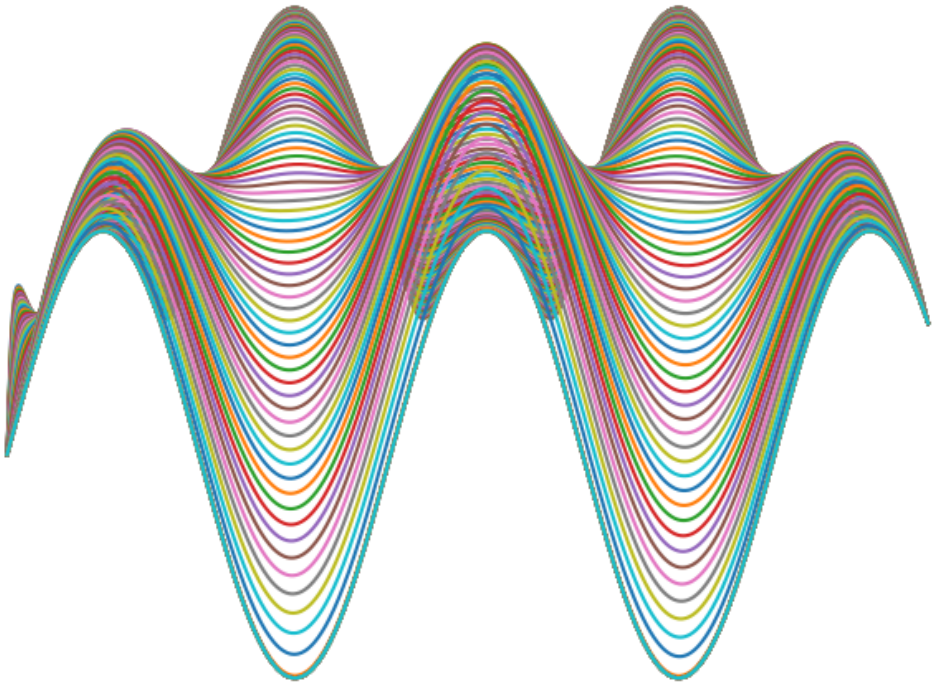
Reduced Basis Method for semilinear elliptic PDEs with strong Nonlinearity

Masterthesis

Hanna Gierling

June 1, 2025



Prof. Dr. Karsten Urban
Institut für Numerische Mathematik
Universität Ulm

Prof. Dr. Stephan Schlüter
Institut für Energie- und Antriebstechnik
Technische Hochschule Ulm

Contents

1 Introduction

Reduced Basis (RB) methods are a popular model order reduction technique. They are designed to efficiently solve parametrized partial differential equations (PPDEs). These are partial differential equations (PDEs) that depend not only on space (and time), but also on one ore more parameters — such as diffusion constants, material coefficients, or boundary conditions among others. Hence they are a popular choice in either real-time scenarios or multi-query scenarios.

The main idea of RB methods is to approximate the solution manifold of a PPDE using only a small number N of basis functions, denoted as the reduced basis. The solution manifold is the collection of all possible solutions $u(\boldsymbol{\mu})$ of the given PPDE, where the parameter $\boldsymbol{\mu}$ varies within a certain parameter domain $\mathcal{P} \subset \mathbb{R}^p$. This manifold is written as:

$$\mathcal{M} = \{u(\boldsymbol{\mu}) : \boldsymbol{\mu} \in \mathcal{P}\}.$$

To measure how well this solution set $\mathcal{M}$ can be approximated by a space of dimension N, the *Kolmogorov N-width*, denoted by $d_N(\mathcal{M})$, is used. It measures the best possible worst-case error when approximating $\mathcal{M}$ using a N dimensional linear space. The goal of RB methods is therfore to find an approximation space V_N, where the actual approximation error is close to this theoretical best error given by $d_N(\mathcal{M})$.

RB methods have proven particularly powerful for certain parametrized, linear and coercive problems. In such cases, it has been shown (see [1]), that the Kolmogorov N-width decreases exponantially with N:

$$d_N(\mathcal{M}) \leq Ce^{-\beta N}$$

for some positive constants C and β. This means very accurate approximations can be achieved with only a few basis functions.

Moreover, it is proven in [2] that if the solution manifold can be approximated this efficiently ($d_N(\mathcal{M})$ decays exponantially), then the greedy algorithm - a standard way to compute the reduced basis — will also construct an exponentially converging approximation:

$$\|u - P_{V_N} u\| \leq ce^{-\alpha N}.$$

However, many problems of practical interest are not linear but involve semilinear or even fully nonlinear PDEs. In these cases, the theoretical foundation of RB methods is not as advanced. In particular, it is still an open question whether the exponential decay of the Kolmogorov N-width—and hence the rapid convergence of reduced basis approximations— also holds for semilinear elliptic PDEs.

Therfore this thesis investigates effectiveness of reduced Basis (RB) methods for semilinear, elliptic PPDEs with strong nonlinearities. Specifically, we consider equations of the form

$$-\nabla \cdot (\boldsymbol{\mu} \nabla u) - q(u) = f,$$

where the paramater $\boldsymbol{\mu}$ represents the diffusion coefficient, f is some forcing term and $q(u)$ denotes a nonlinear function of the solution u.

As primary model problem, we consider the elliptic Fisher equation, a semilinear PDE with strong sensitivty to parameter variations. This makes it a challenging and interesting test case for both nonlinear solvers and model reduction techniques. We also consider a semilinear

Poisson equation as a well-posed reference problem with a unique solution, and a modified Fisher equation with a known exact solution to validate numerical solvers.

We begin by commenting on the software used for the numerical experiments throughout this thesis in section 2.

Before evaluating the performance of RB methods, a stable Newton solver must be chosen. This thesis investigates several damping strategies to stabilize Newton's method for the elliptic semilinear problem. Specifically, we test:

- an adaptive damping method based on a dynamical systems approach from [3],
- a simple error-oriented damping strategy from [4], and
- an affine-covariant, error-oriented damping method described in [5].

We begin with an introduction to Newton's method for semilinear PDEs in section 3.1, followed by a full overview of RB methods for nonlinear problems in section 4. The Newton solvers are tested in section 5, where we observe two solution branches in the Fisher-type equations.

Once a robust nonlinear solver is established, we analyze the performance of RB methods in section 6. We first study the singular value decay for all test problems. Then, we use the Greedy algorithm to construct reduced models for both solution branches of the Fisher equation. Additionally, we apply a h-type Greedy algorithm from [6] to build localized reduced spaces. This algorithm splits the parameter domain into smaller subdomains, based on how the solution behaves in different regions. It is therefore an effective approach for problems where the solution changes significantly across the parameter domain.

Implementation details are briefly introduced where appropriate, to provide insight on how all results were computed.

2 Remarks on the Choice of Software

When choosing a software framework for this thesis, several options, that are PyMOR, RBniCS and FEniCS, were considered. Each tool has its own strengths and weaknesses. Below is a short discussion of the software packages considered.

FEniCS (Finite Element Computational Software) is a powerful, flexible and high-performance computing (HPC) capable open-source library designed for solving PDEs using the finite element method (FEM). It is not intended for RB methods directly, but its strength lies in its clear structure and a wide range of capabilities for solving differential equations. Many tasks can be implemented in a very elegant and readable way, which helped a lot during this thesis.

One of the main advantages of FEniCS is its excellent documentation and the large number of well-written tutorials available. Especially notably are the books [7] and [8]. In addition, FEniCS has a large and active community forum where users can ask questions, share knowledge, and get help with specific problems. These resources make it very easy to overcome technical challenges, even for users who are new to the field of numerical PDEs. Overall, FEniCS proved to be a very helpful tool.

RBniCS (Reduced Basis Computational Software) is an extension of FEniCS that adds support for Reduced Basis methods. However, in practice, it turned out to be quite difficult to use. The documentation is almost non-existent, and while well structured tutorials are available, they do not provide a deeper understanding of the software. As a result, it is very hard to apply them to custom problems or adapt them to specific needs. This lack of guidance makes the use of RBniCS very challenging. Especially for users who are still building a solid understanding of the mathematical foundations of RB methods.

This software is also referenced as a companion to the introductory handbook [9]. However, this book is not freely available. For installation instructions and tutorials, see [10].

PyMOR (Model Order Reduction with Python) is a Python-based library that was developed to support model order reduction, especially RB methods. Unlike RBniCS, it does not rely on existing PDE solvers. Instead all pyMOR algorithms allow generic implementations to work with different backends (e.g. external PDE solver packages). PyMOR offers very well-organized and informative tutorials, making it relatively easy to get started.

However, for nonlinear problems, PyMOR's built-in solvers are somewhat limited. For instance, its nonlinear solver only supports finite volume discretizations. Fortunately, PyMOR can be coupled with external solvers like FEniCS, which allows to use FEM. Still, nonlinear terms are linearized using Empirical Interpolation Method (EIM). While EIM is commonly used in RB frameworks, we specifically aim to avoid it in this work in order to eliminate the interpolation error it introduces.[11] Further the available nonlinear solver relies on an ordinary Newton solver, which is not sufficient for our highly nonlinear problem. Introducing new ideas or custom modifications is often difficult for someone without a deeper understanding of PyMOR's internal structure. This makes it especially difficult to use PyMOR when one is not yet fully confident with the underlying RB concepts. For further information as installation and tutorials we refer to [12].

Final Decision In the end, FEniCS was chosen as the main software tool for this thesis. While it does not include RB methods out of the box, its high flexibility, elegant design, and good documentation made it the most practical option. With FEniCS, it was possible to build a well-structured numerical framework, while still leaving room for future extensions and adaptions. Since RB methods are not built into the framework, any RB-related techniques must be implemented manually. While this requires more effort, it also provides the freedom to experiment with new ideas and fully control the implementation.

Remark 2.1
It is important to mention that the version of FEniCS (2019.1.0) used in this thesis is the "old" FEniCS, which is the predecessor of FEniCSx. While FEniCSx is the newer and actively developed version, we chose to work with the older version for following reasons. At the beginning of the project, we considered using RBniCS, which is built entirely on the classic FEniCS interface. Then we tried to work with PyMOR, which already provides a working interface for the classic FEniCS version, but not yet for FEniCSx. As a result, we had already spent significant time getting used to the older FEniCS version. To make use of this existing experience and avoid additional setup effort, we decided to stay with the classic FEniCS framework throughout this work.

In the meantime, a new version of RBniCS—called RBniCSx—has been released. It is designed to work with FEniCSx. However, at the time of writing, RBniCSx is still in an early development stage.

3 Newton's Method

In this chapter, Newton's method is introduced. We begin by outlining the basic principles of the classical Newton method for general problems. It is important to note that the Newton's method is only locally convergent. This means that convergence is generally guaranteed only if the initial guess is sufficiently close to the true solution. For many problems, finding such a suitable initial guess can be challenging. To address these difficulties, we introduce three different damping strategies aimed to enlarge the basin of convergence.
Following this, we extend the method to the context of semilinear elliptic PDEs. Finally, we present a tutorial on how to solve a semilinear elliptic PDE using FEniCS, and an overview of the implementation of the damping strategies introduced in this section.

3.1 The General Newton's Method

We begin by considering Newtons's Method for general nonlinear problems. Let X and Y be Banach spaces with norms $\|\cdot\|_X$ and $\|\cdot\|_Y$. Given an nonlinear operator $F : D \subset X \to Y$, we are interested in solving the equation

$$F(x) = 0. \tag{3.1}$$

Starting from an initial guess $x^{(0)}$ for the unknown solution x^*, we linearize F around $x^{(k)}$, $k = 0, 1, ...$ using first-order Taylor expansion. Instead of solving the nonlinear equation (3.1) directly, we iteratively solve the linearized equation, which leads to the ordinary Newton method:[5]

$$\begin{aligned} F'(x^{(k)})\Delta x^{(k)} &= -F(x^{(k)}) \\ x^{(k+1)} &= x^{(k)} + \Delta x^{(k)} \quad k = 0, 1, ... \end{aligned} \tag{3.2}$$

Assume D is open and convex, $F'(x)^{-1}$ exists, is bounded for all $x \in D$ and satisfies a Lipschitz condition. Then the ordinary Newton method (3.2) converges *locally* quadratically (see chapter 1.2.1 in [5]). This requires an initial guess $x^{(0)}$ that is sufficiently close to x^*, which is often a nontrivial task to find. To address this issue, damping strategies are used to extend the basin of convergence. In such strategies, the Newton correction $\Delta x^{(k)}$ is scaled by some damping factor $\lambda^{(k)} \in (0, 1]$. Some strategies for determining the damping factor $\lambda^{(k)}$ will be introduced in the following section.

If no suitable damping factor $\lambda^{(k)}$ is found, the damping strategy returns $\lambda^{(k)} = 0$. The Newton algorithm (algorithm 3) interprets this as λ-FAIL and as a consequence, terminates without converging to a solution.

3.2 Damping strategies

If the initial guess $x^{(0)}$ is far from the solution x^*, damping can improve the reliability of Newton's method. In this section, we present three different damping strategies. First, we introduce a method based on a dynamical systems perspective, as developed in [3]. Next, we describe a simple error-oriented approach from [4]. Finally, we discuss the error-oriented affine covariant strategy proposed in [5].

From now on we consider the case where $X, Y = \mathbb{R}^n$.

3.2.1 Damping based on a dynamical systems approach (adaptDamp)

The following damping strategy was developed in [3]. For a detailed discussion and proofs, please refer to the original publication. The damped Newton iterate is

$$x^{(k+1)} = x^{(k)} - \lambda^{(k)} F'(x^{(k)})^{-1} F(x^{(k)}) \quad k \geq 0. \tag{3.3}$$

with an adaptive step size $\lambda^{(k)} > 0$.

Rearranging terms in (3.3) we get

$$\frac{x^{(k+1)} - x^{(k)}}{\lambda^{(k)}} = F'(x^{(k)})^{-1} F(x^{(k)}) =: N_F(x^{(k)}), \quad k \geq 0. \tag{3.4}$$

With $\Delta t := \lambda$, the equation from above can be seen as a forward Euler discretization of the continuous dynamical system

$$\dot{x}(t) = N_F(x(t)), \quad t \geq 0, \quad x(0) = x_0. \tag{3.5}$$

The goal is to ensure that the discrete solution $x^{(k+1)}$ stays close to the continuous one $x(\lambda^{(k)})$. This means, that

$$\left\| x(\lambda^{(k)}) - x^{(k+1)} \right\|_X \approx \tau.$$

This can be achieved by choosing the stepsize

$$\lambda^{(k)} = \min(\sqrt{2\tau \left\| N_F(x^{(k)}) \right\|_X^{-1}}, 1)$$

for a given tolerance $\tau > 0$. Therefore the exact trajectory given by the solution of (3.5) and its forward Euler approximation from (3.3) remain τ-close in the $\|\cdot\|_X$-norm for the given damping factor $\lambda^{(k)}$.

In addition to the procedure proposed in [3] we set a minimal stepsize $\lambda_{\min}$ to ensure that the stepsize $\lambda^{(k)}$ does not get too small. If the calculated stepsize $\lambda^{(k)}$ is smaller than $\lambda_{\min}$ we set $\lambda^{(k)} = \lambda_{\min}$. The computation of this damping strategy is, compared to those introduced in the following sections, very inexpensive, as it mainly involves computing the norm of an already computed Newton correction and taking a square root. The resulting damping strategy is summerized in algorithm 1. Note that this algorithm never returns a λ-FAIL .

Algorithm 1: adaptive damping strategy

Data: current Newton correction $\Delta x^{(k)}$, tolerance τ, minimal stepsize $\lambda_{\min}$
Result: damping factor λ

$\lambda \leftarrow \min(\sqrt{2\tau \left\| \Delta x^{(k)} \right\|_X^{-1}}, 1)$
if $\lambda < \lambda_{\min}$ **then**
  return $\lambda_{\min}$
end
return λ

3.2.2 Simple damping strategy (simpleDamp)

The following damping strategy is given in [4]. To approximate the error between the current Newton iterate $x^{(k)}$ and the exact solution x^* of the nonlinear problem (3.1) we consider the following estimate:

$$\left\|x^{(k)} - x^*\right\|_X \approx \left\|F'(x^{(k)})^{-1}(F(x^{(k)}) - F(x^*))\right\|_X = \left\|F'(x^{(k)})^{-1}F(x^{(k)})\right\|_X . \tag{3.6}$$

We want to ensure that the error decreases monotonically in each iteration. Therfore a natural monotonicity test is

$$\left\|F'(x^{(k)})^{-1}F(x^{(k+1)})\right\|_X \leq \gamma \left\|F'(x^{(k)})^{-1}F(x^{(k)})\right\|_X . \tag{3.7}$$

Here the reduction factor is chosen as $\gamma = (1 - \frac{\lambda^{(k)}}{2})$. To enforce this, we choose the damping factor $\lambda^{(k)} \in \{1, \frac{1}{2}, \frac{1}{4}, ..., \lambda_{\min}\}$ such that the above inequality holds. If no suitable $\lambda^{(k)} > \lambda_{\min}$ is found, we terminate the procedure and return λ-FAIL , which will terminate the Newton iterations (see algorithm 3). The damping strategy is summerized in following algorithm (2).

Algorithm 2: simple damping strategy

Data: current iterate x^k, Jacobian $F'(x^{(k)})$, function F, minimal stepsize $\lambda_{\min}$
Result: damping factor λ

```
λ = 1
while λ^(k) > λ_min do
    x_i^(k+1) ← x^(k) + λ^(k) · Δx^(k)
    if ‖F'(x^(k))^-1 F(x^(k+1))‖_X ≤ (1 − λ^(k)/2) ‖F'(x^(k))^-1 F(x^(k))‖_X then
        return λ^(k)
    end
    λ^(k) = λ^(k)/2
end
return 0
```

3.2.3 Affine-covariant damping strategy (nleqerr)

This damping strategy is used in the Error-oriented affine-covariant globalised Newton algorithm introduced by Deuflhard in chapter 3 in [5]. Affine covariance means that the Newton-iterates $x^{(k+1)}$ are invariant under transformation (by A) of the image space, i.e. the iterates are the same for

$$F(x) = 0 \text{ and } G(x) := AF(x) = 0$$

for any nonsingular matrix $A \in \mathbb{R}^{n\times n}$. Furthermore, a globalisation of Newton's method should preserve this property.

In order to develop an affine covariant convergence theory (see Theorem 2.2 in [5]), the classical Lipschitz condition on F, typically used in standard convergence theorems, is replaced by an affine covariant telescoped Lipschitz condition

$$\left\|F'(x^{(0)})^{-1}(F'(x) - F'(\bar{x}))\right\|_X \leq \omega \left\|x - \bar{x}\right\|_X .$$

This formulation ensures that the Lipschitz constant ω is affine covariant. A natural monotonicity test ,similar to (3.7), is given by

$$\theta_k(\lambda^{(k)}) = \frac{\left\|-F'(x^{(k)})^{-1}F(x^{(k)} + \lambda^{(k)}\Delta x^{(k)})\right\|_X}{\left\|\Delta x^{(k)}\right\|_X} = \frac{\left\|\overline{\Delta x}^{(k+1)}\right\|_X}{\left\|\Delta x^{(k)}\right\|_X} \leq 1. \tag{3.8}$$

In Lemma 3.16 in [5] it is shown, that

$$\theta_k(\lambda) \leq 1 - \lambda + \frac{1}{2}\lambda^2\omega \left\|\Delta x^{(k)}\right\|_X. \tag{3.9}$$

From this, the optimal damping factor follows

$$\lambda_{opt}^k = \min(1, \frac{1}{\omega \left\|\Delta x^{(k)}\right\|}). \tag{3.10}$$

In practice, the exact Lipschitz constant ω is unknown. Therefore, it is replaced by a computable approximation, which still preserves the affine covariance of the method. For further details on how to compute an estimate for ω and implement the algorithm, refer to chapter 3.3.3 in [5]. Finally we obtain an actually computable estimated damping factor $[\lambda_{opt}]$. But for this the following inequality holds:

$$[\lambda_{opt}] \geq \lambda_{opt}$$

Clearly, this factor $[\lambda_{opt}]$ may be too large. Therfore, a reduction of this damping factor may be needed. That means, this damping strategy has to be split into a prediction strategy and a correction strategy. First an estimated optimal damping factor $[\lambda_{opt}]$ will be predicted. Then this damping factor will be tested using the monotonicity test (3.8). If the condition is not satisfied, a correction strategy follows, where λ is successively reduced based on a posteriori estimates (see Chapter 3.3.3 in [5] for details).

$$\lambda_{i+1}^{(k)} = \min(\frac{1}{2}\lambda_i^{(k)}, \frac{1}{\omega \left\|\Delta x_i^k\right\|}) \tag{3.11}$$

If $\lambda < \lambda_{\min}$ is necessary to fullfill the monotonicity test, the procedure is terminated with no convergence because of a λ-FAIL.

3.3 Application to semilinear elliptic PDEs

In this section, we apply Newton's method to a semilinear elliptic PDE and explain how it is solved using the finite element method (FEM). We begin by introducing the problem formulation. Then, we apply Newton's method to the continuous problem. Finally, we discretize the resultig linearized problem using FEM.

3.3.1 Problem Formulation

Now consider a **semilinear elliptic PDE** on a domain $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$.

Find $u : \Omega \to \mathbb{R}$ such that

$$\begin{aligned} -\nabla \cdot (\boldsymbol{\mu}\nabla u) - q(u) &= 0 \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{3.12}$$

where $\boldsymbol{\mu} > 0$ is a constant Diffusion parameter and q a nonlinear functional.

Choose a suitable Hilbert space V (function space of test functions) as

$$V := H_0^1(\Omega) = \{v \in H^1(\Omega) : \exists (v_k)_{k\in\mathbb{N}} \subset C_0^\infty : v_k \to v \text{ in } H^1(\Omega)\}.$$

Here $H^1(\Omega)$ is the Sobolev space

$$H^1(\Omega) = \{v \in L^2(\Omega) : \exists D^1 v \in L^2(\Omega)\},$$

where $D^1 v$ is the weak derivative of v and $L^2(\Omega)$ is the space of square integrable functions over the domain Ω.

By multiplying the PDE (3.12) by a test function $v \in V$, integration over the domain Ω and applying the technique of integration by parts (see theorem 1.4.8 in [13]), we now can define the **variational formulation** of (3.12):

Find $u \in V$ such that

$$\boldsymbol{\mu} \langle \nabla u, \nabla v \rangle_{L^2(\Omega)} - \langle q(u), v \rangle_{L^2(\Omega)} = 0 \quad \forall v \in V. \tag{3.13}$$

Here $\langle \cdot, \cdot \rangle_{L^2(\Omega)}$ denotes the inner product of the $L^2(\Omega)$-space:

$$\langle f, g \rangle_{L^2(\Omega)} = \int_\Omega \{f \cdot g\} dx.$$

Note that V is a Hilbert space. Therefore, according to the Riesz representation theorem, every element of its dual space

$$V' := \{v : V \to \mathbb{R} : v \text{ linear}\}$$

can be uniquely represented as an inner product with an element from V. We now define a nonlinear operator $F_{\boldsymbol{\mu}} : V \to V'$ by

$$\langle F_{\boldsymbol{\mu}}(u), v \rangle_{V' \times V} := \int_\Omega \{\boldsymbol{\mu} \nabla u \cdot \nabla v - r\, q(u)\, v\} dx \quad \forall v \in V. \tag{3.14}$$

Here $\langle \cdot, \cdot \rangle_{V' \times V}$ denots the dual pairing in $V' \times V$,

$$\langle f, v \rangle_{V' \times V} = f(v) \quad \text{for } v \in V,\, f \in V'.$$

Thus, the PDE (3.12) can be reformulated as a nonlinear operator in V':

$$\text{Find } u \in V : \quad F_{\boldsymbol{\mu}}(u) = 0 \quad \text{in } V'. \tag{3.15}$$

3.3.2 Application of Newton's method

Since V and V' are Banach spaces, we can apply Newton's Method from secton 3.1. To do so, we require a derivative of $F_{\boldsymbol{\mu}}$. This is given by the Fréchet derivative, which can be seen as a generalization of the classical derivative in $\mathbb{R}^n$ to normed spaces. The Fréchet-derivative of $F_{\boldsymbol{\mu}}$ at $u \in V$ in direction $\partial u \in V$ is given by

$$\langle F'_{\boldsymbol{\mu}}(u)\partial u, v \rangle = \int_\Omega \{\boldsymbol{\mu}\, \nabla \partial u \cdot \nabla v - q'(u) \partial v\}\, dx, \quad v, w \in V. \tag{3.16}$$

From this we get a linear PDE in every Newton interation:

$$F'_{\boldsymbol{\mu}}(u^{(k)})\partial u^{(k)} = -F_{\boldsymbol{\mu}}(u^{(k)}) \quad \text{in } V' \tag{3.17}$$

$$u^{(k+1)} = u^{(k)} + \lambda^{(k)} \partial u^{(k)}. \tag{3.18}$$

Of course (3.17) can be expressed as **linear variational formulation**:

Find $u \in V$ such that

$$a[u^{(k)}](\partial u^{(k)}, v) = -l[u^{(k)}](v) \quad \forall v \in V, \tag{3.19}$$

where the bilinearform $a[u](\cdot,\cdot) : V \times V \to \mathbb{R}$ and the linear form $f[u](\cdot) : V \to \mathbb{R}$ are defined as

$$a[u^{(k)}](\partial u^{(k)}, v) = \langle F'_{\boldsymbol{\mu}}(u^{(k)})\partial u^{(k)}, v\rangle_{L^2(\Omega)} \tag{3.20}$$

$$l[u^{(k)}](v) = \langle F_{\boldsymbol{\mu}}(u^{(k)}), v\rangle_{L^2(\Omega)} \tag{3.21}$$

Note that the bilinear form $a[u](\cdot,\cdot)$ corresponds to the derivative of the nonlinear function in Newton's method (3.2) and the linear form $l[u](\cdot)$ to the residual.

3.3.3 Spatial Discretization

Now we want to discretize (3.19). Therfore to define a suitable finite dimensional approximation of $V = H_0^1(\Omega)$, we introduce a triangulation $\mathcal{T}_h$ of the domain Ω. One triangle of the triangulation is denoted by $T \in \mathcal{T}_h$ and h is its diameter. A natural way to define finite element spaces is to consider globally continuous piecewise affine functions which satifsy homogeneous Dirichlet boundary conditions

$$V_h = \{v_h \in C_0(\Omega) : v_h|_T \in \mathbb{P}(T), T \in \mathcal{T}\} \subset V,$$

where $\mathbb{P}(T)$ defines the set of affine functions from $\bar{T}$ to $\mathbb{R}$. Indeed, $V_h \subset V = H_0^1(\Omega)$ holds, since it consists of differentiable functions except for at most a finite number of points (at the vertices of $\mathcal{T}_h$).[14]

Now let $V_h \subset V$ be a finite-dimensional subspace of V with $h > 0$.
The **discretized linear problem** (3.19) takes the form:

Find $u_h \in V_h$ such that

$$a[u_h^{(k)}](\partial u_h^{(k)}, v_h) = -l[u_h^{(k)}](v_h) \quad \forall v_h \in V_h. \tag{3.22}$$

Denoting by $\Phi_h := \{\phi^j\}_{j=1}^{N_h}$ a basis for V_h, every $\partial u_h, u_h \in V_h$ can be expressed by

$$\partial u_h = \sum_{j=1}^{N_h} \partial u_h^{(j)} \phi^j, \quad \text{with } \boldsymbol{\partial \mathbf{u}}_h = (\partial u_h^{(1)}, ..., \partial u_h^{(N_h)})^T \in \mathbb{R}^{N_h}, \tag{3.23}$$

$$u_h = \sum_{j=1}^{N_h} u_h^{(j)} \phi^j, \quad \text{with } \mathbf{u}_h = (u_h^{(1)}, ..., u_h^{(N_h)})^T \in \mathbb{R}^{N_h}. \tag{3.24}$$

Inserting (3.23) and (3.24) into (3.22) and choosing $v_h = \phi^i$, $i = 1, ..., N_h$ yields

$$\sum_{j=1}^{N_h} a[\sum_{i=1}^{N_h} u_h^{(i)} \phi^j](\phi^j, \phi^i)\partial u_h^{(j)} = -l[\sum_{i=1}^{N_h} u_h^{(i)} \phi^j](\phi^i) \quad \forall i = 1, ..., N_h. \tag{3.25}$$

This is equivalent to the **linear system of equations**:

Find $\boldsymbol{\partial}\mathbf{u}_h \in \mathbb{R}^{N_h}$ such that

$$\mathbb{J}_h(\mathbf{u}_h^{(k)})\boldsymbol{\partial}\mathbf{u}_h = -\mathbf{f}_h(\mathbf{u}_h^{(k)}), \tag{3.26}$$

with Jacobian matrix $\mathbb{J}_h(\mathbf{u}_h^{(k)}) \in \mathbb{R}^{N_h \times N_h}$ and residual vector $\mathbf{f}_h(\mathbf{u}_h^{(k)}) \in \mathbb{R}^{N_h}$ defined as

$$(\mathbb{J}_h(\mathbf{u}_h^{(k)}))_{i,j} = a[u^{(k)}](\phi^j, \phi^i), \quad i,j = 1, ..., N_h,$$
$$(\mathbf{f}_h(\mathbf{u}_h^{(k)}))_i = l[u^{(k)}](\phi^i), \quad i = 1, ..., N_h.$$

The above system of linear equations has to be solved in every Newton iteration. This yields an Newton algorithm that is shown in algorithm 3.

