tau_h)}[q]^t}\right)_i := \frac{1}{\left|\tau_h^{\hat{\nu}_i^t}\right|} \sum_{t' \in \tau_h^{\hat{\nu}_i^t}} q^{t'}\left(\hat{\Phi}^{t'^{-1}}\left(\hat{\Phi}^t(\hat{\nu}_i^t)\right)\right),$$

where $\tau_h^{\hat{\nu}_i^t} \subset \tau_h$ *denotes the set of those elements of the grid, that share the Lagrange point* $\hat{\Phi}^t(\hat{\nu}_i^t)$*.*

Though locally defined, the generalized Lagrange projection operator might involve global computations (in order to determine $\tau_h^{\hat{\nu}_i^t}$), depending on the implementation of the grid.

If the range of the projection operator is a discontinuous discrete function space, we can also carry out the projection locally.

Definition 3.1.40 ((Local) L^2 projection)**.** *Let* $Q_h^k(\tau_h)$ *be a discontinuous Galerkin discrete function space in the sense of Definition 3.1.22 and let* q *be localizable with respect to* τ_h*. We define the* L^2 projection operator $\Pi_{Q_h^k(\tau_h)} : Q(\tau_h) \to Q_h^k(\tau_h)$*, by specifying the local DoF vector* $\underline{\Pi_{Q_h^k(\tau_h)}[q]^t} \in \mathbb{R}^{I(t)}$ *of the local functions of its image* $\Pi_{Q_h^k(\tau_h)}[q]$*, as the solution of*

$$\underline{L_h^{2\,t}}\ \underline{\Pi_{Q_h^k(\tau_h)}[q]^t} = \underline{l_{h,q}^t}, \qquad\qquad \textit{for all } t \in \tau_h,$$

with the local mass matrix $\underline{L^2_h}^t \in \mathbb{R}^{I(t)\times I(t)}$ *and the local right hand side* $\underline{l^t_{h,q}} \in \mathbb{R}^{I(t)}$ *given by*

$$\big(\underline{L^2_h}^t\big)_{i,j} := \big(\varphi^t_i, \varphi^t_j\big)^t_{L^2} \qquad \text{and} \qquad \big(\underline{l^t_{h,q}}\big)_j := l^t_{h,q}(\varphi^t_j)$$

respectively, with the local basis functions $\varphi^t_i, \varphi^t_j \in \phi^t_{k(t)}$ *of the discrete function space, for all* $0 \le i, j < I(t) = |\phi^t_{k(t)}|$, *the local* L^2 *product operator* $(\cdot,\cdot)^t_{L^2}$ *from Example 3.1.30 and the local* L^2*-volume functional* $l^t_{h,q}$ *from Example 3.1.10 (with* f *replaced by* q*).*

We can carry out the above projection locally due to the discontinuity of the discrete function space. In general, however, we require the inversion of a global mass matrix.

Definition 3.1.41 ((Global) L^2 projection)**.** *Let* $\mathbb{V}_h(\tau_h)$ *be a discrete function space in the sense of Definition 3.1.16 and let* q *be localizable with respect to* τ_h*. We define the* L^2 projection operator $\Pi_{\mathbb{V}_h(\tau_h)} : Q(\tau_h) \to \mathbb{V}_h(\tau_h)$, *by specifying the DoF vector* $\underline{\Pi_{\mathbb{V}_h(\tau_h)}[q]} \in \mathbb{R}^I$ *of its image* $\Pi_{\mathbb{V}_h(\tau_h)}[q]$, *as the solution of*

$$\underline{L^2_h}\, \underline{\Pi_{\mathbb{V}_h(\tau_h)}[q]} = \underline{l_{h,q}},$$

with the global mass matrix $\underline{L^2_h} := \underline{(\cdot,\cdot)_{L^2}} \in \mathbb{R}^{I\times I}$ *given by the matrix representation of the* L^2 *product from above (compare Algorithm 3.1.34) and the global right hand side* $\underline{l_{h,q}} \in \mathbb{R}^I$ *given by the vector representation of the discrete* L^2 *functional from Definition 3.1.14 (compare also* (3.1.14) *with* f *replaced by* q*).*

Note that these global containers can also be assembled in a localized fashion, compare Algorithms 3.1.18 and 3.1.34.

Following this presentation and discussion of the (mathematical) ingredients we require of a discretization framework, we continue with a presentation of the main design principles for a designated discretization framework (which are a direct consequence of these ingredients).

3.1.2 Abstract design principles and technical requirements

We revisit the main concepts behind the mathematical foundation introduced in the previous section and derive abstract design principles that shall be constitutive for a discretization framework. From a bird's eye perspective, these principles are:

- localizable functions
- local operators, local functionals and local evaluations
- direct access to building blocks
- abstract interfaces and generic algorithms

We discuss those in the following paragraphs and touch on some aspects of a possible implementation of such a discretization framework.

Localizable functions. Perhaps the most basic principle is the concept of localizable functions, as expressed in Definition 3.1.4. Since all data functions are required to be localizable and all discrete functions are localizable as well, we can formulate operators, functionals, products and norms purely in terms of localizable functions. This yields generic constructs that can be applied in a large variety circumstances: for instance, the discrete elliptic operator B_h from Definition 3.1.14 can be used either to be assembled into a system matrix, given any discrete function space, or to be assembled into an energy or H^1-semi product matrix, given any discrete function space, or to compute the energy or H^1-semi product of two arbitrary localizable functions. This is possible since all localizable functions yield local functions in the sense of Definition 3.1.2, which can be evaluated on the reference elements of the grid. Thus, all operators, functionals and products can be realized by local counterparts (see the next paragraph).

As a consequence, all discrete function spaces yield local bases as well (the elements of which are local functions, see Definition 3.1.16). (Given the interpretation of a local basis function as the local function of a localizable global basis function, those do in particular include all necessary transformations, for instance in the context of Raviart-Thomas-Nédélec spaces.) We thus only require a single implementation of a discrete function that works with arbitrary discrete function spaces.

Local operators, local functionals and local evaluations. All operators, two-forms and functionals can be localized with respect to the grid and integrals can be transformed to the respective reference element. We treat the resulting local integrals by modeling the integrand as a local evaluation and applying generic local operators and functionals to approximate the integral by a numerical quadrature. This splitting allows for a reuse of local evaluations in different circumstances (see the local SWIPDG evaluations from Examples 3.1.23, 3.1.25 and 3.1.27 that are used in the context of the SWIPDG discretization as well as in the context of a posteriori error estimation) yielding powerful and versatile constructs. For instance, a combination of the local product evaluation (Example 3.1.30) with the generic local volume operator (Definition 3.1.9) yields a local L^2-volume operator. Since the local operators act on local functions only, they can be used to form localizable operators and products (acting on localizable functions) as well as assembled operators and products (acting on the DoF vectors of discrete functions). The very same local L^2 operator, for instance, can be used to form several L^2 products (Definition 3.1.32 and Algorithm 3.1.34) or L^2 projection operators (Definitions 3.1.40 and 3.1.41).

The above separation of duties leads to simple objects with a distinct purpose, which has several benefits regarding a possible implementation. The sole purpose of a local volume operator, for instance, is to approximate a local volume integral by a numerical quadrature (without bothering with the form of the integrand). On the one hand, the resulting implementation can thus be easily understood. On the other hand, this leads to less code duplication and easier to track down errors, since there need only be two objects implementing a volume quadrature in the whole discretization framework (namely the local volume operator and the local volume functional). While implementing the use of a quadrature may seem like a simple example, a safe and well performing implementation still needs to obtain and check the polynomial degree of the integral, try to obtain a suitable quadrature and handle the situation where such a quadrature is not available.

Since it is good practice not to duplicate such code and effort, a possible implementation should follow the proposed separation of local operators, functionals and evaluations.

Direct access to building blocks. When designing new discretization schemes it is important for the developer to have direct access to the underlying building blocks, such as the DoF map ι (Definition 3.1.16), the properties of the local Finite Element (such as Lagrange points or the interpretation of the local basis of a Raviart-Thomas-Nédélec space, see Definitions 3.1.19 and 3.1.35), or the containers constituting discrete functions and assembled operators and functionals. If all underlying parts are individually accessible and exposed, they can be used and combined in new ways that not even the original authors may have thought of. Such a library of reusable and flexible building blocks is the basis for an easy development of new discretization schemes and error estimators.

Abstract interfaces and generic algorithms. All mathematical concepts should be represented by abstract interfaces, in particular matrices and vectors, operators and functionals, discrete function spaces, discrete functions and localizable functions and local functions. Such interfaces are the only reliable way to allow for generic implementations (as mentioned above). Discrete functions can be implemented generically, given any discrete function space and vector; localizable operators and functionals can be implemented generically given any localizable functions; assemblable operators and functionals can be implemented generically given any discrete function space; and so forth...

A considerable amount of time and effort has to be invested into an implementation of a discretization framework, in particular regarding maintenance. In the long run, this effort can only be kept at bay through the use of abstract interfaces and generic algorithms, keeping code duplication at a minimum.

In addition, for instance in the context of model reduction, we require access to at least the vectors, operators and discretizations of a discretization framework (compare [MRS2015, Section 3.3]). When used in conjunction with an external model reduction library, this access can easily be granted (and implemented) in terms of the abstract interfaces. Any implementation of these interfaces then automatically benefits from the exposure of the interfaces and can be used in novel ways, far beyond the traditional scope of discretization frameworks.

While one might regard abstract interfaces and the required direct access to building blocks and underlying structures as opposite philosophies, we consider both beneficial and necessary (in particular if access to the underlying structures can be provided in a generic way).

In addition to these design principles there are other (more technical) aspects which we require of a discretizations framework:

- We require a means to model and identify different parts of the boundary of the domain (such as Γ_D and Γ_N) in a well defined way to allow for generic implementations of local boundary operators and functionals.
- We require a well defined way to create, access and combine (dense and sparse) matrices and vectors to allow the use of different existing linear algebra backends. We also require linear solvers which should be exchangeable at runtime.
- Since iterating over the grid may be costly we want to be able to execute as many local operations on one element of the grid as desired. This allows to assemble several containers or apply products in one iteration over the grid.
- In order to fully use all available resources it should be possible to execute most operations in parallel, in particular the assembly of containers and the inversion of system matrices. We do not want to enforce a specific parallelization paradigm, allowing in particular for shared memory and distributed approaches. All objects representing mathematical concepts (such as local evaluations, operators, discrete functions, containers) should be usable without explicit knowledge of the parallelization involved; a user of the discretization framework should be able to focus mainly on the mathematical aspects of discretization schemes.
- It is clear from the above discussion that we aim for a system language (such as C, Fortran or C++) for performance critical parts of the implementation and a modern object oriented language (such as a recent version of C++ or `Python` in conjunction with `NumPy` [Oli2007] and `SciPy` [JOPo2001] for all parts that are visible to the user.

  In case of a statically typed language, such as C++, we aim for a balance between static and dynamic polymorphism: while performance critical parts (such as the type of the grid, field types, dimensions, local evaluations) should be fixed at compile time, it should be possible to switch between data functions (of the same dimension) at runtime.

3.1.3 Existing implementations

Needless to say, we are interested in an open source and freely available software framework with a strong scientific background in the developer- and user-base. There exist several such frameworks:

deal.ii [BHH+2015], which is available at `https://www.dealii.org/`, "is a C++ program library targeted at the computational solution of partial differential equations using adaptive finite elements"[11].

DUNE [BBD+2008, BBD+2008a], which is available at `http://www.dune-project.org/` is a "modular toolbox for solving partial differential equations (PDEs) with grid-based methods"[12].

[11]`https://www.dealii.org/about.html`, 22.07.2015
[12]`http://www.dune-project.org/dune.html`, 22.07.2015

Feel++ [PCD+2012], which is available at `http://www.feelpp.org/`, “is a C++ library for partial differential equation solves using generalized Galerkin methods such as the finite element method, the h/p finite element method, the spectral element method or the reduced basis method.”[13].

FreeFem++ [Hec2012], which is available at `http://www.freefem.org/ff++/index.htm`, “is a partial differential equation solver”[14].

The FEniCS Project [LMW2012], which is available at `http://fenicsproject.org/` “is a collection of free software with an extensive list of features for automated, efficient solution of differential equations”[15].

libMesh [KPSC2006], which is available at `http://libmesh.github.io/`, “provides a framework for the numerical simulation of partial differential equations using arbitrary unstructured discretizations on serial and parallel platforms”[16].

Most of these frameworks allow for adaptive mesh refinement, different parallelization paradigms and the use of external libraries. They vary in their provided features, aims and philosophies, user friendliness and target audience.

We chose to realize our discretization framework within the context of the **DUNE** project: the *Distributed and Unified Numerics Environment* [BBD+2008, BBD+2008a], written in C++. **DUNE** is free and open source software, its philosophy aligns well with the abstract design principles identified in Section 3.1.2, it yields highly efficient programs, is fairly well documented and has a large developer and user base with a strong background in numerical analysis and scientific computing. However, **DUNE** is mainly targeted at researchers and thus has a steep learning curve.

DUNE has a modular structure, with the core modules being used and developed by all research groups involved:

`dune-common` contains abstract parallelization helpers, dense vectors modeling coordinates and many abstractions of advanced C++ features to yield portable code which works with most compilers of the last decade.

`dune-geometry` contains generic reference elements, embeddings and quadratures.

`dune-grid` contains the abstract definition of a grid and is probably the most widely used module. It also contains some reference implementations of the abstract grid interface and wrapper code to allow the use of external grid managers. (Note that a grid in **DUNE** includes all elements on all refinement levels. The concept of a grid τ_h, as used throughout this work, is modeled by a `GridView`.)

`dune-istl` provides an iterative solver template library with generic sparse matrices, vectors and linear solvers.

[13]`http://www.feelpp.org/`, 22.07.2015
[14]`http://www.freefem.org/ff++/index.htm`, 22.07.2015
[15]`http://fenicsproject.org/`, 22.07.2015
[16]`http://libmesh.github.io/index.html`, 22.07.2015

`dune-localfunctions` contains shape functions defined on the reference elements together with interpolation operators which form the basis of a local Finite Element. (Note that these shape functions do not coincide with the notion of local functions from Definition 3.1.2.)

On top of these core modules, there exist two main discretization modules, which are independently developed by different research groups: `dune-fem`[17] [DKNO2010] and `dune-pdelab`[18], each with different philosophies, design approaches and implementations.

`dune-fem` is formulated in terms of discrete function spaces and discrete operators. It has a long history for adaptive grid refinement and distributed parallel computations.

`dune-pdelab` is based on the residual formulation of PDEs. Is supports different linear algebra backends and arbitrary function spaces based on local Finite Elements.

Being the younger project of the two, `dune-pdelab` uses more modern programming techniques and is developed more openly than `dune-fem`. Its residual formulation and automatic differentiation make it easy to solve systems of complicated PDEs. From the perspective of a developer of new numerical schemes, however, it does not allow for an easy access to the underlying building blocks. In addition, the residual formulation is not suitable for the context of model reduction, where we require access to individual operators and functionals.

While `dune-fem` supports the notion of a discrete operator it lacks support for functionals. In addition, its monolithic design hinders shared memory parallel implementations due to many global singletons.

Both frameworks do not support the notion of a localizable function and a discrete function space as proposed in Section 3.1.1.1 and neither of the two supports the separation of integration and integrand. We could have added some of the required functionality to either of these frameworks. However, due to our requirements (see the previous sections), this would have only been possible to a certain degree. Each framework would bring severe restrictions regarding the required flexibility and its use in the context of model reduction.

3.1.4 A new discretization framework

There was no suitable discretization framework available when we started to work on efficient and reliable discretizations in the context of parametric multiscale problems in 2011. We thus provide our own implementation of such a framework within the context of the **DUNE** project, based on the mathematical foundation and design principles discussed in Sections 3.1.1 and 3.1.2.

[17]`http://dune.mathematik.uni-freiburg.de/`
[18]`http://www.dune-project.org/pdelab/`

Following the modular structure of **DUNE**, we provide several modules which implement different aspects of our discretization framework (we discuss each in detail in the following sections):

> `dune-stuff` provides extensions of the core modules `dune-common` and `dune-grid`, which are mostly centered around improved usability and generic algorithms, together with important build and test infrastructures. In particular it contains the concept of a `GridProvider` (which encapsulates a grid and provides an abstraction of `GridParts` and `GridViews`) and the concept of a `BoundaryInfo` (which provides a standardized way to identify different parts of the boundary). In addition, `dune-stuff` contains the abstract interface modeling localizable functions and local functions (in the sense of Definitions 3.1.4 and 3.1.2) and various ready to use implementations. Lastly, it provides an abstraction for linear algebra containers and linear solvers along with several implementations supporting different parallelization paradigms (for instance based on `dune-istl` or `eigen`[19] [GJo2010]).
>
> `dune-gdt`, the *generic discretization toolbox*, is the heart of our discretization framework. It provides abstract interfaces for local evaluations, local operators and functionals, discrete function spaces, operators, products and functionals and so forth, in the spirit of Section 3.1.1. In addition, it contains many ready to use implementations, problem definitions and discretizations. It is currently mainly centered around linear elliptic and hyperbolic problems.

The remainder of this chapter gives an in-depth discussion of these modules which together make up our implementation of a discretization framework. We provide example code snippets along the way that are not necessarily meant to be directly usable (for instance, we shall omit `int main(...)` in C++ code). In addition, we provide reference to code locations, where "`dune/foo/bar.hh`" denotes the location of a file within the `dune-foo` module. However, we do not provide individual references for elements of the standard C++ library, which are prefixed by `std::`, and refer to `http://en.cppreference.com/`.

3.1.4.1 `dune-stuff`

The **DUNE** module `dune-stuff` is open source software and freely available on GitHub: `https://github.com/wwu-numerik/dune-stuff`. It is mainly developed by R. Milk and F. Schindler with contributions from A. Buhr, S. Girke, S. Kaulmann, T. Leibner, B. Verfürth and K. Weber. As mentioned above, `dune-stuff` contains helpful infrastructure, extensions of `dune-common` and `dune-grid` and the definition and implementation of localizable functions and linear algebra containers and solvers. Accordingly, it contains the following submodules, modeled as namespaces below the `Dune::Stuff` namespace: `Common`, `Functions`, `Grid` and `LA`. We discuss relevant content of these submodules in the following paragraphs.

[19]`http://eigen.tuxfamily.org/`

Improved handling of dense containers in `Dune::Stuff::Common`. `dune-common` contains the `FieldVector`[20] and `FieldMatrix`[21] classes, which model small dense vectors and matrices of fixed size (in particular used for coordinates, affine reference maps and function evaluations). These containers are implemented to provide maximum performance, but lack some convenience features. In particular the lack of certain operators and contructors make it difficult to write generic code. Consider, for instance, a generic string conversion utility (assuming `T` is a matrix type):

```
1  template< class T >
2  static inline T fromString(const std::string ss,
3                             const size_t rows = 0,
4                             const size_t cols = 0)
5  {
6    T result(rows, cols); // <- does not compile for FieldMatrix
7    // fill result from ss
8    // ...
9    return result;
10 }
```

The above example does compile if `T` is a `DynamicMatrix`[22], but not if `T` is a `FieldMatrix`, which makes it extremely difficult to write generic code.[23]

In order to allow for generic algorithms we provide templated `VectorAbstraction` and `MatrixAbstraction` classes in `dune/stuff/common/{matrix,vector}.hh` (along with specializations for all sensible vector and matrix classes), which allow for generic creation of and access to matrices and vectors. Additionally, we provide `is_vector` and `is_matrix` traits, which allow to rewrite the above example:

```
1  #include <dune/stuff/common/matrix.hh>
2
3  using namespace Dune::Stuff::Common;
4
5  template< class T >
6      static inline typename std::enable_if< is_matrix< T >::value, T >::value
7  fromString(const std::string ss,
8             const size_t rows = 0,
9             const size_t cols = 0)
10 {
11   auto result = MatrixAbstraction< T >::create(rows, cols);
12   // fill result from ss using MatrixAbstraction< T >::set_entry(...)
13   // ...
14   return result;
15 }
```

It is thus possible to use `fromString` with matrices of different type:

[20] `dune/common/fvector.hh`

[21] `dune/common/fmatrix.hh`

[22] `dune/common/dynmatrix.hh`

[23] While it is clear that constructing a `FieldMatrix` of fixed size `M`×`N` is not sensible for other values of `rows` and `cols`, the construction in line 6 should be possible for `rows = M` and `cols = N` (throwing an appropriate exception otherwise).

```
#include <dune/common/fmatrix.hh>
#include <dune/common/dynmatrix.hh>

auto fmat = fromString< Dune::FieldMatrix< double, 2, 2 > >("[1. 2.; 3. 4.]");
auto dmat = fromString< Dune::DynamicMatrix< double >      >("[1. 2.; 3. 4.]");
```

In particular, one can use it with any custom matrix implementation by providing a specialization of `MatrixAbstraction` within the user code:

```
template< class FieldType >
class CustomMatrix { /* implement custom matrix */ };

template< class FieldType>
struct MatrixAbstraction< CustomMatrix< FieldType > > { /* implement specialization */ };

auto cmat = fromString< CustomMatrix< FieldType > >("[1. 2.; 3. 4.]");
```

Based on these abstractions we provide many generic implementations in `dune/stuff/common/`. For instance, we provide an extension of the `FloatCmp`[24] mechanism from `dune-common` for any combination of vectors (which allows for the same syntax as its counterpart in `dune-common`, including compare styles and tolerances):

```
#include <dune/common/dynvector.hh>
#include <dune/stuff/common/float_cmp.hh>

std::vector< double >         svector({1., 1.});
Dune::DynamicVector< double > dvector(2, 1.);

Dune::Stuff::Common::FloatCmp::eq(svector, dvector);
```

Improved string handling and `Configuration` in `Dune::Stuff::Common`. As already hinted at in the previous paragraph, we provide the string conversion utilities

```
template< class T >
static inline T fromString(const std::string ss,
                           const size_t size = 0, const size_t cols = 0);

template< class T >
static inline std::string toString(const T& ss);
```

in `dune/stuff/common/string.hh`. These can be used with any basic type as well as with all matrices and vectors supported by the abstractions from the previous paragraph. We use standard notation for vectors (`"[1 2]"`) and matrices (`"[1 2; 3 4]"`), see the previous paragraph for examples. The `fromString` function takes optional arguments which determine the size of the resulting container (for containers of dynamic size), where 0 means automatic detection.

Based on these string conversion utilities, we provide an extension of `dune-common`'s `ParameterTree`[25] in `dune/stuff/common/configuration.hh`: the `Configuration` class.

[24] dune/common/float_cmp.hh
[25] dune/common/parametertree.hh

The `Configuration` is derived from `ParameterTree` and can thus be used in all places where a `ParameterTree` is expected. While it also adds an additional layer of checks (in particular regarding provided defaults), report and serialization facilities, one of its main features is to allow the user to extract any type that is supported by the string conversion facilities. Given a sample configuration file in `.ini` format (for example the default configuration of the `Cube` grid provider discussed further below),

Listing 1 Contents of the `default_config()` of `Stuff::Grid::Providers::Cube`.

```
lower_left       = [0 0 0 0]
upper_right      = [1 1 1 1]
num_elements     = [8 8 8 8]
num_refinements  = 0
overlap          = 1
```

we can query the resulting `Configuration` object `config` for the types supported by the `ParameterTree`,

```
auto num_refinements = config.get< int >("num_refinements");
```

as well as for all types supported by our string conversion utilities (including custom matrix and vector types as explained in the previous paragraph):

```
auto lower_left = config.get< FieldVector< double, dimDomain > >("lower_left");
```

The above is valid for all $0 \leq$ `dimDomain` ≤ 4, due to the automatic size detection of `fromString`.

Identification of domain boundaries in `Dune::Stuff::Grid`. As noted in Section 3.1.2 we require a generic way to identify parts of the boundary of the computational domain, such as those associated with Dirichlet- or Neumann boundary values. Unfortunately, `dune-grid` does not provide such a mechanism. In `dune/stuff/grid/boundaryinfo.hh`, we thus provide a virtual interface,

```
template< class IntersectionType >
class BoundaryInfoInterface
{
  virtual bool has_dirichlet() const;
  virtual bool has_neumann() const;

  virtual bool dirichlet(const IntersectionType& intersection) const = 0;
  virtual bool neumann(const IntersectionType& intersection) const = 0;
};
```

based on which we can provide generic algorithms, which act only on parts of the domain boundary (such as the Dirichlet projection from Definition 3.1.20). We also provide the following implementations of the `BoundaryInfoInterface` within the `BoundaryInfos` namespace:

`AllDirichlet` and `AllNeumann`, the purpose of which is self-explanatory.

`IdBased`, using the now-deprecated `boundaryId()` method of an `intersection` (given key-value pairs such as `{"dirichlet", {1, 2}}, {"neumann", {3, 4}}`, mapping the reported boundary ids to the respective boundary type).

`NormalBased`, which allows to identify boundary intersections by the direction of their outward pointing normal.

In order to allow for problem definition classes to define domain boundaries independently of the type of the grid, we also provide the `BoundaryInfoProvider` factory. Classes can hold a complete description of one of the boundary infos above in a `Configuration`. Given the type of the grid (and thus the type of an intersection), one is then able to create an instance of one of the implementations of the `BoundaryInfoInterface` of correct type, as required:

```
#include <dune/grid/yaspgrid.hh>
#include <dune/grid/sgrid.hh>
#include <dune/stuff/common/configuration.hh>
#include <dune/stuff/grid/boundaryinfo.hh>

using namespace Dune::Stuff::Common;
using namespace Dune::Stuff::Grid;

class Problem
{
public:
  Configuration boundary_info_cfg()
  {
    Configuration config;
    config["type"]      = "stuff.grid.boundaryinfo.normalbased";
    config["default"]   = "dirichlet";
    config["neumann.0"] = "[ 1. 0.]";
    config["neumann.1"] = "[-1. 0.]";
    return config;
  }
}; // class Problem

typedef typename Dune::YaspGrid< 2 >::LeafIntersection YI;
typedef typename Dune::SGrid< 2, 2 >::LeafIntersection SI;

Problem problem;
auto boundary_info_y = BoundaryInfoProvider< YI >::create(problem.boundary_info_cfg());
auto boundary_info_s = BoundaryInfoProvider< SI >::create(problem.boundary_info_cfg());
```

The type of `boundary_info_y`, for instance, is `std::unique_ptr< BoundaryInterface< YI > >`. Given a rectangular domain in $\mathbb{R}^2$, it models a Neumann boundary left and right and a Dirichlet boundary everywhere else.

Walking the grid in `Dune::Stuff::Grid`. Since we are considering grid-based numerical methods, we frequently need to iterate over the elements $t \in \tau_h$ of a grid (compare

Algorithms 3.1.18 and 3.1.29). In order to minimize the amount of required grid iterations, we want to be able to carry out several operations on each grid element (instead of several iterations over the grid). We thus provide in `dune/stuff/grid/walker.hh` the templated `Walker` class, working with any `GridView` (from `dune-grid`) or `GridPart` (from `dune-fem`):[26]

```cpp
template< class GridViewType >
class Walker
{
  // not all methods and types shown ...
public:
  void add(Functor::Codim0< GridViewType >& functor/*, ... */);
  void add(Functor::Codim1< GridViewType >& functor/*, ... */);
  void add(Functor::Codim0And1< GridViewType >& functor/*, ... */);

  void walk(const bool use_tbb = false);
};
```

The user can add an arbitrary amount of functors to the `Walker`, all of which are then locally executed on each grid element. Each functor is derived from one of the virtual interfaces `Functor::Codim0`, `Functor::Codim1` or `Functor::Codim0And1`, for instance

```cpp
template< class GridViewType >
class Codim0
{
public:
  typedef typename Stuff::Grid::Entity< GridViewType >::Type EntityType;

  virtual void prepare() {}
  virtual void apply_local(const EntityType& entity) = 0; // called for all t ∈ τ_h
  virtual void finalize() {}
};
```

where `prepare` (and `finalize`) are called before (and after) iterating over the grid, while `apply_local` is called on each element of the grid. Each `add` method of the `Walker` accepts an additional argument which allows to select the elements and faces, the functor will be applied on. For instance, in the context of the SWIPDG discretization we want to apply the local coupling operators from Example 3.1.27 on all inner faces of the grid and the local boundary operators from Example 3.1.25 on all Dirichlet faces of the grid (compare Section 3.1.1.1). Presuming we were given suitable implementations of these local operators as functors and a `BoundaryInfo` object in the sense of the previous paragraph, the following would realize just that:

```cpp
#include <dune/stuff/grid/walker.hh>

using namespace Dune::Stuff::Grid;

Walker< GV > walker(grid_view);
```

[26]We provide traits in `dune/stuff/grid/{entity,intersection}.hh` to extract required information from a `GridView` or `GridPart` in a generic way (see the `Codim0` functor example).

```cpp
walker.add(coupling_operator, new ApplyOn::InnerIntersectionsPrimally< GV >());
walker.add(boundary_operator, new ApplyOn::DirichletIntersections< GV >(boundary_info));
// add more, if required...
walker.walk()
```

Note that the `walk` method allows to switch between a serial and a shared memory parallel iteration over the grid, at runtime (via the `use_tbb` switch). In particular, the user only has to provide implementations of the functors and need not deal with any parallelization issues (or different types of grid walkers, depending on the parallelization paradigm).

Providing generic access to `GridViews` and `GridParts` in `Dune::Stuff::Grid`. There exists an unfortunate disagreement between `dune-grid` and `dune-fem`, whether `GridViews` or `GridParts` are to be used to model a collection of grid elements, which makes it hard to implement generic algorithms. Think of some code which needs to create a `LevelGridView` or `LevelGridPart`, depending on the further use for a discrete function space implemented via `dune-pdelab` or `dune-fem`:

```cpp
#include <dune/fem/gridpart/levelgridpart.hh>

template< class GridType >
void create_level(GridType& grid, const int lv) {
  // either
  auto level_view = grid.levelGridView(lv);
  // or
  Dune::Fem::LevelGridPart< GridType > level_part(grid, lv);
  // ...
}
```

To allow for generic code despite this disagreement we provide the `ProviderInterface` in `dune/stuff/grid/provider.hh`, the purpose of which is to be passed around instead of a grid:

```cpp
template< class GridType >
class ProviderInterface
{
  // not all methods and types shown ...
public:
  virtual GridType& grid() = 0;

  template< ChooseLayer layer_type, ChoosePartView part_view_type >
  typename Layer< layer_type, part_view_type >::Type layer(const int lv = 0);

  template< ChoosePartView type >
  typename Level< type >::Type                       level(const int lv);

  template< ChoosePartView type >
  typename Leaf< type >::Type                        leaf();
};
```

Together with the `ChooseLayer` and `ChoosePartView` enum classes[27], this allows to write generic code by passing a grid provider along with the required tag (this will be of significant importance in the context of `dune-gdt` further below):

```cpp
#include <dune/stuff/grid/provider.hh>

using namespace Dune::Stuff::Grid;

template< class GridType, ChoosePartView type >
void create_level(ProviderInterface< GridType >& grid_provider, const int lv) {
  auto level_part_or_view = grid_provider.level< type >(lv);
  // ...
}
```

We provide several implementations of the `ProviderInterface` (for instance `Providers::Default`, wrapping an existing grid and `Providers::Cube`, which creates grids of rectangular domains). In addition, we provide a means to select a grid provider at runtime using the `GridProviders` struct, given the type of the grid `G` and a configuration (for instance using the default one of the `Cube` provider, see Listing 3.1.4.1),

```cpp
#include <dune/stuff/grid/provider.hh>

auto grid_provider = Dune::Stuff::GridProviders< G >::create("stuff.grid.provider.cube",
                                                             config);
```

which is particularly useful in conjunction with configuration files or `Python` bindings.

Local functions and localizable functions in `Dune::Stuff::Functions`. As discussed in Section 3.1.2, the concept of a local function in the sense of Definition 3.1.2 is crucial for a discretization framework (not to be confused with shape functions from `dune-localfunctions`). Given a grid element $t \in \tau_h$, we can evaluate a (scalar-, vector- or matrix-valued) local function $\varphi^t : \hat{t} \to \mathbb{R}^{r \times c}$, for $r, c \in \mathbb{N}$, on the reference element $\hat{t}$ associated with t. To model such functions (as well as a set of local basis functions) we provide the `LocalfunctionSetInterface` in `dune/stuff/functions/interfaces.hh`:

```cpp
template< class EntityType,
          class DomainFieldType, size_t dimDomain,
          class RangeFieldType,  size_t dimRange, size_t dimRangeCols = 1 >
class LocalfunctionSetInterface
{
  // not all methods and types show ...
public:
  virtual const EntityType& entity() const

  virtual size_t size() const = 0;
  virtual size_t order() const = 0;

  virtual void evaluate(const DomainType& xx, std::vector< RangeType >& ret const = 0;
  virtual void jacobian(const DomainType& xx,
                        std::vector< JacobianRangeType >& ret) const = 0;
};
```

[27] `dune/stuff/grid/layers.hh`

The template parameter `EntityType` models the type of the grid element $t \in \tau_h$, the parameters `DomainFieldType` and `dimDomain` model $\mathbb{R}^d \supset \tau_h$ while `RangeFieldType`, `dimRange` and `dimRangeCols` model $\mathbb{R}^{r \times c}$. The resulting `DomainType` is a `FieldVector` from `dune-common`, while `RangeType` and `JacobianRangeType` are composed of `FieldVector` and `FieldMatrix`, depending on the dimensions. As a shorthand, we often write

```
template< class E, class D, size_t d, class R, size_t r, size_t rC >
```

for these arguments. Each set of local functions has to report its polynomial order and size. The methods `evaluate` and `jacobian` expect vectors of size `size()` for `ret`.

While local bases of discrete function spaces will be realized as implementations of `LocalfunctionSetInterface` (see below), we also provide an interface for individual local functions, which is used by data functions and local functions of discrete functions:

```
template< class E, class D, size_t d, class R, size_t r, size_t rC >
class LocalfunctionInterface
  : public LocalfunctionSetInterface< E, D, d, R, r, rC >
{
  // not all methods and types shown ...
public:
  virtual void evaluate(const DomainType& xx, RangeType& ret) const = 0;
  virtual void jacobian(const DomainType& xx, JacobianRangeType& ret) const = 0;

  virtual size_t size() const override final
  {
    return 1;
  }
}
```

All local evaluations, operators and functionals in `dune-gdt` are implemented using `LocalfunctionSetInterface` and thus work for local bases, local functions and local discrete functions at the same time.

Alongside, we also provide the following interface for localizable functions according to Definition 3.1.4, which merely work as containers of localizable functions:

```
template< class E, class D, size_t d, class R, size_t r, size_t rC >
class LocalizableFunctionInterface
{
  // not all methods and types shown ...
public:
  virtual std::string name() const;

      virtual std::unique_ptr< LocalfunctionInterface< E, D, d, R, r, rC > >
  local_function(const EntityType& entity) const = 0;
};
```

Based on this interface we provide visualization and convenience operators, allowing for

```
(p - p_h).visualize(grid_view, "difference");
```

where `p` might denote a localizable data function and `p_h` might denote a discrete function (see further below), if both are localizable with respect to the same `grid_view`. Expressions such as `p - p_h`, `p + p_h` or `p*p_h` yield localizable functions via generically implemented local functions (if the dimensions allow it).

We also provide numerous implementations of `LocalizableFunctionInterface` in `dune-stuff`, most of which can also be created using the `FunctionsProvider`[28] factory class, given a `Configuration`:

- `Functions::Checkerboard` in `dune/stuff/functions/checkerboard.hh` models a piecewise constant function, the values of which are associated with an equidistant regular partition of a domain. Sample configuration for a function $\mathbb{R}^2 \to \mathbb{R}$:

  ```
  lower_left   = [0. 0.]
  upper_right  = [1. 1.]
  num_elements = [2 2]
  values       = [1. 2. 3. 4.]
  ```

- `Functions::Constant` in `dune/stuff/functions/constant.hh` models a constant function. Sample configuration for a function $\mathbb{R}^d \to \mathbb{R}^{2\times 2}$, for any $d \in \mathbb{N}$, mapping to the unit matrix in $\mathbb{R}^2$:

  ```
  value = [1. 0.; 0. 1.]
  ```

- `Functions::Expression` in `dune/stuff/functions/expression.hh` models continuous functions, given an expression and order at runtime (expressions for gradients can be optionally provided). Sample configuration for $f : \mathbb{R}^2 \to \mathbb{R}^2$ given by $(x, y) \mapsto \big(x, \sin(y)\big)$:

  ```
  variable   = x
  order      = 3
  expression = [x[0] sin(x[1])]
  gradient.0 = [1 0]
  gradient.1 = [0 cos(x[1])]
  ```

  Note that the user has to provide the approximation order, resulting in f being locally approximated as a third order polynomial on each grid element.

- `GlobalLambdaFunction` in `dune/stuff/functions/global.hh` models continuous functions by evaluating a C++ lambda expression. Sample usage for $f : \mathbb{R}^2 \to \mathbb{R}$ given by $(x, y) \mapsto x$:

  ```
  GlobalLambdaFunction< E, double, 2, double, 1 > f([](DomainType x){ return x[0]; },
                                                     1); // <- local polynomial order
  ```

- `Functions::Spe10::Model1` in `dune/stuff/functions/spe10.hh` models the permeability field of the SPE10 model1 test case, given the appropriate data file[29].

[28] `dune/stuff/functions.hh`
[29] Available at `http://www.spe.org/web/csp/datasets/set01.htm`

Generic linear algebra containers in `Dune::Stuff::LA`. We discussed the use of small dense vectors and matrices in the context of coordinates and function evaluations above. In addition we require large vectors and matrices (usually sparse) to represent assembled operators and functionals, compare Section 3.1.2. Linear algebra containers are a performance critical aspect of the discretization framework and we do not want to bet on a single horse: there exists external backends which are well suited for serial and shared memory parallel computations (such as `eigen`) while others are more suited for distributed memory parallel computations (such as `dune-istl`). Neither is fitting for every purpose and we thus require a means to exchange the implementation of matrices and vectors depending on the circumstances. This calls for abstract interfaces for containers, which we provide within the `Dune::Stuff::LA` namespace.

We chose a combination of static and dynamic inheritance for these interfaces, allowing for virtual function calls that act on the whole container (such as operators) while using the "Curiously recurring template patterns" (`CRTP`, see [Cop1995]) paradigm for methods that are called frequently (such as access to individual elements in loops), allowing the compiler to optimize performance critical calls (for instance by inlining). We provide a thread-safe helper class `CRTPInterface` in `dune/stuff/common/crtp.hh` along with thread-safe `CHECK_...` macros for debugging (since the tools provided in `dune/common/bartonnackmanifcheck.hh` do not work properly in a shared memory parallel program).

