ETH Zürich

Master Thesis

Shape Optimization in Magnetostatics

Author:
Federico Danieli

Supervisor:
Prof. Dr. Ralf Hiptmair

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in

Computational Science and Engineering
Department of Mathematics and Physics

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

August 2015

"Or if he takes whatever dull job he's stuck with... and they are all, sooner or later, dull... and, just to keep himself amused, starts to look for options of Quality, and secretly pursues these options, just for their own sake, thus making an art out of what he is doing, he's likely to discover that he becomes a much more interesting person and much less of an object to the people around him because his Quality decisions change him too. And not only the job and him, but others too because the Quality tends to fan out like waves. The Quality job he didn't think anyone was going to see is seen, and the person who sees it feels a little better because of it, and is likely to pass that feeling on to others, and in that way the Quality tends to keep on going."

Robert M. Pirsig, *Zen and the Art of Motorcycle Maintenance*,

(William Morrow & Company, 1974)

ETH ZÜRICH

Abstract

Computational Science and Engineering
Department of Mathematics and Physics

Master of Science

Shape Optimization in Magnetostatics

by Federico Danieli

The implementation of a solver for a shape optimization problem via pursuing diffeomorphism is described, discussed and tested. The problem at hand consists in finding an optimal shape for a toroidal conductor employed in electromagnetic induction heating. Electric currents inside the conductor are evaluated using a scalar potential approach, recovered solving an underlying *Partial Differential Equation* with a *Finite-Element* discretization that has been tuned to take discontinuities into account. The magnetic field on the object that needs to be heated is instead computed applying *Biot-Savart* law. The shape itself is described in a parametric manner, using a map built on B-spline basis functions. With regards to the optimization algorithm, a *steepest descent* method is employed.

The code is written using the template library BETL2 developed at ETH, and it is here described with a focus on computational performance and memory usage. Relevant parts of the implementation are tested and validated. Numerical convergence results are reported and discussed for various experiments conducted.

Acknowledgements

I would like to thank Prof. Dr. Ralf Hiptmair for his supervision during my work (not to mention all the cookies he offered me during our meetings) which has been crucial, particularly in the last stages. Special thanks go to Alberto Paganini, whose valuable insights and pieces of advice have always been reassuring and have helped me noticeably throughout my last year at ETH. I am also grateful to Elke Spindler, for guiding and assisting me when I was moving my first steps into the huge BETL2 library, to Oded Stein, who provided a part of his code that turned out to be very useful for my validation, and to Mylonas Charilaos, with whom I shared the hardships of this project.

Many thanks to my family: without your support and your sacrifices I would have not been able to be here and live this experience. For this and many other things, I will forever be grateful.

And then there are all the others, and the list is oh, so very long.

Thanks to all those who have accompanied me this far. Thanks to Wookie and Anphi, for listening to my whining when the code was not working, and for keeping me company during the long working days at ETH (well, some of them). Thanks to Nico, for the enlightening nights spent talking about random things. Many thanks to Silvia and Marce, for having been there when I most needed you. Thanks to the *LordoLoft*, and in particular to Rik and Pot, for some of the best moments during these years, and for preserving our precious friendship even though our paths are always diverging. Special thanks to Diana, as well as Vale, for your love and affection: it is good to know that I can always rely on you. Thanks to Ilaria and her family: you have all been such a strong support throughout this whole experience.

And many, many others: all the guys from *The Threads*, and then Bio, Leo, Manny, Azzo... too many to mention them all. Some of you were lost along the way, others stayed until the end, all of you I hope will still be accompanying me for a long time.

Contents

Chapter 1

Introduction

Shape optimization belongs to a particular set of mathematical problems in which the unknown that needs to be recovered is a domain $\Omega \in \mathbb{R}^d$. In general, the optimization process consists in adapting the shape of Ω in such a way that it minimizes a cost functional $\mathcal{J}(\Omega)$. Often we are interested in monitoring a so-called *state variable* defined on the domain, $u(\Omega)$, hence the cost functional might also be expressed in terms of u, in addition to the shape itself: $\mathcal{J}(\Omega, u(\Omega))$. When u can be recovered by solving a Partial Differential Equation (PDE), we are dealing with a PDE-constrained shape optimization problem [3].

This kind of problems finds applications in a broad range of Engineering fields, ranging from aerodynamics to thermodynamics and mechanics. The following examples fall within this category: finding the shape of a wing that minimizes drag effects, the design of a thermal conductor which can optimally dissipate heat, or again finding the shape of an arc which can best support a prescribed load. In these examples, the state variables would be the velocity field, the temperature distribution, or the displacements, respectively.

In our work, an application to magnetic heat induction [4, Chap. 2] is analyzed. The process consists in surrounding an electrically conducting object with a circular-shaped conductor traversed by current. The current induces a magnetic field inside the conductor, which in turn generates eddy currents in the object itself. Consequently, the object temperature rises due to resistive heating. Compared to other heating processes, this procedure has some remarkable advantages, in which no contact with an external heat source is needed, and for this and other reasons it is widely used in industry. Unfortunately, analytic expressions for an optimal shape are limited to very simple model problems and are, in general, not available for complicated geometries or particular applications. This makes it necessary to rely on numerical approximations.

Many numerical schemes for shape optimization are available, and most of them can be gathered into two major groups: *fixed-mesh* methods (see for example [7, 8]), and their counterpart, *moving-mesh* methods [9, 10]. The latter, as the name suggest, try to approach an optimal shape by progressively updating an initial meshed domain. An issue of these methods lies in the fact that they need to recover a way to adequately update the mesh itself, in a manner such that it does not negatively affect its quality. On the other hand, fixed mesh methods mostly make use of a map to transform the original shape into the optimized one. The map itself is then sought via step-by-step updates throughout the whole optimization algorithm.

The work done by R. Hiptmair and A. Paganini in [6] has been used as a guideline for this project, which takes care of extending it to a 3D case and presenting the results of its application. In Chap. 2 a mathematical description of the problem at hand is given, providing a definition of the variables involved, as well as the equations to retrieve them. Chap. 3 deals with an extended definition of derivative, that can be applied to our cost functional. This derivative is employed in the algorithm itself, as described in Chap. 4. Chap. 5 provides some technical aspects related to the discretization methods introduced to recover an approximation of the solution. Chap. 6 gives a more detailed insight of the actual implementation of the code, describing the most relevant classes that have been used, together with their general relationships and purposes. Chap. 7 introduces some of the tests conducted to verify the correctness of the code. Chap. 8 presents an overview of the results from the applications of the code to various examples. Finally, in Chap. 9 are reported some ideas for further expansion and improvements that could be applied to this work.

Chapter 2

Problem definition

The problem set-up is the following. A current flows in a circular path through a conductor, thus creating a magnetic field which is particularly intense in the region encircled by the conductor itself. The object which we intend to heat via magnetic induction is placed inside this region. To improve the efficiency of the heating system, we are interested in modifying the shape of the conductor in such a manner that the magnetic field is more intense on the surface of the object.

In order to provide a proper mathematical formulation of the problem at hand, we need to introduce some concepts, which are described next.

2.1 Cost functional

We are interested in monitoring the magnetic field $\mathbf{H}$ on the surface $\partial\mathcal{D}$ of an object encircled by a conductor Ω. For a sketch of the possible reciprocal positions of Ω and $\partial\mathcal{D}$, see Fig. 2.1. In particular, we consider the component of $\mathbf{H}$ that is tangential to the surface $\partial\mathcal{D}$,

$$\mathbf{H_t} \coloneqq \mathbf{H} - (\mathbf{H} \cdot \mathbf{n})\mathbf{n},$$

with $\mathbf{n}$ being the normal to $\partial\mathcal{D}$. We would like this to be as close as possible to a given tangential target function $\mathbf{p}$. To this purpose, the *cost functional* we introduce takes into account the difference between $\mathbf{H_t}$ and $\mathbf{p}$, and is defined in the following manner:

Definition 2.1. *Cost functional* $\mathcal{J}$

$$\mathcal{J} \coloneqq \int_{\partial\mathcal{D}} \|\mathbf{H_t} - \mathbf{p}\|^2 \, d\mathbf{x}.$$

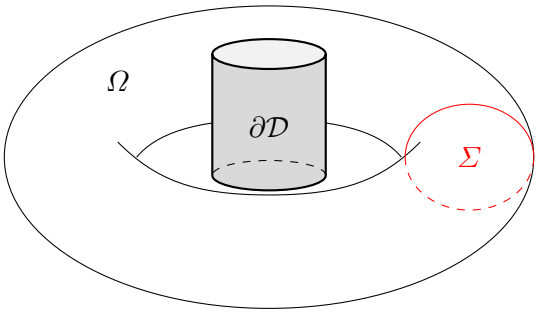

FIGURE 2.1: Sketch of the conductor Ω (here in toroidal shape, as will be used in this work), surrounding the object we intend to heat via magnetic induction, $\partial\mathcal{D}$ (a simple cylinder). The discontinuity surface Σ is a cross-section of the torus itself, and represents the area where the current is imposed.

By means of the *Biot-Savart law*[1] [11, Par. 8.1], Def. 2.1 can be rewritten as

$$\mathcal{J} = \int_{\partial\mathcal{D}} \left\| \frac{\mu_0}{4\pi} \left(\int_{\Omega} -\mathbf{I}(\hat{\mathbf{y}}) \times \nabla_{\mathbf{x}} \left(\frac{1}{\|\mathbf{x}-\hat{\mathbf{y}}\|} \right) d\hat{\mathbf{y}} \right)_t - \mathbf{p} \right\|^2 d\mathbf{x},$$

where $\nabla_{(*)}(\cdot)$ denotes the gradient of $(\cdot)$ with respect to the variable $(*)$ (when misunderstanding could arise), $\mathbf{I}(\hat{\mathbf{y}})$ indicates the electric current flowing inside the conductor Ω, and $(\cdot)_t$ represents the part of $(\cdot)$ that is tangential to the surface $\partial\mathcal{D}$. Introducing the electric potential $\hat{u}(\hat{\mathbf{y}})$ in $\Omega \setminus \Sigma$ and substituting the relation $\mathbf{I}(\hat{\mathbf{y}}) = -\nabla \hat{u}(\hat{\mathbf{y}})$ in the formula above, we obtain

$$\mathcal{J}(\Omega, \hat{u}(\Omega)) = \int_{\partial\mathcal{D}} \left\| \frac{\mu_0}{4\pi} \left(\int_{\Omega\setminus\Sigma} \nabla \hat{u} \times \nabla_{\mathbf{x}} \left(\frac{1}{\|\mathbf{x}-\hat{\mathbf{y}}\|} \right) d\hat{\mathbf{y}} \right)_t - \mathbf{p} \right\|^2 d\mathbf{x}. \tag{2.1}$$

2.2 State equation

Given the domain Ω occupied by the conductor, the resulting electric potential $\hat{u}(\Omega)$ is retrieved by solving the following Partial Differential Equation (PDE):

$$\begin{cases} -\operatorname{div}(\nabla \hat{u}) = 0 & in\ \Omega \setminus \Sigma \\ \nabla \hat{u} \cdot \mathbf{n}_{\partial\Omega} = 0 & on\ \partial\Omega \setminus \Sigma \\ [\hat{u}]_{\Sigma} = const & \\ [\nabla \hat{u} \cdot \mathbf{n}]_{\Sigma} = 0 & \\ \int_{\Sigma} \nabla \hat{u} \cdot \mathbf{n}\, d\mathbf{x} = I & \end{cases}, \tag{2.2}$$

[1] For a more correct physical description, a coupled problem using Maxwell's equations should be solved: this is just an approximation based on the assumption that the magnetic field in $\partial\mathcal{D}$ has no influence on the current distribution in Ω. Nonetheless, neglecting this influence has only a minor impact.

where $\mathbf{n}_{\partial\Omega}$ is the outer normal with respect to the boundary $\partial\Omega$ of the domain Ω, $\mathbf{n}$ is the outer normal with respect to the surface Σ, *const* is an unknown constant, and I indicates the imposed current flux across Σ. In general $[\hat{v}]_\Sigma(\hat{\mathbf{x}})$ denotes the jump of $\hat{v}$ across Σ evaluated at a point $\hat{\mathbf{x}} \in \Sigma$:

$$[\hat{v}]_\Sigma(\hat{\mathbf{x}}) \coloneqq \hat{v}^+(\hat{\mathbf{x}}) - \hat{v}^-(\hat{\mathbf{x}}) = \lim_{\epsilon \to 0} \left[\hat{v}(\hat{\mathbf{x}} + \epsilon\,\mathbf{n}) - \hat{v}(\hat{\mathbf{x}} - \epsilon\,\mathbf{n})\right].$$

The PDE defined in (2.2) is also called *state problem* or *state equation*, and its resulting solution, $\hat{u}$, is referred to as *state variable*. A sketch of the domain Ω together with its discontinuity surface Σ is drawn in Fig. 2.1.

2.2.1 Weak formulation

Since the solution $\hat{u}$ of (2.2) shows a discontinuity across the surface Σ, the canonical Hilbert space $H^1(\Omega)$ [15, Par. 6.3] is not suitable. In this respect, we introduce an enrichment of $H^1(\Omega)$, defined as follows.