Algorithm 3: damped Newton Method for elliptic semilinear PDEs

Data: initial guess $\mathbf{u}_h^{(0)}$, tolerance ϵ, minimal damping factor $\lambda_{\min}$
Result: Solution $\mathbf{u}_h$

for $k = 0$ **to** k_{max} **do**
  if $\left\|\mathbf{f}_h(\mathbf{u}_h^{(k)})\right\| \leq \epsilon$ **then**
    return $\mathbf{u}_h^{(k)}$
  end
  $\boldsymbol{\partial}\mathbf{u}_h^{(k)} \leftarrow$ solve $(\mathbb{J}_h(\mathbf{u}_h^{(k)})\boldsymbol{\partial}\mathbf{u}_h = -\mathbf{f}_h(\mathbf{u}_h^{(k)}))$
  calculate $\lambda^{(k)}$ with some damping strategy (section 3.2)
  if $\lambda^{(k)} < \lambda_{\min}$ **then**
    return "λ-FAIL"
  end
  $\mathbf{u}_h^{(k+1)} \leftarrow \mathbf{u}_h^{(k)} + \lambda^{(k)} \cdot \partial\mathbf{u}_h^{(k)}$
end
return "OUTMAX"

3.4 Tutorial: Solving a semilinear PDE in FEniCS

In this section we give a brief tutorial on how to solve a semilinear PDE using FEniCS. The goal is to provide a basic understanding on how FEniCS works. We begin by presenting an example problem and the corresponding FEniCS code used to sove it. After that we discuss the code in detail. The example is closely based on how the code in this thesis is structered. While it has be simplified for clarity, it still gives a understanding of the main ideas.

We consider a semilinear elliptic PDE

$$-\nabla \cdot (\boldsymbol{\mu} \nabla u) - q(u) = f \quad \text{in } \Omega, \qquad u = u_D \quad \text{on } \partial\Omega$$

where the domain is $\Omega = [0, 1]$, the source term is set to $f = 0$ and the nonlinearity is given by $q(u) = -u^3$. The Dirichlet boundary conditions are defined as:

$$u_D(x) = \begin{cases} -0.1, & x = 0 \\ 0.4, & x = 1 \end{cases}.$$

This problem is well-posed and can be solved using an ordinary Newton solver with a zero initial guess. Because of its simplicity, it serves as a good introductory example. It will be introduced as test problem **slP** in section 5.

The starting point for FEM in FEniCS is a discretized PDE expressed in weak formulation. We choose a finite dimensional subspace $V_h \subset V = H_0^1(\Omega)$ and define:
Find $u \in V_h$ such that

$$\int_\Omega \{\nabla u \cdot \nabla v\} dx - \int_\Omega \{q(u) \cdot v\} dx = 0 \quad \forall v \in V_h.$$

Note that the functions in V_h only satisfy the homogeneous Dirichlet boundary conditions. In FEniCS the Dirichlet boundary conditions u_D are enfored during the assembly of the discrete linear system.

A FEniCS program for solving our semilinear PDE is given as follows:

```
from fenics import *
import matplotlib.pyplot as plt

class MyTestProblem(NonlinearProblem):
    def __init__(self, N_h:int):
        NonlinearProblem.__init__(self)

        # Parameter value
        self.mu = Expression("mu", mu=1, degree=1)

        # Create mesh and define function space
        self.N_h = N_h
        mesh = IntervalMesh(N_h, 0,1)
        self.V_h = FunctionSpace(mesh, "P", 1)

        # Define variational problem
        u = TrialFunction(self.V_h)
        v = TestFunction(self.V_h)
        self.f = Expression("0") # or better: self.f = Constant(0)
        q = lambda u: -u**3
        self.weakform = self.mu * inner(grad(u),grad(v))*dx \
                                - inner(q(u),v)*dx - inner(self.f,v)*dx
```

```
23
24          # Define boundary condition
25          u_D = Expression("x[0]<0.5 ? a : b", a=-0.1, b=0.4,degree=1)
26          boundary = lambda x, on_boundary: on_boundary
27          self.bcs = [DirichletBC(self.V_h, u_D, boundary)]
28
29          # Alternatively: Define boundary condition
30          tol = 1e-14
31          bdry1 = lambda x : abs(x) < tol
32          bdry2 = lambda x : abs(x-1) < tol
33          self.bcs = [DirichletBC(self.V_h, Constant(-0.1), bdry1),
34                      DirichletBC(self.V_h, Constant(+0.4), bdry2)]
35
36          # Unknown solution function (zero initialized)
37          self.u = Function(self.V_h)
38
39          # Define linear and bilinear form for Newton iterations
40          self._l = action(self.weakform, self.u)
41          du = TrialFunction(self.V_h)
42          self._a = derivative(self._l,self.u, du)
43
44      # Set parameter value
45      def set_mu(self, mu):
46          self.mu.mu = mu
47          self.f.mu = mu
48
49      # Assemble residual vector
50      def F(self, b:PETScVector, x:PETScVector):
51          assemble(self._l, tensor=b)
52          [bc.apply(b, x) for bc in self.bcs]
53
54      # Assemble Jacobian matrix
55      def J(self, A:PETScMatrix, x:PETScVector):
56          assemble(self.sa, tensor=A)
57          [bc.apply(A) for bc in self.bcs]
58
59  # Newton solver
60  solver = NewtonSolver()
61  prm = solver.parameters
62  prm["maximum_iterations"] = 100
63  prm["absolute_tolerance"]  = 1/1000**2
64
65  # solve myTestProblem
66  myTestProblem = MyTestProblem(N_h = 1000)
67  # --> initFunc(myTestProblem.u)
68  myTestProblem.set_mu(mu=0.05)
69  nit, conv = solver.solve(myTestProblem, myTestProblem.u.vector())
70
71  # visualization of solution
72  plot(myTestProblem.u, label=f"mu={0.05}")
73  plt.xlabel("x"); plt.ylabel("u")
74  plt.legend()
75  plt.show()
```

Listing 1: Solving a Semilinear elliptic PDE in FEniCS

The class `NonlinearProblem` is an interface provided by FEniCS for defining a nonlinear variational problem. These problems can be solved using the class `NewtonSolver`, which also

is provided by FEniCS. To solve our example problem, we wrap it in a custom class called `MyTestProblem`, which inherits from `NonlinearProblem`. We begin by defining this problem class. The initializer of `MyTestProblem` takes only the parameter `N_h`, which specifies the number of intervals into which the one-dimensional domain is divided.

We now dissect this class in detail:

 # Parameter value Since we consider parametrized PDEs, we store the parameter value in the `Expression` object `self.mu`. Note that this parameter value is not given to the initializer of the problem class. Instead it must be set manually bevor solving the problem. The class `Expression` will be explained in more detail later.

 # Create mesh and define function space We define a uniform finite element mesh over the domain $[0, 1]$. Next we create a finite element function space `self.V_h` as a `FunctionSpace` object. The second argument, "Lagrange", specifies the type of finite element and the third argument its degree. In our case, we always use the standard P_1 linear Lagrange element. Every `FunctionSpace` object has a method `tabulate_dof_coordinates()` which returns a table of the spatial coordinates corresponding to the degrees of freedom (DOFs) of a `Function` object, that is derived from it.

 # Define variational problem We define the trial and the test functions, u and v, as `TrialFunction` and `TestFunction` objects corresponding of the function space `self.V_h`. Next we define the forcing term f. If f is constant over the domain, it is more efficient to represent it using a `Constant` object. We will discuss the `Expression` class more detailed in the following. The last component is the nonlinear function of solution, $q(u)$, which we define as a Python function `q(u)`. With all these components we can now represent the weak formulation of the PDE using a UFL form in `self.weakform`.

 # Define boundary condition The boundary condition $u = u_D$ on $\partial\Omega$ is specified by

```
1 bc = DirichletBC(self.V_h, u_D, boundary)
```

Here, `u_D` is an `Expression` object that specifies the solution values on the boundary. An `Expression` represents a mathematical function and is created as follows:

```
1 u_D = Expression(formula, degree=1)
```

The `formula` is a string containing a mathematical expression in C++ syntax. Depending on the dimension of the problem, this expression may depend on the variables `x[0], x[1], x[2]`, which correspond to the coordinates x, y and z. One can also include parameters in the formula string, as long as they are initialized via keyword arguments when creating the `Expression` object, such as `a` and `b` in line 24.

The second argument is the function `boundary` (defined in line 25), which must return a boolean value indicating whether a given point `x` lies on the Dirichlet boundary. The argument `on_boundary` is provided by FEniCS and equals `True` if `x` lies on the boundary of the mesh. In this simple example, where we want to apply the condition to the entire boundary, we can simply return the value of `on_boundary`. Since a problem may involve multiple Dirichlet boundary conditions, it is common practice to collect them in a list, as is done in `self.bcs`.

 # Alternatively: Define boundary condition To give a better idea of how more advanced Dirichlet boundary conditions can be defined, we show an alternative approach here. It produces exactly the same result as the code in lines 24–26, but is implemeted in a more flexible way.

 # Unknown solution function `self.u` represents the unknown solution of the problem and is a `Function` object from the function space `self.V_h`. The degrees of freedom (DOFs) of a `Function` are stored in its `PETScVector`, which can be accessed using `u.vector()`. To convert this `PETScVector` to a `numpy.ndarray`, one can use

```
    u_dof = u.vector().get_local()
```

One also can set a `numpy.ndarray` as new DOF values of an function by

```
    u.vector().set_local(u_dof)
```

However, it is important to ensure that the array `u_dof` corresponds to the same DOF-coordinate table as the original function space, otherwise the values may be assigned to the wrong locations. The `FunctionSpace` corresponding to a `Function u` can be accessed via

```
    V = u.funcion_space()
```

 # Define linear and bilinear forms We define the linear and bilinear forms required for Newton's method. First, we apply the current solution `self.u` to the weak formulation using the `action` function. The resulting linear form `self.l` corresponds to $l[u](\cdot)$ in equation (3.19). Next, we define the Newton update direction ∂u as a new trial function `du` and compute the bilinear form as the derivative of the linear form w.r.t. u in direction ∂u. This is done using the `derivative` function provided by the UFL (Unified Form Language), which is used to specify the weak forms. UFL supports symbolic differentation of variational forms. This feature allows us to automatically compute the Fréchet derivative needed for solving the nonlinear PDE. The resulting bilinear form `self._a` corresponds to $a[u](\cdot,\cdot)$ in equation (3.19).

 # Set parameter value The `set_mu` method assigns a given value `mu` to the current parameter value `self.mu` of the problem.

 # Assemble residual Vector The Method `F` must be implemented to enable solving the `NonlinearProblem` object with an `NewtonSolver`. It assembles the residual vector of the problem (corresponding to $\mathbf{f}_h(\mathbf{u}_h^{(k)})$ in equation (3.26)) into the `PETScVector b` using the `assemble` function. Additionaly the Dirichlet boundary conditions `self.bcs` of the problem are applied to the residual vector.

 # Assemble Jacobian matrix The Method `J` is also necessary for using the `NewtonSolver` class. It assembles the Jacobian matrix of the problem (corresponding to $\mathbb{J}_h(\mathbf{u}_h^{(k)})$ in equation (3.26)) into the `PETScMatrix A`. Additionaly the Dirichlet boundary conditions are applied to the Jacobian matrix.

 # Newton Solver Now we define a `NewtonSolver` object and set some of its parameter. To inspect all available parameters with their values, one can use one of the following commands:

```
print(dict(solver.parameters))
print(info(solver.parameters,True))
```

 # Solve myTestProblem First, we create an object `myTestProblem` from the previously defined class `MyTestProblem`. Normally, the function `myTestProblem.u` should be initialized with an initial guess using a Python function defined by the user. However, since our problem is well-posed and the Newton solver will converge regardless of the initial guess, we skip this step and use the default zero-initialized function. Next, we set the parameter value of the problem using its `set_mu()` method. Finally, we call the `solve()` method of the Newton solver. This method takes two arguments: the `myTestProblem` object and a `PETScVector` that holds the solution. Here we have to pass `myTestProblem.u.vector()`, which is the underlying `PETScVector` of the unknown solution function. The `solve` method returns the number of iterations perform and a flag indicating whether the solver has successfully converged.

 # visualization of solution The solution can now be visualized using the `plot()` function provided by FEniCS. The resulting plot can then be edited and displayed using the standard functions from the `matplotlib.pyplot` package.

3.5 Solver Classes for Damped Newton Methods in FEniCS

For each damping strategy introduced in section 3.2, we implemented a corresponding Newton solver class. The full source code of these solver classes is provided in the package `rb_semilinear.nl_solver` in [15]. Most of the strategies were implemented using FEniCS's built-in `NewtonSolver` class. Specifically, we defined the following classes:

- `MyNewton_ord` for the ordinary Newton method (**ord**)
- `MyNewton_adaptDamp` for the adaptive damping strategy (**adaptDamp**)
- `MyNewton_simplDamp` for the simple damping strategy (**simplDamp**)

These classes inherit from `NewtonSolver` but additionally provide methods, that implement the corresponding damping strategies and collect relevant informations during the iterations.

One exception is the **nleqerr** strategy. This damping strategy is already included in PETSc, an open-source library for solving PDE-based problems. FEniCS allows to use PETSc solvers through the `PETScSNESSolver` interface. We used this interface to implement the class `MyNewton_nleqerr`, which inherits from `PETScSNESSolver` and automatically sets the `nleqerr` strategy. More details on the implementation of this PETSc Solver can be found in [16].

All solver classes can be initialized with the following parameters:

- `tol`: convergence tolerance,
- `maxit`: maximum number of Newton iterations,
- `report`: whether to print iteration progress,
- `lam_min`: minimum damping parameter (only for damped methods).

The four custom Newton solver classes are collected in a factory class `MyNewtonSolver`. Additionally to the parameters summarized above, this class is initialized with a parameter `solver_type`. Depending on this parameter, an instance of the corresponding Newton solver class is returned. The parameter `solver_type` can be set to:

- `nleqerr`:→ `MyNewton_nleqerr`
- `adaptDamp`:→ `MyNewton_adaptDamp`
- `simplDamp`:→ `MyNewton_simplDamp`
- `ord`:→ `MyNewton_ord`

Remark 3.1
It is important to note, that the previously defined `MyTestProblem` class cannot yet be solved using our custom Newton solver classes. In each Newton iteration, the solver calls the methods `F` and `J`. The `NewtonSolver` provides a `PETScVector b` (or `PETScMatrix A`) to store the assembled residual vector (or Jacobian matrix). It also provides a `PETScVector x`, which holds the current degrees of freedom of the solution, i.e. the current Newton iterate.

However, in our implementation of `MyTestProblem`, we do not use the vector `x` directly. Instead, we rely on the class variable `self.u`. This might seem confusing, but it is common practice if using FEniCS's built-in `NewtonSolver`, which updates `myTestProblem.u.vector()` (the parameter which has to be passed to its `solve` method) automatically before each call of `F` and `J`. In that case, `x` and `self.u.vector()` share the same data.

But this changes when using our custom Newton solvers with damping strategies. These solvers pass a vector `x` that may not match the current `self.u.vector()`. To make the code compatible, we need to redefine the linear and bilinear forms as functions of an input `Function u`:

```
        self._a = lambda u: action(self.weakform, u)
        du = TrialFunction(self.V_h)
        self._l = lambda u: derivative(self._a(u), u, du)
```

Then, for example in the `F` method, we must first create a `Function` from the vector `x` and use it to assemble the correct form:

```
        def F(self, b: PETScVector, x: PETScVector):
          u = Function(self.V_h)
          u.vector().set_local(x.get_local())
          assemble(self._l(u), tensor=b)
          [bc.apply(b, x) for bc in self.bcs]
```

Now the `MyTestProblem` class can be solved using a custom Newton solver class just like before. In the following example, `solver` is an instance of the class `MyNewton_nleqerr`.

```
  # solve myTestProblem
  myTestProblem = MyTestProblem(N_h = 1000)
  solver = MyNewtonSolver(tol=1/1000**2, maxit=100, report=True,
                          solver_type="nleqerr")
  # --> initFunc(myTestProblem.u)
  myTestProblem.set_mu(mu=0.05)
  nit, conv = solver.solve(myTestProblem, myTestProblem.u.vector())
```

4 Reduced Basis Method for Nonlinear Elliptic PDEs

In this section, we first provide a short overview of the Reduced Basis (RB) method, before introducing the RB Method for the case of nonlinear elliptic PDEs. We then introduce two standard techniques for constructing a reduced basis: the Greedy algorithm and Proper Orthogonal Decomposition (POD). Finally, we present the h-type Greedy algorithm - an adaptive sampling strategy. The following sections are based on [14] and [17], unless stated otherwise. Note that we skip any theorems on Wellposedness of the problems, as they do not necessary hold for the problems considered in this thesis.

4.1 Overview

The Reduced Basis (RB) Method is a model order reduction technique for parametrized PDEs, aiming to approximate high-fidelity solutions $u_h \in V_h$ in a low-dimensional subspace $V_N \subset V_h$ with $N \ll \dim(V_h)$ to reduce computational costs.

Given a bounded parameter domain $\mathcal{P} \subset \mathbb{R}^p$, $p \geq 1$ we consider the pPDE with parameter $\boldsymbol{\mu} \in \mathcal{P}$

$$G(u(\boldsymbol{\mu}), \boldsymbol{\mu}) = 0$$

where $G(\cdot, \boldsymbol{\mu})$ is a parameter dependend (linear or nonlinear) differential operator and $u(\boldsymbol{\mu})$ denotes a solution of the problem. Moreover we define a parametric manifold

$$\mathcal{M} = \{u(\boldsymbol{\mu}) : \boldsymbol{\mu} \in \mathcal{P}\}.$$

Now let $\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_N \in \mathcal{P}$ and the corresponding snapshots $u(\boldsymbol{\mu}_1), ..., u(\boldsymbol{\mu}_N) \in \mathcal{M}$ be given. If $\mathcal{M}$ is smooth, that is $u(\boldsymbol{\mu})$ depends smoothly on the paramters, and if the parameter samples are well chosen, it is possible for any $\boldsymbol{\mu} \in \mathcal{P}$ to compute good approximation of $u(\boldsymbol{\mu})$ as linear combination of the N snapshots $u(\boldsymbol{\mu}_1), ..., u(\boldsymbol{\mu}_N)$. That means the reduced space V_N is spanned by snapshots and the reduced basis is the collection of the snapshots. Since in general the snapshots $u(\boldsymbol{\mu})$ are not known, we use high fidelity approximations $u_h(\boldsymbol{\mu}_1), ..., u_h(\boldsymbol{\mu}_N)$ by finite element method (FEM) yielding the discrete manifold

$$\mathcal{M}_h = \{u_h(\boldsymbol{\mu}) \in V_h \mid \boldsymbol{\mu} \in \mathcal{P}\}. \tag{4.1}$$

For a given dimension N, an optimal subspace V_N is defined by the Kolmogorov N-width, which is the maximum error of the best-approximating linear subspace

$$d_N(\mathcal{M}_h) := \inf_{\substack{V_N \subset V_h \\ \dim(V_N)=N}} \sup_{u_h \in \mathcal{M}_h} \inf_{v_N \in V_N} \|u_h - v_N\|_V.$$

A rapid decay of $d_N(\mathcal{M}_h)$ for increasing N ensures that low-dimensional approximations are accurate and the RB method effective.

The RB method is decomposed in an offline and an online phase. During the offline phase a reduced problem is built by generating a reduced basis for the subspace V_N and precomputing further auxiliary quantities. This phase is computationally very expensive, as several high fidelity approximations have to be computed. However, this pays off in the online phase, where for varying parameters $\boldsymbol{\mu}$ the reduced problem is solved, that is an approximated solution can be computed rapidly. Depending on the underlying problem type , even a posteriori error bounds can be provided.

4.2 Problem Formulation (Nonlinear problems)

For the RB approximation we begin with the weak formulation of our parameterized PDE. Consider a nonlinear, elliptic parameterized differential operator $G(\cdot, \boldsymbol{\mu})$ over the spatial domain $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$. For any given $\boldsymbol{\mu} \in \mathcal{P}$ we get the strong form of the PDE with Dirichlet boundary conditions:

For given $\boldsymbol{\mu} \in \mathcal{P}$ find $u(\boldsymbol{\mu})$

$$\begin{aligned} G(u(\boldsymbol{\mu}), \boldsymbol{\mu}) &= 0 \quad \text{in } \Omega, \\ u(\boldsymbol{\mu}) &= 0 \quad \text{on } \partial\Omega. \end{aligned}$$

Define the function space of test functions as

$$V := H_0^1(\Omega) = \{v \in H^1(\Omega) : \exists (v_k)_{k\in\mathbb{N}} \subset C_0^\infty : v_k \to v \text{ in } H^1(\Omega)\}.$$

Now for every $\boldsymbol{\mu} \in \mathcal{P}$, $G(\cdot, \boldsymbol{\mu}) : V \to V'$ denotes a mapping representing the semilinear PDE.

The **strong formulation** of the parametrized nonlinear problem reads:

For given $\boldsymbol{\mu} \in \mathcal{P}$ find $u(\boldsymbol{\mu}) \in V$ such that

$$G(u(\boldsymbol{\mu}), \boldsymbol{\mu}) = 0 \quad \text{in } V' \tag{4.2}$$

The corresponding **weak formulation** is:

For given $\boldsymbol{\mu} \in \mathcal{P}$ find $u(\boldsymbol{\mu}) \in V$ such that

$$g(u(\boldsymbol{\mu}), v, \boldsymbol{\mu}) = 0 \quad \forall v \in V, \tag{4.3}$$

with $g(u(\boldsymbol{\mu}); v; \boldsymbol{\mu}) = \langle G(u(\boldsymbol{\mu}); \boldsymbol{\mu}), v\rangle_{L^2(\Omega)}$.

4.3 High-Fidelity Approximation

In principal, this chapter is a repetition of section 3.3. However, to give a complete overview of the reduced basis method, we now summarize that section using a consistent notation.

Now let $V_h \subset V$ be a finite-dimensional subspace of V with $h > 0$.
The problem for the **high-fidelity approximation** then reads:

For given $\boldsymbol{\mu} \in \mathcal{P}$ find $u_h(\boldsymbol{\mu}) \in V_h$ such that

$$g(u_h(\boldsymbol{\mu}), v_h, \boldsymbol{\mu}) = 0 \quad \forall v_h \in V_h. \tag{4.4}$$

This system of semilinear equations is solved using Newton's method, as described in section 3.3. Note that in section 3.3 we first linearize the PDE (as it is given in 4.2) before applying a discretization. Alternatively, one could first discretize the nonlinear problem, as shown in (4.4), and then linearize the resulting system. Regardless of the chosen approach, both methods lead to the same iterative process in which a sequence of linear systems must be solved, as it is shown in (3.26). Here we denote the Fréchet derivative of $g(z, \cdot; \boldsymbol{\mu})$ at $z \in V$ by

$$dg[z](w, v, \boldsymbol{\mu}) = \langle G'(z; \boldsymbol{\mu})w, v\rangle \quad \forall w, v \in V.$$

At each Newton iteration, this results in solving a **linear system of equations** of the form:

For given $\boldsymbol{\mu} \in \mathcal{P}$ find $\boldsymbol{\partial}\mathbf{u}_h \in \mathbb{R}^{N_h}$ such that

$$\mathbb{J}_h(\mathbf{u}_h^{(k)}; \boldsymbol{\mu})\boldsymbol{\partial}\mathbf{u}_h = -\mathbf{G}_h(\mathbf{u}_h^{(k)}; \boldsymbol{\mu}), \tag{4.5}$$

with Jacobian matrix $\mathbb{J}_h(\mathbf{u}_h^{(k)}; \boldsymbol{\mu}) \in \mathbb{R}^{N_h \times N_h}$ and residual vector $\mathbf{G}_h(\mathbf{u}_h^{(k)}; \boldsymbol{\mu}) \in \mathbb{R}^{N_h}$ defined as

$$\begin{aligned}(\mathbb{J}_h(\mathbf{u}_h^{(k)}; \boldsymbol{\mu}))_{i,j} &= dg[u_h^{(k)}(\boldsymbol{\mu})](\phi^j, \phi^i; \boldsymbol{\mu}), \quad i,j = 1, ..., N_h,\\ (\mathbf{G}_h(\mathbf{u}_h^{(k)}; \boldsymbol{\mu}))_i &= g(u_h^{(k)}(\boldsymbol{\mu}); \phi^i; \boldsymbol{\mu}), \quad i = 1, ..., N_h.\end{aligned}$$

4.4 Reduced Basis Approximation

To speed up the computation of high-fidelity solutions, we construct a RB model by projecting the high-fidelity problem (4.4) onto a low-dimensional subspace $V_N \subset V_h$. This subspace is generated from a set of snapshots of the high-fidelity problem by means of either proper orthogonal decomposition (POD) or greedy algorithm, which will be introduced in the following section. For now, RB space is spanned by high-fidelity snapshots:

$$V_N = \text{span}\{u(\boldsymbol{\mu}_1), ..., u(\boldsymbol{\mu}_N)\} = \text{span}\{\xi_1, ..., \xi_N\}.$$

The functions ξ_i, $i = 1, ..., N$ denote orthonormal basis functions.

Then the **reduced problem** is:

For given $\boldsymbol{\mu} \in \mathcal{P}$ find $u_N(\boldsymbol{\mu}) \in V_N$ such that

$$g(u_N(\boldsymbol{\mu}), v_N, \boldsymbol{\mu}) = 0 \quad \forall v_N \in V_N, \tag{4.6}$$

with $g(u_N(\boldsymbol{\mu}); v_N; \boldsymbol{\mu}) = \langle G(u_N(\boldsymbol{\mu}); \boldsymbol{\mu}), v_N \rangle_{L^2(\Omega)}$.

This reduced problem is solved by Newton's method as described in the following. Given the reduced basis $\Phi_N = \{\xi_m\}_{m=1}^{N_h}$, a solution in the reduced space can be expressed as a linear combination of reduced basis functions

$$u_N(\boldsymbol{\mu}) = \sum_{m=1}^{N} u_N^{(m)}(\boldsymbol{\mu})\xi_m \ \in V_N, \tag{4.7}$$

where $\mathbf{u}_N(\boldsymbol{\mu}) = (u_N^{(1)}(\boldsymbol{\mu}), ..., u_N^{(N)}(\boldsymbol{\mu})) \in \mathbb{R}^N$ denote the *RB coefficients*.

Next, we define a transformation matrix $\mathbb{V} \in \mathbb{R}^{N_h \times N}$. For this we first expand each RB basis function w.r.t. the basis functions $\{\phi^i\}_{i=1}^{N_h}$ of V_h

$$\xi_m = \sum_{i=1}^{N_h} \xi_m^{(i)} \phi^i, \quad 1 \le m \le N \quad \text{with } \boldsymbol{\xi}_m = (\xi_m^{(1)}, ..., \xi_m^{(N_h)}) \in \mathbb{R}^{N_h}. \tag{4.8}$$

The columns of the transformation matrix $\mathbb{V}$ contain the DOFs of the RB functions

$$\mathbb{V} = [\boldsymbol{\xi}_1 | ... | \boldsymbol{\xi}_N] \in \mathbb{R}^{N_h \times N}. \tag{4.9}$$

A reduced solution $u_N(\boldsymbol{\mu}) \in V_N$ with RB coefficients $\mathbf{u}_N(\boldsymbol{\mu}) \in \mathbb{R}^N$ can now be expressed as

$$u_N(\boldsymbol{\mu}) = \sum_{i=1}^{N_h} (\mathbb{V}\mathbf{u}_N(\boldsymbol{\mu}))_i \phi^{(i)} \in V_N \subset V_h. \tag{4.10}$$

Note that the columns of $\mathbb{V}$ are basis vectors for the degrees of freedom of a reduced solution $u_N(\boldsymbol{\mu}) \in V_N$. Therfore, we refer to $\mathbb{V}$ as the *discrete reduced basis* and the space spanned by the vectors $(\{\boldsymbol{\xi}_i\}_{i=1}^N)$ as *discrete reduced space*.