All matrices and vectors are derived from `ContainerInterface` in `dune/stuff/la/container/container-interface.hh`:

```cpp
template< class Traits, class ScalarType = typename Traits::ScalarType >
class ContainerInterface
  : public CRTPInterface< ContainerInterface< Traits, ScalarType >, Traits >
{
  // not all methods and types shown ...
public:
  typedef typename Traits::derived_type derived_type;
                                                     // Sample CRTP implementation:
  inline void scal(const ScalarType& alpha)          // as_imp() from CRTPInterface
  {                                                  // performs a static_cast into
    CHECK_AND_CALL_CRTP(this->as_imp().scal(alpha)); // derived_type. Thus, scal() of
  }                                                  // the derived class is called.

  inline void axpy(const ScalarType& alpha, const derived_type& xx);
  inline derived_type copy() const;

  virtual derived_type& operator*=(const ScalarType& alpha) // Default implementation:
  {                                                         // could be overriden by
    scal(alpha);                                            // any derived class.
    return this->as_imp();
  }
};
```

The `ContainerInterface` enforces just enough functionality to assemble a linear combination of matrices or vectors, which we require for instance in the context of model reduction. Given an affine decomposition of a parametric matrix or vector B (compare

Definition 1.3.7), we need to assemble $B(\boldsymbol{\mu}) := \sum_{q=0}^{Q-1} \theta_q(\boldsymbol{\mu})B_q$ for given components B_q and coefficients $\theta_q(\boldsymbol{\mu})$. The following generic code will work for any matrix or vector type `C` derived from `ContainerInterface`:

```
 1 #include <dune/stuff/la/container/container-interface.hh>
 2
 3 using namespace Dune::Stuff::LA;
 4
 5 template< class C >
 6     typename std::enable_if< is_container< C >::value, C >::type
 7 assemble_lincomb(const std::vector< C >& components,
 8                  const std::vector< double >& coefficients)
 9 {
10   auto result = components[0].copy();
11   result *= coefficients[0];
12   for (size_t qq = 1; qq < components.size(); ++qq)
13     result.axpy(coefficients[qq], components[qq]);
14   return result;
15 }
```

Note that since we implement copy-on-write and move semantics for all matrices and vectors in `dune-stuff`, the only deep copy is actually done in line 11 (neither in line 10 nor in line 14).[30]

Based on `ContainerInterface` we provide the `VectorInterface`[31] for dense vectors and the `MatrixInterface`[32] for dense and sparse matrices. Each derived vector class has to implement the methods `size`, `add_to_entry`, `set_entry` and `get_entry_ref`, which allow to access and change individual entries of the vector. The interface provides default implementations for all relevant mathematical operators, support for range-based for loops and many useful methods, such as `dot`, `mean`, `standard_deviation` and `l2_norm`, just to name a few.

Each derived matrix class has to implement `rows`, `cols`, `add_to_entry`, `set_entry` and `get_entry` to allow for access to individual entries, `mv` for matrix/vector multiplication and `clear_row`, `clear_col`, `unit_row` and `unit_col`, which are required in the context of Dirichlet Constraints and pure Neumann problems. Every matrix implementation (even dense ones) is constructible from a sparsity pattern (which we provide in `dune/stuff/la/container/pattern.hh`) and the access methods `..._entry` are only required to work on entries that are contained in the pattern. The interface provides several mathematical operators, norms and a means to obtained a pruned matrix (where all entries close to zero are removed from the pattern).

We also provide several vector and matrix implementations:

- The `CommonDenseVector` and `CommonDenseMatrix` in `dune/stuff/la/container/`

[30] Note also the use of the `is_container` traits in line 6 that we provide in `dune/stuff/la/container/container-interface.hh`. Together with `enable_if`, this checks that `C` is derived from `Container-Interface` at compile time. We will see further below how to deal with `CRTP` interfaces in a much nicer way, if the situation allows for it (e.g., if the argument is not wrapped inside a vector).

[31] `dune/stuff/la/container/vector-interface.hh`

[32] `dune/stuff/la/container/matrix-interface.hh`

common.hh, based on the DynamicVector and DynamicMatrix from dune-common. These are always available.

- The EigenDenseVector, EigenMappedDenseVector, EigenDenseMatrix and EigenRowMajorSparseMatrix in dune/stuff/la/container/eigen.hh, based on the eigen package (if available). The EigenMappedDenseVector allows to wrap an existing double* array and the EigenRowMajorSparseMatrix allows to wrap existing matrices in standard CSR format, which allows to wrap other container (for instance in the context of Python bindings).

- The IstlDenseVector and IstlRowMajorSparseMatrix in dune/stuff/la/container/istl.hh, based on dune-istl (if available).

To allow for generic algorithms we also provide the LA::ChooseBackend enum class along with the default_backend, default_sparse_backend and default_dense_backend defines (which are set depending on the build configuration). For instance, these could be used together with the LA::Container traits to implement a local L^2 projection (see Definition 3.1.40), given a local basis and an appropriate quadrature:

```cpp
#include <dune/stuff/la/container.hh>

using namespace Dune::Stuff::LA;

typedef typename Container< double, default_dense_backend >::MatrixType LocalMatrixType;
typedef typename Container< double, default_dense_backend >::VectorType LocalVectorType;

// ... on each grid element
LocalMatrixType local_matrix(local_basis.size(), local_basis.size(), 0.);
LocalVectorType local_vector(local_basis.size(), 0.);
LocalVectorType local_DoFs(local_basis.size(), 0.);

// ... at each quadrature point, given evauations of the local basis and the source function
for (size_t ii = 0; ii < local_basis.size(); ++ii) {
  local_vector[ii] += integration_element * quadrature_weight
                    * (source_value * basis_values[ii]);
  for (size_t jj = 0; jj < local_basis.size(); ++jj) {
    local_matrix.add_to_entry(ii, jj,
                              integration_element * quadrature_weight
                              * (basis_values[ii] * basis_values[jj]));
  }
}
```

Note that we neither have to manually specify the correctly matching matrix and vector types nor to include the correct headers (which depend on the current build configuration). The Container traits together with any of the default_... defines always yields appropriate available types.

We will see further below how these matrices and vectors are used in a variety of circumstances and examples.

Generic linear solvers in `Dune::Stuff::LA`. The last example from the previous paragraph illustrates how to generically create appropriate dense matrices and vectors. In order to determine the local DoF vector in the above example, we need to solve the algebraic problem: find `local_DoFs`, such that

$$\texttt{local\_matrix} \cdot \texttt{local\_DoFs} = \texttt{local\_vector}.$$

In addition to such small dense problems we also require the inversion of large (sparse) system and product matrices (compare Problems (3.1.15) and (3.1.24)). For interesting large and real-world problems, however, there are few linear solvers available which can be used as a black box (if at all). Most problems require a careful choice and detailed configuration of the correct linear solver. We thus require access to linear solvers which can be used in a generic way but also exchanged and configured at runtime.

In `dune/stuff/la/solver.hh` we provide such solvers via the `Solver` class:

```cpp
template< class MatrixType >
class Solver
{
  // simplified variant
public:
  Solver(const MatrixType& matrix);

  static std::vector< std::string > types();

  static Configuration options(const std::string type = "");

  template< class RhsType, class SolutionType >
  void apply(const RhsType& rhs, SolutionType& solution) const;

  template< class RhsType, class SolutionType >
  void apply(const RhsType& rhs, SolutionType& solution, const std::string& type) const;

  template< class RhsType, class SolutionType >
  void apply(const RhsType& rhs, SolutionType& solution, const Configuration& options) const;
};
```

We provide specializations of the `Solver` class for all matrix implementations derived from `MatrixInterface` (see the previous paragraph). Continuing the example from the previous paragraph, this allows to determine the local DoF vector of an L^2 projection:

```cpp
#include <dune/stuff/common/exceptions.hh>
#include <dune/stuff/la/solver.hh>

try {
  Stuff::LA::Solver< LocalMatrixType >(local_matrix).apply(local_vector, local_DoFs);
} catch (Stuff::Exceptions::linear_solver_failed& ee) {
  DUNE_THROW(Exceptions::projection_error,
             "L2 projection failed because a local matrix could not be inverted!\n\n"
             << "This was the original error: " << ee.what());
}
```

The above example shows a typical situation within the library code of `dune-gdt`: we need to solve a small dense system for provided matrices and vectors of unknown type. We can do so by instantiating a `Solver` and calling the black-box variant of `apply` (line 5). This apply variant is default implemented by calling

```
apply(rhs, solution, types()[0]);
```

where `types()` always returns a (non-empty) list of available linear solvers for the given matrix type, in descending priority (meaning the first is supposed to "work best"). For instance, if `local_matrix` was an `EigenDenseMatrix`, a call to `types()` would reveal the following available linear solvers:

```
{"lu.partialpiv", "qr.householder", "llt", "ldlt", "qr.colpivhouseholder",
 "qr.fullpivhouseholder", "lu.fullpiv"}
```

On the other hand, if the matrix was an `EigenRowMajorSparseMatrix`, a call to `types()` would yield

```
{"bicgstab.ilut", "lu.sparse", "llt.simplicial", "ldlt.simplicial", "bicgstab.diagonal",
 "bicgstab.identity", "qr.sparse", "cg.diagonal.lower", "cg.diagonal.upper",
 "cg.identity.lower", "cg.identity.upper"}
```

Given a (large sparse) `system_matrix` (for instance stemming from a discretized elliptic operator) and `rhs` vector, we can solve the corresponding linear system using a specific solver by calling

```
#include <dune/stuff/la/solver.hh>

Stuff::LA::Solver< SystemMatrixType > linear_solver(system_matrix);
linear_solver.apply(rhs, solution, "ldlt.simplicial");
```

The above call to `apply` is default implemented by calling

```
apply(rhs, solution, options(type));
```

where `options(type)` always returns a `Configuration` object with appropriate options for the selected `type`. With `type = "ldlt.simplicial"`, for instance, we are implicitly using the following options (which are the default for `"ldlt.simplicial"`):

```
type                        = ldlt.simplicial
post_check_solves_system = 1e-5
check_for_inf_nan           = 1
pre_check_symmetry          = 1e-8
```

All implemented solvers per default check whether the computed solution does actually solve the linear system and provide additional sanity checks. We make extensive use of exceptions if any check is violated, which allows to recover from undesirable situations in library code (as shown in the example above).[33] The `"ldlt.simplicial"` solver,

[33]We also provide our own implementation of the `DUNE_THROW` macro in `dune/stuff/common/exceptions.hh` which replaces the macro from `dune-common` and can be used with any `Dune::Exception`. Apart from a different formatting (and colorized output), it also provides information of interest in distributed memory parallel computations.

for instance, does only work for symmetric matrices and we thus check the matrix for symmetry (which can be disabled by setting `"pre_check_symmetry"` to 0). Each type of linear solver provides its own options, which allow the user to fine-tune the linear solver to his needs. For instance, the iterative `"bicgstab"` solver with `"ilut"` preconditioning accepts the following options (in addition to `"post_check_solves_system"` and `"check_for_inf_nan"`, which are always supported):

```
#include <dune/stuff/la/solver.hh>

Stuff::LA::Solver< SystemMatrixType > linear_solver(system_matrix);
auto options = linear_solver.options("bicgstab.ilut");

options["max_iter"]  = 1000;
options["precision"] = "1e-14";
options["preconditioner.fill_factor"] = 10;
options["preconditioner.drop_tol"]    = "1e-4";

linear_solver.apply(rhs, solution, options);
```

The actually implemented variant of `Solver` in `dune-stuff` also takes a communicator as an optional argument, to allow for distributed parallel linear solvers. We provide several implementations of such parallel solvers based on `dune-istl`. A linear solver for `IstlRowMajorSparseMatrix`, for instance, supports the following `types()`, most of which can readily be used in distributed parallel environments:

```
{
#if !HAVE_MPI && HAVE_SUPERLU
 "superlu",
#endif
           "bicgstab.amg.ssor", "bicgstab.amg.ilu0", "bicgstab.ilut", "bicgstab.ssor",
 "bicgstab"
#if HAVE_UMFPACK
          , "umfpack"
#endif
                      }
```

This finalizes our discussion of the main features of `dune-stuff` that we require for our discretization framework. Of course, `dune-stuff` provides many additional features, and we redirect any further interest to the project homepage: `https://github.com/wwu-numerik/dune-stuff`.

3.1.4.2 `dune-gdt`

The **DUNE** module `dune-gdt` is open source software and freely available on GitHub: `https://github.com/pymor/dune-gdt`. It is mainly developed by R. Milk and F. Schindler with contributions from M. Drohmann, S. Girke, S. Kaulmann, T. Leibner, M. Nolte and K. Weber. It forms the main part of our discretization framework and makes use of `dune-fem` and `dune-pdelab` (if available). Similar to the previous section we give illustrating code excerpts that are not necessarily directly usable.

As motivated in Section 3.1.1.1, there exist two main ingredients of discretization schemes: an approximation of ansatz and test spaces by discrete function spaces and an approximation of operators and functionals by discrete counterparts. We begin with a discussion of the former.

Generic discrete function spaces, mappers and base function sets. We already identified the mathematical requirements of a discrete function space in Definition 3.1.16. In `dune/gdt/spaces/interface.hh` we provide a `CRTP` interface which realizes these requirements:

```
 1 template< class Traits, size_t dimDomain, size_t dimRange, size_t dimRangeCols = 1 >
 2 class SpaceInterface
 3   : public Stuff::CRTPInterface< SpaceInterface< Traits, dimDomain, dimRange, dimRangeCols >
 4                                 , Traits >
 5 {
 6   // not all methods and types shown ...
 7 public:
 8   const BackendType&  backend()   const;
 9   const GridViewType& grid_view() const; // $\tau_h$
10   const MapperType&   mapper()    const; // $\iota$
11
12   BaseFunctionSetType base_function_set(const EntityType& entity) const; // $\phi^t_{k(t)}$
13
14   template< class G, class S, size_t d, size_t r, size_t rC >
15   PatternType compute_pattern(const GridView< G >& local_grid_view,
16                               const SpaceInterface< S, d, r, rC >& ansatz_space) const;
17
18   template< class S, size_t d, size_t r, size_t rC, class C >
19   void local_constraints(const SpaceInterface< S, d, r, rC >& ansatz_space,
20                          const EntityType& entity,
21                          Spaces::ConstraintsInterface< C >& ret) const;
22 };
```

All spaces in `dune-gdt` provide access to the DoF mapping ι and the set of local base functions $\phi^t_{k(t)}$, both of which we discuss further below. In addition, each space can compute local constraints (such as hanging nodes or Dirichlet constraints) and compute the sparsity pattern for operators acting on this space. The implementation of either is optional and depends on the space in question (for instance, a discontinuous Galerkin space does not provide any constraints while a continuous Lagrange space implements Dirichlet constraints, compare Definition 3.1.21). The `SpaceInterface` also grants access to a communicator for parallel computations and provides several default implementations, for instance to visualize the local basis functions or compute several types of patterns (not shown).[34]

Accompanying the `SpaceInterface`, we provide the `MapperInterface` in `dune/gdt/mapper/interface.hh`, modeling the DoF map ι (see Definition 3.1.16):

[34]We can also observe a nice use of `CRTP` interfaces in the above example (lines 15, 16, 19 and 21), following up on our discussion in the previous section. This use of `CRTP` enforces the correct type of arguments at compile time and documents expectations (as opposed to Duck typing, for instance).

```
template< class Traits >
class MapperInterface
  : public Stuff::CRTPInterface< MapperInterface< Traits >, Traits >
{
  // not all methods and types shown ...
public:
  const BackendType& backend() const;
  size_t size()              const;             // I
  size_t maxNumDofs()        const;             // max_{t∈τ_h} I(t)

  size_t numDofs(const EntityType& entity) const; // I(t)
  void globalIndices(const EntityType& entity, Dune::DynamicVector< size_t >& ret) const;
  size_t mapToGlobal(const EntityType& entity, const size_t& localIndex) const;
};
```

Given an element $t \in \tau_h$ of the grid (modeled by `entity`) and the size of the local basis $I(t) \in \mathbb{N}$ (modeled by `numDofs(entity)`), we have $\iota(t, i)$ = `mapToGlobal(entity, i)`, for all $0 \leq i < I(t)$. Since we usually treat all elements of a local basis at once, we also provide `globalIndices`, which computes `ret` $= \big(\iota(t,i)\big)_{i=0}^{I(t)-1}$. Depending on the backend, one of the methods is usually implemented in terms of the other (which is documented in each implementation). Thus, the interface allows for non-optimal methods, which one might argue against. However, we see this as a strength, as it facilitates quick prototyping of new discretization schemes.

A call to `space.base_function_set(entity)` yields a set of local basis functions, an interface for which we provide in `dune/gdt/basefunctionset/interface.hh` (recall the shorthands `D` and `d`, modeling $\mathbb{R}^d$ and `R`, `r` and `rC`, modeling $R^{r \times c}$, from the previous section):

```
template< class Traits, class D, size_t d, class R, size_t r, size_t rC = 1 >
class BaseFunctionSetInterface
  : public Stuff::LocalfunctionSetInterface< typename Traits::EntityType, D, d, R, r, rC >
  , public Stuff::CRTPInterface< BaseFunctionSetInterface< Traits, D, d, R, r, rC >, Traits >
{
  // not all methods and types shown ...
public:
  const BackendType& backend() const;
};
```

Obviously, the main purpose of a set of basis functions is to behave like a set of local functions and all relevant requirements are inherited by `LocalfunctionSetInterface`.

Since we use existing implementations of discrete function spaces from `dune-fem` and `dune-pdelab` we always provide direct access to the underlying space, mapper, local base function set or local space object from these modules, by means of the `backend()` method in `SpaceInterface`, `MapperInterface` and `BaseFunctionSetInterface`.[35] At this point, a review of the above interfaces is in order.

Remark 3.1.42 (Scope of the interfaces in `dune-gdt` and `dune-stuff`)**.** *It is apparent that the above interfaces are intended to model a single function space, and only in a*

[35]The same is true for the matrices and vectors from the previous section: a user can always call `backend()` on any matrix or vector to access the underlying wrapped object.

straightforward way. For instance, we always presume DoF vectors to be consecutively indexed and do not realize adaptivity by means of an adaptive grid view (we rather consider spaces on individual levels of the grid), in contrast to `dune-fem`*.*

For the approximation of systems (or in the context of domain decomposition) we provide support for blocked spaces and mappers. While this allows to realize different local DoF orderings, we do not provide support for arbitrary nested and combined spaces as `dune-pdelab` *does (though it is by no means prohibited by the interfaces).*

One motivation for this was for `dune-gdt` *to be usable in the broader context of model reduction, where we often require access to underlying (consecutive) containers. The main motivation, however, was to keep things simple and to enable developers and users to easily understand and use the spaces, mappers and base function sets in* `dune-gdt` *when developing new schemes, and to provide generic and transparent access to all underlying (mathematically relevant) components and concepts. It is clear that these design decisions limit the scope of* `dune-gdt`*; on the other hand* `dune-gdt` *may thus be easier to pick up than other discretization frameworks.*

In addition to the `SpaceInterface`, we provide interfaces for each class of discrete function spaces. For instance, we provide the `Spaces::CGInterface` in `dune/gdt/spaces/cg/interface.hh`, modeling a continuous Lagrange discrete function space in the sense of Definition 3.1.19:

```cpp
template< class Traits, size_t dimDomain, size_t dimRange, size_t dimRangeCols = 1 >
class CGInterface
  : public SpaceInterface< Traits, dimDomain, dimRange, dimRangeCols >
{
  // not all methods and types shown ...
public:
  // required by any derived class
  std::vector< DomainType > lagrange_points(const EntityType& entity) const;

  std::set< size_t > local_dirichlet_DoFs(const EntityType& entity,
                                          const BoundaryInfoType& boundaryInfo) const;

  // default implemented
  template< class G, class S, size_t d, size_t r, size_t rC >
  PatternType compute_pattern(const GridView< G >& local_grid_view,
                              const SpaceInterface< S, d, r, rC >& ansatz_space) const
  {
    return BaseType::compute_volume_pattern(local_grid_view, ansatz_space);
  }

  template< class S, size_t d, size_t r, size_t rC, class ConstraintsType >
  void local_constraints(const SpaceInterface< S, d, r, rC >& /*other*/,
                         const EntityType& /*entity*/,   // Use of CRTP: reject
                         ConstraintsType& /*ret*/) const // arbitrary ConstraintsType,
  {
    static_assert(AlwaysFalse< S >::value, "Not implemented for these constraints!");
  }
                                                          // but implement for
  template< class S, size_t d, size_t r, size_t rC >      // DirichletConstraints.
  void local_constraints(const SpaceInterface< S, d, r, rC >& /*other*/,
                         const EntityType& entity,
                         DirichletConstraints< IntersectionType >& ret) const
  {
    const auto local_DoFs = this->local_dirichlet_DoFs(entity, ret.boundary_info());
    if (local_DoFs.size() > 0) {
      const auto global_indices = this->mapper().globalIndices(entity);
      for (const auto& local_DoF : local_DoFs) {
        ret.insert(global_indices[local_DoF]);
      }
    }
  } // ... local_constraints(...)
}; // class CGInterface
```

The `CGInterface` enforces all derived classes to provide the local Lagrange points $\big(\hat{\nu}_i^t\big)_{i=0}^{I(t)-1}$ (see Definition 3.1.19), as well the indices of those points, which lie on the Dirichlet boundary (modeled by `local_dirichlet_DoFs`, given a boundary info as discussed in the previous section). It also computes the correct sparsity pattern (by calling the appropriate method on `SpaceInterface`, not shown) and Dirichlet constraints, once again demonstrating a nice use for the `CRTP` paradigm.

As already hinted at we provide implementations of these interfaces based on the `dune-fem` and `dune-pdelab` modules in `dune/gdt/spaces/cg/{fem,pdelab}.hh`. Either implementation has its strengths and weaknesses, in particular regarding parallelization and supported grid types. Mathematically, however, both behave identically and model the same continuous Lagrange discrete function space. In order to facilitate the use of different backends we provide the `Spaces::CGProvider` traits in `dune/gdt/spaces/cg.hh`, along with the `ChooseSpaceBackend` enum class. They allow for generic code by using the grid provider discussed in the previous section and the appropriate tags:

```
#include <dune/gdt/spaces/cg.hh>

using namespace Dune::Stuff::Grid;
using namespace Dune::GDT::Spaces;

template< class GridType, ChooseLayer layer_type, ChooseSpaceBackend backend_type >
void discretize(ProviderInterface< GridType >& grid_provider,
                const int lv = 0)
{
  // create a CG space of scalar piecewise linear functions
  typedef CGProvider< GridType, layer_type, backend_type, 1, double, 1 > SpaceProvider;
  auto cg_space = SpaceProvider::create(grid_provider, lv);
  // ...
}
```

Note that the user neither has to provide the correct includes (which depend on the build configuration), nor does he have to create the correct grid view or grid part to match the designated space. By providing a `backend_type` (e.g., `fem`, `pdelab` or `default_cg_backend`) and a `layer_type` (e.g., `level` or `leaf`), the correct leaf/level view/part is automatically created from the given `grid_provider` (where `lv` is ignored, if `layer_type == leaf`). It is thus finally possible to completely exchange the implementation of a discrete function space by only changing one compile-time constant.

In addition to the `CGInterface`, we also provide respective interfaces and implementations for discontinuous Galerkin, Finite Volume and Raviart-Thomas-Nédélec discrete function spaces in `dune/gdt/spaces/{dg,fv,rt}.hh` (compare Sections 3.1.1.1 and 3.1.1.2).

The discrete function. The discrete function spaces from the previous paragraph are modeled after the mathematical requirements in Definition 3.1.16. The resulting interpretation of each local basis function as a local function of a global basis functions greatly simplifies the implementation of discrete functions in the sense of Definition 3.1.17: conceptually, only a single implementation is required. In `dune/gdt/discretefunction/default.hh`, we provide the

```
template< class SpaceType, class VectorType >
class ConstDiscreteFunction
  : public Stuff::LocalizableFunctionInterface< ... >
{
  // not all methods and types shown ...
public:
  const SpaceType&  space()  const;
  const VectorType& vector() const;

      virtual std::unique_ptr< LocalfunctionType >
  local_function(const EntityType& entity) const override;

      std::unique_ptr< ConstLocalDiscreteFunctionType >
  local_discrete_function(const EntityType& entity) const;
};
```

which allows to interpret a given const vector as a discrete function (read-only) and

```
template< class SpaceType, class VectorType >
class DiscreteFunction
  : Stuff::Common::StorageProvider< VectorType >
  , public ConstDiscreteFunction< SpaceType, VectorType >
{
  // not all methods an types shown ...
public:
  VectorType& vector();

      std::unique_ptr< LocalDiscreteFunctionType >
  local_discrete_function(const EntityType& entity);
};
```

which also allows to alter the DoF `vector`. Both implementations provide a local discrete function (in the sense of Definition 3.1.17), which in turn allows to access the local DoF vector (read-only or writable, respectively).

The main purpose of the `ConstDiscreteFunction` is to allow to represent a given DoF vector as a discrete function of a given space. It can for instance be used to visualize an approximate solution:

```cpp
#include <dune/gdt/discretefunction/default.hh>

using namespace Dune::GDT;

template< class SpaceType, class VectorType >
void visualize_solution(const SpaceType& space, const VectorType& solution,
                        const std::string& filename)
{
  ConstDiscreteFunction< SpaceType, VectorType >(space, vector).visualize(filename);
}
```

The `DiscreteFunction`, on the other hand, can also be created without an existing vector (in which case a vector of correct size is automatically created, see lines 19 and 21):[36]

```cpp
 1  #include <dune/stuff/la/container/vector-interface.hh>
 2  #include <dune/gdt/discretefunction/default.hh>
 3  #include <dune/gdt/spaces/interface.hh>
 4
 5  using namespace Dune::Stuff::LA;
 6  using namespace Dune::GDT;
 7
 8  template< class S, size_t d, size_t r, size_t rC, class V, class R >
 9      typename VectorInterface< V >::derived_type
10  prolong_vector(const SpaceInterface< S, d, r, rC >& source_space,
11                 const VectorInterface< V >&          source_vector,
12                 const SpaceInterface< R, d, r, rC >& range_space)
13  {
14    typedef typename SpaceInterface< S, d, r, rC >::derived_type SourceSpaceType;
15    typedef typename VectorInterface< V >::derived_type          VectorType;
16    typedef typename SpaceInterface< R, d, r, rC >::derived_type RangeSpaceType;
17    ConstDiscreteFunction< SourceSpaceType, VectorType > source(source_space.as_imp(),
18                                                                source_vector.as_imp());
19    DiscreteFunction< RangeSpaceType, VectorType >       range(range_space.as_imp());
20    // carry out the prolongation ...
21    return range.vector();             // <- No copy required here, due to copy-on-write
22  }                                    //    and move semantics for all containers.
```

The above (ficticous) example also demonstrates the use of `CRTP` interfaces to enforce matching dimensions of the spaces (modeled by `d`, `r` and `rC`); the template arguments `S`, `V` and `R` model the traits of the source space, vector and range space type, respectively.

The main purpose of the `DiscreteFunction` is to allow write access to the DoF vector. It is thus used for projections and prolongations, as presented further below.

A note on operators and two-forms. Before we introduce the local building blocks for operators and functionals we revisit the concept of operators and two-forms. As already discussed, for instance in Remark 3.1.1, there is a close relationship between operators and two-forms (such as products and bilinear forms). In the discrete (finite dimensional)

[36]This is possible since the `StorageProvider` from `dune/stuff/common/memory.hh` allows to generically handle the case where we want to hold a reference to an existing object or create a new object.

setting the situation is similar, but at the same time slightly more difficult regarding the implementation. Consider two spaces of localizable functions, $Q(\tau_h^s)$ and $R(\tau_h^r)$, and two discrete function spaces $\mathbb{V}_h(\tau_h^s) \subset Q(\tau_h^s)$ and $\mathbb{W}_h(\tau_h^r) \subset R(\tau_h^r)$. In general, we think of a discrete operator B and its inverse B^{-1} as mappings between two spaces of localizable functions[37] (which we denote the *source* and *range* of the operator),

$$B : Q(\tau_h^s) \to \mathbb{W}_h(\tau_h^r) \qquad \text{and} \qquad B^{-1} : R(\tau_h^r) \to \mathbb{V}_h(\tau_h^s),$$

and of products (or bilinear forms) as an interpretation of the operator as a two-form:

$$(\cdot,\cdot)_B : R(\tau_h^r) \times Q(\tau_h^s) \to \mathbb{R}, \qquad (r,q) \mapsto (r,q)_B := B[q](r).$$

Actual implementations of operators and two-forms may also be restricted to acting only on the discrete spaces $\mathbb{V}_h(\tau_h^s)$ and $\mathbb{W}_h(\tau_h^r)$. This already indicates that the implementation of an operator differs from that of a two-form (although the mathematical concepts are closely related): operators shall provide

```
void apply(const SourceType& source, RangeType& range);
```

allowing them to alter the `range` discrete function, while the interpretation of an operator as a two-form shall provide

```
FieldType apply2(const RangeType& range, const SourceType& source);
```

only requiring read access to its arguments. While we discuss the `OperatorInterface` in detail further below, the above distinction is vital for an understanding of the local building blocks introduced in the following paragraph.

Local evaluations, two-forms and functionals. As discussed extensively in Sections 3.1.1 and 3.1.2, we require operators, two-forms and functionals, which can be expressed in a localized fashion with respect to a grid. We first consider operators and functionals based on integrals, which in turn are transformed to local integrals on the reference elements of the grid. We approximate those local integrals by a splitting into the integrand and the numerical quadrature. The former is modeled by local evaluations, while the latter is modeled by local integral operators and functionals. We discuss our implementation of this concept for volume integral operators; coupling and boundary integral operators and integral functionals work analogously.

Therefore, we consider the example of an elliptic PDE from section 3.1.1.1. At some point we need to evaluate the local elliptic operator which approximates the following local integral on the reference element $\hat{t}$, where λ and κ model the diffusion factor and tensor, φ^t and ψ^t denote local ansatz and test basis functions and Λ_t stems from the geometric transformation of the integral (compare Definition 3.1.6 and Example 3.1.7):

$$B_h^t[\varphi^t](\psi^t) = \Sigma_h^t\big[\Upsilon_{\text{ell.}}^t[\lambda,\kappa]\big]\big(\psi^t,\varphi^t\big) \approx \int_{\hat{t}} \Lambda_t \, \underbrace{\big(\lambda^t\kappa^t\nabla_t\psi^t\big)\cdot\varphi^t}_{\Upsilon_{\text{ell.}}^t} \mathrm{d}x.$$

[37]Note that each localizable function has locally fixed polynomial degrees. Thus, we can think of $Q(\tau_h^s)$ as finite dimensional.

The local elliptic operator B_h^t is composed of the elliptic evaluation $\Upsilon_{\text{ell.}}^t$ (modeling the integrand) and the local volume operator Σ_h^t, the purpose of which is to approximate the integral (given any integrand). We provide an implementation of Σ_h^t in `dune/gdt/localoperator/integrals.hh` (of which we show a simplified variant). Note that the `LocalVolumeIntegralOperator` is actually a local two-form and thus derived from `LocalVolumeTwoFormInterface`:

```cpp
#include <dune/common/dynmatrix.hh>
#include <dune/geometry/quadraturerules.hh>
#include <dune/stuff/functions/interfaces.hh>
#include "interface.hh"

using namespace Dune;
using namespace Dune::Stuff;

template< class BinaryEvaluationType >
class LocalVolumeIntegralOperator // $\Sigma_h^t$ from Definition 3.1.6.
  : public LocalVolumeTwoFormInterface< ... >
{
  // not all methods and types shown ...
public:
  template< class ...Args >
  explicit LocalVolumeIntegralOperator(const size_t over_integrate, Args&& ...args)
    : integrand_(std::forward< Args >(args)...)
    , over_integrate_(over_integrate)
  {}

  template< class E, class D, size_t d, class R, size_t rT, size_t rCT, size_t rA, size_t rCA >
  void apply2(const LocalfunctionSetInterface< E, D, d, R, rT, rCT >& test_base,   // $(\psi_i^t)_i$
              const LocalfunctionSetInterface< E, D, d, R, rA, rCA >& ansatz_base, // $(\varphi_j^t)_j$
              Dune::DynamicMatrix< R >& ret) const // $(\Sigma_h^t[\dots][\varphi_j^t](\psi_i^t))_{i,j}$
  {
    const auto& entity = ansatz_base.entity();
    const auto local_functions = integrand_.localFunctions(entity);
    // create quadrature
    const size_t integrand_order = integrand_.order(local_functions, ansatz_base, test_base)
                                   + over_integrate_;
    const auto& quadrature
        = QuadratureRules< D, d >::rule(entity.type(),
                                        boost::numeric_cast< int >(integrand_order));
    // prepare storage
    ret *= 0.0;
    const size_t rows = test_base.size();
    const size_t cols = ansatz_base.size();
    DynamicMatrix< R > evaluation_result(rows, cols, 0.);
    // loop over all quadrature points
    for (const auto& quadrature_point : quadrature) {
      const auto xx = quadrature_point.position();                          // $x_n^t$
      // integration factors
      const auto integration_factor = entity.geometry().integrationElement(xx); // $\Lambda_t(x_n^t)$
      const auto quadrature_weight = quadrature_point.weight();             // $w_n^t$
      // evaluate the integrand
      integrand_.evaluate(local_functions, ansatz_base, test_base, xx, evaluation_result);
      // compute integral
      for (size_t ii = 0; ii < rows; ++ii) {
        auto& ret_row = ret[ii];
        const auto& evaluation_result_row = evaluation_result[ii];
        for (size_t jj = 0; jj < cols; ++jj)
          ret_row[jj] += evaluation_result_row[jj] * integration_factor * quadrature_weight;
      } // compute integral
    } // loop over all quadrature points
  } // ... apply2(...)

private:
  const BinaryEvaluationType integrand_;
  const size_t over_integrate_;
}; // class LocalVolumeIntegralOperator
```

The only method that is enforced by the `LocalVolumeTwoFormInterface` is the above `apply2` method. As input, `apply2` takes any two sets of local functions, $\big(\varphi_j^t\big)_j$ and $\big(\psi_i^t\big)_i$ modeled by `test_base` and `ansatz_base`, which are defined on the same grid (modeled by the template parameters `E`, `D` and `d`) but may map to different dimension (modeled by `rT` and `rCT` for the test and by `rA` and `rCA` for the ansatz functions). As output, the evaluation of the local two-form operator for all combinations of these local functions, $\big(\Sigma_h^t[\dots][\varphi_j^t](\psi_i^t)\big)_{i,j}$, is written to `ret`.

Before we continue with the discussion of the implementation of `apply2`, we demonstrate how to combine the `LocalVolumeIntegralOperator` (modeling Σ_h^t) with the `Elliptic` evaluation from `dune/gdt/localevaluation/elliptic.hh` (modeling $\Upsilon_{\text{ell.}}^t$), in order to form the elliptic operator B_h^t from Example 3.1.7. The `Elliptic` evaluation needs to hold the diffusion factor λ and tensor κ, while the `LocalVolumeIntegral-Operator` holds the `Elliptic` evaluation as the `integrand_` member. In order to simplify the creation of the resulting operator, all integral operators provide perfect forwarding (lines 15–17). We thus create the final operator by simply defining its type and passing on the required arguments of the `Elliptic` evaluation (without the need to create the evaluation first):

```
#include <dune/gdt/localevaluation/elliptic.hh>
#include <dune/gdt/localoperator/integrals.hh>

using namespace Dune::GDT;

typedef LocalVolumeIntegralOperator< LocalEvaluation::Elliptic<
    DiffusionFactorType, DiffusionTensorType > > EllipticOperatorType;

EllipticOperatorType elliptic_operator(1, // Integrate one order higher than required.
                                       diffusion_factor,  // <- λ
                                       diffusion_tensor); // <- κ
```

The elliptic evaluation is a binary (since it accepts two sets of local functions) codim 0 evaluation (since it can be evaluated on a codim 0 reference element) and thus derived from `LocalEvaluation::Codim0Interface< ..., 2 >`, which we provide in `dune/gdt/localevaluation/interface.hh`:

```
template< class Traits >
class /*LocalEvaluation::*/Codim0Interface< Traits, 2 >
  : public Stuff::CRTPInterface< Codim0Interface< Traits, 2 >, Traits >
{
  // not all methods and types shown ...
public:
  LocalfunctionTupleType localFunctions(const EntityType& entity) const,

  template< class R, size_t rT, size_t rCT, size_t rA, size_t rCA >
  size_t order(const LocalfunctionTupleType& local_functions_tuple,
               const Stuff::LocalfunctionSetInterface< E, D, d, R, rT, rCT >& test_base,
               const Stuff::LocalfunctionSetInterface< E, D, d, R, rA, rCA >& ansatz_base) const;

  template< class R, size_t rT, size_t rCT, size_t rA, size_t rCA >
  void evaluate(const LocalfunctionTupleType& local_functions_tuple,
                const Stuff::LocalfunctionSetInterface< E, D, d, R, rT, rCT >& test_base,
                const Stuff::LocalfunctionSetInterface< E, D, d, R, rA, rCA >& ansatz_base,
                const FieldVector< D, d >& local_point,
                DynamicMatrix< R >& ret) const;
};
```

All interfaces for local evaluations enforce the above three methods (`localFunctions`, `order` and `evaluate`) and only differ in their respective arguments.

We return to the discussion of the implementation of the `apply2` method of the `LocalVolumeIntegralOperator` above. This method is called on each grid element $t \in \tau_h$ (modeled by `entity`) and in turn calls the elliptic evaluation at each quadrature point. The elliptic evaluation holds the localizable functions λ and κ, but requires the respective local functions on each grid element, λ^t and κ^t, in order to compute the elliptic evaluation $\big(\lambda^t(x_n^t)\kappa^t(x_n^t)\nabla_t\psi^t(x_n^t)\big) \cdot \nabla_t\varphi^t(x_n^t)$ for each quadrature point x_n^t. Since we do not want the elliptic evaluation to create the local functions λ^t and κ^t for each quadrature point anew, each local evaluation exports its required local functions for a grid element via the `localFunctions` method (returning a `std::tuple` of `std::shared_ptrs` of local functions). And since the local operator does not know about the individual local functions (and does not have to), it just passes this tuple back to the local evaluation in each call (for the local evaluation to extract the required local functions).

For instance, in order to obtain a quadrature of correct order, the local operator passes λ^t and κ^t wrapped inside `local_functions` as well as the test and ansatz bases to the local evaluation to obtain the polynomial `order` of the integrand (see line 29 of the `apply2` implementation above). At each quadrature point x_n^t (modeled by `xx`), the operator passes the same arguments along with `xx` to the `evaluate` method of the local evaluation (see line 45 of the `apply2` example above). After evaluating the integrand, the operator computes the approximation of the integral by multiplication with the quadrature weight w_n^t and $\Lambda_t(x_n^t)$, for all combinations of local test and ansatz functions (see lines 47–52 of the `apply2` implementation above).