Definition 2.2. *Vector space for state variable*

$$H^1_\Sigma(\Omega) \coloneqq \left\{\hat{v} \in H^1(\Omega \setminus \Sigma),\ [\hat{v}]_\Sigma = const\right\}.$$

This allows to consider also functions that present a fixed jump across Σ. The weak formulation [15, Chap. 8] of (2.2) becomes:
Find $\hat{u} \in H^1_\Sigma(\Omega) : \forall \hat{v} \in H^1_\Sigma(\Omega)$,

$$\int_{\Omega\setminus\Sigma} \nabla\hat{u} \cdot \nabla\hat{v}\, d\hat{\mathbf{x}} = -I\,[\hat{v}]_\Sigma. \tag{2.3}$$

Proof. Since the behaviour of the derivative of $\hat{u}$ is not defined on Σ, we need to split the domain Ω in two sub-domains Ω^+ and Ω^- divided by the discontinuity surface Σ, such that $\Omega \setminus \Sigma = \Omega^+ \cup \Omega^-$ and $\Omega^+ \cap \Omega^- = \emptyset$. We can then multiply the strong form (2.2) by a generic function $\hat{v} \in H^1_\Sigma(\Omega)$ and integrate over the two sub-domains. For ease of notation, the sum of these two volume integrals will be written as

$$-\int_{\Omega^+} \mathrm{div}(\nabla\hat{u})\,\hat{v}\, d\hat{\mathbf{x}} - \int_{\Omega^-} \mathrm{div}(\nabla\hat{u})\,\hat{v}\, d\hat{\mathbf{x}} = -\int_{\Omega\setminus\Sigma} \mathrm{div}(\nabla\hat{u})\,\hat{v}\, d\hat{\mathbf{x}} = 0,$$

where the first equation in (2.2) has been considered. Integration by parts leads to

$$\int_{\Omega\setminus\Sigma} \nabla\,\hat{u}\cdot\nabla\,\hat{v}\;d\hat{\mathbf{x}} - \int_{\partial\Omega} \nabla\,\hat{u}\cdot\mathbf{n}_{\partial\Omega}\;\hat{v}\;d\hat{\mathbf{x}} - \int_{\Sigma} \nabla\,\hat{u}^{+}\cdot\mathbf{n}^{+}\;\hat{v}^{+}\;d\hat{\mathbf{x}} - \int_{\Sigma} \nabla\,\hat{u}^{-}\cdot\mathbf{n}^{-}\;\hat{v}^{-}\;d\hat{\mathbf{x}} = 0,$$

where $\mathbf{n}_{\partial\Omega}$ is the outer normal of the boundary $\partial\Omega$, while $\mathbf{n}^{+}$ and $\mathbf{n}^{-}$ denote the inner/outer normals to the surface Σ. Of course, we have $\mathbf{n}^{+} = -\mathbf{n}^{-}$. Since homogeneous Neumann boundary conditions are imposed on $\partial\Omega$ (2.2), we can get rid of the second term and obtain

$$\int_{\Omega\setminus\Sigma} \nabla\,\hat{u}\cdot\nabla\,\hat{v}\;d\hat{\mathbf{x}} - \int_{\Sigma} \nabla\,\hat{u}^{-}\cdot\mathbf{n}^{+}\;\hat{v}^{-}\;d\hat{\mathbf{x}} + \int_{\Sigma} \nabla\,\hat{u}^{+}\cdot\mathbf{n}^{+}\;\hat{v}^{-}\;d\hat{\mathbf{x}} = 0.$$

The additional boundary conditions allow us to further simplify the equation:

$$\begin{aligned}
&\int_{\Omega\setminus\Sigma} \nabla\,\hat{u}\cdot\nabla\,\hat{v}\;d\hat{\mathbf{x}} - \int_{\Sigma} \nabla\,\hat{u}^{-}\cdot\mathbf{n}^{+}\;\hat{v}^{-}\;d\hat{\mathbf{x}} + \int_{\Sigma} \nabla\,\hat{u}^{+}\cdot\mathbf{n}^{+}\;\hat{v}^{+}\;d\hat{\mathbf{x}} = 0 \\
&\xrightarrow{[\nabla\,\hat{u}\cdot\mathbf{n}]_{\Sigma}=0} \int_{\Omega\setminus\Sigma} \nabla\,\hat{u}\cdot\nabla\,\hat{v}\;d\hat{\mathbf{x}} + \int_{\Sigma} \nabla\,\hat{u}\cdot\mathbf{n}^{+}\;[\hat{v}]_{\Sigma}\;d\hat{\mathbf{x}} = 0 \\
&\xrightarrow{[\hat{v}]_{\Sigma}=const} \int_{\Omega\setminus\Sigma} \nabla\,\hat{u}\cdot\nabla\,\hat{v}\;d\hat{\mathbf{x}} + [\hat{v}]_{\Sigma}\int_{\Sigma} \nabla\,\hat{u}\cdot\mathbf{n}^{+}\;d\hat{\mathbf{x}} = 0 \\
&\xrightarrow{\int_{\Sigma}\nabla\,\hat{u}\cdot\mathbf{n}^{+}d\hat{\mathbf{x}}=I} \int_{\Omega\setminus\Sigma} \nabla\,\hat{u}\cdot\nabla\,\hat{v}\;d\mathbf{x} + I[\hat{v}]_{\Sigma} = 0.
\end{aligned}$$

□

2.2.2 Additional constraint

The problem described in (2.2) is not well-defined: it can easily be noticed that, if $\hat{u}$ is a solution, then also $\hat{u}+c$, $\forall c\in\mathbb{R}$, satisfies the equations. In order to ensure uniqueness of solution, we consider the additional constraint

$$\int_{\Omega} \hat{u}\;d\hat{\mathbf{x}} = 0. \tag{2.4}$$

This constraint is taken into account by means of a *Lagrange multiplier method* [12]. In the frame of this method, the state equation we need to solve to find $\hat{u}$ is re-cast as a saddle-point problem [13], where an additional variable λ (also called *Lagrange multiplier*) is introduced as a coefficient of the constraint itself. To this purpose, we

introduce the *Lagrangian function* associated to (2.3) and the constraint (2.4),

$$\mathcal{L}_{st}(\hat{u}, \lambda) = \frac{1}{2} \int_{\Omega \setminus \Sigma} \nabla \hat{u} \cdot \nabla \hat{u} \, d\hat{\mathbf{x}} + I\,[\hat{u}] + \lambda \int_{\Omega} \hat{u} \, d\hat{\mathbf{x}},$$

and we look for the point $(\hat{u}, \lambda) \in H^1_{\Sigma}(\Omega) \times \mathbb{R}$ where the derivatives of $\mathcal{L}_{st}$ with respect to $\hat{u}$ and λ are 0 for each direction $(\hat{v}, \mu) \in H^1_{\Sigma}(\Omega) \times \mathbb{R}$. This produces the following problem:
Find $(\hat{u}, \lambda) \in H^1_{\Sigma}(\Omega) \times \mathbb{R} : \forall (\hat{v}, \mu) \in H^1_{\Sigma}(\Omega) \times \mathbb{R}$,

$$\begin{cases} \int_{\Omega \setminus \Sigma} \nabla \hat{u} \cdot \nabla \hat{v} \, d\hat{\mathbf{x}} + \lambda \int_{\Omega} \hat{v} \, d\hat{\mathbf{x}} = -I\,[\hat{v}]_{\Sigma} \\ \mu \int_{\Omega} \hat{u} \, d\hat{\mathbf{x}} = 0 \end{cases}, \tag{2.5}$$

which is equivalent to solving (2.3) under the constraint (2.4).

More details about the implementation of the Lagrange multiplier method are given in Chap. 5.

2.3 Admissible shapes

We define the set of admissible shapes the conductor can take as

Definition 2.3. *Set $\mathcal{V}_{ad}$ of admissible shapes*

$$\mathcal{V}_{ad} := \{\Omega = T_{\mathcal{V}}(\Omega_0) \mid T_{\mathcal{V}} := \mathcal{I} + \mathcal{V},\ \|\mathcal{V}\|_{C^1(\overline{D};\mathbb{R}^3)} \leq 1 - \epsilon\},$$

with $\epsilon \in \mathbb{R}$ fixed and small.

Here, Ω_0 is the initial domain[2] which is assumed to have Lipschitz boundary [15, Par. 1.6], $\overline{D}$ is a bounded convex domain that includes Ω_0, $\mathcal{I}$ represents the identity operator, $\mathcal{I}(\mathbf{x}) = \mathbf{x}$, and $T_{\mathcal{V}}$ is a map that depends on the vector field $\mathcal{V}$. This map is defined in the following way:

Definition 2.4. *Map $T_{\mathcal{V}}$*

$$\begin{aligned} T_{\mathcal{V}} : \mathbb{R}^3 &\to \mathbb{R}^3, \\ \mathbf{x} &\mapsto T_{\mathcal{V}}(\mathbf{x}) := \mathbf{x} + \mathcal{V}(\mathbf{x}). \end{aligned}$$

By $T_{\mathcal{V}}(\Omega_0)$ we denote the image set of the map $T_{\mathcal{V}}$ restricted to Ω_0, as depicted in Fig. 2.2.

[2] In our work, the reference domain Ω_0 is a torus of outer radius 1 and inner radius 0.3, similar to the one shown in Fig. 2.1.

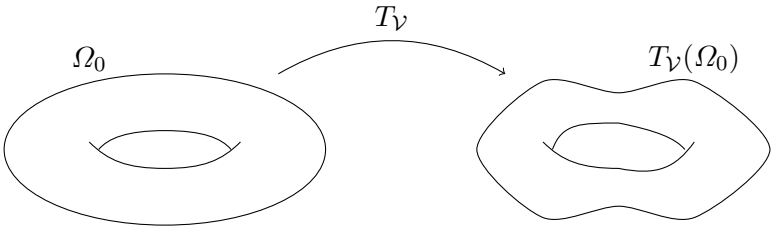

FIGURE 2.2: Sketch of the action of map $T_{\mathcal{V}}$ on the initial unmodified domain Ω_0, and of the resulting mapped domain $\Omega = T_{\mathcal{V}}(\Omega_0)$.

Furthermore, we assume that the vector field

$$\mathcal{V} : \overline{D} \to \mathbb{R}^3$$

satisfies the condition $\|\mathcal{V}\|_{C^1(\overline{D};\mathbb{R}^3)} < 1$, so that $T_{\mathcal{V}}$ is a *diffeomorphism* [1, Lemma 6.13]. As a consequence of this requirement, we have that

$$\det\left(\mathbf{D}T_{\mathcal{V}}\right)(\mathbf{x}) > 0 \quad \forall \mathbf{x} \in \mathbb{R}^3,$$

where $\mathbf{D}T_{\mathcal{V}}$ denotes the *Jacobian* of $T_{\mathcal{V}}$,

$$(\mathbf{D}T_{\mathcal{V}})_{ij} = \frac{\partial (T_{\mathcal{V}})_i}{\partial x_j} = \delta_{i,j} + (\mathbf{D}\mathcal{V})_{ij},$$

with δ_{ij} being the *Kronecker delta*[3].

2.4 Shape optimization in parametric form

With the change of coordinates induced by the map in Def. 2.4, it is possible to reformulate the problem on an initial reference domain Ω_0.

2.4.1 Mapped state equation

In particular, the equation (2.5) used to recover the state becomes:
Find $(u, \lambda) \in H^1_{\Sigma_0}(\Omega_0) \times \mathbb{R} : \forall (v, \mu) \in H^1_{\Sigma_0}(\Omega_0) \times \mathbb{R}$,

$$\begin{cases} \int_{\Omega_0 \setminus \Sigma_0} \nabla u \, \mathbf{M}_{\mathcal{V}} \nabla v \, d\mathbf{x} + \lambda \int_{\Omega_0} \left|\det\left(\mathbf{D}T_{\mathcal{V}}\right)\right| v \, d\mathbf{x} = -I\left[v\right]_{\Sigma_0} \\ \mu \int_{\Omega_0} \left|\det\left(\mathbf{D}T_{\mathcal{V}}\right)\right| u \, d\mathbf{x} = 0 \end{cases}, \tag{2.6}$$

[3] $\delta_{i,j} = 1$ if $i = j$; $\delta_{i,j} = 0$ if $i \neq j$.

where Σ_0 is the reference discontinuity surface defined on Ω_0, while $u(\mathbf{x}) = \hat{u}(T_\mathcal{V}(\mathbf{x})) = \hat{u}(\hat{\mathbf{x}})$, with $\mathbf{x} \in \Omega_0$ and consequently $\hat{\mathbf{x}} \in \Omega$. The definition for $\mathbf{M}_\mathcal{V}$, also called *pull-back matrix*, follows.

