University of Münster
Faculty of Mathematics and Computer Science

Master's thesis

Machine learning based surrogate modeling to accelerate parabolic PDE constrained optimization

Submitted by: Andy Kevin Wert
Student ID number: 461478
Supervisor: Prof. Dr. Mario Ohlberger
Assisting Supervisor: Dr. Stephan Rave
Date of submission: 9th July, 2024

Contents

1 Introduction

The selection of a good oil recovery method requires a lot of effort since there are many parameters that influence the resulting output [1]. To achieve a greater profit, one often wants to find control parameters that optimize a net present value (NPV) function which evaluates the economic value of the chosen oil recovery strategy. A difficulty here is that the computation of a NPV function is usually expensive due to the complex reservoir simulation models. A popular algorithm for the maximization of the NPV function is the ensemble-based (EnOpt) optimization method [2] which uses samples around each iterate to compute an approximation of a preconditioned gradient at that iterate. With this gradient, the next iterate is computed by a gradient ascent method. An extension of the EnOpt method is the so-called adaptive EnOpt algorithm. Here, the covariance matrix, that is used for the sampling around the iterates, is adjusted with respect to the result from the previous iteration [3].

In [4], a modification of the adaptive EnOpt method is proposed which is called the adaptive machine learning EnOpt (Adaptive-ML-EnOpt) algorithm. In this procedure, a neural network-based surrogate is trained in each iteration to reduce costly calls of the full order model NPV function and thus speed up the computation. This method is applied to an enhanced oil recovery strategy [5] where a polymer-water mixture is injected into the reservoir to bring the oil to the surface.

In this thesis, we apply the adaptive EnOpt and the Adaptive-ML-EnOpt algorithms to an optimal control problem that is constrained by parabolic equations. The source code can be found at [https://github.com/AndyWert/master-thesis]. Further instructions to run this code are specified in the file 'README.md'.

The constrained control problem, along with the solution strategy of the corresponding discretized parabolic equations, is presented in chapter 2. After that, chapter 3 presents the adaptive EnOpt algorithm. In chapter 4 the Adaptive-ML-EnOpt procedure from [4] is described. Finally, we test and compare these two algorithms in chapter 5.

2 Parabolic optimal control problems

2.1 Introduction to the problem

Our optimization problem is based on the problem that is presented in [6]. We consider a state variable u and a control variable q, defined on $(0,T)\times\Omega$ with $T\in\mathbb{R}$ and $\Omega\subset\mathbb{R}^n$.

The goal of this thesis is to minimize the function

$$J(q,u)=\frac{1}{2}\int_0^T\int_\Omega (u(t,x)-\hat{u}(t,x))^2\,\mathrm{d}x\,\mathrm{d}t+\frac{\alpha}{2}\int_0^T\int_\Omega q(t,x)^2\,\mathrm{d}x\,\mathrm{d}t, \tag{2.1a}$$

subject to the constraints

$$\begin{aligned}\partial_t u-\Delta u&=f+q \quad \text{in } (0,T)\times\Omega,\\ u(0)&=u_0 \qquad\qquad \text{in } \Omega,\end{aligned} \tag{2.1b}$$

with homogeneous Dirichlet boundary conditions on $(0,T)\times\partial\Omega$.

Let $V=H_0^1(\Omega)$, $H=L^2(\Omega)$ and $I=(0,T)$. We define our state space as

$$X:=\{v\mid v\in L^2(I,V) \text{ and } \partial_t v\in L^2(I,V^*)\}$$

and the control space as

$$Q:=L^2(I,L^2(\Omega)).$$

The notion of the inner products and norms on $L^2(\Omega)$ and $L^2(I,L^2(\Omega))$ is introduced as

$$\begin{aligned}(v,w)&:=(v,w)_{L^2(\Omega)}, & (v,w)_I&:=(v,w)_{L^2(I,L^2(\Omega))},\\ \|v\|&:=\|v\|_{L^2(\Omega)}, & \|v\|_I&:=\|v\|_{L^2(I,L^2(\Omega))}.\end{aligned}$$

By using the inner product, the weak form of the state equations (2.1b) for $q,f\in Q$ and $u_0\in V$ is given as

$$\begin{aligned}(\partial_t u,\phi)_I+(\nabla u,\nabla\phi)_I&=(f+q,\phi)_I \quad \forall\phi\in X,\\ u(0)&=u_0 \qquad\qquad \text{in } \Omega.\end{aligned} \tag{2.2}$$

With the weak state equations (2.2), we define the weak formulation of the optimal control problem (2.1) as

$$\text{Minimize } J(q,u):=\frac{1}{2}\|u-\hat{u}\|_I^2+\frac{\alpha}{2}\|q\|_I^2 \text{ subject to (2.2) and } (q,u)\in Q\times X. \tag{2.3}$$

Now we cite two results of the problems (2.2) and (2.3).

Proposition 2.1 ([6])**.** *For fixed $q,f\in Q$, and $u_0\in V$ there exists a unique solution $u\in X$ of problem* (2.2)*. Moreover, the solution exhibits the improved regularity*

$$u\in L^2(I,H^2(\Omega)\cap V)\cap H^1(I,L^2(\Omega))\hookrightarrow C(\bar{I},V).$$

It holds the stability estimate

$$\|\partial_t u\|_I+\|\nabla^2 u\|_I\le C\{\|f+q\|_I+\|\nabla u_0\|\}.$$

Proposition 2.2 ([6]). *For given $f, \hat{u} \in L^2(I,H)$, $u_0 \in V$, and $\alpha > 0$, the optimal control problem (2.3) admits a unique solution $(\bar{q}, \bar{u}) \in Q \times X$. The optimal control $\bar{q}$ posesses the regularity*

$$\bar{q} \in L^2(I, H^2(\Omega)) \cap H^1(I, L^2(\Omega)).$$

Due to the existence and uniqueness results from Proposition 2.1, we define $u(q)$ as the unique solution of (2.2) with respect to some $q \in Q$. This enables us to define a reduced cost functional $j : Q \to \mathbb{R}$ that is only dependent on the control q as

$$j(q) := J(q, u(q)).$$

From now on, the optimal control problem that we examine is:

$$\text{minimize } j(q) \text{ subject to } q \in Q. \tag{2.4}$$

2.2 Finite element discretization

In order to solve the optimization problem (2.4) numerically, the discretization of our model is now discussed. We use finite elements methods [7], [8] for this. At first, the discretization in space with a n-D continuous Galerkin method is presented. Then, we look at the discretization in time, which is done with a 1D continuous Galerkin method. From now on, we will also discuss some implementation details, so, in this chapter, how we handle the calculation of the objective functional j. To solve the partial equations of (2.2), we use the Python package pyMOR.

2.2.1 Discretization in space

The discretization in space is shown on a 2-dimensional rectangular space $\Omega \subset \mathbb{R}^2$ with linear finite elements. We assume to have a vertex set $\mathcal{V} = (x_1, \ldots, x_{N_\mathcal{V}}) \in (\mathbb{R}^2)^{N_\mathcal{V}}$ with a convex hull that is equal to $\bar{\Omega}$ and $x_i \neq x_j$ for all $i \neq j$ in $\{1, \ldots, N_\mathcal{V}\}$. Let $\hat{T} = \{(x,y) \in [0,1]^2 \mid y \leq 1 - x\}$ be the reference triangle. Then,

$$\theta_l(\xi) = x_{l_1} + D\theta_l \begin{pmatrix} \xi_1 \\ \xi_2 \end{pmatrix} \text{ with } D\theta_l = \begin{pmatrix} x_{l_2} - x_{l_1} & x_{l_3} - x_{l_1} \end{pmatrix}$$

is a transformation from the reference triangle $\hat{T}$ to some other triangle T_l with the corners $x_{l_1}, x_{l_2}, x_{l_3} \in \mathcal{V}$.

We define now a mesh $\mathcal{T} = \{T_l\}$ which consists of triangles $T_l = \theta_l(\hat{T})$, where $T_l \cap T_m$ for $T_l, T_m \in \mathcal{T}$ is either a common side, a common corner, or empty, and where $\bar{\Omega} = \cup_{T_l \in \mathcal{T}} T_l$. We also assume that every vertex in $\mathcal{V}$ is a corner of at least one triangle of $\mathcal{T}$.

In the pyMOR implementation, we discretize a rectangular domain by specifying the number of grid intervals first. Then, the domain is subdivided into smaller rectangles of the same size, so that the number of rectangles along the x- and the y-axis is equal to the predefined number of grid intervals. Each smaller rectangular unit is then partitioned into four equally sized triangles by adding a vertex into the center of the rectangle which is connected with the corners of the unit. The vertex set of the whole domain is now given by the union of the corners of all triangles. As an example, if a domain $\Omega = [a, a]$ with $a > 0$ is given and we set the number of grid intervals to two, then our mesh would look like that:

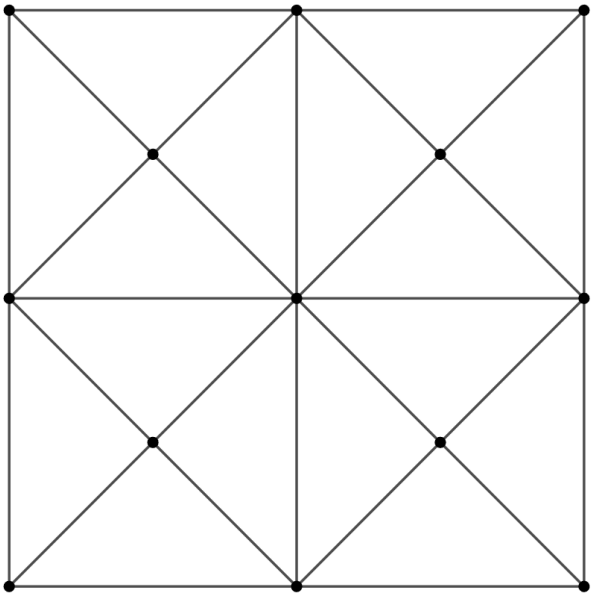

Figure 2.1: Example of a mesh with two grid intervals in a square-shaped domain. The dots denote elements of the vertex set $\mathcal{V}$.

Now, let $\mathcal{P}_1(\hat{T}, \mathbb{R})$ be the space of polynomials up to order one in $\hat{T}$. Then, $\{\psi_1, \psi_2, \psi_3\}$ with $\psi_1(\xi) = 1 - \xi_1 - \xi_2$, $\psi_2(\xi) = \xi_1$, $\psi_3(\xi) = \xi_2$ defines a basis of $\mathcal{P}_1(\hat{T}, \mathbb{R})$. Using this basis, we set

$$V_h = \operatorname{span}\{\phi_i, i = 0, \dots, N_\mathcal{V}\} \cap V$$

as the finite element space of our state variables with

$$\phi_i|_{T_l} = \begin{cases} 0 & \text{if } x_i \notin T_l \\ \psi_1 \circ \theta_l^{-1} & \text{if } \theta_l\left(\begin{pmatrix}0\\0\end{pmatrix}\right) = x_i \\ \psi_2 \circ \theta_l^{-1} & \text{if } \theta_l\left(\begin{pmatrix}1\\0\end{pmatrix}\right) = x_i \\ \psi_3 \circ \theta_l^{-1} & \text{if } \theta_l\left(\begin{pmatrix}0\\1\end{pmatrix}\right) = x_i \end{cases}$$

for all $T_l \in \mathcal{T}$ and $i = 1, \dots, N_\mathcal{V}$.

By construction, every $u \in V_h$ is uniquely defined by

$$u = \sum_{i=1}^{N_\mathcal{V}} U_i \phi_i \tag{2.5}$$

with $U_i = u(x_i)$.

Now we want to calculate $\int_\Omega u \cdot v \,\mathrm{d}x$ and $\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x$ for all $u, v \in V_h$. In order to do that, we set the mass matrix $\mathbf{M}_n = \left(\int_\Omega \phi_i \cdot \phi_j \,\mathrm{d}x\right)_{i,j=1,\dots,N_\mathcal{V}}$ and the stiffness matrix

$\mathbf{L}_n = \left(\int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}x \right)_{i,j=1,\dots,N_\mathcal{V}}$. Let

$$\mathbf{U} = \begin{pmatrix} U_1 \\ \vdots \\ U_{N_\mathcal{V}} \end{pmatrix} \text{ and } \mathbf{V} = \begin{pmatrix} V_1 \\ \vdots \\ V_{N_\mathcal{V}} \end{pmatrix},$$

where $U_i = u(x_i)$, $V_i = v(x_i)$ for $i = 1, \dots, N_\mathcal{V}$ as in (2.5). Then we have, based on the representation of elements in V_h from equation (2.5), that

$$\int_\Omega u \cdot v \, \mathrm{d}x = \mathbf{U}^T \mathbf{M}_n \mathbf{V} \text{ and } \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x = \mathbf{U}^T \mathbf{L}_n \mathbf{V}.$$

2.2.2 Discretization in time

For the discretization in time, we first partition the time interval $\bar{I} = [0, T]$ as

$$\bar{I} = \{0\} \cup I_1 \cup I_2 \cup \cdots \cup I_{N_t}$$

with subintervals $I_m = (t_{m-1}, t_m]$, where $t_m = m\dfrac{T}{N_t}$ for $m = 0, \dots, N_t$ and $N_t \in \mathbb{N}$. We want that the discretizations of our functions are continuous in $\bar{I}$ and piecewise polynomial of order one in all subintervals I_m, so the discretization space of our state variables is

$$X_{k,h} := \{v \in C(\bar{I}, V_h) \mid v|_{I_m} \in \mathcal{P}_1(I_m, V_h), m = 1, 2, \dots, N_t\},$$

where $\mathcal{P}_1(I_m, V_h)$ denotes the space of polynomials up to order one, defined on I_m with values in V_h. Similarly, we define the time-discretized space of our control variables as

$$Q_d := \{v \in C(\bar{I}, H) \mid v|_{I_m} \in \mathcal{P}_1(I_m, H), m = 1, 2, \dots, N_t\} \supset X_{k,h}.$$

By using the Lagrange basis of $\mathcal{P}_1(I_m, \mathbb{R})$, we can write every function $v \in Q_d$ as

$$v(t, x) = \left(m - t\frac{N_t}{T} \right) v_{m-1}(x) + \left(t\frac{N_t}{T} - m + 1 \right) v_m(x) \text{ for } t \in I_m, \tag{2.6}$$

where $v_m(x) = v(t_m, x)$.

2.2.3 Crank-Nicolson scheme

Now we solve the weak state equations (2.2) for the state $u \in X_{k,h}$, the control $q \in Q_d$, and $f \in Q$ numerically. Let $u_m = u(t_m, \cdot)$ and $U_{m,i} := u_m(x_i)$ for $m = 0, \dots, N_t$ and $i = 1, \dots, N_\mathcal{V}$. We set

$$\mathbf{U}_m = \begin{pmatrix} U_{m,1} \\ \vdots \\ U_{m,N_\mathcal{V}} \end{pmatrix}$$

for $m = 0, \dots, N_t$. The initial discretized state vector $\mathbf{U}_0$ is already given by the values of u_0. Next, we want to solve the weak state equations for $\mathbf{U}_m$ at the other time steps $m = 1, \dots, N_t$, so that we can set $u_m = \sum_{i=1}^{N_\mathcal{V}} U_{m,i} \phi_i$ according to equation (2.5).

For this, we find solutions of the Crank-Nicolson scheme [8], so that for $m = 1, \ldots, N_t$ and for all $v \in V_h$:

$$\begin{aligned}(u_m, v) + \frac{T}{2N_t}(\nabla u_m, \nabla v) &= (u_{m-1}, v) - \frac{T}{2N_t}(\nabla u_{m-1}, \nabla v) \\ &\quad + \frac{T}{2N_t}(f_{m-1} + q_{m-1}, v) + \frac{T}{2N_t}(f_m + q_m, v),\end{aligned}$$

where $f_m = f(t_m, \cdot)$ and $q_m = q(t_m, \cdot)$. To solve the above equation, while considering the homogeneous Dirichlet boundary conditions, we define the matrix $\tilde{\mathbf{M}}_n \in \mathbb{R}^{N_\mathcal{V} \times N_\mathcal{V}}$ as

$$\left(\tilde{\mathbf{M}}_n\right)_{i,j} = \begin{cases} 0 & \text{if } x_i \text{ or } x_j \text{ in } \partial\Omega \text{ and } i \neq j \\ 1 & \text{if } x_i \text{ or } x_j \text{ in } \partial\Omega \text{ and } i = j \\ (\mathbf{M}_n)_{i,j} & \text{else} \end{cases}$$

and the matrix $\tilde{\mathbf{L}}_n \in \mathbb{R}^{N_\mathcal{V} \times N_\mathcal{V}}$ as

$$\left(\tilde{\mathbf{L}}_n\right)_{i,j} = \begin{cases} 0 & \text{if } x_j \text{ in } \partial\Omega \\ (\mathbf{L}_n)_{i,j} & \text{else,} \end{cases}$$

so that $(u_m, v) = \mathbf{U}_m^T \tilde{\mathbf{M}}_n \mathbf{V}$ and $(\nabla u_m, \nabla v) = \mathbf{U}_m^T \tilde{\mathbf{L}}_n \mathbf{V}$ for all $m = 0, \ldots, N_t$, which is giving us

$$\begin{aligned}\mathbf{V}^T \tilde{\mathbf{M}}_n^T \mathbf{U}_m + \frac{T}{2N_t} \mathbf{V}^T \tilde{\mathbf{L}}_n^T \mathbf{U}_m &= \mathbf{V}^T \tilde{\mathbf{M}}_n^T \mathbf{U}_{m-1} - \frac{T}{2N_t} \mathbf{V}^T \tilde{\mathbf{L}}_n^T \mathbf{U}_{m-1} \\ &\quad + \frac{T}{2N_t}(f_{m-1} + q_{m-1}, v) + \frac{T}{2N_t}(f_m + q_m, v).\end{aligned}$$

In the pyMOR implementation, vectors $\mathbf{F}_m$ for $m = 0, \ldots, N_t$ are defined such that $\mathbf{V}^T \mathbf{F}_m \approx (f_m + q_m, v)$ for all $v \in \mathbf{V}_h$ and $(\mathbf{F}_m)_i = 0$ if the i-th entry in the vertex set $\mathcal{V}$ lies on the boundary of Ω. By using these vectors, we get the equation

$$\left(\tilde{\mathbf{M}}_n^T + \frac{T}{2N_t}\tilde{\mathbf{L}}_n^T\right) \mathbf{U}_m = \left(\tilde{\mathbf{M}}_n^T - \frac{T}{2N_t}\tilde{\mathbf{L}}_n^T\right) \mathbf{U}_{m-1} + \frac{T}{2N_t}\mathbf{F}_{m-1} + \frac{T}{2N_t}\mathbf{F}_m, \tag{2.7}$$

which is solved after $\mathbf{U}_m$ with algorithms from the Python package SciPy.

2.2.4 Calculation of the objective function value

For fixed $\hat{u}, f \in Q$, we define $u = u(q)$ for all $q \in Q_d$, so that it satisfies (2.7). We calculate $j(q)$ now in the following way:

$$\begin{aligned}j(q) \approx \quad & \frac{1}{2} \sum_{m=1}^{N_t} \int_{t_{m-1}}^{t_m} \quad \Bigg(\left(m - t\frac{N_t}{T}\right)(u_{m-1} - \hat{u}_{m-1}) + \left(t\frac{N_t}{T} - m + 1\right)(u_m - \hat{u}_m), \\ & \qquad \left(m - t\frac{N_t}{T}\right)(u_{m-1} - \hat{u}_{m-1}) + \left(t\frac{N_t}{T} - m + 1\right)(u_m - \hat{u}_m) \Bigg) \, \mathrm{d}t \\ + & \frac{\alpha}{2} \sum_{m=1}^{N_t} \int_{t_{m-1}}^{t_m} \quad \Bigg(\left(m - t\frac{N_t}{T}\right) q_{m-1} + \left(t\frac{N_t}{T} - m + 1\right) q_m, \\ & \qquad \left(m - t\frac{N_t}{T}\right) q_{m-1} + \left(t\frac{N_t}{T} - m + 1\right) q_m \Bigg) \, \mathrm{d}t,\end{aligned}$$

where $\hat{u}_m = \hat{u}(t_m, \cdot)$. Integration by substitution yields

$$\begin{aligned} j(q) \approx \quad & \frac{T}{6N_t} \sum_{m=1}^{N_t} \quad (u_{m-1} - \hat{u}_{m-1}, u_{m-1} - \hat{u}_{m-1}) + (u_{m-1} - \hat{u}_{m-1}, u_m - \hat{u}_m) \\ & \qquad + (u_m - \hat{u}_m, u_m - \hat{u}_m) \\ & + \frac{\alpha T}{6N_t} \sum_{m=1}^{N_t} \quad (q_{m-1}, q_{m-1}) + (q_{m-1}, q_m) + (q_m, q_m) \\ \approx \quad & \frac{T}{6N_t} \sum_{m=1}^{N_t} \quad \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \mathbf{M}_n \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \\ & \qquad + \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \mathbf{M}_n \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \\ & \qquad + \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \mathbf{M}_n \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \\ & + \frac{\alpha T}{6N_t} \sum_{m=1}^{N_t} \quad \mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_{m-1} + \mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_m + \mathbf{Q}_m\mathbf{M}_n\mathbf{Q}_m, \end{aligned}$$

where $\hat{\mathbf{U}}_m = (\hat{u}(t_m, x_i))_{i=1,\dots,N_\mathcal{V}}$ and $\mathbf{Q}_m = (q(t_m, x_i))_{i=1,\dots,N_\mathcal{V}}$ for $m = 0, \dots, N_t$.

2.3 Optimization of the control variable

To optimize the control variable, we write every $q \in Q_d$, by using a fixed set of $N_b \in \mathbb{N}$ shape functionals

$$\Phi = \{\phi_1, \dots, \phi_{N_b}\} \tag{2.8}$$

with $\phi_1, \dots, \phi_{N_b} \in H$ and scalars $q_1^0, q_1^1, \dots, q_1^{N_t}, \dots, q_{N_b}^0, q_{N_b}^1 \dots, q_{N_b}^{N_t} \in \mathbb{R}$, as a linear combination of the shape functionals in Φ with time-dependent factors. That is

$$q(t, x) = \sum_{i=1}^{N_b} \alpha_i(t)\phi_i(x), \tag{2.9}$$

where the factors $\alpha_i(t)$ are for $t \in I_m$ a weighted sum between q_i^{m-1} and q_i^m with $m = 1, \dots, N_t$ as in (2.6):

$$\alpha_i(t) = \begin{cases} q_i^{m-1}\left(m - t\frac{N_t}{T}\right) + q_i^m\left(t\frac{N_t}{T} - m + 1\right) & \text{if } t \in I_m \text{ with } m = 1, \dots, N_t \\ q_i^0 & \text{if } t = 0. \end{cases}$$

Each control variable that is written in this form can be represented by a vector

$$\mathbf{q} = \left[q_1^0, q_1^1, \dots, q_1^{N_t}, \dots, q_{N_b}^0, q_{N_b}^1 \dots, q_{N_b}^{N_t}\right]^T \in \mathcal{D},$$

where $\mathcal{D} := \mathbb{R}^{N_\mathbf{q}}$ with $N_\mathbf{q} = (N_t + 1) \cdot N_b$ is the domain of the control vector. Hence, we write

$$j(\mathbf{q}) := j(q) \tag{2.10}$$

for each q that is defined like in (2.9). In the remaining chapters, we want to find a $\mathbf{q} \in \mathcal{D}$ that minimizes this functional. That is

$$\underset{\mathbf{q}\in\mathcal{D}}{\text{minimize}}\, j(\mathbf{q}). \tag{2.11}$$

In this thesis, we refer to evaluations of the functional j in (2.10) with

$$\begin{aligned} j(q) = \frac{T}{6N_t} \sum_{m=1}^{N_t} & \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \mathbf{M}_n \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \\ & + \left(\mathbf{U}_{m-1} - \hat{\mathbf{U}}_{m-1}\right) \mathbf{M}_n \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \\ & + \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \mathbf{M}_n \left(\mathbf{U}_m - \hat{\mathbf{U}}_m\right) \\ + \frac{\alpha T}{6N_t} \sum_{m=1}^{N_t} & \mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_{m-1} + \mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_m + \mathbf{Q}_m\mathbf{M}_n\mathbf{Q}_m, \end{aligned}$$

as full order model (FOM) functional evaluations. The optimization problem in (2.11) is called the FOM optimization problem. In the next chapters, we present algorithms that solve the FOM optimization problem. Thus, the FOM functional is also the objective functional.