Let us now consider the implementation of the `Elliptic` evaluation, for example the `evaluate` method. As demonstrated in `apply2`, `evaluate` is called for each quadrature point, given a tuple of local functions as the first argument. The first step in the implementation of the `Elliptic` evaluation is to extract the individual local functions

λ^t and κ^t and redirect the actual computation of the evaluation:

```
1  template< class R, size_t rT, size_t rCT, size_t rA, size_t rCA >
2  void evaluate(const LocalfunctionTupleType& local_functions_tuple,
3                const Stuff::LocalfunctionSetInterface< E, D, d, R, rT, rCT >& test_base,
4                const Stuff::LocalfunctionSetInterface< E, D, d, R, rA, rCA >& ansatz_base,
5                const FieldVector< D, d >& local_point,
6                DynamicMatrix< R >& ret) const
7  {
8    const auto local_diffusion_factor = std::get< 0 >(local_functions_tuple);
9    const auto local_diffusion_tensor = std::get< 1 >(local_functions_tuple);
10   evaluate(*local_diffusion_factor, *local_diffusion_tensor,
11            test_base, ansatz_base, local_point, ret);
12 }
```

After extracting the local functions, an implementation of the redirected `evaluate` method (line 10) could be provided by

```
1  template< class R >
2  void evaluate(const Stuff::LocalfunctionInterface< E, D, d, R, 1, 1 >& diffusion_factor,
3                const Stuff::LocalfunctionInterface< E, D, d, R, d, d >& diffusion_tensor,
4                const Stuff::LocalfunctionSetInterface< E, D, d, R, 1, 1 >& test_base,
5                const Stuff::LocalfunctionSetInterface< E, D, d, R, 1, 1 >& ansatz_base,
6                const Dune::FieldVector< D, d >& local_point,
7                Dune::DynamicMatrix< R >& ret) const
8  {
9    // evaluate local functions
10   const auto       diffusion_factor_value = diffusion_factor.evaluate(local_point);
11   const TensorType diffusion_tensor_value = diffusion_tensor.evaluate(local_point);
12   const auto diffusion_value = diffusion_tensor_value * diffusion_factor_value;
13   // evaluate basis functions (yields std::vector< ... > of jacobian evaluations)
14   const auto test_gradients   = test_base.jacobian(local_point);
15   const auto ansatz_gradients = ansatz_base.jacobian(local_point);
16   // compute elliptic evaluation
17   for (size_t ii = 0; ii < test_base.size(); ++ii)
18     for (size_t jj = 0; jj < ansatz_base.size(); ++jj)
19       ret[ii][jj] = (diffusion_value * test_gradients[jj][0]) * ansatz_gradients[ii][0];
20 }
```

making use of our extension of `dune-common`'s `FieldMatrix`[38] (as `TensorType` above) to provide convenience multiplication operators (lines 12 and 19).[39] The above example demonstrates the power of the `LocalfunctionSetInterface`: we can exchange functions at runtime, but only those with respect to the same grid and dimensions. Together with the matching of template arguments above, we can ensure the correctness of arguments at compile time (for instance the dimensions of λ^t, see line 3) and at the same time allow the compiler to generate optimal code (since all vector and matrix types and

[38]`dune/stuff/common/fmatrix.hh`

[39]The use of such convenience operators does involve the creation of temporary objects, which is not optimal. On the other hand, they allow for generic code such as the one displayed above, which would otherwise not compile for all dimensions (since the implementation of `Dune::FieldMatrix< double, 1, 1 >` does not provide the required `mv` method, in contrast to the implementation for all other dimensions).

sizes are fixed at compile time). While a fairly simple example, the implementation of the `Elliptic` evaluation demonstrates how developers and users can provide custom implementations of their integrands for different combinations of dimensions, if desired.

In addition to the `LocalVolumeIntegralOperator`, we provide similar implementations for coupling and boundary integral operators and volume and boundary integral functionals, which together cover all possibly arising integrals. In the context of continuous and discontinuous discretizations of linear elliptic PDEs, we provide the following local evaluations within the `LocalEvaluation` namespace:

- `Elliptic` in `dune/gdt/localevaluation/elliptic.hh`, implementing the local elliptic evaluation $\Upsilon^t_{\text{ell.}}$ from Example 3.1.7.
- `Product` in `dune/gdt/localevaluation/product.hh`, implementing the local product evaluation $\Upsilon^t_{\text{prod.}}$ from Examples 3.1.10, 3.1.13 and 3.1.30. We do so by deriving from `Codim0Interface< ..., 1 >`, `Codim0Interface< ..., 2 >`, `Codim1Interface< ..., 1 >` and `Codim1Interface< ..., 2 >`, and by providing the appropriate specializations of `order` and `evaluate` for all cases.
- The `SWIPDG` namespace in `dune/gdt/localevaluations/swipdg.hh`, providing
  - `Inner`, implementing the local SWIPDG coupling evaluation $\mathring{\Upsilon}_{\text{SWIP}}$ from Example 3.1.27 and
  - `Boundary{LHS,RHS}`, implementing the local SWIPDG boundary evaluation $\overline{\Upsilon}^e_{\text{SWIP}}$ from Examples 3.1.25 and 3.1.23.

Local operators. In addition to the integral operators and functionals discussed in the previous paragraph we also provide an interface for local building blocks for other more general operators in `dune/gdt/localoperator/interfaces.hh`:

```cpp
template< class Traits >
class LocalOperatorInterface
  : public Stuff::CRTPInterface< LocalOperatorInterface< Traits >, Traits >
{
  // not all types shown ...
public:
  template< class SourceType, class RangeSpaceType, class VectorType >
  void apply(const SourceType& source,
             LocalDiscreteFunction< RangeSpaceType, VectorType >& local_range) const;
};
```

The `apply` methods of the `LocalOperatorInterface` allows for a localizable function or a discrete function as `source`. While being sufficient for our purposes this interface might have to be extended for general matrix-free operators in the future.

The `LocalOperatorInterface` is for instance suitable for the local L^2-projection operator from Definition 3.1.40, which is given by its action on the local discrete function of its range. A possible implementation of the local operator constituting the local L^2-projection operator could be realized in terms of the `LocalVolumeIntegralOperator`, `LocalVolumeIntegralFunctional` and `LocalEvaluation::Product`:

```
 1 #include <dune/stuff/functions/interfaces.hh>
 2 #include <dune/stuff/la/container/common.hh>
 3 #include <dune/stuff/la/solver.hh>
 4
 5 #include <dune/gdt/discretefunction/local.hh>
 6 #include <dune/gdt/localevaluation/product.hh>
 7 #include <dune/gdt/localfunctional/integrals.hh>
 8 #include <dune/gdt/localoperators/integrals.hh>
 9
10 class LocalL2ProjectionOperator
11   : public LocalOperatorInterface< ... >
12 {
13   // not all methods and types shown ...
14 public:
15   template< class E, class D, size_t d, class R, size_t r, size_t rC
16           , class RangeSpaceType, class VectorType >
17   void apply(const Stuff::LocalizableFunctionInterface< E, D, d, R, r, rC >& source,
18              LocalDiscreteFunction< RangeSpaceType, VectorType >& local_range) const
19   {
20     // create local L2 operator and functional
21     const LocalVolumeIntegralOperator< LocalEvaluation::Product< /*...*/ > >
22         local_l2_operator(over_integrate_/*, ...*/);
23     const LocalVolumeIntegralFunctional< LocalEvaluation::Product< SourceType >
24         local_l2_functional(over_integrate_, source);
25     // create local lhs and rhs
26     const auto& local_basis = local_range.basis();
27     Stuff::LA::CommonDenseMatrix< R > local_matrix(local_basis.size(), local_basis.size());
28     Stuff::LA::CommonDenseVector< R > local_vector(local_basis.size());
29     // assemble
30     local_l2_operator.apply2(local_basis, local_basis, local_matrix.backend());
31     local_l2_functional.apply(            local_basis, local_vector.backend());
32     // solve
33     Stuff::LA::CommonDenseVector< R > local_solution(local_basis.size());
34     try {
35       Stuff::LA::Solver< Stuff::LA::CommonDenseMatrix< R > >(local_matrix).apply(tmp_rhs,
36                                                                                local_solution);
37     } catch (Stuff::Exceptions::linear_solver_failed& ee) {
38       DUNE_THROW(Exceptions::projection_error,
39                  "L2 projection failed because a local matrix could not be inverted!\n\n"
40                  << "This was the original error: " << ee.what());
41     }
42     // set local DoFs
43     for (size_t ii = 0; ii < local_range_vector.size(); ++ii)
44       local_range.vector().set(ii, local_solution[ii]);
45   } // ... apply(...)
46
47 private:
48   const size_t over_integrate_;
49 }; // class LocalL2ProjectionOperator
```

While possibly not an optimal implementation, the above example demonstrates how to realize a local L^2-projection using existing building blocks. We initialize the L^2-product operator from Definition 3.1.30 (line 22) and the local L^2-volume functional from Example 3.1.10 (line 24) and assemble both into appropriate local containers (lines 30–31). We immediately obtain the local DoF vector as the solution of the local dense linear system using the linear solvers introduced in the previous Section (line 35).

Putting it all together: unified computation of localized quantities. With the local operators, two-forms and functionals we have all the local building blocks we need to approximate the solution of a linear elliptic PDE. Further below, we also introduce ready built global operators, two-forms and functionals that are more convenient in many scenarios. However, sometimes convenience has to be traded in for full control over all building blocks (for instance when developing a new discretization scheme). For such use cases we provide local assemblers (which copy a local matrix into the respective entries of a global matrix) and the `SystemAssembler` as an extension of the `GridWalker` from `dune-stuff` (introduced in the previous section).

Given a test and ansatz discrete function space, the purpose of the `SystemAssembler` is to allow for the assembly and computation of any local operator, local two-form or local functional (in addition to the functors already accepted by the `GridWalker`). Presuming we have already created the local building blocks (for instance as detailed in the previous paragraph), we can add them to the `SystemAssembler` in order to compute them all in one iteration over the grid:

```
 1  #include <dune/gdt/assembler/local/codim0.hh>
 2  #include <dune/gdt/assembler/local/codim1.hh>
 3  #include <dune/gdt/assembler/system.hh>
 4
 5  // Given a local elliptic operator for $\lambda$ and $\kappa$, a local $L^2$ volume functional for $f$,
 6  // a local $L^2$ boundary functional for $g_N$ and a dirichlet projection for $g_D$, ...
 7  // create local assemblers,
 8  LocalAssembler::Codim0Matrix
 9      < LocalEllipticOperatorType >      elliptic_assembler(local_elliptic_operator);
10  LocalAssembler::Codim0Vector
11      < LocalL2VolumeFunctionalrType >   force_assembler(local_force_functional);
12  LocalAssembler::Codim1Vector
13      < LocalL2BoundaryFunctionalrType > neumann_assembler(local_neumann_functional);
14  // create system matrix and right hand side,
15  MatrixType system_matrix(test_space.mapper().size(),       // (Choose the correct sparsity
16                           ansatz_space.mapper().size(),     // pattern for elliptic operator
17                           test_space.compute_volume_pattern(ansatz_space); // and CG space.)
18  VectorType rhs_vector(test_space.mapper().size());
19  // add them to the system assembler,
20  SystemAssembler< SpaceType > assembler(test_space, ansatz_space);
21  assembler.add(elliptic_assembler, system_matrix);
22  assembler.add(force_assembler, rhs_vector);
23  assembler.add(neumann_assembler, rhs_vector,       // (only on Neumann boundary intersections)
24                new Stuff::Grid::ApplyOn::NeumannIntersections< GridViewType >(boundary_info));
25  assembler.add(dirichlet_projection);               // (is not assembled but locally computed)
26  // compute everything in one grid iteration and
27  assembler.assemble();
28  // solve the linear system (see previous section) ...
```

We manually create the global system matrix with the correct sparsity pattern (lines 15–

17) and add the corresponding local assembler (line 21). During the grid walk triggered in line 27, the local elliptic two-form operator will compute the elliptic integrals for all test and ansatz basis functions and the local assembler (created in lines 8–9) will copy the resulting local matrix into the global matrix using the `mapper()`s of the test and ansatz space. In the same spririt, we create a global right hand side vector (line 18) that the two local vector assemblers (created in lines 10–13) will assemble into. Note that we restrict the assembly of the local L^2 boundary integral for the Neummann boundary values to those intersections of the grid that lie on the Neumann boundary, by simply providing the `ApplyOn::NeumannIntersections` tag (line 24) together with a `boundary_info` object (see the previous section). We provide various tags of this kind in `dune/stuff/grid/walker/apply-on.hh` to manually restrict the set of entities or intersections a local functor is applied on.

We demonstrate further below how many of these manual tasks are provided by ready built global operators, two-forms and functionals and how a complete approximation of a linear elliptic PDE can be easily realized in terms of these global objects. Nevertheless, the above example demonstrates how a user or developer has full control over all local building blocks. Since these local building blocks can be combined in many ways, he or she is empowered to rapidly develop new discretization schemes by combining existing building blocks with new custom implementations.

We would also like to note (again), that the above `assemble` method can be used to trigger a sequential as well as a shared memory parallel iteration over the grid by simply providing a runtime switch.

Operators, two-forms and functionals. We already introduced the local building blocks which are essential to our discretization framework, above, in a paragraph on local operators, local two-forms, local functionals and local evaluations. However, as demonstrated in the previous paragraph, those need to be combined in the correct way to successfully define a discretization scheme (e.g., the sparsity pattern of the system matrix has to match the local operator and discrete function space, the assembly of the boundary integrals has to be restricted to the right set of intersections, and so forth). While there exist circumstances where this fine-grained control is exactly what we wish for, it is also convenient to have access to ready built global operators, two-forms and functionals.

In addition, the local building blocks lack important information that is only available if we consider the respective global counterpart: while a local operator might be enough to determine how to apply an operator, it is for instance not sufficient to define how the operator should be inverted (e.g., by an iterative linear solver repeatedly applying the operator or by a Newton scheme requiring access to the operators jacobian). Lastly, we require global operators and functionals in the context of model reduction. Most model reduction algorithms can be generically implemented acting on (global) operators and functionals and do not require any knowledge about the local building blocks of the operator (compare Section 3.2).

We briefly revisit our view on operators, two-forms and functionals before presenting their respective implementation in `dune-gdt`. In general, we think of a discrete operator

B and its inverse B^{-1} as mappings between two spaces of localizable functions (which we denote the *source* and *range* of the operator),

$$B : Q(\tau_h^s) \to R(\tau_h^r) \qquad \text{and} \qquad B^{-1} : R(\tau_h^r) \to Q(\tau_h^s),$$

and of products or bilinear forms as an interpretation of the operator as a two-form:

$$(\cdot,\cdot)_B : R(\tau_h^r) \times Q(\tau_h^s) \to \mathbb{R}, \qquad (r,q) \mapsto (r,q)_B := B[q](r). \tag{3.1.31}$$

While we do provide implementations of products acting on localizable functions (for instance in the spirit of Definition 3.1.32), operators usually need to act on discrete function spaces: the application of the operator or its inverse requires access to the image, which is only possible through the DoF vector of a discrete function. To illustrate this point we consider some examples:

- Consider a Finite Volume operator L_h in the context of a (possibly nonlinear) time-dependent PDE. Such an operator would act on the solution at the previous time step to produce the solution at the next time step (which would be sufficient for an explicit scheme). Thus, L_h would map onto a (Finite Volume) discrete function space $\mathbb{W}_h(\tau_h^r) \subset R(\tau_h^r)$, taking as input any localizable function or possibly only Finite Volume discrete functions (for performance reasons). We thus either have

  $$L_h : Q(\tau_h^s) \to \mathbb{W}_h(\tau_h^r) \qquad \text{or} \qquad L_h : \mathbb{V}_h(\tau_h^s) \to \mathbb{W}_h(\tau_h^r), \tag{3.1.32}$$

  with a (Finite Volume) discrete function space $\mathbb{V}_h(\tau_h^s) \subset Q(\tau_h^s)$. If required (for instance for an implicit scheme), the inverse of the operator would be realized by a generic iterative linear solver if L_h was linear (solely requiring the application of the operator) or a variant of a Newton scheme if L_h was nonlinear (requiring the application of the operator and its jacobian). Either way, the inverse of L_h would exclusively map between discrete function spaces:

  $$L_h^{-1} : \mathbb{W}_h(\tau_h^r) \to \mathbb{V}_h(\tau_h^s).$$

  If the operator was implemented to act on $\mathbb{V}_h(\tau_h^s)$ (in contrast to $Q(\tau_h^s)$), it could also be interpreted as a two-form, by means of the Euclidean product on $\mathbb{W}_h(\tau_h^r)$:

  $$(\cdot,\cdot)_{L_h} : \mathbb{W}_h(\tau_h^r) \times \mathbb{V}_h(\tau_h^s) \to \mathbb{R}, \qquad (w_h, v_h)_{L_h} := \underline{w_h} \cdot L_h[v_h] \tag{3.1.33}$$

- Consider on the other hand the elliptic operator B_h and its matrix representation $\underline{B_h}$ in the context of a stationary linear elliptic PDE (see Definition 3.1.14). If interpreted as a product, B_h can act on localizable functions:

  $$(\cdot,\cdot)_{B_h} : R(\tau_h^r) \times Q(\tau_h^s) \to \mathbb{R}. \tag{3.1.34}$$

  However, by means of its matrix representation, we obtain an operator (and its inverse) acting on discrete functions,

  $$B_h : \mathbb{V}_h(\tau_h^s) \to \mathbb{W}_h(\tau_h^r), \qquad v_h \mapsto w_h := B_h[v_h], \text{ with } \underline{w_h} := \underline{B_h}\,\underline{v_h}, \tag{3.1.35}$$

  $$B_h^{-1} : \mathbb{W}_h(\tau_h^r) \to \mathbb{V}_h(\tau_h^s), \qquad w_h \mapsto v_h, \text{ such that } \underline{B_h}\,\underline{v_h} \overset{!}{=} \underline{w_h}, \tag{3.1.36}$$

where $\underline{v_h} \in \mathbb{R}^{\dim \mathbb{V}_h(\tau_h^s)}$ and $\underline{w_h} \in \mathbb{R}^{\dim \mathbb{W}_h(\tau_h^r)}$ denote the DoF vectors of $v_h \in \mathbb{V}_h(\tau_h^s)$ and $w_h \in \mathbb{W}_h(\tau_h^r)$, respectively. In addition, the matrix representation of B_h allows to provide a variant of the product acting on discrete functions only:

$$(\cdot,\cdot)_{B_h} : \mathbb{W}_h(\tau_h^r) \times \mathbb{V}_h(\tau_h^s) \to \mathbb{R}, \qquad (w_h, v_h)_{B_h} := \underline{w_h}^\perp \cdot \big(\underline{B_h}\,\underline{v_h}\big). \tag{3.1.37}$$

While less general than the variant acting on localizable functions (since this one also works for discrete functions) this one might be preferable, depending on the circumstances: while the matrix representation requires more storage, it only has to be computed once and can repeatedly be applied to different discrete functions.

For functionals $l_h : Q(\tau_h) \to \mathbb{R}$ or $l_h : \mathbb{V}_h(\tau_h) \to \mathbb{R}$, the situation is similar (though slightly simpler) and we do not discuss it further in this context.

The above considerations on operators and products are reflected by the abstract interface we provide in `dune/gdt/operators/interfaces.hh`, representing an operator, its inverse and its interpretation as a two-form:

```
template< class Traits >
class OperatorInterface
  : public Stuff::CRTPInterface< OperatorInterface< Traits >, Traits >
{
  // not all methods and types shown ...
public:
  template< class SourceType, class RangeType >          // $w_h = B_h[v_h]$
  void apply(const SourceType& source, RangeType& range) const;

  template< class RangeType, class SourceType >          // $(w_h, v_h)_{B_h}$
  FieldType apply2(const RangeType& range, const SourceType& source) const;

  template< class SourceType >
  JacobianType jacobian(const SourceType& source) const;

  template< class RangeType, class SourceType >          // $v_h = B_h^{-1}[w_h]$
  void apply_inverse(const RangeType& range, SourceType& source,
                     const Stuff::Common::Configuration& opts) const;

  std::vector< std::string > invert_options() const; // To be used like types() and options()
                                                      // of a linear solver (see previous
  Stuff::Common::Configuration invert_options(const std::string& type) const; // section).

  // default implemented
  template< class RangeType >
  FieldType induced_norm(const RangeType& range) const
  {
    return std::sqrt(apply2(range, range));
  }
};
```

This templated interface pays respect to the fact that we know little about the source and range of an operator (at least in the sense of C++ types). Again, the CRTP paradigm is a perfect fit for this situation, as it allows us to define what functionality we expect of an operator while allowing the actual argument types of the methods to differ for each implementation. It is clear from the above discussion that not every implementation of an operator will completely fulfill the above `OperatorInterface` and we raise an exception which provides additional information, whenever an operator does not implement one of the methods.

Nevertheless, following the above examples, we identify three archetypes of operators: (*i*) an operator given by its matrix representation, (*ii*) a product acting on localizable functions and (*iii*) a matrix-free (linear or nonlinear) operator acting on localizable or discrete functions. In order to facilitate the implementation of new operators, products and bilinear forms we provide a default implementation for each of the three archetypes in `dune/gdt/operators/default.hh` (the use of which is demonstrated further below):

(*i*) For all operators $\underline{B_h}$, which are given by their matrix representation, we provide the `MatrixOperatorDefault`: for fixed source and range discrete function spaces $\mathbb{V}_h(\tau_h^s)$ and $\mathbb{W}_h(\tau_h^r)$, such operators provide an assembled matrix representation of themselves. They fully comply to the `OperatorInterface` and implement the `apply`, `apply2` and `apply_inverse` methods for discrete functions v_h and w_h and their respective DoF vectors $\underline{v_h}$ and $\underline{w_h}$ (compare Equations 3.1.35, 3.1.36 and 3.1.37).

(*ii*) For all product operators B, which work on arbitrary localizable functions, we provide the `LocalizableProductDefault`: for fixed source and range localizable functions q and r, such operators can be evaluated as a two-form. `LocalizableProductDefault` does not derive from `OperatorInterface` but can be used to implement an operator derived from `OperatorInterface`, which only implements the `apply2` method (compare Equation 3.1.33).

(*iii*) For all matrix-free operators L_h, we provide the `LocalizableOperatorDefault`: for a fixed source function v_h and a range discrete function w_h, such operators can be applied. `LocalizableOperatorDefault` does not derive from `OperatorInterface` but can be used to implement an operator derived from `OperatorInterface` which implements the `apply` method (compare Equation 3.1.32), the `apply2` method (if the source function is a discrete function, compare Equation 3.1.33) and the `apply_inverse` method, if proper generic linear solvers or Newton schemes are available (where we do not yet provide the latter).

The purpose of these default implementations is to greatly simplify the implementation of new operators by providing as much infrastructure as possible. Therefore, the `MatrixOperatorDefault` is derived from `SystemAssembler`, while `LocalizableProductDefault` and `LocalizableOperatorDefault` are derived from `dune-stuff`'s `GridWalker`. In order to implement a new operator, one simply has to derive from the appropriate default implementation and `add` the constituting local operator or two-form.

For instance, we implement the elliptic operator B_h from above (that is given by its matrix representation) in `dune/gdt/operators/elliptic.hh` as follows:

```
template< /*...*/ >
class EllipticMatrixOperator
```

```
3     : public MatrixOperatorDefault< /*...,*/ ChoosePattern::volume >
4   {
5     // not all types shown ...
6     typedef LocalVolumeIntegralOperator
7         < LocalEvaluation::Elliptic< DiffusionFactorType, DiffusionTensorType > >
8             LocalEllipticOperatorType;
9   public:
10    template< class ...Args >
11    explicit EllipticMatrixOperator(const DiffusionFactorType& diffusion_factor,
12                                    const DiffusionTensorType& diffusion_tensor,
13                                    Args&& ...args)
14      : MatrixOperatorDefault< /*...*/ >(std::forward< Args >(args)...)
15      , local_elliptic_operator_(diffusion_factor, diffusion_tensor)
16    {
17      this->add(local_elliptic_operator_);
18    }
19
20  private:
21    const LocalEllipticOperatorType local_elliptic_operator_;
22  };
```

This example shows the minimum amount of boilerplate code which is required to implement new matrix-based operators. We create the local two-form constituting the operators and pass all other arguments to the `MatrixOperatorDefault` base using perfect forwarding. All we have to do is to register the local two-form with the system assembler (line 17), and the base class provides the assembly as well as the full functionality of the `OperatorInterface`. The above `EllipticMatrixOperator` can be created with an existing matrix as well as without. In the latter case, a matrix with the appropriate sparsity pattern is automatically created (since we provide the `volume` tag, see line 3).

In addition, we provide a generator function which further alleviates the user from manually specifying the correct template arguments. Given a discrete function space and data functions, one can obtain an elliptic matrix operator with

```
auto elliptic_operator = make_elliptic_matrix_operator< MatrixType >(diffusion_factor,
                                                                     diffusion_tensor,
                                                                     space);
```

the use of which we demonstrate in the next paragraph.

In addition, we provide an implementation of the elliptic operator as a localizable product (compare Equation 3.1.34), acting on localizable functions:

```
class EllipticLocalizableProduct
  : public LocalizableProductDefault< /*...*/ >
{
  // not all methods and types shown ...
  typedef LocalVolumeIntegralOperator
      < LocalEvaluation::Elliptic< DiffusionFactorType, DiffusionTensorType> >
          LocalEllipticOperatorType;
public:
  template< class ...Args >
  explicit EllipticLocalizableProduct(const DiffusionFactorType& diffusion_factor,
                                      const DiffusionTensorType& diffusion_tensor,
                                      Args&& ...args)
    : BaseType(std::forward< Args >(args)...)
    , local_elliptic_operator_(diffusion_factor, diffusion_tensor)
  {
    this->add(local_elliptic_operator_);
  }

private:
  const LocalEllipticOperatorType local_elliptic_operator_;
};
```

Note that the `EllipticLocalizableProduct` and the `EllipticMatrixOperator` are very similar and mostly differ in the base class they are derived from. In particular, they both use the same local elliptic two-form operator, which is, however, interpreted by the respective base class in a different way: while the `MatrixOperatorDefault` computes local matrices for all test and base functions on all grid elements (and assembles those together into the global matrix), the `LocalizableProductDefault` evaluates the local two-form with the local functions of the prescribed source and range and accumulates the local results.

Given such fixed source and range localizable functions v_h and w_h, the purpose of the `EllipticLocalizableProduct` is to represent the evaluation of the product $(w_h, v_h)_{B_h}$ as an object, the computation of which can be localized by means of `dune-stuff`'s `GridWalker`. The source and range functions are passed on to `LocalizableProductDefault` by means of perfect forwarding, again.

Localizable products can be used to compute several products or norms of localizable functions in a single grid walk. Suppose we are given the exact solution `p` and a discrete approximation `p_h` and want to compute several error norms:

```
// prepare the products
EllipticLocalizableProduct< /*...*/ > energy_product(diffusion_factor, diffusion_tensor,
                                                     grid_view, p - p_h, p - p_h);
L2LocalizableProduct< /*...*/ >       l2_product(grid_view, p - p_h, p - p_h);
// use the first as grid walker
energy_product.add(l2_product);  // register the second for computation
energy_product.walk();           // compute both in one grid walk
// access the results
double energy_error = energy_product.induced_norm();
double l2_error     = l2_product.induced_norm();
```

This is the computationally most efficient way to compute several products in one grid walk. If we are only interested in a single product, however, the specification of the `energy_product`, for instance, is not very intuitive. We thus additionally provide a generic `EllipticOperator` in `dune/gdt/operators/elliptic.hh`, which derives from `OperatorInterface` and implements `apply`, `apply2` and `apply_inverse` by redirecting the computation to the `EllipticMatrixOperator` or `EllipticLocalizableProduct`, depending on the method in question and its arguments. Using this generic operator (and a generator method as discussed above) we can obtain the same error norm in a more intuitive way:

```
auto elliptic_operator = make_elliptic_operator(diffusion_factor, diffusion_tensor,
                                                grid_view);
double energy_error = elliptic_operator->induced_norm(p - p_h);
```

The `LocalizableOperatorDefault` can be used in a similar way to form any matrix-free operator. For instance, we realize the L^2 projection operator from Definition 3.1.40 by deriving from `LocalizableOperatorDefault` and using the `LocalL2ProjectionOperator` presented above.

This finalizes our discussion of how we implement global operators and two-forms in `dune-gdt`. Of course, we also provide similar default implementations for functionals.

System tests: a full example of discretizing a linear elliptic PDE. With `dune-gdt`, we provide and ship an extensive test suite in `dune-gdt/tests` (based on the googletest framework[40]). We provide unit tests for all spaces, products and operators as well as system tests. Given predefined grid and problem descriptions, the latter compute a series of approximate solutions, the associated discretization errors in several norms and the corresponding EOCs (compare Section 3.1.1.2). The resulting error- and EOC-values are then compared to an internal database to ensure the quality of the approximation.

To finalize the presentation of `dune-gdt`, we provide the actual code that is used in `dune/gdt/tests/linearelliptic/discretizers/cg.hh` in order to compute a continuous Galerkin FE approximation of a linear elliptic PDE:

```
using namespace Dune::Stuff::Grid;

static DiscretizationType discretize(ProviderInterface< GridType >& grid_provider,
                                     const ProblemType& problem,
                                     const int level = 0)
{
  // create discrete function space
  auto space = Spaces::CGProvider< /*...*/ >::create(grid_provider, level);
  auto boundary_info = BoundaryInfoProvider< /*...*/ >::create(problem.boundary_info_cfg());
  // define all operators and functionals
  auto elliptic_operator
      = make_elliptic_matrix_operator< MatrixType >(problem.diffusion_factor(),
                                                    problem.diffusion_tensor(),
                                                    space);
  auto l2_force_functional
      = make_l2_volume_functional< VectorType >(problem.force(), space);
  auto l2_neumann_functional                                       // Assemble both
      = make_l2_boundary_functional(problem.neumann(),             // functionals into
                                    l2_force_functional->vector(), // the same vector.
                                    space,
                                    new ApplyOn::NeumannIntersections< /*...*/ >(*boundary_info));
  // prepare the dirichlet projection and constraints
  auto dirichlet_function
      = make_discrete_function< VectorType >(space, "dirichlet values");
  auto dirichlet_projection
      = Operators::make_localizable_dirichlet_projection(space.grid_view(),
```

[40]https://code.google.com/p/googletest/

```
27                                                          *boundary_info,
28                                                          problem.dirichlet(),
29                                                          dirichlet_function);
30    Spaces::DirichletConstraints< /*...*/ >
31        dirichlet_constraints(*boundary_info, space.mapper().size());
32    // register everything for assembly in one grid walk
33    SystemAssembler< SpaceType > assembler(space);
34    assembler.add(*elliptic_operator);
35    assembler.add(*l2_force_functional);
36    assembler.add(*l2_neumann_functional);
37    assembler.add(dirichlet_projection);
38    assembler.add(dirichlet_constraints);
39    assembler.assemble();
40    // assemble the dirichlet shift
41    auto& system_matrix = elliptic_operator->matrix();
42    auto& rhs_vector = l2_force_functional->vector();
43    auto& dirichlet_shift = dirichlet_function.vector();
44    rhs_vector -= system_matrix * dirichlet_shift;
45    dirichlet_constraints.apply(system_matrix, rhs_vector);
46    // create the discretization (no copy of the containers done here, bc. of cow)
47    return DiscretizationType(problem, space, system_matrix, rhs_vector, dirichlet_shift);
48  }
```

This static method is the main methods of the `CGDiscretizer` and returns a stationary discretization object, which holds all assembled containers and provides a `solve` method. The `CGDiscretizer` is instantiated (and `discretize` and `solve` are called) for all available combinations of discretization and linear algebra backends; the appropriate tags are passed on to `Spaces::CGProvider` (line 7, not shown) and `Stuff::LA::Container` (also not shown). Note (again), that we are thus able to switch between different backends for the linear algebra containers and linear solvers as well as for the discrete function spaces by just passing on two compile-time constants, yielding the above generic code that works for all implementations.

This finalizes the demonstration and discussion of our discretization framework, which we utilize for all high-dimensional computations in the context of elliptic parametric multiscale problems. The remainder of this chapter gives a presentation and discussion of our model reduction software framework.

3.2 Model reduction framework

Similar to the presentation of our discretization framework in the previous section, we begin the discussion of our designated model reduction framework by revisiting our requirements. Following our earlier work [MRS2015] we continue with a discussion of existing model reduction frameworks and motivate the design of our framework, which mainly consists of `pyMOR`. We finish this chapter with a presentation of `pyMOR`'s architecture and implementation and a brief presentation of our related software packages, `dune-pymor` and `dune-hdd`.

3.2.1 Requirements

We refer to Sections 1.3 for a presentation of the reduced basis (RB) method and briefly summarize the required building blocks in view of their relevance for a model reduction software framework.

The reduced basis method is by nature a very generic model reduction technique which can be applied to a wide range of high-dimensional problems. It is an important feature of RB methods that existing high-dimensional discretizations can be used (and are usually required) in order to derive a reduced order model: any discretization (in particular those discussed in Section 3.1.1.1) can be used as long as the high-dimensional problem is of an appropriate form, such as

- in Definition 1.3.4 for stationary linear problems; or
- in the form of: for $\boldsymbol{\mu} \in \mathcal{P}$ find $p_h(\boldsymbol{\mu}) \in Q_h$, such that

$$B_h[p_h(\boldsymbol{\mu}); \boldsymbol{\mu}] = l_h \qquad \text{in } Q_h \tag{3.2.1}$$

  for stationary nonlinear problems; or
- in the form of: for $\boldsymbol{\mu} \in \mathcal{P}$ find $p_h(t; \boldsymbol{\mu}) \in Q_h$, such that

$$\begin{aligned} < \partial_t p_h(t; \boldsymbol{\mu}), q_h > + B_h[p_h(t; \boldsymbol{\mu}); \boldsymbol{\mu}](q_h) &= l_h(q_h), && \text{for all } q_h \in Q_h \text{ and} \\ p_h(0; \boldsymbol{\mu}) &= p_{0,h}(\boldsymbol{\mu}) && \text{for some } p_{0,h}(\boldsymbol{\mu}) \in Q_h, \end{aligned} \tag{3.2.2}$$

  for instationary problems,

where Q_h denotes a high-dimensional discrete function space, $l_h \in Q_h'$ denotes a linear functional and $B_h : \mathcal{P} \to [Q_h \to Q_h']$ denotes a parametric operator.[41]

While high-dimensional discretizations are essential to model reduction and while the reduced scheme only depends on the form of the above problem (not its implementation), the code implementing the high-dimensional discretization nearly always has to be adapted for model order reduction. We thus briefly summarize which main operations are needed when implementing reduced basis schemes.

[41]The two problems sketched here are of course not mathematically complete, can be further extended (by additional parameter and time-dependencies) and are only meant to be illustrating.

3.2.1.1 High-dimensional operations

Going through the reduction process as laid out in Section 1.3, the following operations associated with the high-dimensional discretization have to be performed:

Computation of solution snapshots $p_h(\boldsymbol{\mu})$: During any variant of the greedy basis generation (Algorithm 1.3.10), solution snapshots $p_h(\boldsymbol{\mu})$ need to be computed for certain training parameters $\boldsymbol{\mu} \in \mathcal{P}_{\text{train}}$ selected by the algorithm. Since these parameters are not known a priori, either the high-dimensional solver has to stay initialized in memory during the whole offline phase, or the solver has to be re-initialized (grid generation, matrix assembly, etc.) for each new solution snapshot.

Basis extension: To ensure the numerical stability of the reduced model, system matrices need to be assembled with respect to an orthonormal basis of Q_h. For greedy basis generation, a stabilized Gram-Schmidt algorithm is the standard choice for orthonormalization, as it generates hierarchical bases which are compatible with the nesting of the reduced spaces $Q_{\text{red}}^{(0)} \subset Q_{\text{red}}^{(1)} \subset \cdots$. For instationary problems such as (3.2.2), the solution trajectory has to be orthogonally projected onto the current reduced space Q_{red} and a proper orthogonal decomposition (POD) of the projection errors has to be computed.

Assembly of reduced system matrices: To assemble the reduced Problem 1.3.6, the high-dimensional bilinear form b_h and the right-hand side l_h have to be evaluated for each (combination of) basis vector(s) of Q_{red}. The implementation needs to be aware of the affine decomposition 1.3.7 in order to achieve offline/online decomposition.

Assembly of reduction error estimator: In order to estimate the model reduction error (see Section 2.3.1), all scalar products between Riesz representatives of the components of the residual $r_{\varepsilon,h}[\cdot;\boldsymbol{\mu}]$ with respect to the affine decomposition 1.3.7 need to be computed. This standard approach for offline/online decomposition of the residual norm shows bad numerical stability, however. An improved algorithm [BEOR2014] requires the computation of an orthonormal basis for the span of the Riesz representatives of the residual components and the computation of the coefficients of these components with respect to the orthonormal basis.

Generation of empirical interpolation data: For the reduction of nonlinear problems such as (3.2.1), we need to perform an offline/online decomposition according to [DHO2012] or [MRS2015, Section 2.4.3]. Therefore, we have to evaluate the nonlinear operator at appropriately selected solution snapshots, generate a collateral basis and interpolation points using the EI-Greedy [HOR2008] or DEIM [CS2010] algorithm and finally compute projections and evaluate restricted operators (see the above mentioned references).

Reconstruction and visualization: In many cases we want to be able to access the reduced solutions $p_{\text{red}}(\boldsymbol{\mu})$ of the reduced Problem 1.3.6 as elements of the high-dimensional discrete function space Q_h, to perform visualizations or to compute quantities of interest not available in the reduced model. Therefore, a reconstruction as a linear combination of the reduced basis functions has to be computed. Additional discretization data such as the grid used for discretizing will be needed for visualization.

3.2.1.2 Low-dimensional operations

The following operations within reduced basis schemes are low-dimensional in nature and do not depend on high-dimensional data:

Solution of reduced problem: For the solution of the reduced Problem 1.3.6, the reduced system matrix needs to be assembled for a given parameter $\boldsymbol{\mu}$ and a dense linear solver is invoked to determine the coefficient vector $\underline{p_{\mathrm{red}}(\boldsymbol{\mu})}$. For instationary problems such as (3.2.2), a time-stepping scheme is additionally required.

Error estimation: Given a coefficient vector $\underline{p_{\mathrm{red}}(\boldsymbol{\mu})}$ the norm of the residual has to be evaluated using the pre-computed data.

greedy basis generation: While many high-dimensional operations have to be performed during the offline phase, the logic of the main greedy search loop driving these operations is low-dimensional in nature. While a greedy Algorithm such as 1.3.11 is quite simple and easily implemented, much more complicated search strategies, for instance using an adaptive parameter training set $\mathcal{P}_{\mathrm{train}}$ or dictionaries of reduced bases for different parameter space regions, are conceivable and are used in practice (see for instance [HDO2011]).

Interpolated operator evaluation: When empirical interpolation is used, interpolated operators have to be evaluated, in the linear case for the assembly of the system matrix, in the nonlinear case to compute the residual inside a Newton algorithm (see [MRS2015]). This does in particular involve the evaluation of a restricted operator, which therefore needs to be available during the online phase. For Newton schemes, access to the Jacobian of the restricted operator is additionally required.

3.2.2 Existing implementations

As discussed, implementations of reduced basis schemes involve several building blocks, which we have categorized by whether they involve direct manipulation of high-dimensional data or not. This naturally leads to the following three basic designs for reduced basis software: 1. Implement the whole reduced basis scheme as an individual software package, loading high-dimensional solution snapshots and system matrices from disk. 2. Implement the whole reduced basis scheme as a ‘model reduction mode’ of the high-dimensional solver software. 3. Only implement low-dimensional building blocks as a separate model reduction software package and communicate via interfaces with the high-dimensional solver, which performs the high-dimensional operations.

We discuss the main advantages and disadvantages of these designs, before presenting the new approach which has been taken by `pyMOR`.

3.2.2.1 Approach 1: Separate software

Maybe the most simplistic design of reduced basis software is to implement the whole reduced basis machinery as a single software package which is able to read and process high-dimensional data which has been produced by some external high-dimensional solver. All high-dimensional operations (Section 3.2.1.1) within the offline phase are carried out by the reduced basis software, with the exemption of the solution snapshot computation which may be performed by the solver. A typical example of such a software is the `rbMIT` [PR2006] package for `Matlab`.