Definition 2.5. *Pull-back matrix*

$$\mathbf{M}_\mathcal{V} \coloneqq \mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}T_\mathcal{V}^{-T} \left|\det\left(\mathbf{D}T_\mathcal{V}\right)\right|.^{4} \tag{2.7}$$

Proof. This can be easily seen by noting that, given a regular enough map $\mathbf{x} = F(\tilde{\mathbf{x}})$, defined on a generic domain $\tilde{\Omega}$ and such that $F(\tilde{\Omega}) = \Omega$, it holds

$$\int_\Omega \nabla u(\mathbf{x}) \nabla v(\mathbf{x})\, d\mathbf{x} = \int_{\tilde{\Omega}} (\mathbf{D}F^{-T}\tilde{\nabla}\tilde{u}(\tilde{\mathbf{x}}))(\mathbf{D}F^{-T}\tilde{\nabla}\tilde{v}(\tilde{\mathbf{x}})) \left|\det\left(\mathbf{D}F\right)\right| \, d\tilde{\mathbf{x}},$$

where $\tilde{\nabla}$ denotes the gradient in $\tilde{\mathbf{x}}$ coordinates, and $\tilde{u}(\tilde{\mathbf{x}}) = u(F(\tilde{x})) = u(\mathbf{x})$. In fact,

$$\frac{\partial u(\mathbf{x})}{\partial x_i} = \frac{\partial u(F(\mathbf{x}))}{\partial \tilde{x_1}}\frac{\partial \tilde{x_1}}{\partial x_i} + \frac{\partial u(F(\mathbf{x}))}{\partial \tilde{x_2}}\frac{\partial \tilde{x_2}}{\partial x_i} + \frac{\partial u(F(\mathbf{x}))}{\partial \tilde{x_3}}\frac{\partial \tilde{x_3}}{\partial x_i},$$

hence, using $\mathbf{D}F(\mathbf{x}) = \mathbf{D}F(\tilde{\mathbf{x}})^{-1}$,

$$\nabla u(\mathbf{x}) = \mathbf{D}F(\mathbf{x})^T\tilde{\nabla}\tilde{u}(\tilde{\mathbf{x}}) = \mathbf{D}F(\tilde{\mathbf{x}})^{-T}\tilde{\nabla}\tilde{u}(\tilde{\mathbf{x}}).$$

Finally, the scaling factor $\left|\det\left(\mathbf{D}F\right)\right|$ comes from the change of coordinates inside the integral: $d\mathbf{x} = \left|\det\left(\mathbf{D}F\right)\right| d\tilde{\mathbf{x}}$. In our case, we just have to substitute $\tilde{\Omega} = \Omega_0 \setminus \Sigma_0$ and $F = T_\mathcal{V}$ to complete the proof. □

2.4.2 Mapped cost functional

Similarly, the cost functional in Def. 2.1 can be re-cast in a form that depends on the mapping $T_\mathcal{V}$ or, more precisely, on $\mathcal{V}$:

$$\begin{aligned}\mathcal{J} \quad & (\mathcal{V}, u(\mathcal{V})) = \\ & \int_{\partial\mathcal{D}} \left\| \tfrac{\mu_0}{4\pi} \left(\int_{\Omega_0\setminus\Sigma_0} \left|\det\left(\mathbf{D}T_\mathcal{V}\right)\right| \mathbf{D}T_\mathcal{V}^{-T} \nabla u \times \nabla_\mathbf{x} \left(\tfrac{1}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|} \right) d\mathbf{y} \right)_t - \mathbf{p} \right\|^2 d\mathbf{x}.\end{aligned} \tag{2.8}$$

In this way, we have dropped the dependency of $\mathcal{J}$ on the shape itself Ω, substituting it with a parametrization described by $\mathcal{V}$.

[4] We also point out that $\mathbf{M}_\mathcal{V}$ is symmetric: we have in fact $\mathbf{M}_\mathcal{V}^T = (\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}T_\mathcal{V}^{-T})^T \left|\det\left(\mathbf{D}T_\mathcal{V}\right)\right| = (\mathbf{D}T_\mathcal{V}^{-T})^T(\mathbf{D}T_\mathcal{V}^{-1})^T \left|\det\left(\mathbf{D}T_\mathcal{V}\right)\right| = \mathbf{M}_\mathcal{V}$. The consequences of this are described in Par. 5.1.

The shape optimization problem is then defined in the following parametric form,

$$\min_{\|\mathcal{V}\|_{C^1(\overline{D};\mathbb{R}^3)} \leq 1-\epsilon} \mathcal{J}(\mathcal{V}, u(\mathcal{V})) \; s.t. \; (2.6), \tag{2.9}$$

with $\mathcal{V}$ becoming our so-called *control variable*.

Chapter 3

First order Fréchet derivative

The concept of Fréchet derivative can be used to extend the notion of derivative to a functional defined on a Banach space [15, Par. 6.2]. In this work, it is applied to the cost functional (2.8) and is employed in the optimization algorithm.

3.1 Fréchet differentiability

Here, the definitions of *Fréchet differentiability* and *Fréchet derivative* are given; for a more detailed discussion, see [3, Par. 1.4.1].

Definition 3.1. *Fréchet differentiability* Let $F : U \to \mathbb{R}$ be a functional defined on an open set $U \neq \emptyset$ in a Banach space V. F is said to be *Fréchet differentiable* at $v \in U$ if:

1. The limit
$$dF(v,h) = \lim_{\epsilon \to 0} \frac{F(v+\epsilon h) - F(v)}{\epsilon} \in \mathbb{R}$$
exists $\forall h \in V$.

2. The functional
$$F'(v) : h \ni V \mapsto dF(v,h) \in \mathbb{R}$$
is bounded and linear (*Gâteaux differentiability*).

3. It holds that
$$\left|F(v+h) - F(v) - \left\langle F'(v), h \right\rangle\right| = o(\|h\|_V),$$
as $\|h\|_V \to 0$. $\langle F'(v), h\rangle$ is the result of $F'(v)$ evaluated at h.

The functional $F'(v)$ is called *Fréchet derivative* of F at v.

For ease of reading, from here on we refer to the Fréchet derivative simply with the term derivative.

3.2 Relevant derivatives

In order to recover a formula for the first derivative of the cost functional (2.8), we need to provide relations to express the derivatives of some quantities of interest with respect to the variable $\mathcal{V}$, along the direction $\tilde{\mathcal{V}}$. This is denoted as

$$\delta_{\tilde{\mathcal{V}}}(*) = \left\langle (*)', \tilde{\mathcal{V}} \right\rangle.$$

Lemma 3.2. *Fréchet derivative of* $\det(\mathbf{D}T_{\mathcal{V}})$

$$\delta_{\tilde{\mathcal{V}}}(\det(\mathbf{D}T_{\mathcal{V}})) = \det(\mathbf{D}T_{\mathcal{V}})\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}}). \tag{3.1}$$

Proof. According to Def. 3.1, we have:

$$\delta_{\tilde{\mathcal{V}}}(\det(\mathbf{D}T_{\mathcal{V}})) = \lim_{\epsilon\to 0} \frac{\det(\mathbf{D}T_{\mathcal{V}} + \epsilon\,\mathbf{D}\tilde{\mathcal{V}}) - \det(\mathbf{D}T_{\mathcal{V}})}{\epsilon},$$

where Def. 2.4 has been used. If we consider the property of the determinant, $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$, the first term of the numerator can be rewritten as follows (here, I represents the identity matrix):

$$\begin{aligned}\det(\mathbf{D}T_{\mathcal{V}} + \epsilon\,\mathbf{D}\tilde{\mathcal{V}}) &= \det((\mathbf{D}T_{\mathcal{V}})(I + \epsilon\,\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}})) \\ &= \det(\mathbf{D}T_{\mathcal{V}})\det(I + \epsilon\,\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}}).\end{aligned}$$

We can perform a first-order Taylor expansion on the second factor above to retrieve

$$\det(\mathbf{D}T_{\mathcal{V}} + \epsilon\,\mathbf{D}\tilde{\mathcal{V}}) = \det(\mathbf{D}T_{\mathcal{V}})(1 + \epsilon\,\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}})) + \mathcal{O}(\epsilon^2).^{1}$$

If we substitute this in the limit, we obtain

$$\delta_{\tilde{\mathcal{V}}}\ (\det(\mathbf{D}T_{\mathcal{V}})) = \lim_{\epsilon\to 0} \frac{\cancel{\det(\mathbf{D}T_{\mathcal{V}})} + \epsilon\det(\mathbf{D}T_{\mathcal{V}})\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}}) - \cancel{\det(\mathbf{D}T_{\mathcal{V}})} + \mathcal{O}(\epsilon^2)}{\epsilon},$$

which also shows how the candidate in (3.1) for the Fréchet derivative provides linear approximation (Pt. 3 in 3.1). Given the requirements on $\tilde{\mathcal{V}}$ in Def. 2.3, the limit above

[1] This result comes from the application of *Jacobi's formula* [14] for the derivative of the determinant of a matrix:

$$(\det\mathbf{A}(t))' = \mathrm{tr}(\mathrm{adj}(\mathbf{A}(t))\ \mathbf{A}'(t)),$$

with adj(*) being the adjugate matrix of $*$.

exists (Pt. 1 in 3.1). To complete the proof, we need to check that the operator in (3.1) is linear and bounded (Pt. 2 in 3.1). Linearity is a direct consequence of the properties of the trace and derivative operators

$$\begin{aligned}\delta_{(\alpha\tilde{\mathcal{V}}+\beta\tilde{\mathcal{W}})}(\det(\mathbf{D}T_{\mathcal{V}})) &= \det(\mathbf{D}T_{\mathcal{V}})\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}(\alpha\tilde{\mathcal{V}}+\beta\tilde{\mathcal{W}}))\\ &= \det(\mathbf{D}T_{\mathcal{V}})[\alpha\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}})+\beta\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{W}})]\\ &= \alpha\delta_{\tilde{\mathcal{V}}}(\det(\mathbf{D}T_{\mathcal{V}}))+\beta\delta_{\tilde{\mathcal{W}}}(\det(\mathbf{D}T_{\mathcal{V}})).\end{aligned}$$

Since the operator is linear, continuity ensures boundedness [2]. In Def. 2.3 we require $\tilde{\mathcal{V}}$ to be differentiable, which means that $\mathbf{D}\tilde{\mathcal{V}}$ is continuous (and so is the trace operator). This completes the proof. □

Lemma 3.3. *Fréchet derivative of* $\mathbf{D}T_{\mathcal{V}}^{-1}$ *and* $\mathbf{D}T_{\mathcal{V}}^{-T}$

$$\delta_{\tilde{\mathcal{V}}}(\mathbf{D}T_{\mathcal{V}}^{-1}) = -\mathbf{D}T_{\mathcal{V}}^{-1}(\mathbf{D}\tilde{\mathcal{V}})\mathbf{D}T_{\mathcal{V}}^{-1}. \tag{3.2}$$

$$\delta_{\tilde{\mathcal{V}}}(\mathbf{D}T_{\mathcal{V}}^{-T}) = -\mathbf{D}T_{\mathcal{V}}^{-T}(\mathbf{D}\tilde{\mathcal{V}}^{T})\mathbf{D}T_{\mathcal{V}}^{-T}. \tag{3.3}$$

Proof. We first prove (3.2). Again, we employ Def. 3.1 to get

$$\delta_{\tilde{\mathcal{V}}}(\mathbf{D}T_{\mathcal{V}}^{-1}) = \lim_{\epsilon\to 0}\frac{(\mathbf{D}T_{\mathcal{V}}+\epsilon\,\mathbf{D}\tilde{\mathcal{V}})^{-1}-\mathbf{D}T_{\mathcal{V}}^{-1}}{\epsilon}.$$

By re-writing the first term in the numerator as

$$\begin{aligned}(\mathbf{D}T_{\mathcal{V}}+\epsilon\,\mathbf{D}\tilde{\mathcal{V}})^{-1} &= ((\mathbf{D}T_{\mathcal{V}})(I+\epsilon\,\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}}))^{-1}\\ &= (I+\epsilon\,\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}})^{-1}(\mathbf{D}T_{\mathcal{V}})^{-1},\end{aligned}$$

and expanding the *Neumann series*[2] of the first factor, we obtain

$$(\mathbf{D}T_{\mathcal{V}}+\epsilon\,\mathbf{D}\tilde{\mathcal{V}})^{-1} = (I-\epsilon\,\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}})(\mathbf{D}T_{\mathcal{V}})^{-1}+\mathcal{O}(\epsilon^2).$$

Then, substituting this in the definition above,

$$\delta_{\tilde{\mathcal{V}}}(\mathbf{D}T_{\mathcal{V}}^{-1}) = \lim_{\epsilon\to 0}\frac{\cancel{\mathbf{D}T_{\mathcal{V}}^{-1}}-\epsilon\,\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}}\mathbf{D}T_{\mathcal{V}}^{-1}-\cancel{\mathbf{D}T_{\mathcal{V}}^{-1}}+\mathcal{O}(\epsilon^2)}{\epsilon},$$

and the proof is completed. Following an analogous procedure, (3.3) can be recovered as well. □

[2] Under the assumption that the spectral radius of $\mathbf{A}$ is < 1, we have that

$$(I-\mathbf{A})^{-1} = \sum_{n=0}^{\infty}\mathbf{A}^n.$$

Lemma 3.4. *Fréchet derivative of* $\mathbf{M}_\mathcal{V}$

$$\begin{aligned}\delta_{\tilde{\mathcal{V}}}\mathbf{M}_\mathcal{V} &= \det(\mathbf{D}T_\mathcal{V})\mathbf{D}T_\mathcal{V}^{-1}\Big[\mathrm{tr}(\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}\tilde{\mathcal{V}})I \\ &\quad - \mathbf{D}\tilde{\mathcal{V}}\mathbf{D}T_\mathcal{V}^{-1} - \mathbf{D}T_\mathcal{V}^{-T}\mathbf{D}\tilde{\mathcal{V}}^T\Big]\mathbf{D}T_\mathcal{V}^{-T}.\end{aligned} \tag{3.4}$$

Proof. Starting from Def. 2.5 and differentiating it brings to

$$\begin{aligned}\delta_{\tilde{\mathcal{V}}}\mathbf{M}_\mathcal{V} &= \delta_{\tilde{\mathcal{V}}}(\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}T_\mathcal{V}^{-T}\det(\mathbf{D}T_\mathcal{V})) \\ &= \delta_{\tilde{\mathcal{V}}}(\mathbf{D}T_\mathcal{V}^{-1})\mathbf{D}T_\mathcal{V}^{-T}\det(\mathbf{D}T_\mathcal{V}) \\ &\quad +\mathbf{D}T_\mathcal{V}^{-1}\delta_{\tilde{\mathcal{V}}}(\mathbf{D}T_\mathcal{V}^{-T})\det(\mathbf{D}T_\mathcal{V}) \\ &\quad +\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}T_\mathcal{V}^{-T}\delta_{\tilde{\mathcal{V}}}(\det(\mathbf{D}T_\mathcal{V})).\end{aligned}$$