3 Ensemble-based optimization algorithm

The adaptive ensemble-based optimization (EnOpt) algorithm is often used to maximize the net present value of oil recovery methods with respect to a control vector. Examples are presented in [4], [9], [10]. In this chapter, we want to utilize the EnOpt algorithm to optimize the objective functional j. Our implementation is similar to that in [4].

In the following, the EnOpt algorithm is presented for a general functional $F : \mathbb{R}^{N_\mathbf{q}} \to \mathbb{R}$ to iteratively solve the optimization problem

$$\underset{\mathbf{q}\in\mathcal{D}}{\text{maximize}}\, F(\mathbf{q}).$$

We start at an initialization $\mathbf{q}_0$, which is updated iteratively with a preconditioned gradient ascent method that is given by

$$\mathbf{q}_{k+1} = \mathbf{q}_k + \beta_k \mathbf{d}_k,$$
$$\mathbf{d}_k \approx \frac{\mathbf{C}_k \mathbf{G}_k}{\|\mathbf{C}_k \mathbf{G}_k\|_\infty},$$

where $k = 0, 1, 2, \ldots$ denotes the iteration of the optimization. The step size β_k, with $0 < \beta_k \leq 1$, is computed by using a line search procedure. We mentioned in the introduction that we use a sample around the iterate $\mathbf{q}_k$ to calculate the next iterate $\mathbf{q}_{k+1}$. The matrix $\mathbf{C}_k \in \mathbb{R}^{N_\mathbf{q} \times N_\mathbf{q}}$ denotes the covariance matrix of this sample set at the k-th iteration. The vector $\mathbf{G}_k \in \mathbb{R}^{N_\mathbf{q}}$ is the gradient of F at the iterate $\mathbf{q}_k$. So in this definition, the step direction $\mathbf{d}_k$ is approximately the gradient of the objective functional, preconditioned with the covariance matrix $\mathbf{C}_k$ and normalized by dividing by the maximum norm of itself. Since we do not have access to the gradient of F, we use an estimator of $\mathbf{C}_k\mathbf{G}_k/\|\mathbf{C}_k\mathbf{G}_k\|_\infty$ as the step direction. A more detailed description of the EnOpt algorithm is shown next.

We define the initial covariance matrix $\mathbf{C}_0$ so that the covariance between controls of different shape functionals $\phi_i, \phi_j \in \Phi$ from (2.8) is zero. That means, we have for $i, j = 1, \ldots, N_b$ with $i \neq j$:

$$\mathrm{Cov}(q_i^m, q_j^n) = 0, \text{ for all } m, n \in \{0, \ldots, N_t\}.$$

We expect controls of the same shape functional $\phi_j \in \Phi$ to correlate. Based on that, we set the controls for $j = 1, \ldots, N_b$ and $m = 0, \ldots, N_t$ as

$$\mathrm{Cov}(q_j^m, q_j^{m+h}) = \sigma_j^2 \rho^h \left(\frac{1}{1-\rho^2}\right), \text{ for all } h \in \{0, \ldots, N_t - i\}, \tag{3.1}$$

where $\sigma_j^2 > 0$ is the variance for the basis functional ϕ_j and $\rho \in (-1, 1)$ the correlation coefficient. This defines the covariances between all controls since $\mathrm{Cov}(q_j^m, q_j^{m+h}) = \mathrm{Cov}(q_j^{m+h}, q_j^m)$. By adjusting the variance and correlation coefficient, we can influence how the controls of the same shape functional depend on each other.

For example, if ρ is close to 1, then the covariance of the controls is large, resulting in them to show a similar behaviour. That means, if a control variable of a shape functional at time step m is large, then the control variables of this shape functional also tend to be large at time steps close to m. Analogously, if a control variable of a shape functional at time step m is small, then the control variables of this shape functional at time steps close to m have a tendency to be small, too.

If ρ is close to -1, then the covariance is large when the h in (3.1) is even and small, and the covariance is negative if h is odd. This leads to high oscillations of the control variables.

Now that we have the covariances between the individual controls, we can set the initial covariance matrix. For that, let the matrices $\mathbf{C}_0^j \in \mathbb{R}^{(N_t+1)\times(N_t+1)}$ for $j = 1, \dots, N_b$ be defined as $\mathbf{C}_0^j := \left(\mathrm{Cov}(q_j^i, q_j^k)\right)_{i,k=0,\dots,N_t}$. Then the initial covariance matrix is

$$\mathbf{C}_0 = \begin{pmatrix} \mathbf{C}_0^1 & 0 & \cdots & 0 \\ 0 & \mathbf{C}_0^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{C}_0^{N_b} \end{pmatrix}. \tag{3.2}$$

To compute the step direction $\mathbf{d}_k$ at iteration step k, we sample $\mathbf{q}_{k,m} \in \mathcal{D}$ for $m = 1, \dots, N$, with $N \in \mathbb{N}$, from a multivariate Gaussian distribution with the mean $\mathbf{q}_k$ and the covariance $\mathbf{C}_k$. Then we define the vector $\mathbf{C}_F^k \in \mathbb{R}^{N_\mathbf{q}}$ with these samples and the current iterate $\mathbf{q}_k$ as

$$\mathbf{C}_F^k := \frac{1}{N-1} \sum_{m=1}^{N} (\mathbf{q}_{k,m} - \mathbf{q}_k)(F(\mathbf{q}_{k,m}) - F(\mathbf{q}_k)). \tag{3.3}$$

Now we set the step direction as $\mathbf{d}_k = \dfrac{\mathbf{C}_F^k}{\|\mathbf{C}_F^k\|_\infty}$. This is valid since $\mathbf{C}_F^k$ is an estimation of $\mathbf{C}_k\mathbf{G}_k$, which can by shown like in [9]. Here, we begin with the Taylor expansion around $\mathbf{q}_k$ and get for some $\mathbf{q} \in \mathcal{D}$

$$\begin{aligned} & F(\mathbf{q}) = F(\mathbf{q}_k) + (\mathbf{q} - \mathbf{q}_k)^T \nabla F(\mathbf{q}_k) + O(\|\mathbf{q} - \mathbf{q}_k\|^2) \\ \Longrightarrow \quad & F(\mathbf{q}) - F(\mathbf{q}_k) = (\mathbf{q} - \mathbf{q}_k)^T \mathbf{G}_k + O(\|\mathbf{q} - \mathbf{q}_k\|^2). \end{aligned}$$

Multiplying both sides by $(\mathbf{q} - \mathbf{q}_k)$ and setting $\mathbf{q} = \mathbf{q}_{k,m}$ yields

$$\begin{aligned} & (\mathbf{q}_{k,m} - \mathbf{q}_k)(F(\mathbf{q}_{k,m}) - F(\mathbf{q}_k)) \\ = \quad & (\mathbf{q}_{k,m} - \mathbf{q}_k)(\mathbf{q}_{k,m} - \mathbf{q}_k)^T \mathbf{G}_k + O(\|\mathbf{q}_{k,m} - \mathbf{q}_k\|^3), \end{aligned}$$

where $O(\|\mathbf{q}_{k,m} - \mathbf{q}_k\|^3)$ are the remaining terms containing order ≥ 3 of $(\mathbf{q}_{k,m} - \mathbf{q}_k)$. Neglecting $O(\|\mathbf{q}_{k,m} - \mathbf{q}_k\|^3)$ gives by summation over all samples and multiplication of both

sides with $\frac{1}{N-1}$:

$$\begin{aligned} & \frac{1}{N-1}\sum_{m=1}^{N}(\mathbf{q}_{k,m}-\mathbf{q}_k)(F(\mathbf{q}_{k,m})-F(\mathbf{q}_k)) \\ \approx \quad & \left(\frac{1}{N-1}\sum_{m=1}^{N}(\mathbf{q}_{k,m}-\mathbf{q}_k)(\mathbf{q}_{k,m}-\mathbf{q}_k)^T\right)\mathbf{G}_k \\ \Longrightarrow \quad & \mathbf{C}_F^k \approx \mathbf{C}_k\mathbf{G}_k, \end{aligned}$$

since $\frac{1}{N-1}\sum_{m=1}^{N}(\mathbf{q}_{k,m}-\mathbf{q}_k)(\mathbf{q}_{k,m}-\mathbf{q}_k)^T$ is itself an approximation of $\mathbf{C}_k$.

By using the samples $\{\mathbf{q}_{k-1,m}\}_{m=1}^N$ and the covariance matrix $\mathbf{C}_{k-1}$ from the last iteration, we update $\mathbf{C}_{k-1}$, like in [3], by setting

$$\mathbf{C}_k = \mathbf{C}_{k-1} + \tilde{\beta}_k\tilde{\mathbf{d}}_k \text{ with} \tag{3.4}$$

$$\tilde{\mathbf{d}}_k = N^{-1}\sum_{m=1}^{N}(F(\mathbf{q}_{k-1,m})-F(\mathbf{q}_k))((\mathbf{q}_{k-1,m}-\mathbf{q}_k)(\mathbf{q}_{k-1,m}-\mathbf{q}_k)^T-\mathbf{C}_{k-1}), \tag{3.5}$$

where $\tilde{\beta}_k$ is a step size that is chosen so that no entries of the diagonal of $\mathbf{C}_k$ are negative. How we set $\tilde{\beta}_k$ is shown below at the implementation of the entire EnOpt algorithm. We expect to get better samples in the next iteration due to the updated covariance matrix. Therefore, we call this procedure the adaptive EnOpt algorithm.

Now that we have described the optimization steps of this algorithm, we iterate until $F(\mathbf{q}_k) \leq F(\mathbf{q}_{k-1}) + \varepsilon$, where $\varepsilon > 0$.

Next, we describe our implementation of the EnOpt algorithm. The EnOpt algorithm takes the objective function $\mathrm{F} : \mathbb{R}^{N_\mathbf{q}} \to \mathbb{R}$, our initial iterate $\mathbf{q}_0 \in \mathcal{D}$, the sample size $N \in \mathbb{N}$, the tolerance $\varepsilon > 0$, the maximum number of iterations $k^* \in \mathbb{N}$, the initial step size $\beta_1 > 0$ for the computation of the next iterate, the initial step size $\beta_2 > 0$ for the update of the covariance matrix, the step size contraction $r \in (0,1)$, the maximum number of step size trials $\nu^* \in \mathbb{N}$, the variance $\sigma^2 \in \mathbb{R}^{N_b}$ with positive elements, the correlation coefficient $\rho \in (-1,1)$, the number of time steps $N_t \in \mathbb{N}$, the number of basis functions $N_b \in \mathbb{N}$, a projection PR and an initial covariance $\mathbf{C}_{\text{init}} \in \mathbb{R}^{N_\mathbf{q}\times N_\mathbf{q}}$. PR is set to the identity function **lambda** mu : mu and $\mathbf{C}_{\text{init}}$ is set to None if they are not specified.

PR is used to project the inputs onto a given set and $\mathbf{C}_{\text{init}}$ is an alternative definition for the initialization of the covariance. In this chapter we do not need to specify these inputs, however PR could be used if we had an optimization problem where the iterates were restricted to a certain spatial domain, which is here not the case.

The implementation in pseudo code is shown below:

Algorithm 1 EnOpt algorithm

1: **function** EnOpt($\mathrm{F}, \mathbf{q}_0, N, \varepsilon, k^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b, \mathrm{PR} \leftarrow$ **lambda** mu : mu, $\mathbf{C}_{\text{init}} \leftarrow$ None)
2: $\quad F_k^{\text{prev}} \leftarrow \mathrm{F}(\mathbf{q}_0)$
3: $\quad \mathbf{q}_k, T_k, \mathbf{C}_k, F_k \leftarrow$ OptStep($\mathrm{F}, \mathbf{q}_0, N, 0, [\,], \mathbf{C}_{\text{init}}, F_k^{\text{prev}}, \beta_1, \beta_2, r, \varepsilon, \nu^*, \sigma^2, \rho, N_t, N_b, \mathrm{PR}$)
4: $\quad k \leftarrow 1$
5: $\quad$ **while** $F_k > F_k^{\text{prev}} + \varepsilon$ and $k < k^*$ **do**
6: $\quad\quad F_k^{\text{prev}} \leftarrow F_k$
7: $\quad\quad \mathbf{q}_k, T_k, \mathbf{C}_k, F_k \leftarrow$ OptStep($\mathrm{F}, \mathbf{q}_k, N, k, T_k, \mathbf{C}_k, F_k, \beta_1, \beta_2, r, \varepsilon, \nu^*, \sigma^2, \rho, N_t, N_b, \mathrm{PR}$)
8: $\quad\quad k \leftarrow k + 1$
9: $\quad$ **return** $\mathbf{q}_k$

We begin by initializing F_k^{prev} as $\mathrm{F}(\mathbf{q}_0)$. The next iterate $\mathbf{q}_k$, along with the sample T_k, the covariance $\mathbf{C}_k$ and the functional value of $\mathbf{q}_k$, denoted by F_k, are computed by calling the initial optimization step OptStep, which is shown in algorithm 2.

After we initialized k as 1, we enter a while-loop until the stop criterion is satisfied. That is, we stop the loop when the functional value F_k of the current iterate is no longer significantly larger than the functional value F_k^{prev} of the last iterate, so when $F_k \leq F_k^{\text{prev}} + \varepsilon$. There is also a limit k^* to the number of loop repetitions, so when that is reached, we also stop the while-loop. In each optimization loop, F_k^{prev} is set to the functional value of the last iterate and $\mathbf{q}_k, T_k, \mathbf{C}_k$ and F_k are updated by calling OptStep again. After we leave the while-loop, the last iterate $\mathbf{q}_k$ is returned.

The next algorithms use some functions from Python such as len for the length of a list or an array, as well as functions from the Python package NumPy, which are identified by starting with 'np.'.

Algorithm 2 OptStep algorithm

function OptStep(F, $\mathbf{q}_k, N, k, T_k, \mathbf{C}_k, F_k, \beta_1, \beta_2, r, \varepsilon, \nu^*, \sigma^2, \rho, N_t, N_b$, PR $\leftarrow$ **lambda** mu : mu)
  $N_{\mathbf{q}} \leftarrow$ len($\mathbf{q}_k$)
  $\mathbf{C}_k^{\text{next}} \leftarrow$ np.zeros($(N_{\mathbf{q}}, N_{\mathbf{q}})$)
  if $k = 0$ **then**
    if $\mathbf{C}_k$ is None **then**
      $\mathbf{C}_k^{\text{next}} \leftarrow$ initCov(len($\mathbf{q}_k$), σ^2, ρ, N_t, N_b)
    else
      $\mathbf{C}_k^{\text{next}} \leftarrow \mathbf{C}_k$.copy( )
  else
    $\mathbf{C}_k^{\text{next}} \leftarrow$ updateCov($\mathbf{q}_k, T_k, \mathbf{C}_k, F_k, \beta_2$)
  sample $\leftarrow$ np.random.multivariate_normal($\mathbf{q}_k, \mathbf{C}_k^{\text{next}}$, size $\leftarrow N$)
  $T_k^{\text{next}} \leftarrow [\text{PR}(\text{sample}[j]), \text{F}(\text{PR}(\text{sample}[j]))]_{j=0}^{N-1}$
  $\mathbf{C}_F^k \leftarrow$ np.zeros($N_{\mathbf{q}}$)
  for $m = 0, \ldots, N-1$ **do**
    $\mathbf{C}_F^k \leftarrow \mathbf{C}_F^k + (T_k^{\text{next}}[m][0] - \mathbf{q}_k) \cdot (T_k^{\text{next}}[m][1] - F_k)$
  $\mathbf{C}_F^k \leftarrow 1/(N-1) \cdot \mathbf{C}_F^k$
  $\mathbf{d}_k \leftarrow$ np.zeros($N_{\mathbf{q}}$)
  $\mathbf{q}_k^{\text{next}}, F_k^{\text{next}} \leftarrow$ np.copy($\mathbf{q}_k$), F_k
  if not np.all($\mathbf{C}_F^k == 0$) **then**
    $\mathbf{d}_k \leftarrow \mathbf{C}_F^k$/np.max(np.abs($\mathbf{C}_F^k$))
    $\mathbf{q}_k^{\text{next}}, F_k^{\text{next}} \leftarrow$ LineSearch(F, $\mathbf{q}_k, F_k, \mathbf{d}_k, \beta_1, r, \varepsilon, \nu^*$, PR)
  return $\mathbf{q}_k^{\text{next}}, T_k^{\text{next}}, \mathbf{C}_k^{\text{next}}, F_k^{\text{next}}$

We start the OptStep algorithm by updating the covariance matrix $\mathbf{C}_k$ or, if k is zero, initializing it. In the case that k is zero, we first check if there is a predefined covariance matrix $\mathbf{C}_k$. When there is one, $\mathbf{C}_k^{\text{next}}$ is set to that matrix. Otherwise, we define $\mathbf{C}_k^{\text{next}}$ by calling the function initCov, which sets the matrix like it is described in (3.2).

If k is not zero, we get the updated covariance matrix by calling updateCov, which is described in (3.4) and algorithm 3.

The covariance matrix $\mathbf{C}_k^{\text{next}}$ is now used to get a Gaussian distributed sample with N elements around $\mathbf{q}_k$. The projected samples PR(sample[j]) and their respective function values F(PR(sample[j])) for $j = 0, \ldots, N-1$ are stored in the list T_k^{next}. The definition is here abbreviated as $[\text{PR}(\text{sample}[j]), \text{F}(\text{PR}(\text{sample}[j]))]_{j=0}^{N-1}$. In the code, this is done with an iteration through a for-loop.

After that, we set $\mathbf{C}_F^k$ like it is defined in (3.3). Then we check if $\mathbf{C}_F^k$ is equal to the zero vector. That can happen if the functional F is constant on the sample set. In that case, we let the next iterate $\mathbf{q}_k^{\text{next}}$ and the corresponding functional value F_k unchanged.

Otherwise, we set $\mathbf{d}_k$ to $\mathbf{C}_F^k$ divided by its maximum norm np.max(np.abs($\mathbf{C}_F^k$)) $= \|\mathbf{C}_F^k\|_\infty$. The next iterate $\mathbf{q}_k^{\text{next}}$ and its functional value F_k^{next} is now computed with the line search algorithm LineSearch, that is shown in algorithm 4.

We return $\mathbf{q}_k^{\text{next}}, T_k^{\text{next}}, \mathbf{C}_k^{\text{next}}$ and F_k^{next}.

Algorithm 3 Covariance matrix update

1: **function** UPDATECOV($\mathbf{q}_k, T_k, \mathbf{C}_k, F_k, \beta_2$)
2: $N_{\mathbf{q}} \leftarrow$ LEN($\mathbf{q}_k$)
3: $N \leftarrow$ LEN(T_k)
4: $\mathbf{d}_k^{\text{cov}} \leftarrow$ NP.ZEROS($(N_{\mathbf{q}}, N_{\mathbf{q}})$)
5: **for** $m = 0, \ldots, N-1$ **do**
6: $\mathbf{d}_k^{\text{cov}} \leftarrow \mathbf{d}_k^{\text{cov}} + (T_k[m][1] - F_k) \cdot ((T_k[m][0] - \mathbf{q}_k)$.RESHAPE$((N_{\mathbf{q}}, 1)) \cdot (T_k[m][0] - \mathbf{q}_k)$.RESHAPE$((1, N_{\mathbf{q}})) - \mathbf{C}_k)$
7: $\mathbf{d}_k^{\text{cov}} \leftarrow \mathbf{d}_k^{\text{cov}} / N$
8: $\mathbf{C}_{\text{diag}}, \mathbf{d}_{\text{diag}} \leftarrow$ np.ZEROS$(N_{\mathbf{q}})$, np.ZEROS$(N_{\mathbf{q}})$
9: **for** $i = 0, \ldots, N_{\mathbf{q}} - 1$ **do**
10: $\mathbf{C}_{\text{diag}}[i] \leftarrow \mathbf{C}_k[i, i]$
11: $\mathbf{d}_{\text{diag}}[i] \leftarrow \mathbf{d}_k^{\text{cov}}[i, i]$
12: $\beta_2^{\text{iter}} \leftarrow \beta_2$
13: **while** np.MIN$(\mathbf{C}_{\text{diag}} + \beta_2^{\text{iter}} \cdot \mathbf{d}_{\text{diag}}) \leq 0$ **do**
14: $\beta_2^{\text{iter}} \leftarrow \beta_2^{\text{iter}} / 2$
15: **return** $\mathbf{C}_k + \beta_2^{\text{iter}} * \mathbf{d}_k^{\text{cov}}$

For the update of the covariance matrix, we calculate the step direction $\mathbf{d}_k^{\text{cov}}$ like in 3.5 first. Then we want to make sure that the updated covariance matrix has only positive values on its diagonal.

For this purpose, $\mathbf{C}_{\text{diag}}$ and $\mathbf{d}_{\text{diag}}$ are defined from line 8 to line 11 as vectors whose elements are the diagonal values of $\mathbf{C}_k$ and $\mathbf{d}_k^{\text{cov}}$ respectively. Then, the step size β_2^{iter}, initialized as β_2, is halfed until np.MIN$(\mathbf{C}_{\text{diag}} + \beta_2^{\text{iter}} * \mathbf{d}_{\text{diag}})$ is positive. Finally, the matrix $\mathbf{C}_k + \beta_2^{\text{iter}} * \mathbf{d}_k^{\text{cov}}$, like in (3.4), is returned.

Algorithm 4 Line search

1: **function** LINESEARCH(F, $\mathbf{q}_k, F_k, \mathbf{d}_k, \beta_1, r, \varepsilon, \nu^*$, PR)
2: $\beta_1^{\text{iter}} \leftarrow \beta_1$
3: $\mathbf{q}_k^{\text{next}} \leftarrow \text{PR}(\mathbf{q}_k + \beta_1^{\text{iter}} \mathbf{d}_k)$
4: $F_k^{\text{next}} \leftarrow \text{F}(\mathbf{q}_k^{\text{next}})$
5: $\nu \leftarrow 0$
6: **while** $F_k^{\text{next}} - F_k \leq \varepsilon$ and $\nu < \nu^*$ **do**
7: $\beta_1^{\text{iter}} \leftarrow r \beta_1^{\text{iter}}$
8: $\mathbf{q}_k^{\text{next}} \leftarrow \text{PR}(\mathbf{q}_k + \beta_1^{\text{iter}} \mathbf{d}_k)$
9: $F_k^{\text{next}} \leftarrow \text{F}(\mathbf{q}_k^{\text{next}})$
10: $\nu \leftarrow \nu + 1$
return $\mathbf{q}_k^{\text{next}}, F_k^{\text{next}}$

At the start of the line search algorithm, we initialize the step size β_1^{iter} as β_1. Then we repeatedly calculate $\mathbf{q}_k^{\text{next}} = \text{PR}(\mathbf{q}_k + \beta_1^{\text{iter}} \mathbf{d}_k)$ and $F_k^{\text{next}} = \text{F}(\mathbf{q}_k^{\text{next}})$ with simultaneous reduction of β_1^{iter} by multiplication with r in a while-loop until either $F_k^{\text{next}} - F_k > \varepsilon$ or $\nu \geq \nu^*$. After the termination of the while-loop, $\mathbf{q}_k^{\text{next}}$ and F_k^{next} are returned.

The cause for stopping the while-loop also shows when $\mathbf{q}_k^{\text{next}}$ is the last iterate of the EnOpt algorithm, as the EnOpt algorithm stops when the function value of the next iteration is not greater than the function value of the last iteration plus ε, i.e. when

$F_k^{\text{next}} \leq F_k + \varepsilon$. Hence, if the while-loop in algorithm 4 terminates because $\nu \geq \nu^*$ while $F_k^{\text{next}} - F_k \leq \varepsilon$ still holds, then we know already that the while-loop in algorithm 1 will also stop after this iteration.

Now we use the adaptive EnOpt algorithm to optimize our FOM objective functional j from (2.10). Since this is a maximization procedure and j should be minimized, we apply $-j$, denoted by $-\text{J}$, to the EnOpt algorithm, which gives us algorithm 5. We call this the FOM-EnOpt algorithm because the FOM functional j is applied to the EnOpt algorithm 1.

Algorithm 5 FOM-EnOpt algorithm

1: **function** FOM-EnOpt($\mathbf{q}_0, N, \varepsilon, k^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, \mathbf{q}_{\text{base}}$)
2: **return** EnOpt($-\text{J}, \mathbf{q}_0, N, \varepsilon, k^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, \text{LEN}(\mathbf{q}_{\text{base}})$)

$\mathbf{q}_{\text{base}}$ is here a list that consists of the shape functions, so Φ in (2.8). There are some more inputs that FOM-EnOpt requires, but we omit these as they are only needed for the calculation of J.