Advantages: 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.

Disadvantages: Implementing all high-dimensional operations within the model reduction software means that the software has to be able to efficiently work with the high-dimensional data produced by the external solver. For instance, this means that matrix-vector products for the (usually) sparse matrix format at hand have to be computed for the assembly of the reduced system matrices, and a linear solver for the computation of the residual 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 nonlinear problems is problematic as well: since evaluating the high-dimensional nonlinear operator inside the reduced basis software would amount to a more or less complete re-implementation of the high-dimensional discretization, evaluations of the operator for certain solution snapshots have to be produced by the high-dimensional solver. However, the reduced basis software still needs to be able to evaluate the restricted operators. Thus the restricted operator has to be re-implemented within the reduced basis software. This is not only a duplication of effort but also only possible if the exact details of the high-dimensional discretization are known. Alternatively, the restricted operator has to be provided by the external solver in some way, which however does not fit the paradigm of this approach very well.

3.2.2.2 Approach 2: Inside high-dimensional solver

In this approach the complete reduction process is carried out by the high-dimensional solver which has been extended by a reduced basis module. Examples for this approach are the reduced basis implementations of the `libMesh` [KP2011] and `feel`++ [DVTP2013] PDE solver packages.

Advantages: Implementing all reduction algorithms as part of the high-dimensional solver offers the tightest possible integration between the high-dimensional model and the reduced basis code. This allows for maximum performance of the implementation and easy development of more advanced reduction techniques, which might require special operations on the high-dimensional data. A single code base also eliminates any interoperability issues between different versions of the model reduction and the solver code.

Disadvantages: An obvious consequence of this design is that, despite the generality of reduced basis methods, implemented algorithms can only be used within a specific software ecosystem. Given the large number of PDE solver software libraries developed by research groups and software vendors around the world, this vastly diminishes the reusability of the code and ultimately hinders collaboration between researchers. 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 only evaluated for ad hoc implementations of academic ‘toy problems’.

3.2.2.3 Approach 3: Separate low- and high-dimensional operations

In this approach, as a compromise between the aforementioned ones, all low-dimensional operations (Section 3.2.1.2) are implemented as a reduced basis software package which communicates over well-defined software interfaces with the high-dimensional solver carrying out all high-dimensional operations (Section 3.2.1.1). This approach has been pursued by the integration of `RBmatlab` with the `dune-rb` module of the **DUNE** numerics environment [DHKO2012].

Advantages: This interface-based approach shares many benefits of the previous two approaches. All low-dimensional algorithms can be developed independently from the high-dimensional solver in a programming language of choice and these algorithms can be integrated with any external solver implementing the necessary interfaces. Since all high-dimensional operations are carried out by the PDE solver, the size of the high-dimensional model is only limited by the performance of the solver’s implementation. Memory distributed or matrix-free implementations can be easily utilized.

Disadvantages: To fulfill the interfaces required by the model reduction software, external solvers have to be extended to implement the high-dimensional operations defined in Section 3.2.1.1. Apart from the computation of solution snapshots, all these operations are model reduction specific. Thus, implementing these interfaces requires substantial work on the solver side which can only be done with some background on reduced basis methods. Moreover, apart from certain special cases, a change of the reduction algorithm will almost always require modification of the reduced basis code in the high-dimensional solver. (Think of switching from Galerkin projection to a Petrov-Galerkin approach as in [DPW2014].) This can be a major issue when the high-dimensional solver is developed by a different team than the group implementing the model reduction algorithms, in particular for research projects where both the PDE solver and the reduction algorithm are under constant development. Moreover, as with design approach 1, the evaluation of restricted nonlinear operators is problematic: either the nonlinear operator has to be re-implemented in the reduced basis software, or the restricted evaluations have to be performed by the PDE solver which breaks the paradigm of separating high- and low-dimensional operations between the two software packages.

3.2.3 Design principles

The main goal for the development of `pyMOR` is to create a model reduction software library that serves as a flexible tool for scientists in education and research which, at the same time, can be effectively used for the reduction of large real-world application problems.

`pyMOR`'s design is based on the central observation that all high-dimensional operations during the offline phase of reduced basis schemes (Section 3.2.1.1) can be expressed using only a small set of basic operations on *operators*, (collections of) *vectors* and the *discretization*, which are independent from the concrete reduction scheme in use. This observation leads to the following basic design paradigm for `pyMOR`:

1. Define model reduction agnostic interfaces for the mathematical objects involved with reduced basis methods and related schemes.

2. Generically implement all algorithms through operations provided by these interfaces.

We shall discuss `pyMOR`'s interfaces and how the high-dimensional reduction operations can be implemented via these interfaces in Section 3.2.4.1 in more detail. For the remainder of this section, we consider the general consequences of the design approach.

As with design approach 1, `pyMOR` contains all model reduction code as a separate self-contained software package, allowing us to choose an implementation language which optimally fits our requirements. For `pyMOR` we chose the `Python` scripting language to make `pyMOR` an easy to pick up tool for rapid algorithm development. In addition, `pyMOR` provides its own basic discretization toolkit to easily test new reduction algorithms. The data types provided by the toolkit can also be used to import high-dimensional data from disk which have been produced by an external solver. This allows for `pyMOR`'s algorithms to be used in a workflow similar to design approach 1.

The interface-based design of `pyMOR` allows to integrate `pyMOR`'s algorithms with any high-dimensional solver implementing these interfaces. As with design approach 3, doing so requires additional work on the solver side. However, a major benefit of the generic nature of `pyMOR`'s interfaces over design approach 3 is that no model reduction specific code has to enter the high-dimensional solver, allowing to develop the model reduction algorithms independently from the solver. This, for instance, allows a workflow where for a given high-dimensional problem the reduction algorithm can be effortlessly developed and tested using a low-fidelity version of the high-dimensional model implemented with `pyMOR`'s own discretization toolkit, before the very same algorithms are used to reduce the actual high-dimensional model implemented in a high-performance solver running on a large computer cluster.

In addition, our approach offers strongly improved code reusability and maintainability since the complete reduction algorithms can be implemented in a single software library. This is particularly important when the model reduction library is used in conjunction with different PDE solver ecosystems.

`pyMOR`'s interfaces correspond to data structures which are present in most PDE solver designs: operators, vectors and discretizations. Thus, implementing `pyMOR`'s interfaces is basically equivalent to exposing the solver's internal data structures via a public API. Refactoring a PDE solver to offer such an API has many benefits apart from allowing model order reduction, as it enables the user to easily use and extend the solver in new ways which have not been envisioned by the developers of the solver. For instance, `pyMOR` implements time-stepping algorithms via its interfaces which can be used to easily create instationary discretizations out of stationary discretizations (see [MRS2015, Section 5.3]). While implementing time-stepping schemes is clearly not a focus of `pyMOR`'s development, a software library of advanced time-stepping algorithms could use such an API to allow for easy testing of these algorithms with a given discretization, after which a selected algorithm might be implemented in the solver for maximum performance.

Apart from such opportunities for extending the solver, a public API also allows to interactively control the solver, especially when used in conjunction with dynamic languages such as `Python`. Such interactive sessions can be a powerful tool for debugging and allow to inspect and modify the solvers state in ways, which are 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 instance, `pyMOR` makes it natural to perform preliminary analyses of models which have been reduced only partially and still require high-dimensional operations for solving, e.g., a reduced basis projection of problems for which an affine decomposition is not yet available. In particular, using the PDE solver for a restricted evaluation of nonlinear operators is a natural option in our design.

While design approaches 2 and 3 allow to perform all high-dimensional reduction operations with the maximum efficiency the high-dimensional solver has to offer, it is clear that `pyMOR`'s interface design comes at a price of sub-optimal 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 asses the possible loss in performance, we conduct several performance benchmarks (Section 4.3) which show that, while a certain overhead exists, this overhead usually becomes negligible for large enough problems. In the rare cases where `pyMOR`'s generic algorithms might not perform sufficiently well for a given problem, the possibility still remains to re-implement the critical parts inside the solver.

While we have designed `pyMOR`'s interfaces with model reduction in mind, we are convinced that there should be a general paradigm shift towards a more modular, library oriented design of high-performance PDE solvers, allowing non-expert scientists to use these solvers more easily and in new ways, using a programming language which suits their needs. We see `pyMOR` as one step in this direction.

3.2.4 A new model reduction framework

As discussed in Section 3.2.2, none of the existing model reduction frameworks provided the functionality we required (compare Section 3.2.3), in particular since we provide our own discretization framework. We thus provide our own implementation of a model reduction framework, which mainly consists of `pyMOR` and the support packages `dune-pymor` and `dune-hdd`:

`pyMOR` forms the heart of our model reduction framework: it contains all required interfaces and generic algorithms, based on the design principles from Section 3.2.3. It also contains vectorized grids and discretizations itself, but one of its purposes is to be coupled with existing discretization frameworks.

`dune-pymor` introduceds the concepts of parameter dependency and affine decomposition, which are required to treat parametric problems in the sense of Section 1.3, to the C++ discretization framework from Section 3.1. It provides parametric variants of functions from `dune-stuff` and operators from `dune-gdt` as well as affinely decomposed variants of the containers from `dune-stuff` and the operators form `dune-gdt`. In addition, it provides `Python` bindings for these objects. This allows to access most parts of the discretization framework within an interactive `Python` session, providing a low entry point for users as well as powerful interactive debugging for developers. It also enables our discretization framework to be used by other software packages such as `pyMOR`, for instance in the context of model reduction.

`dune-hdd` bundles the other modules together and provides ready to use parametric discretization objects, which are required for model reduction with `pyMOR`.

In the following sections, we give a brief presentation of these software packages.

3.2.4.1 pyMOR

The Python package pyMOR is open source software and freely available: http://pymor.org. It is mainly developed by R. Milk, S. Rave and F. Schindler with contributions from A. Buhr, M. Laier, P. Mlinaric and M. Schaefer.[42]

In this section we present the architecture of pyMOR in more detail. We demonstrate how high–dimensional model reduction operations (see Section 3.2.1.1) can be expressed via pyMOR's interfaces and cover the parallelization of pyMOR's reduction algorithms. First of all, however, we discuss our reasons for choosing Python as the implementation language.

Implementation. One of pyMOR's main design goals is to provide an easy to use library of tools for the research of new model order reduction algorithms. It was therefore clear to us to choose for pyMOR a managed, dynamically typed language such as Python, which is easy to pick up (even for unexperienced programmers) and in which the user does not have to care about memory management or data types during his or her daily work.

In contrast to Matlab, Python does not have copy-on-write semantics for assignment, which allows for more precise control over data, but often raises the issue of object ownership. To alleviate the user from having to care too much about ownership, pyMOR enforces immutablility on all Discretizations and Operators, as well as all classes of pyMOR's own discretization toolbox. In combination with Python's dynamic memory management, this makes the question of ownership irrelevant for these objects.[43] While Python is designed as a general purpose language, the NumPy [Oli2007] package offers a multi-dimensional array class with a very similar feature set and performance as Matlab matrices. Many numerical algorithms and several sparse matrix types can be found in the SciPy [JOPo2001] package. Both packages are used extensively in pyMOR for all low–dimensional operations 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[44] or some custom protocol. 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 spirit of the tight coupling between pyMOR and the external solver as promoted by our design, we favour, whenever possible, to integrate the solver by recompiling it as a Python extension module, which gives Python direct access to the solver's data structures. This design not only delivers maximum performance as no

[42]Regarding the contributions in pyMOR: the design is based on discussions which mainly involve R. Milk, S. Rave and F. Schindler, while S. Rave is responsible for most of the implementation.

[43]Since VectorArrays are mutated very often in pyMOR, enforcing immutability on VectorArrays is not feasible. However, we plan to enforce deep-copy-on-write semantics for the copy operation as part of the interface in future version of pyMOR.

[44]http://xmlrpc.scripting.com/

communication overhead is present, it also allows to directly manipulate the solvers state from within Python beyond the operations available via pyMOR's interfaces. This makes it possible to quickly extend or modify the solver's behavior using NumPy or SciPy for debugging or the exploration of new ideas, and gives the user an interactive Python debugging shell with direct access to the solver's memory.

Again, Python's long-standing tradition as a glue language vastly facilitates this approach with a wide array of tools for generating Python bindings for foreign language libraries. We took this approach in [MRS2015, Sections 5.2 and 5.3] and showed its feasability, even for MPI-distributed solvers running on high-performence 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 [ORSZ2014].

Finally, we note that choosing arrays of vectors instead of single vectors as elementary objects in pyMOR not only allows to concisely express many model reduction operations in a vectorized manner (see the examples below), it also makes vectorized VectorArray implementations possible, which optimally exploit the vectorized structure of the operation for increased performance. An example of such an implementation is pyMOR's NumpyVectorArray (see the benchmarks in Section 4.3). However, since few external solvers use such data structures, pyMOR also ships a generic ListVectorArray which manages a Python list of vector objects provided by the external solver.

Algorithms and Interfaces. From a bird's eye perspective, pyMOR can be seen as a collection of generic algorithms operating on VectorArray, Operator and Discretization objects. These objects are expected to be instances of subclasses of the abstract base classes VectorArrayInterface, OperatorInterface and DiscretizationInterface, which define methods each derived object is expected to provide. To integrate pyMOR with an external high–dimensional solver, wrapper classes for these types have to be implemented (as discussed above) to represent high–dimensional objects inside the solver, which can be manipulated through the interface methods implemented by these classes.

It is an important property of pyMOR's interfaces that each interface method returns either 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. Instead, all high–dimensional manipulations can be performed by specialized routines already implemented inside the solver.

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 using data structures and solvers provided by the NumPy package. This allows 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.

A full documentation of pyMOR's interface classes can be found online[45]. We present

[45]http://docs.pymor.org/

some of their methods by indicating how the operations of Section 3.2.1.1 can be expressed through the interfaces. In the following paragraphs, the variables `V`, `o`, `d` will always represent `VectorArray`, `Operator` and `Discretization` objects, respectively.

Computation of solution snapshots $p_h(\boldsymbol{\mu})$: Each high–dimensional (or low-dimensional) model is represented by a `Discretization` object `d`. We obtain a solution snapshot $p_h(\boldsymbol{\mu})$ by executing

```
V = d.solve(mu)
```

which returns for a given parameter $\boldsymbol{\mu}$ a `VectorArray` `V` containing the solution $p_h(\boldsymbol{\mu})$.

`VectorArrays` are ordered collections of vectors of the same dimension. For a stationary problem, we have `len(V) == 1`, whereas for instationary problems `V` usually contains multiple vectors forming the discrete solution trajectory.

Basis extension: Reduced bases are stored inside `VectorArrays`. To extend a basis `RB` by a new solution snapshot contained in `V`, we could call

```
RB.append(V, remove_from_other=True)
```

which moves the vector(s) in `V` to the end of `RB`. However, in order to guarantee numerical stability, `RB` has to be orthonormalized, for instance using Gram-Schmidt orthonormalization (Algorithm 3.2.1). In this algorithm, scalar products are computed using the `V.dot(other, ind, o_ind)` method, which returns a matrix of all scalar products between vectors in `V` and `other`. The optional `ind` and `o_ind` parameters can be numbers or lists of numbers specifying the vectors in `V` and `other` on which to operate. (`ind` parameters can be passed to most methods of `VectorArrays` and `Operators`.)

Algorithm 3.2.1 Simplified version of the stabilized Gram-Schmidt orthonormalization algorithm contained in `pyMOR`.

```
def gram_schmidt(V, offset=0):
    for i in range(offset, len(A)):
        for j in range(i):
            alpha = V.dot(V, ind=i, o_ind=j)[0, 0]
            V.axpy(-alpha, V, ind=i, x_ind=j)
        norm = V.l2_norm(ind=i)[0]
        V.scal(1./norm, ind=i)
```

`pyMOR` provides an extended version of Algorithm 3.2.1 which includes re-orthonormalization of vectors for improved numerical accuracy, absolute and relative tolerances to detect linear dependent vectors and an additional orthonormality check for the resulting array (see `pymor.algorithms.gram_schmidt`). Moreover, it is possible to provide a scalar product with respect to which to orthonormalize, given as an `Operator`. The scalar product `alpha` is then obtained as

```
alpha = product.apply2(V, V, U_ind=i, V_ind=j)[0, 0].
```

In general, every operator has an `apply` method which returns (in the parametric case for a given parameter `mu`) the applications of the operator to the vectors of a provided `VectorArray` as a new `VectorArray`. Calling `o.apply2(V, U, U_ind, V_ind)` is then equivalent to

```
V.dot(o.apply(U, ind=U_ind), ind=V_ind)
```

as it is default implemented in `pymor.operators.basic.OperatorBase`.

Algorithm 3.2.2 Simplified version of the POD algorithm contained in `pyMOR`.

```
def pod(V, modes):
    a = V.gramian()

    evals, evecs = eigh(a, eigvals=(len(V) - modes, len(V) - 1))
    singular_values = np.sqrt(evals)

    return V.lincomb((evecs / singular_values).T)
```

An alternative orthonormalization procedure is given by *proper orthogonal decomposition* (POD). `pyMOR` includes a POD algorithm based on the so called 'method of snapshots' (Algorithm 3.2.2). In this algorithm, the Gramian of the vectors in `V` is computed using the `gramian` method which is default implemented as `V.dot(V)`. Then the `SciPy` method `eigh` is used to compute the first `modes` vectors of the eigenvalue decomposition of the Gramian. Finally, the POD modes are computed by forming the linear combinations of the vectors in `V` with the Gramian's eigenvectors as coefficients, scaled by its singular values. Note that the size of the eigenproblem is given by the number of vectors in `V` which we always assume to be small in relation to their dimensions, making this low–dimensional problem easy to solve (compare the benchmarks in Section 4.3).

Again, `pyMOR`'s actual implementation of Algorithm 3.2.2 (`pymor.algorithms.pod`) is numerically more robust, allows to choose the number of computed POD modes based on relative and absolute tolerances for the singular values and allows to compute the POD with respect to arbitrary scalar products.

Assembly of reduced system matrices: The reduced basis projection (compare Section 1.3.2.1) of bilinear forms (or operators) in `pyMOR` is performed by calling the operator's `projected` method

```
o.projected(range_basis=RB, source_basis=RB)
```

which returns a new `Operator` representing the projected bilinear form (operator). For linear, nonparametric `Operator`s, this method can be default implemented as

```
NumpyMatrixOperator(o.apply2(range_basis, source_basis))
```

Here, `NumpyMatrixOperator` is an `Operator` provided by `pyMOR`, which implements a thin wrapper around `NumPy` arrays and `SciPy` sparse matrices in order to fulfill `pyMOR`'s `Operator` interface. `NumpyMatrixOperators` can be applied to `NumpyVectorArrays` which is a vectorized `VectorArray` implementation, again based on `NumPy` arrays.

Affinely decomposed bilinear forms (compare Definition 1.3.7) are represented in pyMOR as instances of LincombOperator which hold lists of the Operators b_ξ and the ParameterFunctionals θ_ξ. The projected method for LincombOperator is then simply implemented as

```
LincombOperator([o.projected(range_basis, source_basis) for o in self.operators],
                self.coefficients)
```

constructing a new LincombOperator holding the projected summands with the same coefficient functions as the original operator.

Note that LincombOperators can hold arbitrary Operators as summands. In particular, if such an operator itself contains LincombOperators as summands, the reduced basis projection is performed recursively, yielding automatic offline/online decomposition of arbitrary nested trees of operators.

To keep the amount of interface classes in pyMOR as small as possible, Operators also represent functionals. Such operators have the special property that they map Vector-Arrrays (of an arbitrary fixed type) to NumpyVectorArrays of dimension 1 which hold the functional evaluations. For an operator representing a functional, the reduced basis projection is obtained by calling

```
o.projected(range_basis=None, source_basis=RB).
```

In the same spirit, parameter dependent vectors are represented by linear operators mapping one–dimensional NumpyVectorArrays to VectorArrays of appropriate type, such that 1 maps to the vector the Operator represents. Vectors are (usually) projected orthogonally onto Q_{red} which is for such vector-operators o achieved by invoking

```
o.projected(range_basis=RB, source_basis=None, product=p),
```

where p represents the Q_h-scalar product. For vector-like operators, this call to projected returns a new vector-like operator that, for a parameter mu, maps 1 to

```
NumpyVectorArray(p.apply2(v.apply(NumpyVectorarray([1]), mu), RB)),
```

which, assuming RB is p-orthogonal, is the coefficient vector of the p-orthogonal projection of the vector onto the linear span of RB.

Note that pyMOR provides a default implementation for projected which allows to project any given Operator in pyMOR. While for arbitrary parametric (or nonlinear) Operators no offline/online decomposition is performed, this allows to easily test the approximation quality of the reduced model before further steps for offline/online decomposition are taken.

pyMOR's Discretizations hold dictionaries of all operators, functionals, vectors and scalar products which appear in the definition of the discrete problem. As the reduced basis projection does not change the structure of the problem but merely replaces high–dimensional objects by their respective projected low–dimensional counterparts, the reduced model for a standard reduced basis approximation can be computed generically using Algorithm 3.2.3. In this algorithm, the with_ method of the Discretization

Algorithm 3.2.3 Generic reduced basis projection of arbitrary discretizations (simplified).

```
def reduce_generic_rb(d, RB, product=None):
    p_o = {k: o.projected(RB, RB)
           for k, o in d.operators.items()}
    p_f = {k: f.projected(None, RB)
           for k, f in d.functionals.items()}
    p_v = {k: v.projected(RB, None, product=product)
           for k, v in d.vector_operators.items()}
    p_p = {k: p.projected(RB, RB)
           for k, p in d.products.items()}
    return d.with_(operators=p_o, functionals=p_f, vector_operators=p_v, products=p_p)
```

`d` is called to return a new (reduced) `Discretization` in which the high–dimensional `Operators` have been replaced by the provided projected `Operators`.

`pyMOR` provides generic `Discretization` classes for stationary and instationary problems which implement `solve` by operations on the `Operators` they contain. For these `Discretizations`, the reduced problem can be described by an instance of the very same class. `Discretizations` which call a specialized `solve` method of the external solver are usually implemented as a subclass of one of these classes, and the `with_` method is adapted to return an instance of the base class.

Assembly of reduction error estimator: In order to evaluate the estimator for the model reduction error, Riesz representatives of the affine summands of the residual have to be computed with respect to a scalar product (compare Section 2.3.1). If this product is given by the operator `p`, the Riesz representatives of vectors in `V` can be obtained by calling

```
p.apply_inverse(V)
```

which gives access to linear (and possibly nonlinear) solvers for the `Operator p` and right-hand side `V`.

`pyMOR` includes a generic, numerically stable algorithm for the assembly of residual-based error estimators based on [BEOR2014], which (recursively) computes the residual component Riesz representatives (for a given reduced basis), determines a `p`-orthonormal basis for the vectors and then projects the residual operator onto this basis (`pymor.reductors.residual`). This algorithm can be used in conjunction with any operator for which an appropriate estimate of its range when applied to a set of vectors is known. In particular, this includes empirically interpolated operators for which the range is given by the span of the collateral basis (compare [MRS2015, Section 2.4.3]).

Generation of empirical interpolation data: For the computation of the interpolation points and the collateral basis, which are required for the empirical interpolation of `Operators` (compare [MRS2015, Section 2.4.3]), `pyMOR` provides two commonly used, similar algorithms: EI-Greedy [HOR2008] and DEIM [CS2010] (`pymor.algorithms.`

`ei`). These algorithms require access to individual entries of the high–dimensional vectors at indices selected such that the absolute values of the entries of a certain vector are maximized. Both operations can be supported by `pyMOR`'s `VectorArray`s by implementing

```
V.components(component_indices)
```

for vectorized extraction of the entries at the given list of indices and

```
V.amax()
```

to obtain the maximizing indices. Note that `len(component_indices)` is expected to be small. Thus, the `components` method is not intended for communication of high-dimensional data.

Reconstruction and visualization: A high–dimensional reconstruction of the solution vector `V` from a reduced coefficient vector `v` is easily performed by executing

```
V = RB.lincomb(v.data).
```

Here, the `data` property of `NumpyVectorArray` is used to extract the actual (low-dimensional) array data from the `VectorArray` to pass it into the `lincomb` method. To allow more complex usage scenarios, reduction algorithms in `pyMOR` return a reconstructor class `rc` along with the reduced discretization which allows to obtain `V` by calling

```
V = rc.reconstruct(v).
```

In the basic setting presented here, `rc` simply holds a reference to `RB` in order to perform the `lincomb` call, but more complex reconstructors are conceivable.

Visualization of solution vectors may be supported by the high–dimensional solver by implementing `d.visualize(V)`.

Parallelism in `pyMOR`. Reduced basis methods have proven themselves to provide reliable, high-accuracy reduced order models for various application problems, which can offer savings of computation time of multiple orders of magnitude. However, for many applications, the time needed in the offline phase for computing the reduced order model – which can take days or longer – needs to be taken into account. Therefore, a reduced basis software which is suited to handle large-scale problem needs to be able to perform offline computations with good computational efficiency. For modern computing architectures this requires parallelization of algorithms.

Considering a standard greedy basis generation algorithm (Algorithm 1.3.10) for the offline phase, the main loop consists of three parts: 1. solving of the reduced problem and error estimation on a training set $\mathcal{P}_{\text{train}}$ to obtain the parameter $\boldsymbol{\mu}^*$, 2. computation of the solution snapshot $p_h(\boldsymbol{\mu}^*)$, 3. extension of the reduced basis using $p_h(\boldsymbol{\mu}^*)$ and reduction of the high–dimensional model. Note that for small to moderate sizes of the reduced basis, steps 2 and 3 are mainly limited by the dimension of Q_h, whereas step 1 is mainly limited by the size of $\mathcal{P}_{\text{train}}$. (In particular for high–dimensional parameter spaces $\mathcal{P}$, $\mathcal{P}_{\text{train}}$ can easily contain millions of parameters.) Therefore, in pyMOR we chose different strategies for the parallelization of step 1 and steps 2 and 3.

For the parallelization of steps 2 and 3, pyMOR relies on an 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 non-distributed data.

Adapting the solver to perform memory distributed operations on operators and vectors at pyMOR's command, however, might be a non-trivial task. pyMOR therefore offers tools which help transitioning to an MPI parallel use of the solver when pyMOR bindings for the serial case already exist: based on the mpi4py [DPSD2008] library, a simple event loop is provided to run as the main loop on all MPI ranks except for rank 0 (see `pymor.tools.mpi`). This allows the user to execute arbitrary Python functions simultaneously on all MPI ranks. A basic resource manager allows to locally store and return function return values via a data handle.

Based on these tools, `MPIVectorArray`, `MPIOperator` and `MPIDiscretization` classes are implemented which hold data handles to the distributed objects they represent and pass interface method calls via MPI to these distributed objects. Communication between the distributed objects (for instance exchange of shared degrees of freedom after operator application) is still implemented by the solver as usual. However, for `VectorArrays` we also provide `MPIVectorArrayAutoComm` which implements communication and summation of local scalar products, etc., in Python. These tools have also been used to integrate the high–dimensional solver for the numerical examples in [MRS2015, Section 5.3].

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`): after instantiation of a worker pool `p`, a single function `f` can be applied to a sequence of arguments `args` in parallel by calling

```
r = p.map(f, args)
```

which is equivalent to the sequential code

```
r = [f(arg) for arg in args]
```

The function and all arguments are automatically serialized and distributed to the workers of the pool.

For the case that the same arguments are required repeatedly on the workers, single objects can be distributed to all workers using the `distribute` method, whereas lists of objects can be scattered among the workers using `distribute_list`. In both cases, a handle object is returned which can later be passed to `map` or the simpler `apply` function (which does not scatter arguments among workers), which transparently map the handle to the already distributed data on the workers. This mechanism is used in Algorithm 3.2.4 to estimate the model reduction error in parallel over a previously distributed training set of parameters.

Algorithm 3.2.4 Parallel error estimation in greedy algorithm.

```
with p.distribute_list(training_set) as s:
    while ...
      ...
      errs, mus = zip(*p.apply(_estimate, rd=rd, training_set=s))
      max_err_ind = np.argmax(errs)
      max_err, max_mu = errs[max_err_ind], mus[max_err_ind]
      ...

def _estimate(rd=None, training_set=None):
    errs = [rd.estimate(rd.solve(mu), mu) for mu in samples]
    max_err_ind = np.argmax(errs)
    return errs[max_err_ind], training_set[max_err_ind]
```

`pyMOR` currently provides a worker pool implementation based on the `IPython` [PG2007] toolkit, which easily allows for parallel computation with large collections of heterogeneous compute nodes. An `MPI`-based implementation using `pyMOR`'s event loop, which can seamlessly be used in conjunction with external solvers using the same event loop, is planned.

3.2.4.2 `dune-pymor`

The **DUNE** module `dune-pymor` is open source software and freely available on GitHub: `https://github.com/pymor/dune-pymor`. It is mainly developed by S. Rave and F. Schindler with contributions from R. Milk. `dune-pymor` serves several purposes: first, it provides the concept of a parameter $\boldsymbol{\mu}$ and parametric objects for C++-based discretizations of parametric problems. Second, it provides the infrastructure for `Python` bindings of these objects and the linear algebra components of `dune-stuff`, based on `pybindgen`[46]. Third, it provides `Python` wrappers around these objects which conform to the corresponding `pyMOR` interfaces. In this section, we briefly discuss some of `dune-pymor`'s features and give illustrating examples by discussing a possible implementation of the thermal block problem (compare Example 1.3.2).

Parameters in `dune/pymor/parameters/` Similar to `pyMOR`, we model parameters $\boldsymbol{\mu} \in \mathcal{P}$ as key/value pairs represented by the `Parameter` class, where the key is an identifier and the value is a collection of `doubles`. Each parameter is of a `ParameterType`, consisting of its keys and the size of the corresponding values.[47] For instance, if we consider a thermal block problem with four subdomains and a parametric right hand side, the parameter types of the data functions could be modeled by

```
Dune::Pymor::ParameterType("diffusion", 4);
Dune::Pymor::ParameterType("force",     1);
```

[46]`https://code.google.com/p/pybindgen/`

[47]For simplicity, we restrict parameter values to vectors in `dune-pymor`, while `pyMOR` also allows for arrays of arbitrary dimensions.

Every parametric object is derived from `Parametric`, the purpose of which is to export the `ParameterType` of the object via `type()` (and to provide some management tools for the implementor of parametric objects). As in `pyMOR`, parameters (and their types) can be joined: suppose we are given two parametric data functions (`diffusion` and `force`) of the above `ParameterTypes`, we could construct a simple parametric problem class:

```cpp
#include <dune/pymor/parameters/base.hh>

using namespace Dune::Pymor;

class ThermalblockProblem
 : public Parametric
{
public:
  ThermalblockProblem(const DiffusionType& diffusion,
                      const ForceType& force)
  {
    this->inherit_parameter_type("d", diffusion); // Provided by
    this->inherit_parameter_type("f", force);     // Parametric.
  }

  void visualize(const Parameter& mu, const std::string filename);
};
```

The parameter `type()` of this problem is given by `({"diffusion", "force"}, {4, 1})`. A parameter $\boldsymbol{\mu}$ corresponding to this type is for instance given by

```cpp
Dune::Pymor::Paramer mu({"diffusion", "force"}, {{0.1, 0.2, 3.0, 4.0}, 0.5});
```

If such a combined parameter is passed to a parametric function, say by calling `problem.visualize(mu, "problem")`, it can be split into the individual parameters again:

```cpp
void visualize(const Parameter& mu, const std::string filename)
{
  assert(mu.type() == this->type());
  auto mu_diffusion = this->map_parameter(mu, "d"); // ("diffusion", {0.1, 0.2, 3.0, 4.0})
  auto mu_force     = this->map_parameter(mu, "f"); // ("force",     0.5)
  // ...
}
```

The parameters `mu_diffusion` and `mu_force` match the respective parameter types of `diffusion` and `force` and can be passed on to these data functions.

To realize affine decompositions as in Definition 1.3.7, we provide the `ParameterFunctional`, modeling $\theta_\xi : \mathcal{P} \to \mathbb{R}$. `ParameterFunctionals` are created with a `ParameterType` and an expression modeling the evaluation of θ_ξ. The four parameter functionals required for the above thermal block problem (which map to the respective parameter component) could for instance be created by:

```cpp
#include <dune/pymor/parameters/functional.hh>

Dune::Pymor::ParameterFunctional theta_0(diffusion.type(), "diffusion[0]");
Dune::Pymor::ParameterFunctional theta_1(diffusion.type(), "diffusion[1]");
Dune::Pymor::ParameterFunctional theta_2(diffusion.type(), "diffusion[2]");
Dune::Pymor::ParameterFunctional theta_3(diffusion.type(), "diffusion[3]");
```

The keys of the parameter type (`"diffusion"`) can be used as variables in expressions, which (in contrast to the simple ones above) can contain mathematical expressions and function evaluations (compare `Stuff::Functions::Expression` in Section 3.1.4.1). `ParameterFunctionals` can be evaluated with a matching `Parameter`: an evaluation of $\theta_2(\mu)$ by calling `theta_2.evaluate(mu_diffusion)` would yield `3.0`.

Parametric functions in `dune/pymor/functions/`. To facilitate the handling of parametric data functions we provide the `AffinelyDecomposableFunctionInterface` for all affinely decomposable functions, such as in Example 1.3.8. It provides access to the components and coefficients of the affine decomposition, where `component(`ξ`)` yields a `Stuff::LocalizableFunctionInterface` (see Section 3.1.4.1) and `coefficient(`ξ`)` yields a `ParameterFunctional` as discussed above, for $0 \leq \xi < \varXi =:$ `num_components()`.

With `Pymor::Functions::Checkerboard`, we also provide a ready-to-use parametric thermal conductivity for the thermal block Example 1.3.2 in `dune/pymor/functions/checkerboard.hh`. It can be created with a `Stuff::Common::Configuration` similar to `Stuff::Functions::Checkerboard`. Instead of prescribing the functions value on each subdomain, however, we have to provide a name that is used as the key to determine the parameter type. In order to obtain the data function for the example above we could use

```
lower_left     = [0 0]
upper_right    = [1 1]
num_elements   = [2 2]
parameter_name = diffusion
```

yielding a function with parameter type `("diffusion", 4)`.

Apart from this specific function we also provide two generic default implementations to facilitate the creation of functions in `dune/pymor/functions/default.hh`:

> `Functions::NonparametricDefault` wraps any function derived from `Stuff::LocalizableFunctionInterface` to conform to `AffinelyDecomposableFunctionInterface`. This allows to implement generic discretizations which can always assume to be provided with a function derived from `AffinelyDecomposableFunctionInterface`, even in the nonparametric case.
>
> `Functions::AffinelyDecomposableDefault` can be used to build arbitrary affinely decomposed functions out of functions derived from `Stuff::LocalizableFunctionInterface`. Via the `register_component(comp, coeff)` method, one can register any number of nonparametric component functions together with the corresponding `ParameterFunctional` modeling the coefficient; the parameter type of the resulting function is automatically deduced from the given coefficient functionals.

We also provide the `AffinelyDecomposableFunctionsProvider` factory class in `dune/pymor/functions.hh` which can be used similarly to the `FunctionsProvider` (compare Section 3.1.4.1) to create affinely decomposed functions at runtime, which are given by a `Stuff::Common::Configuration` (for instance obtained from a data file). The following `Configuration` would yield a parametric force $f(x;\boldsymbol{\mu}) = \boldsymbol{\mu}\,5 + (1-\boldsymbol{\mu})\cos(x)$, for instance:

```
component.0.type  = stuff.functions.constant
component.0.value = 5.0
coefficient.0.force      = 1   # determines parameter type: ("force", 1)
coefficient.0.expression = force[0]
component.1.type       = stuff.functions.expression
component.1.variable   = x
component.1.expression = cos(x[0])
coefficient.1.force      = 1   # determines parameter type: ("force", 1)
coefficient.1.expression = -force[0]
affine_part.type       = stuff.functions.expression
affine_part.variable   = x
affine_part.expression = cos(x[0])
```

The final parameter type of the above function, `("force", 1)`, is determined by setting `force = 1` in the above `Configuration`, yielding $\boldsymbol{\mu} \in \mathbb{R}$.

Parametric operators and functionals. We provide interfaces for parametric operators and functionals (`Pymor::FunctionalInterface` and `Pymor::OperatorInterface`) in `dune/pymor/{functionals,operators}/interfaces.hh`, which are very similar to the ones provided by `dune-gdt` (see Section 3.1.4.2). In contrast to their respective non-parametric counterparts in `dune-gdt`, the interfaces in `dune-pymor` are restricted to act on vectors only and provide some additional information about their source and range (which are fixed per functional/operator to yield non-templated methods). Of course, they are derived from `Parametric` and every method additionally accepts an optional `Parameter`, e.g.:

```
void apply(const SourceType& source, RangeType& range,
           const Parameter mu = Parameter()) const;
```

For nonparametric or affinely decomposed functionals (operators) which are based on vectors (matrices), we provide default implementations of the above interfaces. They are based on `LA::AffinelyDecomposedConstContainer` and `LA::AffinelyDecomposedContainer` in `dune/pymor/la/affine.hh`, which allows (const) access to the components and coefficients of the affine decomposition of the vector representation of a functional (or matrix representation of an operator) similar to `AffinelyDecomposableFunctionInterface` above. Analogously to `Functions::AffinelyDecomposableDefault`, they also allow to create arbitrary affinely decomposed containers by registering any container from `dune-stuff` (see Section 3.1.4.1) along with a matching coefficient `ParameterFunctional`.

Parametric discretization. In the spirit of `pyMOR`, we provide the `StationaryDiscretizationInterface` in `dune/pymor/discretizations/interfaces.hh`. It is meant as a container of a collection of discrete operators, functionals, products and vectors:

```
template< class Traits >
class StationaryDiscretizationInterface
  : public Parametric
  , public Stuff::CRTPInterface< StationaryDiscretizationInterface< Traits >, Traits >
{
  // not all methods and types shown ...
public:
  OperatorType   get_operator() const;
  FunctionalType get_rhs()      const;

  std::vector< std::string > available_products() const;
  ProductType get_product(const std::string id) const;

  std::vector< std::string > available_vectors() const;
  AffinelyDecomposedVectorType get_vector(const std::string id) const;

  std::vector< std::string > solver_types() const;          // To be used like types() and
  DSC::Configuration solver_options(const std::string type = "") const;    // options() of
  void solve(const DSC::Configuration options, VectorType& vector, // a Stuff::LA::Solver.
             const Parameter mu = Parameter()) const;
};
```

Each operator, product, functional or vector may be parametric or nonparametric and is meant to be created by a discretization framework such as `dune-gdt` (see the next section). Each call to `solve` is redirected to the `apply_inverse` method of the system operator (which can be obtained by `get_operator()`).

The purpose of the above operators, functionals and discretizations is two-fold: as pure C++ classes, they can be used by a discretization framework to mange parameter dependent discretizations. Their main intention, however, is to facilitate `Python` bindings and to be used in conjunction with `pyMOR`.

Python bindings. With `dune-pymor`, we ship the required `Python` and `cmake` infrastructure to generate `Python` bindings for `dune-pymor`, parts of `dune-stuff` and any user defined classes derived from `dune-pymor`'s interfaces. Within the `Python` module `dune.pymor.core.bindings`, we provide the `prepare_python_bindings`, `inject_lib_dune_stuff`, `inject_lib_dune_pymor` and `finalize_python_bindings` functions. We also provide the `cmake` macro `add_python_bindings`, which is available to all **DUNE** modules which depend on `dune-pymor`.