This is achieved by applying the product rule for differentiation, which also holds for Fréchet derivatives. It is possible now to use the relations (3.1), (3.2), (3.3) to rewrite $\delta_{\tilde{\mathcal{V}}}\mathbf{M}_\mathcal{V}$ as

$$\begin{aligned}\delta_{\tilde{\mathcal{V}}}\mathbf{M}_\mathcal{V} &= -\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}\tilde{\mathcal{V}}\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}T_\mathcal{V}^{-T}\det(\mathbf{D}T_\mathcal{V}) \\ &\quad -\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}T_\mathcal{V}^{-T}\mathbf{D}\tilde{\mathcal{V}}^T\mathbf{D}T_\mathcal{V}^{-T}\det(\mathbf{D}T_\mathcal{V}) \\ &\quad +\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}T_\mathcal{V}^{-T}\mathrm{tr}(\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}\tilde{\mathcal{V}})\det(\mathbf{D}T_\mathcal{V}),\end{aligned}$$

which, after some rearrangements, gives (3.4). □

Lemma 3.5. *Fréchet derivative of* $\nabla_\mathbf{x}(\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^{-1})$

$$\delta_{\tilde{\mathcal{V}}}\left(\nabla_\mathbf{x}\left(\frac{1}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|}\right)\right) = \frac{\tilde{\mathcal{V}}}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^3} - 3\frac{\Big[(\mathbf{x}-T_\mathcal{V}(\mathbf{y}))\cdot\tilde{\mathcal{V}}\Big](\mathbf{x}-T_\mathcal{V}(\mathbf{y}))}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^5}. \tag{3.5}$$

Proof. Remembering that $\|\mathbf{x}\|' = \mathbf{x}/\|\mathbf{x}\|$, we can first retrieve

$$\nabla_\mathbf{x}\left(\frac{1}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|}\right) = -\frac{\mathbf{x}-T_\mathcal{V}(\mathbf{y})}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^3}.$$

Using then the chain rule,

$$\begin{aligned}\delta_{\tilde{\mathcal{V}}}&\left(-\frac{\mathbf{x}-T_\mathcal{V}(\mathbf{y})}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^3}\right) = \\ &-\frac{\delta_{\tilde{\mathcal{V}}}(\mathbf{x}-T_\mathcal{V}(\mathbf{y}))\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^3 - \delta_{\tilde{\mathcal{V}}}(\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^3)(\mathbf{x}-T_\mathcal{V}(\mathbf{y}))}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^6}.\end{aligned}$$

According to Def. 2.4, it is easy to notice that $\delta_{\tilde{\mathcal{V}}}T_\mathcal{V}(\mathbf{y}) = \tilde{\mathcal{V}}(\mathbf{y})$. This gives us

$$\begin{aligned}\delta_{\tilde{\mathcal{V}}}&\left(-\frac{\mathbf{x}-T_\mathcal{V}(\mathbf{y})}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^3}\right) = \\ &\frac{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^3\tilde{\mathcal{V}}(\mathbf{y}) - 3\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|\Big[(\mathbf{x}-T_\mathcal{V}(\mathbf{y}))\cdot\tilde{\mathcal{V}}\Big](\mathbf{x}-T_\mathcal{V}(\mathbf{y}))}{\|\mathbf{x}-T_\mathcal{V}(\mathbf{y})\|^6},\end{aligned}$$

which is the formula we wanted to prove. □

3.3 Sensitivity-based formula for the Fréchet derivative of cost functional

Thanks to the relations proven above, we can provide an expression for the derivative of the cost functional (2.8) with respect to the control $\mathcal{V}$. With a mild abuse of terminology, we also refer to it with the term *shape derivative*.

Proposition 3.6. The first order Fréchet derivative of the cost functional (2.8), evaluated along the direction $\tilde{\mathcal{V}}$, is given by the formula

$$\begin{aligned}\left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle &= 2\int_{\partial\mathcal{D}} (\mathbf{H_t}(u) - \mathbf{p}) \cdot \Big[\mathbf{H_t}(\delta_{\tilde{\mathcal{V}}} u) \\ &\quad + \left(\int_{\Omega_0 \setminus \Sigma_0} \delta_{\tilde{\mathcal{V}}}\mathbf{A} \times \mathbf{B}\, d\mathbf{y}\right)_t \\ &\quad + \left(\int_{\Omega_0 \setminus \Sigma_0} \mathbf{A} \times \delta_{\tilde{\mathcal{V}}}\mathbf{B}\, d\mathbf{y}\right)_t\Big]\, d\mathbf{x},\end{aligned} \tag{3.6}$$

where:

$$\begin{aligned}\mathbf{A} &= \tfrac{\mu_0}{4\pi} \det(\mathbf{D}T_{\mathcal{V}}) \mathbf{D}T_{\mathcal{V}}^{-T} \nabla u, \\ \mathbf{B} &= -\tfrac{\mathbf{x} - T_{\mathcal{V}}(\mathbf{y})}{\|\mathbf{x} - T_{\mathcal{V}}(\mathbf{y})\|^3}, \\ \delta_{\tilde{\mathcal{V}}}\mathbf{A} &= \tfrac{\mu_0}{4\pi} \det(\mathbf{D}T_{\mathcal{V}}) \mathbf{D}T_{\mathcal{V}}^{-T} \left[\operatorname{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\tilde{\mathcal{V}}) I - \mathbf{D}\tilde{\mathcal{V}}^T \mathbf{D}T_{\mathcal{V}}^{-T}\right] \nabla u, \\ \delta_{\tilde{\mathcal{V}}}\mathbf{B} &= \tfrac{\tilde{\mathcal{V}}}{\|\mathbf{x} - T_{\mathcal{V}}(\mathbf{y})\|^3} - 3\tfrac{\left[(\mathbf{x} - T_{\mathcal{V}}(\mathbf{y})) \cdot \tilde{\mathcal{V}}\right](\mathbf{x} - T_{\mathcal{V}}(\mathbf{y}))}{\|\mathbf{x} - T_{\mathcal{V}}(\mathbf{y})\|^5}.\end{aligned} \tag{3.7}$$

The term

$$\delta_{\tilde{\mathcal{V}}} u = \left\langle u'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle \tag{3.8}$$

is also referred to as *sensitivity* of the state u with respect to the control $\tilde{\mathcal{V}}$.

Proof. To prove (3.6), we can follow the analogous procedure presented in [3, Par. 1.6.1]. Through partial differentiation, we easily get

$$\left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle = \left\langle \mathcal{J}_u, \delta_{\tilde{\mathcal{V}}} u \right\rangle + \left\langle \mathcal{J}_{\mathcal{V}}, \tilde{\mathcal{V}} \right\rangle, \tag{3.9}$$

where $(\cdot)_*$ denotes the first partial derivative of $(\cdot)$ with respect to the variable $*$. Focusing on the first term of (3.9) we have, using the chain rule of differentiation,

$$\left\langle \mathcal{J}_u, \delta_{\tilde{\mathcal{V}}} u \right\rangle = 2\int_{\partial\mathcal{D}} (\mathbf{H_t}(u) - \mathbf{p}) \cdot (\mathbf{H_t}(u) - \mathbf{p})_u (\delta_{\tilde{\mathcal{V}}} u)\, d\mathbf{x},$$

but since the operators involved in the evaluation of $\mathbf{H_t}(u)$ are linear in u, its derivative along $\delta_{\tilde{\mathcal{V}}} u$ is $\mathbf{H_t}(\delta_{\tilde{\mathcal{V}}} u)$ itself. On the other hand, $\mathbf{p}_u$ cancels out. This gives us the first

term in (3.6). In general, we can recover a formula for the partial derivative $\mathcal{J}_u$ along v:

$$\langle \mathcal{J}_u, v \rangle = 2 \int_{\partial \mathcal{D}} (\mathbf{H_t}(u) - \mathbf{p}) \cdot \mathbf{H_t}(v) \, d\mathbf{x}, \tag{3.10}$$

which will be useful in Sec. 3.4.2. For the second term in (3.9), again, we start with the chain rule,

$$\left\langle \mathcal{J}_{\mathcal{V}}, \tilde{\mathcal{V}} \right\rangle = 2 \int_{\partial \mathcal{D}} (\mathbf{H_t}(u) - \mathbf{p}) \cdot \left(\delta_{\tilde{\mathcal{V}}} \mathbf{H_t}(u) - \cancel{\delta_{\tilde{\mathcal{V}}} \mathbf{p}} \right) \, d\mathbf{x}. \tag{3.11}$$

Remembering the definitions of the cost functional (2.8) and of the terms $\mathbf{A}$ and $\mathbf{B}$ in (3.7), we can substitute

$$\begin{aligned} \delta_{\tilde{\mathcal{V}}} \mathbf{H_t}(u) &= \delta_{\tilde{\mathcal{V}}} \left(\textstyle\int_{\Omega_0 \setminus \Sigma_0} \mathbf{A} \times \mathbf{B} \, d\mathbf{y} \right)_t \\ &= \left(\textstyle\int_{\Omega_0 \setminus \Sigma_0} \delta_{\tilde{\mathcal{V}}} \mathbf{A} \times \mathbf{B} \, d\mathbf{y} \right)_t + \left(\textstyle\int_{\Omega_0 \setminus \Sigma_0} \mathbf{A} \times \delta_{\tilde{\mathcal{V}}} \mathbf{B} \, d\mathbf{y} \right)_t . \end{aligned}$$

These are the last two terms of (3.6). It can be easily seen that the expression provided for the derivative of $\mathbf{B}$ is exactly the one proved in (3.5); for the derivative of $\mathbf{A}$, instead, formulas (3.1) and (3.2) have been used:

$$\begin{aligned} \delta_{\tilde{\mathcal{V}}} \mathbf{A} &= \tfrac{\mu_0}{4\pi} \left(\delta_{\tilde{\mathcal{V}}} (\det(\mathbf{D}T_{\mathcal{V}})) \mathbf{D}T_{\mathcal{V}}^{-T} \nabla u + \det(\mathbf{D}T_{\mathcal{V}}) \delta_{\tilde{\mathcal{V}}} (\mathbf{D}T_{\mathcal{V}}^{-T}) \nabla u \right) \\ &= \tfrac{\mu_0}{4\pi} \det(\mathbf{D}T_{\mathcal{V}}) \mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}\tilde{\mathcal{V}}) \mathbf{D}T_{\mathcal{V}}^{-T} \nabla u \\ &- \tfrac{\mu_0}{4\pi} \det(\mathbf{D}T_{\mathcal{V}}) \mathbf{D}T_{\mathcal{V}}^{-T} \mathbf{D}\tilde{\mathcal{V}}^T \mathbf{D}T_{\mathcal{V}}^{-T} \nabla u. \end{aligned}$$

□

3.4 Adjoint approach

We are interested in describing the action of the whole operator $\mathcal{J}'(\mathcal{V})$. If the formula (3.6) was used for this purpose, this would imply having to evaluate the sensitivity (3.8) for every possible direction $\tilde{\mathcal{V}}$. This can be done by linearizing the constraint imposed by the state equation. Differentiation of (2.6) with respect to $\mathcal{V}$, in fact, gives:
$\forall (v, \mu) \in H^1_{\Sigma_0}(\Omega_0) \times \mathbb{R}^3$,

$$\begin{cases} \int_{\Omega_0 \setminus \Sigma_0} \nabla \, \delta_{\tilde{\mathcal{V}}} u \, \mathbf{M}_{\mathcal{V}} \nabla v \, d\mathbf{x} & + \int_{\Omega_0 \setminus \Sigma_0} \nabla u \, \delta_{\tilde{\mathcal{V}}} \mathbf{M}_{\mathcal{V}} \nabla v \, d\mathbf{x} \\ & + \tilde{\lambda} \int_{\Omega_0} \delta_{\tilde{\mathcal{V}}} (|\det(\mathbf{D}T_{\mathcal{V}})|) v \, d\mathbf{x} & = 0 \\ \mu \int_{\Omega_0} \delta_{\tilde{\mathcal{V}}} (|\det(\mathbf{D}T_{\mathcal{V}})|) u \, d\mathbf{x} & + \mu \int_{\Omega_0} (|\det(\mathbf{D}T_{\mathcal{V}})|) \delta_{\tilde{\mathcal{V}}} u \, d\mathbf{x} & = 0 \end{cases}, \tag{3.12}$$

which can be solved to recover $(\delta_{\tilde{\mathcal{V}}} u, \tilde{\lambda})$[3]. As described in [3, Par. 1.6], though, this would require to find a solution to a PDE for every choice of $\tilde{\mathcal{V}}$, which is often unfeasible. For this reason, the so-called *Adjoint approach* [3, Par. 1.6.2] is preferred, whose characteristics are explained next.