For $v_N = \xi_n$, we obtain a **discrete reduced problem** (nonlinear system of equations):

Given $\boldsymbol{\mu} \in \mathcal{P}$, find $\mathbf{u}_N(\boldsymbol{\mu}) \in \mathbb{R}^N$ s.t.

$$g(u_N(\boldsymbol{\mu}); \xi_n; \boldsymbol{\mu}) = 0 \quad \forall n = 1, ..., N. \tag{4.11}$$

where $u_N(\boldsymbol{\mu}) = \sum_{m=1}^N u_N^{(m)}(\boldsymbol{\mu})\xi_m = \sum_{i=1}^{N_h} (\mathbb{V}\mathbf{u}_N(\boldsymbol{\mu}))_i \phi^{(i)} \in V_N$.

Inserting (4.8) and (4.10) into (4.11) yields

$$(\mathbf{G}_N(\mathbf{u}_N(\boldsymbol{\mu}); \boldsymbol{\mu}))_n := g(\sum_{i=1}^{N_h} (\mathbb{V}\mathbf{u}_N(\boldsymbol{\mu}))_i \phi^{(i)}; \sum_{r=1}^{N_h} \xi_n^{(r)} \phi^r; \boldsymbol{\mu}) = (\mathbb{V}^T \mathbf{G}_h(\mathbb{V}\mathbf{u}_N(\boldsymbol{\mu}); \boldsymbol{\mu}))_n$$

Therefore the reduced residual vector is

$$\mathbf{G}_N(\mathbf{u}_N(\boldsymbol{\mu}); \boldsymbol{\mu}) = \mathbb{V}^T \mathbf{G}_h(\mathbb{V}\mathbf{u}_N(\boldsymbol{\mu}); \boldsymbol{\mu}). \tag{4.12}$$

In a similar way we get the reduced Jacobian matrix

$$\mathbb{J}_N(\mathbf{u}_N; \boldsymbol{\mu}) = \mathbb{V}^T \mathbb{J}_h(\mathbb{V}\mathbf{u}_N; \boldsymbol{\mu})\mathbb{V} \tag{4.13}$$

This results in k**-th step of Newton's method**

Given $\boldsymbol{\mu} \in \mathcal{P}$, find $\partial\mathbf{u}_N(\boldsymbol{\mu}) \in \mathbb{R}^N$ s.t.

$$\mathbb{J}_N(\mathbf{u}_N^k; \boldsymbol{\mu})\partial\mathbf{u}_N = -\mathbf{G}_N(\mathbf{u}_N^k; \boldsymbol{\mu}) \tag{4.14}$$

with reduced Jacobian matrix $\mathbb{J}_N(\mathbf{u}_N^{(k)}; \boldsymbol{\mu}) \in \mathbb{R}^{N \times N}$ and reduced residual vector $\mathbf{G}_N(\mathbf{u}_N^{(k)}; \boldsymbol{\mu}) \in \mathbb{R}^N$ defined as

$$(\mathbb{J}_N(\mathbf{u}_N^{(k)}; \boldsymbol{\mu})) = \mathbb{V}^T \mathbb{J}_h(\mathbb{V}\mathbf{u}_N; \boldsymbol{\mu})\mathbb{V},$$
$$(\mathbf{G}_N(\mathbf{u}_N^{(k)}; \boldsymbol{\mu})) = \mathbb{V}^T \mathbf{G}_h(\mathbb{V}\mathbf{u}_N(\boldsymbol{\mu}); \boldsymbol{\mu}).$$

Remark 4.1
Obviously this does not yet fully achieve the desired goal of computational efficiency. As seen in (4.12) and (4.13), the assembly of the reduced Jacobian matrix and the reduced residual vector (required in every iteration of Newton's method) still requires computing their high-fidelity counterparts. In cases with low polynomial nonlinearity, efficiency can be improved by assuming affine parameter dependence and defining appropriate multilinear forms. Otherwise, hyper-reduction techniques such as Empirical Interpolation Method (EIM) can be used to approximate the nonlinear terms.[11] Since our primary goal is to assess whether the RB approach is feasible in this context, we will not pursue further optimizations here. However, for completeness, we introduce a strategy for online-offline decomposition in case of quadratic nonlinearity in the following section.

4.5 Offline-Online decomposition for quadratically nonlinear PDE

In this section, we describe how to perform an efficient offline-online decomposition when dealing with a quadratically nonlinearity. A key requirement for the RB method to be efficient is affine

parameter dependence of the PDE. If this is not the case, an affine structure can be achieved using the Empirical Interpolation Method (EIM).[11] Given affine parameter dependence, the high-fidelity residual vector can be expressed as:

$$\mathbf{G}_h(\mathbf{u}_h, \boldsymbol{\mu}) = \sum_{q=1}^{Q_g} \theta_g^q(\boldsymbol{\mu}) \mathbf{G}_h^q(\mathbf{u}_h).$$

Now, if dealing with a quadratically nonlinear PDE, this residual vector can be written in a bilinear form:

$$\mathbf{G}_h(\mathbf{u}_h; \boldsymbol{\mu}) = \tilde{\mathbf{G}}_h(\mathbf{u}_h, \mathbf{u}_h; \boldsymbol{\mu}).$$

Assuming affine parameter dependence of $\mathbf{G}_h$ we have:

$$\mathbf{G}_h(\mathbf{u}_h; \boldsymbol{\mu}) = \sum_{q=1}^{Q_g} \theta_g^q(\boldsymbol{\mu}) \mathbf{G}_h^q(\mathbf{u}_h) = \sum_{q=1}^{Q_g} \theta_g^q(\boldsymbol{\mu}) \tilde{\mathbf{G}}_h^q(\mathbf{u}_h, \mathbf{u}_h).$$

With equation (4.12) and the equation from above, the assmbly of the reduced residual is as follows:

$$\mathbf{G}_N(\mathbf{u}_N; \boldsymbol{\mu}) = \mathbb{V}^T \mathbf{G}_h(\mathbb{V}\mathbf{u}_N; \boldsymbol{\mu}) = \sum_{q=1}^{Q_g} \theta_g^q(\boldsymbol{\mu}) \mathbb{V}^T \tilde{\mathbf{G}}_h^q(\mathbb{V}\mathbf{u}_N, \mathbb{V}\mathbf{u}_N)$$

Since $\mathbb{V}\mathbf{u}_N = \sum_{n=1}^{N} u_N^{(n)} \boldsymbol{\xi}_n$ with basis vectors $\boldsymbol{\xi}_n$ (columns of $\mathbb{V}$), we can express the reduced residual as:

$$\mathbf{G}_N(\mathbf{u}_N; \boldsymbol{\mu}) = \sum_{q=1}^{Q_g} \theta_g^q(\boldsymbol{\mu}) \sum_{n,m=1}^{N} u_N^{(n)} u_N^{(m)} \mathbb{V}^T \tilde{\mathbf{G}}_h^q(\boldsymbol{\xi}_n, \boldsymbol{\xi}_m) \tag{4.15}$$

Similary the reduced Jacobian matrix can be expressed as:

$$\mathbb{J}_N(\mathbf{u}_N; \boldsymbol{\mu}) = \sum_{q=1}^{Q_j} \theta_j^q(\boldsymbol{\mu}) \sum_{n=1}^{N} u_N^{(n)} \mathbb{V}^T \mathbb{J}_h^q(\boldsymbol{\xi}_N) \mathbb{V}. \tag{4.16}$$

From this we can derive the following Offline-Online Decomposition:

- **Offline:**
  - Precompute $N^2 Q_g$ vectors: $\mathbb{V}^T \tilde{\mathbf{G}}_q(\xi_n, \xi_m)$ for $n, m = 1, \ldots, N$ and $q = 1, \ldots, Q_g$.
  - Precompute $N Q_j$ matrices: $\mathbb{V}^T \mathbb{J}_q(\boldsymbol{\xi}_n) \mathbb{V}$ for $n = 1, \ldots, N$ and $q = 1, \ldots, Q_j$.
- **Online:**
  - Assemble reduced problem using precomputed vectors and matrices, RB coefficients $u_N^{(n)}$ and parameter-dependent coefficient functions $\theta_g^q(\boldsymbol{\mu})$ and $\theta_j^q(\boldsymbol{\mu})$.
  - Solve reduced problem with number of operations depending in N.

This decomposition ensures that, in the online phase, the reduced model can be evaluated very efficiently, independent of the size of the high-fidelity system.

4.6 Construction of Reduced Basis Spaces

In this section we address two common methods for constructing a reduced basis space V_N: the Proper Orthogonal Decomposition (POD) and the Greedy algorithm.

The goal is to approximate the discrete solution manifold

$$\mathcal{M}_h := \{u_h(\boldsymbol{\mu}) : \boldsymbol{\mu} \in \mathcal{P}\} \subset V_h$$

as good as possible using a low-dimensional linear subspace V_N with $\dim(V_N) << V_h$.

The theoretical possible quality of such an approximation is measured by the Kolmogorov N-width, which quantifies how well the manifold $\mathcal{M}$ can be approximated by the best possible N-dimensional subspace:

$$d_N(\mathcal{M}_h) := \inf_{\substack{V_N \subset V_h \\ \dim(V_N)=N}} \sup_{u_h \in \mathcal{M}_h} \inf_{v_N \in V_N} \|u_h - v_N\|.$$

Since in our case V is a Hilbert space, we can replace the inner infimum with the norm of the projection error:

$$d_N(\mathcal{M}_h) := \inf_{\substack{V_N \subset V_h \\ \dim(V_N)=N}} \sup_{u_h \in \mathcal{M}_h} \|u_h - P_{V_N} u_h\|.$$

where P_{V_N} denotes the orthogonal projection onto V_N.

However, finding the exact optimal space V_N is computationally infeasible. To make the problem tractable, we approximate the full solution manifold by evaluating it at a finite number of parameter values. Specifically, we define a training set of parameters:

$$\mathcal{P}_{ns} = \{\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_{ns}\} \subset \mathcal{P}.$$

and consider the corresponding solutions (snapshots)

$$S = \{u_h(\boldsymbol{\mu}_i)\}_{i=1}^{ns} \subset \mathcal{M}_h,$$

which should represent the solution manifold $\mathcal{M}_h$ as good as possible. We then build a reduced space V_N that approximates this set of snapshots as good as possible. For this we consider two strategies: POD and Greedy algorithm. Defining the projection error for a parameter $\boldsymbol{\mu}$ as

$$E(\boldsymbol{\mu}) := \|u_h(\boldsymbol{\mu}) - P_{V_N} u_h(\boldsymbol{\mu})\|,$$

POD is aiming at choosing a reduced space V_N that minimizes the projection error for all $\boldsymbol{\mu} \in \mathcal{P}_{ns}$ in a mean squared sense, while the greedy procedure is aiming at minimizing the maximum approximation error.

The resulting reduced basis Φ_N spans the reduced basis space V_N. Both methods produce a reduced basis which is hierarchical. That is $\Phi_n \subset \Phi_m$ for $n \leq m$.

4.6.1 Greedy Algorithm

The Greedy algorithm is a standard approach for constructing a reduced basis in a problem oriented, iterative manner. Let

$$\Delta_N(V_N, \boldsymbol{\mu}) \in \mathbb{R}^+$$

be an a posteriori error erstimate that predicts the error for the parameter $\boldsymbol{\mu}$ when using V_N as reduced space. That means following inequation holds:

$$\|u_h(\boldsymbol{\mu}) - u_N(\boldsymbol{\mu})\|_V \leq \Delta_N(V_N, \boldsymbol{\mu}).$$

where $u_N(\boldsymbol{\mu})$ is the solution of the reduced problem (4.6) and $u_h(\boldsymbol{\mu})$ of the high-fidelity problem (4.4).

At each step, the reduced basis is expanded by a new snapshot $u_h(\boldsymbol{\mu}) \in S$ corresponding to a parameter $\boldsymbol{\mu} \in \mathcal{P}_{ns}$, chosen specifically to minimize the maximal projection error $\max_{\boldsymbol{\mu}\in\mathcal{P}_{ns}} E(\boldsymbol{\mu})$. More precisely, among all paramter values $\boldsymbol{\mu} \in \mathcal{P}_{ns}$ the algorithm selects the one for which the current reduced model performs the poorest, i.e. it yields the largest estimated error $\Delta_N(Y, \boldsymbol{\mu})$. Here, Y denotes the current reduced basis space. The corresponding high-fidelity solution is then orthonormalized by using the Gram-Schmidt procedure w.r.t. $\|\cdot\|_{L^2(\Omega)}$ and added to the reduced basis. Note that the inner product $\langle\cdot,\cdot\rangle_{L^2(\Omega)}$ can be computed as

$$\langle u, v\rangle_{L^2(\Omega)} = \langle\sum_{i=1}^{N_h} u^{(i)}\phi^i, \sum_{j=1}^{N_h} v^{(i)}\phi^i\rangle_{L^2(\Omega)} = \sum_{i=1}^{N_h}\sum_{j=1}^{N_h} u^{(i)}v^{(j)}\langle\phi^i,\phi^j\rangle_{L^2} = \boldsymbol{v}^T\mathbb{M}\mathbf{u} =: \langle\mathbf{u},\boldsymbol{v}\rangle_{\mathbb{M}}$$

and therfore the norm $\|\cdot\|_{L^2(\Omega)}$ by

$$\|u\|_{L^2(\Omega)} = \sqrt{\langle\mathbf{u},\mathbf{u}\rangle_{\mathbb{M}}} =: \|\mathbf{u}\|_{\mathbb{M}},$$

where $\mathbb{M} = \{\langle\phi^i,\phi^j\rangle_{L^2(\Omega)}\}_{i,j=1}^{N_h}$ denotes the mass matrix. The orthogonal projection of an high-fidelity solution $u \in V_h$ onto a subspace $V_N \subset V_h$ w.r.t. $\|\cdot\|_{L^2(\Omega)}$ is given by

$$P_{V_N}^{L^2} u = \sum_{j=1}^{N}\langle u, \xi_j\rangle_{L^2(\Omega)}\xi_j \in V_N \subset V_h,$$

where ξ_j, $j = 1, ..., N$ denote the basis functions of the reduced basis Φ_N, that spans the reduced space V_N. This projection can be computationally realised by projecting the degrees of freedom $\mathbf{u} \in \mathbb{R}^{N_h}$ of the high-fidelity function $u \in V_h$ onto the discrete reduced space spanned by the columns of $\mathbb{V}$ w.r.t. $\mathbb{M}$:

$$\mathbb{P}_{\mathbb{V}}^{\mathbb{M}}\mathbf{u} = \sum_{j=1}^{N}\langle\mathbf{u},\boldsymbol{\xi}_j\rangle_{\mathbb{M}}\boldsymbol{\xi}_j = \mathbb{V}\mathbb{V}^T\mathbb{M}\mathbf{u} \tag{4.17}$$

With this the discrete Greedy algorithm can be formulated as in algorithm 4. Here $\mathbf{u}_h(\boldsymbol{\mu})\mathbb{R}^{N_h}$ denotes the vector of degrees of freedom of the solution $u_h(\boldsymbol{\mu})$. Further $\mathbb{M}$ denotes the mass matrix.

There are different possibilities for the error indicator $\Delta(Y, \boldsymbol{\mu})$. The selection of this indicator yields different types of the Greedy algorithm:

- **strong Greedy** - Projection error as indicator:

$$\Delta(Y, \boldsymbol{\mu}) := \inf_{v\in Y}\|u_h(\boldsymbol{\mu}) - v\|_{L^2} = \left\|u_h(\boldsymbol{\mu}) - P_Y^{L^2}u_h(\boldsymbol{\mu})\right\|_{L^2} \tag{4.18}$$

  Here $P_Y : V_h \to Y$ denotes the orthogonal projection of a function $u_h \in V_h$ into the reduced space Y. The error indicator here is very computationally expensive to evaluate as high-dimensional operators and all snapshots $u_h(\boldsymbol{\mu})$ are required, hence it can be necessary that $\mathcal{P}_{ns}$ is small. But the advantage is that no RB-model or a posteriori error estimator is needed.

Algorithm 4: Greedy($P_{ns}, \boldsymbol{\mu}_1, \epsilon_{\text{tol}}, N_{\max}$)

Data: parameter training set $\mathcal{P}_{ns} \subset \mathcal{P}$, first parameter $\boldsymbol{\mu}_1$, error tolerance $\epsilon_{\text{tol}} > 0$, maximum number of iterations $N_{\max}$

Result: Basis $\mathbb{V} \subset \mathbb{R}^{N_h \times N}$

$N = 0, \epsilon_0 = \epsilon_{\text{tol}} + 1$

$\mathcal{P}_g = [], \mathbb{V} = []$

while $N < N_{\max}$ *and* $\epsilon_0 > \epsilon_{tol}$ **do**

  $N \leftarrow N + 1$

  compute $\mathbf{u}_h(\boldsymbol{\mu}_N)$

  $\boldsymbol{\zeta}_N$=GRAMSCHMIDT($\mathbb{V}, \mathbf{u}_h(\boldsymbol{\mu}), \mathbb{M}$)

  $\mathbb{V} \leftarrow [\mathbb{V}\boldsymbol{\zeta}_N]$

  $\mathcal{M}_g \leftarrow \mathcal{M}_g \cup \{\boldsymbol{\mu}_N\}$

  $[\epsilon_N, \boldsymbol{\mu}_{N+1}] = \max_{\boldsymbol{\mu} \in \mathcal{P}_{\text{train}}} \Delta_N(\boldsymbol{\mu})$

end

return $\mathbb{V} = [\boldsymbol{\zeta}_1, \ldots, \boldsymbol{\zeta}_N]$, $\mathcal{M}_g$

Algorithm 5: Gram-Schmidt orthonormalization

Function *GRAMSCHMIDT*($\mathbb{V}, \mathbf{u}, \mathbb{M}$)**:**

  if $\mathbb{V} = []$ **then**

    $\mathbf{x} = \mathbf{u}$

  else

    $\mathbf{x} = \mathbf{u} - \mathbb{V}\mathbb{V}^T\mathbb{M}\mathbf{u}$

  end

  $\mathbf{x} \leftarrow \mathbf{x}/\left\|\mathbf{x}\right\|_{\mathbb{M}}$

return $\mathbf{x}$

- **weak Greedy** - A posteriori error estimator as indicator:

$$\Delta(Y, \boldsymbol{\mu}) := \Delta_u(\boldsymbol{\mu}) \tag{4.19}$$

  where $\Delta_u(\boldsymbol{\mu})$ is an a posteriori error estimator, the evaluation of which is very cheap. Moreover, only N solves of the high-fidelity problem have to be computed. Therefore the training set $\mathcal{P}_{ns}$ can be selected much larger. Such estimators have been established for linear problems [18] as well as for certain solutions of nonlinear problems (e.g. Nonlinear Elliptic PDEs with coercive, symmetric and continuous linear part [19]).

Since we do not have an a posteriori error estimator at hand, we will use the strong Greedy algorithm in the following sections.

The greedy algorithm is also theoretically well justified. In [2] a theoretic result is presented that establishes the convergence behavior of the Greedy algorithm. Specifically it shows, that if $\mathcal{M}_h$ can be well approximated by some linear subspaces, the Kolmogorov-N-width decays exponantially

$$d_N(\mathcal{M}) \leq Ce^{-\beta N}$$

for positive constants C and β. Then the Greedy algorithm is guaranteed to construct an exponentially converging approximation space:

$$\|u - P_{V_N} u\| \leq ce^{-\alpha N}.$$

4.6.2 Proper Orthogonal Decomposition

Proper Orthogonal Decomposition (POD) is a technique used to construct a reduced basis that represents a set of high-fidelity solutions as accurately as possible in the mean square error sense. It identifies an optimal low-dimensional basis by using Singular Value Decomposition (SVD).

Let a set of parameter samples and the corresponding snapshot set be given by

$$\begin{aligned}\mathcal{P}_{ns} &= \{\boldsymbol{\mu}_i \in \mathcal{P}\}_{i=1}^{n_s},\\ S &= \{u_h(\boldsymbol{\mu}_i)\}_{i=1}^{n_s} \subset \mathcal{M}_h \subset V_h.\end{aligned}$$

Denoting by $\Phi_h := \{\phi^j\}_{j=1}^{N_h}$ a basis for V_h, each solution $u_h \in S$ can be expressed as linear combination of basis functions

$$u_h(\boldsymbol{\mu}_j) = \sum_{i=1}^{N_h} u_j^{(i)} \phi_i,$$

where $\mathbf{u}_{h,j} = (u_{h,j}^{(1)}, ..., u_{h,j}^{(N_h)}) \in \mathbb{R}^{N_h}$ denote the degrees of freedom of the high-fidelity solution. The degrees of freedom are collected in the snapshot matrix $\mathbb{S}$, that is a matrix of coefficients:

$$\mathbb{S} = [\mathbf{u}_{h,1}|....|\mathbf{u}_{h,ns}] \in \mathbb{R}^{N_h \times ns}.$$

Now we want to apply **Singular Value Decomposition (SVD)** to $\mathbb{S}$, that is:

For a real matrix $\mathbb{S} \in \mathbb{R}^{N_h \times ns}$, there exist two orthogonal matrices

$$\mathbb{U} = [\boldsymbol{\zeta}_1|....|\boldsymbol{\zeta}_{N_h}] \in \mathbb{R}^{N_h \times N_h}, \quad \mathbb{Z} = [\boldsymbol{\psi}_1|...|\boldsymbol{\psi}_n s] \in \mathbb{R}^{ns \times ns}$$

such that

$$\mathbb{S} = \mathbb{U}\Sigma\mathbb{Z}^T, \quad \text{where } \Sigma = \text{diag}(\sigma_1, ..., \sigma_p) \in \mathbb{R}^{N_h \times ns}$$

with singular values orderd as $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p \geq 0$, for $p = \min(N_h, ns)$
The colums of $\mathbb{U}$ are denoted left singular vectors and the colums of $\mathbb{Z}$ right singular vectors.

The singular vectors and values satisfy

$$\begin{aligned}\mathbb{S}\boldsymbol{\psi}_i = \sigma_i\boldsymbol{\zeta}_i \text{ and } \mathbb{S}^T\boldsymbol{\zeta}_i = \sigma_i\boldsymbol{\psi}_i, \quad i = 1, ..., r\\ \Rightarrow \mathbb{S}^T\mathbb{S}\boldsymbol{\psi}_i = \sigma_i^2\boldsymbol{\psi}_i \text{ and } \mathbb{S}\mathbb{S}^T\boldsymbol{\zeta}_i = \sigma_i^2\boldsymbol{\zeta}_i, \quad i = 1, ..., r\end{aligned}$$

Aboves equations show a relationship between the SVD of $\mathbb{S}$ and the spectral decompositions of $\mathbb{S}^T\mathbb{S}$ and $\mathbb{S}\mathbb{S}^T$. Since $\mathbb{S}\mathbb{S}^T$ and $\mathbb{S}^T\mathbb{S}$ are symmetric matrices, the left (resp. right) singular vectors of $\mathbb{S}$ turn out to be the eigenvectors of $\mathbb{S}\mathbb{S}^T$ (resp. $\mathbb{S}^T\mathbb{S}$).

Note that thanks to SVD, a simple characterization of the rank of $\mathbb{S}$ is possible, because $\text{rank}(\mathbb{S}) = \text{rank}(\Sigma)$. Since the rank of a diagonal matrix is equal to the number of nonzero diagonal entries, the number of positive singular values of $\mathbb{S}$ corresponds to its rank. Furthermore, if $\mathbb{S}$ has rank equal to r, then it can be written as

$$\mathbb{S} = \sum_{i=1}^{r} \sigma_i \boldsymbol{\zeta}_i \boldsymbol{\psi}_i^T.$$

This is very powerful, since the partial sum of $k \leq r$ terms is the best possible low-rank approximation of $\mathbb{S}$ w.r.t. the 2-norm or the Frobenius norm. This property is provided in the Schmidt-Eckard-Young theorem (Theorem 6.1 in [14]).

Now we build an optimal **POD-basis** w.r.t. the 2-norm in the following way:

For any $N \leq n_s$ the POD basis $\mathbb{V} \in \mathbb{R}^{N_h \times N}$ of dimension N is defined as the set of the first N left singular vectors $\boldsymbol{\zeta}_1, ..., \boldsymbol{\zeta}_N$ of $\mathbb{U}$ or equivalently the set of vectors

$$\boldsymbol{\zeta}_j = \frac{1}{\sigma_j} \mathbb{S} \boldsymbol{\psi}_j, \quad 1 \leq j \leq N$$

obtained from the first N eigenvectors $\boldsymbol{\psi}_1, ..., \boldsymbol{\psi}_N$ and eigenvalues $\sigma_1, ..., \sigma_N$ of the correlation matrix $\mathbb{C} = \mathbb{S}^T \mathbb{S}$.

This basis $\mathbb{V}$ is orthonormal by construction. The projection of an vector $\mathbf{x}$ onto a subspace spanned by an orthonormal basis $\mathbb{W} = [\mathbf{w}_1 | ... | \mathbf{w}_N] \in \mathbb{R}^{N_h \times N}$ w.r.t. $\|\cdot\|_2$ is given by

$$\mathbb{P}_W \mathbf{x} = \sum_{j=1}^{N} \langle \mathbf{x}, \mathbf{w}_j \rangle_2 \mathbf{w}_j = \mathbb{W}\mathbb{W}^T \mathbf{x}. \tag{4.20}$$

The following theorem states, that $\mathbb{V}$ is the best orthonormal basis w.r.t. $\|\cdot\|_2$. A proof to this theorem can be found in [14], Proposition 6.1.

Theorem 4.2 - $\mathbb{V}$ ***holds the best orthonomal basis w.r.t.*** $\|\cdot\|_2$
Let $\mathcal{V}_N = \{\mathbb{W} \in \mathbb{R}^{N_h \times N} : \mathbb{W}^T \mathbb{W} = \mathbb{I}_N\}$ be the set of all N-dimensional orthonormal bases.
Then

$$\sum_{i=1}^{n_s} \left\| \mathbf{u}_i - \mathbb{V}\mathbb{V}^T \mathbf{u}_i \right\|_2^2 = \min_{\mathbb{W} \in \mathcal{V}_N} \sum_{i=1}^{n_s} \left\| \mathbf{u}_i - \mathbb{W}\mathbb{W}^T \mathbf{u}_i \right\|_2^2 = \sum_{i=N+1}^{r} \sigma_i^2$$

The resulting algorithm is given in algorithm 6. The matrix $\mathbb{C} = \mathbb{S}^T \mathbb{S}$ is denoted correlation matrix. Further $I(N)$ is the percentage of energy of the snapshots captured by the first N POD modes

$$I(N) = \frac{\sum_{i=1}^{N} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2} \geq 1 - \epsilon_{\text{POD}}^2.$$

This implies that:

$$\sum_{i=1}^{n_s} \left\| \mathbf{u}_i - \mathbb{V}\mathbb{V}^T \mathbf{u}_i \right\|_2^2 = \sum_{i=N+1}^{r} \sigma_i^2 \leq \epsilon_{\text{POD}}^2 \sum_{i=1}^{r} \sigma_i^2.$$

Algorithm 6: POD w.r.t. 2-norm

Data: Snapshot-Matrix $\mathbb{S} \in \mathbb{R}^{N_h \times ns}$, $ns < N_h$, $\epsilon_{\text{POD}} > 0$
Result: POD-Moden $\mathbb{V}$

1 $\mathbb{C} \leftarrow \mathbb{S}^T \mathbb{S}$
2 $\mathbb{C}\boldsymbol{\psi}_i = \sigma_i^2 \boldsymbol{\psi}_i, \quad i = 1, \ldots, ns$
3 $\boldsymbol{\zeta}_i = \frac{1}{\sigma_i} S \boldsymbol{\psi}_i, \quad i = 1, \ldots, ns$
4 $N \leftarrow$ minimum integer such that $I(N) \geq 1 - \epsilon_{\text{POD}}^2$
5 **return** $\mathbb{V} = [\boldsymbol{\zeta}_1, \ldots, \boldsymbol{\zeta}_N]$

Remark 4.3
The behavior of the singular values can provide valuable insight into whether a problem is well-suited for model reduction. In the context of POD, we aim to approximate a set of high-fidelity

solutions represented by a set of snapshots S—with as few basis functions as possible.
If S is a good approximation of the underlying solution manifold $\mathcal{M}_h$, then POD yields a near-optimal reduced basis for $\mathcal{M}_h$. A rapid decay of the singular values of S indicates that the problem is well suited to reduction using reduced basis method.