Before we move on to the Adaptive-ML-EnOpt method, we want to discuss the computational effort of running the FOM-EnOpt algorithm. Most computations during the FOM-EnOpt procedure require only basic operations. The most expensive calculations are calls of the FOM functional j. In algorithm 1, the only computation of the FOM objective functional value is in line 2. The other objective functional values get returned by calls of the optimization step procedure which is algorithm 2. Here, the FOM functional value is calculated N times in line 12 and also possibly multiple times in line 21 when the line search procedure, algorithm 4, is called. In algorithm 4, the functional value of j is calculated once in line 4 and up to ν^* times in line 9. Updating the covariance matrix in algorithm 3 does not require any FOM computations since all necessary objective functional values are already contained in T_k.

We note that calls of the OptStep algorithm on the FOM functional j are expensive. In the next chapter, we introduce an algorithm that replaces most of these calls by computations of the EnOpt algorithm on a surrogate functional. This surrogate is a machine learning-based approximation of j around the current iterate.

4 Adaptive-ML-EnOpt algorithm

In this chapter, we describe the Adaptive-ML-EnOpt algorithm [4], which is a modified version of the EnOpt algorithm. This algorithm is supposed to reduce the number of FOM evaluations by using a machine learning-based surrogate functional, which improves the computation speed with respect to the EnOpt algorithm. Therefore, we introduce deep neural networks (DNNs) next. After that, the Adaptive-ML-EnOpt-algorithm is presented.

4.1 Deep neural networks

This description of deep neural networks is based on the definitions in [4].

DNNs are used here to approximate a function $f : \mathbb{R}^{N_{\text{in}}} \to \mathbb{R}^{N_{\text{out}}}$ with $N_{\text{in}}, N_{\text{out}} \in \mathbb{N}$. We call $L \in \mathbb{N}$ the number of layers and $N_{\text{in}} = N_0, N_1, \ldots, N_{L-1}, N_L = N_{\text{out}}$ the numbers of neurons in each layer. We refer to the layers 1 to $L-1$ as the hidden layers. $W_i \in \mathbb{R}^{N_i \times N_{i-1}}$ denotes the weights in layer $i \in \{1, \ldots, L\}$ and $b_i \in \mathbb{R}^{N_i}$ the biases of the layer $i \in \{1, \ldots, L\}$. These are composed as $\mathbf{W} = ((W_1, b_1), \ldots, (W_L, b_L))$, which is a tuple of pairs of corresponding weights and biases.

$\rho : \mathbb{R} \to \mathbb{R}$ is the so-called activation function. A popular example is the rectified linear unit funtion $\rho(x) = \max(x, 0)$, however we will use the hyperbolic tangent funtion

$$\rho(x) = \tanh(x) = \frac{\exp(2x) - 1}{\exp(2x) + 1}.$$

$\rho_n^* : \mathbb{R}^n \to \mathbb{R}^n$ is now defined as the component-wise application of ρ onto a vector of dimension n, so $\rho_n^*(x) = [\rho(x_1), \ldots, \rho(x_n)]^T$ for $x \in \mathbb{R}^n$.

To calculate the output $\Phi_{\mathbf{W}}(x) \in \mathbb{R}^{N_{\text{out}}}$ of a DNN for an input $x \in \mathbb{R}^{N_{\text{in}}}$, we apply the weights, biases, and activation function multiple times onto the input. It is computed iteratively with the functions $r_0 : \mathbb{R}^{N_0} \to \mathbb{R}^{N_0}$ and $r_i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ for $i = 1, \ldots, L$ by

$$\begin{aligned} r_0(x) &:= x, \\ r_i(x) &:= \rho_{N_i}^*(W_i r_{i-1}(x) + b_i) \text{ for } i = 1, \ldots, L-1, \\ r_L(x) &:= W_L r_{L-1}(x) + b_L, \\ \Phi_{\mathbf{W}}(x) &:= r_L(x). \end{aligned}$$

Now we try to optimize the parameters in $\mathbf{W}$ such that $\Phi_{\mathbf{W}} \approx f$. To achieve this, we sample a set that consists of inputs $x_i \in X \subset \mathbb{R}^{N_{\text{in}}}$ and corresponding outputs $f(x_i) \in \mathbb{R}^{N_{\text{out}}}$ and assemble them in the so-called training set

$$T_{\text{train}} = \{(x_1, f(x_1)), \ldots, (x_{N_{\text{train}}}, f(x_{N_{\text{train}}}))\} \subset X \times \mathbb{R}^{N_{\text{out}}}. \tag{4.1}$$

To evaluate the performance of our chosen $\mathbf{W}$, we use the mean squared error loss $\mathscr{L}(\Phi_{\mathbf{W}}, T_{\text{train}})$ to measure the distance between $\Phi_{\mathbf{W}}$ and f on a training set. The mean squared error loss

is defined as

$$\mathscr{L}(\Phi_{\mathbf{W}}, T_{\text{train}}) := \frac{1}{|T_{\text{train}}|} \sum_{(x,y)\in T_{\text{train}}} \|\Phi_{\mathbf{W}}(x) - y\|_2^2.$$

Since we want $\Phi_{\mathbf{W}}$ to be close to f, we minimize the loss function with respect to $\mathbf{W}$. For that, we use some gradient-based optimization method. By the structure of the neural network, we can use the chain rule multiple times to divide the gradient of $\mathscr{L}$ into much simpler gradient computations.

We want that $\Phi_{\mathbf{W}}$ is close to f on X but we train it only on a sample set of X, so we achieve that $\Phi_{\mathbf{W}}$ is only on T_{train} close to f. During the training, the loss function will eventually decrease on the training set, but at some point the loss on different samples, that are not in the training set, will get worse [11]. This is called 'overfitting'.

To prevent overfitting, we use early stopping. For early stopping, we evaluate the loss function on a validation set $T_{\text{val}} \subset X \times \mathbb{R}^{N_{\text{out}}}$, where usually $T_{\text{val}} \cap T_{\text{train}} = \emptyset$. Our algorithm for early stopping is described in the following.

For that, let $\mathbf{W}^{(k)}$ be the weights in epoch k. To evaluate the quality of the neural network that results from $\mathbf{W}^{(k)}$, the MSE loss on the validation set $\mathscr{L}(\Phi_{\mathbf{W}^{(k)}}, T_{\text{val}})$ is computed at the end of each training epoch. Now let k_0 be the iteration where $\mathbf{W}^{(k_0)}$ is smaller than the losses in all previous iterations. If that is the case, we save $\mathbf{W}^{(k_0)}$ and its corresponding loss $\mathscr{L}(\Phi_{\mathbf{W}^{(k_0)}}, T_{\text{val}})$. In the next k^* iterations, where k^* is a prescribed number, we check if $\mathscr{L}(\Phi_{\mathbf{W}^{(k_0+i)}}, T_{\text{val}}) < \mathscr{L}(\Phi_{\mathbf{W}^{(k_0)}}, T_{\text{val}})$ for $i = 1, \ldots, k^*$. If that is the case for some i, we update $\mathbf{W}^{(k_0)}$ and $\mathscr{L}(\Phi_{\mathbf{W}^{(k_0)}}, T_{\text{val}})$ by setting them to $\mathbf{W}^{(k_0+i)}$ and $\mathscr{L}(\Phi_{\mathbf{W}^{(k_0+i)}}, T_{\text{val}})$. Then the last step is repeated. If there is no such i between 1 and k^*, the training is aborted and the last saved weights returned.

So we abort the training if the minimum loss is not decreasing over a prescribed number of consecutive epochs. Our reasoning behind this is that the loss on the validation set is not srictly decreasing and can even increase over some epochs, but that is fine for us as long as we can decrease the loss over time.

Figure 4.1 shows an example where we used early stopping with $k^* = 15$. Here, the validation loss of the last training epochs is presented. We see that a minimum is reached after 44 iterations. Since the next 15 training epochs yield no decrease of the MSE loss, the training is aborted after iteration 59.

We present now the construction of a DNN that approximates a function $f : \mathbb{R}^n \to \mathbb{R}$ with $n \in \mathbb{N}$. The training of one neural network is shown in algorithm 6. It takes the initialization of the deep neural network (DNN), the inputs and outputs of the training set $(x_{\text{train}}, y_{\text{train}})$, the inputs and outputs of the validation set $(x_{\text{val}}, y_{\text{val}})$, the loss function (LOSS_FN), the optimizer (optimizer), the number of training epochs (epochs) and the number (earlyStop) that describes after how many iterations without improvement of the validation loss the training gets aborted, so that is the k^* from above.

In our algorithm, the loss function is chosen as the mean squared error loss and we use the L-BFGS algorithm [12], which is a limited memory quasi-Newton method that uses approximations of the objective functions Hessian matrix, with strong Wolfe line-search [13], [14] as our optimizer. The number of training epochs is only the maximum number of iterations since we apply early stopping to our training algorithm. Usually, the training

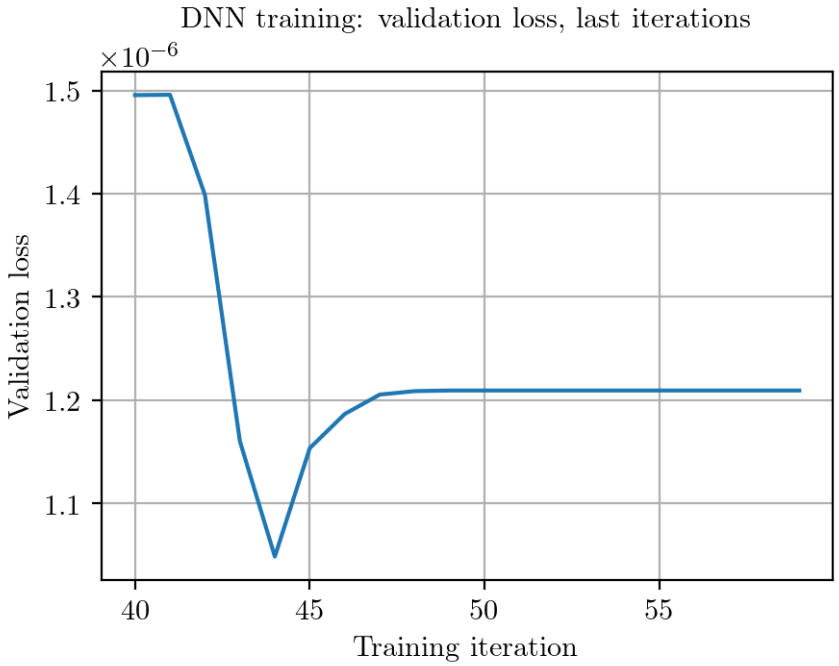

Figure 4.1: Validation loss during the last iterations of a neural network training procedure where early stopping was applied

terminates earlier because the loss over the validation set is not decreasing further.

The function TESTDNN in algorithm 6 returns the loss of the function LOSS_FN between the output of the DNN with the current parameters and the output of the objective function over the validation set which is denoted by y_{val}. So if LOSS_FN $= \mathscr{L}$, we have TESTDNN$(\text{DNN}, x_{\text{val}}, y_{\text{val}}, \text{LOSS_FN}) = \mathscr{L}(\text{DNN}, T_{\text{val}})$ with

$$T_{\text{val}} = \{((x_{\text{val}})_1, (y_{\text{val}})_1), \ldots, ((x_{\text{val}})_{N_{\text{val}}}, (y_{\text{val}})_{N_{\text{val}}})\}.$$

In addition, some funtions are imported from the Python package PyTorch. These functions are identified by the beginning 'torch.'.

Algorithm 6 DNN training

```
1: function trainDNN(DNN, x_train, y_train, x_val, y_val, loss_fn, optimizer, epochs, earlyStop)
2:     wait ← 0
3:     minimalValidationLoss ← testDNN(DNN, x_val, y_val, loss_fn)
4:     torch.save(DNN.state_dict( ), 'checkpoint.pth')
5:     for epoch = 1, ..., epochs do
6:         DNN.train( )
7:         function closure( )
8:             y_pred ← DNN(x_train).reshape(len(y_train))
9:             loss ← loss_fn(y_pred, y_train)
10:            optimizer.zero_grad( )
11:            loss.backward( )
12:            return loss
13:        optimizer.step(closure)
14:        validationLoss ← testDNN(DNN, x_val, y_val, loss_fn)
15:        if validationLoss < minimalValidationLoss then
16:            wait ← 0
17:            minimalValidationLoss ← validationLoss
18:            torch.save(DNN.state_dict( ), 'checkpoint.pth')
19:        else
20:            wait ← wait + 1
21:        if wait ≥ earlyStop then
22:            DNN.load_state_dict(torch.load('checkpoint.pth'))
23:            return
```

We start the algorithm trainDNN by initializing the variable wait, which indicates the number of training epochs without a decrease of the loss on the evaluation set, as zero and the variable minimalValidationLoss, which shows the loss that was achieved 'wait' epochs ago, as the loss on the validation set for the DNN before the training begins. Then the weights and biases of the DNN are saved in the file 'checkpoint.pth'. After that, the following operations are executed in every training epoch.

The procedures that are performed between line 6 and line 13 can be described as doing one optimization step with the optimizer to decrease the loss on the training set by adjusting the weights and biases of the DNN. Then we check if the loss on the evaluation set is currently smaller than the minimal validation loss over all previous epochs.

If that is the case, the variable wait is set to zero, indicating that the current parameters of the DNN have the best performance over the validation set, and the variable minimalValidationLoss is updated to the current validation loss. Since the parameters of the DNN will be changed in the next epochs, the current weights and biases are saved again in the file 'checkpoint.pth'. If the validation loss is not less than the minimal validation loss, the variable wait is increased by one.

To implement early stopping as described above, we check at the end of each training epoch whether the minimum loss has not decreased over so many consecutive epochs that we terminate the algorithm prematurely. If wait $\geq$ earlyStop, the current parameters of the neural network are overwritten with the parameters that were saved when the minimum loss was reached and the algorithm is aborted.

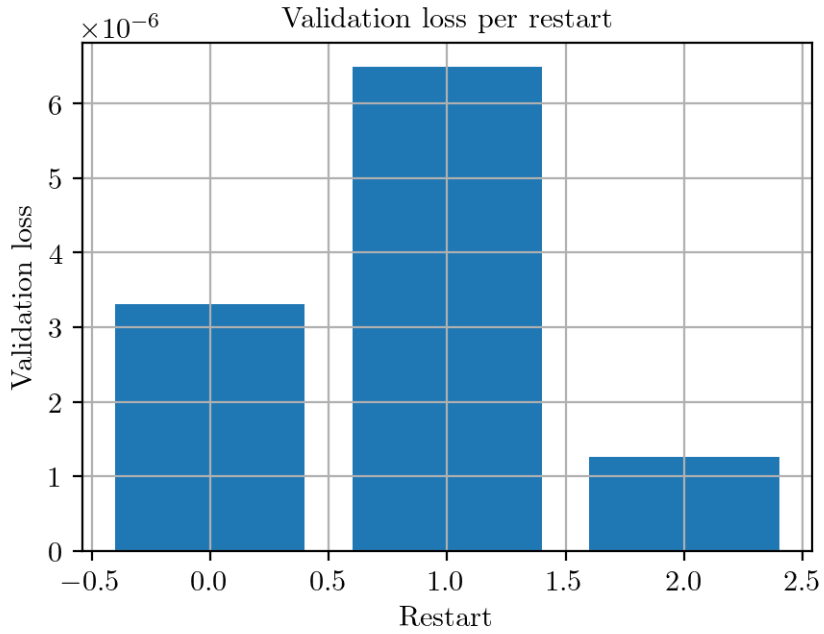

Figure 4.2: Example of the different validation losses resulting from the training of three neural networks

Since we search for local minima of the loss function, the initial value $\mathbf{W}^{(0)}$ of our iteration effects the local optimum that we get and therefore the performance. We use Kaiming initialization [15] to set our initial value $\mathbf{W}^{(0)}$. With Kaiming initialization, the starting values are initialized randomly since the elements of the weights W_i are sampled from a zero-mean Gaussian distribution whose standard deviation is $\sqrt{2/N_{i-1}}$ for $i \in \{1, \ldots, L\}$. The idea behind this type of random sampling is that the specified standard deviation prevents the exponential increase/ reduction of the input as shown in [15].

For the training of the DNN, we perform multiple restarts of the training algorithm with different initializations of $\mathbf{W}^{(0)}$ which minimizes the dependence of our neural network from the initial values. After we have trained enough DNNs, we select the neural network $\Phi_{\mathbf{W}^*}$ that has the smallest evaluation loss $\mathscr{L}(\Phi_{\mathbf{W}^*}, T_{\text{val}})$ over all restarts.

Figure 4.2 shows that training multiple neural networks can help to reduce the validation loss. If we did not restart the training, our DNN would have a validation loss of slightly above $3 \cdot 10^{-6}$. After two restarts, we obtain a neural network with a validation loss that is less than half of the DNN with no restarts. Of course, there are iterations where repeating the training does not lead to a reduction of the validation loss, but in general, the more neural networks we train, the lower the resulting loss tends to be. However, training more DNNs also increases the runtime.

Before the whole algorithm for the construction of the DNN is presented, we look at the data that we use for the training. If we sample the inputs in a small area, it is likely that the corresponding outputs are also close to each other. Since we convert the values for the training of the neural network from 64-bit floating point numbers to 32-bit floating point

numbers, it can even happen that the converted inputs or outputs are constant. In that case, the digits of these values that differ from each other get cut off at the conversion.

We want the values of the inputs and outputs to be distributed in such a way that significant differences are correctly represented. For that, the inputs $x \in \mathbb{R}^n$ and outputs $y \in \mathbb{R}$ are scaled to $\tilde{x} \in [0,1]^n$ and $\tilde{y} \in [0,1]$.

Let

$$T = \{(x_1, y_1), \ldots, (x_N, y_N)\} \tag{4.2}$$

be a sample set of size N and

$$T_x = \{x_1, \ldots, x_N\}, \qquad T_y = \{y_1, \ldots, y_N\}$$

the sets that contain the inputs and outputs of that sample.

We define $x^{\text{low}}, x^{\text{upp}} \in \mathbb{R}^n$ and $y^{\text{low}}, y^{\text{upp}} \in \mathbb{R}$ as

$$\begin{aligned} x_i^{\text{low}} &:= \min\{x_i \mid x \in T_x\} \text{ for } i = 1, \ldots, n, && (4.3)\\ x_i^{\text{upp}} &:= \max\{x_i \mid x \in T_x\} \text{ for } i = 1, \ldots, n, && (4.4)\\ y^{\text{low}} &:= \min T_y, && (4.5)\\ y^{\text{upp}} &:= \max T_y. && (4.6) \end{aligned}$$

Now, $\tilde{x}$ and $\tilde{y}$ are calculated as

$$\tilde{x}_i = \frac{x_i - x_i^{\text{low}}}{x_i^{\text{upp}} - x_i^{\text{low}}} \text{ for } i = 1, \ldots, n, \qquad \tilde{y} = \frac{y - y^{\text{low}}}{y^{\text{upp}} - y^{\text{low}}}. \tag{4.7}$$

After we have trained the neural network, the DNN outputs need to be rescaled so that we get a proper approximation of the function f. The output $\Phi(\tilde{x})$ of the DNN Φ is rescaled with the calculation $\Phi(\tilde{x}) \cdot (y^{\text{upp}} - y^{\text{low}}) + y^{\text{low}}$.

To summarize this, we present now the construction of a DNN as pseudo code in algorithm 7. Training parameters like the neural network structure (DNNStructure), the activation function (activFunc), the number of restarts of different DNN initializations (restarts), the number of training epochs (epochs), the number of epochs without decrease of the evaluation loss after which early stopping is applied (earlyStop), the fraction of the sample that is used for training (trainFrac) and the learning rate (learning_rate) are stored in V_{DNN}. We denote x^{low} as minIn, x^{upp} as maxIn, y^{low} as minOut and y^{upp} as maxOut. minIn and maxIn are calculated before the construction of the DNN and are taken as an input. The function FullyConnectedNN is imported from pymor.models.neural_network which is included in the Python package pyMOR. It builds a neural network with Kaiming initialization where the number of neurons in each layer gets specified by the first and the activation function of the neural network by the second argument.

Algorithm 7 DNN construction

1: **function** CONSTRUCTDNN(sample, V_{DNN}, minIn, maxIn)
2: normSample ← np.ZEROS(LEN(sample), LEN(sample[0][0]))
3: normVal ← np.ZEROS(LEN(sample))
4: **for** $i = 0, \ldots,$ LEN(sample) $- 1$ **do**
5: normSample$[i, :]$ ← sample$[i][0]$
6: normVal$[i]$ ← sample$[i][1]$
7: minOut ← np.MIN(normVal)
8: maxOut ← np.MAX(normVal)
9: normSample ← (normSample − minIn)/(maxIn − minIn)
10: normVal ← (normVal − minOut)/(maxOut − minOut)
11: x ← torch.FROM_NUMPY(normSample).TO(torch.float32)
12: y ← torch.FROM_NUMPY(normVal).TO(torch.float32)
13: trainSplit ← INT(trainFrac ∗ LEN(x))
14: $x_{\text{train}}, y_{\text{train}} \leftarrow x[:\text{trainSplit}], y[:\text{trainSplit}]$
15: $x_{\text{val}}, y_{\text{val}} \leftarrow x[\text{trainSplit}:], y[\text{trainSplit}:]$
16: DNN ← FULLYCONNECTEDNN(DNNStructure, ACTIVATION_FUNCTION ← ACTIVFUNC)
17: LOSS_FN ← torch.nn.MSELOSS( )
18: optimizer ← torch.optim.LBFGS(DNN.PARAMETERS( ), lr ← learning_rate, line_search_fn ← 'strong_wolfe')
19: TRAINDNN(DNN, $x_{\text{train}}, y_{\text{train}}, x_{\text{val}}, y_{\text{val}}$, LOSS_FN, optimizer, epochs, earlyStop)
20: evalDNN ← TESTDNN(DNN, $x_{\text{val}}, y_{\text{val}}$, LOSS_FN)
21: **for** $i = 1, \ldots,$ restarts **do**
22: DNN_i ← FULLYCONNECTEDNN(DNNStructure, ACTIVATION_FUNCTION ← ACTIVFUNC)
23: optimizer ← torch.optim.LBFGS(DNN_i.PARAMETERS( ), lr ← learning_rate, line_search_fn ← 'strong_wolfe')
24: TRAINDNN($\text{DNN}_i, x_{\text{train}}, y_{\text{train}}, x_{\text{val}}, y_{\text{val}}$, LOSS_FN, optimizer, epochs, earlyStop)
25: evalDNN_i ← TESTDNN($\text{DNN}_i, x_{\text{val}}, y_{\text{val}}$, LOSS_FN)
26: **if** $\text{evalDNN}_i <$ evalDNN **then**
27: evalDNN ← evalDNN_i
28: DNN ← DNN_i
29: **function** $\text{F}_{\text{ML}}(x_{\text{inp}})$
30: scaledInput ← torch.FROM_NUMPY($(x_{\text{inp}}$ − minIn)/(maxIn − minIn)).TO(torch.float32)
31: scaledOutput ← DNN(scaledInput)
32: **return** scaledOutput.NUMPY( )$[0] \cdot$ (maxOut − minOut) + minOut
33: **return** F_{ML}

The algorithm CONSTRUCTDNN begins by setting the scaled samples for training and testing. For that, normSample is defined as a matrix where each row i is equal to the input sample sample$[i][0]$ and normVal is a vector where each element i is set to the output sample sample$[i][1]$. Then we scale and translate them in line 9 and 10 as it is shown in (4.7).

After normSample and normVal are converted to a 32-bit floating point data type tensor from the torch package in lines 11 and 12, the training samples x_{train} and y_{train} are set to

a fraction of size trainFrac from the scaled and converted samples x and y. The remaining elements of x and y are used for the evaluation samples x_{val} and y_{val}.

Next, the neural network DNN with the structure DNNStructure and the activation function activFunc is initialized with Kaiming initialization. As an example, if DNNStructure would be equal to $[N_0, N_1, \ldots, N_L]$, we would get a DNN with L layers and N_i neurons in layer $i = 0, 1, \ldots, L$.

After the loss function is set to the MSE loss and the optimizer is set to the L-BFGS optimizer, we train the neural network DNN by calling trainDNN from the algorithm 6. evalDNN in line 20 is the loss of DNN on the scaled and converted validation set.

In the for-loop, multiple neural networks DNN_i are trained. If there is a neural network with a loss that is less than evalDNN, we overwrite DNN with the neural network DNN_i that has the smalles loss on the evaluation set.

After the for-loop, the function F_{ML} is defined. It takes an input x_{inp} and scales it like in line 9 while also converting it to a 32-bit float tensor like in line 11. Then the function calls DNN on the scaled and converted input, converts the neural network output back to a non-tensor float and rescales it in an inverted way compared to line 10. This value is then the output of F_{ML}.