3.4.1 Lagrange function

Similarly to how it has been done with the state equation (2.2) to take into account the additional constraint (2.4), we can modify the cost functional (2.8) by considering the constraint imposed by (2.6). One way to do this is to re-cast (2.9) as a saddle-point problem, following the procedure described in [3, Par. 1.4.6]. Accordingly, we introduce the corresponding *Lagrangian function*:

Definition 3.7. *Lagrangian function associated to* (2.9)

$$\mathcal{L} : (C^2(\overline{D}; \mathbb{R}^3) \times H^1_{\Sigma_0}(\Omega_0) \times \mathbb{R} \times (H^1_{\Sigma_0}(\Omega_0)^*)^* \times \mathbb{R}) \to \mathbb{R},$$

$$\begin{aligned} (\mathcal{V}, u, \lambda, z, \mu) \mapsto \mathcal{L}(\mathcal{V}, u, \lambda, z, \mu) \coloneqq \quad & \mathcal{J}(\mathcal{V}, u) + \textstyle\int_{\Omega_0 \setminus \Sigma_0} \nabla u \, \mathbf{M}_{\mathcal{V}} \nabla z \, d\mathbf{x} \\ & + I\,[z]_{\Sigma_0} + \lambda \textstyle\int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, z \, d\mathbf{x} \\ & + \mu \textstyle\int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, u \, d\mathbf{x}. \end{aligned} \tag{3.13}$$

Here, $(H^1_{\Sigma}(\Omega_0)^*)^*$ indicates the double dual space of $H^1_{\Sigma}(\Omega_0)$, which can be identified with $H^1_{\Sigma}(\Omega_0)$ itself by means of Riesz representation theorem [3, Par. 1.2]; $z \in H^1_{\Sigma}(\Omega_0)$ covers the role of the Lagrangian multiplier with respect to the constraint (2.6).

Solving (2.9) is equivalent to finding the saddle point of $\mathcal{L}(\mathcal{V}, u, \lambda, z, \mu)$, which can be done by setting all its partial derivatives to 0. Partial differentiation of $\mathcal{L}(\mathcal{V}, u, \lambda, z, \mu)$ with respect to the Lagrangian multipliers μ and z gives the corresponding constraints, (2.6). Differentiating with respect to λ produces a constraint on z that is analogous to (2.4). Lastly, differentiations with respect to $\mathcal{V}$ and u provide respectively a representation of the gradient of $\mathcal{J}$, described in Sec. 3.4.3, and an additional equation, presented in Sec. 3.4.2.

3.4.2 Adjoint equation

We need to find $(z, \mu) \in H^1_{\Sigma_0}(\Omega_0) \times \mathbb{R}$ such that $\langle \mathcal{L}_u(\mathcal{V}, u, \lambda, z, \mu), v \rangle = 0 \;\; \forall v \in H^1_{\Sigma_0}(\Omega_0)$ and $\langle \mathcal{L}_\lambda(\mathcal{V}, u, \lambda, z, \mu), \nu \rangle = 0 \;\; \forall \nu \in \mathbb{R}$. It is straightforward to see that this translates

[3] The exact same result can also be found as a consequence of the implicit function theorem [17]. See for example [3, Par. 1.4.2].

into the following saddle-point problem:
Find $(z,\mu) \in H^1_{\Sigma_0}(\Omega_0) \times \mathbb{R} : \forall (v,\nu) \in H^1_{\Sigma_0}(\Omega_0) \times \mathbb{R}$,

$$\begin{cases} \int_{\Omega_0 \setminus \Sigma_0} \nabla v \, \mathbf{M}_{\mathcal{V}} \nabla z \, d\mathbf{x} + \mu \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, v \, d\mathbf{x} = -\langle \mathcal{J}_u, v \rangle \\ \nu \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, z \, d\mathbf{x} = 0 \end{cases} . \tag{3.14}$$

This equation is also referred to as *adjoint equation*, and its solution z is called *adjoint variable*.

3.4.3 Adjoint gradient representation

If we choose (z,μ) as the solutions of (3.14), a simpler and more suitable representation of the gradient of the cost functional (3.6) can be given that does not make use of the sensitivity (3.8).

Proposition 3.8. The first order Fréchet derivative of the cost functional (2.8), evaluated along the direction $\tilde{\mathcal{V}}$, can also be expressed as

$$\begin{aligned} \left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle &= \left\langle \mathcal{J}_{\mathcal{V}}, \tilde{\mathcal{V}} \right\rangle + \int_{\Omega_0 \setminus \Sigma_0} \nabla u \, \delta_{\tilde{\mathcal{V}}} \mathbf{M}_{\mathcal{V}} \nabla z \\ &+ \lambda \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \operatorname{tr}(\mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}\tilde{\mathcal{V}}) z \, d\mathbf{x} \\ &+ \mu \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \operatorname{tr}(\mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}\tilde{\mathcal{V}}) u \, d\mathbf{x}, \end{aligned} \tag{3.15}$$

with (z,μ) solutions of (3.14) and (u,λ) solutions of (2.6).

Proof. We can immediately notice that, whenever the constraints (2.6) (together with the equivalent of (2.4) for z) are satisfied, we have that $\mathcal{J}(\mathcal{V},u) = \mathcal{L}(\mathcal{V},u,\lambda,z,\mu)$, $\forall z \in H^1_{\Sigma_0}(\Omega_0)$ and $\forall \lambda,\mu \in \mathbb{R}$. Moreover, considering $u = u(\mathcal{V})$ and differentiating with respect to $\mathcal{V}$ gives us

$$\begin{aligned} \left\langle \mathcal{L}'(\mathcal{V},u(\mathcal{V}),\lambda,z,\mu), \tilde{\mathcal{V}} \right\rangle &= \langle \mathcal{L}_u(\mathcal{V},u(\mathcal{V}),\lambda,z,\mu), \delta_{\tilde{\mathcal{V}}} u \rangle \\ &+ \left\langle \mathcal{L}_{\mathcal{V}}(\mathcal{V},u(\mathcal{V}),\lambda,z,\mu), \tilde{\mathcal{V}} \right\rangle, \end{aligned}$$

which is equal to the derivative of $\mathcal{J}$ along $\tilde{\mathcal{V}}$, for the reason previously stated. The sensitivity term $\delta_{\tilde{\mathcal{V}}} u$ still appears in this formulation, but it can be easily dropped if z is chosen as solution of (3.14). Hence, we just need to show that (3.15) corresponds to $\left\langle \mathcal{L}_{\mathcal{V}}, \tilde{\mathcal{V}} \right\rangle$. The first term in (3.14) comes directly from Def. 3.7; for the third and fourth terms, formula (3.1) has been once again used; for the second, the linearity of the operators involved has been applied. An expression for $\delta_{\tilde{\mathcal{V}}} \mathbf{M}_{\mathcal{V}}$ has previously been given in (3.4). □

Chapter 4

Algorithms

To solve the shape optimization problem (2.9), an algorithm belonging to the class of *descent methods* [3, Chap. 2.2] has been set up. In this chapter, its detailed description is provided.

4.1 Descent method

The basic form of the algorithm is analogous to the one presented in [6]. A pseudo-code version of it is reported in Alg. 1.

Algorithm 1: Generic descent method

Start from an initial design Ω_0;
Initialize current map $\mathcal{V} = 0$ and current update $\tilde{\mathcal{V}} = 0$;
for *k=0 to* it_{max} **do**
    Evaluate solution $u(\mathcal{V})$ of (2.6);
    Evaluate cost functional at updated position: $\mathcal{J}^k = \mathcal{J}(\mathcal{V} + \tilde{\sigma}\tilde{\mathcal{V}}, u(\mathcal{V} + \tilde{\sigma}\tilde{\mathcal{V}}))$;
    if $(k > 0)$ *and update is not admissible according to Armijo rule* **then**
        Halve step size: $\tilde{\sigma} = \tilde{\sigma}/2$;
    else
        Set current map $\mathcal{V} = \mathcal{V} + \tilde{\sigma}\tilde{\mathcal{V}}$;
        Evaluate gradient at current position: $\mathcal{J}'(\mathcal{V})$;
        Choose a descent direction: $\tilde{\mathcal{V}}$;
        Choose a step size: $\tilde{\sigma} = \sigma$;
        while $\left(\min(\det\left(\mathbf{D}T_{\mathcal{V}+\tilde{\sigma}\tilde{\mathcal{V}}}\right)(\Omega_0)) < \epsilon\right)$ **do**
            Halve step size: $\tilde{\sigma} = \tilde{\sigma}/2$;

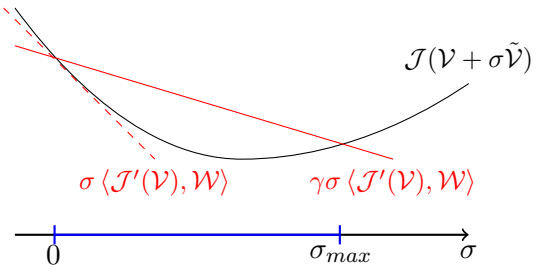

FIGURE 4.1: Effect of applying Armijo rule for finding an admissible step size. The range of admissible σ is highlighted in blue, and is influenced by the choice of γ, that modifies the inclination of the red line. $\tilde{\sigma}$ in (4.1) is found by progressively halving an initial choice for the step size.

The reason why the condition $\min(\det(\mathbf{D}T_{\mathcal{V}})(\Omega_0)) > \epsilon$ must be fulfilled is to ensure that the map $T_{\mathcal{V}}$ remains a diffeomorphism. How strictly this constraint must be respected is indicated by ϵ, which is a fixed parameter, just like the initial step size σ.

4.2 Armijo rule

The condition called *Armijo rule*, which needs to be satisfied from the first iteration on, ensures that the chosen update step size $\tilde{\sigma}$ is admissible [3, Par.2.2.1.1], and it can be expressed as follows:
Find the maximum $\tilde{\sigma} \in \{\sigma, \sigma/2, \sigma/4, \sigma/8, \cdots\}$ so that

$$\mathcal{J}(\mathcal{V}+\tilde{\sigma}\tilde{\mathcal{V}}) - \mathcal{J}(\mathcal{V}) > \gamma\tilde{\sigma}\left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}}\right\rangle. \tag{4.1}$$

Again, $\gamma \in (0,1)$ is a fixed parameter that relaxes the condition the closer it gets to 0. Its effect is described in Fig. 4.1.

4.3 Steepest descent method

In general, the methods used to recover the descent direction $\tilde{\mathcal{V}}$ and the step size $\tilde{\sigma}$ determine the algorithm. As described above, in this work a fixed initial step size σ is chosen, while for the update $\tilde{\mathcal{V}}$, the opposite of a representative of the gradient $\mathcal{J}'(\mathcal{V})$ is selected. This identifies the algorithm as a *steepest descent method*. The representative is found by solving the problem[1]

$$(\tilde{\mathcal{V}}, \mathcal{W})_* = -\left\langle \mathcal{J}'(\mathcal{V}), \mathcal{W}\right\rangle \quad \forall \mathcal{W} \in (*) \tag{4.2}$$

[1]This procedure provides the so-called Riesz representative in $(*)$ of the functional $-\mathcal{J}'(\mathcal{V})$.

for a particular scalar product $(\cdot,\cdot)_*$, and then by rescaling it so that $\left\|\tilde{\mathcal{V}}\right\|_* = 1$[2]. In our work, the H^1 norm is chosen, since after a few first experiments it has shown better stability than the L^2 or euclidean counterparts. Being H_{H^1} the operator associated to the H^1 scalar product, so that

$$\langle H_{H^1}(\mathcal{V}), \mathcal{W}\rangle = (\mathcal{V}, \mathcal{W})_{H^1} \quad \forall \mathcal{V}, \mathcal{W} \in H^1, \tag{4.3}$$

the normalized solution to (4.2) can be expressed as

$$\tilde{\mathcal{V}} = -\frac{H_{H^1}^{-1}(\mathcal{J}'(\mathcal{V}))}{\sqrt{\langle \mathcal{J}'(\mathcal{V}), H_{H^1}^{-1}(\mathcal{J}'(\mathcal{V}))\rangle}}. \tag{4.4}$$

More information on how to find the discretized solution to (4.2) is given in Sec. 5.2.

[2] $\|\cdot\|_*$ is the norm induced by $(\cdot,\cdot)_*$, *i.e.*, $\|\cdot\|_* = \sqrt{(\cdot,\cdot)_*}$

Chapter 5

Discretization aspects

In order to recover a numerical solution of the problem (2.9), discrete approximations for the state variable u, the adjoint variable z and the control variable $\mathcal{V}$ need to be given. The details are described in the following.

5.1 State and adjoint

A *Finite-Element Method* (FEM) [26, Chap. 4] is used to provide a discretization for both the state variable $u_h \approx u$ and the adjoint variable $z_h \approx z$. To do this, a set of piecewise linear Lagrangian basis functions ϕ_i is constructed on a tetrahedral mesh that covers the 3D domain Ω_0. These functions are defined such that $\phi_i(\mathbf{x_j}) = \delta_{i,j}$ for each node $\mathbf{x_j}$ of the mesh. The purpose is to express our approximations u_h and z_h as linear combinations of these basis functions. Unfortunately, both variables still cannot be adequately represented, since they show a jump discontinuity across the surface Σ_0. To overcome this problem, the set of ϕ_i is expanded by considering also a so-called *cut-off basis function*, ϕ_{CO}, in a procedure similar to the one described in [27], although very simplified. This function has the following characteristics:

- $[\phi_{CO}]_{\Sigma_0}(\mathbf{x_i}) = 1 \quad \forall \mathbf{x_i} \in \Sigma_0$,
- $\phi_{CO}(\mathbf{x_i}) = 0 \quad \forall \mathbf{x_i} \notin \Sigma_0$.