To generate `Python` bindings, a user has to provide a C++ header and a corresponding `Python` script, say `foo_bindings_generator.hh` and `foo_bindings_generator.py`. The header should include all required C++ headers and define all required C++ symbols. The `Python` script can make use of the aforementioned functions to generate the bindings:

```
# not all imports shown ...
from dune.pymor.core import (prepare_python_bindings, inject_lib_dune_pymor,
                             finalize_python_bindings)
```

```
# Prepare the module,
module, pybindgen_filename, config_h_filename = prepare_python_bindings(sys.argv[1:])
# add all of libdunepymor,
module, exceptions, interfaces, CONFIG_H = inject_lib_dune_pymor(module, config_h_filename)
# add example user code (see below),
inject_Example(module, exceptions, interfaces, CONFIG_H)
# and write the resulting .cc file.
finalize_python_bindings(module, pybindgen_filename)
```

Using the functions provided in `dune.pymor.core.bindings`, the user obtains the `Python` module (in the sense of `pybindgen`), access to the interfaces of `dune-pymor` and a `Python` dict modeling the `config.h` file of **DUNE**. The latter is required to enable or disable the generation of `Python` bindings of features that depend on external libraries (which may or may not be present in the current build configuration). After adding the user code (see below), the C++ file `foo_bindings_generator.cc` is written to disk. Using the `cmake` macro in a corresponding `CMakeLists.txt` file,

```
add_python_bindings(foo
                    foo_bindings_generator.py
                    foo_bindings_generator.hh
                    libfoo) # optional: linked against libfoo
```

generates a `cmake` target `foo`, which can be treated as any target (e.g., to add additional compile flags):

```
add_dune_alugrid_flags(foo)
add_dune_mpi_flags(foo)
```

Compilation of the `foo` target yields a shared object file `foo.so` which can be imported as a `Python` module.

In the above example, the user code is added via the `inject_Example` function. For instance, if `foo_bindings_generator.hh` provides a discretization derived from `StationaryDiscretizationInterface` (say `FooDisc`, depending on the type of the grid and the containers), the user code could be added as:

```
# not all imports shown ...
from dune.pymor.discretizations import inject_StationaryDiscretizationImplementation

def inject_Example(module, exceptions, interfaces, CONFIG_H):
    assert CONFIG_H['HAVE_ALUGRID']
    assert CONFIG_H['HAVE_DUNE_ISTL']
    # define all types
    GridType = 'Dune::ALUGrid< 2, 2, Dune::simplex, Dune::conforming >'
    MatrixType = 'Dune::Stuff::LA::IstlRowMajorSparseMatrix< double >'
    VectorType = 'Dune::Stuff::LA::IstlDenseVector< double >'
    OperatorType = ('Dune::Pymor::Operators::LinearAffinelyDecomposedContainerBased< '
                    + MatrixType + ', ' + VectorType + ' >')
    FunctionalType = ('Dune::Pymor::Functionals::LinearAffinelyDecomposedVectorBased< '
                      + VectorType + ' >')
    DiscretizationType = 'FooDisc< ' + GridType + ', ' + MatrixType + ', ' + VectorType + ' >'
    # add FooDisc to the python module
    inject_StationaryDiscretizationImplementation(module, exceptions, interfaces, CONFIG_H,
                                                  DiscretizationName,
                                                  Traits={'VectorType': VectorType,
                                                          'OperatorType': OperatorType,
                                                          'FunctionalType': FunctionalType,
                                                          'ProductType': OperatorType},
                                                  template_parameters=[GridType,
                                                                       MatrixType,
                                                                       Vectortype])
```

In addition to `inject_StationaryDiscretizationImplementation`, we provide similar functions to add user classes derived from any of the interfaces in `dune-pymor`. These functions always return the injected class (not shown), which allows the user to add additional functionality (like special constructors) using the syntax of `pybindgen`.

As we can see from the above example, the C++ code has to be exactly mirrored within the `Python` script to successfully generate `Python` bindings. This may become quite elaborate for more complicated examples. In addition, this induces quite a maintenance overhead since any change to the C++ classes in `dune-pymor` has to be mirrored in the `inject_...` functions on the `Python` side.[48]

Python wrappers. In addition to the `Python` bindings we also provide pure `Python` wrappers of the resulting objects, which conform to the interfaces in `pyMOR` and finally allow the C++-based discretization to be used in conjunction with `pyMOR`. In `dune.pymor.core` we therefore provide the `wrap_module` function, which recursively traverses the generated `Python` module and creates a `wrapper` instance which allows to convert user defined classes to `pyMOR` conforming classes (if they are derived from the interfaces in `dune-pymor` and injected into the module with the above mentioned `inject_...` functions).

Let us return to the above example and let us assume that there exists a successfully generated `foo.so` containing a simple `Example` class to setup the grid and the discretization. Using `wrap_module`, we wrap the resulting discretization, which is then directly usable by all algorithms in `pyMOR`:

```
from dune.pymor.core import wrap_module

# import and wrap foo.so
include foo as dune_module
_, wrapper = wrap_module(dune_module)

# create the example and obtain the discretization
example = dune_module.Example()
discretization = example.discretization()

# wrap the discretization
discretization = wrapper[discretization]
```

[48]Since our use of `pybindgen` is purely historically motivated it is likely that in the future we will switch to using `boost.python`: `http://www.boost.org/doc/libs/1_58_0/libs/python/doc/`.

```
# start using the discretization
# ...
```

The resulting `discretization` object is derived from `DiscretizationInterface` in `pymor.discretizations.interfaces`. Consequently, all operators, functionals, products and vectors are also wrapped and derived from the respective interface in `pyMOR`. While the user can always access the original objects via the `_impl` attribute, the wrapped objects can now be used like native `pyMOR` operators, functionals, products, discretizations and vector arrays.

While `dune-pymor` provides the infrastructure for handling parametric problems, it does not actually provide discretizations of such problems. This is the purpose of `dune-hdd`, based on the nonparametric discretizations from `dune-gdt`.

3.2.4.3 `dune-hdd`

The **DUNE** module `dune-hdd` is open source software and freely available on GitHub: `https://github.com/pymor/dune-hdd`. It is mainly developed by F. Schindler with contributions from S. Kaulmann, R. Milk, S. Rave and K. Weber. The purpose of `dune-hdd` is to provide ready-to-use discretizations for parametric problems. In contrast to `dune-gdt` it is not meant as a flexible discretization toolbox but rather as a complete PDE solver that can be coupled with `pyMOR` for model reduction of parametric PDEs.

To treat linear elliptic parametric PDEs (compare Section 1.3) we provide several problem and discretization classes within the `Dune::HDD::Linearelliptic` namespace.

Parametric linear elliptic problems. We provide the virtual `ProblemInterface` in `dune/hdd/linearelliptic/problems/interfaces.hh`, the purpose of which is to act as a container of all required data functions (compare Example 1.3.2):

```
#include <dune/grid/common/gridview.hh>
#include <dune/pymor/parameters/base.hh>
#include <dune/pymor/functions/interfaces.hh>

using namespace Pymor;

template< class E, class D, int d, class R, int r >
class ProblemInterface
  : public Pymor::Parametric
{
  // not all methods and type shown ...
public:
  typedef AffinelyDecomposableFunctionInterface< E, D, d, R, 1, 1 > DiffusionFactorType;
  typedef AffinelyDecomposableFunctionInterface< E, D, d, R, d, d > DiffusionTensorType;
  typedef AffinelyDecomposableFunctionInterface< E, D, d, R, r >    FunctionType;

  virtual const std::shared_ptr< const DiffusionFactorType >& diffusion_factor() const = 0; // λ
  virtual const std::shared_ptr< const DiffusionTensorType >& diffusion_tensor() const = 0; // κ
  virtual const std::shared_ptr< const FunctionType >& force()     const = 0; // f
20   virtual const std::shared_ptr< const FunctionType >& dirichlet() const = 0; // g_D
21   virtual const std::shared_ptr< const FunctionType >& neumann()   const = 0; // g_N
22
23   template< class G >
24   void visualize(const GridView< G >& grid_view,
25                  std::string filename,
26                  const bool subsampling = true,
27                  const VTK::OutputType vtk_output_type = VTK::appendedraw) const;
28
29   std::shared_ptr< NonparametricType > with_mu(const Parameter mu = Parameter()) const;
30 };
```

Similar to the nonparametric functions in `dune-stuff` and the parametric functions in `dune-pymor`, this interface class is templated with the grid elements $t \in \tau_h$ (modeled by `E`), the domain $\mathbb{R}^d$ (modeled by `D` and `d`) and the range $\mathbb{R}^r$ of the solution (modeled by `R` and `r`). Given these template arguments, the types and dimensions of all data functions can be fixed (lines 13–15); implementations of this interface can thus be exchanged at runtime. In addition to providing the data functions (lines 17–21), the problem allows to `visualize` those for a given `grid_view`. Being derived from `Pymor::Parametric`, the problem inherits the `Pymor::ParameterType` of the data functions. If the problem is parametric, it also yields a nonparametric variant of itself via `with_mu(mu)` (given a `Pymor::Parameter mu` of appropriate type), where all data functions have been made non-parametric by inserting the respective parameter components (compare the previous section).

We provide ready-to-use implementations of this interface, most of which can be created by a `Configuration` from `dune-stuff`. Similar to the `Stuff::Grid::BoundaryInfoProvider`, the `Stuff::GridProviders`, the `Stuff::FunctionsProvider` and the `Pymor::AffinelyDecomposableFunctionsProvider`, we also provide the `ProblemsProvider` factory class in `dune/hdd/linearelliptic/problems.hh`, which allows to select one of the following problem implementations from `dune/hdd/linearelliptic/problems/` at runtime, given a suitable `Configuration` (for instance by a corresponding `.ini` file):

`Problems::Default` in `default.hh` provides any combination of nonparametric functions from `dune-stuff` and affinely decomposed parametric functions from `dune-pymor`, given a configuration where the sub-`Configurations` corresponding to the keys `diffusion_factor`, `diffusion_tensor`, `force`, `dirichlet` and `neumann` are passed on to the respective functions provider.

`Problems::ESV2007` in `ESV2007.hh` models the nonparametric problem from [ESV2007, Page 23] and Section 4.2.2.

`Problems::OS2014::ParametricESV2007` in `OS2014.hh` models the parametric problem from Sections 4.2.4.1 and 4.4.1.

`Problems::OS2014::LocalThermalblock` in `OS2014.hh` models the parametric problem from Section 4.2.1.

`Problems::Spe10::Model1` in `spe10.hh` models the parametric problems from Sections 4.2.3 and 4.4.2.

`Problems::Thermalblock` in `thermalblock.hh` models the thermal block problem from Section 4.1.1.

We also provide a test case in `dune/hdd/linearelliptic/testcases/` for each of the above problems, which bundles the problem together with a corresponding `BoundaryInfo` and a suitably refined grid. These test cases are used in automatic system testing to reproduce the results of the respective publications.

Discretizations of parametric linear elliptic problems. We provide several discretizations for the above parametric linear elliptic problem within the `LinearElliptic` namespace. They are all derived from `Discretizations::ContainerBasedDefault`, which is derived from `Discretizations::CachedDefault`, which is derived from `LinearElliptic::DiscretizationInterface`, which is derived from `Pymor::StationaryDiscretizationInterface`.

In addition to the functionality required by `Pymor::StationaryDiscretizationInterface`, the `LinearElliptic::DiscretizationInterface` enforces access to the underlying `grid_view()`, the `test_space()`, the `ansatz_space()`, the `boundary_info()` and the `problem()`. Those are required, for instance, in the context of error estimation and to locally use arbitrary discretizations in the context of the LRBMS (compare Section 2.1).

The `Discretizations::CachedDefault` adds a caching functionality to the `solve` method by holding copies of the already computed solution vectors for each parameter and solver options. This is particularly useful in the context of model reduction, where we might want to solve for the same parameter several times. The caching is implemented using a `std::map` and by enforcing any derived class to implement an `uncached_solve` method.

Finally, `Discretizations::ContainerBasedDefault` implements the methods enforced by all of these interfaces, given matrix and vector representations of the involved operators, products and functionals. The `uncached_solve` method is implemented using the `LA::Solver` infrastructure from `dune-stuff` (see Section 3.1.4.1).

We provide ready-to-use implementations of the CG as well as the SWIPDG discretizations presented in Section 3.1.1.1 for the discretization of parametric linear elliptic problems from above. These are based on `dune-gdt` to discretize the nonparametric operators, functionals and products, which are then combined to form affinely decomposed parametric objects with the tools from `dune-pymor` (see previous Section). We make use of the abstractions from `dune-stuff` and `dune-gdt` to allow for generic discretizations which work for any linear algebra and discrete function space backend. The `Discretizations::CG` in `dune/hdd/linearelliptic/discretizations/cg.hh`, for instance is templated by

```
template< class GridType, Stuff::Grid::ChooseLayer layer,
          class RangeFieldType, int dimRange, int polOrder = 1,
          GDT::ChooseSpaceBackend space_backend = GDT::ChooseSpaceBackend::pdelab,
          Stuff::LA::ChooseBackend la_backend = Stuff::LA::default_sparse_backend >
class CG;
```

and provides the following constructor:

```
CG(GridProviderType& grid_provider,
   Stuff::Common::Configuration bound_inf_cfg,
   const ProblemType& prob,
   const int level = 0)
```

As demonstrated in Section 3.1.4.2, we automatically deduce the correct implementation of the discrete function space (based on `dune-fem` or `dune-pdelab`) using the `GridType`, the `dimRange`, the `polOrder` and the `GDT::ChooseSpaceBackend space_backend` tag. Accordingly, we deduce the required `GridPart` or `GridView` given the `GDT::ChooseSpaceBackend space_backend`, the `Stuff::Grid::ChooseLayer layer` and the `GridType`, as demonstrated in Section 3.1.4.1.

Similarly, we provide the `Discretizations::SWIPDG` in `dune/hdd/linearelliptic/discretizations/swipdg.hh`, which accepts the same template and constructor arguments. Both discretizations only differ in the respective implementation of the `init()` method, which assembles the matrix and vector representations of the nonparametric components of the operators, products and functionals, based on the corresponding operators, products and functionals from `dune-gdt` (see Section 3.1.4.1).

Based on the two discretizations we also provide the `Discretizations::BlockSWIPDG` in `dune/hdd/linearelliptic/discretizations/block-swipdg.hh` as an implementation of the discretization proposed in the context of the LRBMS (Section 2.1). This discretization is created with a `Multiscale::ProviderInterface< GridType >` grid provider from `dune-grid-multiscale`[49]. On each subdomain, one of the discretizations from above is created by passing on the multiscale grid provider along with the `Stuff::Grid::ChooseLayer::local` tag and the subdomain encoded in the `level` variable.

Since all of these implementations are derived from `Pymor::StationaryDiscretizationInterface`, `pyMOR` compatible `Python` bindings and wrappers are automatically available through `dune-pymor` (compare previous section). Together with the ability to exchange the problem at runtime (given any string-based `Configuration`), this allows for an easy coupling of the high-performance discretizations from `dune-gdt` with the interactive model reduction framework `pyMOR`.

[49]`https://github.com/pymor/dune-grid-multiscale/`

4 Numerical experiments

This chapter presents experiments which demonstrate various aspects of the LRBMS (Chapter 2), as well as experiments and benchmarks of the discretization and model reduction software framework (Chapter 3). We present those experiments in chronological order to demonstrate the evolution of the methodology from a localization of traditional reduced basis methods to a fully adaptive reduced basis multiscale method. In addition, we discuss some of the obstacles we met along the way, which motivated and guided the further development of the LRBMS and the required software packages.

4.1 The localized reduced basis (multiscale) method

First ideas of localized reduced basis methods for multiscale problems emerged in the context of S. Kaulmann's Diploma thesis [Kau2011, KOH2011]. The experiments in [Kau2011] were realized within `dune-rb` [DHKO2012, DHO2012, Dro2012, KFH+2014], the development of which was started by B. Haasdonk in 2008 and soon picked up by M. Drohmann and S. Kaulmann. `dune-rb` provided a tight integration of discretization and reduced basis schemes and could be coupled with `RBmatlab`[1] [Dro2012]. (Compare Section 3.2.2.)

Based on [KOH2011], the first variant of the LRBMS [AHKO2012] (joint work with S. Kaulmann, B. Haasdonk and M. Ohlberger) was in essence a localization of the traditional reduced basis method, applied to multiscale problems. The detailed discretization in both [KOH2011] and [AHKO2012] was limited to an SIPDG discretization on the full fine triangulation, in contrast to the variant discussed in Chapter 2.

The a posteriori error estimate in [KOH2011] was based on local lifting operators, which turned out to be disadvantageous in practice. In the context of a DFG project on *"Multiscale analysis of two-phase flow in porous media with complex heterogeneities" (grant number OH 98/4-1)*, we decided to complement the existing method with the residual-based a posteriori error estimator from Section 2.3.1 in [AHKO2012].[2]

4.1.1 The thermal block experiment

In [AHKO2012, Section 5.1] we considered the *thermal block* problem of stationary heat diffusion from Example 1.3.2 on the unit cube $\Omega = [0,1]^2$, which is a popular toy-problem within the RB community. We chose 16 subdomains Ω_ξ, given by an equidistant cubic

[1] `http://www.morepas.org/software`

[2] The individual contributions in [AHKO2012] were as follows: the experiments were conducted by S. Kaulmann while the development of the error estimator and the compilation of the publication was carried out by F. Schindler.

partitioning of Ω, yielding a 16-dimensional parameter, where each component models the constant value of the thermal conductivity in the corresponding subdomain (compare Figure 4.1). The problem definition was completed by setting $f = 1$, $\kappa \in \mathbb{R}^{2\times 2}$ as the unit matrix and by prescribing homogeneous Dirichlet boundary values ($g_D = 0$) on the top an bottom of the domain as well as homogeneous Neumann boundary values ($g_N = 0$) on the left and right. We discretized the above problem with a first order SIPDG scheme (see Section 2.1) on a 30×30 equidistant rectangular grid ($|\mathcal{T}_h| = 900$). Both the high-dimensional as well as the reduced discretization along with all required algorithms were implemented in `dune-rb`, allowing for the tightest possible integration of high- and low-dimensional schemes.[3]

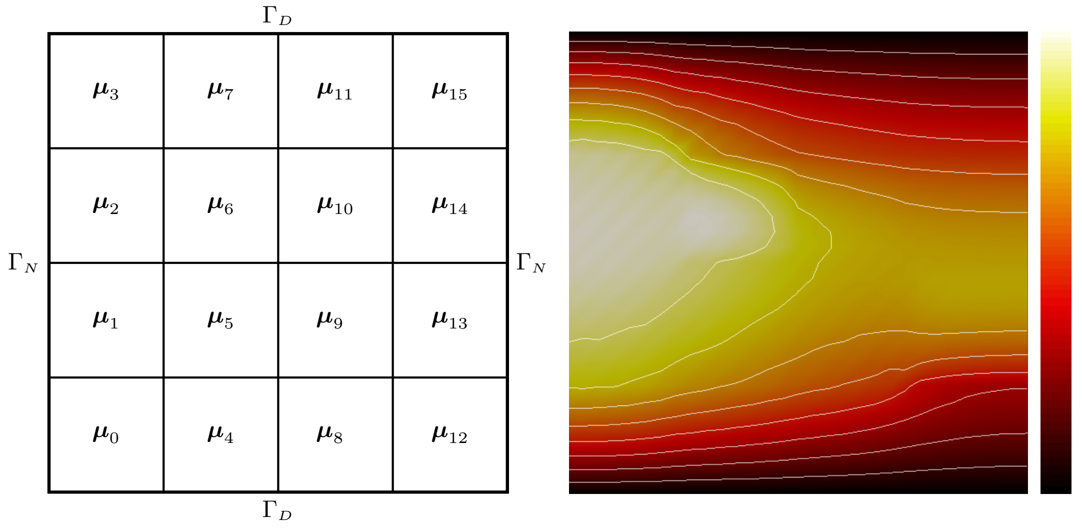

Figure 4.1: Problem setup for the thermal block problem from Example 1.3.2 with 16 subdomains and indicated boundary types (left) and sample solution for $\boldsymbol{\mu} = (3, 6, 9, 2, 5, 8, 1, 4, 7, 7, 10, 3, 6, 9, 2, 5, 8)$ (right), with values ranging between 0 (dark) and $3.05 \cdot 10^{-2}$ (light).

We considered the 16-dimensional parameter space $\mathcal{P} = [0.1, 1]^{16}$ for this parametric problem. For the basis generation we used the discrete weak greedy algorithm from Definition 1.3.11 on a training set $\mathcal{P}_{\text{train}} \subset \mathcal{P}$ of 100 randomly chosen parameters with the localized `extend` method from Definition 2.4.1 (using a local Gram-Schmidt algorithm for the orthonormalization of the local bases), together with η_{red} from Theorem 2.3.1 for `estimate` and an empty basis in `init`.

Although this problem was computationally quite small and simple, it allowed us to investigate a special property of the LRBMS: formally, the LRBMS interpolates between a standard RB and a standard SIPDG method. By choosing different coarse grids $\mathcal{T}_H$

[3]The implementation corresponding to these experiments can be obtained at `https://users.dune-project.org/projects/dune-rb`, roughly at around commit `493977c`. The experiments were conducted on standard Desktop machines of that time.

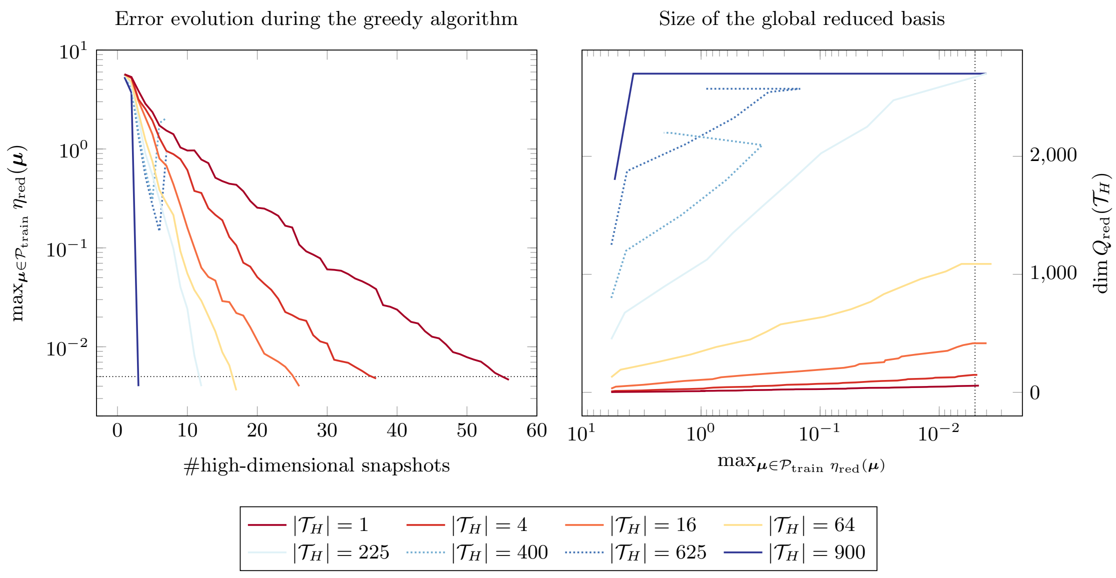

Figure 4.2: Estimated error evolution and size of the global reduced basis to reach the desired tolerance (thin black dots) during the greedy basis generation for the *thermal block* experiment from Section 4.1.1 for different configurations of the coarse grid (data for 225 and 900 subdomains beyond the tolerance is not shown). Left: logarithmic plot of the maximum estimated model reduction error depending on the number of high-dimensional solution snapshots. Right: logarithmic plot of the resulting size of the global reduced basis. Note the numerical instabilities for 400 and 625 subdomains (thick colored dots).

as equidistant rectangular partitions of Ω with 1, 2×2, 4×4, 8×8, 15×15, 20×20, 25×25 and 30×30 subdomains, respectively, we were able to cover the complete range between the two extremes: for $|\mathcal{T}_H| = 1$ the resulting scheme corresponds to a standard RB approach where nearly all work is done offline, while for $|\mathcal{T}_H| = 900$ the resulting scheme corresponds to a standard SIPDG discretizations where nearly all work is done online.

Let us fist discuss the generation of the reduced basis, where we disregard the numerical instabilities, which are visible in Figure 4.2 for 400 and 625 subdomains, for the moment. As we observe in Figure 4.2 (left), by increasing the number of subdomains, we lower the number of high-dimensional solution snapshots, which are required to reach the desired tolerance during the greedy algorithm: while a standard RB method ($|\mathcal{T}_H| = 1$) requires 56 snapshots, only three suffice for a standard SIPDG method ($|\mathcal{T}_H| = 900$). The latter is not surprising, since three DoFs are enough to fully specify piecewise linear functions on rectangles in 2 dimensions.

On the other hand, as we observe in Figure 4.2 (right), the size of the global reduced basis grows considerably with an increasing number of subdomains, since for each

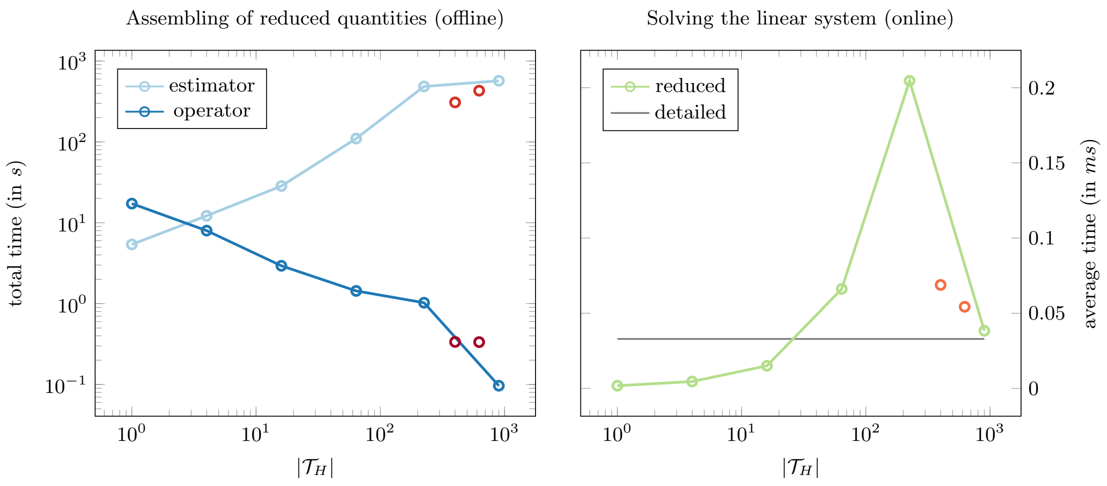

Figure 4.3: Comparison of selected offline and online timings for the *thermal block* example from Section 4.1.1. Left: Log/log plot of the measured total time spent for assembling the reduced operator and estimator during the greedy basis generation (over all extension steps). Right: Log plot of the measured average time for computing a reduced solution, compared to the average measured time for computing a high-dimensional solution (over 25 randomly selected test parameters). Note the outliers (marked as individual red circles) corresponding to untrustworthy measurements obtained for numerically unstable configurations (400 and 625 subdomains, compare Figure 4.2).

snapshot roughly $|\mathcal{T}_H|$ basis functions are added to the global reduced basis (see the discussion on local vs. global interpretation of the reduced basis in the context of the LRBMS in Section 2.2).

The low number of required solution snapshots reflects the sizes of the *local* reduced bases. Since many parts of the basis generation can be carried out locally in parallel, we hope to significantly lower the offline time by increasing the number of subdomains. At the same time, the size of the *global* basis reflects the size of the resulting reduced linear system and we expect reduced solutions to take more time during the online phase.

As we observe in Figure 4.3, both is only partially true. By increasing the number of subdomains we can decrease the total time spent for assembling the reduced operator by several orders of magnitude, while the total time spent for assembling the reduced estimator increases by several orders of magnitude (compare Figure 4.3, left). The latter is not surprising, since the residual based estimator requires the inversion of a global product matrix for each global basis function (compare Section 2.3.1).

As expected, the average time to solve the resulting reduced problem online increases for more subdomains, eventually surpassing the average time to solve the full high-dimensional problem (for 64 subdomains and above, compare Figure 4.3, right). While the reduced system matrix is sparse, its pattern is less favorable due to larger dense blocks (compare Section 2.2). However, for the extreme case where the LRBMS coincides with a SWIPDG discretization ($|\mathcal{T}_H| = 900$), the reduced system matrix corresponds to the high-dimensional one again, which is reflected in the average time required for a reduced solution.

Remark 4.1.1 (Discussion of the *thermal block* experiment)**.** *There are several things worth noting regarding the comparison of standard RB, LRBMS and standard SIPDG schemes in the context of the* thermal block *example. First of all, we could show that the resulting local reduced bases allow for an increased flexibility of the global reduced space (which is reflected in the size of the global reduced basis) and thus for greatly improved reduced approximation quality. In particular, much less global solution snapshots were required to reach the same accuracy, compared to standard RB. As a drawback (resulting from the larger reduced system), we could observe an increase in average solution times.*

Of course, the biggest effect of an increased number of subdomains was visible in the offline basis generation. While we could observe a large potential for savings (e.g., regarding the assembly of the reduced operator) it also became clear that the standard residual based a posteriori error estimator was not suitable for the LRBMS. Using that estimator, the total offline time was much higher for the LRBMS than for the standard RB method.

The purpose of the thermal block experiment was mainly to get a feeling for the possible effects of varying the number of subdomains, though it was rather simple and academic in nature. We thus conducted another experiment that was closer to the original motivation for the LRBMS. At that time we were involved in the above mentioned DFG project on *"Multiscale analysis of two-phase flow in porous media with complex heterogeneities"* and were exploiting the possibilities of model reduction in the context of two-phase flow (compare Section 1.4).

4.1.2 The Spe10 model2 experiment

The problem setup of the second experiment we conducted in [AHKO2012] modeled the flooding of a domain in the context of two-phase flow in porous media (posed in the global pressure formulation, compare Example 1.4.2). We chose the highly heterogeneous data from the *Spe10 model2* benchmark[4] for the permeability field κ_ε (compare Figure 4.4, right), $f = 0$ and boundary values modeling a flow through the domain along the longest axis. To model a flooding of the domain by one of the two fluid phases we chose a parametric total mobility λ as a linear combination of functions modeling a saturation drop at different profiles throughout the domain, spreading from one corner of the domain to the opposite (compare Figure 4.4, left). In this setup, we can think of each parameter component being associated with a different time-step of the global pressure system (1.0.1). We discretized this problem with the same SIPDG scheme as above on a $60 \times 220 \times 42$ rectangular grid ($|\tau_h| = 554,400$), implemented in `dune-rb`.

[4] `http://www.spe.org/web/csp/datasets/set02.htm`

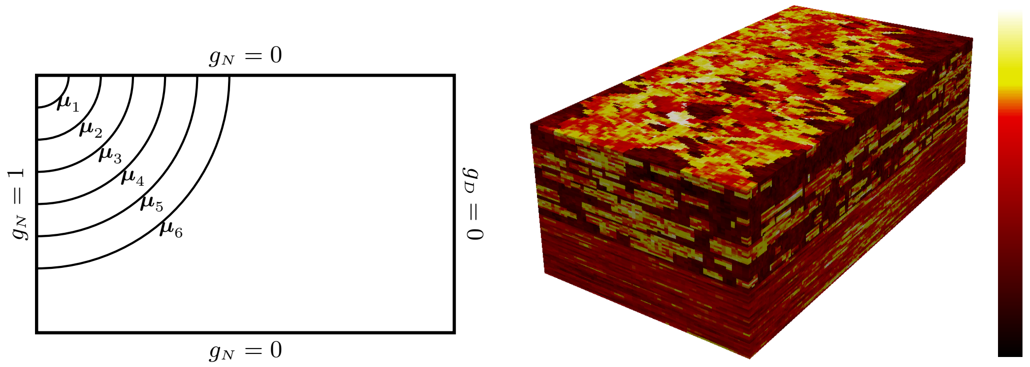

Figure 4.4: Problem setup for the *Spe10 model2* experiment from Section 4.1.2. Left: schematic plot of the positions of the saturation drops with the associated parameter components and plot of the boundary values. Right: Re-scaled logarithmic plot of the absolute value of the permeability field κ_ε (from the *Spe10 model2* benchmark setup), with values ranging between $7 \cdot 10^{-4}$ (dark) and $2 \cdot 10^4$ (light).

For this parametric problem we considered the six-dimensional parameter space $\mathcal{P} = [0.01, 0.95]^6$ and chose for the basis generation the same weak discrete greedy algorithm with local Gram-Schmidt basis extensions as in the *thermal block* example of the previous section, searching over 100 randomly selected parameters. This problem was far more challenging to tackle than the previous one: a single high-dimensional solution snapshot took about 530 seconds on standard desktop machines of that time. While the heterogeneous permeability certainly posed some difficulties for the linear solver, the sheer size of the problem posed challenges for the projection and orthonormalization algorithms in terms of memory and CPU demand (due to large operators and vectors). Thus, this problem was a perfect test bed for the LRBMS, the purpose of which was to allow for model reduction of such problems using less CPU time and memory than standard RB approaches.

We again chose different coarse grids $\mathcal{T}_H$ as equidistant rectangular partitions of Ω. Due to time constraints, however, we only conducted experiments for 1, $2 \times 2 \times 2$, $2 \times 4 \times 2$ and $4 \times 4 \times 2$ subdomains; thus, less data was available compared to the previous example.

Considering the basis generation, in Figure 4.5 (left) we can observe a similar behavior as in the *thermal block* case: with an increasing number of subdomains, less high-dimensional solution snapshots are required to reach the desired tolerance during the greedy algorithm. While the differences are not that pronounced (23 snapshots for the standard RB approach vs. 19 snapshots for the LRBMS with 32 subdomains), this nevertheless resulted in savings of 17% of the computational time spent for the high-dimensional snapshots.

Unfortunately, this did not lead to overly large savings during the offline part: as we observe in Figure 4.5 (right), the size of the global basis grows with an increasing number of subdomains (as discussed in Section 4.1.1). While the total time spent for

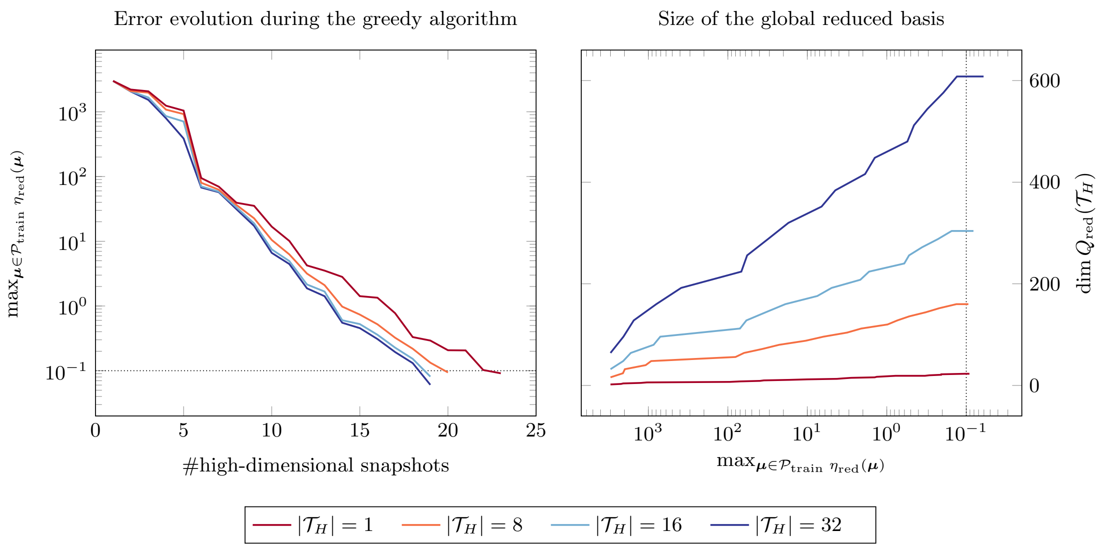

Figure 4.5: Estimated error evolution and size of the global reduced basis to reach the desired tolerance (dotted) during the greedy basis generation for the *Spe10 model2* experiment from Section 4.1.2 for different configurations of the coarse grid. Left: logarithmic plot of the maximum estimated model reduction error depending on the number of high-dimensional solution snapshots. Right: logarithmic plot of the resulting size of the global reduced basis.

computing solution snapshots decreases for an increasing number of subdomain, the total time spent for the evaluation of the reduced estimator increases: although only a low-dimensional summation is required, we still need to compute the sum of $6^2 \cdot 608$ reduced quantities for 32 subdomains, since the performance of the reduced estimator depends on the size of the coarse grid (compare Section 2.3.1). Thus, the overall part of the offline computation that was not spent for the assembly of the operator or the estimator only slightly decreased ("rest" in Figure 4.6, left).

As expected, the growing size of the global reduced basis also led to an increase of the average time required to solve the reduced problem, roughly by a factor of 100 (from $0.3ms$ to $34ms$, see Figure 4.6, right).

Considering the time spent in individual parts of the offline computation (displayed in Figure 4.6), we observe a similar behavior as in the *thermal block* case: while the total time required to assemble the reduced operator decreases significantly, the total time required for the assembly of the reduced estimator grows dramatically with an increasing number of subdomains. Thus, the total offline time of the LRBMS is significantly larger than for standard RB methods.

Remark 4.1.2 (Discussion of the *Spe10 model2* experiment)**.** *There are several things worth noting regarding the* Spe10 model2 *experiment. As already observed in the* thermalb block *experiment, the residual based a posteriori error estimate proved to be ill-fitted for the LRBMS. While perfectly usable in the greedy basis generation in terms of its esti-*

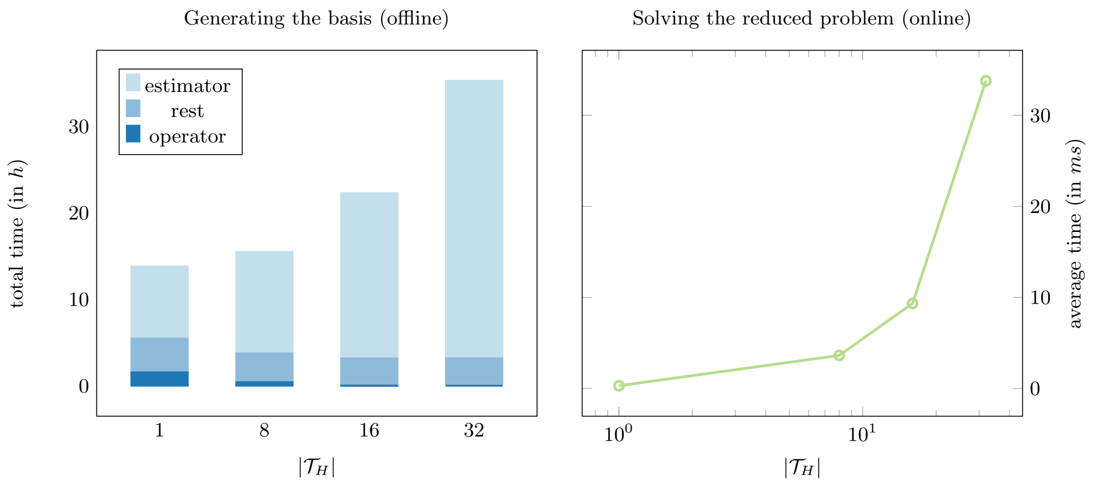

Figure 4.6: Comparison of offline and online timings for the *Spe10 model2* experiment from Section 4.1.2. Left: Breakdown of the individual parts of the measured total time spent for the greedy basis generation during the offline part of the computation (over all extension steps): assembling of the reduced operator (dark), assembling of the reduced estimator (light) and remaining part of the offline computation (medium). Right: Log plot of the measured average time for computing a reduced solution (over 25 randomly selected test parameters). Note that computing a single high-dimensional solution takes about $530s$.

mation qualities, the computational time required to assemble the reduced estimator were not acceptable.