Finally, F_{ML} is returned by constructDNN.

4.2 Modifying the EnOpt algorithm by using a neural network-based surrogate

For the next step, we use neural networks like in [4] to get an improved version of the EnOpt algorithm. Here, we want to replace calls of the FOM objective functional by surrogate functionals which are based on neural networks. An advantage of using neural networks to construct the surrogate is that the only information we need about the full order model is the objective functional values at the elements of the training and validation set. Hence, the operations that take place during the computations of the objective functional values, such as the solution of the underlying PDEs, can be treated as a black box model. This means that this method can also be applied to models where we only have access to the objective functional values.

The idea of the Adaptive-ML-EnOpt algorithm is to iterate through an outer iteration loop, where we construct a neural network-based surrogate with FOM evaluations, and an inner iteration loop, which optimizes the surrogate functional that was constructed before. Before every outer iteration, we do a single FOM optimization step. Instead of using the resulting iterate, we utilize the sample set T_k that is returned along with this iterate. Specifically, the samples are used to train a DNN and construct a surrogate functional. This surrogate is supposed to be a local approximation of the FOM functional around the current iterate. Defining a surrogate functional with globally sufficient accuracy would not be possible, since $\mathcal{D}$ is unbounded, and even if it where bounded, it would usually be too expensive computationally. Therefore we introduce a trust-region (TR) approach [16] next.

Trust-region methods are applied in cases like this where we have a surrogate that can be trusted to be a good representation of the objective functional within a certain region around the current iterate. The idea is to do one optimization step on the surrogate functional so that the solution is in this region. The next iterate is then set to this solution.

After that, the quality of the new iterate is evaluated. If this iterate is acceptable, so if the corresponding objective functional value gives a better result, the trust-region might be enlarged for the next iteration step. In case the iterate yields not a sufficient improvement of the objective functional value, the trust-region is reduced and the last iteration step could be repeated in this new region.

In our algorithm, the optimization step that is executed within the trust-region is called the inner iteration loop. Here, we use the EnOpt algorithm with a stopping criterion $\varepsilon_i > 0$ to optimize the surrogate functional while setting the projection PR in algorithm 1 so that the resulting solution is in the trust-region.

At the end of each outer iteration, we call another FOM optimization step. One of the outputs is the FOM objective functional value, denoted by $\tilde{F}_k$, of the resulting iterate $\tilde{\mathbf{q}}_k$. We use that to get a FOM-based stopping criterion. Our current iterate $\mathbf{q}_k$ has an objective functional value of F_k. At the end of each outer iteration, we check if $\tilde{F}_k > F_k + \varepsilon_o$ for some $\varepsilon_o > 0$. If that is true, there exists a sufficiently increasing point, which is $\tilde{\mathbf{q}}_k$, and another outer iteration follows, where the sample set that was returned from the last FOM OptStep call is used again for the construction of a surrogate. If that is not true, the procedure is terminated. In that case the FOM-EnOpt algorithm would also stop at that point if it had a stopping criterion of ε_o.

We describe now the Adaptive-ML-EnOpt algorithm with pseudo code. This algorithm takes a functional F, the initial guess $\mathbf{q}_0 \in \mathbb{R}^{N_\mathbf{q}}$, the sample size $N \in \mathbb{N}$, the tolerances $\varepsilon_o, \varepsilon_i > 0$ for the outer and inner iterations, the maximum number of outer and inner iterations $k_o^*, k_i^* \in \mathbb{N}$, the maximum number of trust-regoin method repetitions k_{TR}^*, the DNN-specific variables V_{DNN}, the initial step size $\beta > 0$, the step size contraction $r \in (0, 1)$, the maximum number of step size trials $\nu^* \in \mathbb{N}$, the variance $\sigma^2 \in \mathbb{R}^{N_b}$ with positive elements, the correlation coefficient $\rho \in (-1, 1)$, the number of time steps $N_t \in \mathbb{N}$ and the number of basis functions $N_b \in \mathbb{N}$.

Algorithm 8 Adaptive-ML-EnOpt algorithm

function AML-EnOpt($\mathrm{F}, \mathbf{q}_0, N, \varepsilon_o, \varepsilon_i, k_o^*, k_i^*, k_{\mathrm{TR}}^*, V_{\mathrm{DNN}}, \delta_{\mathrm{init}}, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b$)
  $N_{\mathbf{q}} \leftarrow \mathrm{len}(\mathbf{q}_0)$
  $F_k \leftarrow \mathrm{F}(\mathbf{q}_0)$
  $F_k^{\mathrm{next}} \leftarrow F_k$
  $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k \leftarrow \mathrm{OptStep}(\mathrm{F}, \mathbf{q}_0, N, 0, [\,], \mathrm{None}, F_k, \beta_1, \beta_2, r, \varepsilon_o, \nu^*, \sigma^2, \rho, N_t, N_n)$
  $k \leftarrow 1$
  $\mathbf{q}_k \leftarrow \mathbf{q}_0$
  $\mathbf{q}_k^{\mathrm{next}} \leftarrow \mathbf{q}_k.\mathrm{copy}(\,)$
  $\delta \leftarrow \delta_{\mathrm{init}}$
  while $\tilde{F}_k > F_k + \varepsilon_o$ and $k < k_o^*$ **do**
    $T_k^x \leftarrow \mathrm{np.zeros}((N, N_{\mathbf{q}}))$
    for $i = 0, \ldots, N-1$ **do**
      $T_k^x[i, :] \leftarrow T_k[i][0]$
    $\mathrm{minIn} \leftarrow \mathrm{np.zeros}(N_{\mathbf{q}})$
    $\mathrm{maxIn} \leftarrow \mathrm{np.zeros}(N_{\mathbf{q}})$
    for $i = 0, \ldots, N_{\mathbf{q}} - 1$ **do**
      $\mathrm{minIn}[i] \leftarrow \mathrm{np.min}(T_k^x[:, i])$
      $\mathrm{maxIn}[i] \leftarrow \mathrm{np.max}(T_k^x[:, i])$
    $\mathbf{d}_k \leftarrow \mathrm{np.abs}(\mathbf{q}_k - \tilde{\mathbf{q}}_k)$
    $\mathrm{tr} \leftarrow 1$
    while $F_k^{\mathrm{next}} \leq F_k + \varepsilon_o$ **do**
      assert $\mathrm{tr} \leq k_{\mathrm{TR}}^*$
      $\mathrm{F}_{\mathrm{ML}}^k \leftarrow \mathrm{constructDNN}(T_k, V_{\mathrm{DNN}}, \mathrm{minIn}, \mathrm{maxIn})$
      $F_k^{\mathrm{approx}} \leftarrow \mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k)$
      $\mathrm{flag}_{\mathrm{TR}} \leftarrow \mathrm{True}$
      while $\mathrm{flag}_{\mathrm{TR}}$ **do**
        $\mathbf{d}_k^{\mathrm{iter}} \leftarrow \delta \cdot \mathbf{d}_k$
        $\mathbf{q}_k^{\mathrm{next}} \leftarrow \mathrm{EnOpt}(\mathrm{F}_{\mathrm{ML}}^k, \mathbf{q}_k, N, \varepsilon_i, k_i^*, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b, \mathrm{PR} \leftarrow \mathbf{lambda}\ \mathrm{mu} : \mathrm{TR\text{-}Projection}(\mathrm{mu}, \mathbf{q}_k, \mathbf{d}_k^{\mathrm{iter}}), \mathbf{C}_{\mathrm{init}} \leftarrow \mathbf{C}_k)[0]$
        $F_k^{\mathrm{next}} \leftarrow \mathrm{F}(\mathbf{q}_k^{\mathrm{next}})$
        $\rho_k \leftarrow \dfrac{F_k^{\mathrm{next}} - F_k}{\mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k^{\mathrm{next}}) - F_k^{\mathrm{approx}}}$
        if $\rho_k < 0.25$ **then**
          $\delta \leftarrow 0.25 \cdot \delta$
        else
          if $\rho_k > 0.75$ and $\mathrm{np.any}(\mathrm{np.abs}(\mathbf{q}_k - \mathbf{q}_k^{\mathrm{next}}) - \mathbf{d}_k^{\mathrm{iter}} = 0)$ **then**
            $\delta \leftarrow 2 \cdot \delta$
        if $\rho_k > 0$ **then**
          $\mathrm{flag}_{\mathrm{TR}} \leftarrow$ **False**
      $\mathrm{tr} \leftarrow \mathrm{tr} + 1$
    $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k \leftarrow \mathrm{OptStep}(\mathrm{F}, \mathbf{q}_k^{\mathrm{next}}, N, k, T_k, \mathbf{C}_k, F_k^{\mathrm{next}}, \beta_1, \beta_2, r, \varepsilon_o, \nu^*, \sigma^2, \rho, N_t, N_b)$
    $F_k \leftarrow F_k^{\mathrm{next}}$
    $\mathbf{q}_k \leftarrow \mathbf{q}_k^{\mathrm{next}}.\mathrm{copy}(\,)$
    $k \leftarrow k + 1$
  return $\mathbf{q}^* \leftarrow \mathbf{q}_k$

We start the Adaptive-ML-EnOpt algorithm by initializing $N_{\mathbf{q}}$ as the length of $\mathbf{q}_0$ and F_k and F_k^{next} as the FOM objective functional value at $\mathbf{q}_0$. Similar to the EnOpt algorithm 1, we set $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k$ to the output of the function OptStep on the FOM objective functional F. Instead of repeating the optimization step algorithm on the FOM objective functional and using $\tilde{\mathbf{q}}_k$ as our iterate, we initialize the iterates $\mathbf{q}_k$ and $\mathbf{q}_k^{\text{next}}$ as $\mathbf{q}_0$. The $\tilde{F}_k$ from line 5 is then used in line 10 to check if the FOM objective functional value can be improved by more than ε_o. The while-loop is repeated until this is no longer the case.

In the while loop, x^{low} from (4.3), denoted as minIn, and x^{upp} from (4.4), denoted as maxIn, are computed first. This happens between line 11 and line 18. $\mathbf{d}_k$ is the absolute value of the difference between our current iterate $\mathbf{q}_k$ and the $\tilde{\mathbf{q}}_k$ that resulted from the FOM optimization step in line 5. This is np.abs$(\mathbf{q}_k - \tilde{\mathbf{q}}_k)$, so $|\mathbf{q}_k - \tilde{\mathbf{q}}_k|$, in line 19.

Now we want to compute the next iterate. We know from line 10 that it is possible to increase the FOM objective functional value of the current iterate by at least ε_o, for example by setting it to $\tilde{\mathbf{q}}_k$. Instead of using the $\tilde{\mathbf{q}}_k$ that we got from evaluations of the full order model functional F, the neural network-based surrogate functional F_{ML}^k is introduced by calling constructDNN in line 23. It should be noticed here that the sample T_k, that was obtained from the FOM optimization step in line 5 (and for the next outer iterations from line 39), is used for the training and testing of this DNN. This means that only samples around $\mathbf{q}_k$ are used for the training. Therefore we expect that the error between the FOM objective functional F and its surrogate F_{ML}^k is only sufficiently small for points that are close to $\mathbf{q}_k$. To take this into account, we use a trust-region method as described above.

Here we repeat a while-loop until a certain condition is met. We allow this while-loop to only repeat k_{TR}^* times. If this number is exceeded, we stop the algorithm. This can happen if the parameters are chosen inappropriately for this problem. In that case, we might want to run this algorithm with other parameters again.

The trust-region is characterized by $\mathbf{q}_k$, $\mathbf{d}_k$ and $\delta > 0$, which is initialized in line 9. It is defined by the algorithm TR-Projection which projects inputs into the trust-region as shown here:

Algorithm 9 Projection

```
1: function TR-Projection(x, q_k, d_k)
2:     upp ← q_k + d_k
3:     low ← q_k − d_k
4:     return np.maximum(np.minimum(x, upp), low)
```

Our trust-region is the area between $\mathbf{q}_k - \delta \cdot \mathbf{d}_k$ and $\mathbf{q}_k + \delta \cdot \mathbf{d}_k$. TR-Projection projects a point x into the trust-region by checking for each element $x[i]$ individually if it lies below $(\mathbf{q}_k - \delta \cdot \mathbf{d}_k)[i]$ or above $(\mathbf{q}_k + \delta \cdot \mathbf{d}_k)[i]$. If the former applies, $x[i]$ is set to $(\mathbf{q}_k - \delta \cdot \mathbf{d}_k)[i]$. If the latter is true, $x[i]$ is set to $(\mathbf{q}_k + \delta \cdot \mathbf{d}_k)[i]$. Otherwise, $x[i]$ is not changed.

The next iterate $\mathbf{q}_k^{\text{next}}$ is now computed by calling the EnOpt algorithm on the surrogate functional F_{ML}^k. For the inputs of the EnOpt algorithm, we set here the projection pr to **lambda** mu : TR-Projection$(\text{mu}, \mathbf{q}_k, \delta \cdot \mathbf{d}_k)$, so that the operations are inside the trust-region, and the initial covariance matrix $\mathbf{C}_{\text{init}}$ to $\mathbf{C}_k$, so that the samples for the first iteration are distributed like the sample set that was used for the training of the surrogate functional.

To examine the quality of the iterate $\mathbf{q}_k^{\text{next}}$ and the surrogate F_{ML}^k, ρ_k is defined in line 30. If $\rho_k < 0.25$, the surrogate is not sufficiently accurate for us and we decrease the trust-region

by multiplying δ with 0.25. If the condition in line 34 is true, the surrogate is a sufficiently good approximation of F and from np.ANY(np.ABS($\mathbf{q}_k - \mathbf{q}_k^{\text{next}}$) $- \mathbf{d}_k^{\text{iter}} = 0$) follows that $\mathbf{q}_k^{\text{next}}$ is at the boundary of the trust-region. In this case, the trust-region is extended by multiplying δ with 2. If ρ_k is greater than zero, we leave the inner while-loop since F_k^{next} is greater than F_k. The denominator in line 30, $\mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k^{\text{next}}) - \mathrm{F}_{\mathrm{ML}}^k(\mathbf{q}_k)$, should be positive because $\mathbf{q}_k^{\text{next}}$ results from the EnOpt algorithm on $\mathrm{F}_{\mathrm{ML}}^k$ with the initialization $\mathbf{q}_k$.

If the condition in line 21 is still not satisfied, the same procedure is repeated with another surrogate. If it is satisfied, $\tilde{\mathbf{q}}_k, T_k, \mathbf{C}_k, \tilde{F}_k$ is updated in line 39 similar to line 5. $\tilde{F}_k$ is used again in line 10 to check if an improvement of the FOM objective functional value by more than ε_o is possible until the condition in this line is no longer satisfied.

We minimize our objective function j now by applying $-j$ to the Adaptive-ML-EnOpt algorithm as it is shown in algorithm 10.

Algorithm 10 ROM-EnOpt algorithm

1: **function** ROM-EnOpt($\mathbf{q}_0, N, \varepsilon_o, \varepsilon_i, k_o^*, k_i^*, k_{\text{TR}}^*, V_{\text{DNN}}, \delta_{\text{init}}, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, \mathbf{q}_{\text{base}}$)
2: $N_b \leftarrow$ LEN($\mathbf{q}_{\text{base}}$)
3: **return** AML-EnOpt($-\mathrm{J}, \mathbf{q}_0, N, \varepsilon_o, \varepsilon_i, k_o^*, k_i^*, k_{\text{TR}}^*, V_{\text{DNN}}, \delta_{\text{init}}, \beta_1, \beta_2, r, \nu^*, \sigma^2, \rho, N_t, N_b$)

Like in the FOM-EnOpt algorithm 5, there are some more inputs that ROM-EnOpt requires, but we also omit these because they are only needed for the calculation of J. We require $\mathbf{q}_{\text{base}}$ instead of N_b as an input because $\mathbf{q}_{\text{base}}$ is used for the computation of J.

We introduced this algorithm to speed up the FOM-EnOpt algorithm by decreasing the number of FOM evaluation. Like in the FOM-EnOpt procedure, we begin the algorithm with a FOM optimization step in line 5 and call one at the end of each outer iteration in line 39. Single FOM functional evaluations are performed in line 3 for the initialization of F_k and in line 29 to update F_k^{next}.

We have mentioned at the end of chapter 3 that calls of the FOM optimization step procedure are the most expensive computations in the FOM-EnOpt algorithm. In the Adaptive-ML-EnOpt procedure, we replaced most FOM optimization step calls by surrogate functional calls. Calculations of surrogate functional values are much cheaper to compute since they require only a single forward pass through a neural network.

Since most optimization steps are done with surrogate functionals, we expect that the number of FOM evaluations is smaller than in the FOM-EnOpt algorithm. However, now the FOM evaluations are not the only costly computations. We should note that the training of the neural network required for the surrogate can also take some time. In our experiments in chapter 5, we examine the number of FOM evaluations and computation times of these two algorithms, among other things.

5 Numerical experiments

In this chapter we are going to measure the performance of our algorithms by applying them to an example of the weak problem (2.3). For this purpose, the solution of this problem is derived first. Afterwards, we compare the analytical solution with the solution from the FOM-EnOpt and the Adaptive-ML-EnOpt (AML-EnOpt) algorithm. Additionally, we examine how the optimization algorithms behave if parameters, such as the number of time steps, are changed or if other neural network structures are used.

5.1 Example of an analytical problem

The problem (2.3) is solved in [6] with the adjoint state method [17], [18]. For this method, we use the Lagrangian $\mathcal{L} : Q \times X \times X \times V \to \mathbb{R}$ with

$$\mathcal{L}(q, u, z, \tilde{z}) = J(q, u) - (\partial_t u, z)_I - (\nabla u, \nabla z)_I + (f + q, z)_I + (u_0 - u(0), \tilde{z}) \tag{5.1}$$

We want to find now a stationary point $\bar{q}$ of j such that

$$j'(\bar{q})(\delta_q) = 0 \quad \forall \delta_q \in Q. \tag{5.2}$$

$j'(\bar{q})(\delta_q)$ is here the directional derivative, which is defined as

$$j'(\bar{q})(\delta_q) = \lim_{\tau \downarrow 0} \frac{j(q + \tau\delta_q) - j(q)}{\tau}.$$

If (5.2) holds, then $\bar{q}$ satisfies the first order optimality conditions of (2.4). In this case, $\bar{q}$ is even optimal due to the linear-quadratic structure of the optimal control problem [6].

If we choose u now as $u(q)$, so that it satisfies the weak state equations (2.2), we get

$$j(q) = J(q, u(q)) = \mathcal{L}(q, u(q), z, \tilde{z}),$$

since all terms after '$J(q, u)$' in (5.1) are equal to zero.

We get now the directional derivative of j by taking the directional derivative of $\mathcal{L}$ with respect to q. We note beforehand that

$$\begin{aligned} -(\partial_t u(q), z)_I - (\nabla u(q), \nabla z)_I + (f + q, z)_I &= 0 \quad \forall q \in Q, z \in X, \\ (u_0 - u(q)(0), \tilde{z}) &= 0 \quad \forall q \in Q, \tilde{z} \in X. \end{aligned}$$

This means that for all $q \in Q$ and $z, \tilde{z} \in X$ an alteration of q does not change the function value of $\mathcal{L}(q, u(q), z, \tilde{z})$. Thus, we get the directional derivative

$$\begin{aligned} j'(q)(\delta_q) =& \mathcal{L}'(q, u(q), z, \tilde{z})(\delta_q) \\ =& \mathcal{L}'_q(q, u(q), z, \tilde{z})(\delta_q) + \mathcal{L}'_u(q, u(q), z, \tilde{z})(\delta_u). \end{aligned} \tag{5.3}$$

The first summand in the second line denotes the directional derivative of the Lagrangian with respect to q in direction $\delta_q \in Q$. Analogously, the second summand denotes the directional derivative of the Lagrangian with respect to u in direction $\delta_u \in X$. δ_u is here uniquely defined so that it satisfies

$$\begin{aligned} (\partial_t \delta_u, \phi)_I + (\nabla \delta_u, \nabla \phi)_I &= (\delta_q, \phi)_I \quad \forall \phi \in X, \\ \delta_u(0) &= 0 \qquad\qquad \text{in } \Omega. \end{aligned} \tag{5.4}$$

Such a unique solution exists by Proposition 2.1. Now it holds, that for any $\tau \in \mathbb{R}$,

$$\begin{aligned} (\partial_t (u(q) + \tau\delta_u), \phi)_I + (\nabla (u(q) + \tau\delta_u), \nabla\phi)_I &= (f + q + \tau\delta_q, \phi)_I \quad \forall \phi \in X, \\ (u(q) + \tau\delta_u)(0) &= u_0 \qquad\qquad \text{in } \Omega \end{aligned}$$

by multplication of the equations in (5.4) with τ and addition of the product to the weak state equations (2.2). Since $u(q + \tau\delta_q)$ is uniquely defined to satisfy the equations

$$\begin{aligned} (\partial_t u(q + \tau\delta_q), \phi)_I + (\nabla u(q + \tau\delta_q), \nabla\phi)_I &= (f + q + \tau\delta_q, \phi)_I \quad \forall \phi \in X, \\ u(q + \tau\delta_q)(0) &= u_0 \qquad\qquad \text{in } \Omega, \end{aligned}$$

it follows that $u(q + \tau\delta_q) = u(q) + \tau\delta_u$.

Therefore we get for every functional g which has a directional derivative in δ_u-direction:

$$\begin{aligned} g'_q(u(q))(\delta_q) &= \lim_{\tau \downarrow 0} \frac{g(u(q + \tau\delta_q)) - g(u(q))}{\tau} \\ &= \lim_{\tau \downarrow 0} \frac{g(u(q) + \tau\delta_u) - g(u(q))}{\tau} \\ &= g'_u(u(q))(\delta_u). \end{aligned}$$

Because of this, we have the summand $\mathcal{L}'_u(q, u(q), z, \tilde{z})(\delta_u)$ in equation (5.3).

We calculate now $\mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u)$. For this we compute $J'_u(q, u)(\delta_u)$ first:

$$\begin{aligned} J'_u(q, u)(\delta_u) &= \lim_{\tau \downarrow 0} \frac{1}{2\tau} \int_0^T \int_\Omega (u(t,x) + \tau\delta_u(t,x) - \hat{u}(t,x))^2 - (u(t,x) - \hat{u}(t,x))^2 \,\mathrm{d}x \,\mathrm{d}t \\ &= \lim_{\tau \downarrow 0} \int_0^T \int_\Omega \delta_u(t,x)(u(t,x) - \hat{u}(t,x)) \,\mathrm{d}x \,\mathrm{d}t + \frac{\tau}{2} \int_0^T \int_\Omega (\delta_u(t,x))^2 \,\mathrm{d}x \,\mathrm{d}t \\ &= \int_0^T \int_\Omega \delta_u(t,x)(u(t,x) - \hat{u}(t,x)) \,\mathrm{d}x \,\mathrm{d}t \\ &= (\delta_u, u - \hat{u})_I \end{aligned}$$

We get with similar calculations for the other summands in the Lagrangian:

$$\mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u) = (\delta_u, u - \hat{u})_I - (\partial_t \delta_u, z)_I - (\nabla \delta_u, \nabla z)_I - (\delta_u(0), \tilde{z}).$$

Since we can choose $z \in X$ and $\tilde{z} \in V$ freely, they are set such that $\mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u) = 0$. Integration by parts gives us

$$\begin{aligned} & & \mathcal{L}'_u(q, u, z, \tilde{z})(\delta_u) &= 0 \\ \iff & & (\delta_u(T), z(T)) - (\delta_u(0), z(0)) - (\delta_u, \partial_t z)_I + (\nabla \delta_u, \nabla z)_I + (\delta_u(0), \tilde{z}) &= (\delta_u, u - \hat{u})_I. \end{aligned}$$

This holds if we set $\tilde{z} = z(0)$ and z such that

$$\begin{aligned} -(\phi, \partial_t z)_I + (\nabla\phi, \nabla z)_I &= (\phi, u - \hat{u})_I \quad \forall \phi \in X, \\ z(T) &= 0 \qquad\qquad\quad \text{in } \Omega. \end{aligned} \tag{5.5}$$

We call this the adjoined state equation.