A sketch of how such a function could look like in a 1D domain is shown in Fig. 5.1. We can then approximate state and adjoint as

$$u_h = \sum_{i=1}^{n} u_i \phi_i + u_{CO}\phi_{CO}, \quad z_h = \sum_{i=1}^{n} z_i \phi_i + z_{CO}\phi_{CO}, \tag{5.1}$$

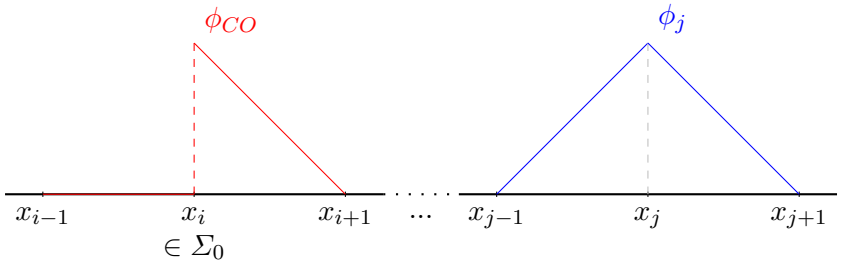

FIGURE 5.1: Sketch of the behaviour for a generic ϕ_j linear basis function (right) and for the cut-off basis function ϕ_{CO} (left) on a 1D mesh.

where n is the number of mesh nodes (which in our case corresponds to the number of canonical basis functions).

PDEs like (2.2) or (3.14) can be approximated via a Galerkin method (again, [26, Chap. 4]), which for our problem gives rise to a linear system in the form $\mathbf{Ax} = \mathbf{b}$. If we consider (2.6), we have

$$\mathbf{A}_{ij} = \int_{\Omega_0} \nabla \phi_j \mathbf{M}_{\mathcal{V}} \nabla \phi_i \, d\mathbf{x}, \tag{5.2}$$

for $i, j = 1 : n$. Here, $\mathbf{A}$ is the classical *stiffness matrix*. In our case, we need to fringe it with an additional row/column to take into account the contributions of the cut-off basis function. Furthermore, another row/column needs to be added in order to consider also the influence of the constraint (2.4) and build the complete matrix associated to the saddle-point problem (2.6). This last row represents the equation

$$\sum_{j=1}^{n} \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, u_j \phi_j \, d\mathbf{x} + \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, u_{CO} \phi_{CO} \, d\mathbf{x} = 0,$$

while the last column contains the terms

$$\lambda \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, \phi_i \, d\mathbf{x}$$

that are added to the weak formulation (2.5). The system matrix $\mathbf{A}_{st}$ for (2.6) looks as

$$\mathbf{A}_{st} = \left[\begin{array}{ccc|c|c} & & & \vdots & \vdots \\ & \mathbf{A} & & \int_{\Omega_0 \setminus \Sigma_0} \nabla \phi_{CO} \mathbf{M}_{\mathcal{V}} \nabla \phi_i \, d\mathbf{x} & \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, \phi_i \, d\mathbf{x} \\ & & & \vdots & \vdots \\ \hline \cdots & \int_{\Omega_0 \setminus \Sigma_0} \nabla \phi_j \mathbf{M}_{\mathcal{V}} \nabla \phi_{CO} \, d\mathbf{x} & \cdots & \int_{\Omega_0 \setminus \Sigma_0} \nabla \phi_{CO} \mathbf{M}_{\mathcal{V}} \nabla \phi_{CO} \, d\mathbf{x} & \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, \phi_{CO} \, d\mathbf{x} \\ \hline \cdots & \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, \phi_j \, d\mathbf{x} & \cdots & \int_{\Omega_0} |\det(\mathbf{D}T_{\mathcal{V}})| \, \phi_{CO} \, d\mathbf{x} & 0 \end{array}\right], \tag{5.3}$$

where $\mathbf{A}$ is the matrix defined in (5.2). Accordingly, the solution vector $\mathbf{x}_{st}$ is in the form $[\cdots, u_i, \cdots, u_{CO}, \lambda]^T$, with u_i and u_{CO} from (5.1) and λ being the Lagrangian multiplier introduced in (2.4), while the right-hand side of the system, $\mathbf{b}_{st}$, is $[\cdots, 0, \cdots, -I, 0]^T$, where I is the imposed current, as in (2.2).

The matrix associated to the adjoint equation (3.14) is $\mathbf{A}_{adj} = \mathbf{A}_{st}^T$. However, as pointed out in Sec. 2.4.1, the pull-back matrix in Def. 2.5 is symmetric. Consequently, also $\mathbf{A}_{st}$ is symmetric, and the same matrix (5.3) can be used for both state and adjoint problem. As for the right-hand side, $\mathbf{b}_{adj} = [\cdots, -\langle \mathcal{J}_u, \phi_i \rangle, \cdots, -\langle \mathcal{J}_u, \phi_{CO} \rangle, 0]^T$. Analogously to the state, the solution vector of the adjoint saddle-point problem $\mathbf{x}_{adj}$ is $[\cdots, z_i, \cdots, z_{CO}, \mu]^T$.

5.2 Control

The control variable $\mathcal{V}$ too is represented as linear combination of basis functions, although of a different kind. To describe them, we introduce a uniform Cartesian grid that discretizes a cylindrical domain[1] Ω_{SP} surrounding the toroidal domain Ω_0, as in Fig. 5.2. On it, we define a set of m basis functions ψ_i composed of quadratic B-splines [25, Chap. 7.6]. We consider discrete vector fields $\mathcal{V}_h$ in the form

$$\mathcal{V}_h(\mathbf{x}) = \sum_{i=1}^{m} \psi_i(\mathbf{x}) \sum_{d=1}^{3} \nu_{i,d}\, \mathbf{e_d} = \sum_{i=1}^{m} \psi_i(\mathbf{x}) \boldsymbol{\nu}_i, \quad \mathbf{x} \in \Omega_{SP}, \tag{5.4}$$

with $\mathbf{e_d}$ being the basis vectors of $\mathbb{R}^3$ and $\boldsymbol{\nu}_i = (\nu_{i,1}, \nu_{i,2}, \nu_{i,3})^T$. Every $\mathcal{V}_h$ is uniquely characterized by its corresponding coefficients $\boldsymbol{\nu}_i$.

Spline basis functions definition More in detail, in this work the ψ_i are defined on a local cylindrical coordinates system $(\hat{\rho}, \hat{\theta}, \hat{z})^T \in [0, 1]^3$ that covers a cell of the grid. They are expressed as a tensor product of three independent sets of 1D spline functions, one for each variable: $\hat{\psi}^\rho(\hat{\rho})$, $\hat{\psi}^\theta(\hat{\theta})$ and $\hat{\psi}^z(\hat{z})$. In general, the definition for the 1D splines is given recursively via the *Cox-De Boor recursion formula* and depends on the order chosen and on the distribution of control nodes [18, Chap. 3]. In our case, a simplified expression is available, and it is explicitly reported for splines of order $N = 1$

[1] This particular choice for the shape of the splines domain Ω_{SP} has the remarkable advantage of noticeably simplifying evaluations, and is dictated by the configuration of the initial domain Ω_0, which is highly symmetrical. Nonetheless, a more general approach with an arbitrary domain and grid remains feasible.

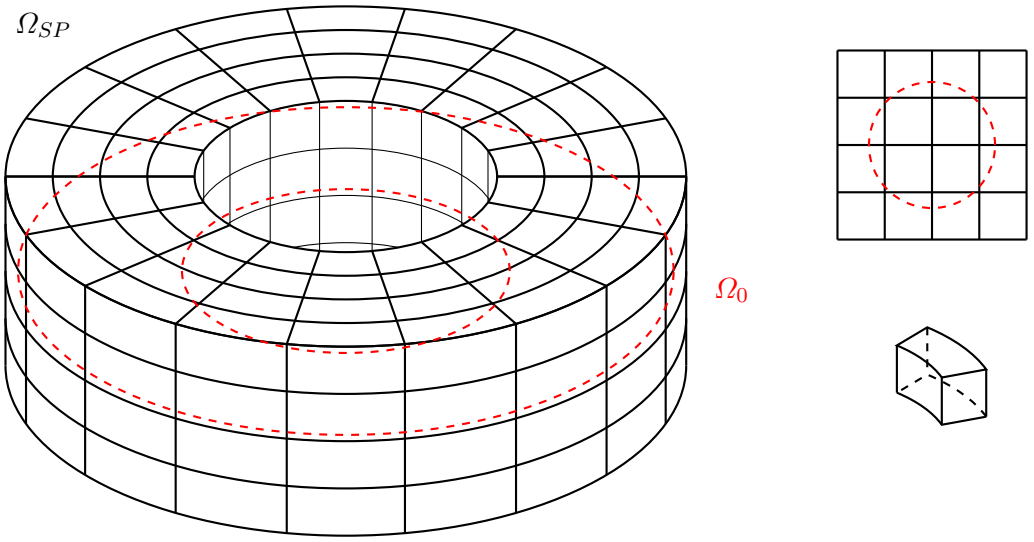

FIGURE 5.2: Representation of the domain and grid for the spline basis functions Ω_{SP}. On the left, we can see how the splines grid is placed so to surround the domain Ω_0 completely (here drawn in a red dashed line). On the top right, a cross-section of the grid itself is given, to better show its reciprocal position with Ω_0. On the bottom right, a grid cell is presented in detail.

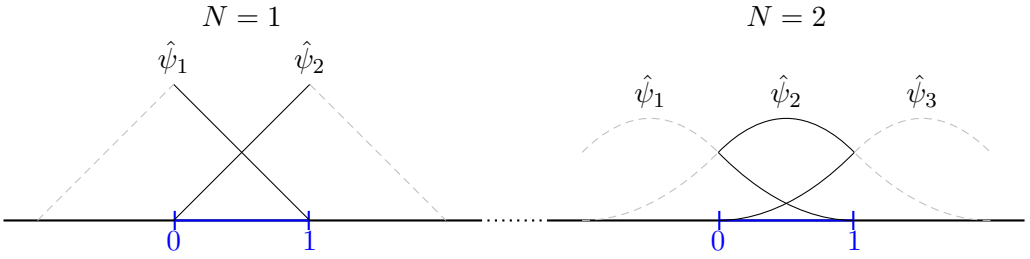

FIGURE 5.3: Possible shapes for the 1D spline basis functions of order 1 (left) and 2 (right) defined on the interval $[0, 1]$. It is noticeable how the size of the support of such functions depends uniquely on the order chosen: splines of order N have support on $N + 1$ cells. This property translates also to 3D basis functions created as tensor products: they have support on $(N + 1)^3$ cells.

and $N = 2$ in (5.5). Their graph is instead shown in Fig. 5.3.

$$\begin{aligned} &N = 1 && && N = 2 \\ &\hat{\psi}_1(x) &=& 1 - x && \hat{\psi}_1(x) &=& (x-1)^2/2 \\ &\hat{\psi}_2(x) &=& x && \hat{\psi}_2(x) &=& -x^2 + x + 1/2 \\ & && && \hat{\psi}_3(x) &=& x^2/2, \end{aligned} \tag{5.5}$$

with $x \in [0, 1]$. This definition is the same for $\hat{\psi}^\rho(\hat{\rho})$, $\hat{\psi}^\theta(\hat{\theta})$ and $\hat{\psi}^z(\hat{z})$.

Maps from local to cylindrical to Cartesian coordinates To describe how the values of ψ are evaluated starting from the values of $\hat{\psi}^\rho$, $\hat{\psi}^\theta$ and $\hat{\psi}^z$, we need to introduce two transformations.

- F, the map from cylindrical to Cartesian coordinates:

$$\begin{aligned} F : [0,\infty) \times [0,2\pi) \times \mathbb{R} &\to \mathbb{R}^3, \\ (\rho,\theta,z)^T &\mapsto \mathbf{x} = (\rho\cos(\theta), \rho\sin(\theta), z)^T. \end{aligned} \tag{5.6}$$

- C, the map from local to cylindrical coordinates:

$$\begin{aligned} &C : [0,1]^3 \to [0,\infty) \times [0,2\pi) \times \mathbb{R}, \\ &(\hat{\rho},\hat{\theta},\hat{z})^T \mapsto (\rho,\theta,z)^T = (\rho_0 + \hat{\rho}\Delta\rho, \theta_0 + \hat{\theta}\Delta\theta, z_0 + \hat{z}\Delta z)^T. \end{aligned} \tag{5.7}$$