4.7 The h-type Greedy algorithm

To ensure accurate reduced solutions across the full parameter domain $\mathcal{P}$, large training sets may be required. This often leads to larger RB spaces and increased computational cost. To address this, the h-type Greedy algorithm introduces an adaptive strategy: it splits the parameter domain into smaller subdomains based on how the solution behaves in different regions. For each subdomain, a separate reduced model is built, aiming to meet a given error tolerance with a limited number of basis functions.

When computing a reduced solution for a new parameter value, the appropriate parameter subdomain is identified using proximity functions, and the corresponding local reduced model is used. This approach is especially effective when the solution varies significantly across the parameter domain, making snapshots from one region less useful in another. The h-type Greedy algorithm, which is introduced in the following, is developed in detail in [6].

4.7.1 Preliminaries

We start by defining a Boolean vector of length l, which we use to label subdomains,

$$B_l = (1, i_2, ..., i_l), \quad i_k \in \{0, 1\} \quad 2 \leq k \leq l.$$

This vector will help us to describe a hierarchy of subdomains: each Boolean vector (the parent) has two children, defined by appending a 0 or a 1:

$$(B_l, i) = (1, i_2, ..., i_l, i), \quad \text{with } i \in \{0, 1\}.$$

Each Boolean vector corresponds to a *subdomain* $\mathcal{V}_{B_l} \subset \mathcal{P}$, where $\mathcal{P}$ is the full parameter domain. These subdomains satisfy the hierarchical condition:

$$\mathcal{V}_{(B_l,0)} \subset \mathcal{V}_{B_l}, \quad \mathcal{V}_{(B_l,1)} \subset \mathcal{V}_{B_l}.$$

Each subdomain $\mathcal{V}_{B_l}$ is assigned with a set of $\bar{N}$ parameter samples, denoted *model*:

$$\mathcal{M}_{\bar{N},B_l} = \{\boldsymbol{\mu}_{1,B_l}, ..., \boldsymbol{\mu}_{\bar{N},B_l}\} \subset \mathcal{P}.$$

The high-fidelity solutions $u_h(\boldsymbol{\mu}) \in V_h$ with $\boldsymbol{\mu} \in \mathcal{M}_{\bar{N},B_l}$ span the RB space for each subdomain:

$$V_{\bar{N},B_l} = \text{ span}\{u_h(\boldsymbol{\mu}_{1,B_l}), ..., u_h(\boldsymbol{\mu}_{\bar{N},B_l})\} = \text{span}\{\xi_{1,B_l}, ..., \xi_{\bar{N},B_l}\} \quad 1 \leq l \leq L.$$

The reduced basis $\Phi_{\bar{N},B_l} = \{\xi_{i,B_l}\}_{i=1}^{\bar{N}}$ is orthonormalized and the corresponding discrete reduced basis is defined by the DOFs of the basis functions:

$$\mathbb{V}_{\bar{N},B_l} = [\boldsymbol{\xi}_{1,B_l}, ..., \boldsymbol{\xi}_{\bar{N},B_l}].$$

Each model $\mathcal{M}_{\bar{N},B_l}$ is also associated with a representative *anchor point* $\hat{\boldsymbol{\mu}}_{B_l} \in \mathcal{M}_{B_l}$. To decide, to which subdomain a given parameter $\boldsymbol{\mu}$ belongs to, we use a proximity function $d_{B_l} : \mathcal{P} \to \mathbb{R}^+$. As an example, we can choose the Euclidean distance between two points

$$d_{B_l}(\boldsymbol{\mu}) = \left\| \boldsymbol{\mu} - \hat{\boldsymbol{\mu}}_{B_l} \right\|_2.$$

This function is used to recursively determine the Boolean vector corresponding to the subdomain $\mathcal{V}_{(1,i_2^*,\dots,i_l^*)}$ that contains a given parameter value $\boldsymbol{\mu}$.

$$\begin{aligned} i_2^* &= \arg\min_{i\in\{0,1\}} d_{(1,i)}(\boldsymbol{\mu}), \\ i_3^* &= \arg\min_{i\in\{0,1\}} d_{(1,i_2^*,i)}(\boldsymbol{\mu}), \\ &\vdots \\ i_l^* &= \arg\min_{i\in\{0,1\}} d_{(1,i_2^*,i_3^*,\dots,i_{l-1}^*,l)}(\boldsymbol{\mu}). \end{aligned}$$

For the h-type algorithm we will need the Greedy algorithm from section 4.6.1 and an a posteriori error estimator $\Delta_{\bar{N}}(\boldsymbol{\mu})$ from (4.18) or (4.19).

4.7.2 Algorithm

We now describe the h-type Greedy algorithm. The whole procedure is summarized in algorithm 7 more compact. The process starts with the initial Boolean vector $B_l = (1)$, corresponding to the full parameter domain $\mathcal{V}_{(1)} := \mathcal{P}$. We also define a finite training parameter set

$$\mathcal{P}_{(1)}^{\text{train}} \subset \mathcal{V}_{(1)}.$$

We choose an initial parameter anchor point $\hat{\boldsymbol{\mu}}_{(1)} \in \mathcal{P}$, an error tolerance ϵ_{tol}^1 and a maximum RB space dimension $\bar{N}$.

The domain is then defined recursively as follows:

1. For the current subdomain $\mathcal{V}_{B_l}$, construct a reduced basis using the Greedy algorithm (algorithm 4) with the training set $\mathcal{P}_{B_l}^{\text{train}}$. The output is the discrete reduced basis space $\mathbb{V}_{\bar{N},B_l}$ and the model $\mathcal{M}_{\bar{N},B_l}$:

   $$\mathbb{V}_{\bar{N},B_l}, \mathcal{M}_{\bar{N},B_l} = Greedy(\mathcal{P}_{B_l}^{\text{train}}, \hat{\boldsymbol{\mu}}_{B_l}, 0, \bar{N}).$$

   We set the tolerance input of the Greedy algorithm to zero to enforce that exactly $\bar{N}$ basis vectors are selected.

2. Using a posteriori error estimator, compute the maximum error bound

   $$\epsilon_{\bar{N},B_l} = \max_{\boldsymbol{\mu}\in\mathcal{P}_{B_l}^{\text{train}}} \Delta_{\bar{N}}(\boldsymbol{\mu})$$

3. If the maximum error satisfies $\epsilon_{\bar{N},B_l} < \epsilon_{\text{tol}}^1$:
   The current RB space is sufficiently accurate for this subdomain and we terminate the refinement for this branch by assigning:

   $$\hat{\boldsymbol{\mu}}_{\bar{N},(1,i_2,\dots,i_l,0)} = \text{NaN}, \qquad \hat{\boldsymbol{\mu}}_{\bar{N},(1,i_2,\dots,i_l,1)} = \text{NaN},$$

   with NaN standing for "not a number".

4. If the maximum error satisfies $\epsilon_{\bar{N},B_l} \geq \epsilon_{\text{tol}}^1$:

   (i) Define New Anchor Points:
   For two new models $\mathcal{M}_{\bar{N},(B_l,0)}$ and $\mathcal{M}_{\bar{N},(B_l,1)}$ we set the anchor points

   $$\hat{\boldsymbol{\mu}}_{(B_l,0)} := \hat{\boldsymbol{\mu}}_{B_l}, \qquad \hat{\boldsymbol{\mu}}_{(B_l,1)} := \boldsymbol{\mu}_{2,B_l},$$

   where $\boldsymbol{\mu}_{2,B_l} \in \mathbb{V}_{\bar{N},B_l}$ is the second parameter chosen by the Greedy algorithm. These two points are maximally distinct w.r.t. the error estimator.

(ii) Create a denser Training Set:
Define a new training sample set $\tilde{\mathcal{P}}_{B_l}^{\text{train}} \subset \mathcal{V}_{B_l}$ which is twice as large as the original training set.

(iii) Split the Training Set:
Split $\tilde{\mathcal{P}}_{B_l}^{\text{train}}$ into two sets for the subdomains $\mathcal{V}_{(B_l,0)}$ and $\mathcal{V}_{(B_l,1)}$ based on proximity:

$$\begin{aligned} &\boldsymbol{\mu} \in \mathcal{P}_{(B_l,0)}^{\text{train}} \quad \text{if} \quad d_{(B_l,0)}(\boldsymbol{\mu}) \leq d_{(B_l,1)}(\boldsymbol{\mu}), \\ &\boldsymbol{\mu} \in \mathcal{P}_{(B_l,1)}^{\text{train}} \quad \text{otherwise.} \end{aligned}$$

5. Split the current branch into two new branches:

$$B_{l+1}^{\text{left}} := (B_l, 0), \qquad B_{l+1}^{\text{right}} := (B_l, 1),$$

Set $l \leftarrow l+1$ and apply the procedure recursively first on the left branch B_{l+1}^{left}, then on the right branch B_{l+1}^{right}.

Algorithm 7: hGreedy$(\mathcal{P}_{B_l}, \hat{\boldsymbol{\mu}}_{B_l}, \epsilon_{\text{tol}}^1, \bar{N})$

$\mathbb{V}_{\bar{N},B_l}, \mathcal{M}_{\bar{N},B_l} \leftarrow$ Greedy$(\mathcal{P}_{B_l}, \hat{\boldsymbol{\mu}}_{B_l}, 0, \bar{N})$
$\epsilon_{\bar{N},B_l} \leftarrow \max_{\boldsymbol{\mu} \in \mathcal{P}_{B_l}^{\text{train}}} \Delta_{\bar{N}}(\boldsymbol{\mu})$
if $\epsilon_{\bar{N},B_l} < \epsilon_{tol}^1$ **then**
  $\hat{\boldsymbol{\mu}}_{\bar{N},(1,i_2,\ldots,i_l,0)} = \text{NaN}$
  $\hat{\boldsymbol{\mu}}_{\bar{N},(1,i_2,\ldots,i_l,1)} = \text{NaN}$
else
  $\hat{\boldsymbol{\mu}}_{(B_l,0)} := \hat{\boldsymbol{\mu}}_{B_l}$
  $\hat{\boldsymbol{\mu}}_{(B_l,1)} := \boldsymbol{\mu}_{2,B_l}$
  Define $\tilde{\mathcal{P}}_{B_l}^{\text{train}} \subset \mathcal{V}_{B_l}$
  Split into $\mathcal{P}_{(B_l,0)}^{\text{train}} \subset \mathcal{V}_{B_l,0}$ and $\mathcal{P}_{(B_l,1)}^{\text{train}} \subset \mathcal{V}_{B_l,1}$
  hGreedy$(\mathcal{P}_{(B_l,0)}^{\text{train}}, \hat{\boldsymbol{\mu}}_{(B_l,0)}, \epsilon_{\text{tol}}^1, \bar{N})$
  hGreedy$(\mathcal{P}_{(B_l,1)}^{\text{train}}, \hat{\boldsymbol{\mu}}_{(B_l,1)}, \epsilon_{\text{tol}}^1, \bar{N})$
end

5 Solving high fidelity problems

In this section, we investgate some test problems that are semilinear and elliptic. All problems depend on a diffusion parameter $\boldsymbol{\mu} \in (0, 1]$. As $\boldsymbol{\mu}$ decreases, the influence of the nonlinear term increases and solving the problem gets more challenging. To overcome these difficulites, different initial guesses and damping strategies for the Newton method are tested and compared.

5.1 Introduction of the test problems

We now introduce the considered test problems. For simplicity we restrict our analysis to one spatial dimension. The domain $\Omega = [0, 1]$ is discretized by dividing it into N_h subintervals, where h denotes the length of a single interval. This results in a system of dimenstion $N_h + 1$.

1. Method of manufactured solutions (mmsF) for elliptic Fisher equation: To validate the correctness of a numerical solver, we employ the Method of manufactured solutions. We assume an exact solution

$$u^*(x) = a \cdot \sin(2\pi b x) \tag{5.1}$$

which is inserted into the elliptic Fisher equation

$$F_{\boldsymbol{\mu}}(u) := -\nabla \cdot (\boldsymbol{\mu} \nabla u) - u(1 - u) = 0 \quad \text{in } \Omega. \tag{5.2}$$

This yields a residual forcing term

$$f_{\boldsymbol{\mu}} := F_{\boldsymbol{\mu}}(u^*). \tag{5.3}$$

The forcing term can be computed using PySim. The resulting term $f_{\boldsymbol{\mu}}$ is then used as a forcing term in a modified version of the equation:

$$-\nabla \cdot (\boldsymbol{\mu} \nabla u) - u(1 - u) - f_{\boldsymbol{\mu}} = 0 \quad \text{in } \Omega. \tag{5.4}$$

The boundary conditions are derived by the exact solution u^*:

$$u|_{\partial\Omega} = u^* \quad \text{on } \partial\Omega. \tag{5.5}$$

2. Elliptic Fisher equation (F): We also consider the unmodified elliptic Fisher equation

$$-\nabla \cdot (\boldsymbol{\mu} \nabla u) - u(1 - u) = 0 \quad \text{in } \Omega. \tag{5.6}$$

with Dirichlet boundary conditions

$$u_D(x) = \begin{cases} -0.1, & x = 0 \\ 0.4, & x = 1 \end{cases}.$$

3. Semilinear Poisson equation (slP): To complement the previous two problems, which are not well-posed in the sense that uniqueness of the solution is not guaranteed, we also consider a well-posed problem

$$-\nabla \cdot (\boldsymbol{\mu} \nabla u) + u^3 = 0 \quad \text{in } \Omega. \tag{5.7}$$

with Dirichlet boundary conditions

$$u_D(x) = \begin{cases} -0.1, & x = 0 \\ 0.4, & x = 1 \end{cases}.$$

For a proof for well-posedness we refer to chapter 11.1.2 in [14].

5.2 Initial Guess Strategies

To solve the test problems introduced in the previous section, we apply Newton's method as described in section 3.3. The choice of the inital guess $u^{(0)}$ can significantly influence the convergence behavior. In this section, we introduce four different strategies for constructing initial guesses.

1. Poisson-based Initial Guess (_P): The initial guess is obtained by solving the Poisson equation

$$-\nabla(\boldsymbol{\mu} \cdot \nabla u) = f \quad \text{in } \Omega, \quad u = u_D \quad \text{on } \partial\Omega.$$

2. Linearized Equation Initial Guess (_LP): Here, the initial guess is based on a simplified linearization of the original problem. Therfore, we solve:

$$-\nabla(\boldsymbol{\mu} \cdot \nabla u) - u = f \quad \text{in } \Omega, \quad u = u_D \quad \text{on } \partial\Omega.$$

3. Zero Initial Guess (_0): In this approach, the initial guess is chosen as a constant zero function:

$$u^{(0)}(x) = \begin{cases} 0, & x \in \Omega \\ u_D(x), & x \in \partial\Omega \end{cases}.$$

4. Constant Initial Guess with Value 0.5 (_0.5): This strategy is motivated by an analysis of the Fréchet derivative for the problems **mmsF** and **F**, since this derivative leads to a linear variational problem, which is solved in every Newton iteration:

$$a(\partial u, v; \boldsymbol{\mu}) := \langle \boldsymbol{\mu} \nabla \partial u, \nabla v \rangle_{L^2(\Omega)} + \langle \underbrace{-(1-2u)}_{=c(x)} \partial u, v \rangle_{L^2(\Omega)} = \langle f, v \rangle_{L^2(\Omega)} \quad \forall v \in H_0^1(\Omega).$$

The form $a(\partial u(k), v; \boldsymbol{\mu})$ is bilinear, symmetric, bounded and coercive according to Lemma 4.2.3 in [20] if $c(x) \geq 0$ for a.e. $x \in \Omega$, that is $u(x) \geq 0.5$ for a.e. $x \in \Omega$. Therfore, in that specific case, wellposedness can be assured by the Lax-Milgram theorem (Theorem 4.2.6 in [20]). While this condition cannot be guaranteed throughout all Newton iterations (since u changes during the process), we can at least ensure well-posedness in the first iteration by selecting

$$u^{(0)}(x) = \begin{cases} 0.5, & x \in \Omega \\ u_D(x), & x \in \partial\Omega \end{cases}.$$

5.3 Problem Classes for Test Problems using FEniCS

For each test problem introduced in section 5.1 we define a custom `NonlinearProblem` class:

- `Fisher` - Elliptic Fisher equation (**F**)
- `Fisher_mms` - Method of manufactured solutions for Fisher equation (**mmsF**)
- `SemilinearPoisson` - Semilinear Poisson equation (**slP**)

The full source code and documentation for these classes can be found in `rb_semilinear.sl_problems` in [15]. These classes are based on FEniCS's `NonlinearProblem` class. This means they can be solved using a `MyNewtonSolver` object as described in section 3.5. In contrast to the basic example `MyTestProblem` class introduced in section 3.4, these classes require additional input for initialization. Specifically, they take three arguments:

- `N_h`: The number of intervals used for spatial discretization.
- `solver`: An instance of `MyNewtonSolver`.
- `initGuess_strategy`: A string determining how the initial guess should be set.

These inputs are stored as class attributes. An insight to the constructor of such a class is:

```
    def __init__(N_h:int,solver:MyNewtonSolver,initGuess_strategy:str) :
      NonlinearProblem.__init__(self)
      self.N_h = N_h
      self.solver = solver
      self.initGuess_strategy = initGuess_strategy
      ...
```

This structure allows the problem to be solved more conveniently, since the solver and initial guess strategy are already prepared inside the class. In addition to the methods introduced with the `MyTestProblem` class, these custom problem classes include a method called `set_initGuess()`. This method sets the solution function `self.u` based on the selected strategy `self.initGuess_strategy`. The available initial guess strategies (see 5.2) are:

- "0.5" - Constant Initial Guess with Value 0.5 (**_0.5**)
- "0" - Zero Initial Guess (**_0**)
- "P" - Poisson-based Initial Guess (**_P**)
- "LP" - Linearized Equation Initial Guess (**_LP**)
- None - No automatic initialization (can be set manually before solving)

Additionally, these classes provide a method `solve(mu)`, which internally

- Sets the problem parameter `self.mu`,
- Sets the initial guess in `self.u`, and
- Solves the problem using `self.solver`.

This makes solving a problem very straightforward. For example:

```
solver = MyNewton_nleqerr(tol=1/1000**2, maxit=100, report=True)
problem = Fisher(N_h=1000, solver=solver, initGuess_strategy="0.5")
problem.solve(mu=0.0005)
plot(problem.u)
plt.show()
```

5.4 Numerical Experiments

We begin by analyzing the inherent difficulty of the test problems introduced in section 5.1. Next, we explore the performance of the ordinary Newton solver using different initial guesses. Then, after validating the solver, we evaluate several damping strategies. All results discussed in the following can be reproduced using the Python script `test_ns.py` in [15].

In the follwing, OUTMAX indicates failures caused by reaching the maximum of permitted number of Newton iterations, which was set to 100. Convergence was determined by a tolerance for the residual defined as TOL=h^2, where h is the mesh size. As minimal damping factor we set $\lambda_{\min} = 10^{-8}$, if a damped Newton method is used.

Problem Difficulty To assess the inherent difficulty of the test problems, we applied an exact ordinary Newton method. As worst-case scenario, we used the simplest – and likely most unstable – initial guess: a constant zero function $u^{(0)} = 0$. Each problem is solved multiple times for decreasing values of the diffusion constant $\boldsymbol{\mu}$ (i.e. increasing nonlinearity). This setup shows the sensitivity of each problem to nonlinearity. The results in Table 1 show that the well-posed problem **slP_0** is solved reliably and efficiently for all tested values of the diffusion parameter $\boldsymbol{\mu}$, despite the poor initial guess. In contrast, the problems **F_0** and **mmsF_0** converge only for sufficiently large parameter values. As $\boldsymbol{\mu}$ decreases, the influence of the nonlinear term increases and the ordinary Newton method fails. In these cases the simple choice for the initial guess is not sufficient.

$\boldsymbol{\mu}$	**F_0**	**mmsF_0**	**slP_0**
1	2	3	2
0.5	3	3	2
0.1	10	16	3
0.05	4	6	3
0.01	OUTMAX	10	3
0.005	OUTMAX	11	4
0.001	OUTMAX	OUTMAX	5
0.0005	OUTMAX	OUTMAX	5
0.0001	OUTMAX	OUTMAX	6

Table 1: Ordinary Newton method ($N_h = 1000$)

Test Different Initial Guesses To improve the convergence for the more challenging, ill-posed test problems even when the parameter $\boldsymbol{\mu}$ takes smaller values, we now investgate how different inital guesses affect the performance of the Newton method. Table 2 shows a comparison of the different initialization strategies introduced in section 5.2 applied to the test problems **F** and **mmsF**. The results show that the strategies **P** and **0.5** extend the range of $\boldsymbol{\mu}$ values for which the Newton Method converges, but the **LP** strategy does not improve the convergence. Among all options, the simple constant guess **0.5** does perform very good for both test problems.

$\boldsymbol{\mu}$	**F_0.5**	**F_LP**	**F_P**	**mmsF_0.5**	**mmsF_LP**	**mmsF_P**
1	3	1	2	3	2	2
0.5	3	2	2	4	2	2
0.1	3	9	4	6	15	5
0.05	3	3	5	5	5	4
0.01	6	OUTMAX	5	7	9	6
0.005	7	OUTMAX	8	8	10	7
0.001	10	OUTMAX	OUTMAX	11	OUTMAX	9
0.0005	11	OUTMAX	14	12	OUTMAX	11
0.0001	13	OUTMAX	OUTMAX	15	OUTMAX	14
0.00005	14	OUTMAX	OUTMAX	16	OUTMAX	15
0.00001	16	OUTMAX	OUTMAX	19	OUTMAX	18

Table 2: Comparison of different initial guesses ($N_h = 1000$)

However, when looking at the computed solutions, we observe, that the Newton method does converge to different solutions of the same problem, depending on the used inital guess. Figure 1 shows this behavior for test problem **F**, showing the solutions obtained with the strategies **0.5** (upper plot) and **P** (lower plot). It appears, that for parameter values greater than 0.1, the

Newton method converges to the same solution regardless of the chosen initial guess. Below this value, however, different strategies may lead to distinct solutions. This suggests that multiple solutions exist for $\boldsymbol{\mu} \leq 0.1$ while the problem may become uniquely solvable for larger $\boldsymbol{\mu}$.

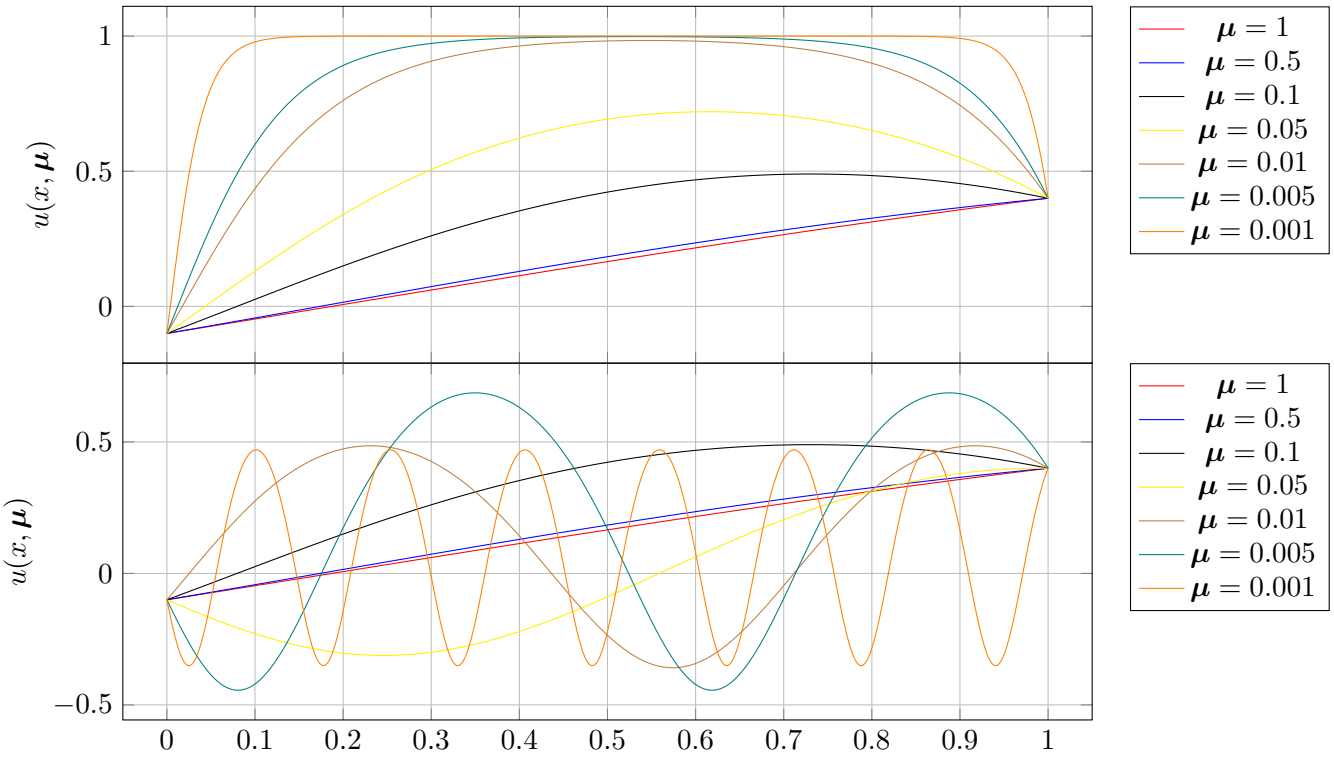


Figure 1: Solutions for $\mathbf{F}$ with $N_h = 1000$, varying diffusion parameters $\boldsymbol{\mu}$, initial guess strategy **0.5** (upper plot) and $\mathbf{P}$ (lower plot).

The computed solutions of the test problem **mmsF** do not depend on the used strategy for the initial guess. Therefore, Figure 2 only shows the results obtained using the **0.5** strategy, as the solutions for $\mathbf{P}$ are the same. But even here we find multiple solutions for $\boldsymbol{\mu} \leq 0.5$. Although we have an exact solution u^*, the computed solutions differ from u^* for lower values of $\boldsymbol{\mu}$.

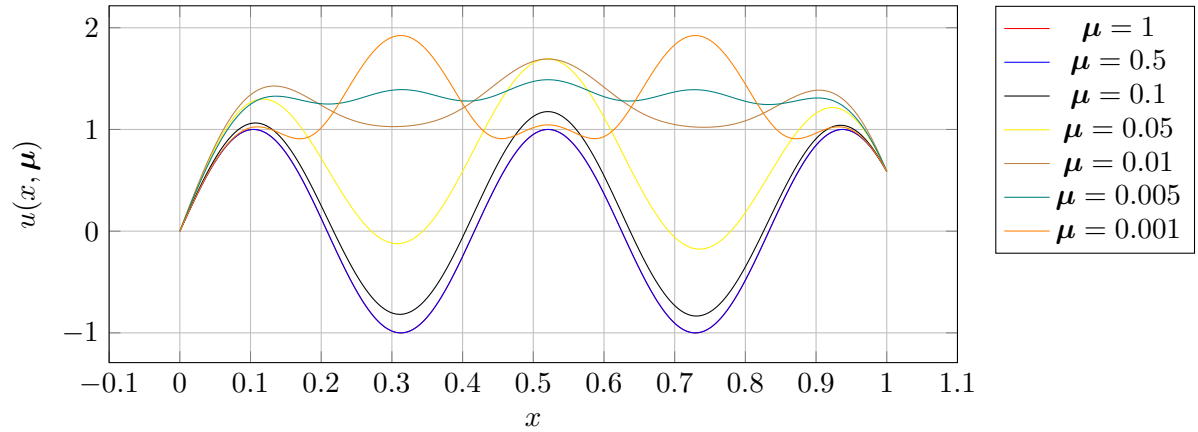


Figure 2: Solutions for **mmsF** ($N_h = 1000$), varying diffusion parameters $\boldsymbol{\mu}$ and strategy **0.5**.