With $z(q)$ defined such as in (5.5) and $\tilde{z}(q) = z(q)(0)$, it holds $\mathcal{L}'_u(q, u(q), z(q), \tilde{z}(q))(\delta_u) = 0$ and therefore the expression in (5.3) is equal to

$$\begin{aligned} j'(q)(\delta_q) &= \mathcal{L}'_q(q, u(q), z(q), \tilde{z}(q))(\delta_q) \\ &= (\alpha q, \delta_q)_I + (z(q), \delta_q)_I \\ &= (\alpha q + z(q), \delta_q). \end{aligned}$$

For the testing of our algorithms, we consider the problem (2.3) on $\Omega \times I = (0,1)^2 \times (0,T)$ with $T = 0.1$. The following example is originally described in [6]. To define the functions in (2.3), we use the eigenfunction

$$w_a(t, x_1, x_2) := \exp(a\pi^2 t)\sin(\pi x_1)\sin(\pi x_2) \text{ for } a \in \mathbb{R}.$$

Now we set

$$\begin{aligned} f(t, x_1, x_2) &:= -\pi^4 w_a(T, x_1, x_2), \\ \hat{u}(t, x_1, x_2) &:= \frac{a^2 - 5}{2 + a}\pi^2 w_a(t, x_1, x_2) + 2\pi^2 w_a(T, x_1, x_2), \\ u_0(x_1, x_2) &:= \frac{-1}{2 + a}\pi^2 w_a(0, x_1, x_2). \end{aligned}$$

If we set the regularization parameter α in the objective functional (2.1a) as π^{-4}, we get the optimal solution $(\bar{q}, \bar{u}, \bar{z})$, where

$$\begin{aligned} \bar{q}(t, x_1, x_2) &:= -\pi^4 \left(w_a(t, x_1, x_2) - w_a(T, x_1, x_2)\right), \\ \bar{u}(t, x_1, x_2) &:= \frac{-1}{2 + a}\pi^2 w_a(t, x_1, x_2), \\ \bar{z}(t, x_1, x_2) &:= w_a(t, x_1, x_2) - w_a(T, x_1, x_2). \end{aligned} \tag{5.6}$$

It holds that $\bar{q} \in Q, \bar{u} \in X, \bar{z} \in X$. We confirm now that $(\bar{q}, \bar{u}, \bar{z})$ is a minimizer by checking if $\bar{q}$ satisfies (5.2), $\bar{u}$ (2.2) and $\bar{z}$ (5.5).

Beginning with $\bar{z}$, we have $\bar{z}(T, x_1, x_2) = 0$ trivially for all $(x_1, x_2) \in \Omega$ and therefore the second equation in (5.5) is satisfied. To check the first equation in (5.5), we write it as

$$(\phi, \partial_t z)_I - (\nabla\phi, \nabla z)_I + (\phi, u - \hat{u})_I = 0 \quad \forall \phi \in X.$$

Integration by parts gives for all $\phi \in X$

$$(\phi, \partial_t z)_I - (\nabla\phi, \nabla z)_I + (\phi, u - \hat{u})_I = (\phi, \partial_t z + \Delta z + u - \hat{u})_I.$$

Now we compute

$$\begin{aligned} \partial_t \bar{z}(t, x_1, x_2) &= \partial_t w_a(t, x_1, x_2) = a\pi^2 w_a(t, x_1, x_2) \\ \Delta \bar{z}(t, x_1, x_2) &= 2\pi^2 (w_a(T, x_1, x_2) - w_a(t, x_1, x_2)) \\ \bar{u}(t, x_1, x_2) - \hat{u}(t, x_1, x_2) &= \pi^2((2 - a)w_a(t, x_1, x_2) - 2w_a(T, x_1, x_2)). \end{aligned}$$

From $\partial_t \bar{z}(t,x_1,x_2) + \Delta \bar{z}(t,x_1,x_2) + \bar{u}(t,x_1,x_2) - \hat{u}(t,x_1,x_2) = 0$ for all $(x_1,x_2) \in \Omega, t \in I$ follows now that $\bar{z}$ satisfies the adjoined state equation (5.5).

By doing the same for $\bar{u}$, we see that it meets the initial condition $\bar{u}(0,x_1,x_2) = u_0(x_1,x_2)$ for all $(x_1,x_2) \in \Omega$ of (2.2). Again, we write the first equation in (2.2) as

$$(\partial_t u, \phi)_I + (\nabla u, \nabla \phi)_I - (f+q, \phi)_I = 0 \quad \forall \phi \in X,$$

and integration by parts gives us for all $\phi \in X$

$$(\partial_t \bar{u}, \phi)_I + (\nabla \bar{u}, \nabla \phi)_I - (f+\bar{q}, \phi)_I = (\partial_t \bar{u} - \Delta \bar{u} - f - \bar{q}, \phi)_I.$$

It holds

$$\begin{aligned} \partial_t \bar{u}(t,x_1,x_2) &= \frac{-a}{2+a}\pi^4 w_a(t,x_1,x_2) \\ \Delta \bar{u}(t,x_1,x_2) &= \frac{2}{2+a}\pi^4 w_a(t,x_1,x_2) \\ \partial_t \bar{u}(t,x_1,x_2) - \Delta \bar{u}(t,x_1,x_2) &= -\pi^4 w_a(t,x_1,x_2) \\ &= f(t,x_1,x_2) + \bar{q}(t,x_1,x_2). \end{aligned}$$

Therefore $\partial_t \bar{u}(t,x_1,x_2) - \Delta \bar{u}(t,x_1,x_2) - f(t,x_1,x_2) - \bar{q}(t,x_1,x_2) = 0$ for all $(x_1,x_2) \in \Omega, t \in I$, so $\bar{u}$ fulfills the conditions of the weak state equation (2.2).

With $\alpha = \pi^{-4}$ we get $\alpha \bar{q}(t,x_1,x_2) + \bar{z}(t,x_1,x_2) = 0$ for all $(x_1,x_2) \in \Omega, t \in I$, which means that $\bar{q}$ satisfies the optimality condition (5.2). Now we conclude that $(\bar{q}, \bar{u}, \bar{z})$ is the optimal solution.

Since the solutions $q \in Q_d$ of our algorithms have the form that is described in (2.9), we need to define a control-shape functional before we search for an optimizer. $\bar{q}$ can be written as

$$\bar{q}(t,x_1,x_2) = -\pi^4(\exp(a\pi^2 t) - \exp(a\pi^2 T))\sin(\pi x_1)\sin(\pi x_2).$$

Now we have $\bar{q}$ as a product of the term $-\pi^4(\exp(a\pi^2 t) - \exp(a\pi^2 T))$, that is only dependent on t, and the shape functional $\sin(\pi x_1)\sin(\pi x_2)$, which depends only on x and is an element of H. Additional shape functionals for the control functional are not used so that we can compare the outputs of our optimization algorithms with the analytical solution $\bar{q}$. We expect that the elements q_1^i of the control vectors $\mathbf{q}$ that we get as outputs are close to

$$\bar{q}_1^i = -\pi^4(\exp(a\pi^2 t_i) - \exp(a\pi^2 T)) \text{ for } i = 0, \ldots, N_t. \tag{5.7}$$

For our tests, we choose a to be $-\sqrt{5}$ which has the consequence that f and $\hat{u}$ do not depend on time.

5.2 Numerical results

Now we test the FOM- and AML-EnOpt algorithms on the example from the last section. We discretize the space $\Omega = (0,1)^2$ as described in subsection 2.2.1 with 50 grid intervals. The time interval $I = (0,T)$ with $T = 0.1$ that we observe is divided into 10 smaller intervals which are all of the same size. This is described in subsection 2.2.2 with $N_t = 10$. Since we have only one shape functional, the control vectors $\mathbf{q}$ that we search for have $N_{\mathbf{q}} = 11$

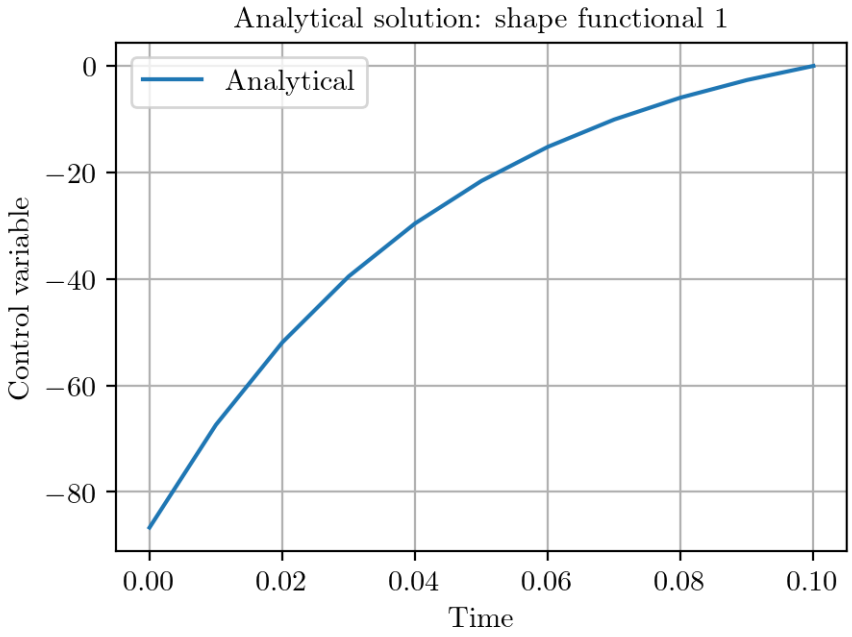

Figure 5.1: The analytical control vector with elements $\bar{q}_1^i$ for $i = 0, \ldots, N_t$ as in (5.7)

elements. The resulting analytical control vector with elements as in (5.7) is depicted in Figure 5.1. Its FOM objective functional value is approximately 4.2299573. If we describe results in this chapter and nothing else is specified, we use the parameters given below.

The initial control vector $\mathbf{q}_0$ is a vector with constant entries of the value -40. This value lies approximately between the minimum element in (5.7) which is $\bar{q}_1^0 \approx -87$ and the maximum $\bar{q}_1^{N_t} = 0$. The values of the constant initial control vector, along with other parameters for the algorithms that are presented in the chapters 3 and 4, are specified in the Table 5.1. The notation in this table corresponds to the notation in these algorithms.

The neural network that is used for the surrogate functional consists of two hidden layers where each hidden layer has 25 neurons. Our activation function is the tanh function as mentioned in section 4.1. Early stopping is applied for the training of the DNN with a maximum of 1000 training epochs. The variable earlyStop in algorithm 6 is set to 15. We use the L-BFGS optimizer with strong Wolfe line-search for the minimization of the MSE loss on the validation set. The learning rate of this optimizer is set to 10^{-2}. The validation set consists of 20% of the sample set, so the other 80% of the sample set are used for the training set. Therefore, trainFrac in algorithm 7 is set to 0.8. The number of training restarts is very small because a higher number would extend the training time of the DNN by so much that it makes the algorthm terminate slower. We restart the training two times, so we train three DNNs in total to construct the surrogate functional.

Before the results are presented, we want to note that the FOM-EnOpt and the Adaptive-ML-EnOpt procedures minimize the objective functional j by maximizing the negative functional $-j$. The results that we show are converted back to the outputs that the objective functional j would give, although the true values during these algorithms are negative. As

Table 5.1: Parameters used in the FOM-EnOpt and AML-EnOpt algorithms

Parameter	Value
Elements of the initial constant control vector $\mathbf{q}_0$	-40
Initial step size β_1	1
Initial covariance matrix adaption step size β_2	0.1
Initial trust-region step size δ_{init}	100
Step size contraction r	0.5
Maximum step size trials ν^*	10
Maximum (outer/ inner) iterations k^*, k_o^*, k_i^*	1000
Maximum trust-region iterations k_{TR}^*	5
Initial control variance σ_1^2	0.1
Constant correlation factor ρ	0.9
Sample size N	100
FOM-EnOpt ε	10^{-8}
Adaptive-ML-EnOpt inner iteration ε_i	10^{-12}
Adaptive-ML-EnOpt outer iteration ε_o	10^{-8}

an example, the graphs in Figure 5.2 would be mirrored on the x-axis if they had shown the values during the respective procedure.

Also, we want to clarify what we consider an outer and inner iteration. If we talk about the FOM-EnOpt algorithm, outer iterations involve computations of FOM optimization steps. Outer iterations in the FOM-EnOpt algorithm 1 are the first OptStep call in line 3 and passes of the while-loop from line 5 to 8. The first optimization step in line 3 is also considered an outer iteration because it has almost the same computational complexity as the calculations in the while-loop. Moreover, it changes the iterate. Inner iterations do not exist in this algorithm. In the AML-EnOpt algorithm, outer iterations include not only FOM optimization steps, but also the construction of a surrogate functional, and are therefore more expensive than the outer iterations of the FOM-EnOpt algorithm. In the Adaptive-ML-EnOpt algorithm 8, the computations from line 10 to line 42 are summed up as one outer iteration. The call of the FOM optimization step in line 5 is not considered an outer iteration loop because it is not as expensive as the computations in the outer iteration loop and also does not change the iterate during its call. Inner iterations are calls of the EnOpt algorithm on the neural network-based surrogate functional which occurs in line 28 of the AML-EnOpt algorithm.

For consistency, we will refer during our explanations in this chapter to $\mathbf{q}_k$ as the iterate at the beginning of the outer iteration k and $\mathbf{q}_k^{\text{next}}$ as the iterate at the end of the outer iteration k. This is based on the notation of the AML-EnOpt algorithm 8. If we talk about the FOM-EnOpt algorithm, we refer to $\mathbf{q}_k$ as the iterate before the call of the optimization step procedure in line 3, respectively line 7, in the k-th outer iteration of the algorithm 1 and thus we denote $\mathbf{q}_k^{\text{next}}$ as the iterate after this call in iteration k. Although, such a $\mathbf{q}_k^{\text{next}}$ is not defined in the FOM-EnOpt algorithm.

Figure 5.2 shows the development of the FOM objective functional value $j(\mathbf{q}_k^{\text{next}})$ after each outer iteration during the FOM- and AML-EnOpt procedures, as well as the respective functional value $j_{\text{ML}}^k(\mathbf{q}_k^{\text{next}})$ of the surrogate functional j_{ML}^k that is used in line 28 of the AML-EnOpt algorithm 8 to compute the iterate $\mathbf{q}_k^{\text{next}}$.

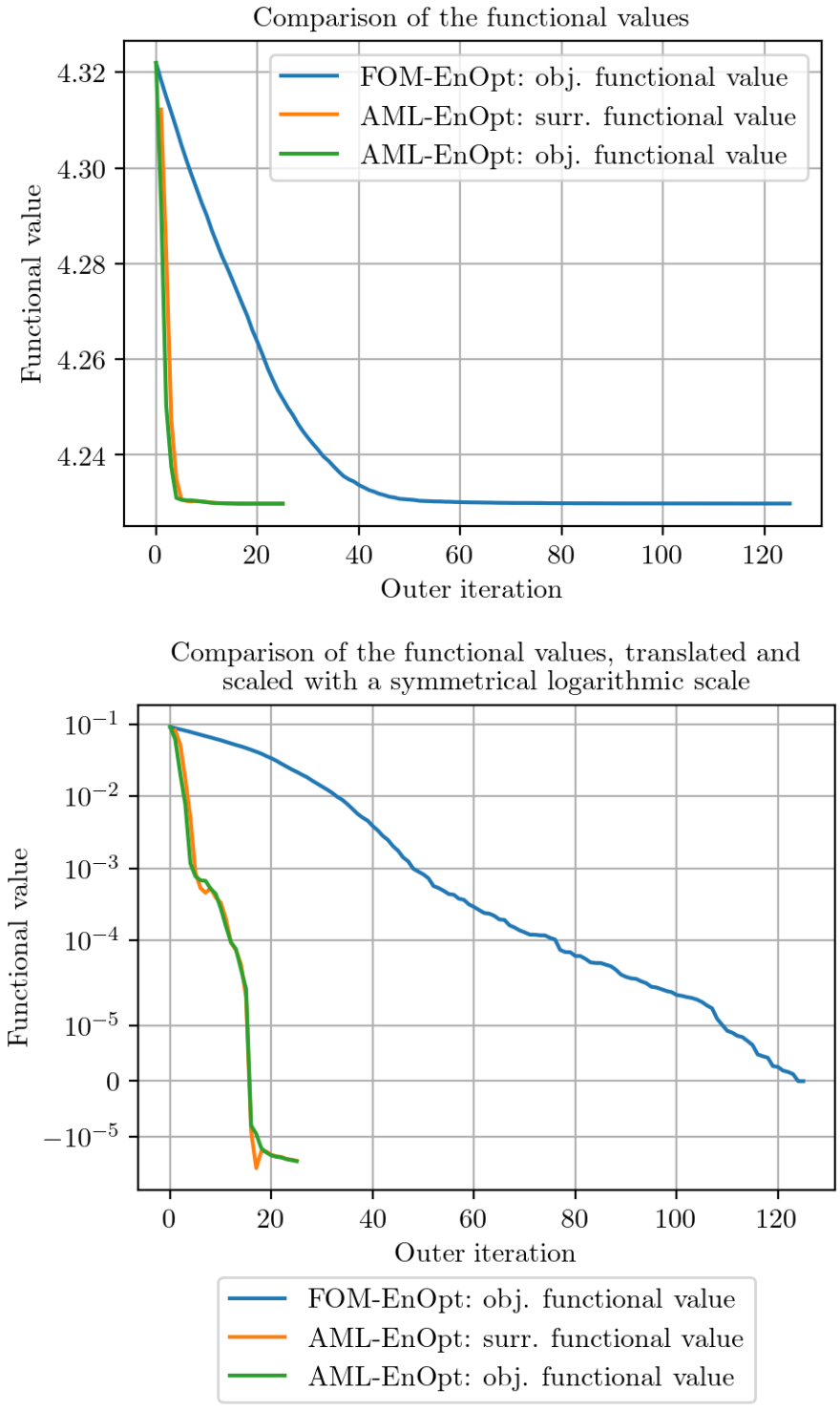

Figure 5.2: The FOM objective functional values obtained during the outer iterations of both EnOpt algorithms and the surrogate functional values obtained during the outer iterations of the AML-EnOpt algorithm at the top. The plot at the bottom shows these values, translated by the objective functional value of the FOM-EnOpt output and scaled with a symmetrical logarithmic scale which is linear at values which are closer to zero than the absolute value of the difference between the objective functional values of the FOM- and AML-EnOpt outputs.

We can see here that the functional values of the FOM-EnOpt and the Adaptive-ML-EnOpt algorithms converge towards a minimum. The output of the FOM-EnOpt algorithm gives an output whose objective functional value is approximately 4.22982802 after 125 iterations. The FOM objective functional value of the output from the AML-EnOpt procedure is approximately 4.22981359 which is reached after only 25 outer iterations. So the Adaptive-EnOpt algorithm gives here not only an output that has a smaller objective functional value, but also requires far fewer outer iterations than the FOM-EnOpt algorithm to terminate.

To compare the functional values of the objective functional and the surrogate functionals of the Adaptive-ML-EnOpt procedure, we examine Figure 5.3. The plot at the top shows the same graphs as in Figure 5.2, except that the objective functional values of the FOM-EnOpt algorithm are not included. The plot at the bottom shows only the last five outer iterations of the plot from above. We can see here that there is quite a big difference between the objective functional values and the surrogate functional values in the first outer iterations. The difference after the first iteration is approximately 0.02. However, this difference gets smaller during the runtime of the procedure and is in the order of 10^{-7} during the last iterations. One exception is the iteration 17, where the FOM objective functional value is approximately 4.22981852 and the surrogate functional value about 4.22981001. The difference between these values is here, compared to the iterations nearly relatively large. This can be seen in Figure 5.4 which shows the values of the plot at the top of Figure 5.3 with a symmetrical logarithmic scale.

One reason for the improving accuracy of the surrogate functional at the iterates is the difference between $\mathbf{q}_k$ and $\mathbf{q}_k^{\text{next}}$ at different outer iterations. In the first iterations, we are relatively far away from an optimum and therefore the iterates change considerably. The surrogate functional is trained by a training and a validation set which are sampled around the iterate $\mathbf{q}_k$, so if the difference between $\mathbf{q}_k^{\text{next}}$ and $\mathbf{q}_k$ is large, the same tends to hold for the difference between $\mathbf{q}_k^{\text{next}}$ and the samples, so the surrogate functional is less precise at $\mathbf{q}_k^{\text{next}}$. As the algorithm progresses, the iterates converge towards an optimum and the differences between successive iterates are smaller, resulting in more accurate surrogate functional values.

This can be seen in Figure 5.5. In the first iteration, the step from $\mathbf{q}_k$ to $\mathbf{q}_k^{\text{next}}$ is much larger than the step from $\mathbf{q}_k$ to $\tilde{\mathbf{q}}_k$ with respect to the L^2-norm. The same holds with regard to the difference between $\mathbf{q}_k$ and the samples in T_k. Therefore the DNN is inaccurate at $\mathbf{q}_k^{\text{next}}$. If this were the FOM-EnOpt algorithm, then $\tilde{\mathbf{q}}_k$ would be the next iterate, so we can see here a reason why the AML-EnOpt algorithm requires far fewer iterations than the FOM-EnOpt procedure which is that the machine learning-based algorithm takes larger steps at the beginning.

Although the step from $\mathbf{q}_k$ to $\mathbf{q}_k^{\text{next}}$ is still larger than the step from $\mathbf{q}_k$ to $\tilde{\mathbf{q}}_k$ at the last iteration, the step sizes differ not by a lot compared to the first iteration. $\mathbf{q}_k^{\text{next}}$ is even closer to $\mathbf{q}_k$ than all the samples are since the value of $\mathbf{q}_k^{\text{next}}$ at the bottom plot in Figure 5.5 is smaller than the value of T_k min which is the sample in T_k with the minimum L^2-distance to $\mathbf{q}_k$.