Nevertheless, we were still satisfied with the overall results of this experiments. The largest bottleneck in the context of multiscale problems is usually the computation of high-dimensional solution snapshots (compare Section 1.2): the time to solve larger real-world problems can grow arbitrarily large (up to month even on high-performance clusters). The ability of the LRBMS to reach the same error tolerance with fewer global solution snapshots thus made it a promising method for the model reduction of multiscale problems.

This early work on the LRBMS [AHKO2012] was the starting point and provided directions on what to pursue next. While we had learned about the strengths of the LRBMS (namely its superior approximation quality and parallelization capabilities due to localization) we had also identified several shortcomings of our approach in [AHKO2012], concerning the error estimator as well as the software framework. On the mathematical side, it was clear that the residual based a posteriori error estimator would have to be replaced. Since we were incorporating spatial localization into RB methods it made sense to develop a spatially localized estimator. In addition we could identify several shortcomings of the existing software framework `dune-rb`. On the one hand, `dune-rb` was completely based upon `dune-fem` and thus suffered from limitations concerning the discretization (as discussed in Section 3.1.3).[5] The strict dependency on `dune-fem` would in particular be an issue when implementing the new localized a posteriori error estimator. On the other hand, `dune-rb` was also responsible for the model reduction process, in particular the greedy basis generation. It thus suffered from the limitations discussed in Section 3.2.2. In the *thermal block* experiment (Section 4.1.1), for instance, we observed numerical instabilities during the basis generation, which probably stemmed from the implementation of the Gram-Schmidt orthonormalization procedure, or the products involved. Due to its nature (implemented in the system language C++, tightly integrated into the **DUNE** framework), `dune-rb` was not a very flexible model reduction framework: it was, in particular, not easily possible to quickly adapt and change the model reduction specific algorithms, try out different products, orthonormalization techniques and so forth...

After the publication of [AHKO2012] we thus split efforts: S. Kaulmann further pursued the original motivation of the LRBMS while M. Ohlberger and F. Schindler were interested in developing the localized error estimator and pursuing new ideas regarding local online adaptation of the reduced bases. S. Kaulmann thus integrated the LRBMS as a model reduction technique for the pressure equation into the tool-chain required for full two-phase flow simulations. In that context he demonstrated how an application of the LRBMS could lead to large computational savings (see [KFH+2014]).

[5]The integration with `dune-fem` went so far that in some stage of the development of `dune-rb`, the reduced basis space was actually derived from `dune-fem`'s discrete function space, in contrast to being a collection of reduced basis vectors. Thus, all operator projections involved iterating over the computational grid, in contrast to simple linear algebra operations.

4.2 A new discretization framework: `dune-gdt`

Due to the above mentioned shortcomings of `dune-rb` and the fact that M. Drohmann left the development team after his PhD, the public development of `dune-rb` came to a halt in 2012 and we required new software frameworks to realize our ideas. In the beginning of 2011, M. Drohmann, S. Girke and F. Schindler had already started to work on a new discretization framework that was later to become `dune-gdt`. In the early stages it was based upon `dune-fem` and later on abstracted further to additionally support discrete function spaces from `dune-fem-localfunctions` [Gir2012] and `dune-pdelab` (see Section 3.1 for further information). By 2012, S. Rave had joined our team and he, R. Milk and F. Schindler began to work on `pyMOR`, which would become exactly the flexible model reduction framework we required (see Section 3.2).

4.2.1 A first online enrichment experiment

The new localized a posteriori error estimate we were developing should not only bring down the total time of the offline computation, it would also provide us with local error information (in a spatial sense) during the offline and online phase. The latter would enable us to identify those local reduced bases the approximation quality of which was insufficient for the given parameter. While traditional RB methods only allow an adaptation of the reduced basis by involving the full high-dimensional model, the localized approach of the LRBMS would then allow us to enrich the insufficient local reduced bases by local computations - much in the spirit of domain decomposition and numerical multiscale methods (compare Chapters 1 and 2).

We thus conducted a first experiment in early 2013 in the context of the *"Numerical Upscaling for Media with Deterministic and Stochastic Heterogeneity"* Oberwolfach mini-workshop [AO2013]. At that time our newly developed discretization framework was in good shape and we had additionally developed the `dune-grid-multiscale`[6] module to realize the coarse grid. In `dune-grid-multiscale`, each subdomain is modeled as a `GridView` or `GridPart` (in the sense of `dune-grid` and `dune-fem`, respectively) which allows to locally use arbitrary discretizations that are not aware of the LRBMS context. Unfortunately, our model reduction framework `pyMOR` was not yet fully coupled with our discretization framework and we conducted a simple experiment, which focused on the aspect of solving local corrector problems (rather than performing the full model reduction).

In [AO2013], we again considered the *thermal block* problem from Example 1.3.2 on the unit cube $\Omega = [0,1]^2$ with only three subdomains, yielding a three-dimensional parameter (compare Figure 4.7, left). We completed the problem definition by setting $f = 1$, $\kappa \in \mathbb{R}^{2\times 2}$ as the unit matrix and by prescribing homogeneous Dirichlet boundary values ($g_D = 0$) on the domain boundary ($\Gamma_D = \partial\Omega$). We discretized the above problem with the generalized DG scheme from Section 2.1 (using a first order continuous Galerkin FE discretization in each coarse grid element and a SWIPDG coupling with respect to

[6]`https://github.com/pymor/dune-grid-multiscale/`

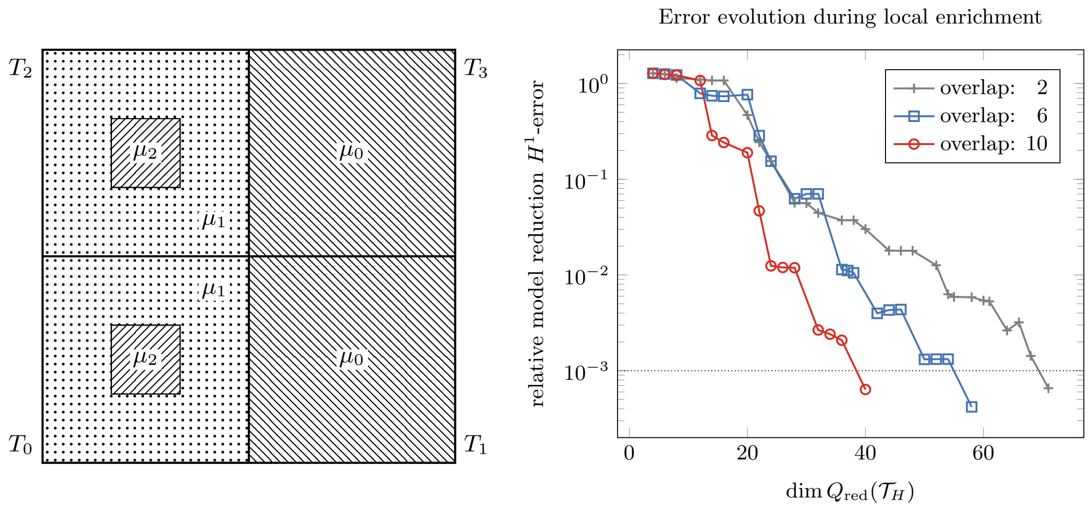

Figure 4.7: Problem setup and error evolution for the enrichment experiment from Section 4.2.1. Left: Layout of the three subdomains (different shadings) for the *thermal block* problem from Example 1.3.2, the associated parameter components and the four coarse grid cells T_i. Right: error evolution during the intermediate local enrichment phase for different overlap sizes (where, for instance, 2 corresponds to an overlap of two layers of fine grid elements around a coarse grid element).

the coarse grid) on a 50×50 equidistant rectangular fine grid ($|\tau_h| = 2500$) and a 2×2 equidistant rectangular coarse grid ($|\mathcal{T}_H| = 4$).[7]

We were interested in modeling the situation outlined in Section 1.4: we are given an insufficient reduced space and need to enrich the local reduced bases to approximate the solution for an untrained parameter during the online phase. To this end we used the discrete weak greedy algorithm from Definition 1.3.11 on a training set $\mathcal{P}_{\mathrm{train}} := \{(0.1, 1, 1)\}$ consisting of only one parameter, resulting in a local reduced basis of size one on each of the four coarse grid elements. We then solved for the parameter $\boldsymbol{\mu} = (0.1, 1, 0.01)$, which differed from the training parameter only in the local region associated with the third parameter component (compare $\boldsymbol{\mu}_2$ in Figure 4.7, left), resulting in a locally high contrast of the thermal conductivity.

The reduced basis was clearly insufficiently trained for this parameter and we started the intermediate local enrichment phase as discussed in Section 2.4.2. Since the development of our new local error estimator was not yet completed and we were mainly interested in observing the error evolution during the basis enrichment, we used the local relative model reduction H^1-error, $\|p_h(\boldsymbol{\mu}) - p_{\mathrm{red}}(\boldsymbol{\mu})\|_{H^1,T} / \|p_h(\boldsymbol{\mu})\|_{H^1,T}$ for all $T \in \mathcal{T}_H$, as local error indicators and marked those subdomains for enrichment, where the local error indicator lay above the average of all indicators.

[7]The implementation corresponding to this experiment can be obtained from `https://github.com/pymor/dune-hdd` by checking out the `oberwolfach-2013-abstract-albrecht-thermalblock` commit.

We conducted the experiment for different overlap sizes and recorded the number of iterations required to reach the desired relative model reduction H^1-error of 0.1%, as well as the size of the resulting reduced basis. As we observe in Figure 4.7 (right), the prescribed tolerance was reached after 14 local enrichment steps for ten overlap layers (which corresponds to 2/5th of a coarse grid cell). The final sizes of the resulting local reduced bases, $|\phi_{\text{red}}^{T_0}| = |\phi_{\text{red}}^{T_2}| = 14$ and $|\phi_{\text{red}}^{T_1}| = |\phi_{\text{red}}^{T_3}| = 6$, show the local influence of the parameter component $\boldsymbol{\mu}_2$ and the symmetry of the problem.

Remark 4.2.1 (Discussion of the first online enrichment experiment)**.** *First of all, the results of our first experiment on local basis enrichment were very promising but at the same time not too surprising. The use of local corrector problems to enrich the solution is a well established technique in the context of numerical multiscale or domain decomposition methods, and a similar error decay can be observed in that area. Nevertheless we were satisfied to observe that the techniques of numerical multiscale methods could be transferred in a straightforward way to the LRBMS.*

In addition, we used this first experiment to demonstrated the state of our new discretization framework, as we were now able to locally use any discretization of at least first order and couple those with respect to the coarse grid in a generic way.

The experiment on local enrichment was only one of several options we were pursuing. By that time the theoretical development of our new localized a posteriori error estimator had proceeded enough to start the corresponding implementation in our new discretization framework, the development of which had proceeded enough to provide the Raviart-Thomas-Nédeléc spaces required for the diffusive flux reconstruction.

4.2.2 A first validation of the new localized estimator

Our new localized a posteriori error estimate [OS2014] was based on the work of A. Ern, A. F. Stephansen and M. Vohralík [ESV2007, ES2008, ESV2010] (compare Section 2.3.2). To be applicable in the context of the LRBMS we had to adapt the original estimate with respect to the parameterization of the problem and the coarse triangulation. In a nonparametric setting our estimate was quite close to the original one, except for a less favorable summation of the local contributions (since we were interested in localization with respect to the coarse grid and not the fine grid, compare Theorem 2.3.3 and [ESV2010, Theorem 3.1]).

In 2014 we conducted a first validation of the new error estimator in a nonparametric setting in the context of the *"Finite Volumes for Complex Applications VII"* conference [OS2014]. We considered Example 1.1.2 on $\Omega = [-1, 1]^2$ with $f = \lambda = 1$, κ as the unit matrix in $\mathbb{R}^{2\times 2}$ and Dirichlet boundary values $g_D(x, y) = \cos(1/2\pi x)\cos(1/2\pi y)$, reproducing the nonparametric example from [ESV2007, Section 8.1], where an exact solution and data on the performance of the original estimator was available. We discretized the above problem with the SWIPDG scheme from Section 2.1 using a 0th order diffusive flux reconstruction on a series of fine grids to exactly match the experiment in [ESV2007,

Section 8.1].[8]

The purpose of this experiment was to study the impact of the coarse grid on the quality of the estimator and we considered several configurations of the coarse grid, each given by equidistant rectangular partitions of Ω. We could reproduce the results from [ESV2007, Section 8.1] with our discretization framework and refer to Section 4.2.4 for a detailed discussion. As we observe in Figure 4.8, our estimator is not as sharp as the original one (which is in particular due to the presence of the width of the coarse grid in the residual component and thus not unexpected) but still very acceptable.

Another purpose of this first validation of the new error estimator was to demonstrate the implementation of the a-posterior error estimator within our discretization framework.

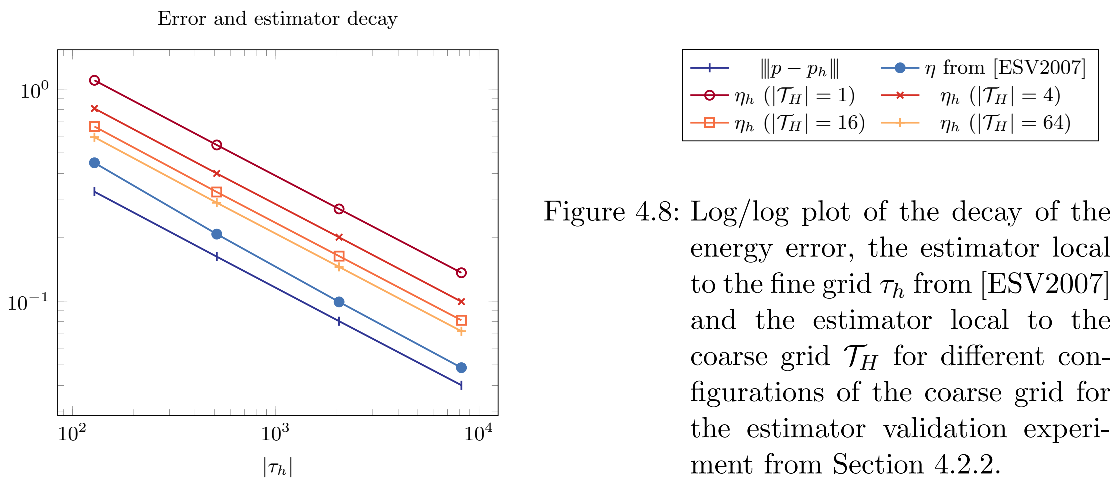

Figure 4.8: Log/log plot of the decay of the energy error, the estimator local to the fine grid τ_h from [ESV2007] and the estimator local to the coarse grid $\mathcal{T}_H$ for different configurations of the coarse grid for the estimator validation experiment from Section 4.2.2.

In addition to this preliminary validation study, we were also interested in the localization qualities of the estimator, in particular in the context of multiscale phenomena.

4.2.3 A first localization study of the new estimator

Once we had successfully implemented the estimator for the nonparametric setting we turned our attention to its localization qualities in the context of highly heterogeneous multiscale problems. For the presentation associated with [OS2014] we thus considered Example 1.1.2 with $\lambda = 1$, homogeneous Dirichlet boundary values ($g_D = 0$) and the highly heterogeneous permeability field κ[9] and forces f as depicted in Figure 4.9 (top). We discretized this problem with the SWIPDG scheme from Section 2.1 using a 0th order diffusive flux reconstruction on a triangular grid with 8,000 elements.[10]

[8] The implementation corresponding to this experiment can be obtained from `https://github.com/pymor/dune-hdd-demos` by checking out the `OS2014-FVCA7-poster` commit.

[9] With κ_ε obtained from `http://www.spe.org/web/csp/datasets/set01.htm`, compare Figure 4.9.

[10] The implementation corresponding to this experiment can as well be obtained from `https://github.com/pymor/dune-hdd-demos` by checking out the `OS2014-FVCA7-poster` commit.

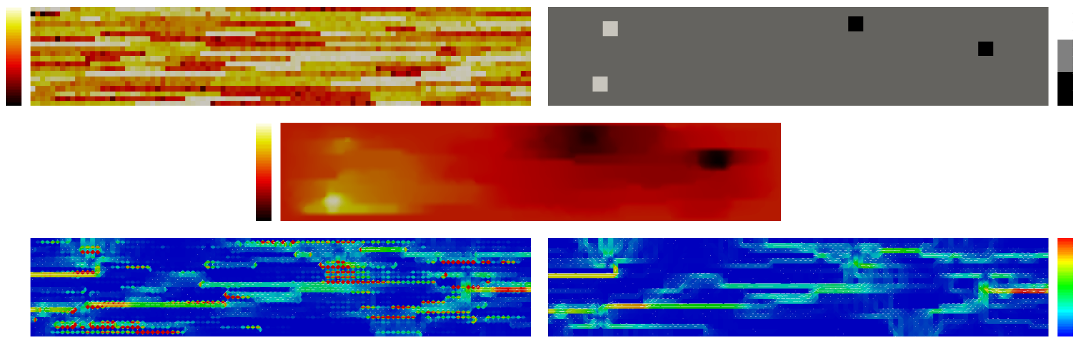

Figure 4.9: Data functions, pressure distribution and velocity reconstruction for the experiment from Section 4.2.3, on a triangulation with $|\tau_h| = 8,000$ elements. Top row, left: logarithmic plot of the absolute value of the permeability field κ_ε (from the *Spe10 model1* benchmark setup) with values ranging from 10^{-3} (dark) to 10^3 (light). Top row, right: plot of f, modeling two producers (-1, light) and two injectors (1, dark), zero elsewhere. Middle row: plot of the pressure solution p_h (dark: low pressure, light: high pressure). Bottom row: comparison of the magnitude of the diffusive flux reconstruction $R_h^0[p_h]$ (right) and the piecewise Darcy velocity $(-\lambda\kappa\nabla_h p_h)|_t$ on each grid element $t \in \tau_h$ (left). Note that both bottom plots share the color map of the diffusive flux reconstruction (blue: low magnitude, red: high magnitude) to better identify the discontinuities and overshoots in the Darcy velocity.

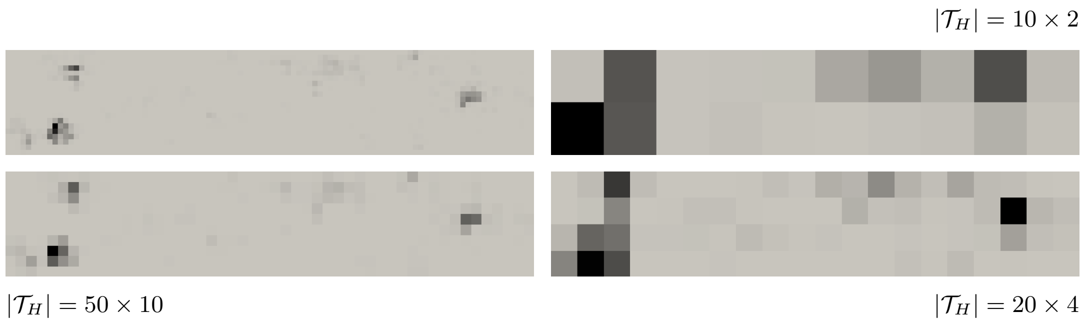

Figure 4.10: Comparison of the spatial distribution of the local energy error contribution on a triangulation with $|\tau_h| = 8,000$ elements (top left) with the local estimator contribution for different coarse grid configurations (small contribution: light, large contribution: dark) for the experiment from Section 4.2.3.

As a side-product, the computation of the error estimator also yields an H_{div}-conforming reconstruction of the Darcy velocity (compare Section 2.3.2.2), which is useful when simulating slow flow (since in that context we require a good approximation of the Darcy velocity, compare Equations 1.0.1). As we observe in Figure 4.9 (bottom), the diffusive flux reconstruction $R_h^0[p_h]$ gives a physically more meaningful reconstruction of the Darcy velocity than a straightforward computation (which gives undesirable results due to the non-conformity of the pressure and the discontinuity of the permeability).

The main purpose of this experiment was to study the localization of the a posteriori error estimator, compared to the localization of the true error in the energy norm.[11] We thus computed the local estimator components (as discussed in Corollary 2.3.6) for several configurations of the coarse grid $|\mathcal{T}_H|$. As we observe in Figure 4.10, the spatial distribution of the estimator contributions aligns well with the spatial distribution of the local energy error contributions.

By now, the theoretical work on the new localized a posteriori error estimator was also completed for the parametric setting. In 2014, we finished the implementation of this estimator in our discretization framework. Naturally, the next step was to study the performance of the estimator in a parametric setting and we extended our previous experiments accordingly.

4.2.4 Detailed study of the parametric localized error estimator

To study the convergence properties of our estimator we consider two experiments [OS2015]. The first one is an extension of the experiment from Section 4.2.2 and thus serves as an academic example and as a comparison to the work of [ESV2007, ESV2010]. The second experiment is an extension of the experiment from Section 4.2.3 and demonstrates the efficiency of the parametric estimator in realistic circumstances. In both experiments we compute estimator components ${\eta_*}^2 := \sum_{T\in\mathcal{T}_H} {\eta_*^T}^2$, for $* =$ nc, r, df, and the estimator η_h as defined in Corollary 2.3.6, using a $0th$ order diffusive flux reconstruction. We denote the efficiency of the estimator, $\eta_h(p_h(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) / |||p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})|||_{\overline{\boldsymbol{\mu}}} \geq 1$, by "eff." and the average (over all refinement steps) experimental order of convergence of a quantity by "order".

In both experiments, we discretized the respective problem with the SWIPDG scheme of first order from Section 2.1 in each subdomain. For the fine grids τ_h we used conforming refinements of triangular grids, represented by instances of `ALUGrid< 2, 2, simplex, conforming >` (see [DKN2014]). All coarse triangulations $\mathcal{T}_H$ consist of equidistant squared elements (though arbitrary shapes are possible), implement with `dune-grid-multiscale`.[12]

[11] Since we did not have access to the true solution, we used an approximation on a refined grid with 128,000 elements (compare the paragraph on "comparison with a more detailed discrete solution" in Section 3.1.1.2).

[12] The implementation corresponding to these benchmarks can be obtained from the sources given in [OS2015, References 40, 41, 49, 48, 50 and 51].

4.2.4.1 Academic example

We again consider Example 1.1.2 on $\Omega = [-1, 1]^2$ with homogeneous Dirichlet boundary values, $f(x, y) = \frac{1}{2}\pi^2 \cos(\frac{1}{2}\pi x) \cos(\frac{1}{2}\pi y)$, $\lambda(x, y; \boldsymbol{\mu}) = 1 + (1 - \boldsymbol{\mu}) \cos(\frac{1}{2}\pi x) \cos(\frac{1}{2}\pi y)$, κ the identity in $\mathbb{R}^{2\times 2}$ and a parameter space $\mathcal{P} = [0.1, 1]$. This setup coincides with the one from Section 4.2.2 if we choose $\boldsymbol{\mu} = 1$ (where $\boldsymbol{\mu}$ models the online parameter in the context of reduced basis methods). Due to the design of the error estimator we have another two parameters to choose ($\overline{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\mu}}$) which are associated with the norm we estimate against and the offline/online decomposition of the estimator (compare Section 2.3.2). We thus study the components of the estimator as well as its efficiency in several circumstances, i.e., for different parameters $\boldsymbol{\mu}$, $\overline{\boldsymbol{\mu}}$, $\hat{\boldsymbol{\mu}} \in \mathcal{P}$ and triangulations τ_h and $\mathcal{T}_H$.[13]

$\lvert\tau_h\rvert$	$\lvert\lvert\lvert p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\rvert\rvert\rvert_{\overline{\boldsymbol{\mu}}}$	$\eta_{\text{nc}}(\cdot; \overline{\boldsymbol{\mu}})$	$\eta_{\text{r}}(\cdot; \boldsymbol{\mu})$	$\eta_{\text{df}}(\cdot; \boldsymbol{\mu}, \hat{\boldsymbol{\mu}})$	eff.
128	$3.28{\cdot}10^{-1}$	$1.66{\cdot}10^{-1}$	$5.79{\cdot}10^{-1}$	$3.55{\cdot}10^{-1}$	3.36
512	$1.60{\cdot}10^{-1}$	$7.89{\cdot}10^{-2}$	$2.90{\cdot}10^{-1}$	$1.76{\cdot}10^{-1}$	3.40
2,048	$7.78{\cdot}10^{-2}$	$3.91{\cdot}10^{-2}$	$1.45{\cdot}10^{-1}$	$8.73{\cdot}10^{-2}$	3.49
8,192	$3.47{\cdot}10^{-2}$	$1.95{\cdot}10^{-2}$	$7.27{\cdot}10^{-2}$	$4.35{\cdot}10^{-2}$	3.91
order	1.08	1.03	1.00	1.01	–

Table 4.1: Discretization error, estimator components and efficiency of the error estimator for the academic example in Section 4.2.4.1 with $\lvert\mathcal{T}_H\rvert = 1$ and $\boldsymbol{\mu} = \overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 1$.

		$\hat{\boldsymbol{\mu}} = 1$		$\hat{\boldsymbol{\mu}} = 0.1$		
$\lvert\tau_h\rvert$ / $\lvert\mathcal{T}_H\rvert$	$\eta_{\text{r}}(\cdot; \boldsymbol{\mu})$	$\eta(\cdot; \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})$	eff.	$\eta_{\text{df}}(\cdot; \boldsymbol{\mu}, \hat{\boldsymbol{\mu}})$	$\eta(\cdot; \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})$	eff.
128 / 2×2	$2.89{\cdot}10^{-1}$	$8.10{\cdot}10^{-1}$	2.47	$3.16{\cdot}10^{-1}$	$7.71{\cdot}10^{-1}$	2.35
512 / 4×4	$7.26{\cdot}10^{-2}$	$3.27{\cdot}10^{-1}$	2.04	$1.56{\cdot}10^{-1}$	$3.08{\cdot}10^{-1}$	1.92
2,048 / 8×8	$1.82{\cdot}10^{-2}$	$1.45{\cdot}10^{-1}$	1.86	$7.74{\cdot}10^{-2}$	$1.35{\cdot}10^{-1}$	1.73
8,192 / 16×16	$4.54{\cdot}10^{-3}$	$6.76{\cdot}10^{-2}$	1.95	$3.85{\cdot}10^{-2}$	$6.26{\cdot}10^{-2}$	1.80
order	2.00	1.20	–	1.01	1.21	–

Table 4.2: Selected estimator components, estimated error and efficiency of the error estimator for the academic example in Section 4.2.4.1 with τ_h and $\mathcal{T}_H$ simultaneously refined, $\boldsymbol{\mu} = 1$ and two choices of $\hat{\boldsymbol{\mu}}$. Note that the estimator components η_{nc} and η_{df} are not influenced by $\mathcal{T}_H$, the estimator components η_{nc} and η_{r} are not influenced by $\hat{\boldsymbol{\mu}}$ and the discretization error is not influenced by either. Thus only η_{r}, η and its efficiency are given for $\hat{\boldsymbol{\mu}} = 1$ and only η_{df}, η and its efficiency are given for $\hat{\boldsymbol{\mu}} = 0.1$ (the other quantities coincide with the ones in Table 4.1).

We choose τ_h just as in [ESV2007, Section 8.1] and Section 4.2.2 and begin with $\boldsymbol{\mu} = \overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 1$, thus reproducing the nonparametric example studied above (and in [ESV2007, Section 8.1], since $\lambda \equiv 1$ and all constants involving α and γ are equal to 1). For this specific choice of parameters an exact solution is available (see [ESV2007,

[13]The additional parameters $\overline{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\mu}}$ in particular determine the constants α and γ in Theorem 2.3.3. A suitable choice for either is thus to minimize these constants.

Section 8.1]). In this configuration, η_{nc} and η_{df} coincide with their respective counterparts defined in [ESV2007, ESV2010] while η_{r} is directly influenced by the choice of the coarse triangulation and the parametric nature of λ (entering c_ε^T). Choosing $\mathcal{T}_H = \Omega$ (the coarse grid configuration with the worst efficiency, compare Section 4.2.2 and Figure 4.8), we observe results similar to [ESV2007, Table 1] for η_{df} and η_{nc} in Table 4.1. In contrast, η_{r} shows only linear convergence while the residual estimator in [ESV2007, Table 1] converges with second order. Overall, the efficiency of the estimator η is around 3.5 (for fixed $|\mathcal{T}_H| = 1$) while the efficiency of the estimator in [ESV2007, Table 1] is around 1.2. (These observations are in line with the earlier validation study we conducted, see Section 4.2.2.)

We can recover the superconvergence of η_{r}, however, by refining $\mathcal{T}_H$ along with τ_h (thus keeping the ratio H/h fixed), see the left columns of Table 4.2. As discussed earlier, we have to fix $\hat{\boldsymbol{\mu}}$ throughout the experiment to make the estimator offline/online decomposable. Choosing $\hat{\boldsymbol{\mu}} = 0.1$ has no negative impact on the efficiency of the estimator, as we observe in the right columns of Table 4.2. Additionally, it is often desirable to fix the error norm throughout the experiments. Choosing $\overline{\boldsymbol{\mu}} = 0.1$ we still observe a very reasonable efficiency in Table 4.3.

$\lvert\tau_h\rvert$ / $\lvert\mathcal{T}_H\rvert$	$\lvert\lvert\lvert p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\rvert\rvert\rvert_{\overline{\boldsymbol{\mu}}}$	$\eta_{\text{nc}}(\cdot;\overline{\boldsymbol{\mu}})$	$\eta(\cdot;\boldsymbol{\mu},\overline{\boldsymbol{\mu}},\hat{\boldsymbol{\mu}})$	eff.
128 / 2×2	$3.81{\cdot}10^{-1}$	$1.82{\cdot}10^{-1}$	$1.18{\cdot}10^{0}$	3.10
512 / 4×4	$1.87{\cdot}10^{-1}$	$8.57{\cdot}10^{-2}$	$5.00{\cdot}10^{-1}$	2.67
2,048 / 8×8	$9.08{\cdot}10^{-2}$	$4.22{\cdot}10^{-2}$	$2.29{\cdot}10^{-1}$	2.52
8,192 / 16×16	$4.05{\cdot}10^{-2}$	$2.11{\cdot}10^{-2}$	$1.10{\cdot}10^{-1}$	2.71
order	1.08	1.03	1.14	–

Table 4.3: Discretization error, selected estimator component, estimated error and efficiency of the error estimator for the academic example in Section 4.2.4.1 with τ_h and $\mathcal{T}_H$ simultaneously refined, $\boldsymbol{\mu} = 1$ and $\overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 0.1$. Note that η_{r} and η_{df} are not influenced by $\overline{\boldsymbol{\mu}}$ and coincide with Table 4.2.

While the setup of the above experiment was rather simple, it allowed us to reconsider the influence of the coarse grid and to recover the exceptional convergence rate of the residual estimator by refining the coarse grid along with the fine grid. In addition, it allowed us to demonstrate the influence of the parametric constants α and γ as well as the two additional parameters $\overline{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\mu}}$. Following this validation of the estimator in the parametric setting the final step was to investigate the performance of the estimator in a parametric multiscale setting.

4.2.4.2 Parametric multiscale example

To investigate the sharpness as well as the localization qualities of the estimator we consider Example 1.1.2 on $\Omega = [0,5] \times [0,1]$ with homogeneous Dirichlet boundary values ($g_D = 0$) everywhere ($\Gamma_D = \partial\Omega$), f modeling one source and two sinks (see Figure 4.11, right), the parametric total mobility given by $\lambda(x,y;\boldsymbol{\mu}) = 1 + (1-\boldsymbol{\mu})\lambda_c(x,y)$ and the permeability given by $\kappa_\varepsilon = \kappa\,\text{id}$, where id denotes the unit matrix in $\mathbb{R}^{2\times 2}$.

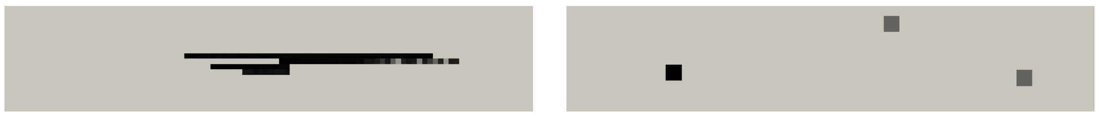
Figure 4.11: Location of the channel λ_c (left) and plot of the force f (right) modeling one source (black: $2 \cdot 10^3$ in $[0.95, 1.10] \times [0.30, 0.45]$) and two sinks (dark gray: $-1 \cdot 10^3$ in $[3.00, 3.15] \times [0.75, 0.90] \cup [4.25, 4.40] \times [0.25, 0.40]$), 0 else.

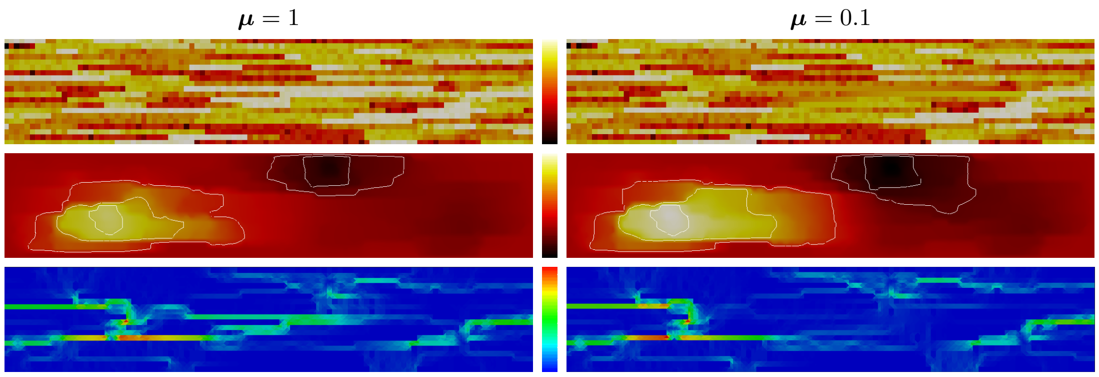

Figure 4.12: Data functions and sample solutions of the parametric multiscale example in Section 4.2.4 on a triangulation with $|\tau_h| = 16{,}000$ elements for parameters $\mu = 1$ (left column) and $\mu = 0.1$ (right column). In each row both plots share the same color map (middle) with different ranges per row. From top to bottom: logarithmic plot of $\lambda(\mu)\kappa$ (dark: $1.41 \cdot 10^{-3}$, light: $1.41 \cdot 10^3$), plot of the pressure $p_h(\mu)$ (dark: $-3.92 \cdot 10^{-1}$, light: $7.61 \cdot 10^{-1}$, isolines at 10%, 20%, 45%, 75% and 95%) and plot of the magnitude of the reconstructed diffusive flux $R_h^0[p_h(\mu); \mu]$ (blue: $3.10 \cdot 10^{-6}$, red: $3.01 \cdot 10^2$).

On each $t \in \tau_h$, $\kappa|_t$ is the corresponding 0th entry of the permeability tensor used in the first model of the 10th SPE Comparative Solution Project[14] (given by 100×20 constant tensors) and λ_c models a channel, as depicted in Figure 4.11 (left). The right hand side f models a strong source in the middle left of the domain and two sinks in the top and right middle of the domain, the influence of which is well visible in the structure of the solutions (see Figure 4.12). We consider the parameter space $\mathcal{P} = [0.1, 1]$, where the role of the parameter μ is to toggle the existence of the channel λ_c: $\mu = 0.1$ models the removal of a high-conductivity region near the center of the domain (compare Figure 4.12, top right) and for $\mu = 1$ the effective diffusivity $\lambda(\mu)\kappa_\varepsilon$ corresponds to the nonparametric one from the experiment in Section 4.2.3. The missing channel has a visible impact on the structure of the pressure distribution as well as the reconstructed

[14]`http://www.spe.org/web/csp/datasets/set01.htm`

$\lvert\tau_h\rvert$ / $\lvert\mathcal{T}_H\rvert$	$\lvert\lvert\lvert p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\rvert\rvert\rvert_{\overline{\boldsymbol{\mu}}}$	$\eta_{\mathrm{nc}}(\cdot;\overline{\boldsymbol{\mu}})$	$\eta_{\mathrm{r}}(\cdot;\boldsymbol{\mu})$	$\eta_{\mathrm{df}}(\cdot;\boldsymbol{\mu},\hat{\boldsymbol{\mu}})$	eff.
16,000 / 25×5	$7.49{\cdot}10^{-1}$	$2.13{\cdot}10^{0}$	$1.88{\cdot}10^{-9}$	$9.66{\cdot}10^{-1}$	4.14
64,000 / 50×10	$4.52{\cdot}10^{-1}$	$1.46{\cdot}10^{0}$	$7.05{\cdot}10^{-10}$	$6.05{\cdot}10^{-1}$	4.58
256,000 / 100×20	$2.58{\cdot}10^{-1}$	$1.02{\cdot}10^{0}$	$7.44{\cdot}10^{-11}$	$3.85{\cdot}10^{-1}$	5.44
1,024,000 / 200×40	$1.26{\cdot}10^{-1}$	$7.20{\cdot}10^{-1}$	$2.00{\cdot}10^{-10}$	$2.49{\cdot}10^{-1}$	7.70
order	0.86	0.52	1.07	0.65	–

Table 4.4: Discretization error, estimator components and efficiency of the error estimator for the parametric multiscale example in Section 4.2.4 with τ_h and $\mathcal{T}_H$ simultaneously refined and $\boldsymbol{\mu} = \overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 1$. Note that η_{r} should be close to zero (since f is piecewise constant, compare Figure 4.11) and suffers from numerical inaccuracies (ignoring the last refinement would yield an average order of 2.33 for η_{r}).