Solver Validation To validate the Newton solver, we applied it to the problem **mmsF_0** with a parameter value of $\boldsymbol{\mu} = 0.5$ – a case where the method is known to converge to the assumed

exact solution u^*. We solved for varying mesh resolutions with

$$N_h \in \{50, 100, 200, 500, 1000, 2000, 3000, 5000, 7000\}$$

and calculated the corresponding error norms between the exact solution u^* and the computed solution $u_h(\boldsymbol{\mu})$. The results are presented in figure 3. The L^2-norm and the infimum norm of the error decrease quadratically, while the H^1-norm decreases linearly with h. The observed rates of convergence are consistent with the possible convergence behavior (see Theorem 4.5.4 in [20]).

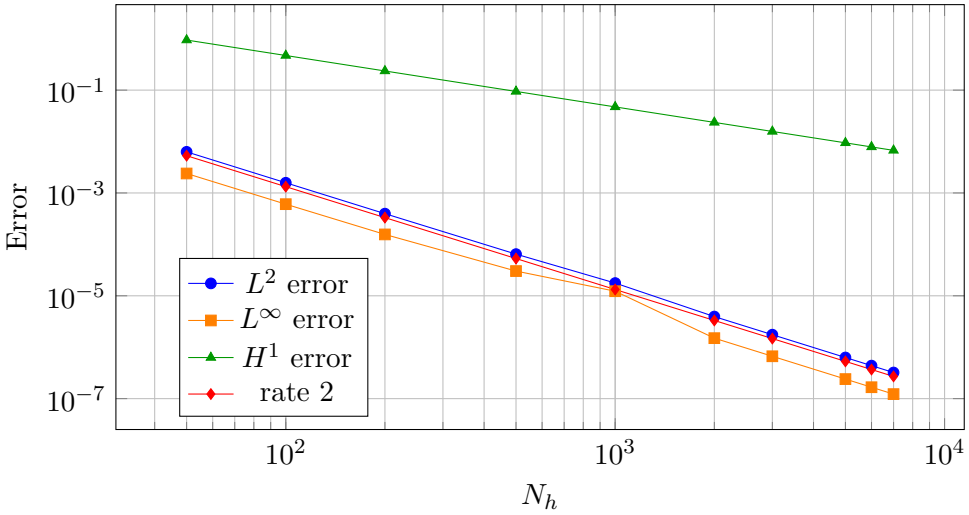


Figure 3: Errors for solutions of **mmsF_0** with $D = 0.5$

From this we can conclude, that our ordinary Newton method works fine in principle, but for problems with high nonlinearity, we have to find a good damping strategy and initial guesses.

Testing Different Damping Strategies To potentially improve convergence for solutions of test problem **F** which are obtained using the initial guess strategy **P** we evaluate several damping strategies (see section 3.2) for the Newton method.

$\boldsymbol{\mu}$	**simpleDamp**	**adapDamp**	**nleqerr**
1	2	2	2
0.5	2	2	2
0.1	3	3	4
0.05	4	4	4
0.01	5	6	5
0.005	6	6	6
0.001	9	8	9
0.0005	14	λ-FAIL	15
0.0001	λ-FAIL	λ-FAIL	37
0.00005	λ-FAIL	23	47
0.00001	λ-FAIL	λ-FAIL	DIV

Table 3: Comparison of different damping strategies for test problem **F_P**$(N_h = 1000)$

The results are shown in Table 3. The damping strategy **nleqerr**has the most robust convergence

behavior. Here DIV indicates that the **nleqerr**-algorithm terminated due to divergence of the residual. It significantly extends the range of paramter values $\boldsymbol{\mu}$ for which the Newton method successfully converges.

Figure 4 shows the solutions for the test problem **F_P** obtained using the damping strategy **nleqerr**. The resulting solutions are periodic functions. As the nonlinearity of the problem increases—that is, as the diffusion coefficient $\boldsymbol{\mu}$ decreases—the frequencies of these periodic functions increase.

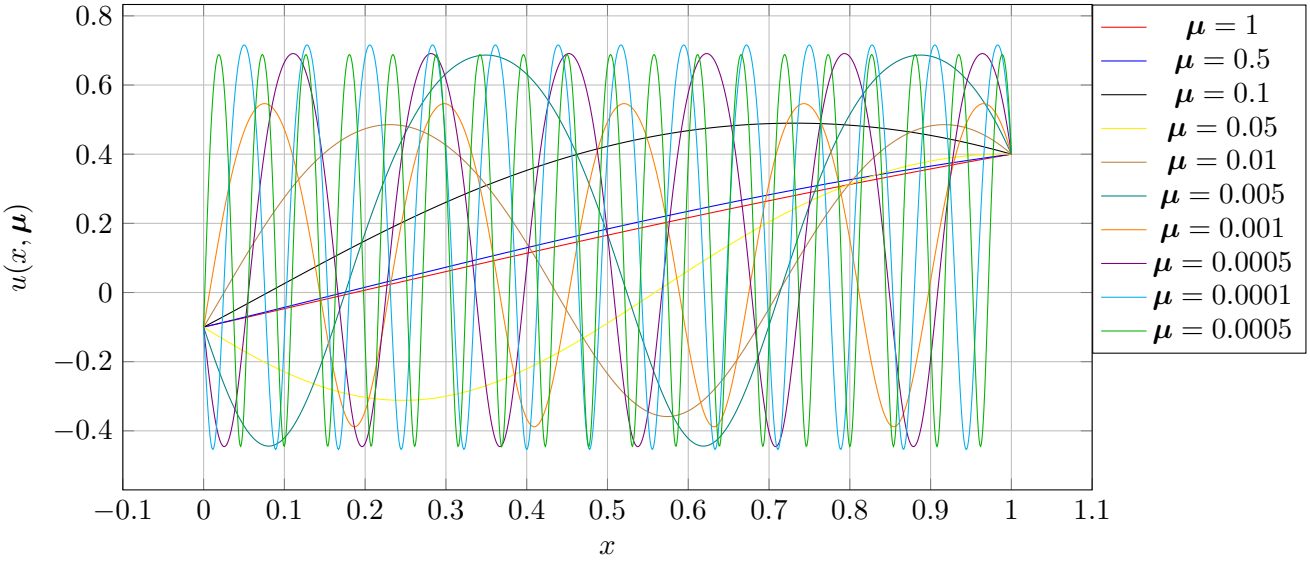


Figure 4: solutions for **F_P** obtained with **nleqerr** ($N_h = 1000$)

In summary, we conclude that the damping strategy **nleqerr** yields the best performance. Although the test problems can often be solved without damping when using the initial guess strategy **0.5**, damping becomes essential when aiming to converge to the periodic solutions of problem **F**. For this reason, we will use the **nleqerr** damping strategy in the following, in combination with either the **0.5** or **P** initial guess strategies.

Remark 5.1
We also tested a homotopy-inspired strategy to improve convergence. That is, for solving the problem with a parameter value $\boldsymbol{\mu}$ we used the solution corresponding to a nearby parameter value as the initial guess. For example, we first solved the problem with $\boldsymbol{\mu} = 1$ and then used that solution as the initial guess for a slightly smaller parameter value and so on. This led to the same convergence results as the previously showed **0.5** strategy. However, when we used solutions from the periodic branch as initial guesses, this did not improve convergence.

6 Application of the Reduced Basis Method

Since all high-fidelity test problems discussed in Section 5 converge with values for the diffusion constants ranging from 1 down to $2e^{-5}$, we define the parameter domain as follows

$$\mathcal{P} := [\boldsymbol{\mu}_{\min}, \boldsymbol{\mu}_{\max}] := [2e^{-5}, 1] \in \mathbb{R}^p, \quad p = 1.$$

Furthermore we will always use a high-fidelity discretization of $N_h = 1000$ intervals. For solving the high-fidelity problem we set the residual tolerance to $\text{TOL} = h^2 = 10^{-6}$.

In this section, we first introduce the discretization strategies used for the parameter domain. Furthermore, a brief overview of the Python package built with FEniCS, which was implemented to explore RB methods for semilinear PDEs, is provided. We then analyze the singular values obtained through POD for the test problems, which provides insights into their reducibility. Next, we construct reduced problems for our test problems using the Greedy algorithm and evaluate their performance. Finally we investigate the performance of the h-type Greedy algorithm.

6.1 Problem class for reduced problems in FEniCS

To ensure consistency, we implemented a class called `MyRedNonlinearProblem`, which represents a reduced problem and follows the same structure as the class used for the high-fidelity model. Therfore, it can also be solved using a `NewtonSolver` via the method `solve(mu)`. The class is implemented in the Python module `rb_semilinear.rb_nl_problem`. The full source code and its documentation are provided in [15].

In this section, we will discuss only some differences between the `MyRedNonlinearProblem` and the high fidelity classes. To do so, we present a simplified implementation of the reduced problem class and explain the key differences.

```
class MyRedNonlinearProblem(NonlinearProblem):
 def __init__(self, hnl_problem:MySemilinearProblem, V:np.ndarray,
        solver:PETScSNESSolver|NewtonSolver, initGuess_strategy:str):
     NonlinearProblem.__init__(self)

     # --- corresponding high-fidelity problem  --- #
     self.hnl_problem = hnl_problem
     # --- reduced basis --- #
     self.V = V
     self.V0 = V.copy(); self.V0[0]=0; self.V0[-1]=0
     # --- reduced basis dimension --- #
     self.N = V.shape[1]
     # --- unknown reduced basis coefficents --- #
     self.u_rbc = PETScVector(PETSc.Vec().createSeq(self.N))
     # solving parameter
     self.solver = solver
     self.initGuess_strategy = initGuess_strategy

 def comp_u_N(self, u_rbc:PETScVector):
     u_N = Function(self.hnl_problem.V_h)
     u_N.vector().set_local(self.V @ u_rbc.get_local())
     return u_N

 def J(self, A:PETScMatrix, x:PETScVector):
     # --- reduced Function u_N --- #
     u_N = self.comp_u_N(u_rbc, self.hnl_problem.bcs)
```

```
        # --- high fidelity Jacobian matrix at u_N --- #
        du = TrialFunction(self.hnl_problem.V_h)
        J_h = assemble(self.hnl_problem._a(u_N))
        self.hnl_problem.bc0.apply(J_h)
        # --- project J_h onto reduced space --- #
        J_N = self.V0.T @ J_h.array() @ self.V0
        # --- assign J_N to A --- #
        J_petsc = PETSc.Mat().createDense(J_N.shape, array=J_N)
        A.mat().zeroEntries()
        J_petsc.copy(result=A.mat())

    def F(self, b:PETScVector, x:PETScVector):
        # --- reduced Function u_N --- #
        u_N = self.comp_u_N(x, self.hnl_problem.bcs)
        # --- high fidelity residual vector at u_N --- #
        G_h = assemble(self.hnl_problem._l(u_N))
        self.hnl_problem.bc0.apply(G_h)
        # --- project G_h into V_N --- #
        G_N = self.V0.T @ G_h.get_local()
        # --- assign G_N to b --- #
        b.set_local(G_N)

    def set_mu(self, mu:float):
        ...
    def set_initGuess(self):
        ...
    def solve(mu:float):
        ...
```

Listing 2: Class for reduced basis problem

The initializer of this class takes four arguments:

- `hf_problem`: the corresponding high-fidelity problem.
- `V`: the discrete reduced basis, which represents $\mathbb{V}$.
- `solver`: An instance of `MyNewtonSolver`.
- `initGuess_strategy`: A string determining how the initial guess should be set.

These parameters are stored as class attributes. Furthermore, an adapted discrete reduced basis self.V0 is stored. In this reduced basis, the DOFs corresponding to the Dirichlet boundary conditions are set to zero. This adapted basis is only used to assemble the reduced linear system of equations to ensure that the boundary conditions are satisfied (the boundary values of the newton increment have to be zero). Unlike the high-fidelity problem, where the current solution is stored as a `Function` object (`self.u`), this class stores the RB coefficients of the current reduced solution as a `PETScVector` in `self.u_rbc`. This vector represents the RB coefficients $\mathbf{u}_N$ (cf. (4.7)) of the reduced basis solution.

To compute a reduced solution in the high-fidelity function space from RB coefficents, the class provides a method `comp_u_N(u_rbc)`. This method takes the RB coefficents `u_rbc` and returns a `Function` object in the high-fidelity function space `self.hnf_problem.V_h`.

The reduced residual vector $\mathbf{G}_N$ and the reduced Jacobian matrix $\mathbb{J}_N$ from equation (4.14) are computed in the methods `F` and `J`. These methods first reconstruct the high-fidelity solution from the given RB coefficients `x`. Then, they assemble the high-fidelity residual vector `G_h` and Jacobian matrix `J_h`, and finally project these onto the reduced basis space to get the reduced versions.

Note that for a efficient computation, the implementation of methods `F` and `J` must be replaced by a summation of precomputed and stored vectors and matrices (cf. (4.15) and (4.16)).

Additionally, we implemented all the algorithms introduced in this section in the package `rb_semilinear`. In the following we provide an overview of the most important functions. For further details, we refer to the documentation in [15].

- `get_P(P_range,P_strategy,P_ns)` - generates the parameter set $\mathcal{P}_{\text{strategy\_ns}}(\text{range})$.
- `comp_S(problem,P_train)` - computes the snapshot matrix.
- `com_POD(S, N_max, tol, folder)` - computes the reduced basis via POD, stores it in a csv-file in the directory `folder` and plots singular values.
- `comp_greedyRB(S,M,P_train, mu_0,N_max,greedy_tol, folder)` - computes the reduced basis via the standard greedy algorithm and stores it in a csv file in the directory `folder`.
- `greedy_htype(hf_problem, P_train, eps_tol1,N_bar, folder)` - implements the hierarchical greedy approach. The resulting reduced basis are stored in csv files in the directory `folder`.

Now, a reduced basis problem can be constructed and solved using a Newton solver class from section 3.5 and a test problem class from 5.3:

```
solver = MyNewton_nleqerr(tol=1/1000**2, maxit=100, report=True)
hf_problem = Fisher(N_h=1000,solver=solver,initGuess_strategy="0.5")
P_train = get_P(P_range=[0.001,1], P_strategy="log", P_ns=50)
S = comp_S(hf_problem, P_train)
V = comp_POD(S, N_max, tol)
rb_problem = MyRedNonlinearProblem(hf_problem, V, solver, "0.5")
rb_problem.solve(mu=0.02)
```

6.2 Training Set Sampling Strategies

The choice of the training sample parameters play a critical role, as the performance of the reduced basis method depends on the quality of this training set. On one hand, the training set should be as large as possible to represent the parameter space $\mathcal{P}$ adequately. But on the other hand, for efficiency in the offline phase, it is desirable to keep it is as small as possible. Especially since we do not have an a posteriori error estimator for the Greedy algorithm which computation is very cheap, but we have to compute the high-fidelity solutions for every parameter in the training set, the latter requirement may be especially important.

In the following sections we consider two simple methods to discretize the parameter space $\mathcal{P}$ yielding parameter sets with ns sample parameters.

- **linear Discretization (lin_ns):**
  The parameter space is divided uniformly on a linear scale, using an equidistant step size $\mathbf{h} = \frac{\boldsymbol{\mu}_{\max} - \boldsymbol{\mu}_{\min}}{ns+1} \in \mathbb{R}^p$. This yields the training set

$$\mathcal{P}_{uni\_ns}(\boldsymbol{\mu}_{\min}, \boldsymbol{\mu}_{\max}) := \{\boldsymbol{\mu}_{\min} + i \cdot \mathbf{h}\}_{i=0}^{ns-1}.$$

- **logarithmic Discretization (log_ns):**
  Since the problem becomes more nonlinear as the diffusion coefficient decreases, the reduced model typically exhibits lower accuracy for small values of μ. To address this, we employ a uniformly discretization on a logarithmic scale, which places more sample points in regions where the problem becomes more nonlinear.

$$\mathcal{P}_{log\_ns}(\boldsymbol{\mu}_{\min}, \boldsymbol{\mu}_{\max}) := \left\{ 10^{(e+i\cdot h)} | e = \log(\boldsymbol{\mu}_{\min}), h = \frac{\log(\boldsymbol{\mu}_{\max}) - \log(\boldsymbol{\mu}_{\min})}{ns+1} \right\}_{i=0}^{ns-1}.$$

We define the training parameter set used for constructing the reduced basis as

$$\mathcal{P}_{m\_ns}^{\text{train}} := \mathcal{P}_{m\_ns}(2e^{-5}, 1) \tag{6.1}$$

where "m" stands for **log** or **uni** discretization. For testing the reduced problem we use a distinct parameter set

$$\mathcal{P}_{m\_ns}^{\text{test}} := \mathcal{P}_{m\_ns}(2.09e^{-5}, 0.89), \tag{6.2}$$

ensuring that the training and testing parameters are not the same. Unless otherwise specified, these definitions for $\mathcal{P}_{m\_ns}^{\text{train}}$ and $\mathcal{P}_{m\_ns}^{\text{test}}$ are consistently used in the following.

For testing, we excluded the parameter range from 1 to 0.89, as the solutions in this interval vary only very little and all models performed well there. Instead, we focused on lower parameter values, where the problem becomes more challenging.

6.3 Analyzing Singular Values

We begin by analyzing the singular values obtained through POD for our test problems. For this we have to compute the high-fidelity solutions $u_h(\boldsymbol{\mu}_i)$ for all parameters $\boldsymbol{\mu}_i$ in a training set $\mathcal{P}_{\text{m\_ns}}^{\text{train}}$ where "m" stands for **log** or **lin** discretization of the parameter domain $\mathcal{P}$.

These high fidelity solutions form the snapshots set

$$S = \{u_h(\boldsymbol{\mu}_i)\}_{i=1}^{ns}.$$

We arange the degrees of freedom of these snapshots into the snapshot matrix

$$\mathbb{S} = [\mathbf{u}_{h,1}|....|\mathbf{u}_{h,ns}] \in \mathbb{R}^{N_h \times ns}.$$

From this matrix, we compute a discrete POD-basis $\mathbb{V} \in \mathbb{R}^{N_h \times ns}$ with ns basis vectors and corresponding singular values σ_i, $i = 1, ..., ns$ as described in Section 4.6.2, algorithm 6.

As described in Remark 4.3 , the decay behavior of these singular values give information whether a problem is well-reduceable. We compare the two sampling strategies—**log** and **lin**— to select the parameter sample set $\mathcal{P}_{m\_150}^{\text{train}}$.

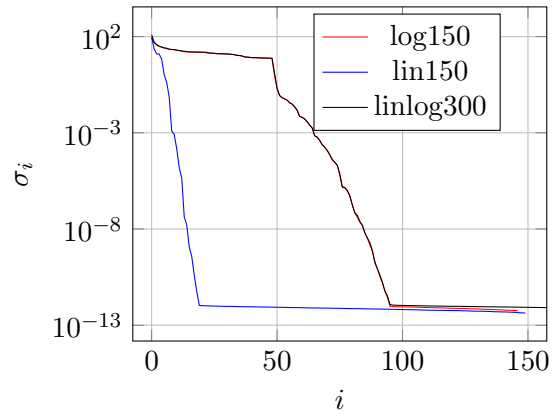
Figure 5 displays the resulting singular values for the different test problems using $ns = 150$ parameter samples. Each subplot shows the singular values for one test problem using either the **log** or **lin** sampling strategy. In general, the singular values exhibit a similar decay trend for all test cases with one exception: For **F_P** the singular values obained with the **log_150** sampling decay significantly slower.

Despite this, all problems show a sharp decay of the singular values at some point, suggesting that a low-dimensional reduced basis can capture most of the solution space. However, singular

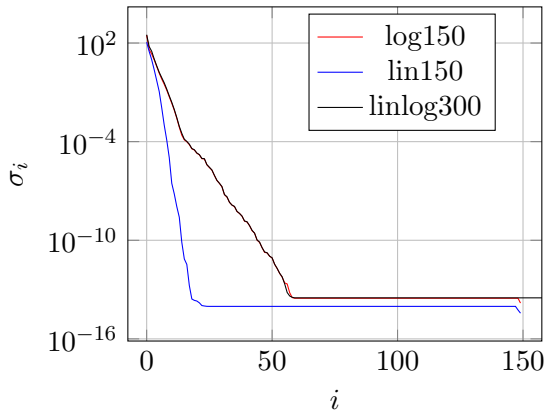
values from the **lin_150** strategy tend to decay faster overall, which could misleadingly suggest that the problem is more easily reducible. But a look at the **log_150** results shows that including more samples from parameter regions which lead to higher nonlinearities of the problem slows down the decay. Combining both training sets (linlog300) results in the same singular value decay as observed for the **log_150** set.

From this we can lead that the sampling strategy greatly influences the performance of the reduced problem, especially for problems with varying strong nonlinearity. It is a hint that it will be necessary to include more low parameter values in the parameter training set.

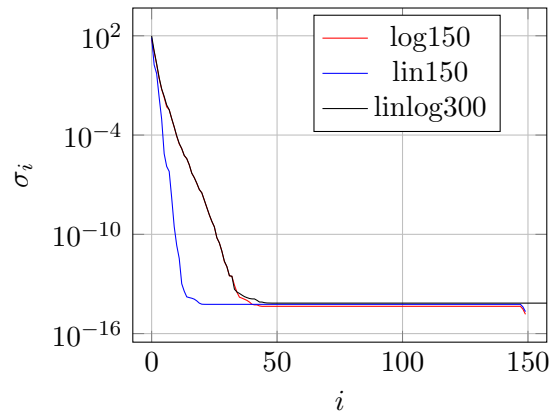
The difference between the sampling strategies **log_150** and **lin_150** is the smallest for the test problem **slP_0**. However, even in this well-posed case, including more low paramater values still slows down the decay of the singular values.



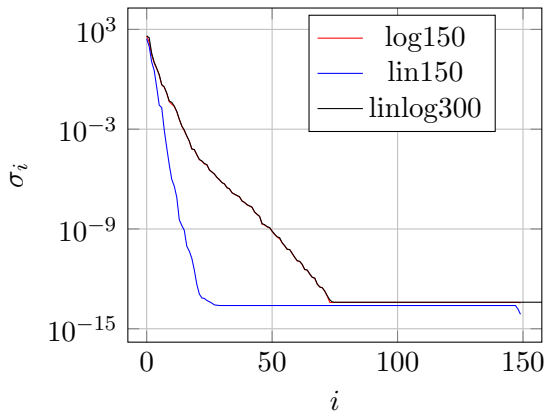
(a) Singular values σ_i for **F_P**



(b) Singular values σ_i for **F_0.5**



(c) Singular values σ_i for **slP_0**



(d) Singular values σ_i for **mms_0.5**

Figure 5: Singular values for test problems obtained through POD by using either **log_150** or **lin_150** as sampling strategy

6.4 Model Order Reduction with Greedy algorithm

In this section, we construct reduced models for our test problems using Greedy algorithm with a parameter training set

$$\mathcal{P}^{\text{train}}_{\text{m\_ns}} := \{\boldsymbol{\mu}_i\}_{i=0}^{ns}.$$

Here "m" stands for **log** or **lin** discretization.

Using the Greedy algorithm 4,we construct the discrete reduced basis $\mathbb{V} \in \mathbb{R}^{N_h \times ns}$. The Greeedy algorithm uses the projection error, defined in equation (4.18), as error indicator. We set the greedy tolerance $\epsilon_{\text{tol}} = 10h^2 = 10^{-5}$ and typically choose $N_{\max} = ns$. This ensures that the Greedy algorithm terminates because of reaching the desired tolerance, provided it is achievable within the specified maximum number of basis vectors.

Using the reduced basis obained from the Greedy algorithm, we formulate the reduced problem as described in equation (4.11). This reduced problem is solved for each parameter $\boldsymbol{\mu}$ in the test set $\mathcal{P}^{\text{test}}_{\text{m\_ns}}$ using Newton's method combined with the damping strategy **nleqerr** (refer to Section 3.2). We set the maximum number of Newton iterations to `maxit`= 100 and the residual tolerance to `RTOL`= $10h^2 = 10^{-5}$. For the initial guess in Newton's method, we use either the **0.5** or **P** strategy, as detailed in Section 5.2. This means, we select a high-fidelity function $u^{(0)} \in V_h$ according to the chosen initial guess strategy. We then reduce this function by calculating the RB coefficients $\mathbf{u}_N^{(0)} \in \mathbb{R}^N$ (see (4.7)) using

$$\mathbf{u}_N^{(0)} := \mathbb{V}^T \mathbf{u},$$

where $\mathbf{u}$ denotes the vector of degrees of freedom corresponding to $u^{(0)} \in V_h$. The reduced solution is denoted by $u_N(\boldsymbol{\mu}) \in V_N \subset V_h$.

To validate the reduced model, we compute the high-fidelity solution $u_h(\boldsymbol{\mu})$ for each test parameter $\boldsymbol{\mu}$ in a test set $\mathcal{P}^{\text{test}}_{\textbf{m\_ns}}$. We then project each high-fidelity solution onto the reduced space V_N and calculate the projection error:

$$e_P(\boldsymbol{\mu}) := \left\| u_h(\boldsymbol{\mu}) - P^{L^2}_{V_N} u_h(\boldsymbol{\mu}) \right\|_{L_2(\Omega)} \tag{6.3}$$

This projection error represents the best possible approximation error in the reduced space and shows how well V_N can represent the high-fidelity solution.

Additionally, we compare the reduced solution $u_N(\boldsymbol{\mu})$ with the corresponding high-fidelity solution $u_h(\boldsymbol{\mu})$ for each test parameter $\boldsymbol{\mu}$. This actual error is quantified by the L^2-norm:

$$e(\boldsymbol{\mu}) := \|u_N(\boldsymbol{\mu}) - u_h(\boldsymbol{\mu})\|_{L_2(\Omega)} \tag{6.4}$$

This shows how well the reduced problem performs in practice across the parameter space.

Since the results for the reduced test problems **F_0.5**, **mms_0.5**, and **slP_0** are very similar, we restrict the discussion in the following section to the first one. The corresponding plots for the other two problems, analogous to those shown for **F_0.5**, are provided in the appendix A.

The following plots illustrate the error norms defined in equations (6.3) and (6.4). In the upper plot, we always display the projection error $e_P(\boldsymbol{\mu})$ and the actual reduced model error $e(\boldsymbol{\mu})$ for each parameter $\boldsymbol{\mu}$ in the test set. The lower plot indicates the paramater values selected by the Greedy algorithm during the training phase.

It is important to note, that the bars in the upper plot represent error norms for the test parameters, while the points in the lower plot correspond to the training parameters. Although the these parameter sets differ, this might not be immediately apperent in the visualization.

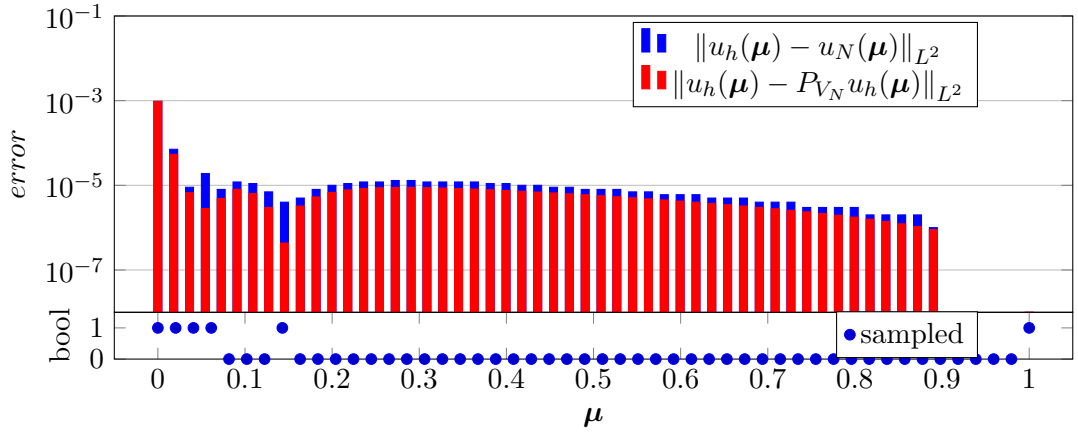
Reduced Problem for Test Problem F_0.5

We begin our analysis with the reduction of test problem **F_0.5**. Using the training parameter set $\mathcal{P}^{\text{train}}_{\textbf{lin\_50}}$, the Greedy algorithm, applied with the specified tolerance, constructs a reduced basis space V_N of dimension $N = 6$. The performance of this reduced model is then evaluated using two test parameter sets: $\mathcal{P}^{\text{test}}_{\textbf{lin\_50}}$ and $\mathcal{P}^{\text{test}}_{\textbf{log\_50}}$.