The analytical solution is not necessarily the optimal solution. To get an optimal solution as a reference, we use the L-BFGS-B minimizer. The stopping criterion is here chosen so that this algorithm terminates when the maximum norm of the gradient is smaller than

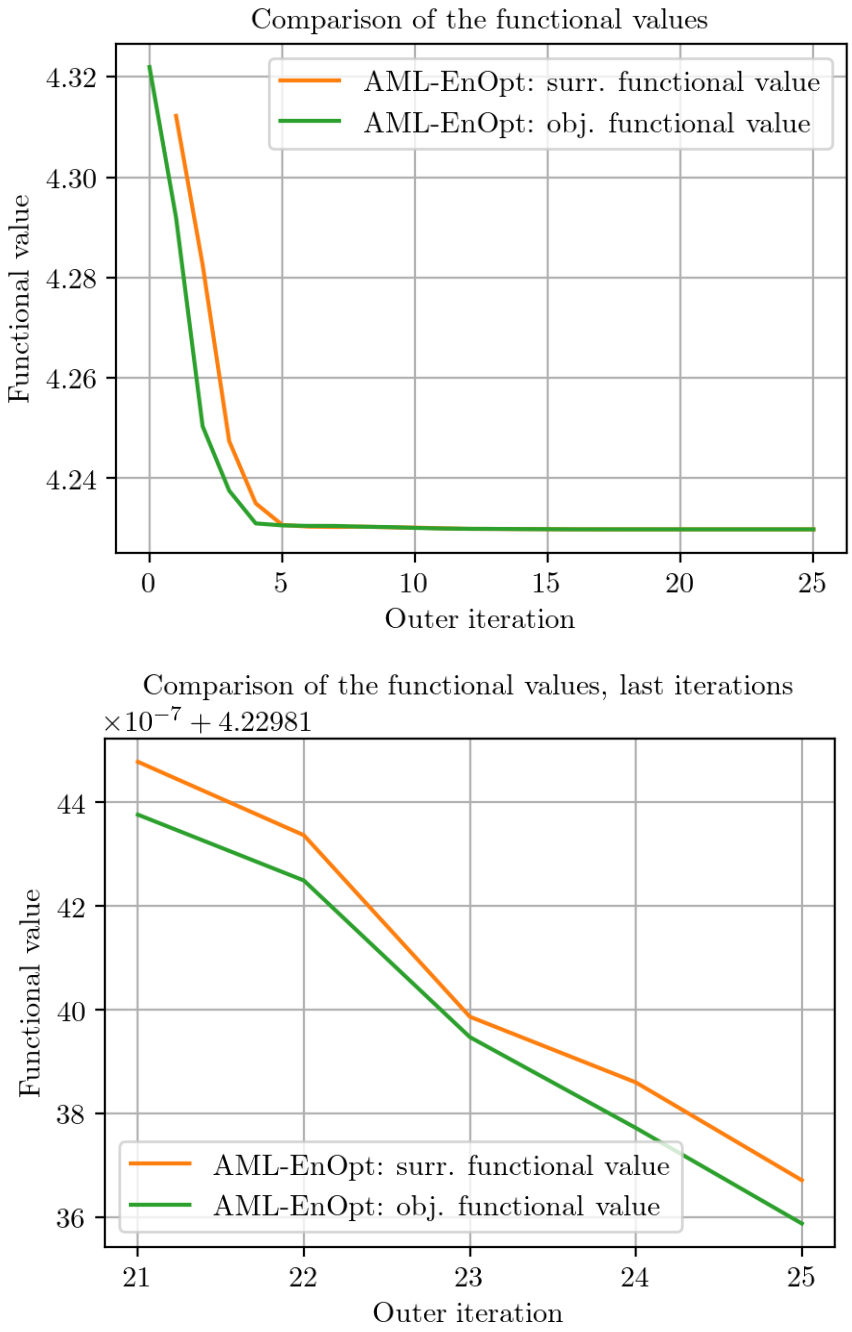

Figure 5.3: Comparison of the FOM objective functional values obtained during the outer iterations of the AML-EnOpt algorithm, as well as the respective surrogate functional values

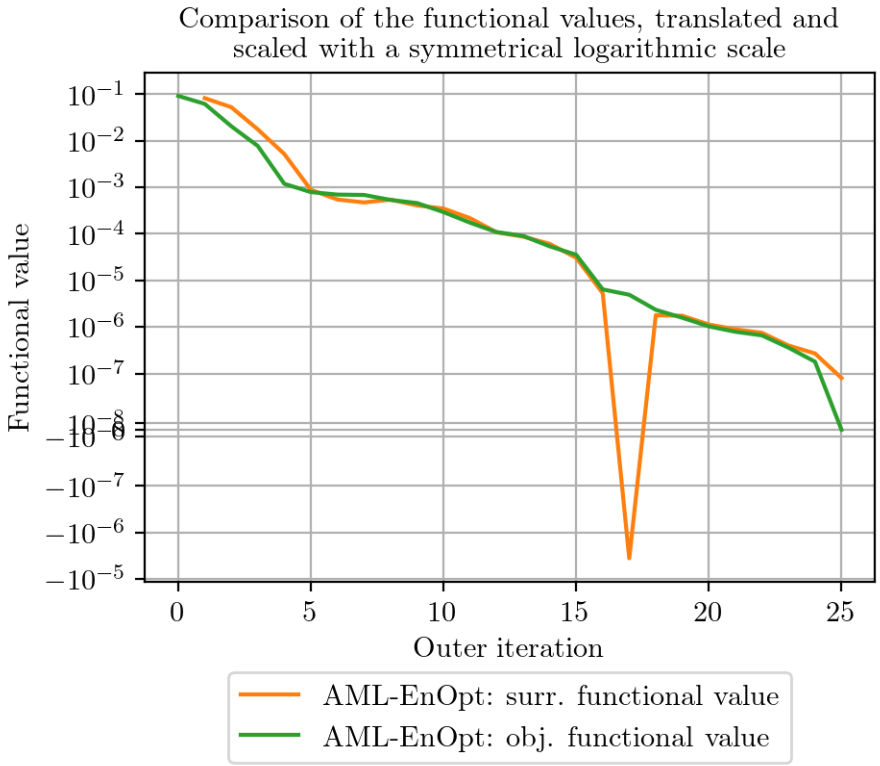

Figure 5.4: Comparison of the FOM objective functional values obtained during the outer iterations of the AML-EnOpt algorithm, as well as the respective surrogate functional values, translated by the objective functional value of the AML-EnOpt output and scaled with a symmetrical logarithmic scale which is linear at values that are closer to zero than the absolute value of the difference between the objective and surrogate functional values of the AML-EnOpt output.

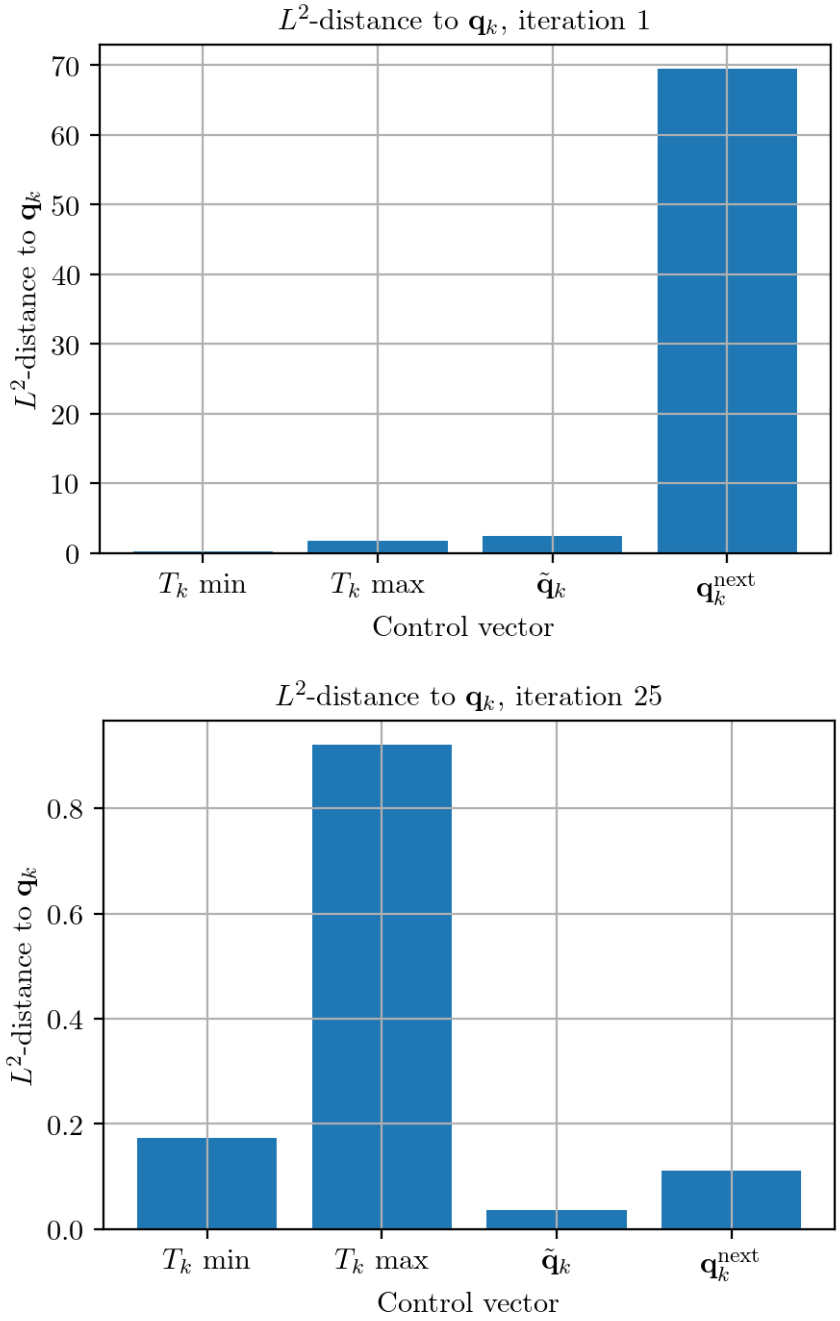

Figure 5.5: L^2-distance between $\mathbf{q}_k$ and T_k min, T_k max, $\tilde{\mathbf{q}}_k$ and $\mathbf{q}_k^{\text{next}}$ at the first (top) and last (bottom) outer iteration. $\mathbf{q}_k$ is here the iterate at the start of the respective iteration and $\mathbf{q}_k^{\text{next}}$ the iterate at the end of the iteration. T_k min is the sample in T_k with the minimum L^2-distance to $\mathbf{q}_k$ and T_k max the sample with the maximum L^2-distance to $\mathbf{q}_k$.

10^{-7}. This gives us an objective functional value of approximately 4.22981275 which is reached after a runtime of only 1.28 minutes. Both the objective functional value and the runtime of the L-BFGS-B optimizer are smaller than the values that we get from the FOM- and AML-EnOpt algorithms. Therefore, we conclude that the L-BFGS-B aglorithm is a better optimization method for this optimization problem than the EnOpt algorithms.

To measure the performance of the FOM- and AML-EnOpt algorithms, we run these procedures three times each. Entries in the columns 'obj. func. absolute error $(\cdot 10^{-7})$' and 'obj. func. relative error $(\cdot 10^{-7})$' state the scaled absolute, respectively relative, error between the objective functional value of the respective EnOpt solution and the objective functional value of the L-BFGS-B solution. These entries have to be multiplied by 10^{-7} to obtain the correct errors. In this table, the objective functional values of the AML-EnOpt procedures are all smaller and much closer to the L-BFGS-B objective functional value than the results of the FOM-EnOpt algorithm. Since the numbers of outer iterations of the AML-EnOpt procedures are significantly lower than those of the FOM-EnOpt procedures, their respective numbers of FOM evaluations are also considerably smaller. This leads to a decrease of the total run times compared to the FOM-EnOpt procedures. In our results, the training time of the AML-EnOpt procedures is more than half as long as their total run time.

Figure 5.6 shows the solutions that we get from the FOM-EnOpt and the Adaptive-ML-EnOpt algorithms, as well as the analytical solution, the L-BFGS-B solution and the initialization of the iterates at the top. At the bottom, the difference between the FOM-EnOpt, respectively AML-EnOpt, solution and the analytical solution is depicted. The plot at the bottom shows us that the two solutions strategies yield similar results. At most points, the solutions of both algorithms are relatively close to the analytical solution. However, the control values for the second, last and especially first time step are far from the analytically optimal values. The optimal control vector that is returned by the L-BFGS-B algorithm and its difference with the analytical solution is shown in Figure 5.7. This solution has similar traits as the EnOpt solutions.

A reason for the large difference between the solutions from the optimization algorithms and the analytical solution might be inaccuracies of the state variable discretizations. The objective functional is calculated according to subsection 2.2.4 as

$$\begin{aligned}\frac{T}{6N_t}\sum_{m=1}^{N_t} &\left(\mathbf{U}_{m-1}-\hat{\mathbf{U}}_{m-1}\right)\mathbf{M}_n\left(\mathbf{U}_{m-1}-\hat{\mathbf{U}}_{m-1}\right)\\ &+\left(\mathbf{U}_{m-1}-\hat{\mathbf{U}}_{m-1}\right)\mathbf{M}_n\left(\mathbf{U}_{m}-\hat{\mathbf{U}}_{m}\right)\\ &+\left(\mathbf{U}_{m}-\hat{\mathbf{U}}_{m}\right)\mathbf{M}_n\left(\mathbf{U}_{m}-\hat{\mathbf{U}}_{m}\right)\\ +\frac{\alpha T}{6N_t}\sum_{m=1}^{N_t} &\mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_{m-1}+\mathbf{Q}_{m-1}\mathbf{M}_n\mathbf{Q}_m+\mathbf{Q}_m\mathbf{M}_n\mathbf{Q}_m,\end{aligned} \tag{5.8}$$

where the notation corresponds to the notation of subsection 2.2.4.

We compute the state vectors $\mathbf{U}_m$ with the Crank-Nicolson scheme (2.7):

$$\left(\tilde{\mathbf{M}}_n^T+\frac{T}{2N_t}\tilde{L}_n^T\right)\mathbf{U}_m=\left(\tilde{\mathbf{M}}_n^T-\frac{T}{2N_t}\tilde{\mathbf{L}}_n^T\right)\mathbf{U}_{m-1}+\frac{T}{2N_t}\mathbf{F}_{m-1}+\frac{T}{2N_t}\mathbf{F}_m,$$

for $m=1,\dots,N_t$. The control q_1^m influences only the vector $\mathbf{F}_m$ directly for $m=0,\dots,N_t$.

Table 5.2: Comparison of the results from the FOM-EnOpt and AML-EnOpt algorithms

Method	Result	FOM objective functional value	Surrogate functional value
FOM – EnOpt	1	4.22982619	-
	2	4.22982927	-
	3	4.22983618	-
AML – EnOpt	1	4.2298133	4.22981349
	2	4.2298136	4.22981371
	3	4.22981313	4.22981329

Method	Result	obj. func. absolute error ($\cdot 10^{-7}$)	obj. func. relative error ($\cdot 10^{-7}$)
FOM – EnOpt	1	134.45	31.79
	2	165.20	39.06
	3	234.32	55.40
AML – EnOpt	1	5.48	1.30
	2	8.47	2.00
	3	3.77	0.89

Method	Result	Outer iterations	Inner iterations
FOM – EnOpt	1	119	-
	2	114	-
	3	124	-
AML – EnOpt	1	27	924
	2	27	877
	3	24	895

Method	Result	FOM evaluations	Surrogate evaluations
FOM – EnOpt	1	12107	-
	2	11629	-
	3	12643	-
AML – EnOpt	1	2912	95047
	2	2905	90241
	3	2589	91967

Method	Result	Total run time (min)	Training time (min)
FOM – EnOpt	1	37.67	-
	2	36.07	-
	3	39.22	-
AML – EnOpt	1	24.05	14.24
	2	23.87	14.17
	3	22.14	13.39

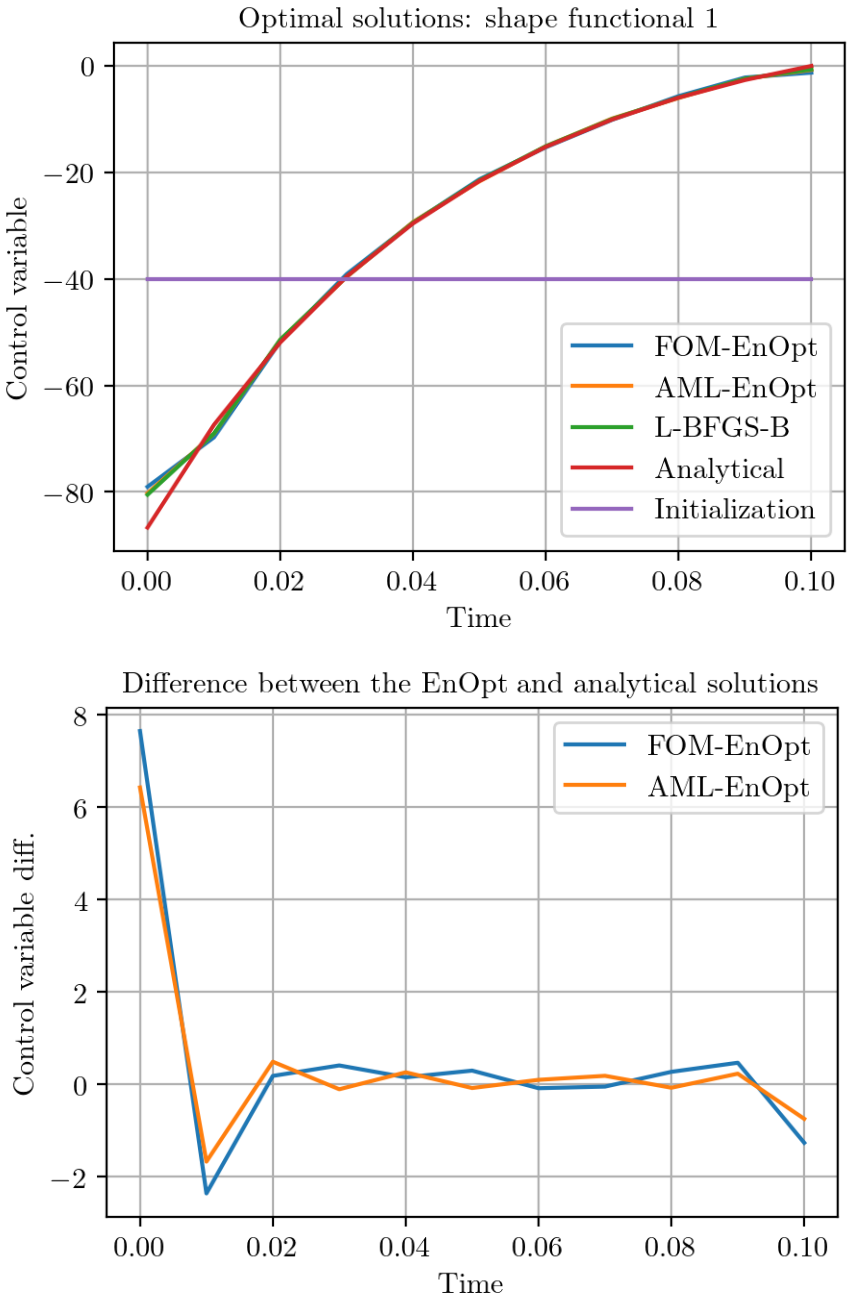

Figure 5.6: Comparison of the initial value of the iterate and the optimal solutions obtained from the FOM-EnOpt, AML-EnOpt and L-BFGS-B algorithms (top) and the differences between the EnOpt solutions and the analytical solution (bottom)

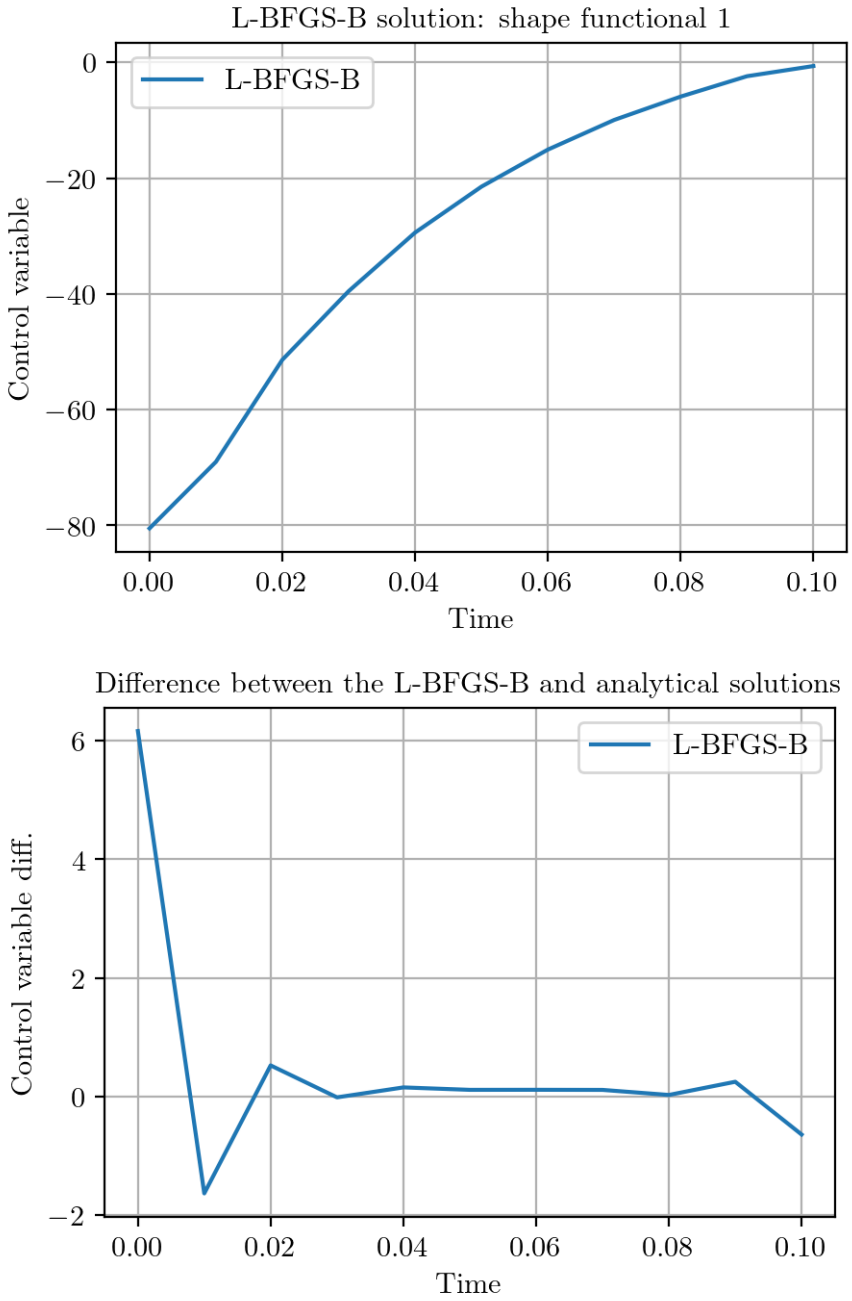

Figure 5.7: The optimal solution obtained from the L-BFGS-B algorithm (top) and its difference with the analytical solution (bottom)

Hence, for $m = 1, \ldots, N_t - 1$, each control q_1^m is used for the definitions of the states $\mathbf{U}_m$ and $\mathbf{U}_{m+1}$. Only q_1^0 and $q_1^{N_t}$ set just one state which is $\mathbf{U}_1$, respectively $\mathbf{U}_{N_t}$, while they are still weighted with the same value of $\frac{T}{2N_t}$ as every other control. Therefore changes of q_1^0 and $q_1^{N_t}$ tend to effect the first part of (5.8) less than the other control variables. Hence, the control variables may reduce the regularization term in (5.8) instead by going closer to zero. This can be seen for q_1^0. The first control of the analytical solution is approximately $\bar{q}_1^0 \approx -87$ which is far from zero. Therefore, with the discretization, it might be beneficial to reduce the regularization term by increasing q_1^0. Since the difference q_1^0 and $\bar{q}_1^0$ is so large, it is not surprising that q_1^1 is also not a good approximation of $\bar{q}_1^1$. In contrast to q_1^0, $q_1^{N_t}$ is further away from zero than its respective control of the analytical solution. However, due to the quadratic nature of the regularization term, this has not such a strong effect on the objective functional. To reduce these inaccuracies, we could increase the number of time steps. This is examined further below.

So far, we have compared the EnOpt solutions with the analytical solution. However, the analytical solution is not the optimum. The objective functional value of L-BFGS-B solution is smaller than the respective functional values of the EnOpt and analytical solutions. Hence, we now compare the EnOpt results with the output of the L-BFGS-B optimizer. The difference between EnOpt results and the L-BFGS-B solution is depicted in Figure 5.8. We can see here that the AML-EnOpt solution with an average absolute error of approximately 0.09 is closer to the L-BFGS-B solution than the FOM-EnOpt result is which has an average absolute error of about 0.42. This is consistent with the fact that the objective functional value of the AML-EnOpt solution is smaller that the objective functional value of the FOM-EnOpt solution.

To investigate the effects of different neural network structures, we test now the AML-EnOpt algorithm with different quantities of neurons in the hidden layer. The number of hidden layers is fixed to two. The progression of the FOM objective functional values for different DNN structures is shown in Figure 5.9. To distinguish the different results, the bottom plot shows the objective functional values of the last outer iterations. Since the procedure with 1000 neurons in the hidden layer has here many more outer iterations than the rest, we do not show this plot to make the differences between the other results clearer. The number of outer iterations, as well as the minimum, maximum, and average training and validation losses are shown in Table 5.3. The values in the Table 5.4, multiplied with 10^{-5} and added to 4.2298, are the FOM objective functional values that the outputs of the respective procedures yield.

The initial shape functional control variance σ_1^2 and the constant correlation factor ρ have a strong effect on the runtime and output of the FOM-EnOpt and AML-EnOpt algorithms. If the variance is too large, the FOM-EnOpt procedure tends to give a worse optimal solution. If the variance is too small, this algorithm needs more optimization steps than necessary to terminate.

We choose the correlation factor to be close to one so that we get a smoother output. However, if this value is too large, the result is similar to a large σ_1^2 which we want to avoid. For example, if we set $\rho = 0.9$, we get from (3.1) that the variance of q_1^i is approximately $5.3 \cdot \sigma_1^2$ for $i = 0, \ldots, N_t$. If ρ is equal to 0.99, the variance is with a value of approximately

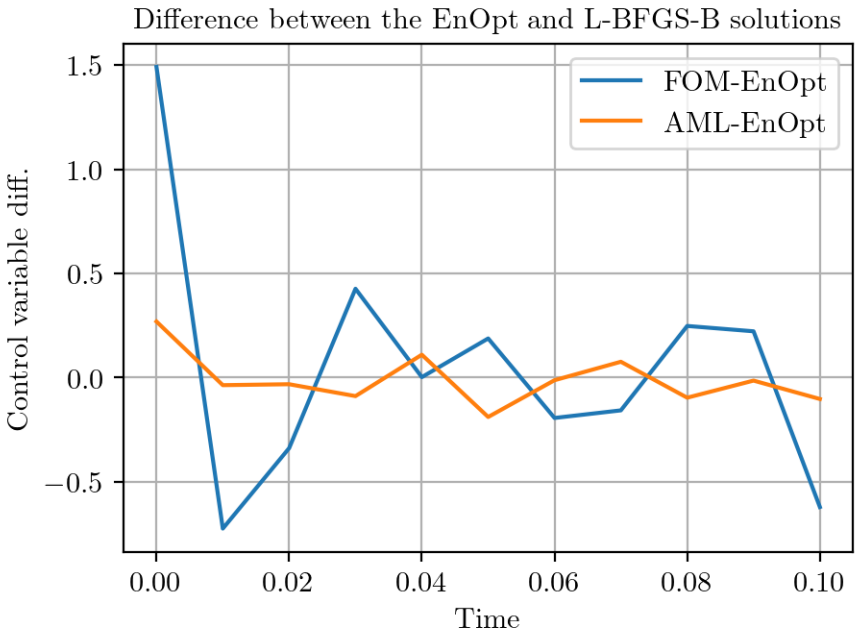

Figure 5.8: The difference between the EnOpt solutions and the solution obtained from the L-BFGS-B optimizer

Table 5.3: Minimum, maximum, and average MSE loss on the training and validation set during the AML-EnOpt procedure with different numbers of neurons in the hidden layers of the neural network. The number of hidden layers is fixed to two.