In our work, ρ_0, θ_0 and z_0 are cell-dependent, while $\Delta\rho$, $\Delta\theta$ and Δz are constant for every cell, and they vary according to the level of refinement of the mesh. Using F and C, it is possible to recover the values of $\psi_i(\mathbf{x})$ and of its gradient in Cartesian coordinates, $\nabla\psi_i(\mathbf{x})$ starting from the values of the 1D spline basis functions[2] (5.5):

$$\begin{aligned} \psi_i(\mathbf{x}) &= \hat{\psi}^\rho_j(\hat{\rho}) \cdot \hat{\psi}^\theta_k(\hat{\theta}) \cdot \hat{\psi}^z_l(\hat{z}), \\ \nabla\psi_i(\mathbf{x}) &= \mathbf{D}F^{-T}\mathbf{D}C^{-T}\hat{\nabla}(\hat{\psi}^\rho_j(\hat{\rho}) \cdot \hat{\psi}^\theta_k(\hat{\theta}) \cdot \hat{\psi}^z_l(\hat{z})) \\ &= \begin{bmatrix} \frac{\cos(\theta)}{\Delta\rho} & -\frac{\sin(\theta)}{\rho\Delta\theta} & 0 \\ \frac{\sin(\theta)}{\Delta\rho} & -\frac{\cos(\theta)}{\rho\Delta\theta} & 0 \\ 0 & 0 & \frac{1}{\Delta z} \end{bmatrix} \begin{bmatrix} d\hat{\psi}^\rho_j(\hat{\rho}) \cdot \hat{\psi}^\theta_k(\hat{\theta}) \cdot \hat{\psi}^z_l(\hat{z}) \\ \hat{\psi}^\rho_j(\hat{\rho}) \cdot d\hat{\psi}^\theta_k(\hat{\theta}) \cdot \hat{\psi}^z_l(\hat{z}) \\ \hat{\psi}^\rho_j(\hat{\rho}) \cdot \hat{\psi}^\theta_k(\hat{\theta}) \cdot d\hat{\psi}^z_l(\hat{z}) \end{bmatrix}, \end{aligned} \tag{5.8}$$

with $\hat{\rho}$, $\hat{\theta}$, $\hat{z}$ and ρ, θ, z such that $\mathbf{x} = F((\rho,\theta,z)^T) = F(C((\hat{\rho},\hat{\theta},\hat{z})^T))$. The writing $\hat{\nabla}(\cdot)$ denotes the gradient in local coordinates $(\hat{\rho},\hat{\theta},\hat{z})$ of $(\cdot)$, while $d(\cdot)$ is the (total) derivative of $(\cdot)$. By substituting (5.4) in Def. 2.4 it can be seen how formulas (5.8) are necessary to compute the approximated map $T_{\mathcal{V}_h}$ and its Jacobian in Cartesian coordinates $\mathbf{D}T_{\mathcal{V}_h}$. In fact, we have:

$$\begin{aligned} T_{\mathcal{V}_h}(\mathbf{x}) &= \mathbf{x} + \mathcal{V}_h(\mathbf{x}) \\ &= \mathbf{x} + \textstyle\sum_{i=1}^m \psi_i(\mathbf{x}) \sum_{d=1}^3 \nu_{i,d}\,\mathbf{e_d} \\ &= \mathcal{I} + \textstyle\sum_{i=1}^m \begin{bmatrix} \nu_{i,1}\psi_i(\mathbf{x}) \\ \nu_{i,2}\psi_i(\mathbf{x}) \\ \nu_{i,3}\psi_i(\mathbf{x}) \end{bmatrix} \\ &= \mathcal{I} + \textstyle\sum_{i=1}^m \boldsymbol{\nu}_i\psi_i, \\ \mathbf{D}T_{\mathcal{V}_h}(\mathbf{x}) &= \mathbf{D}(\mathbf{x} + \mathcal{V}_h(\mathbf{x})) \\ &= \mathcal{I} + \textstyle\sum_{i=1}^m \sum_{d=1}^3 \nu_{i,d}\,\mathbf{D}(\psi_i(\mathbf{x})\mathbf{e_d}) \\ &= \mathcal{I} + \textstyle\sum_{i=1}^m \begin{bmatrix} \nu_{i,1}\nabla\psi_i(\mathbf{x})^T \\ \nu_{i,2}\nabla\psi_i(\mathbf{x})^T \\ \nu_{i,3}\nabla\psi_i(\mathbf{x})^T \end{bmatrix} \\ &= \mathcal{I} + \textstyle\sum_{i=1}^m \boldsymbol{\nu}_i\nabla\psi_i(\mathbf{x})^T. \end{aligned} \tag{5.9}$$

[2] The proof for this is similar to the one used for recovering Def. 2.5.

Gram matrix for H^1 scalar product Using the discretization introduced for the control (5.4), the bilinear form H_{H^1} presented in (4.3) can be substituted with the so-called *Gram matrix* [16] associated to the H^1 norm, $\mathbf{G}_{H^1}$. We have, in fact,

$$\begin{aligned}\langle \mathcal{V}_h, \mathcal{W}_h \rangle_{H^1} &= \textstyle\int_{\Omega_{SP}} [\mathcal{V}_h(\mathbf{x}) \cdot \mathcal{W}_h(\mathbf{x}) + \mathbf{D}\mathcal{V}_h(\mathbf{x}) : \mathbf{D}\mathcal{W}_h(\mathbf{x})] \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m} \int_{\Omega_{SP}} [\psi_i(\mathbf{x})\psi_j(\mathbf{x}) \, (\boldsymbol{\nu}_i \cdot \boldsymbol{w}_j) \\ &+ \quad (\nabla \psi_i(\mathbf{x}) \cdot \nabla \psi_j(\mathbf{x})) \, (\boldsymbol{\nu}_i \cdot \boldsymbol{w}_j)] \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m} \int_{\Omega_{SP}} (\psi_i(\mathbf{x})\psi_j(\mathbf{x}) + \nabla \psi_i(\mathbf{x}) \cdot \nabla \psi_j(\mathbf{x}))(\boldsymbol{\nu}_i \cdot \boldsymbol{w}_j)\end{aligned}$$

where $\boldsymbol{w}_i$ are the coefficients of $\tilde{\mathcal{W}}$. If we define $\mathbf{G}_{H^1}$ such that

$$(\mathbf{G}_{H^1})_{i,j} \coloneqq \int_{\Omega_{SP}} \psi_i(\mathbf{x})\psi_j(\mathbf{x}) + \nabla \psi_i(\mathbf{x}) \cdot \nabla \psi_j(\mathbf{x}), \tag{5.10}$$

it is easy to see how

$$\langle \mathcal{V}_h, \mathcal{W}_h \rangle_{H^1} = \sum_{d=1}^{3}\sum_{i=1}^{m}\sum_{j=1}^{m} w_{j,d}(\mathbf{G}_{H^1})_{i,j}\nu_{i,d} = \sum_{i=1}^{m}\sum_{j=1}^{m}(\mathbf{G}_{H^1})_{i,j}(\boldsymbol{\nu}_j \cdot \boldsymbol{w}_i). \tag{5.11}$$

The discretized solution of (4.2) becomes then

$$(\tilde{\mathcal{V}}_h)_d = -\frac{\mathbf{G}_{H^1}^{-1}(\mathcal{J}_h'(\mathcal{V}_h))_d}{\sqrt{(\mathcal{J}_h'(\mathcal{V}_h))_d^T \mathbf{G}_{H^1}^{-T}(\mathcal{J}_h'(\mathcal{V}_h))_d}}, \tag{5.12}$$

where $(\tilde{\mathcal{V}}_h)_d$ are the d-components of the coefficients $\boldsymbol{\nu}_i$ in (5.4), while with $(\mathcal{J}_h'(\mathcal{V}_h))_d$ we denote the evaluations of $\mathcal{J}'$ at $\mathcal{V}_h$, tested with the spline basis functions ψ that refer to the d-component of the spline-interpolated vector field.

From (5.9) and (5.11), it is already clear how the fact that the same ψ_i are used to describe all three components of the vector field $\mathcal{V}_h$ allows some remarkable simplifications in the computation of the derivatives of (2.8). This and other aspects related to the actual implementation of the code are discussed next.

Chapter 6

Implementation aspects

The code implementation makes use of the C++ template library BETL2 (*Boundary Element Template Library 2*), [19, 20] developed by Lars Kielhorn and Raffael Casagrande for the *Seminar for Applied Mathematics* at ETH Zürich. Although, as the name suggests, BETL2 is originally developed as a tool for *Boundary Element* applications [21], among its functionalities there are various methods for building FEM solvers, which have been widely used in this work.

Next, a detailed description of the implementation is given.

6.1 Most relevant classes

Taking advantage of the perks of an Object-Oriented language like C++, the code has been subdivided into classes, each with different scopes and purposes. Even though some of them are made so to be open to generalization and various parts of the code can be re-used for the development of other tools, many of them remain hard-coded and application-specific, due to efficiency considerations and time limitations. Here, the most important ones are described.

`ShapeOptimizationSolver` This basically represents a Façade class [22] for the whole shape optimization problem. It is responsible for the initialization of the finite elements and the splines grid, as well as their respective basis function spaces. While for the finite element part some built-in functions of BETL2 have been used, the implementation of the splines is handled by instantiations of `SplinesGrid` and `SplinesBasisFunc`, both internal members of `ShapeOptimizationSolver`. In addition, this class stores information about the solutions for state (2.6) and adjoint (3.14) problems, and about

the evolution of the cost functional (2.8). Its main purpose is to run the primary routine for the optimization process, previously described in Alg. 1 and presented in Lst. 6.1.

```
for( iteration_=0; iteration_<maxIterations_; iteration_++ ){

  solveStateEquation();
  evaluateCostFunctional();

  if( iteration_== 0 || CheckArmijoRulePassed() ){

    solveAdjointEquation();
    evaluateShapeDerivative();
    updateShape();

    exportSolution();
  }

  std::cout<<"Iteration "<<iteration_<<" concluded."<<std::endl;
}

//print evolution of cost functional
printJ();

```

Listing 6.1: Main function of `ShapeOptimizationSolver`.

The output too is handled by the class `ShapeOptimizationSolver` via the function `exportSolution()` reported in Lst. 6.1: it exports relevant information throughout the optimization process, like the approximations of the mapped domain $T_{\mathcal{V}_h}(\Omega_0)$, of the magnetic field on the surface of the internal object $\mathbf{H}_h(\partial\mathcal{D})$ and of the solutions u_h, z_h to state and adjoint problems. The actual evaluation of these, though, is conducted by invoking the methods of another internal class, `ProblemsSolver`. Function `updateShape()` is responsible also for the check on the positiveness of the determinant as described in Alg. 1; this control is conducted by computing $\det(\mathbf{D}T_{\mathcal{V}})$ on various points inside the splines grid, more details in Lst. A.5. Function `checkArmijoRulePassed()`, as the name suggests, applies (4.1) and halves the step size in case this check fails. The actual code for this is in Lst. 6.2.

```
const numeric_t gamma = 0.1;
const numeric_t left  = J_( iteration_ ) - J_( iteration_ - 1 );
const numeric_t right = gamma * ( dJ_.cwiseProduct( shapeUpdate_ ) ).sum();

if( left > right ){
  //remove half of the update:
  shapeUpdate_ *= 0.5;
  spBasis_.updateCoeffNU( -shapeUpdate );
```

```
9
10    //keep value of J_ stationary
11    J_( iteration_ ) = J_(iteration_ - 1);
12
13    return false;
14  }
15
16  return true;
17
```

LISTING 6.2: Implementation of Armijo rule (4.1).

`SplinesGrid` It is the class that takes care of the implementation of the uniform Cartesian splines grid described in Sec. 5.2. It provides methods for the various direct and inverse mappings from local to cylindrical and global coordinates, described in (5.7) and (5.6). It also allows for the evaluation of the Jacobians of these mappings, as needed in (5.8). More information about these methods can be found in Lst. A.1 and Lst. A.2.

`SplinesBasisFunc` This class works together with `SplinesGrid` in order to provide a complete description of the approximated control (5.4) and the spline basis functions ψ_i it is built on. The order of the splines used is accepted as a template parameter, but the only templated methods are the one responsible for the evaluation of the 1D spline basis functions (5.5) and their derivatives, given a local coordinate $(\hat{\rho}, \hat{\theta}, \hat{z})^T$. This makes it extremely simple to extend it to use splines of various orders. The class is already equipped with an implementation for linear and quadratic splines, the latter kind used for the largest part of this work. In Lst. 6.3 it can be found an example of the implementation for splines of order $N = 2$,

```
1  //there are 3 types of splines of order 2 with support on the same cell:
       (1-x)^2/2, -x^2+x+1/2, x^2/2
2  results<<0.5*(1-rho)*(1-rho), - rho*rho + rho +0.5, 0.5*rho*rho,
3           0.5*(1- th)*(1- th), - th * th + th  +0.5, 0.5*th * th,
4           0.5*(1- z )*(1- z ), -  z * z  + z   +0.5, 0.5* z * z;
5
```

LISTING 6.3: Evaluation of 1D spline basis functions $\hat{\psi}^{\rho}(\hat{\rho})$, $\hat{\psi}^{\theta}(\hat{\theta})$, $\hat{\psi}^{z}(\hat{z})$, of order $N = 2$.

while in Lst. 6.4 their derivatives are computed.

```
1  //there are 3 types of derivatives of splines of order 2 with support on
       the same cell: x-1, -2x+1, x
2  results << rho-1, - 2*rho + 1, rho,
3             th -1, - 2* th + 1, th,
```

```
4            z - 1, - 2 * z + 1, z;
5
```

LISTING 6.4: Evaluation of derivatives of 1D spline basis functions $d\hat{\psi}^{\rho}(\hat{\rho})$, $d\hat{\psi}^{\theta}(\hat{\theta})$, $d\hat{\psi}^{z}(\hat{z})$, of order $N = 2$.

Here, `rho`, `th`, `z` are the local coordinates $\hat{\rho}$, $\hat{\theta}$, $\hat{z}$. Regarding the evaluation of 3D spline basis functions, the corresponding routine is reported in Lst. 6.5.

```
1  //First, get all possible evaluations of 1D functions psi_j(rho), psi_k(
       theta), psi_l(z)  and their derivatives
2  matrix_t< dim, ORDER+1 > tempRes, tempResDeriv;
3  evaluateSplines1DLocal( localCoord, tempRes );
4  evaluateSplinesDeriv1DLocal( localCoord, tempResDeriv );
5
6  //Now, evaluate all possible psi_i = psi_j(rho)*psi_k(theta)*psi_l(z) and
       their derivatives wrt rho, theta, z, and store their value in results
       and dResults, respectively
7  idx_t i = 0;
8  for(idx_t l=0; l < ORDER+1; l++){
9    for(idx_t k=0; k < ORDER+1; k++){
10     for(idx_t j=0; j < ORDER+1; j++){
11       results(i) = tempRes(0,j)*tempRes(1,k)*tempRes(2,l);
12
13       dResults(0,i)=tempResDeriv(0,j)*tempRes(1,k)*tempRes(2,l);
14       dResults(1,i)=tempRes(0,j)*tempResDeriv(1,k)*tempRes(2,l);
15       dResults(2,i)=tempRes(0,j)*tempRes(1,k)*tempResDeriv(2,l);
16       i++;
17     }
18   }
19 }
20 /*
21 Every column of results/dResults contains the value/gradient of a 3D spline
       basis function psi_i with support on localCoord. These splines are
       cycled in the following fashion, with respect to the node they refer to
       :
22   -From the center to the external border (increasing rho)
23   -Anti-clockwise (increasing theta)
24   -From bottom to top (incresing z)
25 */
26
```

LISTING 6.5: Evaluation of 3D spline basis functions $\psi(\rho, \theta, z)$ and their gradients.