Figure 6a corresponds to the test set $\mathcal{P}^{\text{test}}_{\textbf{lin\_50}}$. The projection error remains below $1 \cdot 10^{-5}$ across nearly all test parameters. However, the error increases as the parameter values decreases, except in regions near the parameters selected by the Greedy algorithm. The error of computed reduced solution is very close to the best possible approximation error.

However, testing the reduced model, particulary for low parameter values, indicates that the reduced problem is not well-suited in this parameter area, as revealed in figure 6b. This figure shows the error norms obtained by using the parameter test set $\mathcal{P}^{\text{test}}_{\textbf{log\_50}}$. The best possible error norm increases for low parameter values reaching up to $4 \cdot 10^{-1}$. This shows, that the chosen training parameter set did not adequately represent the parameter range in this area. Nevertheless, Newton's method always converges nearly to the best possible solution.

(a) Test based on test parameter sampling **lin_50**.



(b) Test based on test parameter sampling **log_50**.

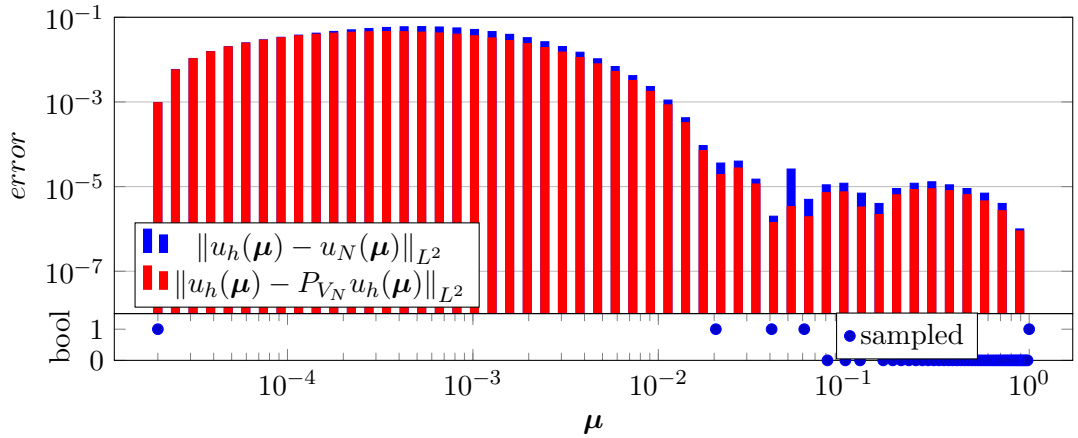


Figure 6: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_05**, trained with test parameter sampling **lin_50**.

To improve the reduced model's performance, particulary for low parameter values, we use the training parameter set $\mathcal{P}^{\text{train}}_{\textbf{log\_50}}$. For this we achieve a reduced space V_N of dimension $N = 14$.

Figure 7 shows the error norms obained using this test set. The results show that the reduced model indeed achieves lower best approximation errors, even for low parameter values. Moreover, the model also performs better for higher parameter values compared to the previous model that used the **lin_50** discretization. Clearly, the reduced problem performs best near the parameter values selected by the Greedy algorithm.

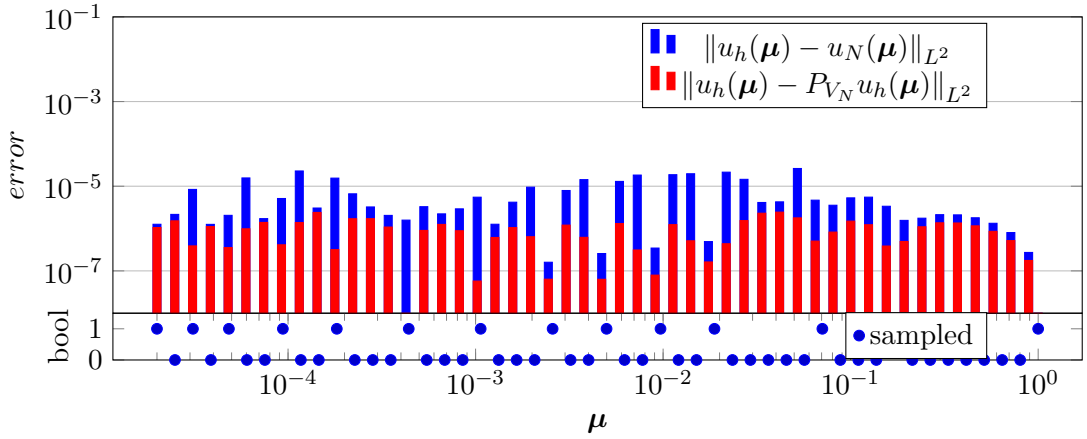


Figure 7: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_05**, trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**.

While this reduced problem space has a dimension of $N = 14$, which is significantly larger than in the previous case $(N = 6)$, it offers better approximation accuracy, particulary for low parameter values. However, when we constrain the maximum number of basis functions to $N_{\max} = 6$, matching the dimension of the earlier reduced basis, the resulting model still performs better than the one derived from the $\mathcal{P}^{\text{train}}_{\text{lin\_50}}$ set. This result can be seen in figure 8.

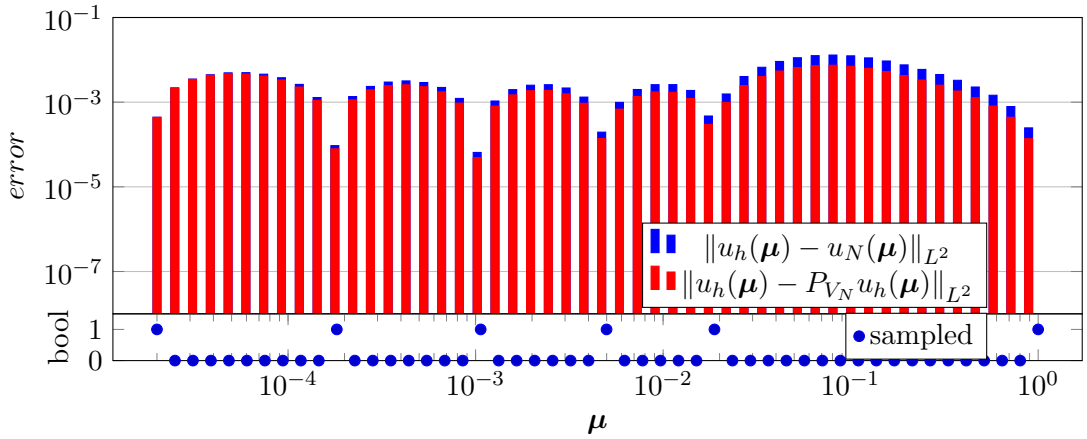


Figure 8: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_05** (N_max=6), trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**.

We observed that the greedy algorithm samples the parameter space nearly uniformly on a logarithmic scale. This suggests that a significantly smaller training set, such as **log_15**, may still capture the essential features of the solution manifold. As shown in Figure 9, using the reduced training set $\mathcal{P}^{\text{train}}_{\textbf{log\_15}}$ with only $ns = 15$ parameters yields comparable accuracy.

Im summary, the RB method combined with the Greedy algorithm performs very well for this test problem **F_0.5** (as well as for the problems **mmsF_0.5** and **slP_0**). Even with a very small training set of parameter values, it is possible to create a good reduced model. Further numerical experiments even showed that similar results can be achieved for parameter values down to $8 \cdot 10^{-8}$. However, as we will see in the next section, this strong performance does not hold for the solution branch of the Fisher equation that involves periodic functions.

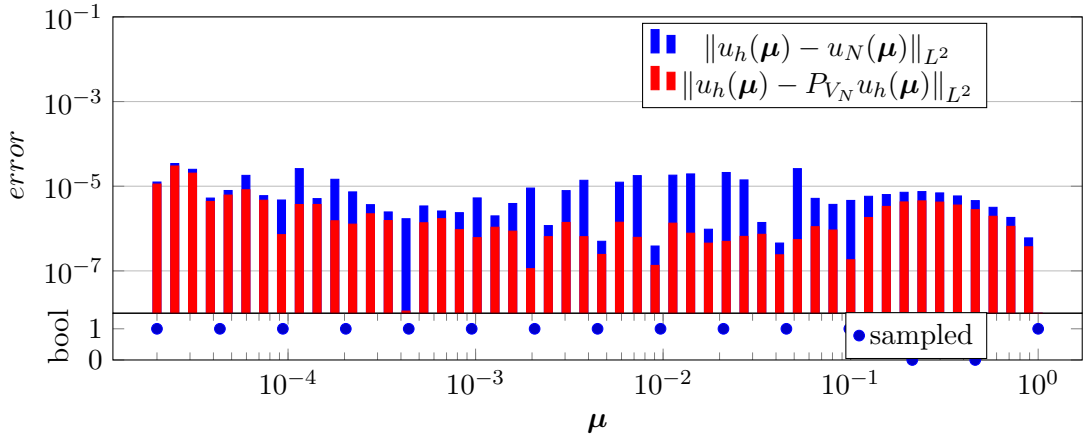
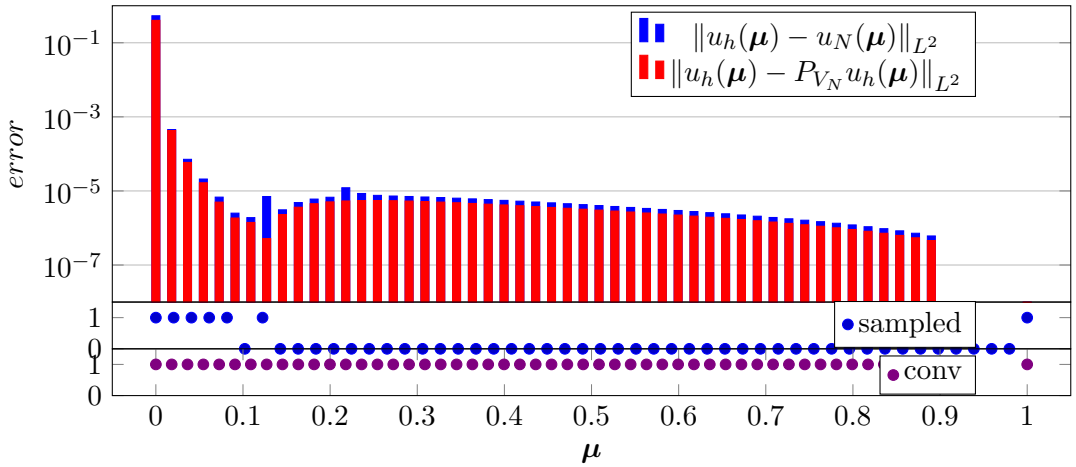


Figure 9: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_05**, trained with test parameter sampling **log_15**, test based on parameter sampling **log_50**.

Reduced Problem for Test Problem F_P

We now consider the reduction of test problem **F_P**. In this case, the Greedy algorithm for the training parameter set $\mathcal{P}^{\text{train}}_{\textbf{lin\_50}}$ results in a reduced basis space V_N of dimension $N = 7$.

(a) Test based on test parameter sampling **lin_50**.



(b) Test based on test parameter sampling **log_50**.

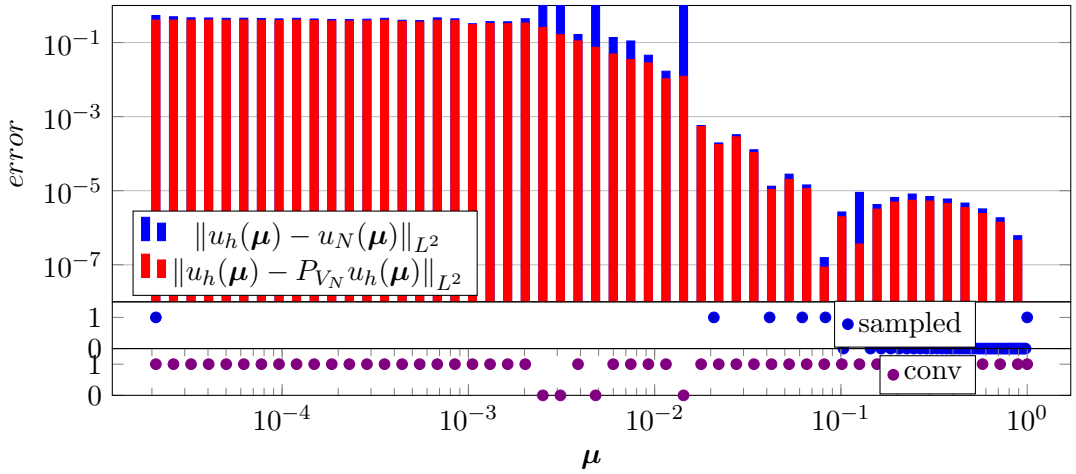


Figure 10: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_P**, trained with test parameter sampling **lin_50**.

Again, for the test parameter sampling strategy **lin_50**, we observe relatively good results, although the error norms tend to increase as the parameter values decrease. In particular, the performance is very bad for the smallest parameter values. In contrast, for the test sampling strategy **log_50**, the resulting error plot shows significantly worse performance, as expected. These observations can be seen in figure 10. Note, that the lowest plot indicates, whether the Newton solver did converge.

Using the parameter training sampling strategy **log_50** did indeed improve the projection error, see figure 11, for most lower parameter values although it still lies above the given greedy tolerance. However, this also increased the dimension of the reduced basis space to $N = 34$. Despite the improved projection error, the actual performance for parameter values below 10^{-3} is poor, as the Newton algorithm fails to converge to the best solution or to converge at all.

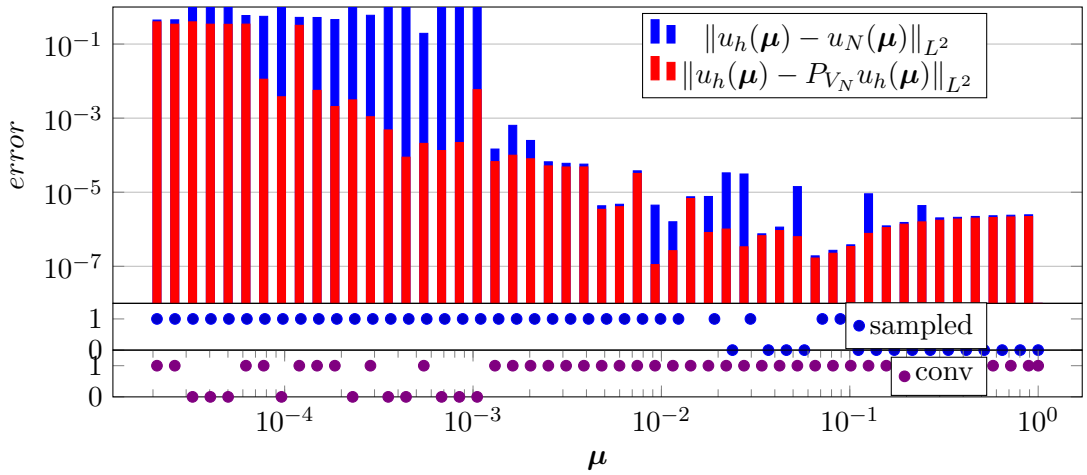


Figure 11: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_P**, trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**.

Even when we tested the reduced model using the high-fidelity solution for a given parameter value—projected onto the reduced space—as the initial guess for the reduced problem with the same parameter, the solver did not necessarily converge to the best possible approximation or achieve convergence at all. This can be seen in figure 12.

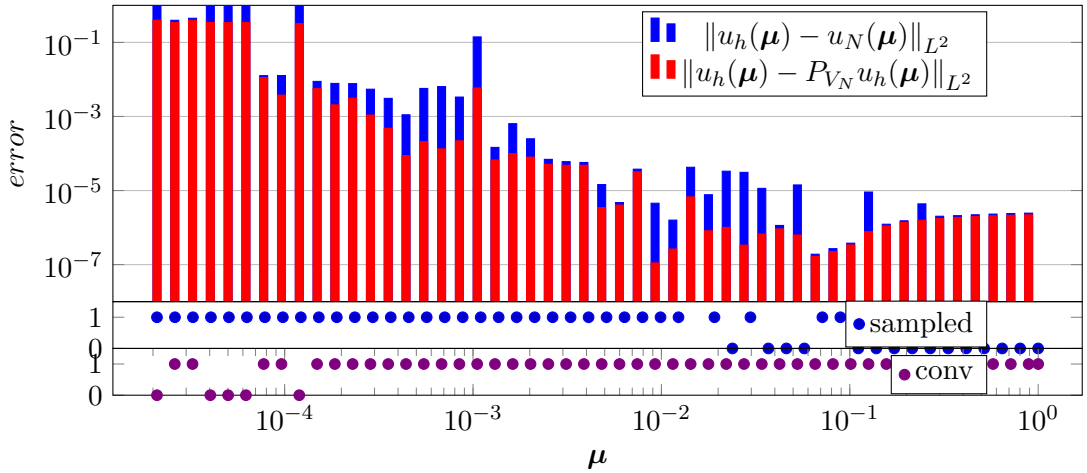


Figure 12: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_P**, trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**. Initial guess for reduced probem solver is the corresponding projected high-fidelity solution $P_{V_N} u_h(\boldsymbol{\mu})$.

Even when testing the reduced problem for the training parameter set $\mathcal{P}^{\text{train}}_{\text{log\_50}}$, the Newton algorithm also showed poor performance (see figure 13). For parameter values below 10^{-3}, it failed to converge to the best possible solution or did not converge at all.

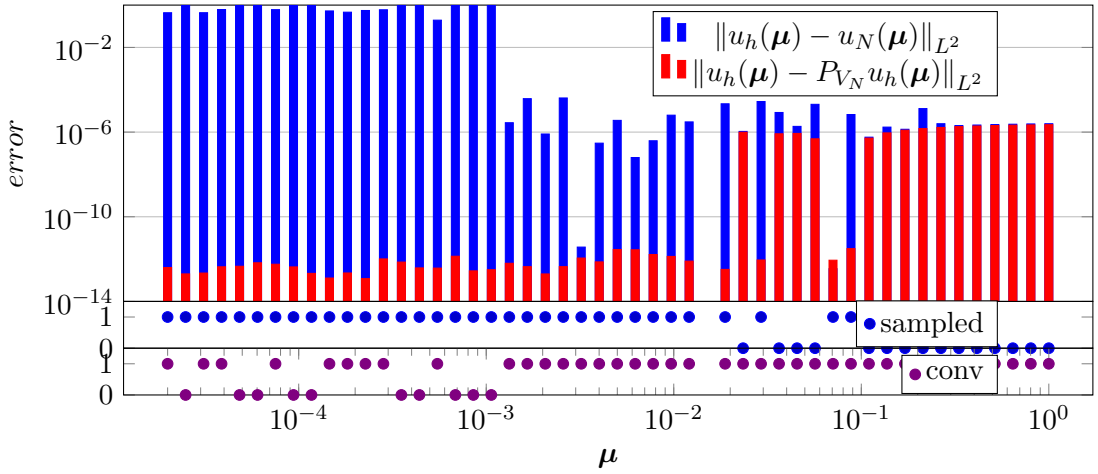


Figure 13: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_P**, trained with test parameter sampling **log_50**, tested for parameter values used for training.

Using a much larger training set with 300 parameter values improved the performance, as shown in figure 14. This large training set yields a reduced space dimension of $N = 94$. Now, the reduced model performs poorly only for parameter values below 10^{-4}.

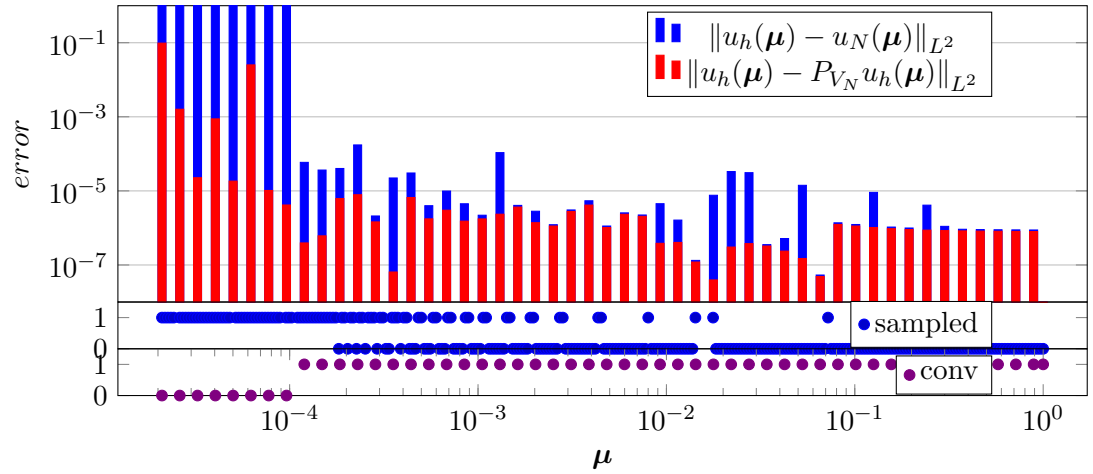


Figure 14: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_P**, trained with test parameter sampling **log_300**, test based on parameter sampling **log_50**.

In summary, we observe that the reduced model for **F_P** struggles to accurately represent solutions for small parameter values, and the Newton solver also performs badly in this parameter range. Improved damping strategies or more suitable initial guesses may be necessary to enhance convergence. Additionally, better-suited reduced basis models could improve performance in challenging parameter ranges. For this, adaptive greedy sampling strategies may be useful. Therfore, in the following section we will test the h-type Greedy algorithm.

6.5 Adaptive Domain Decomposition with h-tpye Greedy Algorithm

In this section, we construct multiple reduced models for a single test problem using an adaptive parameter domain decomposition based on the h-type Greedy algorithm (see Section 4.7).

As initial training parameter set, we define

$$\mathcal{P}^{\text{train}}_{(1)} = \mathcal{P}^{\text{train}}_{\text{log\_ns}} := \{\boldsymbol{\mu}_i\}_{i=0}^{ns}.$$

We use a logarithmic discretization of the parameter domain—and of its subdomains as well—to ensure that the parameter space is adequately represented in regions where the problem becomes more challenging.

Therfore we also define a proximity function that measures distance between parameter values on a logarithmic scale:

$$d_{B_l}(\boldsymbol{\mu}) = |\log_{10}(\boldsymbol{\mu}) - \log_{10}(\hat{\boldsymbol{\mu}}_{B_l})|.$$

The resulting reduced models are evaluated using the same strategy described in the previous section for the standard Greedy algorithm. In the following figures, the top plot illustrates the error norms (6.3) and (6.4). Below that, the second plot shows the parameter values selected by the h-type Greedy algorithm. The different colors used in this plot represent the distinct models $\mathcal{M}_{B_l}$, which are introduced in section 4.7, each corresponding to a different subdomain in the parameter space. The third plot, if present, indicates whether the Newton method converged for each corresponding parameter value. If this plot is not present, the Newton solver successfully converged for all tested parameter values.

We begin our analysis with test problem **F_0.5** because it is expected to be less challenging than the more difficult problem **F_P**.

Test Problem F_0.5

For this test problem, we set the error tolerance to $\epsilon^1_{\text{tol}} = 10 \cdot h^2 = 10^{-5}$, and we limit the reduced basis dimension to $\bar{N} = 5$. The initial parameter training set is generated using the sampling strategy **log_20**.

With these settings, the h-type Greedy algorithm produces 7 subdomains, each associated with a separate reduced model. These submodels are distributed nearly uniformly across the parameter domain on a logarithmic scale.

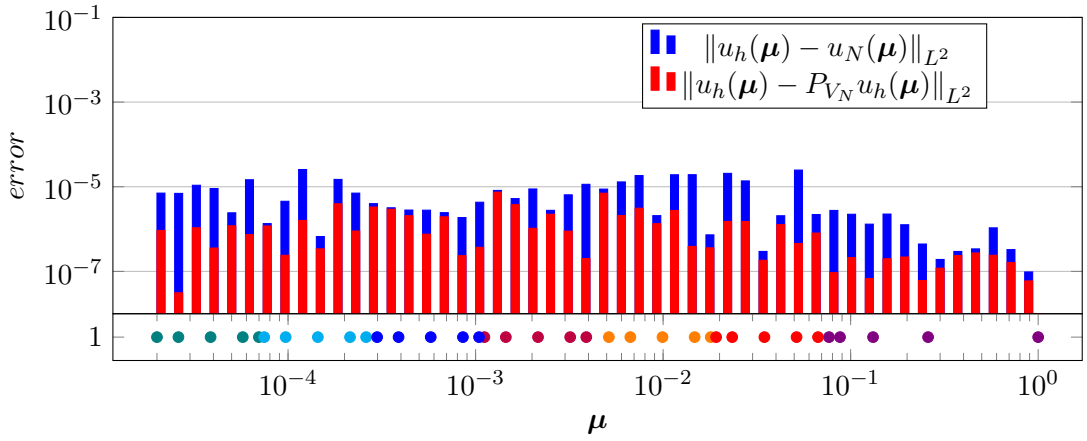


Figure 15: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_05**

The performance results are shown in figure 15. We observe that the parameter domain is sampled very coarsely for higher parameter values, while the subdomains become much smaller in the lower parameter range. Despite using only 5 reduced basis vectors per model, the reduced models achieve an accuracy below the specified tolerance across the entire parameter domain. Furthermore, the Newton algorithm converges at least nearly to the best possible solution in all cases. Sadly, this ideal behavior does not occur for the test problem **F_P**, as we will see in the following section.

Test Problem F_P

Since this problem is way more challenging than the previous one, we slightly increased the error tolerance to $\epsilon_{\text{tol}}^1 = 100 \cdot h^2 = 10^{-4}$. Nevertheless, we also had to increase the maximum allowed reduced basis dimension to $\bar{N} = 15$, to achieve this tolerance whithin a reasonable number of submodels. In addition, to provide a sufficient number of training parameter values, we provided a denser training set by using the sampling strategy **log_50**.

During the numerical tests, we observed that the Newton solver did not converge reliably for parameter values below $5 \cdot 10^{-5}$. Therefore we reduced the training parameter range to $\mathcal{P}^{\text{train}} = [5 \cdot 10^{-5}, 1]$ and also limited the test parameter range to $\mathcal{P}^{\text{test}} = [5.09 \cdot 10^{-5}, 0.89]$.

With these updated settings, the h-type Greedy algorithm generated 14 subdomains, each with its own reduced model. Most of these subdomains correspond to parameter values below 10^{-3}, and their sizes are very small, which reflects the high complexity of the solutions in this region.

Figure 16 shows the resulting error norms. In contrast to the standard Greedy algorithm, we now see that the projection error significantly improved even for very low parameter values. For all test parameters the error remains below the prescribed tolerance.

However, once again, the Newton solver did not always converge. In some cases, it failed to reach the best possible reduced solution, and in other cases, it did not converge at all. This shows that, although the theoretical approximation quality of the reduced basis is good, the practical computation of the reduced solution still remains problematic in low parameter regions.