Neurons	Outer	Training loss			Validation loss		
$N_1 = N_2$	iter.	Min.	Max.	Avg.	Min.	Max.	Avg.
15	23	$2.1 \cdot 10^{-7}$	$2.6 \cdot 10^{-4}$	$2.1 \cdot 10^{-5}$	$7.3 \cdot 10^{-7}$	$1.4 \cdot 10^{-3}$	$2.5 \cdot 10^{-4}$
20	25	$3.3 \cdot 10^{-7}$	$7.7 \cdot 10^{-4}$	$6.2 \cdot 10^{-5}$	$1.9 \cdot 10^{-6}$	$2.9 \cdot 10^{-3}$	$4.7 \cdot 10^{-4}$
25	26	$4.7 \cdot 10^{-7}$	$9.3 \cdot 10^{-5}$	$1.2 \cdot 10^{-5}$	$1.0 \cdot 10^{-6}$	$4.2 \cdot 10^{-4}$	$1.0 \cdot 10^{-4}$
30	30	$3.0 \cdot 10^{-7}$	$1.1 \cdot 10^{-4}$	$9.4 \cdot 10^{-6}$	$3.9 \cdot 10^{-7}$	$1.1 \cdot 10^{-3}$	$2.1 \cdot 10^{-4}$
35	33	$2.5 \cdot 10^{-7}$	$5.7 \cdot 10^{-4}$	$2.2 \cdot 10^{-5}$	$9.3 \cdot 10^{-7}$	$1.6 \cdot 10^{-3}$	$2.0 \cdot 10^{-4}$
50	28	$3.9 \cdot 10^{-7}$	$3.3 \cdot 10^{-4}$	$2.3 \cdot 10^{-5}$	$5.2 \cdot 10^{-7}$	$3.6 \cdot 10^{-4}$	$9.3 \cdot 10^{-5}$
100	31	$5.7 \cdot 10^{-7}$	$6.5 \cdot 10^{-5}$	$1.2 \cdot 10^{-5}$	$1.3 \cdot 10^{-6}$	$6.1 \cdot 10^{-4}$	$1.6 \cdot 10^{-4}$
250	18	$2.9 \cdot 10^{-7}$	$7.2 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$8.4 \cdot 10^{-7}$	$5.2 \cdot 10^{-4}$	$8.6 \cdot 10^{-5}$
500	20	$5.3 \cdot 10^{-7}$	$1.3 \cdot 10^{-4}$	$1.6 \cdot 10^{-5}$	$6.8 \cdot 10^{-7}$	$4.1 \cdot 10^{-4}$	$8.9 \cdot 10^{-5}$
1000	37	$4.0 \cdot 10^{-7}$	$9.0 \cdot 10^{-5}$	$1.3 \cdot 10^{-5}$	$1.4 \cdot 10^{-6}$	$4.2 \cdot 10^{-4}$	$9.0 \cdot 10^{-5}$

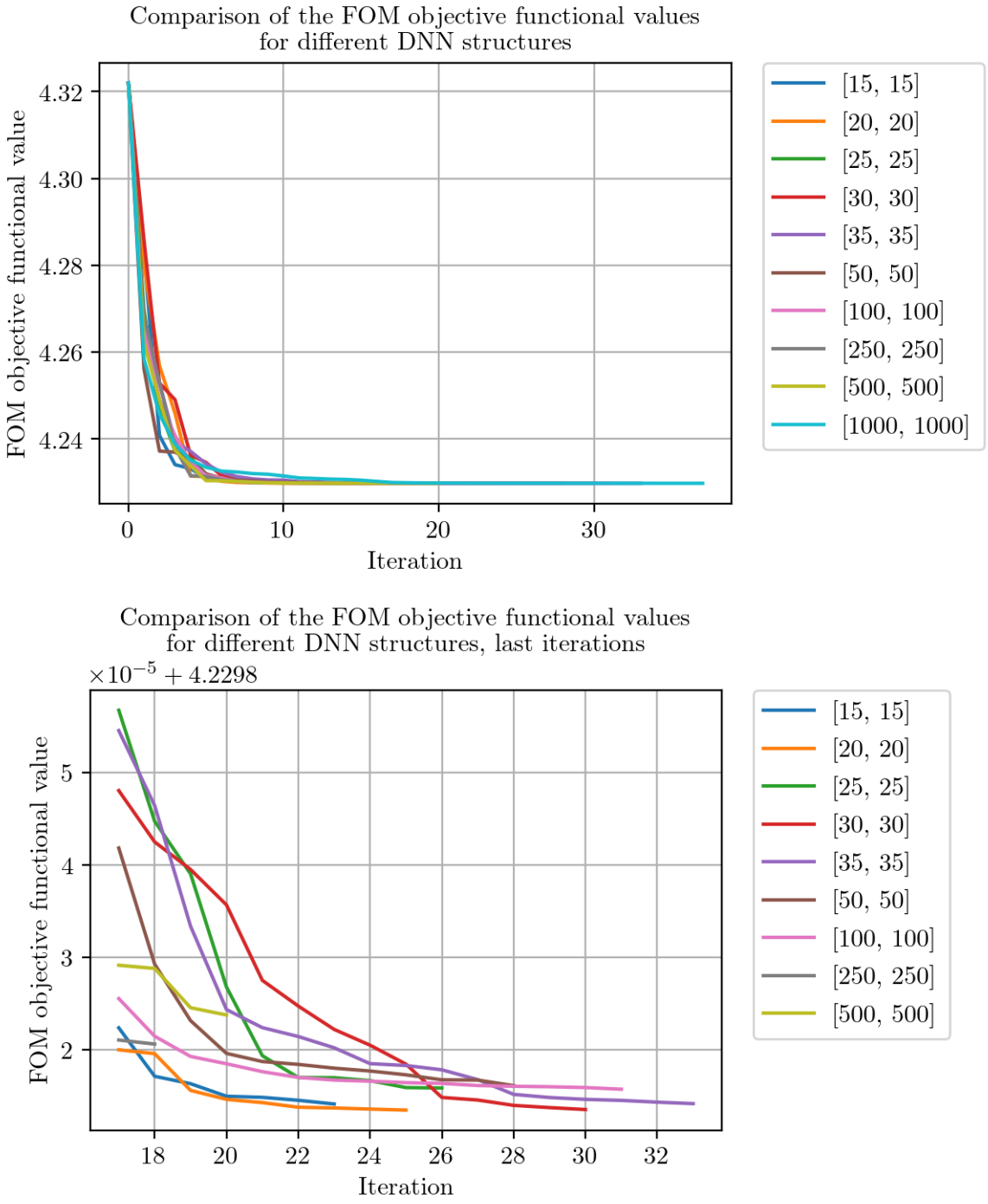

Figure 5.9: Comparison of the FOM objective functional values from the AML-EnOpt algorithm for different numbers of neurons in the hidden layers. The plot at the bottom shows the functional values of the last outer iterations without the result with 1000 neurons in the hidden layers.

Table 5.4: FOM objective functional output values of the AML-EnOpt procedure with different numbers of neurons in the hidden layers of the neural network. The number of hidden layers is fixed to two.

Neurons ($N_1 = N_2$)	15	20	25	30	35	50	100	250	500	1000
FOM obj. func. val. ($\cdot 10^{-5} + 4.2298$)	1.41	1.35	1.59	1.35	1.42	1.61	1.57	2.06	2.38	1.63
Abs. err. wrt. L-BFGS-B sol. ($\cdot 10^{-6}$)	1.38	0.72	3.11	0.78	1.42	3.37	2.98	7.87	11.02	3.57
Rel. err. wrt. L-BFGS-B sol. ($\cdot 10^{-7}$)	3.27	1.70	7.36	1.83	3.35	7.96	7.05	18.61	16.05	8.44
Training time (min)	14.01	15.39	15.17	22.00	20.99	13.40	17.95	7.06	6.59	16.25
Total run time (min)	22.44	25.02	25.23	35.60	37.66	25.51	35.09	16.69	14.51	29.56

$50.3 \cdot \sigma_1^2$ almost ten times as high.

A poor choice of these values may have even more serious consequences for the FOM-EnOpt algorithm. If we set the variance too small, the neural network-based surrogate functional is not approximating the FOM objective functional sufficiently well in a large enough area and the AML-EnOpt algorithm fails. However, we still want to set the variance so small that the surrogate is a good approximation of the objective functional in a small area around the current iterate when we are close to the optimal solution.

We conclude that σ_1^2 and ρ should be chosen carefully and dependent on each other. If we increase ρ to get smoother iterates, we might have to decrease σ_1^2 so that the area which contains our samples is not too large. Also, the covariance matrix adaption step size β_2 should be large enough that the samples deviate not too much when the iterates get close to the optimum but not too large since that might result in variances close to zero when the iterate is not close to the optimum. Hence, it might require some testing to find values that suit the available optimization problem.

Unlike the Adaptive-ML-EnOpt algorithm in [4], we need to employ a trust-region method. If we proceed without it, we get the results of Figure 5.10. The method seems to work up to the eighth iteration, but the machine learning-based surrogate suggests in the ninth outer iteration that there is an optimum near a point that is far from the actual optimum. This point is shown in the bottom plot of Figure 5.10. Some entries of the resulting control vector exceed even the value of 600, although they should be below zero.

Now we compare how both algorithms get to their respective solution. During the FOM-EnOpt procedure, the iterates form a curve with a smooth shape early on. Going on, this curve changes only slightly after each iteration, mainly due to vertical stretching. With the Adaptive-ML-EnOpt method, the iterates are relatively close to the optimal point after a

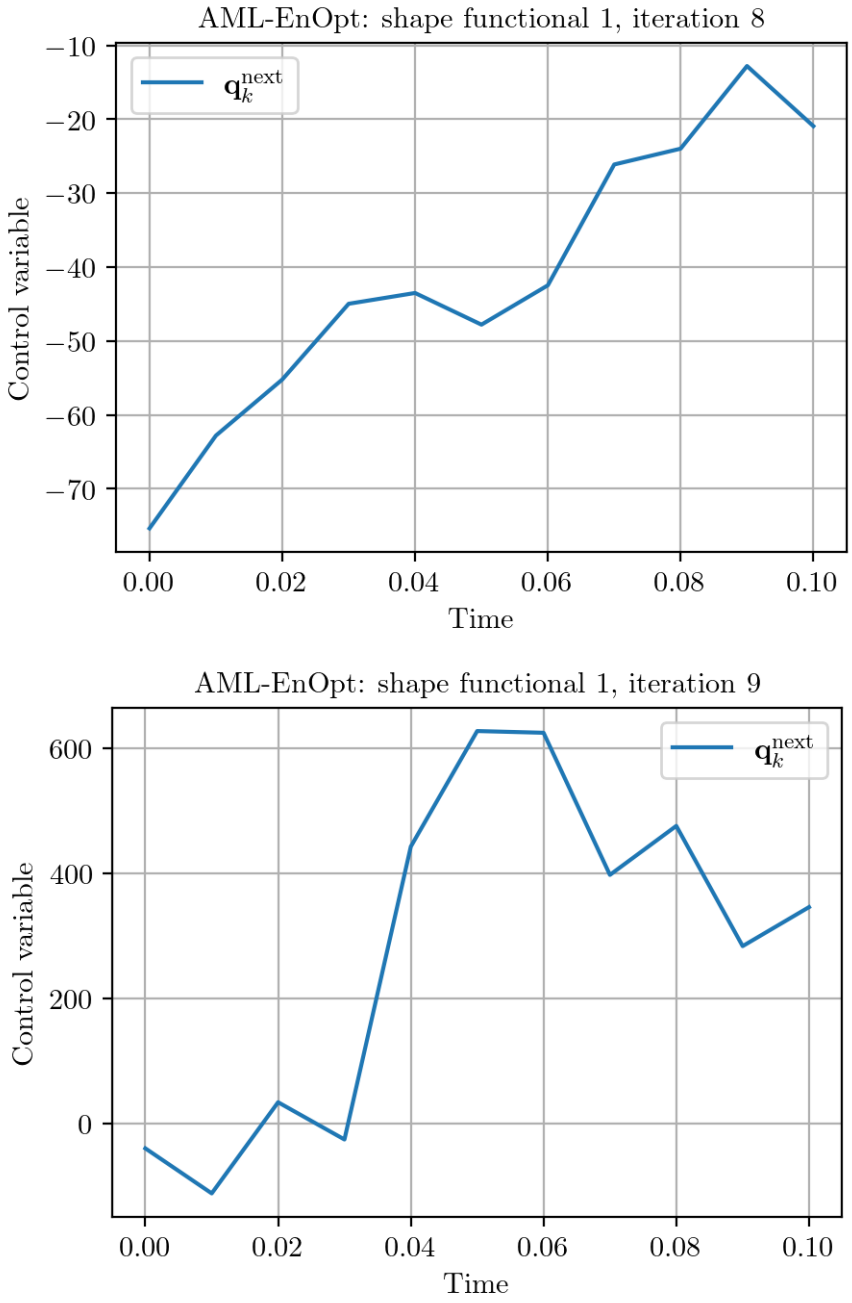

Figure 5.10: An example of iterates after the eighth and ninth outer iteration of an AML-EnOpt procedure if the trust-region method is not applied

few iterations. Therefore, most iterations change the values of the iterates only slightly. To demonstrate this, the plots of the iterates after the fifth outer iteration are shown in Figure 5.11. We see here that the curve of the FOM-EnOpt procedure has a much smoother shape than that of the Adaptive-ML-EnOpt method. However, the values of the FOM method iterate lie in a range between approximately -39 and -35 and are far away from the optimum while their counterparts take a minimum below -70 and a maximum of around 0.

Before we increase the number of time steps, we try out other initializations. Let $\mathbf{q}_0^1$ be the zero vector and $\mathbf{q}_0^2$ a vector where every element has the value -90. The solutions that we get from the FOM and the machine learning-based algorithm when we initialize the first iterate as $\mathbf{q}_0^1$ is depicted in Figure 5.12. The solution with the initialization $\mathbf{q}_0^2$ is shown in Figure 5.13. With an initialization of $\mathbf{q}_0^1$ we get similar results for both initializations. The FOM-EnOpt procedure returns an objective functional value of approximately 4.22982378 after 146 outer iterations. The AML-EnOpt algorithms solution has an objective functional value of about 4.2298234 which is reached after 24 outer iterations. These results are similar to the outcomes of the original initialization. The biggest difference is the number of outer iterations of the FOM-EnOpt procedure. The original initialization needed only 125. The AML-EnOpt procedure with the initialization $\mathbf{q}_0^1$ terminated even one outer iteration earlier than the first initialization, but has a larger objective functional value.

When we initialize these algorithms with $\mathbf{q}_0^2$, we get different results. The AML-EnOpt procedure still yields a similar outcome. It returns an objective functional value of approximately 4.22981306 after 32 outer iterations, so a few more than with the other initializations. The FOM-EnOpt algorithm however, returns a solution that is far off the optimal solution. This iterate is returned after 1000 iterations with an objective functional value of about 4.24305956, so the stopping criterion fulfilled in line 5 of algorithm 1 was $k < k^*$ instead of $F_k > F_k^{\text{prev}} + \varepsilon$. We conclude from this that the quality of the AML-EnOpt output is less dependent on the initialization than that of the FOM-EnOpt algorithm.

Now we increase the number of time steps. That means we set $N_{\mathbf{q}} = 50$ which gives us control vectors of the size 51. This increases the time it takes to compute a FOM objective functional value. Therefore we also increase the time to train the neural network for the surrogate functional by setting the number of training restarts to 10. The other neural network training parameters are like we described it above. The non-DNN-specific parameters are set according to Table 5.5. During the testing we had the problem that the solutions were not smooth enough. Especially the results of the AML-EnOpt algorithm were bad. Therefore we increase the correlation coefficient ρ to 0.99 and thus we must decrease the initial variance σ_1^2 to a smaller number which is here 0.01. Also a decrease of the covariance matrix adaption step size to 0.001 gives us better results. Nevertheless, the FOM- and especially the AML-EnOpt solutions are still not very smooth, as it is shown in Figure 5.14, which depicts these solutions. Still, the AML-EnOpt solution yields a smaller objective functional value as it is shown below.

The FOM-EnOpt procedure terminates after a runtime of 445.99 minutes and its solution has a FOM objective functional value of approximately 4.20986746. The objective functional value of AML-EnOpt solution is about 4.20986518 which is returned after a total runtime of 232.9 minutes and a training time of 146.99 minutes. The L-BFGS-B algorithm terminates after only 7.71 minutes and returns a solution with an objective functional value of approximately 4.209848. So again, the performance of the L-BFGS-B optimizer is much

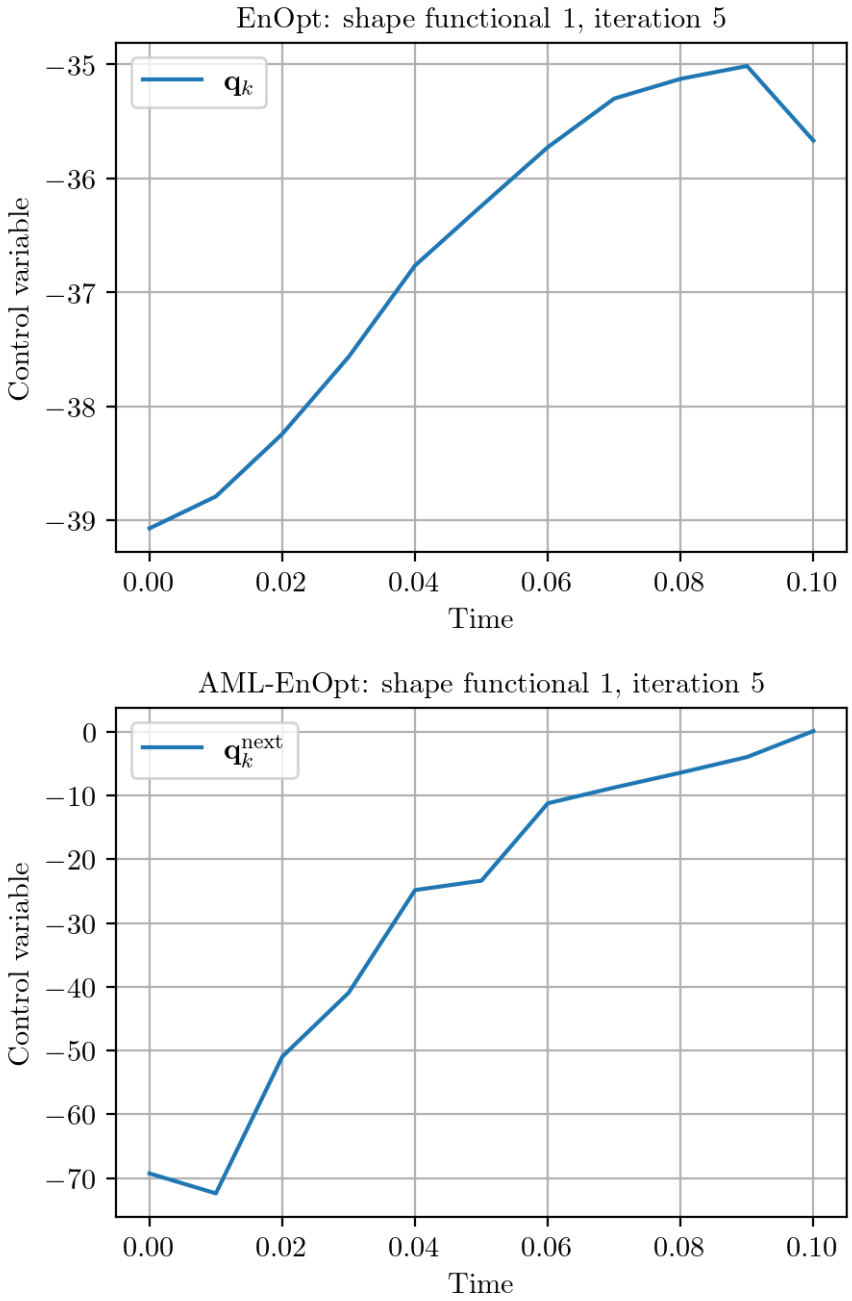

Figure 5.11: The iterates of the FOM-EnOpt (top) and the Adaptive-ML-EnOpt (bottom) algorithms after the fifth iteration

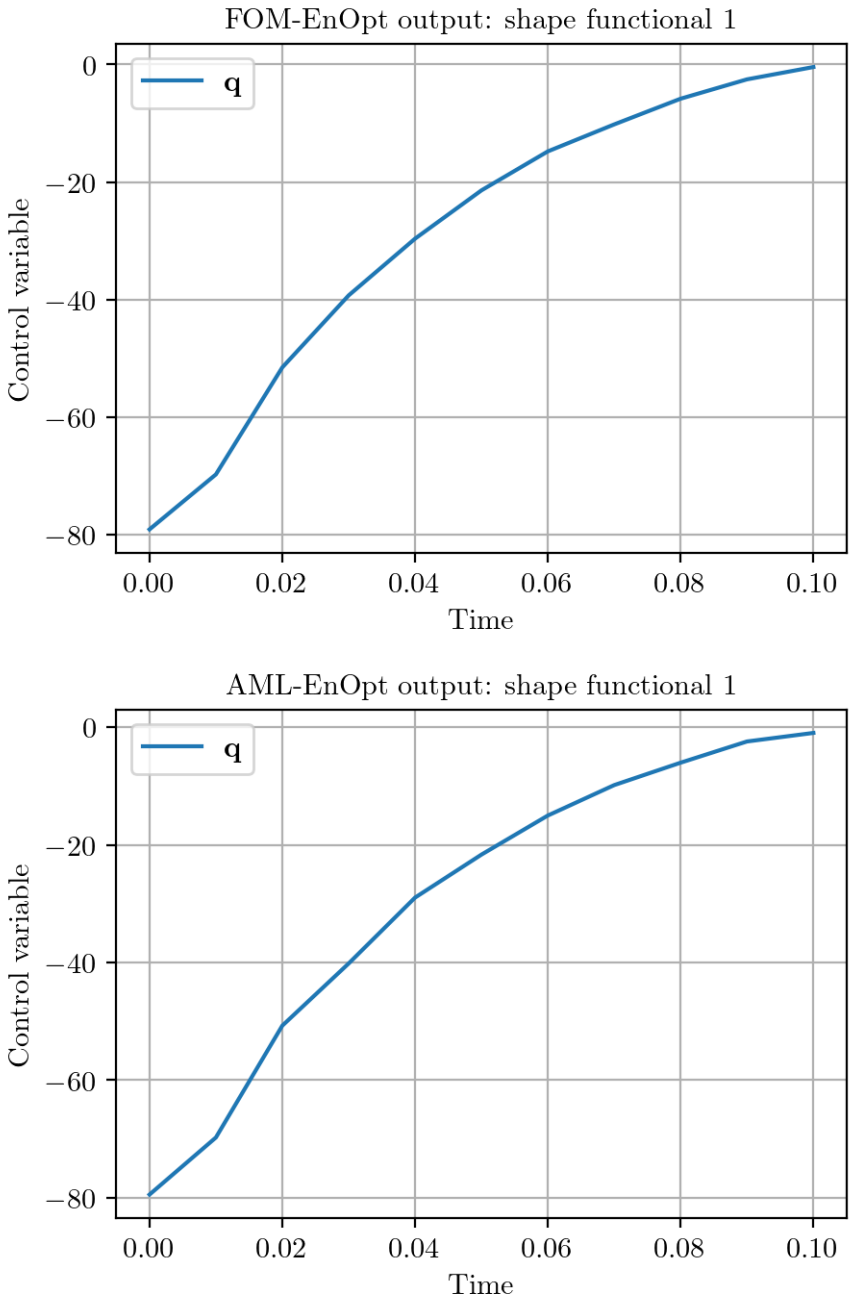

Figure 5.12: Solutions of the FOM-EnOpt (top) and AML-EnOpt (bottom) algorithms with an initialization of $\mathbf{q}_0^1$