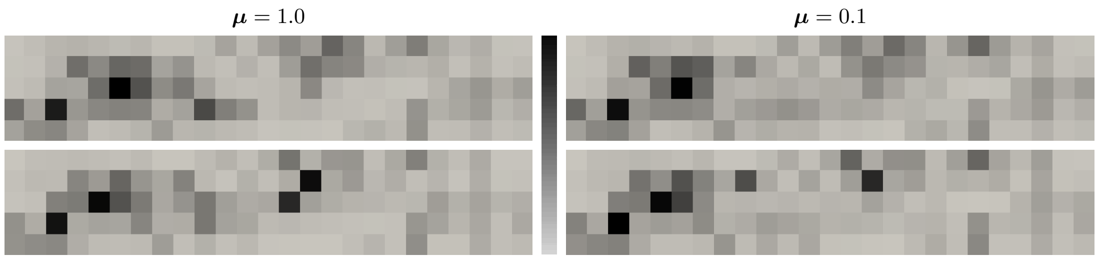

Figure 4.13: Spatial distribution of the relative error contribution $\lvert\lvert\lvert p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\rvert\rvert\rvert_{\overline{\boldsymbol{\mu}},T} / \lvert\lvert\lvert p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})\rvert\rvert\rvert_{\overline{\boldsymbol{\mu}}}$ (top) and the relative estimated error contribution $\eta^T(p_h(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) / \big(\sum_{T\in\mathcal{T}_H} \eta^T(p_h(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})^2\big)^{1/2}$ (bottom), for all $T \in \mathcal{T}_H$, for the parametric multiscale example in Section 4.2.4.2 with $\lvert\mathcal{T}_H\rvert = 25 \times 5$ and $\overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 0.1$ for parameters $\boldsymbol{\mu} = 1$ (left, light: $2.26\cdot10^{-3}$, dark: $3.78\cdot10^{-1}$) and $\boldsymbol{\mu} = 0.1$ (right, light: $4.02\cdot10^{-3}$, dark: $3.73\cdot10^{-1}$).

velocities, as we observe in the left column of Figure 4.12. With a contrast of 10^6 in the diffusion tensor and an ε of about $\lvert\Omega\rvert/2{,}000$ this setup is a challenging heterogeneous multiscale problem.

To study the convergence properties of the estimator we consider a series of refined fine and coarse grids. As we observe in Table 4.4, the convergence rates of the estimator components are not as good as in the academic experiment studied before. However, the estimator shows an average efficiency of 5.5 which is quite remarkable considering the contrast of the data functions. As already discussed above, the additional parameters $\overline{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\mu}}$ need to be fixed to obtain a fully offline/online decomposable estimate with a fixed error norm, while the online parameter $\boldsymbol{\mu}$ is allowed to vary. We thus choose $\overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 0.1$ and consider $\boldsymbol{\mu} = 1$, which yields an average efficiency of 10.2 (not shown). While the resulting estimate is obviously not as sharp as the previous one, the efficiency is still very reasonable (considering that we gain the capability to decompose the evaluation of the estimator into an offline and an online part).

We are also interested in the localization qualities of the estimator in the context of parametric multiscale problems. Since the resulting local error indicators should be fully offline/online decomposable as well we chose $\overline{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\mu}}$ as above and study the spatial distribution of the relative error contribution and the relative local indicators for a moderate-sized coarse grid: as we observe in Figure 4.13, both show a good agreement.

This last (and latest) experiment finalizes the demonstration of our discretization framework in the context of parametric multiscale problems. In particular we presented experiments regarding those aspects of the discretization framework that are concerned with local CG and SWIPDG discretizations and error estimation. What is left for the new implementation of the LRBMS are those parts of the software framework that are concerned with model reduction.

4.3 A new model reduction framework: pyMOR

As already mentioned above we started to work on pyMOR in early 2012, first on its theoretical design and soon on the actual implementation. From the beginning, one of the main design goals was to allow for an easy coupling of pyMOR with external PDE solvers, such as our discretization framework. Alongside pyMOR, we thus developed dune-pymor to provide the necessary bridging code between Python and C++ (compare Section 3.2). We performed several benchmark experiments of pyMOR's core constructs in [MRS2015][15]. We also demonstrated other aspects and applications of pyMOR in [MRS2015], in particular two examples concerning different parallelization paradigms (compare [MRS2015, Sections 5.2 and 5.3]).[16]

The main goal of our benchmarks is to compare the performance of pyMOR's VectorArray and Operator interfaces, when used to access external high-dimensional solver data structures, to native NumPy based 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 for 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 scalar products in the gram_schmidt and pod benchmarks are implemented with NumpyMatrixOperators holding sparse SciPy matrices coming from pyMOR's own discretization toolbox.

The external solver code is based on our discretization framework discussed in Section 3.1. In particular we used vectors, matrices and linear solvers from dune-stuff and parametric discretizations (used here to obtain appropriate scalar products for the gram_schmidt and pod benchmarks) from dune-hdd, which in turn is based on dune-gdt. The integration of the **DUNE** code with pyMOR is done by compiling the solver as a Python extension module using the infrastructure provided by dune-pymor.

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.

4.3.1 Vector array benchmarks

We consider the gramian and axpy methods of the VectorArrayInterface. The first computes the Gramian of the vectors contained in the array A, i.e. the Euclidean scalar products of all combinations of vectors contained in the array, while the latter performs a vectorized BLAS-conforming axpy operation, i.e. pairwise in-place addition of the vectors in the array with vectors of a second array multiplied by a scalar factor.

[15]Regarding the contributions in [MRS2015]: while the publication [MRS2015] was mainly written by S. Rave, F. Schindler was mainly responsible for the benchmarks presented in this section.

[16]The implementation corresponding to these benchmarks can be obtained from the sources given in [MRS2015, References 22, 32, 33, 34, 35 and 36].

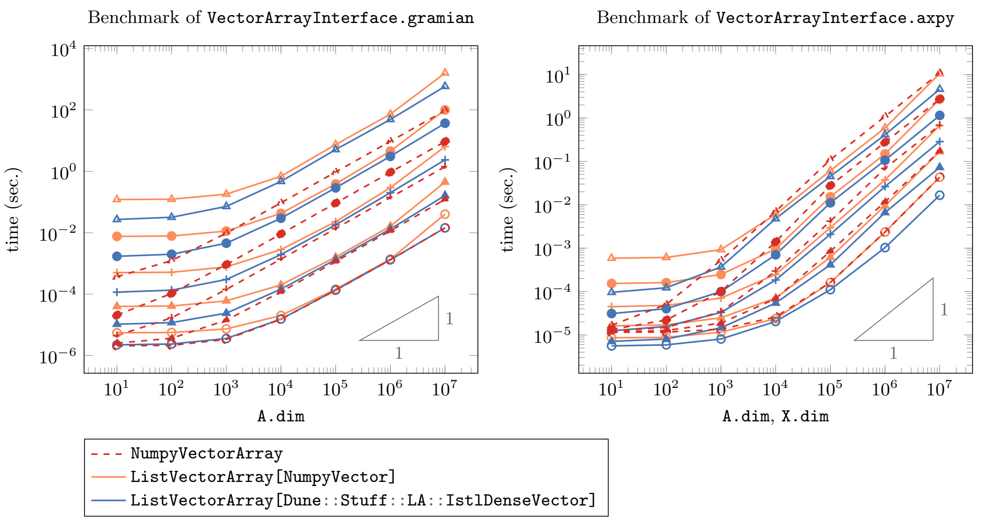

Figure 4.14: Log/log plot of the measured execution time of `A.gramian()` (left) and `A.axpy(X)` (right, with `len(X)==len(A)`) for different implementations and several lengths (`len(A)==1`: ○, `len(A)==4`: ▲, `len(A)==16`: +, `len(A)==64`: ●, `len(A)==256`: △).

In case of the `gramian` method, as we observe in Figure 4.14 (left), both `ListVectorArray`-based implementations show about the same performance for sufficiently large array dimensions. In fact, the **DUNE**-based implementation (`IstlDenseVector`) actually performs better than the NumPy-based implementation (`NumpyVector`), showing the performance of our tight integration between `pyMOR` and the external solver. The vectorized `NumpyVectorArray` implementation, however, clearly outperforms both implementations for sufficiently large array lengths. 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.

As we observe in Figure 4.14 (right), in case of the `axpy` method the `NumpyVectorArray` implementation does not benefit from vectorization for larger array dimensions. We 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. Here, the **DUNE**-based implementation performs better than both of `pyMOR`'s native implementations.

4.3.2 Gram-Schmidt and POD benchmarks

The Gram-Schmidt and POD algorithms are important tools in the context of model reduction. The implementation of a numerically stable Gram-Schmidt algorithm or a correct POD might not be completely straightforward (compare the corresponding discussion in Section 4.1.1) and it is an important benefit that `pyMOR`'s interface design allows to provide tried and tested implementations of these algorithms which can automatically be used with any external solver integrated with pyMOR.

Nevertheless, compared to a native implementation within the external solver, the algorithms provided by pyMOR introduce a certain overhead since every operation on the solver data structures has to pass through wrapper classes implementing VectorArray or Operator interfaces. The main purpose of the gram_schmidt benchmark is thus to measure the overhead introduced by these interface method calls, by comparing pyMOR's gram_schmidt algorithm operating on VectorArrays and Operators with native implementations of the identical algorithm: a C++ implementation operating directly on the **DUNE** vectors within the ListVectorArray[IstlDenseVector] and two Python implementations operating directly on the NumPy arrays contained within NumpyVectorArray and ListVectorArray[NumpyVector].

The POD algorithm mainly consist of three steps with different complexities: 1. computation of a Gramian matrix with respect to the given scalar product (by calling 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 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 VectorArray implementation and only increases with the number of given vectors.

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

As expected, we observe in Figure 4.15 (top left) that for the Gram-Schmidt algorithm the native implementations (dashed) are faster than the generic variants (solid). For higher dimensions, however, the overhead of the generic variant becomes negligible.

For the POD algorithm we can observe again in Figure 4.15 (top right) that both ListVectorArray-based implementation show roughly equal performance. However, the vectorized pyMOR implementation is able to outperform both other implementations thanks to the fact that the computationally dominant steps 1 and 3 of the algorithm (see Figure 4.15, middle and bottom) can be expressed idiomatically via a single interface call.

Considering the benchmark results, we can clearly observe a performance benefit for vectorized VectorArray implementations such as NumpyVectorArray. This justifies our design decision of choosing arrays of vectors (over single vectors) as basic classes for vector data in pyMOR. 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.

In view of the numerical instabilities we observed in our earliest experiments (Section 4.1) it is of particular importance to have access to the numerically stable and thoroughly tested Gram-Schmidt algorithm from pyMOR, since the benefit of numerical stability greatly outweighs the slight performance overhead of using this generic algorithm.

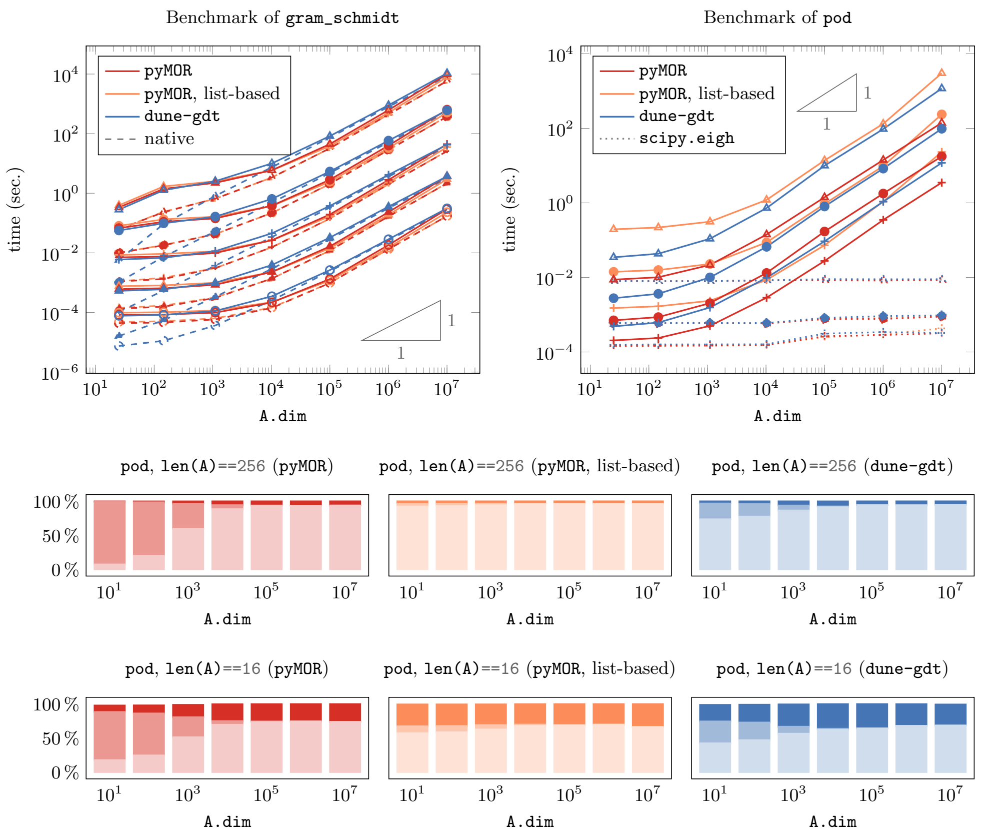

Figure 4.15: Log/log plot of the measured execution time of the Gram-Schmidt (top left) and POD (top right) algorithms for several lengths of the vector array (`len(A)==1`: ○, `len(A)==4`: ▲, `len(A)==16`: +, `len(A)==64`: ●, `len(A)==256`: △) and breakdown of individual parts of the POD algorithm (middle and bottom). Top left: Comparison of `gram_schmidt(A=A, product=h1, check=False, reiterate=False)` for several implementations of `A` and `product` (solid) and the respective native implementation (dashed). Top right: Comparison of `pod(A=A, modes=10, product==h1, orthonormalize=False, check=False)` for several implementations of `A` and `product` (solid) and the respective time spent in `scipy.eigh` (dotted). Middle and bottom: Comparison of the relative time spent in the three parts of the POD algorithm (`product.apply2`: light, `scipy.eigh`: medium, `A.lincomb`: dark) for `len(A)==256` (middle) and `len(A)==16` (bottom).

4.4 The online adaptive LRBMS

We are now in the position where all building blocks for the online adaptive LRBMS (compare Section 2.4) are readily implemented and thoroughly tested. By combining the discretization and model reduction components (`dune-stuff`, `dune-grid-multiscale`, `dune-gdt`, `dune-pymor`, `dune-hdd` and `pyMOR`), presented in Chapter 3, we obtain a software framework that combines the computational power of **DUNE** (implemented in the system language C++) with the flexibility of `pyMOR` (implemented in the interpreted language `Python`).

To demonstrate the proposed adaptive online enrichment Algorithm 2.4.2 and the flexibility of the LRBMS we study two distinct circumstances [OS2015, Section 6.2]. We first consider again the academic example from Section 4.2.4.1 where we disable the greedy procedure and build the reduced bases only by local enrichment online. The second example is again the parametric multiscale one from Section 4.2.4.2 with global channels in the permeability and we allow for very few global solution snapshots offline and adaptively enrich afterwards online.[17]

For the orthonormalization procedure `ONB` in the greedy Algorithm 2.4.1 we use the stabilized Gram Schmidt procedure implemented in `pyMOR`[18] with respect to the scalar product given by $(p, q) \mapsto b^T(p, q; \overline{\boldsymbol{\mu}}) + \sum_{e \in \mathcal{F}_h^T} b_p^T(p, q; \overline{\boldsymbol{\mu}})$ on each $T \in \mathcal{T}_H$. In contrast to other possible basis extension algorithms (e.g., a proper orthogonal decomposition) the Gram Schmidt basis extension yields hierarchical local reduced bases. This is of particular interest in the context of online enrichment, since we can reuse reduced quantities w.r.t. existing basis vectors after enrichment.

In all experiments, we initialize the local reduced bases with the coarse DG basis of order up to one by setting $k_H = 1$ in Algorithm 2.4.1. We choose the same orthonormalization algorithm in the adaptive basis enrichment Algorithm 2.4.2 and consider several marking strategies for `MARK`, depending on the circumstances (detailed below). Regarding the overlap for the local enrichment we always chose the overlapping subdomains $T_\delta \supset T$ to include T and all subdomains that touch it, thus choosing an overlap of $\mathcal{O}(H)$, as motivated in [HP2013]. We also chose the coarse triangulation to be fine enough to represent the coarse behavior of the solution, since in previous Experiments (compare Section 4.2.1) we observed that a small coarse grid together with small overlaps lead to extensive online computations in terms of the number of required enrichment iterations.

Since the error of any reduced solution $|||p(\boldsymbol{\mu}) - p_{\text{red}}(\boldsymbol{\mu})|||_{\overline{\boldsymbol{\mu}}}$ cannot be lower than the error of the corresponding detailed solution $|||p(\boldsymbol{\mu}) - p_h(\boldsymbol{\mu})|||_{\overline{\boldsymbol{\mu}}}$, we choose Δ_{red} in Algorithm 2.4.2 to be slightly larger than $\max_{\boldsymbol{\mu} \in \mathcal{P}_{\text{online}}} \eta_h(p_h(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})$ in our experiments (see below), where $\mathcal{P}_{\text{online}} \subset \mathcal{P}$ is the set of all parameters we consider during the online phase. This is only necessary since we do not allow for an adaptation of τ_h; combining our online adaptive LRBMS with the ideas of [Yan2014] would overcome this restriction (compare the discussion in Section 1.3).

[17]We use the same software configuration as detailed in Section 4.2.4.

[18]`http://docs.pymor.org/en/0.3.x/generated/pymor.la.html#module-pymor.la.gram_schmidt`

4.4.1 Academic example

We again consider the academic example from Section 4.2.4.1 on fixed triangulations with $|\tau_h| = 131{,}072$ and $|\mathcal{T}_H| = 8 \times 8$ elements and choose the set of online parameters $\mathcal{P}_{\text{online}}$ to consist of 10 randomly chosen parameters $\boldsymbol{\mu}_0, \dots, \boldsymbol{\mu}_9 \in \mathcal{P}$. We set $n_{\max} = 0$ and $k_H = 1$, thus disabling any greedy search and initializing the local bases with the coarse DG basis of order up to one (consisting of 4 shape functions). In this setup $\max_{\boldsymbol{\mu} \in \mathcal{P}_{\text{online}}} \eta_h(p_h(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) = 2.79 \cdot 10^{-2}$ and we choose $\Delta_{\text{red}} = 5 \cdot 10^{-2}$ in Algorithm 2.4.2.

We begin by choosing `MARK` such that $\tilde{\mathcal{T}}_H = \mathcal{T}_H$ (all coarse elements are marked; denoted by `uniform` in the following). For each parameter $\boldsymbol{\mu} \in \mathcal{P}_{\text{online}}$ this results in a method that is similar to domain decomposition (DD) methods with overlapping subdomains. In contrast to traditional DD methods, however, we start with an initial coarse basis and perform a reduced solve before each iteration which helps to spatially spread information. As we observe in Figure 4.16, top left, it takes four enrichment steps (or equivalently four DD iterations) to lower the estimated error for the first online parameter $\boldsymbol{\mu}_0$ below the desired tolerance and another two enrichment steps for the next on-line parameter $\boldsymbol{\mu}_1$ (which is $\max_{\boldsymbol{\mu} \in \mathcal{P}_{\text{online}}} \boldsymbol{\mu}$). With uniform marking this increases the local basis sizes from four to ten on each coarse element. The resulting coarse reduced space of dimension 640 is then sufficient to solve for the next four online parameters $\boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_5$ without enrichment. One additional enrichment phase is needed for $\boldsymbol{\mu}_6$ (which is $\min_{\boldsymbol{\mu} \in \mathcal{P}_{\text{online}}} \boldsymbol{\mu}$) and none for the remaining three online parameters. Note that while this uniform marking strategy may be optimal in the number of enrichment steps it takes to reach the desired error for all online parameters it also leads to an unnecessarily high-dimensional coarse reduced space (of $\dim Q_{\text{red}}(\mathcal{T}_H) = 704$) and a high work-load in each enrichment step.

Another popular choice in the context of adaptive mesh refinement is a Dörfler marking strategy (see [MNS2002] and the references therein), where we collect those coarse elements that contribute most to $\theta_{\text{doerf}} \sum_{T \in \mathcal{T}_H} \eta^T(\cdot; \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}})^2$ in $\tilde{\mathcal{T}}_H \subseteq \mathcal{T}_H$, for a given user-dependent parameter $0 < \theta_{\text{doerf}} \leq 1$. In addition, similar to [BDD2004, HDO2011], we count how often each $T \in \mathcal{T}_H$ was not marked and mark those elements the "age" of which is larger than a prescribed $N_{\text{age}} \in \mathbb{N}$ (resetting the age count of each selected element). We denote this marking strategy by `doerfler_age(`$\theta_{\text{doerf}}, N_{\text{age}}$`)`. We found that a combination of $\theta_{\text{doerf}} = 1/3$ and $N_{\text{age}} = 4$ yielded the smallest overall basis size (of 572), compared to other combinations of θ_{doerf} and N_{age} and the `uniform` marking strategy. The number of elements marked per step range between five and 52 (over all online parameters and all enrichment steps; 23 steps in total) with a mean of 14 and a median of ten. Of these marked elements between one and 44 have been marked due to their age in 12 of these 23 steps (with an average of 12 and a median of eight, taken over only those 12 steps where elements have been marked due to their age). We observe in Figure 4.16, top right, that the general behavior of the method with this marking strategy is similar to the one with `uniform` marking, with some commonalities and differences worth noting. First of all it also takes three enrichment phases to reach the prescribed error tolerance, and for the same parameters $\boldsymbol{\mu}_0$, $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_6$ as above. But

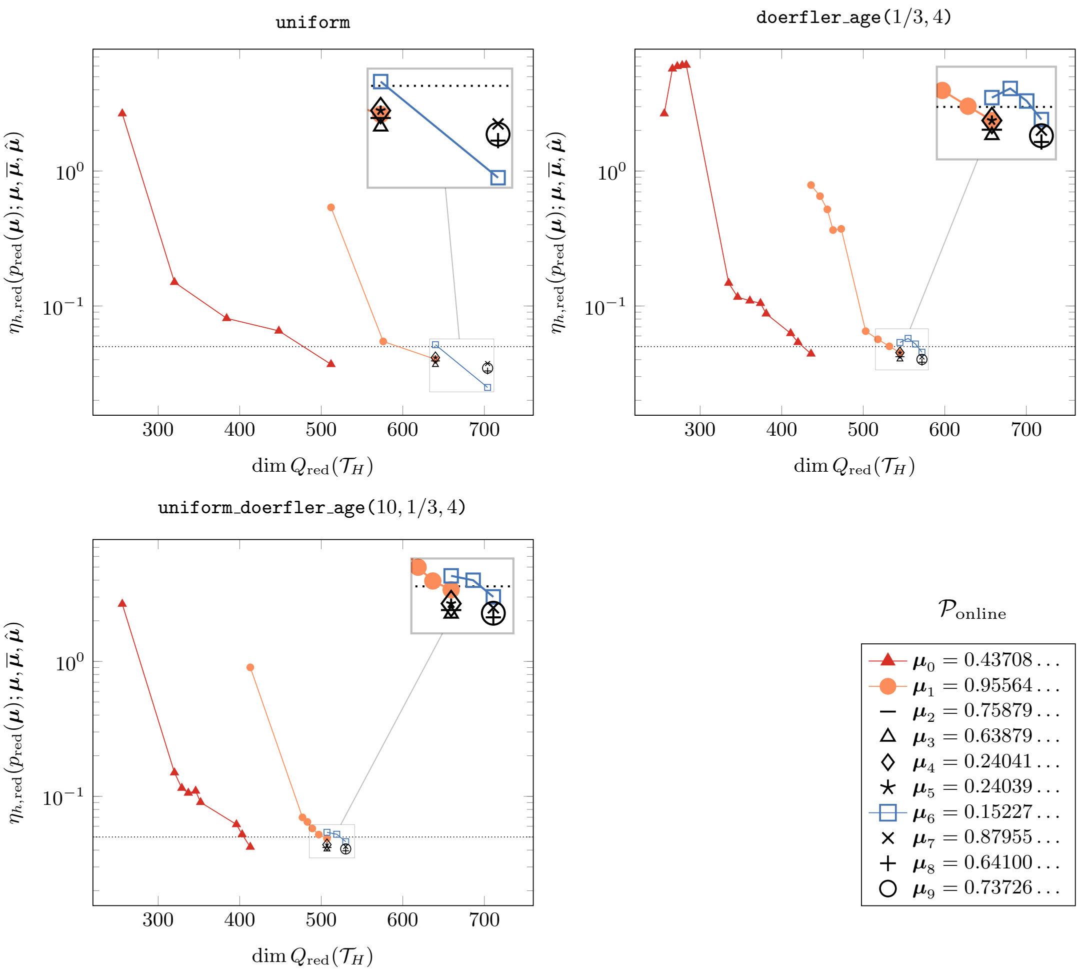

Figure 4.16: Estimated error evolution during the adaptive online phase for the academic example in Section 4.4.1 with $|\mathcal{T}_H| = 64$, $n_{\max} = 0$, $k_H = 1$, $\Delta_{\mathrm{red}} = 5 \cdot 10^{-2}$ (dotted line) and $\overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 0.1$ for several marking strategies: uniform marking of all subdomains (top left), combined Dörfler and age-based marking with $\theta_{\mathrm{doerf}} = 1/3$ and $N_{\mathrm{age}} = 4$ (top right) and additional uniform marking while $\eta(p_{\mathrm{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) > \theta_{\mathrm{uni}} \Delta_{\mathrm{red}}$ with $\theta_{\mathrm{uni}} = 10$ (bottom left). With each strategy the local reduced bases are enriched according to Algorithm 2.4.2 for each subsequently processed online parameter $\boldsymbol{\mu}_0, \dots, \boldsymbol{\mu}_9$ (bottom right).

each of these enrichment phases naturally needs more steps and large improvements can usually be observed after a lot of elements have been marked due to their age count (see for instance the fifth enrichment step for $\boldsymbol{\mu}_0$ or $\boldsymbol{\mu}_1$). In addition we observe that the estimated error for a parameter sometimes increases, in particular in the very beginning (see the first four steps for $\boldsymbol{\mu}_0$, the fourth step for $\boldsymbol{\mu}_1$ or the first step for $\boldsymbol{\mu}_6$). This is not troublesome since we can only expect a strict improvement in the energy norm induced by the bilinear form that is used in the enrichment. This shows, never the less, that there is still room for improvement, although using the `doerfler_age` marking we could reach a significantly lower overall basis size than using the `uniform` marking (572 vs. 704).

We propose a combination of the two strategies, namely a uniform marking while the estimated error is far away from the desired tolerance, i.e., $\eta(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) > \theta_{\text{uni}}\,\Delta_{\text{red}}$ for some $\theta_{\text{uni}} > 0$, followed by a Dörfler and age-based marking as detailed above. We denote this marking strategy by `uniform_doerfler_age(`$\theta_{\text{uni}}, \theta_{\text{doerf}}, N_{\text{age}}$`)`. As we observe in Figure 4.16, bottom left, this marking strategy combines advantages of both previous approaches, recovering the rapid error decrease of the `uniform` marking strategy far away from the desired tolerance (see the first step for $\boldsymbol{\mu}_0$ and $\boldsymbol{\mu}_1$) while yielding the smallest overall basis size of 530 (using a factor of $\theta_{\text{uni}} = 10$) due to the `doerfler_age` marking strategy. The smoothness and symmetry of the problem is reflected in the spatial distribution of the final local basis sizes (see Figure 4.17).

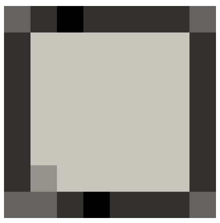
Figure 4.17: Spatial distribution of the final sizes of the local reduced bases, $|\Phi^T|$ (light: 7, dark: 11), for all $T \in \mathcal{T}_H$ after the adaptive online phase for the academic example in Section 4.4.1 with $\Omega = [-1,1]^2$, $|\mathcal{T}_H| = 8 \times 8$ and the `uniform_doerfler_age(`$10, 1/3, 4$`)` marking strategy (see Figure 4.16, bottom left).

4.4.2 Parametric multiscale example

We again consider the multiscale example from Section 4.2.4.2 on fixed triangulations with $|\tau_h| = 1{,}014{,}000$ and $|\mathcal{T}_H| = 25 \times 5$ and choose the set of online parameters $\mathcal{P}_{\text{online}}$ to consist of the same 10 randomly chosen parameters $\boldsymbol{\mu}_0, \dots, \boldsymbol{\mu}_9 \in \mathcal{P}$ as in the previous example. In this setup $\max_{\boldsymbol{\mu} \in \mathcal{P}_{\text{online}}} \eta_h(p_h(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) = 1.66$ and we choose $\Delta_{\text{red}} = 2$ in Algorithm 2.4.2.

We first set $n_{\text{max}} = 0$ and $k_H = 1$ and thus disable any greedy search in the offline phase and initializing the local bases with the coarse DG basis of order up to one; in the online phase we use the `uniform` marking strategy (see above). This results in the same DD-like approach that worked well for the academic example from the previous section. As we observe in Figure 4.18 (top), however, it takes 129 enrichment steps to lower the estimated error below the desired tolerance for the first online parameter $\boldsymbol{\mu}_0$. After this extensive enrichment it takes 12 steps for $\boldsymbol{\mu}_1$ and none or one enrichment steps to reach the desired tolerance for the other online parameters. The resulting coarse reduced space

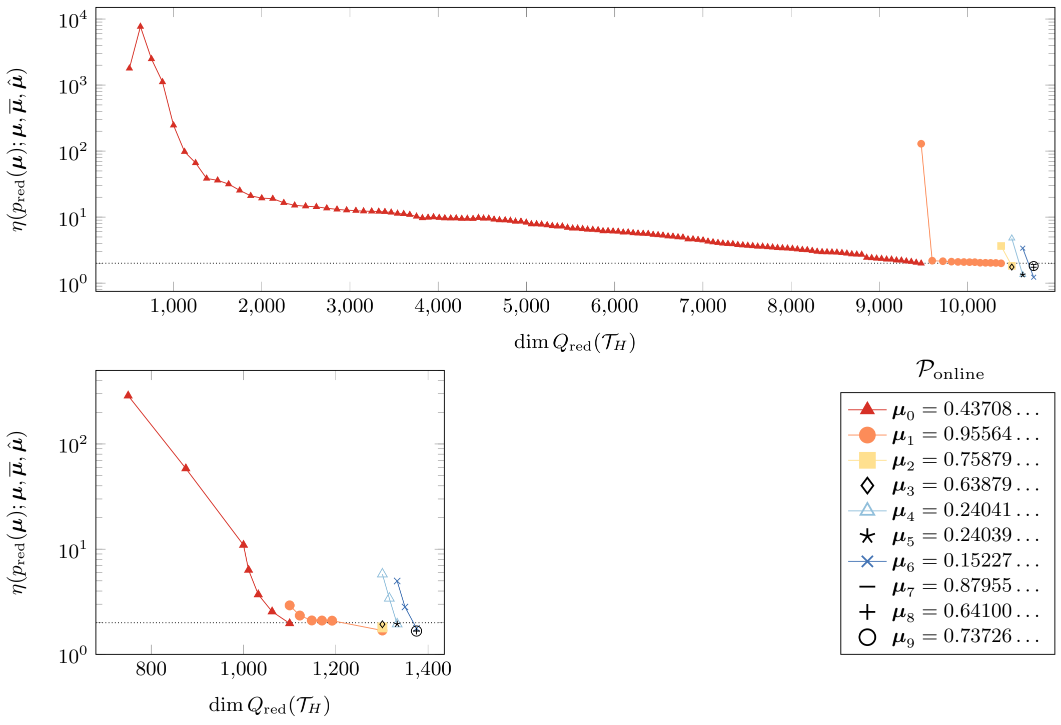

Figure 4.18: Estimated error evolution during the adaptive online phase for the parametric multiscale example in Section 4.4.2 with $|\mathcal{T}_H| = 125$, $k_H = 1$, $\Delta_{\text{online}} = 2$ (dotted line), $\overline{\boldsymbol{\mu}} = \hat{\boldsymbol{\mu}} = 0.1$, for different on-line and offline strategies: no global snapshot (greedy search disabled, $N_{\text{greedy}} = 0$) during the offline phase, uniform marking during the online phase (top) and two global snapshots (greedy search on $\mathcal{P}_{\text{train}} = \{0.1, 1\}$, $N_{\text{greedy}} = 2$) and combined uniform marking while $\eta(p_{\text{red}}(\boldsymbol{\mu}); \boldsymbol{\mu}, \overline{\boldsymbol{\mu}}, \hat{\boldsymbol{\mu}}) > \theta_{\text{uni}}\Delta_{\text{online}}$ with $\theta_{\text{uni}} = 10$, Dörfler marking with $\theta_{\text{doerf}} = 0.85$ and age-based marking with $N_{\text{age}} = 4$ (bottom left); note the different scales. With each strategy the local reduced bases are enriched according to Algorithm 2.4.2 while subsequently processing the online parameters $\boldsymbol{\mu}_0, \ldots, \boldsymbol{\mu}_9$ (bottom right).

is of size 10,749 (with an average of 86 basis functions per subdomain), which is clearly not optimal. Although each subdomain was marked for enrichment, the sizes of the final local reduced bases differ since the local Gram Schmidt basis extension may reject updates (if the added basis function is locally not linearly independent). As we observe in Figure 4.19 (left) this is indeed the case with local basis sizes ranging between 24 and 148. Obviously, a straightforward domain decomposition ansatz without any global solution snapshots is not feasible for this setup. This is not surprising since the data functions exhibit strong multiscale features and non-local high-conductivity channels connecting domain boundaries, see Figure 4.12.

To remedy the situation we allow for two global snapshots during the offline phase (setting $n_{\max} = 2$, $\mathcal{P}_{\text{train}} = \{0.1, 1\}$) and use the adaptive `uniform_doerfler_age` marking strategy (see above) in the online phase. With two global solution snapshots incorporated in the basis the situation improves significantly, as we observe in Figure 4.18 (bottom left) and there is no qualitative difference of the evolution of the estimated error during the adaptive online phase between the academic example studied above and this highly heterogeneous multiscale example (compare Figure 4.16, bottom left). In total we observe only two enrichment steps with uniform marking (see the first two steps for $\boldsymbol{\mu}_0$). The number of elements marked range between 11 and 110 (over all online parameters and all but the first two enrichment steps) with a mean of 29 and a median of 22. Of these marked elements only once have 87 out of 110 elements been marked due to their age (see the last step for $\boldsymbol{\mu}_1$). Overall we could reach a significantly lower overall basis size than in the previous setup (1,375 vs. 10,749) and the sizes of the final local bases range between only nine and 20 (compared to 24 to 148 above). We also observe in Figure 4.19 (right) that the spatial distribution of the basis sizes follows the spatial structure of the data functions (compare Figures 4.11, 4.12), which nicely shows the localization qualities of our error estimator.

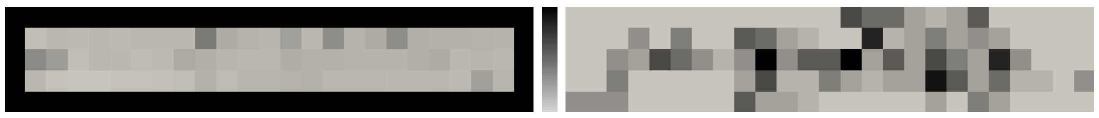

Figure 4.19: Spatial distribution of the final sizes of the local reduced bases, $|\Phi^T|$ for all $T \in \mathcal{T}_H$, after the adaptive online phase for the parametric multiscale example in Section 4.4.2 with $\Omega = [0, 5] \times [0, 1]$, $|\mathcal{T}_H| = 25 \times 5$ for the two strategies shown in Figure 4.18: no global snapshot with uniform enrichment (left, light: 24, dark: 148) and two global snapshots with adaptive enrichment (right, light: 9, dark: 20). Note the pronounced structure (right) reflecting the spatial structure of the data functions (compare Figures 4.11 and 4.12).

It is clear that these experiments do not cover all aspect of the LRBMS in all details. In particular, we did not study the possible computational gain of an offline/online decomposition of the localized error estimator. Instead we rather focused on those aspects of the LRBMS that go beyond traditional reduced basis methods: localized error control and online enrichment of the reduced basis. We could demonstrate that our new localized a posteriori error estimator allows to identify problematic local reduced bases and that the localized ansatz in general allows to enrich the local reduced basis online without resorting to full high-dimensional computations. This combination enables the LRBMS to be applicable in a far wider range of scenarios than traditional RB methods.

Bibliography

[AE2008] J. E. Aarnes, Y. Efendiev. *Mixed multiscale finite element methods for stochastic porous media flows*. SIAM J. Sci. Comput., 30 (2008) (5), pp. 2319–2339. doi:10.1137/07070108X.

[AEJ2008] J. E. Aarnes, Y. Efendiev, L. Jiang. *Mixed multiscale finite element methods using limited global information*. Multiscale Model. Simul., 7 (2008) (2), pp. 655–676. doi:10.1137/070688481.

[ALKK2009] J. E. Aarnes, K.-A. Lie, V. Kippe, S. Krogstad. *Multiscale methods for subsurface flow*. In *Multiscale modeling and simulation in science*, vol. 66 of *Lect. Notes Comput. Sci. Eng.*, pp. 3–48. Springer, Berlin, 2009. doi:10.1007/978-3-540-88857-4_1.

[AB2013] A. Abdulle, Y. Bai. *Adaptive reduced basis finite element heterogeneous multiscale method*. Comput. Methods Appl. Mech. Engrg., 257 (2013), pp. 203–220. doi:10.1016/j.cma.2013.01.002.

[AB2014] A. Abdulle, Y. Bai. *Reduced-order modelling numerical homogenization*. Philos. Trans. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci., 372 (2014) (2021, 20130388). doi:10.1098/rsta.2013.0388.

[AH2014] A. Abdulle, P. Henning. *A reduced basis localized orthogonal decomposition*. ArXiv e-prints [math.NA], (2014). 1410.3253.

[AN2009] A. Abdulle, A. Nonnenmacher. *A posteriori error analysis of the heterogeneous multiscale method for homogenization problems*. C. R. Math. Acad. Sci. Paris, 347 (2009) (17-18), pp. 1081–1086. doi:10.1016/j.crma.2009.07.004.

[AN2011] A. Abdulle, A. Nonnenmacher. *Adaptive finite element heterogeneous multiscale method for homogenization problems*. Comput. Methods Appl. Mech. Engrg., 200 (2011) (37-40), pp. 2710–2726. doi:10.1016/j.cma.2010.06.012.

[AHKO2012] F. Albrecht, B. Haasdonk, S. Kaulmann, M. Ohlberger. *The localized reduced basis multiscale method*. In *Proceedings of Algoritmy 2012, Conference on Scientific Computing, Vysoke Tatry, Podbanske, September 9-14, 2012*. Slovak University of Technology in Bratislava, Publishing House of STU, 2012 pp. 393–403.

[AO2013] F. Albrecht, M. Ohlberger. *The localized reduced basis multi-scale method with online enrichment*. Oberwolfach Rep., 7 (2013), pp. 12–15. doi:10.4171/OWR/2013/07.

[All1992] G. Allaire. *Homogenization and two-scale convergence*. SIAM J. Math. Anal., 23 (1992) (6), pp. 1482–1518. doi:10.1137/0523084.

[ASB1978] B. O. Almroth, P. Stern, F. A. Brogan. *Automatic choice of global shape functions in structural analysis*. AIAA J., 16 (1978) (5), pp. 525–528.

[AF2011] D. Amsallem, C. Farhat. *An online method for interpolating linear parametric reduced-order models*. SIAM J. Sci. Comput., 33 (2011) (5), pp. 2169–2198. doi:10.1137/100813051.

[Ant1972] S. N. Antontsev. *On the solvability of boundary value problems for degenerate two-phase porous flow equations*. Dinamika Splosnoi Sredy Vyp, 10 (1972), pp. 28–53.

[BD1981] I. Babuška, M. R. Dorr. *Error estimates for the combined h and p versions of the finite element method*. Numer. Math., 37 (1981) (2), pp. 257–277. doi:10.1007/BF01398256.