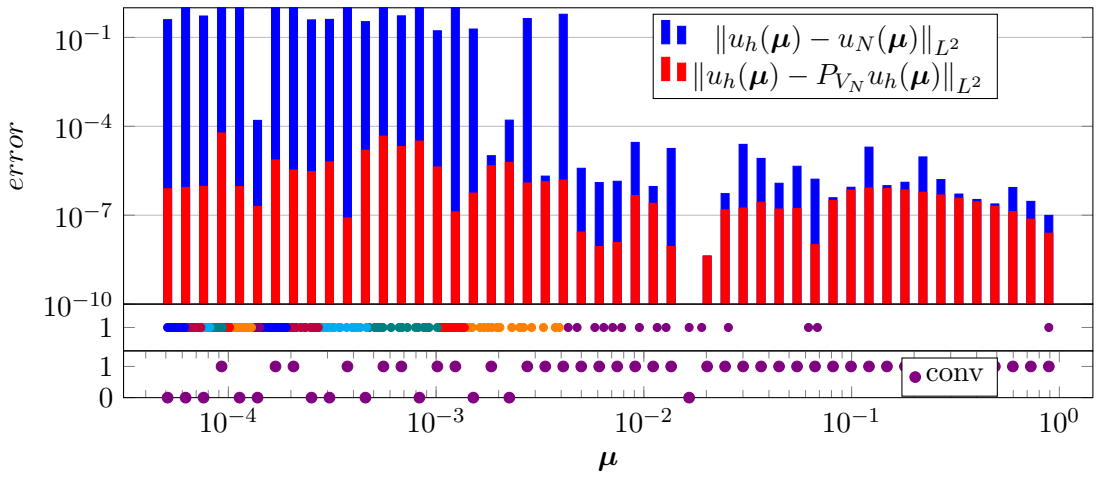


Figure 16: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_P**

But, solving the reduced models using the projected high-fidelity solution as the initial guess yields very accurate results. These results are very close to the best possible solution in the reduced space, as shown in figure 17. This indicates that the best possible solution indeed

satisfies the residual tolerance prescribed for solving the reduced problem. This solution is, however, not reached when using the initial guess strategy **P**.

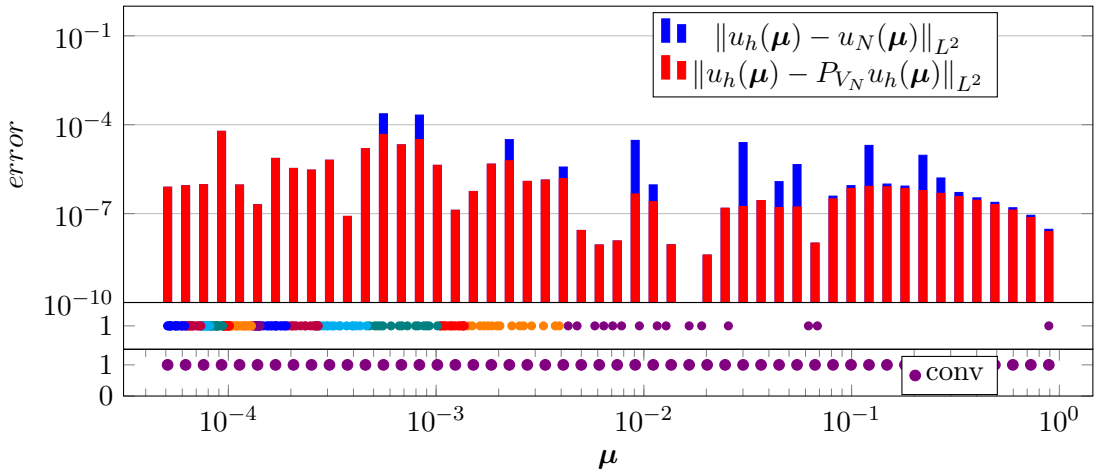


Figure 17: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_P**

Therefore, we tried to improve the convergence of the Newton algorithm by using a specific initial guess for each submodel. In particular, we computed the high-fidelity solution for the anchor parameter of each submodel and projected it onto the corresponding reduced space. The resulting RB coefficients were then used as the initial guess for solving the reduced problem with any parameter belonging to that submodel. As shown in Figure 18, this approach improved convergence only for very few lower parameter values. Furthermore, it also led to a loss of convergence for almost all parameter values in the middle of the parameter domain.

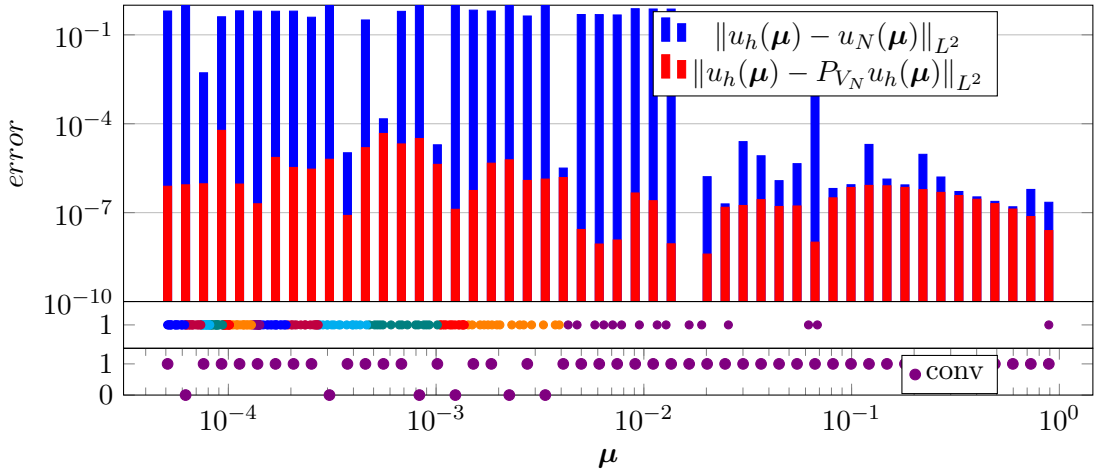


Figure 18: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **F_05**

In summary, the constructed reduced models are in theory capable of representing the solutions accurately within the presrcibed error tolerance across the entire parameter domain, since the projection errors lie below this tolerance. However, in practice we were not able to reliably solve the reduced problems for parameter values below $\boldsymbol{\mu} = 5 \cdot 10^{-3}$, because the Newton solver failed to converge in this range.

7 Addidional Solution Branches

During our previous numerical experiments, we noticed that the high-fidelity problem for test case **F_P** did not converge for some parameter values around $\boldsymbol{\mu} = 0.016$. Note, the missing error bars at $\boldsymbol{\mu} = 0.016$ in the error plots from the previous section. These bars are missing because the solver failed to converge for that parameter value.

To better understand this issue, we examined the solutions in this critical region. First, we solved the high-fidelity problem for parameter values slightly above and below $\boldsymbol{\mu} = 0.016$ and analyzed the resulting solutions. During this section we set the maximum number of Newton iterations to MAXIT $= 100$, the number of mesh intervalls to $N_h = 1000$ and the tolerance for the residual to RTOL $= h^2 = 10^{-6}$. As can be seen in figure 19, the solutions for values $\boldsymbol{\mu} \geq 0.0175$ show a specific structure. The solutions for parameter values $\boldsymbol{\mu} \leq 0.0139$ look significantly different. In the region between these values, the solver did not converge at all. This suggests that there might be a bifurcation point around $\boldsymbol{\mu} = 0.016$, where the structure of the solution changes fundamentally. The numerical instability in this region could be a result of this bifurcation, as the Newton method struggles to decide which solution branch to follow.

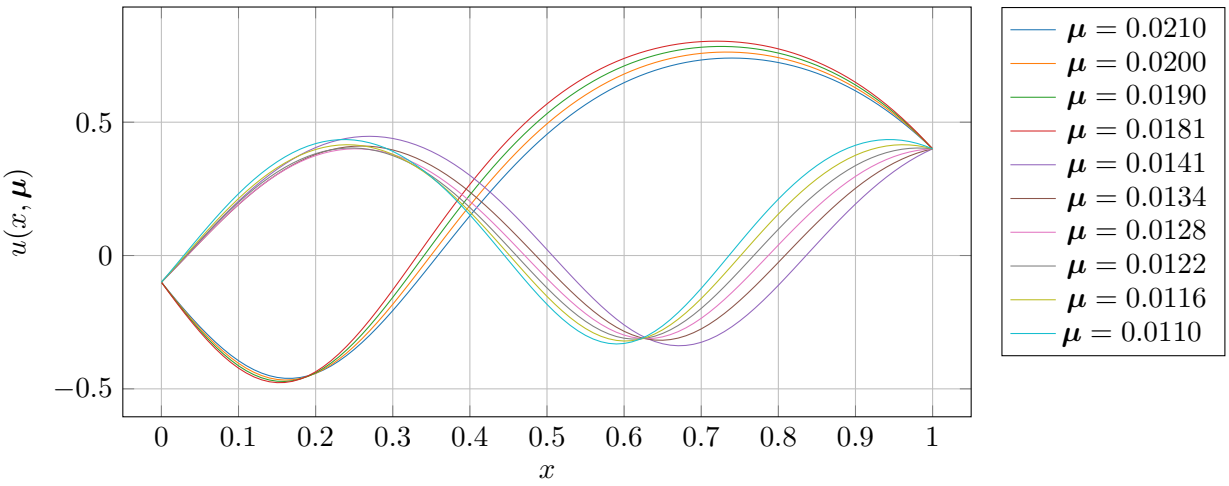


Figure 19: Solutions of **F_P** for parameters around critical value $\boldsymbol{\mu} = 0.016$.

To try to solve this issue, we applied a continuation inspired strategy. We used the solution for one parameter value as the initial guess for a neighboring parameter. First, we tried to move upward through the critical parameter region, starting from a lower parameter value. However, this did not improve the convergence, the solver still failed. Then we tried the opposite direction. Starting with a parameter value above the critical region, we moved downward. In this case, the Newton solver successfully converged for all parameter values, even those in the critical region. However, the solutions obtained in this way for parameter values $\boldsymbol{\mu} \leq 0.016$ were different from those we had computed earlier for the same parameters, as can be seen in figure 20. This means we found an additional solution branch. The new solutions share the same general shape and structure as the known solutions for $\boldsymbol{\mu} > 0.016$.

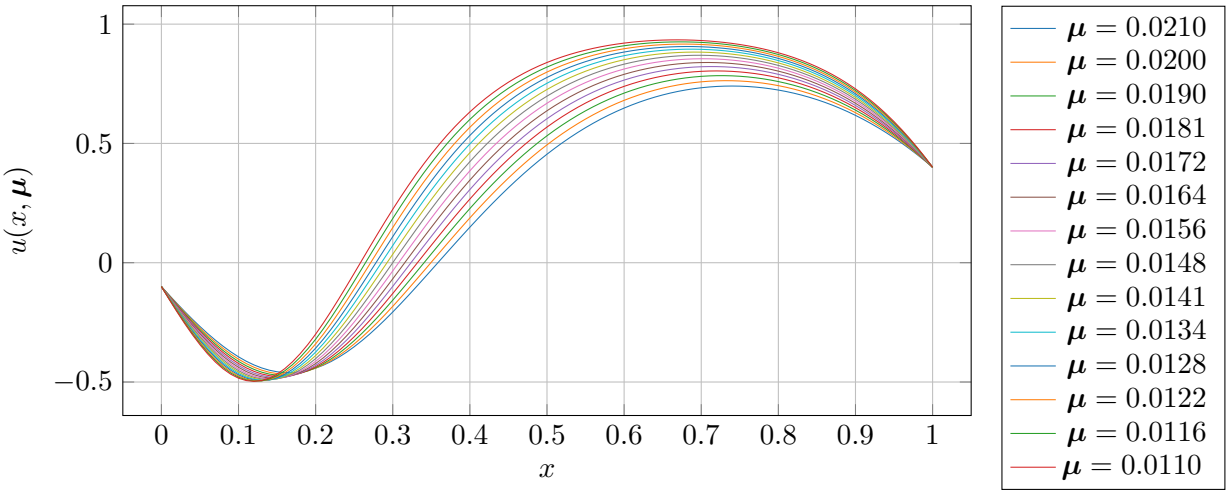


Figure 20: Solutions of $\mathbf{F}$ for parameter around critical value $\boldsymbol{\mu} = 0.016$ obtained using a continuation strategy.

We now summarize all obtained solutions so far. In section 5, we already analyzed the behavior of the high-fidelity problem for the test problem $\mathbf{F}$. Here we identified one solution branch that exits for parameter values in the range $[0.07245, 1]$. These solutions are visualized in figure 21 and can be computed by either using the $\mathbf{P}$ or $\mathbf{0.5}$ initial guess strategy.

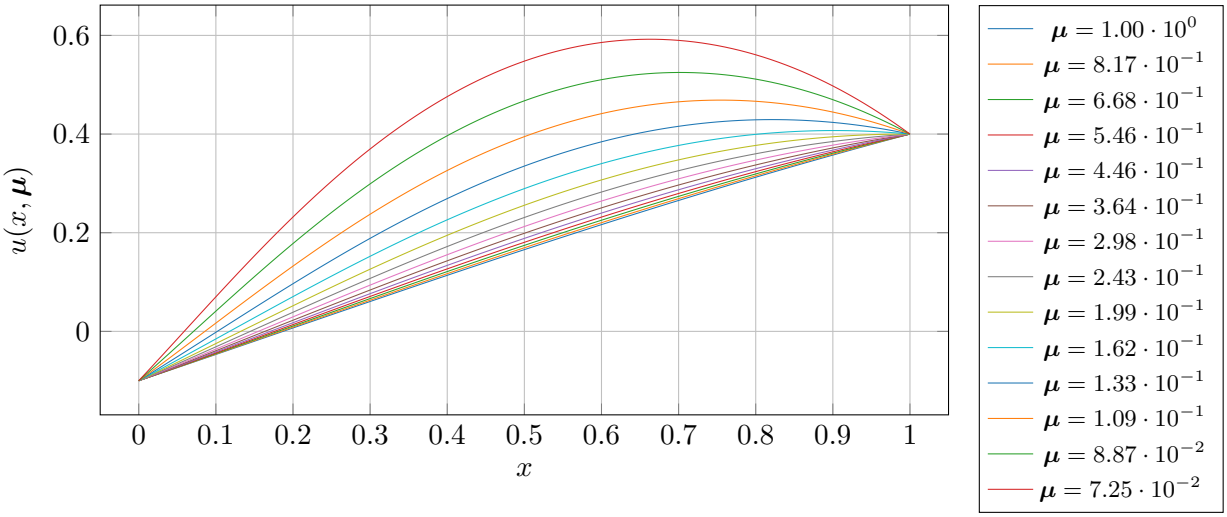


Figure 21: Solutions of $\mathbf{F}$ for parameter values in $[7.25 \cdot 10^{-2}, 1]$.

In the parameter region from $[0.016, 0.07245]$ we observe the presence of two distinct solution branches, as visualized in Figure 22. The first branch is a continuation of the solution path identified earlier and can be reached using the initial guess strategy $\mathbf{0.5}$. The second, new, branch appears around the parameter value $\boldsymbol{\mu} = 0.07245$ and is accessed using the initial guess strategy $\mathbf{P}$.

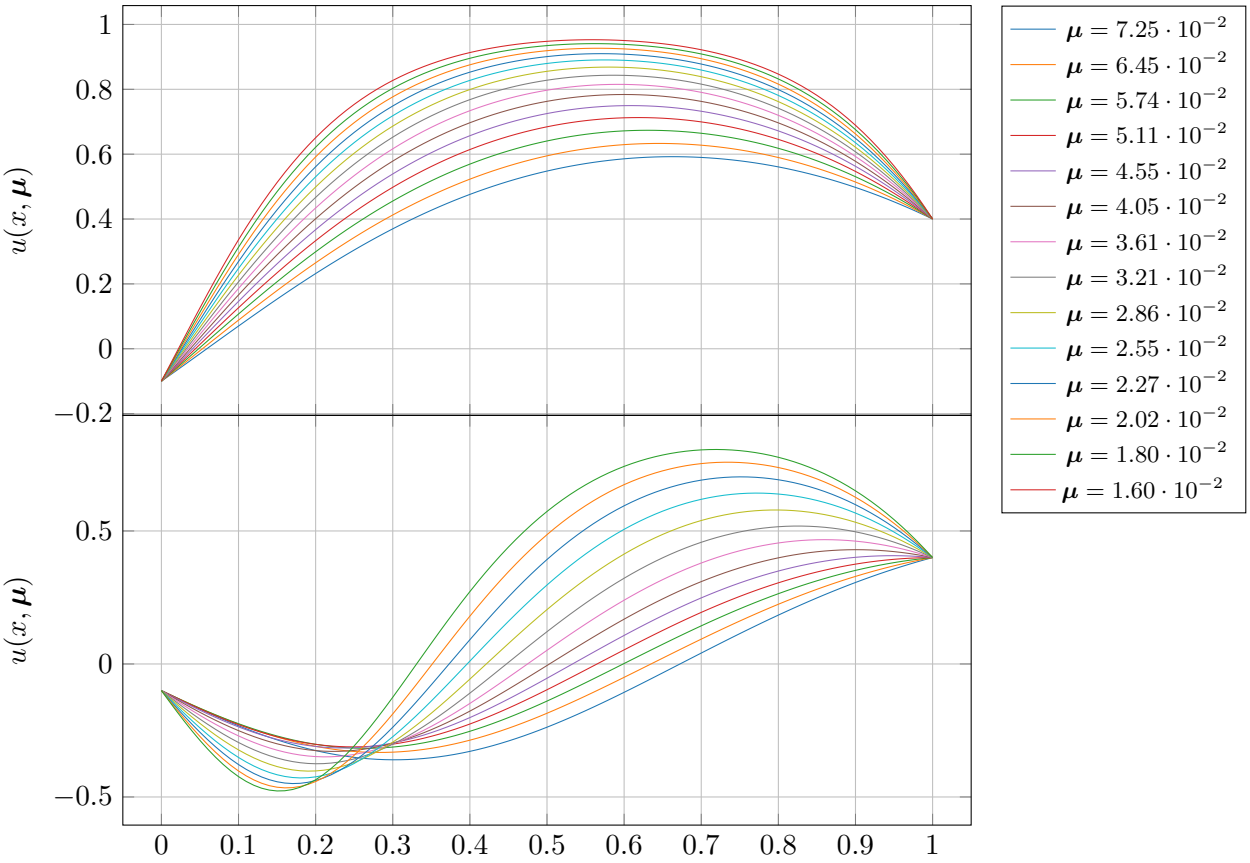


Figure 22: Solutions of $\mathbf{F}$ for parameter values in $[1.6 \cdot 10^{-2}, 7.25 \cdot 10^{-2}]$ obtained with initial guess strategy **0.5** (upper plot) and $\mathbf{P}$ (lower plot).

So far, in the parameter region from $[2 \cdot 10^{-5}, 0.016]$, we have obseved three distinct solution branches, since the second branch appears to split again at around $\boldsymbol{\mu} = 0.016$. This new solution branch, as the previous ones, are shown in Figure 23. The first two branches still are reched using the initial guess strategies **0.5** and $\mathbf{P}$. The third branch is computed with a continuation strategy, where the solution at $\boldsymbol{\mu} = 0.016$ is used as the initial guess for solving the problem at the next smaller parameter, and so on.

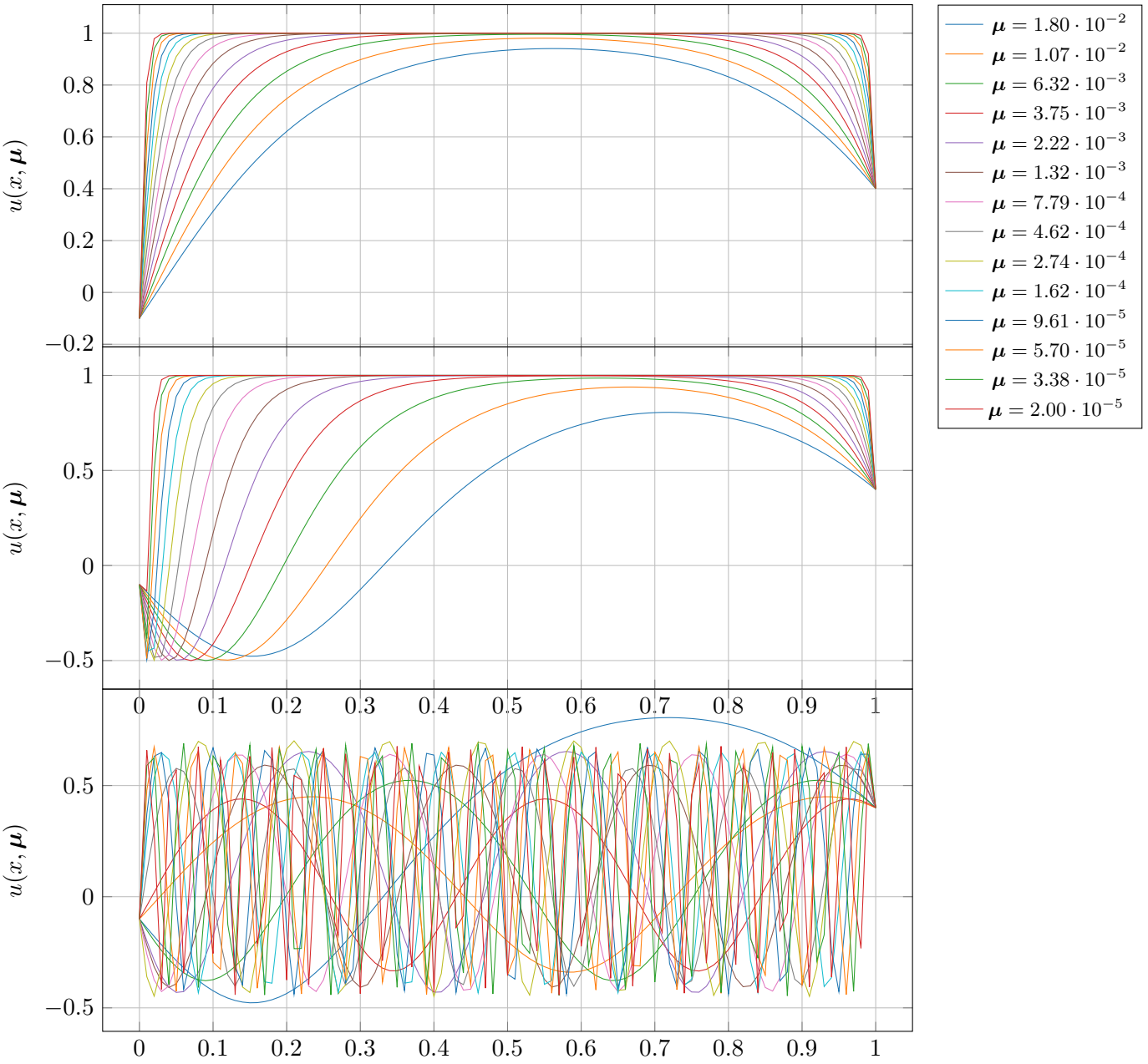


Figure 23: Solutions of $\mathbf{F}$ for parameter values in $[2 \cdot 10^{-5}, 1.6 \cdot 10^{-2}]$ obtained with initial guess strategy **0.5** (upper plot), $\mathbf{P}$ (middle plot) and a continuation strategy (lower plot)

Inspired by this strategy, we applied the continuation strategy starting from several perodic solutions. To do this, we restricted the parameter domain to $[2 \cdot^{-5}, \boldsymbol{\mu}_{\max}]$ and began by solving the problem at $\boldsymbol{\mu}_{\max}$ using the initial guess strategy $\mathbf{P}$. We then used this solution as the initial guess for the next smaller parameter value, and continued this process. Using this method, we were able to discover many additional solution branches. In fact, the number of possible branches appears to be endless.

In the following figure 24, we show two examples of such solution paths. In the first example we started the continuation at $u_h(\boldsymbol{\mu} = ?)$. In the second example we started at $u_h(\boldsymbol{\mu} = ?)$. Note that, to ensure convergence along the continuation paths, we had to compute solutions for many more parameter values than shown. For clarity, we only display a subset of them in the plots.

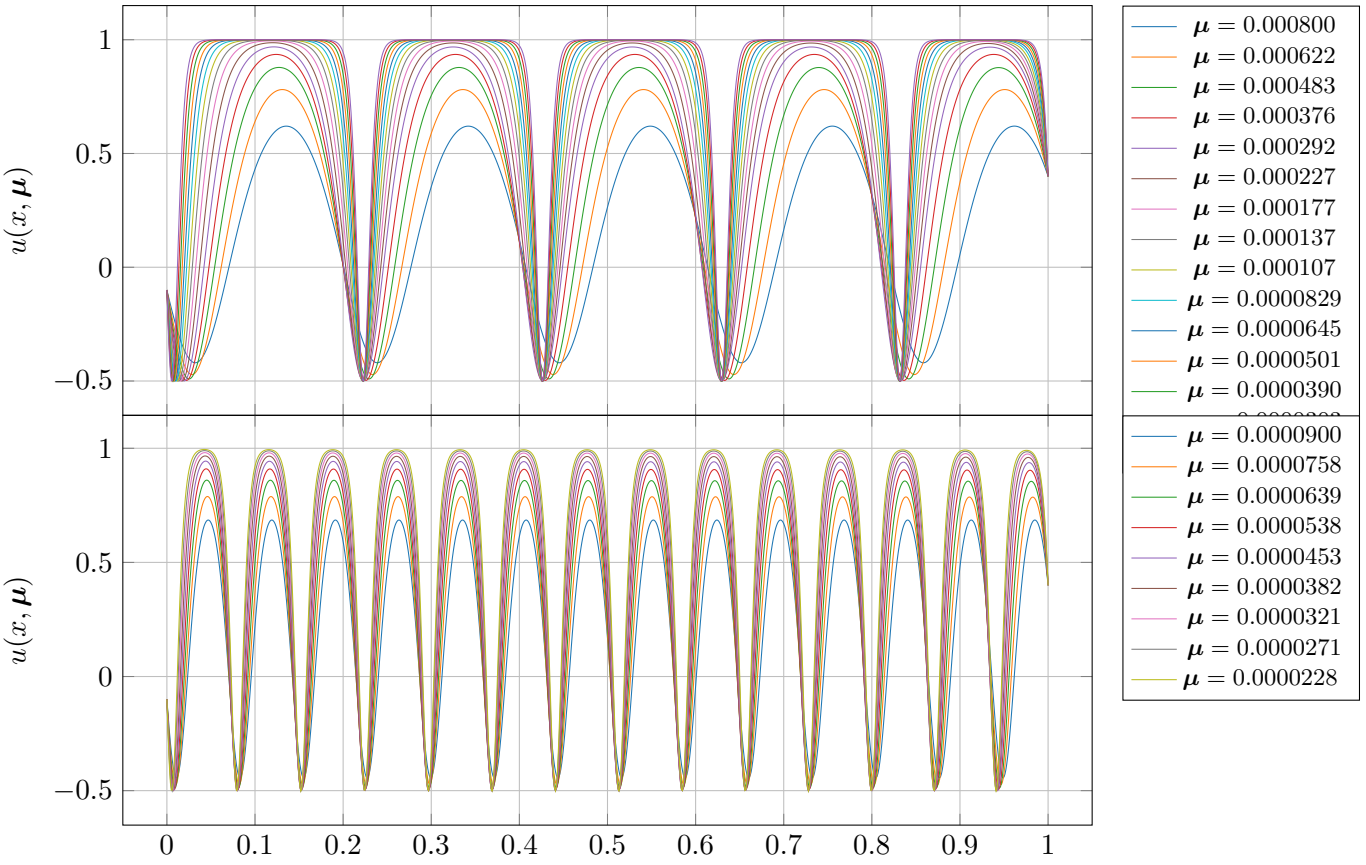


Figure 24: Solutions of $\mathbf{F}$ for parameter values in $[2 \cdot 10^{-5}, 8 \cdot 10^{-4}]$ (upper plot) and $[2 \cdot 10^{-5}, 9 \cdot 10^{-5}]$ (lower plot) obtained using a continuation strategy.

With this observation, we may now have an explanation for the poor performance of the reduced basis method for test problem $\mathbf{F\_P}$ in section 6.4 and 6.5. It appears that the reduced basis was constructed from snapshots belonging to different solution branches. As a result, the approximation space had to capture very different types of solutions at once. However, the structure of the solutions along a single solution branch does not vary much and would likely be well suited for reduced basis methods. Mixing different branches leads to large variations in the snapshot set, which makes it difficult to build an efficient reduced space. Additionally, it is very likely that the Newton solver used to solve the reduced problems did not converge to the intended solution branch. Instead, it may have converged to a different solution that was not well represented by the reduced model.

8 Conclusion

In this thesis, we developed a Python package based on FEniCS to test reduced basis methods for semilinear problems. This package is designed in a modular and flexible way to make future adaptations and extensions easier.

Currently, the implementation is not yet fully efficient. In each Newton iteration, the reduced residual vector and the Jacobian matrix are still assembled using the high-fidelity problem. This limits the speed-up of the reduced model and is an important aspect for future improvement.