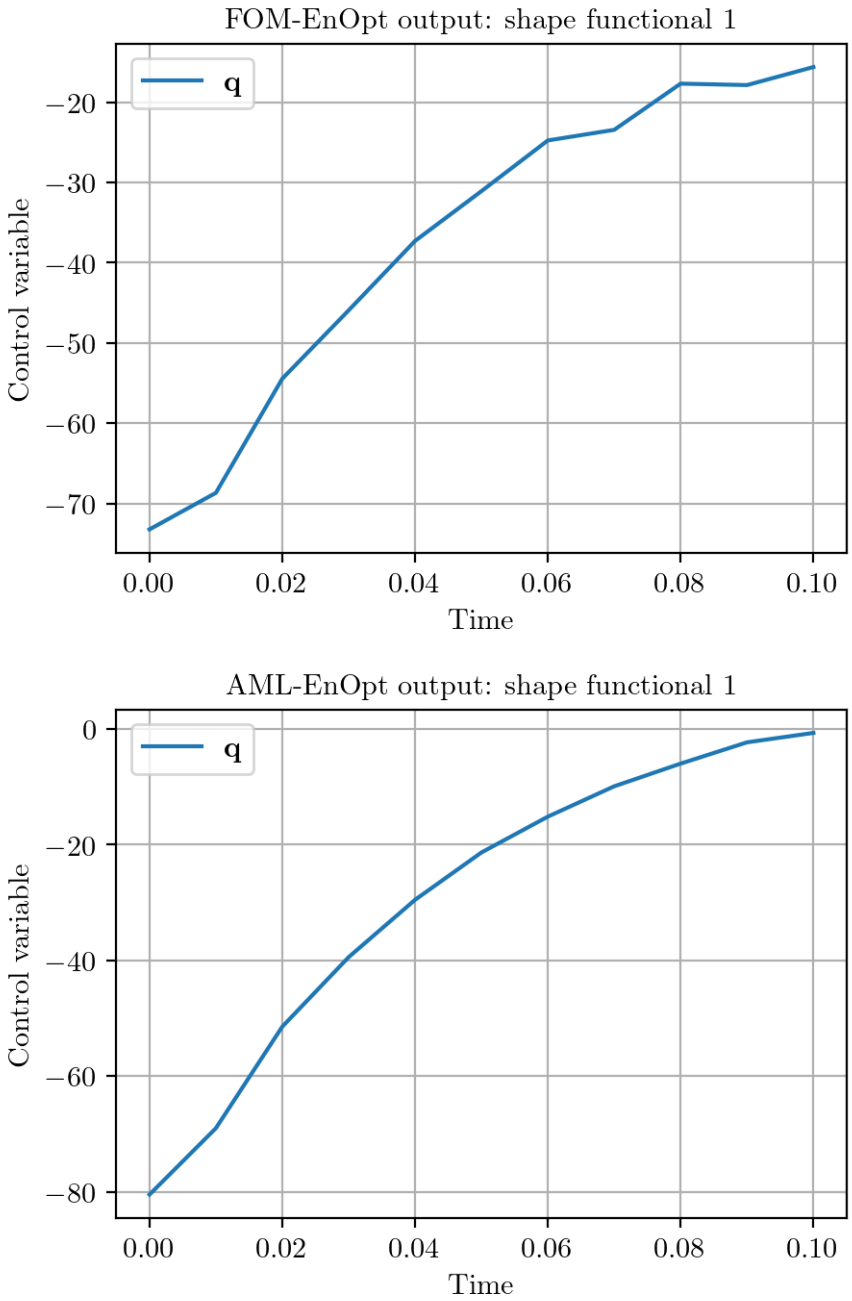

Figure 5.13: Solutions of the FOM-EnOpt (top) and AML-EnOpt (bottom) algorithms with an initialization of $\mathbf{q}_0^2$

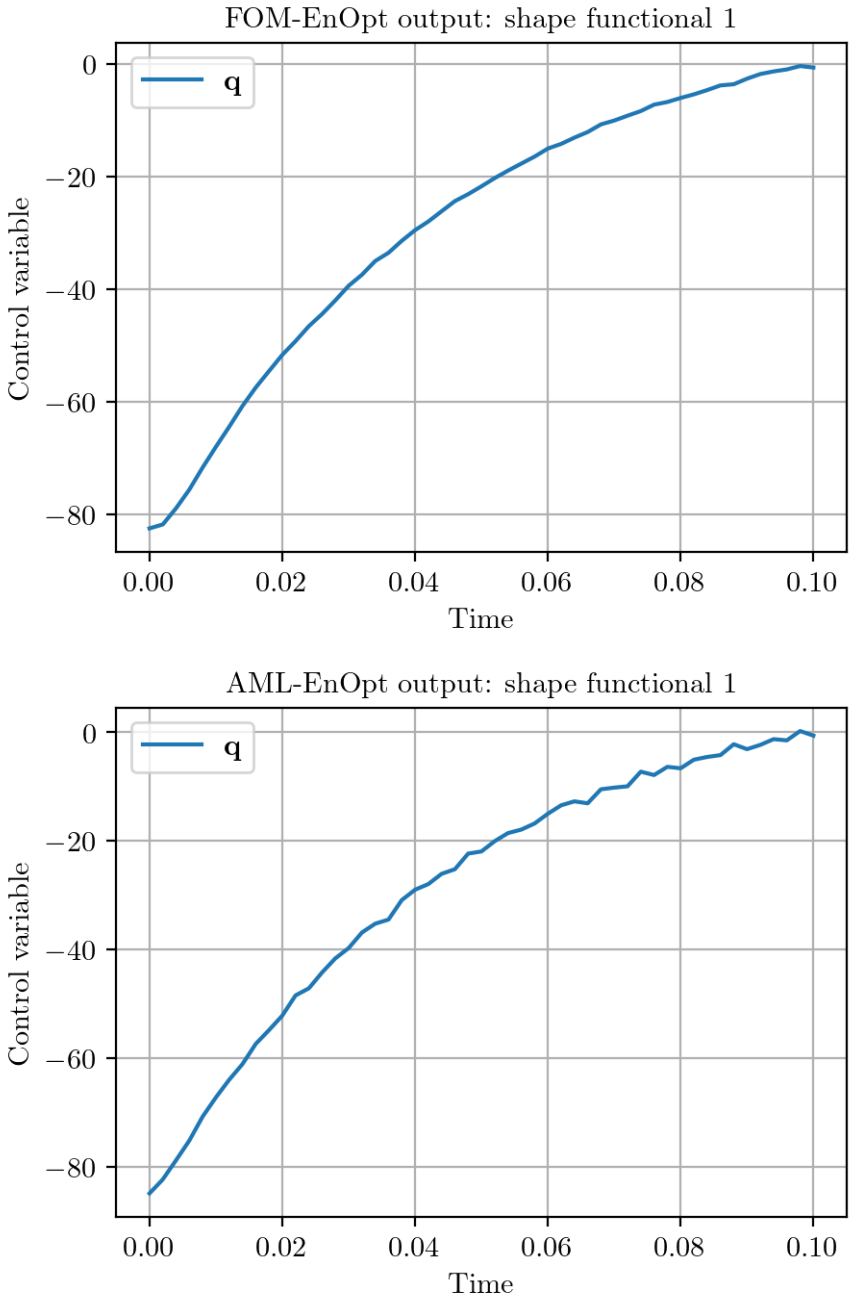

Figure 5.14: Solutions of the FOM-EnOpt (top) and AML-EnOpt (bottom) algorithms with 50 time steps

Table 5.5: Parameters used in the FOM-EnOpt and AML-EnOpt algorithms with 50 time steps

Parameter	Value
Elements of the initial constant control vector $\mathbf{q}_0$	-40
Initial step size β_1	1
Initial covariance matrix adaption step size β_2	0.001
Initial trust-region step size δ_{init}	100
Step size contraction r	0.5
Maximum step size trials ν^*	10
Maximum (outer/ inner) iterations k^*, k_o^*, k_i^*	1000
Maximum trust-region iterations k_{TR}^*	5
Initial control variance σ_1^2	0.01
Constant correlation factor ρ	0.99
Sample size N	100
FOM-EnOpt ε	10^{-14}
Adaptive-ML-EnOpt inner iteration ε_i	10^{-14}
Adaptive-ML-EnOpt outer iteration ε_o	10^{-14}

better than that of the FOM-EnOpt algorithms. As in Figure 5.2, the development of the FOM objective functional values during the runtime of the FOM- and AML-EnOpt algorithms, as well as the progression of the surrogate functional values of the AML-EnOpt procedure are shown in Figure 5.15. The solutions are depicted in Figure 5.16 and the differences with the analytical solution in Figure 5.17. The large differences at the boundaries do still exist, but now they are less noticeable because of the increased number of time steps. The L-BFGS-B optimizer gives here a result that is much closer to the analytical solution than the result of the EnOpt procedures. Figure 5.18 shows the differences between the EnOpt solutions and the L-BFGS-B solution. We see here that the FOM-EnOpt solution with an average absolute difference of about 0.27 is closer to the L-BFGS-B solution than the output of the AML-EnOpt procedure which has an average absolute difference of approximately 0.42. The points in the first time steps are exceptions to this observation. That is probably the reason why the AML-EnOpt objective functional value is smaller than the FOM-EnOpt functional value.

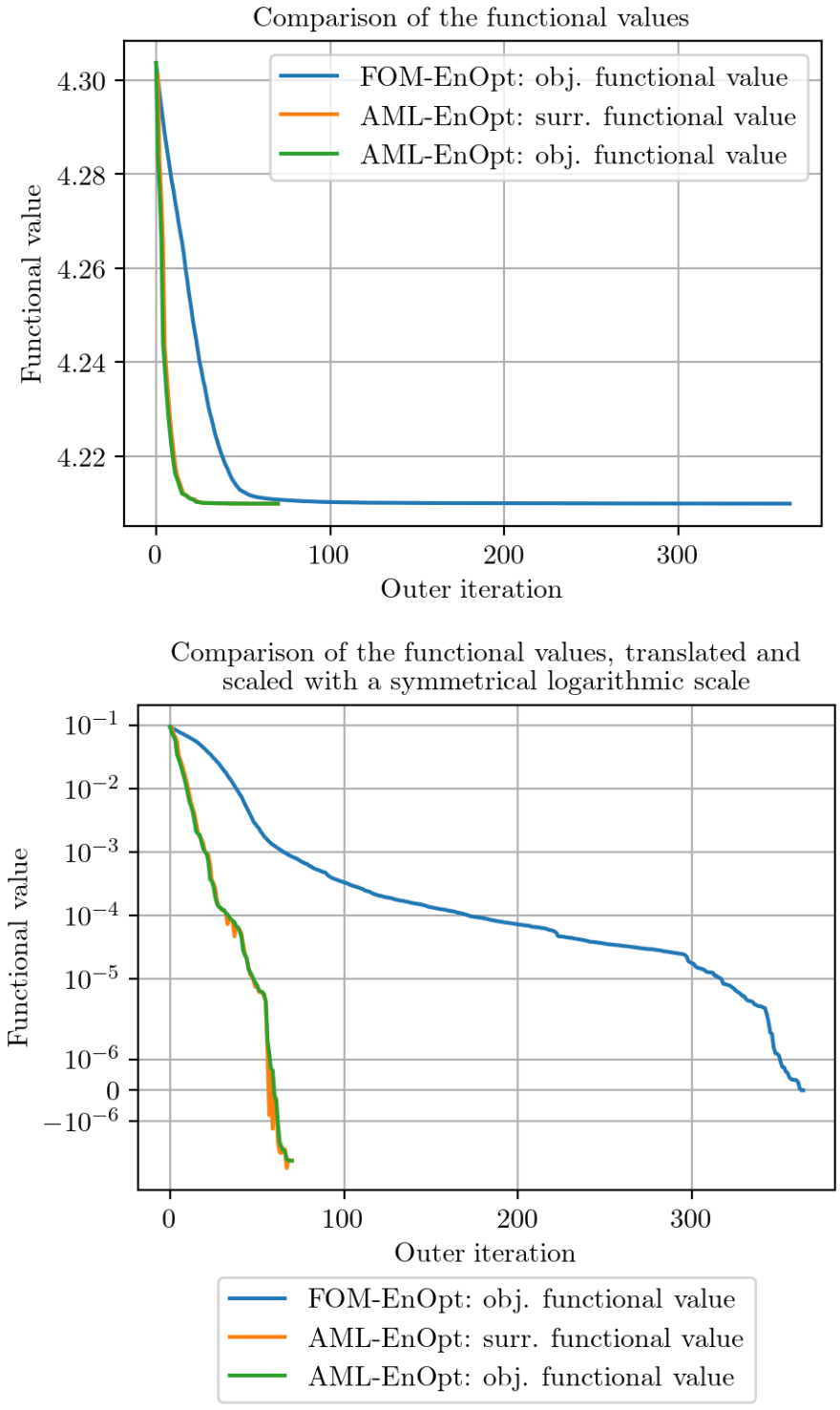

Figure 5.15: The FOM objective functional values obtained during the outer iterations of both EnOpt algorithms and the surrogate functional values obtained during the outer iterations of the AML-EnOpt algorithm with 50 time steps at the top. The plot at the bottom shows these values, translated by the objective functional value of the FOM-EnOpt output and scaled with a symmetrical logarithmic scale as in Figure 5.2.

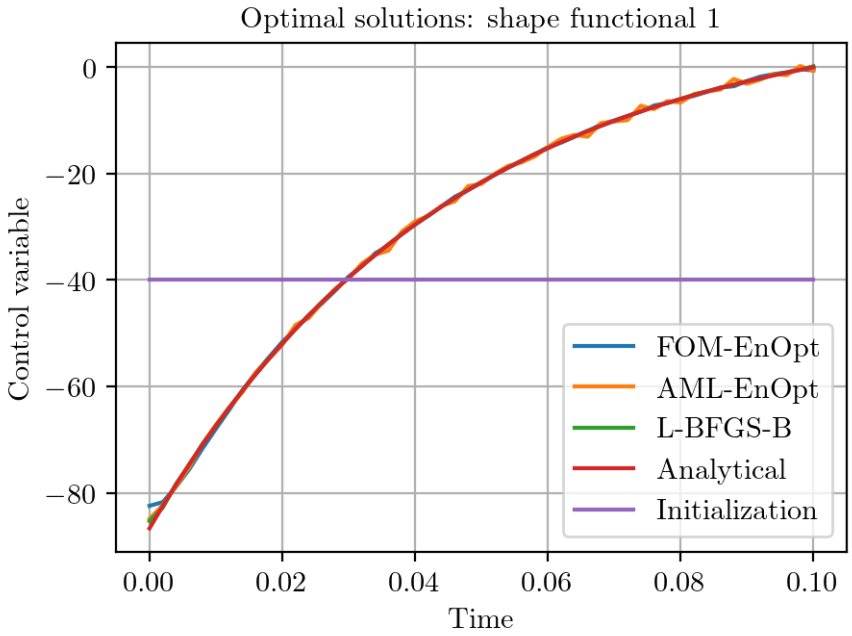

Figure 5.16: Comparison of the initial value of the iterate and the optimal solutions obtained from the FOM-EnOpt, AML-EnOpt and L-BFGS-B algorithms for 50 time steps

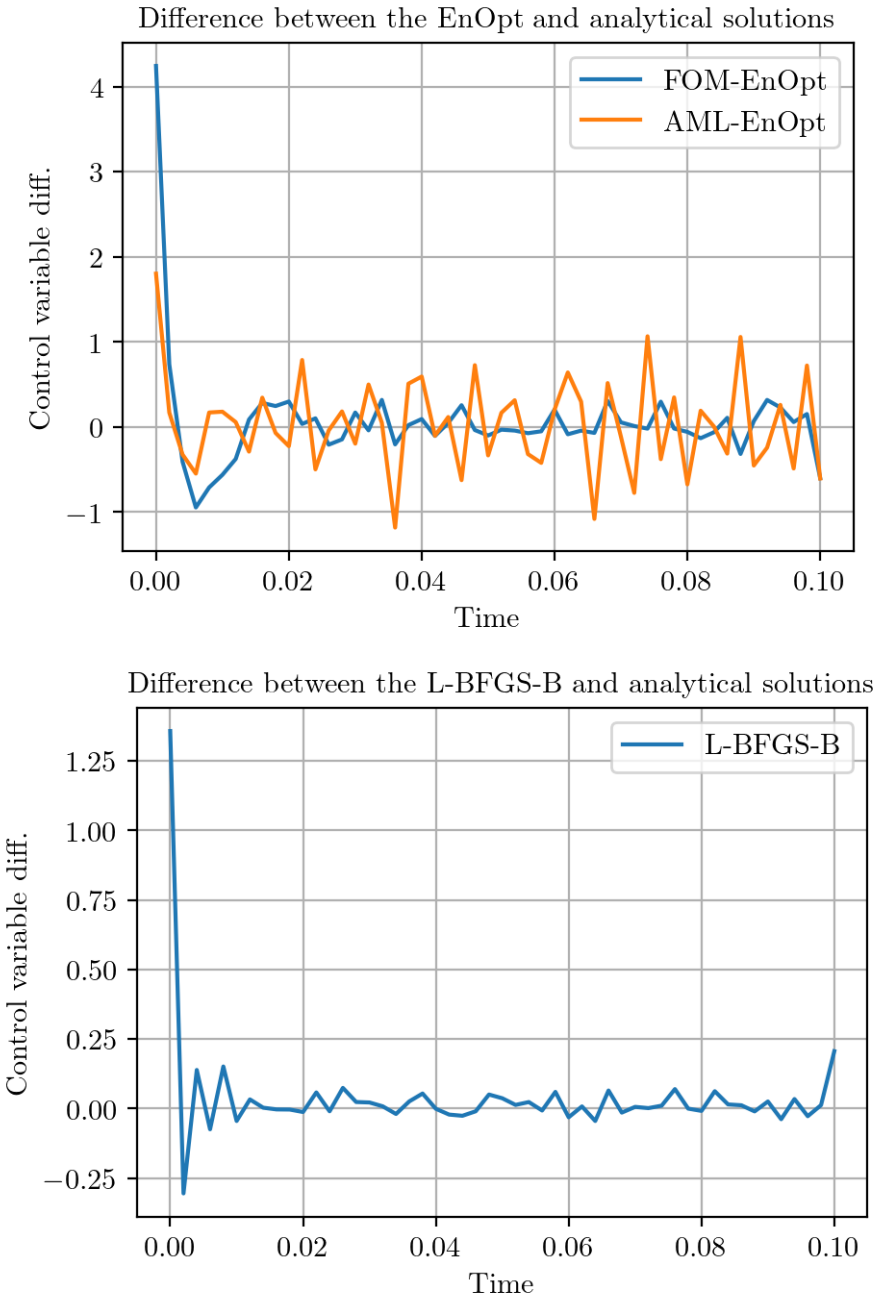

Figure 5.17: Difference between the EnOpt solutions and the analytical solution at the top, and the difference between the L-BFGS-B and analytical solution at the bottom for 50 time steps

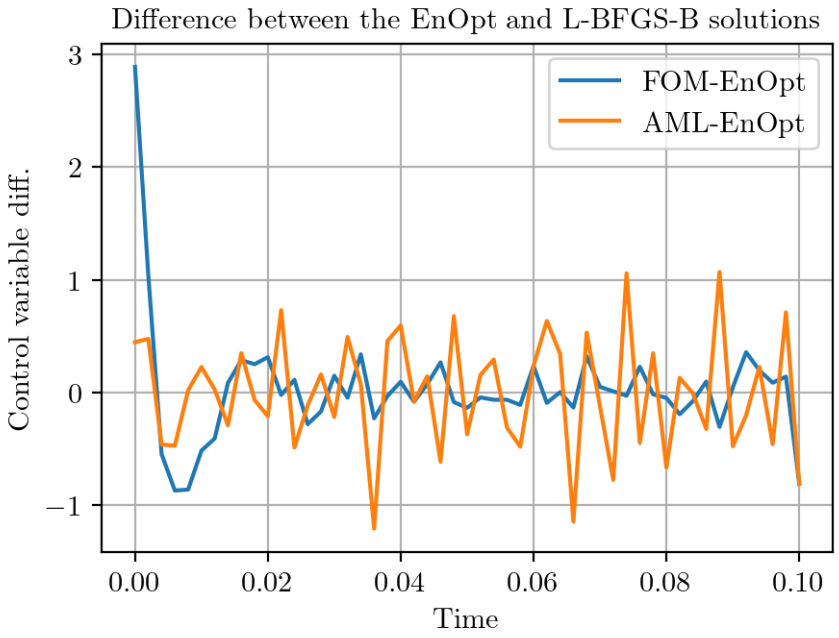

Figure 5.18: Difference between the EnOpt solutions and the L-BFGS-B solution for 50 time steps

Bibliography

[1] Z. Chen, *Reservoir Simulation*. Society for Industrial and Applied Mathematics, 2007. DOI: 10.1137/1.9780898717075. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717075. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898717075.

[2] Y. Chen, D. Oliver, and D. Zhang, "Efficient ensemble-based closed-loop production optimization," *SPE Journal*, vol. 14, Apr. 2013. DOI: 10.2118/112873-MS.

[3] A. S. Stordal, S. P. Szklarz, and O. Leeuwenburgh, "A theoretical look at Ensemble-Based optimization in reservoir management," *Mathematical Geosciences*, vol. 48, no. 4, pp. 399–417, May 2016.

[4] T. Keil, H. Kleikamp, R. J. Lorentzen, M. B. Oguntola, and M. Ohlberger, "Adaptive machine learning-based surrogate modeling to accelerate PDE-constrained optimization in enhanced oil recovery," *Advances in Computational Mathematics*, vol. 48, no. 6, p. 73, Nov. 2022.

[5] A. Abidin, T. Puspasari, and W. Nugroho, "Polymers for enhanced oil recovery technology," *Procedia Chemistry*, vol. 4, pp. 11–16, 2012, The International Conference on Innovation in Polymer Science and Technology, ISSN: 1876-6196. DOI: https://doi.org/10.1016/j.proche.2012.06.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1876619612000034.

[6] D. Meidner and B. Vexler, "A priori error estimates for space-time finite element discretization of parabolic optimal control problems part i: Problems without control constraints," *SIAM Journal on Control and Optimization*, vol. 47, no. 3, pp. 1150–1177, 2008. DOI: 10.1137/070694016. eprint: https://doi.org/10.1137/070694016. [Online]. Available: https://doi.org/10.1137/070694016.

[7] B. Rivière, *Discontinuous Galerkin Methods for Solving Elliptic and Parabolic Equations*. Society for Industrial and Applied Mathematics, 2008. DOI: 10.1137/1.9780898717440. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717440. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898717440.

[8] D. Meidner, "Adaptive space-time finite element methods for optimization problems governed by nonlinear parabolic systems," Jan. 2007.

[9] M. B. Oguntola and R. J. Lorentzen, "Ensemble-based constrained optimization using an exterior penalty method," *Journal of Petroleum Science and Engineering*, vol. 207, p. 109 165, 2021, ISSN: 0920-4105. DOI: https://doi.org/10.1016/j.petrol.2021.109165. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920410521008184.

[10] Y. Zhang, A. S. Stordal, and R. J. Lorentzen, "A natural hessian approximation for ensemble based optimization," en, *Comput. Geosci.*, vol. 27, no. 2, pp. 355–364, Apr. 2023.

[11] L. Prechelt, "Early stopping — but when?" In *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 53–67, ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_5. [Online]. Available: https://doi.org/10.1007/978-3-642-35289-8_5.

[12] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical Programming*, vol. 45, no. 1, pp. 503–528, Aug. 1989.

[13] P. Wolfe, "Convergence conditions for ascent methods," *SIAM Review*, vol. 11, no. 2, pp. 226–235, 1969. DOI: 10.1137/1011036. eprint: https://doi.org/10.1137/1011036. [Online]. Available: https://doi.org/10.1137/1011036.

[14] P. Wolfe, "Convergence conditions for ascent methods. ii: Some corrections," *SIAM Review*, vol. 13, no. 2, pp. 185–188, 1971. DOI: 10.1137/1013035. eprint: https://doi.org/10.1137/1013035. [Online]. Available: https://doi.org/10.1137/1013035.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[16] J. Nocedal and S. Wright, *Numerical Optimization* (Springer Series in Operations Research and Financial Engineering), en, 2nd ed. New York, NY: Springer, Jul. 2006.

[17] R.-É. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, pp. 495–503, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:123458541.

[18] R. Becker, D. Meidner, and B. Vexler, "Efficient numerical solution of parabolic optimization problems by finite element methods," *Optimization Methods and Software*, vol. 22, Oct. 2007. DOI: 10.1080/10556780701228532.