[BSK1981] I. Babuška, B. A. Szabo, I. N. Katz. *The p-version of the finite element method*. SIAM J. Numer. Anal., 18 (1981) (3), pp. 515–545. doi:10.1137/0718033.

[BV1984] I. Babuška, M. Vogelius. *Feedback and adaptive finite element solution of one-dimensional boundary value problems*. Numer. Math., 44 (1984) (1), pp. 75–102. doi:10.1007/BF01389757.

[BHH+2015] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, T. D. Young. *The `deal.II` Library, Version 8.2*. Archive of Numerical Software, 3 (2015).

[BMNP2004] M. Barrault, Y. Maday, N. C. Nguyen, A. T. Patera. *An 'empirical interpolation' method: application to efficient reduced-basis discretization of partial differential equations*. C. R. Math. Acad. Sci. Paris, 339 (2004) (9), pp. 667–672. doi:10.1016/j.crma.2004.08.006.

[BBD+2008a] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, O. Sander. *A generic grid interface for parallel and adaptive scientific computing. II. Implementation and tests in DUNE*. Computing, 82 (2008) (2-3), pp. 121–138. doi:10.1007/s00607-008-0004-9.

[BBD+2008] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander. *A generic grid interface for parallel and adaptive scientific computing. I. Abstract framework*. Computing, 82 (2008) (2-3), pp. 103–119. doi:10.1007/s00607-008-0003-x.

[BBBG2011] U. Baur, C. Beattie, P. Benner, S. Gugercin. *Interpolatory projection methods for parameterized model reduction*. SIAM J. Sci. Comput., 33 (2011) (5), pp. 2489–2518. doi:10.1137/090776925.

[BBH+2015] U. Baur, P. Benner, B. Haasdonk, C. Himpe, I. Martini, M. Ohlberger. *Comparison of methods for parametric model order reduction of instationary problems*. Tech. rep., Max Planck Institute Magdeburg, 2015.

[BHL1993] G. Berkooz, P. Holmes, J. L. Lumley. *The proper orthogonal decomposition in the analysis of turbulent flows*. In *Annual review of fluid mechanics, Vol. 25*, pp. 539–575. Annual Reviews, Palo Alto, CA, 1993.

[BCD+2011] P. Binev, A. Cohen, W. Dahmen, R. DeVore, G. Petrova, P. Wojtaszczyk. *Convergence rates for greedy algorithms in reduced basis methods*. SIAM J. Math. Anal., 43 (2011) (3), pp. 1457–1472. doi:10.1137/100795772.

[BDD2004] P. Binev, W. Dahmen, R. DeVore. *Adaptive finite element methods with convergence rates*. Numer. Math., 97 (2004) (2), pp. 219–268. doi:10.1007/s00211-003-0492-7.

[Boy2008] S. Boyaval. *Reduced-basis approach for homogenization beyond the periodic setting*. Multiscale Model. Simul., 7 (2008) (1), pp. 466–494. doi:10.1137/070688791.

[BF1991] F. Brezzi, M. Fortin. *Mixed and hybrid finite element methods*. Springer-Verlag New York, Inc., 1991.

[Buh2000] M. D. Buhmann. *Radial basis functions*. In *Acta numerica, 2000*, vol. 9 of *Acta Numer.*, pp. 1–38. Cambridge Univ. Press, Cambridge, 2000. doi:10.1017/S0962492900000015.

[BEOR2014] A. Buhr, C. Engwer, M. Ohlberger, S. Rave. *A Numerically Stable A Posteriori Error Estimator for Reduced Basis Approximations of Elliptic Equations*. In E. Onate, X. Oliver, A. Huerta, eds., *Proceedings of the 11th World Congress on Computational Mechanics*. CIMNE, Barcelona, 2014 pp. 4094–4102.

[BZ2006] E. Burman, P. Zunino. *A domain decomposition method based on weighted interior penalties for advection-diffusion-reaction problems*. SIAM J. Numer. Anal., 44 (2006) (4), pp. 1612–1638 (electronic). doi:10.1137/050634736.

[Car2015] K. Carlberg. *Adaptive h-refinement for reduced-order models*. Internat. J. Numer. Methods Engrg., 102 (2015) (5), pp. 1192–1210. doi:10.1002/nme.4800.

[CFCA2013] K. Carlberg, C. Farhat, J. Cortial, D. Amsallem. *The GNAT method for nonlinear model reduction: effective implementation and application to computational fluid dynamics and turbulent flows*. J. Comput. Phys., 242 (2013), pp. 623–647. doi:10.1016/j.jcp.2013.02.028.

[CS2010] S. Chaturantabut, D. C. Sorensen. *Nonlinear model reduction via discrete empirical interpolation*. SIAM J. Sci. Comput., 32 (2010) (5), pp. 2737–2764.

[CJ1986] G. Chavent, J. Jaffré. *Mathematical Models and Finite Elements for Reservoir Simulation: Single Phase, Multiphase and Multicomponent Flows through Porous Media*. Studies in Mathematics and its Applications, Elsevier Science, 1986. ISBN 978-0-444-70099-5.

[Che2001] Z. Chen. *Degenerate two-phase incompressible flow. I. Existence, uniqueness and regularity of a weak solution*. J. Differential Equations, 171 (2001) (2), pp. 203–232. doi:10.1006/jdeq.2000.3848.

[Che2002] Z. Chen. *Degenerate two-phase incompressible flow. II. Regularity, stability and stabilization*. J. Differential Equations, 186 (2002) (2), pp. 345–376. doi:10.1016/S0022-0396(02)00027-X.

[CHM2006] Z. Chen, G. Huan, Y. Ma. *Computational methods for multiphase flows in porous media*, vol. 2. Siam, 2006.

[CS2008] Z. Chen, T. Y. Savchuk. *Analysis of the multiscale finite element method for nonlinear and random homogenization problems*. SIAM J. Numer. Anal., 46 (2008) (1), pp. 260–279. doi:10.1137/060654207.

[Cia1978] P. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland Publishing Company, 1978. ISBN 9780898715149.

[CKSS2002] B. Cockburn, G. Kanschat, D. Schötzau, C. Schwab. *Local discontinuous Galerkin methods for the Stokes system*. SIAM J. Numer. Anal., 40 (2002) (1), pp. 319–343. doi:10.1137/S0036142900380121.

[CD2015] A. Cohen, R. DeVore. *Kolmogorov widths under holomorphic mappings*. ArXiv e-prints [math.AP], (2015). `1502.06795`.

[Cop1995] J. O. Coplien. *Curiously recurring template patterns*. C++ Report, 7 (1995) (2), pp. 24–27.

[DPW2014] W. Dahmen, C. Plesken, G. Welper. *Double greedy algorithms: reduced basis methods for transport dominated problems*. ESAIM Math. Model. Numer. Anal., 48 (2014) (3), pp. 623–663. doi:10.1051/m2an/2013103.

[DPSD2008] L. Dalcín, R. Paz, M. Storti, J. D'Elía. *MPI for Python: Performance improvements and MPI-2 extensions*. Journal of Parallel and Distributed Computing, 68 (2008) (5), pp. 655–662. doi:10.1016/j.jpdc.2007.09.005.

[DVTP2013] C. Daversin, S. Veys, C. Trophime, C. Prud'homme. *A reduced basis framework: application to large scale non-linear multi-physics problems*. In *CEMRACS 2012*, vol. 43 of *ESAIM Proc.*, pp. 225–254. EDP Sci., Les Ulis, 2013. doi:10.1051/proc/201343015.

[DKN2014] A. Dedner, R. Klöfkorn, M. Nolte. *The DUNE-ALUGrid Module*. ArXiv e-prints [cs.MS], (2014). `1407.6954`.

[DKNO2010] A. Dedner, R. Klöfkorn, M. Nolte, M. Ohlberger. *A generic interface for parallel and adaptive discretization schemes: abstraction principles and the DUNE-FEM module*. Computing, 90 (2010) (3-4), pp. 165–196. doi:10.1007/s00607-010-0110-3.

[Doe1996] W. Dörfler. *A convergent adaptive algorithm for Poisson's equation*. SIAM J. Numer. Anal., 33 (1996) (3), pp. 1106–1124. doi:10.1137/0733054.

[DEH2008] P. Dostert, Y. Efendiev, T. Y. Hou. *Multiscale finite element methods for stochastic porous media flow equations and applications to uncertainty quantification*. Comput. Methods Appl. Mech. Engrg., 197 (2008) (43-44), pp. 3445–3455. doi:10.1016/j.cma.2008.02.030.

[Dro2009] M. Drohmann. *Reduzierte Basis Methoden für ungesättigte Grundwasserströmungen*. Diplomarbeit, Institut für Numerische und Angewandte Mathematik, Westfälische Wilhelms-Universität Münster, 2009.

[Dro2012] M. Drohmann. *Reduced basis model reduction for nonlinear evolution equations*. Ph.D. thesis, Institute for Computational and Applied Mathematics, Münster, Einsteinstr. 64, 48149 Münster, 2012.

[DHKO2012] M. Drohmann, B. Haasdonk, S. Kaulmann, M. Ohlberger. *A software framework for reduced basis methods using DUNE-RB and RBmatlab*. In A. Dedner, B. Flemisch, R. Klöfkorn, eds., *Advances in DUNE. Proceedings of the DUNE User Meeting, Held 6.-8.10.2010, in Stuttgart, Germany*. Springer, 2012 pp. 77–88. doi:10.1007/978-3-642-28589-9_6.

[DHO2009] M. Drohmann, B. Haasdonk, M. Ohlberger. *Reduced Basis Method for Finite Volume Approximation of Evolution Equations on Parametrized Geometries*. In *Proceedings of ALGORITMY 2009*, 2009 pp. 111–120.

[DHO2012] M. Drohmann, B. Haasdonk, 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.

[EE2003] W. E, B. Engquist. *The heterogeneous multiscale methods*. Commun. Math. Sci., 1 (2003) (1), pp. 87–132.

[EE2003a] W. E, B. Engquist. *Multiscale modeling and computation*. Notices Amer. Math. Soc., 50 (2003) (9), pp. 1062–1070.

[EE2005] W. E, B. Engquist. *The heterogeneous multi-scale method for homogenization problems*. In *Multiscale methods in science and engineering*, vol. 44 of *Lect. Notes Comput. Sci. Eng.*, pp. 89–110. Springer, Berlin, 2005. doi: 10.1007/3-540-26444-2_4.

[EGH2013] Y. Efendiev, J. Galvis, T. Y. Hou. *Generalized multiscale finite element methods (GMsFEM)*. J. Comput. Phys., 251 (2013), pp. 116–135.

[EGLP2014] Y. Efendiev, J. Galvis, G. Li, M. Presho. *Generalized multiscale finite element methods. Nonlinear elliptic equations*. Commun. Comput. Phys., 15 (2014) (3), pp. 733–755.

[EH2007] Y. Efendiev, T. Hou. *Multiscale finite element methods for porous media flows and their applications*. Appl. Numer. Math., 57 (2007) (5-7), pp. 577–596. doi: 10.1016/j.apnum.2006.07.009.

[EHG2004] Y. Efendiev, T. Hou, V. Ginting. *Multiscale finite element methods for nonlinear problems and their applications*. Commun. Math. Sci., 2 (2004) (4), pp. 553–589.

[EH2009] Y. Efendiev, T. Y. Hou. *Multiscale finite element methods*, vol. 4 of *Theory and applications. Surveys and Tutorials in the Applied Mathematical Sciences*. Springer, New York, 2009. ISBN 978-0-387-09495-3.

[EP2003] Y. Efendiev, A. Pankov. *Numerical homogenization of monotone elliptic operators*. Multiscale Model. Simul., 2 (2003) (1), pp. 62–79. doi:10.1137/S1540345903421611.

[EP2004] Y. Efendiev, A. Pankov. *Numerical homogenization and correctors for nonlinear elliptic equations*. SIAM J. Appl. Math., 65 (2004) (1), pp. 43–68. doi: 10.1137/S0036139903424886.

[EP2005] Y. Efendiev, A. Pankov. *Homogenization of nonlinear random parabolic operators*. Adv. Differential Equations, 10 (2005) (11), pp. 1235–1260.

[EHW2000] Y. R. Efendiev, T. Y. Hou, X.-H. Wu. *Convergence of a nonconforming multiscale finite element method*. SIAM J. Numer. Anal., 37 (2000) (3), pp. 888–910. doi:10.1137/S0036142997330329.

[EP2013a] J. L. Eftang, A. T. Patera. *A port-reduced static condensation reduced basis element method for large component-synthesized structures: approximation and A Posteriori error estimation*. Advanced Modeling and Simulation in Engineering Sciences, 1 (2013) (3). doi:10.1186/2213-7467-1-3.

[EP2013] J. L. Eftang, A. T. Patera. *Port reduction in parametrized component static condensation: approximation and a posteriori error estimation*. Internat. J. Numer. Methods Engrg., 96 (2013) (5), pp. 269–302.

[EGM2013] D. Elfverson, E. H. Georgoulis, A. Målqvist. *An adaptive discontinuous Galerkin multiscale method for elliptic problems*. Multiscale Model. Simul., 11 (2013) (3), pp. 747–765. doi:10.1137/120863162.

[EGMP2013] D. Elfverson, E. H. Georgoulis, A. Målqvist, D. Peterseim. *Convergence of a discontinuous Galerkin multiscale method*. SIAM J. Numer. Anal., 51 (2013) (6), pp. 3351–3372. doi:10.1137/120900113.

[ER2007] Y. Epshteyn, B. Rivière. *Estimation of penalty parameters for symmetric interior penalty Galerkin methods*. J. Comput. Appl. Math., 206 (2007) (2), pp. 843–872.

[ES2008] A. Ern, A. F. Stephansen. *A posteriori energy-norm error estimates for advection-diffusion equations approximated by weighted interior penalty methods*. J. Comput. Math., 26 (2008) (4), pp. 488–510.

[ESV2007] A. Ern, A. F. Stephansen, M. Vohralík. *Improved energy norm a posteriori error estimation based on flux reconstruction for discontinuous Galerkin methods*. Preprint R07050, Laboratoire Jacques-Louis Lions & HAL Preprint, 193540 (2007).

[ESV2010] A. Ern, A. F. Stephansen, M. Vohralík. *Guaranteed and robust discontinuous Galerkin a posteriori error estimates for convection–diffusion–reaction problems*. J. Comput. Appl. Math., 234 (2010) (1), pp. 114–130.

[ESZ2009] A. Ern, A. F. Stephansen, P. Zunino. *A discontinuous Galerkin method with weighted averages for advection–diffusion equations with locally small and anisotropic diffusivity*. IMA J. Numer. Anal., 29 (2009) (2), pp. 235–256.

[FR1983] J. P. Fink, W. C. Rheinboldt. *On the error behavior of the reduced basis technique for nonlinear finite element approximations*. Z. Angew. Math. Mech., 63 (1983) (1), pp. 21–28. doi:10.1002/zamm.19830630105.

[FM1971] R. L. Fox, H. Miura. *An approximate analysis technique for design calculations*. AIAA J., 9 (1971) (1), pp. 177–179.

[Gio1975] E. d. Giorgi. *Sulla convergenza di alcune successioni di integrali del tipo dell'area*. Rend. Mat. Appl. (7), 8 (1975), pp. 277–294.

[Gio1983] E. d. Giorgi. *G-operators and* Γ*-convergence*. In *Proceedings of the International Congress of Mathematicians*, vol. 1, 1983 p. 2.

[Gir2012] S. Girke. *Vereinheitlichter Rahmen zur Implementierung hybridisierter Diskretisierungsverfahren*. Diplomarbeit, Westfälische Wilhelms-Universität Münster, Institut für Numerische und Angewandte Mathematik, Einsteinstr. 62, 48149 Münster, 2012.

[GO1977] D. Gottlieb, S. A. Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*. CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1977. ISBN 9781611970425.

[GJo2010] G. Guennebaud, B. Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org, 2010.

[GAB2008] S. Gugercin, A. C. Antoulas, C. Beattie. $\mathcal{H}_2$ *model reduction for large-scale linear dynamical systems*. SIAM J. Matrix Anal. Appl., 30 (2008) (2), pp. 609–638. doi:10.1137/060666123.

[HDO2011] B. Haasdonk, M. Dihlmann, 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) (4), pp. 423–442. doi:10.1080/13873954.2011.547674.

[HOR2008] B. Haasdonk, M. Ohlberger, G. Rozza. *A reduced basis method for evolution schemes with parameter-dependent explicit operators*. Electron. Trans. Numer. Anal., 32 (2008), pp. 145–161.

[HBHJ2008] H. Hajibeygi, G. Bonfigli, M. A. Hesse, P. Jenny. *Iterative multiscale finite-volume method*. J. Comput. Phys., 227 (2008) (19), pp. 8604–8621. doi: 10.1016/j.jcp.2008.06.013.

[HJ2009] H. Hajibeygi, P. Jenny. *Multiscale finite-volume method for parabolic problems arising from compressible multiphase flow in porous media*. J. Comput. Phys., 228 (2009) (14), pp. 5129–5147. doi:10.1016/j.jcp.2009.04.017.

[HJ2011] H. Hajibeygi, P. Jenny. *Adaptive iterative multiscale finite volume method*. J. Comput. Phys., 230 (2011) (3), pp. 628–643. doi:10.1016/j.jcp.2010.10.009.

[Hec2012] F. Hecht. *New development in freefem++*. J. Numer. Math., 20 (2012) (3-4), pp. 251–265.

[Hen2012] P. Henning. *Convergence of MSFEM approximations for elliptic, non-periodic homogenization problems*. Netw. Heterog. Media, 7 (2012) (3), pp. 503–524. doi: 10.3934/nhm.2012.7.503.

[HM2014] P. Henning, A. Målqvist. *Localized orthogonal decomposition techniques for boundary value problems*. SIAM J. Sci. Comput., 36 (2014) (4), pp. A1609–A1634.

[HMP2014] P. Henning, A. Målqvist, D. Peterseim. *A localized orthogonal decomposition method for semi-linear elliptic problems*. ESAIM Math. Model. Numer. Anal., 48 (2014) (5), pp. 1331–1349.

[HO2009] P. Henning, M. Ohlberger. *The heterogeneous multiscale finite element method for elliptic homogenization problems in perforated domains*. Numer. Math., 113 (2009) (4), pp. 601–629. doi:10.1007/s00211-009-0244-4.

[HO2010] P. Henning, M. Ohlberger. *The heterogeneous multiscale finite element method for advection-diffusion problems with rapidly oscillating coefficients and large expected drift*. Netw. Heterog. Media, 5 (2010) (4), pp. 711–744. doi: 10.3934/nhm.2010.5.711.

[HO2012] P. Henning, M. Ohlberger. *A Newton-scheme framework for multiscale methods for nonlinear elliptic homogenization problems*. In *Proceedings of the Algoritmy 2012, 19th Conference on Scientific Computing, Vysoke Tatry, Podbanske, September 9-14, 2012*, 2012 pp. 65–74. doi:10.13140/2.1.4553.4727.

[HOS2014] P. Henning, M. Ohlberger, B. Schweizer. *An adaptive multiscale finite element method*. Multiscale Model. Simul., 12 (2014) (3), pp. 1078–1107.

[HP2013] P. Henning, D. Peterseim. *Oversampling for the multiscale finite element method*. Multiscale Model. Simul., 11 (2013) (4), pp. 1149–1175. doi:10.1137/120900332.

[HO2014] C. Himpe, M. Ohlberger. *Cross-gramian-based combined state and parameter reduction for large-scale control systems*. Math. Probl. Eng., (2014), pp. Art. ID 843869, 13. doi:10.1155/2014/843869.

[Hor1997] U. Hornung, ed. *Homogenization and porous media*, vol. 6 of *Interdisciplinary Applied Mathematics*. Springer New York, 1997. ISBN 978-1-4612-7339-4. doi: 10.1007/978-1-4612-1920-0.

[HW1997] T. Y. Hou, X.-H. Wu. *A multiscale finite element method for elliptic problems in composite materials and porous media*. J. Comput. Phys., 134 (1997) (1), pp. 169–189. doi:10.1006/jcph.1997.5682.

[HWC1999] T. Y. Hou, X.-H. Wu, Z. Cai. *Convergence of a multiscale finite element method for elliptic problems with rapidly oscillating coefficients*. Math. Comp., 68 (1999) (227), pp. 913–943. doi:10.1090/S0025-5718-99-01077-7.

[Hug1995] T. J. R. Hughes. *Multiscale phenomena: Green's functions, the Dirichlet-to-Neumann formulation, subgrid scale models, bubbles and the origins of stabilized methods*. Comput. Methods Appl. Mech. Engrg., 127 (1995) (1-4), pp. 387–401. doi:10.1016/0045-7825(95)00844-9.

[HFMQ1998] T. J. R. Hughes, G. R. Feijóo, L. Mazzei, J.-B. Quincy. *The variational multiscale method—a paradigm for computational mechanics*. Comput. Methods Appl. Mech. Engrg., 166 (1998) (1-2), pp. 3–24. doi:10.1016/S0045-7825(98)00079-6.

[HKP2013] D. B. P. Huynh, D. J. Knezevic, A. T. Patera. *A static condensation reduced basis element method: approximation and a posteriori error estimation*. ESAIM Math. Model. Numer. Anal., 47 (2013) (1), pp. 213–251. doi:10.1051/m2an/2012022.

[HRSP2007] D. B. P. Huynh, G. Rozza, S. Sen, A. T. Patera. *A successive constraint linear optimization method for lower bounds of parametric coercivity and inf-sup stability constants*. C. R. Math. Acad. Sci. Paris, 345 (2007) (8), pp. 473–478. doi:10.1016/j.crma.2007.09.019.

[IQR2012] L. Iapichino, A. Quarteroni, G. Rozza. *A reduced basis hybrid method for the coupling of parametrized domains represented by fluidic networks*. Comput. Methods Appl. Mech. Engrg., 221/222 (2012), pp. 63–82. doi:10.1016/j.cma.2012.02.005.

[IQRV2014] L. Iapichino, A. Quarteroni, G. Rozza, S. Volkwein. *Reduced basis method for the Stokes equations in decomposable parametrized domains using greedy optimization*. In *ECMI 2014 proceedings*. ECMI book subseries of Mathematics in Industry, Springer, Heildeberg, 2014 pp. 1–7.

[JLT2003] P. Jenny, S. H. Lee, H. A. Tchelepi. *Multi-scale finite-volume method for elliptic problems in subsurface flow simulation*. J. Comput. Phys., 187 (2003) (1), pp. 47–67.

[JLT2004] P. Jenny, S. H. Lee, H. A. Tchelepi. *Adaptive multiscale finite-volume method for multiphase flow and transport in porous media*. Multiscale Model. Simul., 3 (2004) (1), pp. 50–64. doi:10.1137/030600795.

[JLT2006] P. Jenny, S. H. Lee, H. A. Tchelepi. *Adaptive fully implicit multi-scale finite-volume method for multi-phase flow and transport in heterogeneous porous media*. J. Comput. Phys., 217 (2006) (2), pp. 627–641. doi:10.1016/j.jcp.2006.01.028.

[JOPo2001] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python (*`http://www.scipy.org/`*)*, 2001–2015.

[KP2003] O. A. Karakashian, F. Pascal. *A posteriori error estimates for a discontinuous Galerkin approximation of second-order elliptic problems*. SIAM J. Numer. Anal., 41 (2003) (6), pp. 2374–2399 (electronic). doi:10.1137/S0036142902405217.

[Kau2011] S. Kaulmann. *A Localized Reduced Basis Approach for Heterogeneous Multiscale Problems*. Diplomarbeit, Institut für Numerische und Angewandte Mathematik, Westfälische Wilhelms-Universität Münster, 2011.

[KFH+2014] S. Kaulmann, B. Flemisch, B. Haasdonk, K.-A. Lie, M. Ohlberger. *The Localized Reduced Basis Multiscale method for two-phase flows in porous media*. ArXiv e-prints [math.NA], (2014). `1405.2810`.

[KOH2011] S. Kaulmann, M. Ohlberger, B. Haasdonk. *A new local reduced basis discontinuous Galerkin approach for heterogeneous multiscale problems*. C. R. Math. Acad. Sci. Paris, 349 (2011) (23-24), pp. 1233–1238. doi:10.1016/j.crma.2011.10.024.

[KPSC2006] B. S. Kirk, J. W. Peterson, R. H. Stogner, G. F. Carey. `libMesh`*: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations*. Engineering with Computers, 22 (2006) (3–4), pp. 237–254.

[KP2011] D. J. Knezevic, J. W. Peterson. *A high-performance parallel implementation of the certified reduced basis method*. Computer Methods in Applied Mechanics and Engineering, 200 (2011) (13–16), pp. 1455–1466. doi:10.1016/j.cma.2010.12.026.

[Kol1936] A. Kolmogoroff. *Über die beste Annäherung von Funktionen einer gegebenen Funktionenklasse*. Ann. of Math. (2), 37 (1936) (1), pp. 107–110. doi:10.2307/1968691.

[LM2005] M. G. Larson, A. Målqvist. *Adaptive variational multiscale methods based on a posteriori error estimation: duality techniques for elliptic problems*. In *Multiscale methods in science and engineering*, vol. 44 of *Lect. Notes Comput. Sci. Eng.*, pp. 181–193. Springer, Berlin, 2005. doi:10.1007/3-540-26444-2_9.

[LM2007] M. G. Larson, A. Målqvist. *Adaptive variational multiscale methods based on a posteriori error estimation: energy norm estimates for elliptic problems*. Comput. Methods Appl. Mech. Engrg., 196 (2007) (21-24), pp. 2313–2324. doi:10.1016/j.cma.2006.08.019.

[LM2009a] M. G. Larson, A. Målqvist. *An adaptive variational multiscale method for convection-diffusion problems*. Comm. Numer. Methods Engrg., 25 (2009) (1), pp. 65–79. doi:10.1002/cnm.1106.

[LM2009] M. G. Larson, A. Målqvist. *A mixed adaptive variational multiscale method with applications in oil reservoir simulation*. Math. Models Methods Appl. Sci., 19 (2009) (7), pp. 1017–1042. doi:10.1142/S021820250900370X.

[LV2014] O. Lass, S. Volkwein. *Adaptive POD basis computation for parametrized nonlinear systems using optimal snapshot location*. Comput. Optim. Appl., 58 (2014) (3), pp. 645–677. doi:10.1007/s10589-014-9646-z.

[LMW2012] A. Logg, K.-A. Mardal, G. Wells, eds. *Automated Solution of Differential Equations by the Finite Element Method*, vol. 84 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-23098-1. doi:10.1007/978-3-642-23099-8.

[LMR2007] A. E. Løvgren, Y. Maday, E. M. Rønquist. *The reduced basis element method for fluid flows*. In *Analysis and simulation of fluid dynamics*, pp. 129–154. Adv. Math. Fluid Mech., Birkhäuser, Basel, 2007. doi:10.1007/978-3-7643-7742-7_8.

[MMPR2001] L. Machiels, Y. Maday, A. T. Patera, D. V. Rovas. *A blackbox reduced-basis output bound method for shape optimization*. In T. Chan, T. Kako, H. Kawarada, P. O., eds., *Proceedings of the 12th International Conference on Domain Decompostion Methods*, 2001 pp. 429 – 436.

[MR2002] Y. Maday, E. M. Rønquist. *A reduced-basis element method*. C. R. Math. Acad. Sci. Paris, 335 (2002) (2), pp. 195–200. doi:10.1016/S1631-073X(02)02427-5.

[MR2004] Y. Maday, E. M. Rønquist. *The reduced basis element method: application to a thermal fin problem*. SIAM J. Sci. Comput., 26 (2004) (1), pp. 240–258. doi:10.1137/S1064827502419932.

[MS2013] Y. Maday, B. Stamm. *Locally adaptive greedy approximations for anisotropic parameter reduced basis spaces*. SIAM J. Sci. Comput., 35 (2013) (6), pp. A2417–A2441. doi:10.1137/120873868.

[Mal2011] A. Målqvist. *Multiscale methods for elliptic problems*. Multiscale Model. Simul., 9 (2011) (3), pp. 1064–1086. doi:10.1137/090775592.

[MP2014] A. Målqvist, D. Peterseim. *Localization of elliptic multiscale problems*. Math. Comp., 83 (2014) (290), pp. 2583–2603. doi:10.1090/S0025-5718-2014-02868-8.

[MS2002] A.-M. Matache, C. Schwab. *Two-scale FEM for homogenization problems*. M2AN Math. Model. Numer. Anal., 36 (2002) (4), pp. 537–572. doi:10.1051/m2an:2002025.

[MRS2015] R. Milk, S. Rave, F. Schindler. *pyMOR - Generic Algorithms and Interfaces for Model Order Reduction*. arXiv e-prints [cs.MS], (2015). `1506.07094v1`.

[MNS2002] P. Morin, R. H. Nochetto, K. G. Siebert. *Convergence of adaptive finite element methods*. SIAM Rev., 44 (2002) (4), pp. 631–658 (2003). doi:10.1137/S0036144502409093. Revised reprint of "Data oscillation and convergence of adaptive FEM" [SIAM J. Numer. Anal. **3**8 (2000), no. 2, 466–488 (electronic); MR1770058 (2001g:65157)].

[MT1997] F. Murat, L. Tartar. *H-Convergence*. In A. Cherkaev, R. Kohn, eds., *Topics in the Mathematical Modelling of Composite Materials*, vol. 31 of *Progress in Nonlinear Differential Equations and Their Applications*, pp. 21–43. Birkhäuser Boston. ISBN 978-1-4612-7390-5, 1997. doi:10.1007/978-1-4612-2032-9_3.

[Ngu1989] G. Nguetseng. *A general convergence result for a functional related to the theory of homogenization*. SIAM J. Math. Anal., 20 (1989) (3), pp. 608–623. doi:10.1137/0520043.

[Ngu2010] H. T. Nguyen. *p-adaptive and automatic hp-adaptive finite element methods for elliptic partial differential equations*. ProQuest LLC, Ann Arbor, MI, 2010. ISBN 978-1124-10087-6. Thesis (Ph.D.)–University of California, San Diego.

[NB2008] J. M. Nordbotten, P. E. Bjørstad. *On the relationship between the multiscale finite-volume method and domain decomposition preconditioners*. Comput. Geosci., 12 (2008) (3), pp. 367–376. doi:10.1007/s10596-007-9066-6.

[Ohl2005] M. Ohlberger. *A posteriori error estimates for the heterogeneous multiscale finite element method for elliptic homogenization problems*. Multiscale Model. Simul., 4 (2005) (1), pp. 88–114. doi:10.1137/040605229.

[Ohl2012] M. Ohlberger. *Error control based model reduction for multiscale problems*. In *Proceedings of Algoritmy 2012, Conference on Scientific Computing, Vysoke Tatry, Podbanske, September 9-14, 2012*. Slovak University of Technology in Bratislava, Publishing House of STU, 2012 pp. 1–10.

[ORSZ2014] M. Ohlberger, S. Rave, S. Schmidt, S. Zhang. *A Model Reduction Framework for Efficient Simulation of Li-Ion Batteries*. In J. Fuhrmann, M. Ohlberger, C. Rohde, eds., *Finite Volumes for Complex Applications VII-Elliptic, Parabolic and Hyperbolic Problems*, vol. 78 of *Springer Proceedings in Mathematics & Statistics*, pp. 695–702. Springer International Publishing. ISBN 978-3-319-05590-9, 2014. doi:10.1007/978-3-319-05591-6_69.

[OS2014] M. Ohlberger, F. Schindler. *A-Posteriori Error Estimates for the Localized Reduced Basis Multi-Scale Method*. In J. Fuhrmann, M. Ohlberger, C. Rohde, eds., *Finite Volumes for Complex Applications VII-Methods and Theoretical Aspects*, vol. 77 of *Springer Proceedings in Mathematics & Statistics*, pp. 421–429. Springer International Publishing. ISBN 978-3-319-05683-8, 2014. doi:10.1007/978-3-319-05684-5_41.

[OS2015] M. Ohlberger, F. Schindler. *Error control for the localized reduced basis multi-scale method with adaptive on-line enrichment*. ArXiv e-prints [math.NA], (2015). `1501.05202`.

[Oli2007] T. E. Oliphant. *Python for Scientific Computing*. Computing in Science & Engineering, 9 (2007) (3), pp. 10–20. doi:10.1109/MCSE.2007.58.

[PR2006] A. T. Patera, G. Rozza. *Reduced Basis Approximation and A Posteriori Error Estimation for Parametrized Partial Differential Equations, Version 1.0*. Tech. rep., Copyright MIT 2006–2007, to appear in (tentative rubric) MIT Pappalardo Graduate Monographs in Mechanical Engineering, 2006.

[PVWW2013] G. V. Pencheva, M. Vohralík, M. F. Wheeler, T. Wildey. *Robust a posteriori error control and adaptivity for multiscale, multinumerics, and mortar coupling*. SIAM J. Numer. Anal., 51 (2013) (1), pp. 526–554. doi:10.1137/110839047.

[PG2007] F. Pérez, B. E. Granger. *IPython: a System for Interactive Scientific Computing*. Computing in Science and Engineering, 9 (2007) (3), pp. 21–29. doi:10.1109/MCSE.2007.53.

[Pin1985] A. Pinkus. *n-Widths in Approximation Theory*. Ergebnisse der Mathematik und ihrer Grenzgebiete. 3. Folge / A Series of Modern Surveys in Mathematics, Springer-Verlag Berlin Heidelberg, 1985. ISBN 354013638X. doi:10.1007/978-3-642-69894-1.

[Por1985] T. A. Porsching. *Estimation of the error in the reduced basis method solution of nonlinear equations*. Math. Comp., 45 (1985) (172), pp. 487–496. doi:10.2307/2008138.

[PCD+2012] C. Prud'homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, G. Pena. *FEEL++: a computational framework for Galerkin methods and advanced numerical methods*. In *CEMRACS'11: Multiscale coupling of complex models in scientific computing*, vol. 38 of *ESAIM Proc.*, pp. 429–455. EDP Sci., Les Ulis, 2012. doi:10.1051/proc/201238024.

[QRM2011] A. Quarteroni, G. Rozza, A. Manzoni. *Certified reduced basis approximation for parametrized partial differential equations and applications*. J. Math. Ind., 1 (2011), pp. Art. 3, 44. doi:10.1186/2190-5983-1-3.

[QV1999] A. Quarteroni, A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Numerical mathematics and scientific computation, Clarendon Press, 1999. ISBN 9780198501787.

[RHP2008] G. Rozza, D. B. P. Huynh, A. T. Patera. *Reduced basis approximation and a posteriori error estimation for affinely parametrized elliptic coercive partial differential equations: application to transport and continuum mechanics*. Arch. Comput. Methods Eng., 15 (2008) (3), pp. 229–275. doi:10.1007/s11831-008-9019-9.

[SM2002] C. Schwab, A.-M. Matache. *Generalized FEM for homogenization problems*. In *Multiscale and multiresolution methods*, vol. 20 of *Lect. Notes Comput. Sci. Eng.*, pp. 197–237. Springer, Berlin, 2002. doi:10.1007/978-3-642-56205-1_4.

[SV2011] B. Schweizer, M. Veneroni. *The needle problem approach to non-periodic homogenization*. Netw. Heterog. Media, 6 (2011) (4), pp. 755–781. doi:10.3934/nhm.2011.6.755.

[Sha2003] Y. Shapira. *Matrix-Based Multigrid: Theory and Applications*. Numerical methods and algorithms, Kluwer Academic Publishers, 2003. ISBN 9781402074851.

[SZC1959] J. W. Sheldon, B. Zondek, W. T. Cardwell. *One-dimensional, incompressible, non-capillary, two-phase fluid flow in a porous medium*. Trans. SPE AIME, 216 (1959), pp. 290–296.

[Sir1987] L. Sirovich. *Turbulence and the dynamics of coherent structures. I. Coherent structures*. Quart. Appl. Math., 45 (1987) (3), pp. 561–571.

[Sme2015] K. Smetana. *A new certification framework for the port reduced static condensation reduced basis element method*. Comput. Methods Appl. Mech. Engrg., 283 (2015), pp. 352–383. doi:10.1016/j.cma.2014.09.020.

[Spa1968] S. Spagnolo. *Sulla convergenza di soluzioni di equazioni paraboliche ed ellittiche*. Ann. Scuola Norm. Sup. Pisa Cl. Sci. (3), 22 (1968) (4), pp. 571–597.

[SG1961] H. L. Stone, A. O. J. Garder. *Analysis of gas-cap or dissolved-gas reservoirs*. Trans. SPE AIME, 222 (1961), pp. 92–104.

[Tar1976] L. Tartar. *Quelques remarques sur l'homogénéisation*. In *Functional Analysis and Numerical Analysis, Proceedings of the Japan-France Seminar*, 1976 pp. 469–482.

[Tem2008] V. N. Temlyakov. *Greedy approximation*. Acta Numer., 17 (2008), pp. 235–409. doi:10.1017/S0962492906380014.

[TW2005] A. Toselli, O. Widlund. *Domain Decomposition Methods – Algorithms and Theory*. No. 34 in Springer Series in Computational Mathematics, Springer-Verlag Berlin Heidelberg, 2005. ISBN 978-3-540-20696-5. doi:10.1007/b137868.

[Urb2009] K. Urban. *Wavelet methods for elliptic partial differential equations*. Numerical mathematics and scientific computation, Oxford University Press Oxford, 2009. ISBN 978-0-19-852605-6.

[Ver1996] R. Verfürth. *A Review of A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*. Wiley Teubner, 1996. ISBN 978-0-19-967942-3.

[Ver2013] R. Verfürth. *A posteriori error estimation techniques for finite element methods*. Numerical Mathematics and Scientific Computation, Oxford University Press, Oxford, 2013. ISBN 978-0-19-967942-3. doi:10.1093/acprof:oso/9780199679423.001.0001.

[VPP2003] K. Veroy, C. Prud'homme, A. T. Patera. *Reduced-basis approximation of the viscous Burgers equation: rigorous a posteriori error bounds*. C. R. Math. Acad. Sci. Paris, 337 (2003) (9), pp. 619–624. doi:10.1016/j.crma.2003.09.023.

[Voh2007] M. Vohralík. *A posteriori error estimates for lowest-order mixed finite element discretizations of convection-diffusion-reaction equations*. SIAM J. Numer. Anal., 45 (2007) (4), pp. 1570–1599 (electronic). doi:10.1137/060653184.

[Wes1992] P. Wesseling. *An introduction to multigrid methods*. Pure and applied mathematics, John Wiley & Sons Australia, Limited, 1992. ISBN 9780471930839.

[WP2002] K. Willcox, J. Peraire. *Balanced model reduction via the proper orthogonal decomposition*. AIAA J., 40 (2002) (11), pp. 2323–2330.

[WSH2014] D. Wirtz, D. C. Sorensen, B. Haasdonk. *A posteriori error estimation for DEIM reduced nonlinear dynamical systems*. SIAM J. Sci. Comput., 36 (2014) (2), pp. A311–A338. doi:10.1137/120899042.

[Yan2014] M. Yano. *A minimum-residual mixed reduced basis method: Exact residual certification and simultaneous finite-element reduced-basis refinement*. Tech. rep., MIT, 2014.

[ZF2015] M. J. Zahr, C. Farhat. *Progressive construction of a parametric reduced-order model for PDE-constrained optimization*. Internat. J. Numer. Methods Engrg., 102 (2015) (5), pp. 1111–1135. doi:10.1002/nme.4770.