At the moment, the framework supports only one spatial dimension. However, the structure is prepared for generalization. To extend it to higher dimensions, only two classes need to be adapted. In the class `MyRedNonlinearProblem`, the mesh and function space must be modified. Further its methods that depend on the degree of freedom to coordinate mapping (e.g. the method for setting the initial guess) must be revised. In the class `MySemilinearProblem`, the reduced basis used to reduce the Jacobian matrix and residual vector for the Newton method must be adapted such that the DOFs corresponding to the boundary conditions are set to zero.

In our numerical experiments with the high-fidelity problem, we obtained good results using a constant initial guess with a value of 0.5. We also tested an initial guess strategy, where we used the solution of the Poisson equation as initial guess. The latter approach yield periodic solution functions, which differed from the solutions obtained with the former initial guess strategy. To improve convergence to that periodic solution branch, we tested several damping strategies. Among the damping strategies tested, the affine covariant, error-oriented damping strategy introduced by Deuflhard in [5] performed best and led to convergence for parameter values as small as $\boldsymbol{\mu} = 5 \cdot 10^{-5}$.

Furthermore, we investigated the singular value decay of the snapshot matrix obtained via Proper Orthogonal Decomposition with respect to the 2-norm. This was done for the parameter range $[2 \cdot 10^{-5}, 1]$, using a logarithmic and a linear sampling strategy. For the solution branch computed with a constant initial guess of 0.5, the singular values decayed rapidly for both sampling strategies—slightly faster with linear sampling, although the difference was not significant. However, for the periodic solution branch, the decay of the singular values was significantly slower when using logarithmic sampling. This indicates that constructing an efficient reduced basis for the periodic solutions is much more difficult in a low parameter range. In contrast, the solution branch obtained from the constant initial guess appears less challenging and more suitable for reduced basis methods.

This observation turned out to be correct when we tried to build reduced models for both solution branches using the Greedy algorithm. The model order reduction worked well for the solution branch obtained with a constant initial guess. However, it was not very successful for the periodic solution branch. In the first case, a very small reduced basis was built using only a small training parameter set. Still, the reduced model achieved very good accuracy. In contrast, for the periodic solution branch, the reduced model was less accurate even when using a much larger training set. Additionally, the Newton method often had problems solving the reduced problem in this case.

To improve the results, we applied the h-type Greedy algorithm from [6] to divide the parameter domain into smaller regions, where the solutions are more likely to depend on each other. As expected, this approach worked very well for the solution branch obtained with the constant initial guess. In this case, only seven submodels with a reduced basis size of $N = 5$ were enough to achieve high accuracy.

For the periodic solution branch, however, the results were less favorable than expected. Even though we increased the reduced basis size to $N = 15$, we still needed 14 submodels to achieve the given error tolerance. While the projection error for the tested parameter values was below this error tolerance, the Newton solver often failed to converge to the best possible reduced solution—especially in the region with small parameter values. This means that, although model order reduction was theoretically successful, accurate reduced solutions could not be computed in practice.

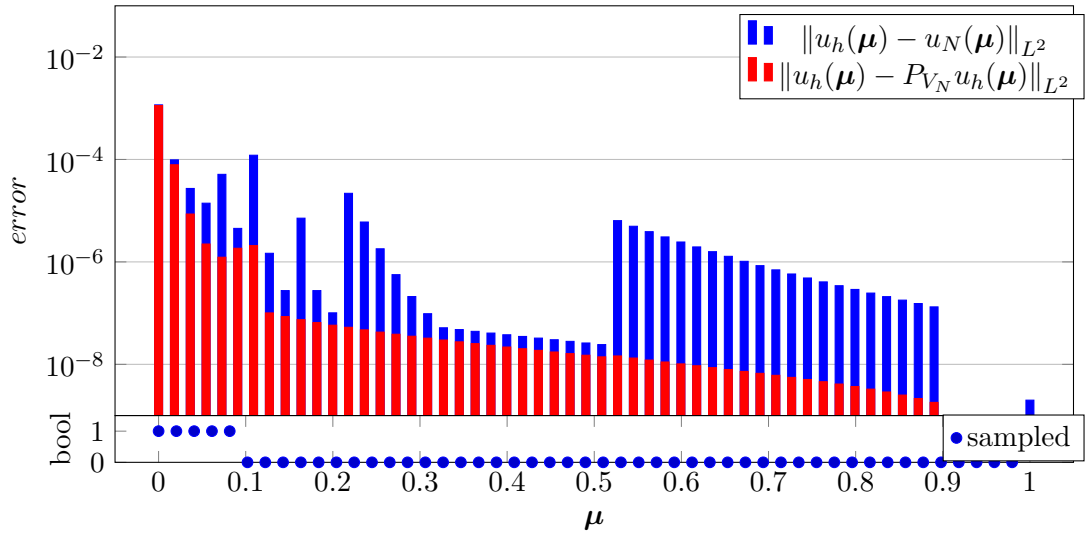
We also tested different strategies for choosing the initial guess for that case. In particular, we used the solution at an anchor parameter as initial guess when solving for parameters within the corresponding subdomain. This improved convergence for some smaller parameter values, but led to a loss of convergence for parameter values in the middle of the parameter domain.

In summary, we achieved good model order reduction when the singular values decay quickly. In the other cases, further investigation is needed to understand why the Newton solver does either not converge, or converges to the wrong solution, although the reduced model is able to represent the solution within the given error tolerance. One idea for future work is to use Greedy sampling based on nonlinear optimization, as introduced in [21]. It would also be interesting to extend this work to problems with higher spatial dimensions or more parameters.

A Performance of Reduced Problems for Test Problems mmsF_05 and slP_0

A.1 Plots of error norms for reduced test problem mmsF_05

(a) Test based on test parameter sampling **lin_50**.



(b) Test based on test parameter sampling **log_50**.

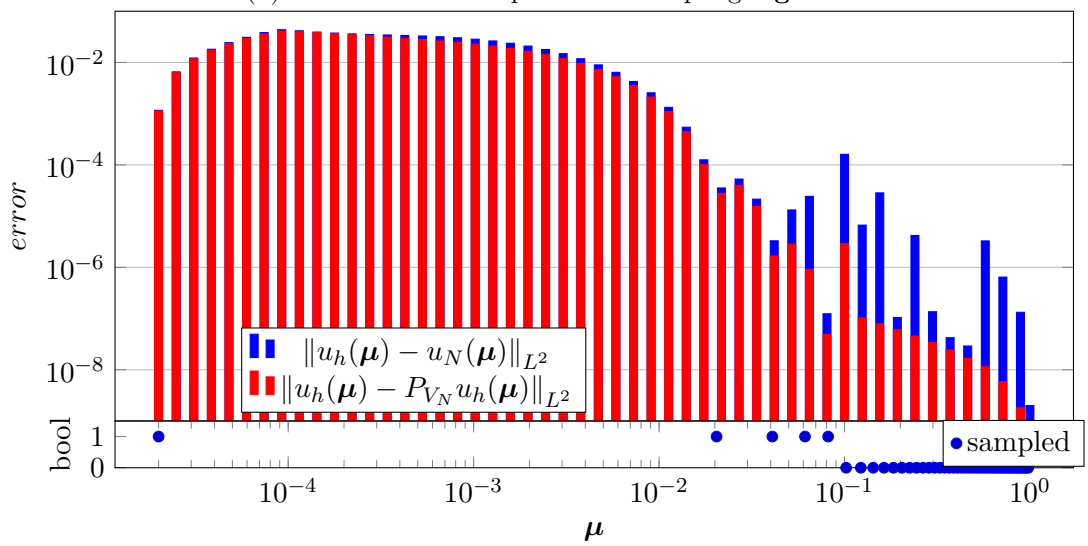


Figure 25: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **mms_05**, trained with test parameter sampling **lin_50**.

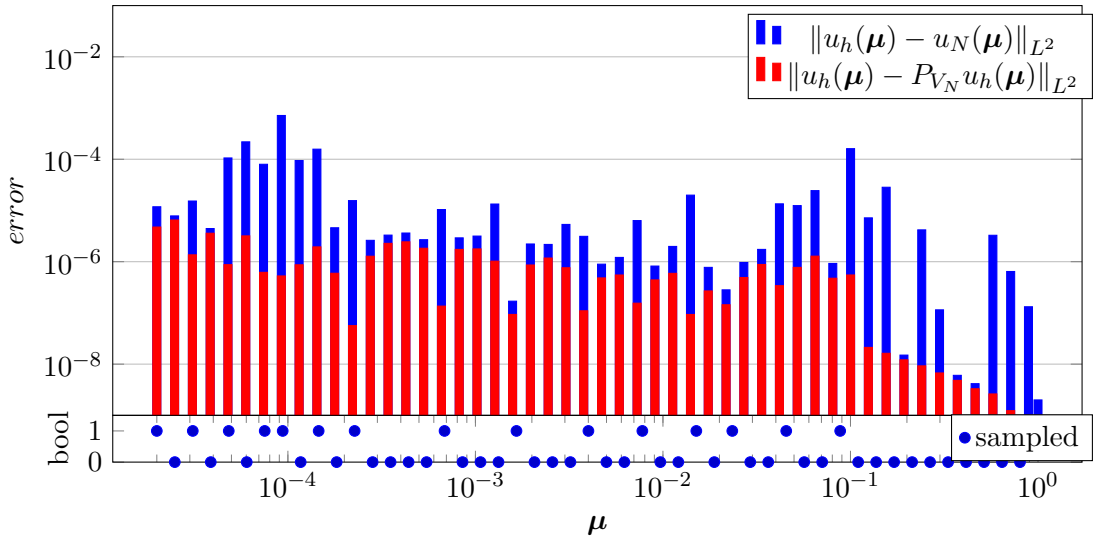


Figure 26: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **mms_05**, trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**.

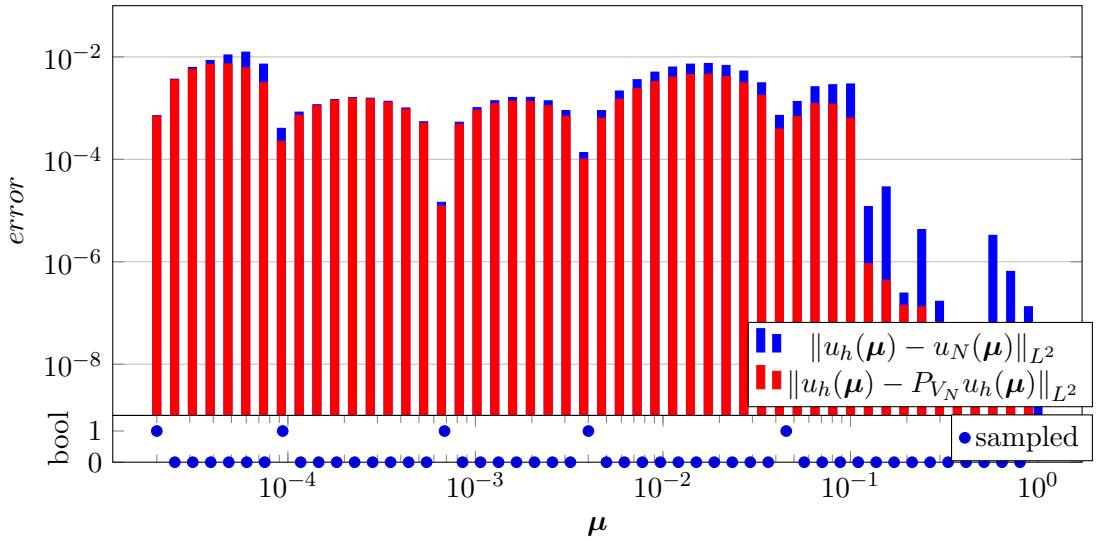


Figure 27: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **mms_05** (N_max=6), trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**.

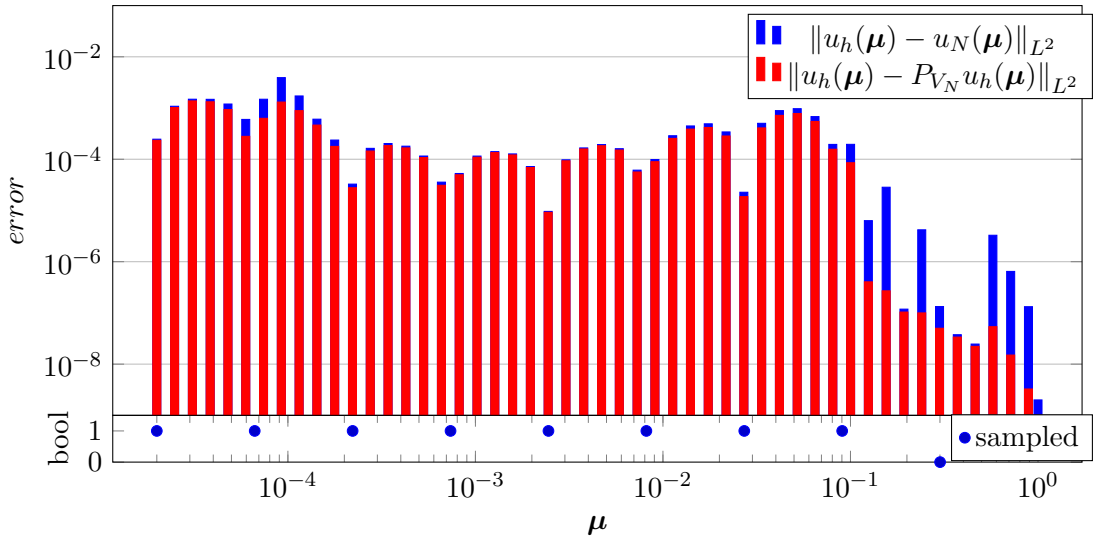
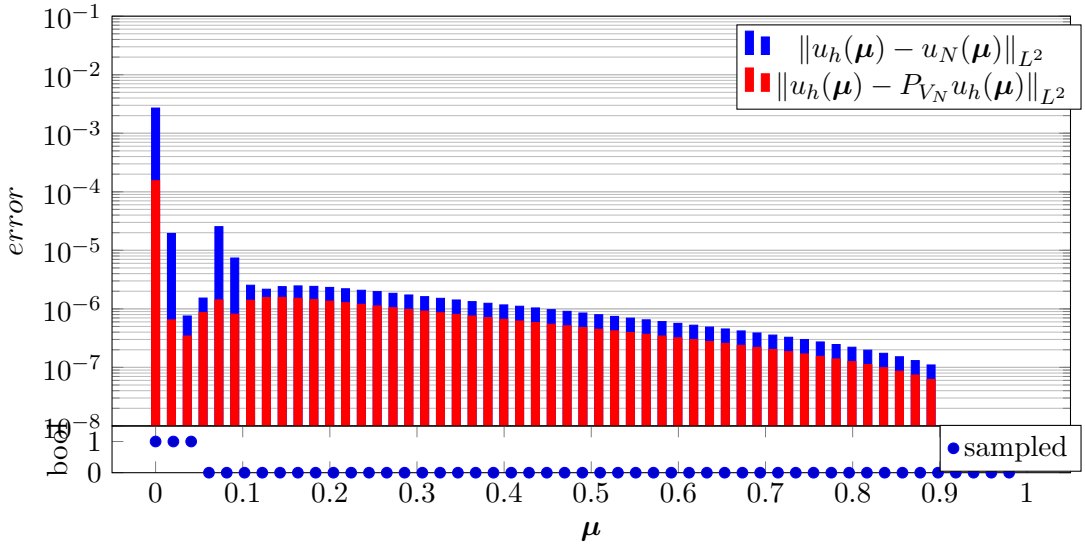


Figure 28: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **mms_05**, trained with test parameter sampling **log_10**, test based on parameter sampling **log_50**.

A.2 Plots of error norms for reduced test problem slP_0

(a) Test based on test parameter sampling **lin_50**.



(b) Test based on test parameter sampling **log_50**.

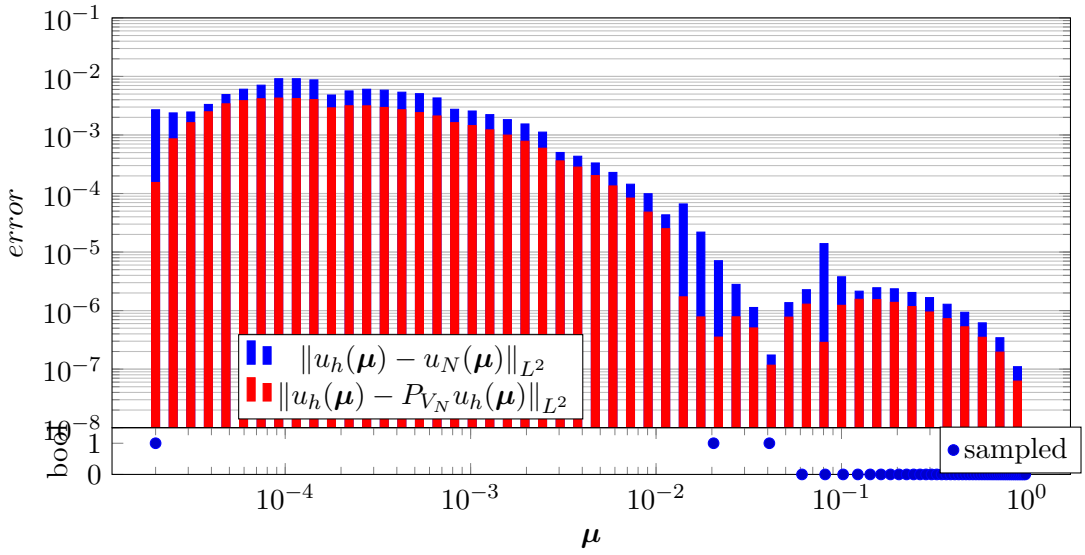


Figure 29: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **slP_05**, trained with test parameter sampling **lin_50**.

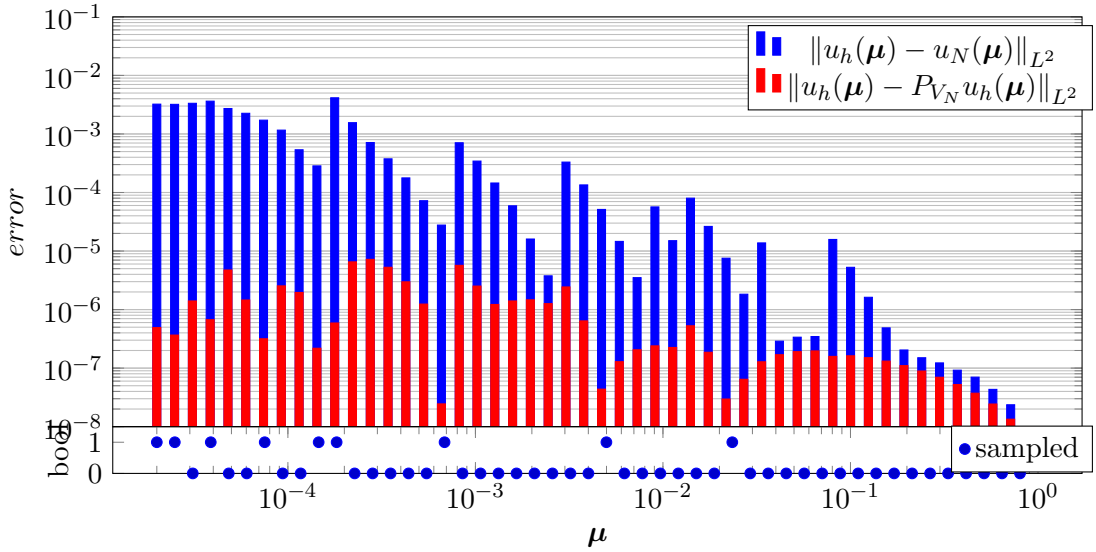


Figure 30: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **slP_0**, trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**.

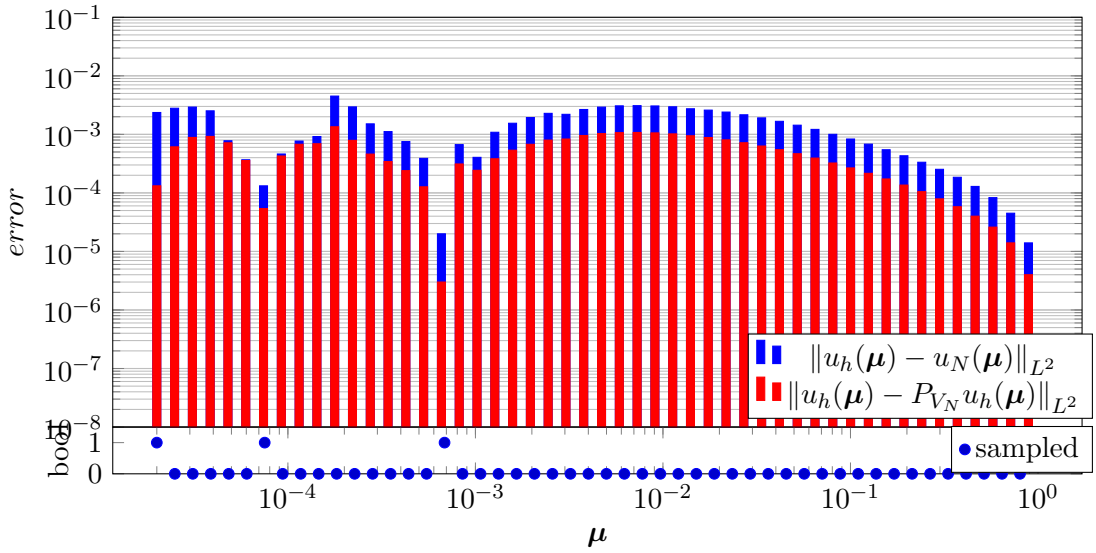


Figure 31: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **slP_0** (N_max=4), trained with test parameter sampling **log_50**, test based on parameter sampling **log_50**.

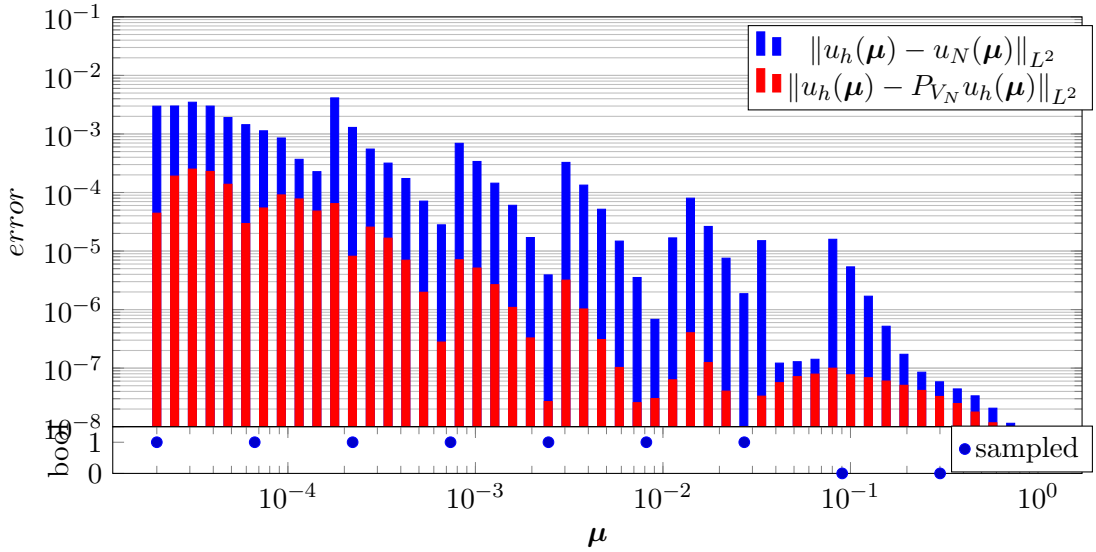


Figure 32: Error norms $e_P(\boldsymbol{\mu})$ and $e(\boldsymbol{\mu})$ for the reduced test problem **slP_0**, trained with test parameter sampling **log_10**, test based on parameter sampling **log_50**.

References

[1] Mario Ohlberger and Stephan Rave. "Reduced Basis Methods: Success, Limitations and Future Challenges". In: (Dec. 2015).

[2] Peter Binev et al. "Convergence Rates for Greedy Algorithms in Reduced Basis Methods". In: *SIAM Journal on Mathematical Analysis* 43.3 (2011), pp. 1457–1472. DOI: 10.1137/100795772.

[3] Mario Amrein and Thomas P. Wihler. "An adaptive Newton-method based on a dynamical systems approach". In: *Communications in Nonlinear Science and Numerical Simulation* 19.9 (Sept. 2014), pp. 2958–2973. ISSN: 1007-5704. DOI: 10.1016/j.cnsns.2014.02.010. URL: http://dx.doi.org/10.1016/j.cnsns.2014.02.010.

[4] Urban Funken Lebiedz. *Numerik 2, Einführung in die numerische Analysis*. Script, Ulm University. 2013.

[5] Peter Deuflhard. *Newton Methods for Nonlinear Problems: Affine Invariance and Adaptive Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-23899-4. DOI: 10.1007/978-3-642-23899-4_1.

[6] Jens Eftang, Anthony Patera, and Einar Rønquist. "An "hp" Certified Reduced Basis Method for Parametrized Elliptic Partial Differential Equations". In: *SIAM J. Scientific Computing* 32 (Sept. 2010), pp. 3170–3200. DOI: 10.1137/090780122.

[7] Hans Petter Langtangen and Anders Logg. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-52462-7. DOI: 10.1007/978-3-319-52462-7_2. URL: https://doi.org/10.1007/978-3-319-52462-7_2.

[8] Anders Logg, Kent-Andre Mardal, and Garth Wells, eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-23099-8. DOI: 10.1007/978-3-642-23099-8_1. URL: https://doi.org/10.1007/978-3-642-23099-8_1.

[9] Gianluigi Rozza et al. *Real Time Reduced Order Computational Mechanics. Parametric PDEs Worked Out Problems*. SISSA Springer Series. Springer Cham, 2024. ISBN: 978-3-031-49891-6.

[10] *RBniCS Project*. https://www.rbnicsproject.org/. Accessed: 2025-05-21. 2025.

[11] Martin A Grepl et al. "Efficient reduced-basis treatment of nonaffine and nonlinear partial differential equations". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 41.3 (2007), pp. 575–605. DOI: 10.1051/m2an:2007031.

[12] *pyMOR Documentation (Version 2024.2.0)*. https://docs.pymor.org/2024-2-0/index.html. Accessed: 2025-05-21. 2024.

[13] Anna Dall'Acqua. *Partille Differentialgleichungen*. Script, Ulm University. 2021.

[14] Alfio Quarteroni, Andrea Manzoni, and Federico Negri. *Reduced Basis Methods for Partial Differential Equations: An Introduction*. Cham: Springer International Publishing, 2016, pp. 11–38. ISBN: 978-3-319-15431-2. DOI: 10.1007/978-3-319-15431-2_2. URL: https://doi.org/10.1007/978-3-319-15431-2_2.

[15] Hanna Gierling. *rb_semilinear: A Python implementation to explore reduced basis methods for semilinear PDEs using FEniCS*. https://github.com/HannaGierling/rb_semilinear. 2025.

[16] PETSc. *SNESLINESEARCHNLEQERR - Line Search Type*. Accessed: 2025-05-21. 2024. URL: https://petsc.org/release/manualpages/SNES/SNESLINESEARCHNLEQERR/.

[17] Karsten Urban. *Model Reduction*. Script, Ulm University. 2023.

[18] Bernard Haasdonk. *Reduced Basis Methods for Parametrized PDEs – A Tutorial Introduction*. 2020. URL: https://pnp.mathematik.uni-stuttgart.de/ians/haasdonk/publications/RBtutorial_preprint_update_with_header.pdf.

[19] Karen Veroy, Christophe Prud'homme, and Dimitrios Rovas. "A Posteriori Error Bounds for Reduced-Basis Approximation of Parametrized Noncoercive and Nonlinear Elliptic Partial Dierential Equations". In: *16th AIAA Computational Fluid Dynamics Conference* (June 2003). DOI: 10.2514/6.2003-3847.

[20] Karsten Urban. *Numerics of Partial Differential Equations*. Script, Ulm University. 2020.

[21] Karsten Urban, Stefan Volkwein, and Oliver Zeeb. "Greedy Sampling Using Nonlinear Optimization". In: *Reduced Order Methods for Modeling and Computational Reduction*. Ed. by Alfio Quarteroni and Gianluigi Rozza. Cham: Springer International Publishing, 2014, pp. 137–157. ISBN: 978-3-319-02090-7. DOI: 10.1007/978-3-319-02090-7_5. URL: https://doi.org/10.1007/978-3-319-02090-7_5.