As stated in the comments in the code above, for efficiency reasons the evaluations are conducted regardless of the global indices of the 3D spline basis functions they refer to. Basically, given the local coordinates `localCoord`, the results from the evaluation

of every possible 3D basis function whose support includes that coordinates is given. A map to the global index is then necessary and is returned by function Lst. 6.6, which receives as input the cell `cellIdx` the local coordinates fall in.

```
//Number of 1D spline basis functions with support on a cell
idx_t lclLength = ORDER+1;

for( idx_t k = 0; k < lclLength; k++){
  //contribution of idx z to global and local idx:
  idx_t spIdxZ  = (cellIdx(2)+k) * splineNumTang_*splineNumRad_;
  idx_t lclIdxZ = k * lclLength*lclLength;

  for( idx_t j = 0; j < lclLength; j++){
    //contribution of idx theta to global and local idx (it his periodic in
     this direction, hence the % ):
    idx_t spIdxTheta  = ((cellIdx(1)+j) % splineNumTang_) * splineNumRad_;
    idx_t lclIdxTheta = j * lclLength;

    for( idx_t i = 0; i < lclLength; i++){
      //this is the index wrt the result of the evaluation of the spines
    with support on cellIdx
      idx_t lclIdx = i + lclIdxTheta + lclIdxZ;
      //this is the global index of the basis function
      idx_t glbIdex = ( cellIdx(0)+i ) + spIdxTheta + spIdxZ;
      //finally, store in the map
      mapl2gSplineIdx(lclIdx) = glbIdex;
    }
  }
}

```

Listing 6.6: Map from local splines evaluations (as returned from the function in Lst. 6.5) to global index.

In addition to these, `SplinesBasisFunc` provides various methods for the evaluation of quantities of interest: in particular, the approximated map $T_{\mathcal{V}_h}$ and its Jacobian, as described in (5.9), as well as the Gram matrix (5.10). These and more can be found in Lst. A.4, Lst. A.3 and Lst. A.5.

`ProblemsSolver` It is the internal member of `shapeOptimizationSolver` responsible for the solution of state and adjoint equations, for the storage of relevant temporary results, and for the evaluation of the derivative of the cost functional (3.15). In order to solve both (2.6) and (3.14), first the corresponding linear systems need to be built, as described in Sec. 5.1. For this part, we make use of various functions provided by

BETL2 for the discretization of matrix (5.2) and of various linear operators such as (2.4). The implementation can be found in Lst. 6.7.

```
// SETUP PULLBACK ==========================================================
//Will be used to retrieve M, used in the evaluation of bilinear form A = int(U)_Omega0 (gradU * M * gradV)
fem::SplinesPullback< VOL_GRID_FACTORY_T, SP_BASIS_FUNC_T > alpha( volGridFac, spBasis );
//This is the same, but it's used to recover the contributions from the cut-off function
fem::SpecialSplinesPullback< VOL_GRID_FACTORY_T, SP_BASIS_FUNC_T > alphaFake( volGridFac, spBasis );


// ADDITIONAL CONSTRAINTS==================================================
//Here the additional condition: int( U )_Omega = int( U * det(DTnu) )_Omega0 is considered
typedef fem::detDTnuFunction< SP_BASIS_FUNC_T >      functor_t;
typedef fem::AnalyticalGridFunction< VOL_GRID_FACTORY_T, functor_t > analyticalVolFunction_t;

const functor_t detDTFunc( spBasis );
const analyticalVolFunction_t detFunc ( volGridFac, detDTFunc );

//Proceed to the evaluation of the weights for the linear operator:
//  Instantiate the linear operators
volMassIntegral_t volIntegral;
volIntegral.compute( volDH.fespace(), detFunc );
//  Store the weights
auto& volIntegralWeights = volIntegral.matrix();
//  Same with the fake version (to recover contributions from the cut-off basis function)
volFakeMassIntegral_t volFakeIntegral;
volFakeIntegral.compute( volDH.fespace(), detFunc );
auto& volFakeIntegralWeights = volFakeIntegral.matrix();

// ASSEMBLE MATRIX A=======================================================
const idx_t sizeI = volDH.numDofs();
Ah.resize( sizeI + 2 , sizeI + 2 );

numeric_t volInt_reduced = 0;

//Matrix Ah is constructed in the following fashion:
//  First 0->volDH_.numDofs()-1 rows/cols: same as bilinear stiffness operator defined on classical fespace
//  Row/Col volDH_.numDofs(): contributions from cut-off basis function
//  Row/Col volDH_.numDofs()+1: lagrange constraint
stiffnessOperator_t Aii, Afake;
```

```
38 Aii.compute( volDH.fespace(), alpha );
39 Afake.compute( volDH.fespace(), alphaFake );
40 Aii.make_sparse( );
41 Afake.make_sparse( );
42
43 auto& Aii_h = Aii.matrix( );
44 auto& Afake_h = Afake.matrix( );
45
46 tripletList_t tripletList;
47 tripletList.reserve( Aii_h.nonZeros() + 2*Afake_h.nonZeros() + 2*
       volIntegralWeights.size() + 2 );
48 Ah.reserve( Aii_h.nonZeros() + 4*volIntegralWeights.size() );
49
50 //FIRST PART: Constributions from Aii ----------------------------
51 for (idx_t k=0; k<Aii_h.outerSize(); ++k){
52   for ( typename sparseMatrix_t::InnerIterator it(Aii_h,k); it; ++it ){
53     tripletList.push_back( triplet_t ( it.row(),
54                                        it.col(),
55                                        it.value() ) );
56   }
57 }
58
59 //SECOND PART: Contributions from Afake (cut-off function)------
60 vector_t Afake_reduced( sizeI );
61
62 //Reduce Afake along those rows that represent a node that lies on the
       discontinuity surface
63 Afake_reduced = Afake_h * nodesOnBorder;
64
65 //And add its contribution to the first additional row/col
66 for(idx_t k=0; k<Afake_reduced.size(); ++k){
67   if(Afake_reduced(k) != 0){
68     tripletList.push_back( triplet_t ( sizeI,
69                                        k,
70                                        Afake_reduced(k) ) );
71     tripletList.push_back( triplet_t ( k,
72                                        sizeI,
73                                        Afake_reduced(k) ) );
74
75   }
76 }
77 //Evaluate contribution of cut-off function tested with itself
78 numeric_t Afake_reduced_reduced = Afake_reduced.transpose() * nodesOnBorder
       ;
79
80 //And include that as well
81 tripletList.push_back( triplet_t ( sizeI,
82                                    sizeI,
```

```
                                    Afake_reduced_reduced ) );

//THIRD PART: additional constraint (lagrange)----------------------
for(idx_t k=0; k<volIntegralWeights.size(); ++k){
  tripletList.push_back( triplet_t ( sizeI + 1,
                                     k,
                                     volIntegralWeights(k) ) );
  tripletList.push_back( triplet_t ( k,
                                     sizeI + 1,
                                     volIntegralWeights(k) ) );
}

//do not forget cut-off basis function contribution to the volume integal
volInt_reduced = volFakeIntegralWeights.transpose() * nodesOnBorder;
tripletList.push_back( triplet_t ( sizeI + 1,
                                   sizeI,
                                   volInt_reduced ) );
tripletList.push_back( triplet_t ( sizeI,
                                   sizeI + 1,
                                   volInt_reduced ) );

//ASSEMBLE A ------------------------------------------------------
//Now that all the contributions have been considered, assemble the matrix
Ah.setFromTriplets(tripletList.begin(), tripletList.end());
Ah.makeCompressed();

//INFO STORAGE ----------------------------------------------------
//  In order to prevent re-evaluation of things, store info of interest:
AhAdj_  = Ah;

```

LISTING 6.7: Function to assemble state problem matrix (5.3) and store it for later use in the solution of the adjoint problem (3.14).

The code in Lst. 6.7 shows how the stiffness matrix (5.2) is fringed in order to recover (5.3), according to its definition in Sec. 5.1. Moreover, we can see how the system matrix assembled here is stored for future use in the solution of the adjoint equation. For solving the systems themselves, a Sparse LU solver [24] from the library Eigen [23] is used. This class also takes care of finding a proper representative of the gradient of $\mathcal{J}$ (3.15), following the procedure in Sec. 4.3. The corresponding code is reported in Lst. 6.8.

```
solver_t solver;
const auto& H = spBasis.getGramMatrix();
solver.compute( H );

const idx_t numSp = spBasis.getNumBasisFunc()/3;
```

```
6
7  dJX = shapeDerivative_.segment(      0, numSp);
8  dJY = shapeDerivative_.segment(  numSp, numSp);
9  dJZ = shapeDerivative_.segment(2*numSp, numSp);
10
11 update.col(0) = solver.solve( dJX );
12 update.col(1) = solver.solve( dJY );
13 update.col(2) = solver.solve( dJZ );
14
15 const numeric_t norm = dJX.dot( update.col(0) )
16                       +dJY.dot( update.col(1) )
17                       +dJZ.dot( update.col(2) );
18
19 //Resize gradient by step size sigma
20 const numeric_t sigma = 0.3;
21 update *= - sigma /sqrt( norm );
22
23
```

LISTING 6.8: Implementation of (5.12) for the computation of the H^1 representative of $\mathcal{J}'$.

Such a structure of the code is only possible since the same spline basis functions ψ_i are used to describe every component of the vector field $\mathcal{V}_h$, as previously noticed in Sec. 5.2. Regarding the effect of the approximated operator $\langle \mathcal{J}_h'(\mathcal{V}_h), \psi_i \rangle$ (stored in `shapeDerivative_` in Lst. 6.8), it is computed by using methods of its internal members belonging to classes `CostFunctionalIntegrator` and `StateEqDerivativeEvaluator`, presented next.

`CostFunctionalIntegrator` The main purpose of this class is to retrieve the approximated values of the cost functional (2.1) and its partial derivatives (3.11), (3.10). Both of these derivatives are necessary for the evaluation of the gradient of $\mathcal{J}$: the first one contributes directly to the formula (3.15), while the latter appears in the right-hand side of the adjoint equation (3.14). The function responsible for this is by far the most expensive from the computational point of view: in fact, (2.1), (3.11) and (3.10) are defined as an integral on the surface $\partial\mathcal{D}$ of a quantity that depends on the magnetic field $\mathbf{H}$; however, the magnetic field itself is expressed as an integral on the whole domain Ω_0. Hence, it is necessary to perform a double integral, approximated using a Gaussian quadrature rule [28]. This basically translates into a double nested `for` loop on two sets of quadrature points $\mathbf{x_q} \in \partial\mathcal{D}$ and $\mathbf{y_q} \in \Omega_0$ where the integrand function must be evaluated. Since they contain many terms in common, in order to spare computational effort, the integrands of each of the three quantities (2.1), (3.10) and (3.11) are evaluated

together. Some pseudo-code describing how this function works is described in Alg. 2, while its actual implementation is reported in Lst. A.6.

Algorithm 2: Main function of `CostFunctionalIntegrator`

Initialize to 0 the vector for storing values of $\mathbf{H}(u;\mathbf{x_q})$, $\forall\mathbf{x_q}$;
Initialize to 0 the matrix for storing values of $\mathbf{H}(\phi_i;\mathbf{x_q})$, $\forall i$ and $\forall\mathbf{x_q}$;
Initialize to 0 the matrix for storing values of $\int_{\Omega_0}\delta_{\psi_i}(\mathbf{A}(u,\mathcal{V})\times\mathbf{B}(\mathcal{V};\mathbf{x_q}))$, $\forall i$ and $\forall\mathbf{x_q}$;
Initialize the vector containing evaluations of $\mathbf{p}(\mathbf{x_q})$, $\forall\mathbf{x_q}$;
for *each quadrature point* $\mathbf{y_q}\in\Omega_0$ **do**
    Evaluate $\mathbf{A}(u,\mathcal{V};\mathbf{y_q})$ and multiply it by the Gaussian weight $w_{\mathbf{y_q}}$;
    for *each FE basis function* ϕ_i *with support on* $\mathbf{y_q}$ **do**
        Evaluate $\mathbf{A}(\phi_i,\mathcal{V};\mathbf{y_q})$ and multiply it by the Gaussian weight $w_{\mathbf{y_q}}$;
        Store results in temporary matrix;
    for *each spline basis function* ψ_i *with support on* $\mathbf{y_q}$ **do**
        Evaluate $\delta_{\psi_i}\mathbf{A}(u,\mathcal{V};\mathbf{y_q})$ and multiply it by the Gaussian weight $w_{\mathbf{y_q}}$;
        Evaluate $\nabla\psi_i(\mathbf{y_q})$;
        Store results in temporary matrix;
    Evaluate $T_{\mathcal{V}}(\mathbf{y_q})$;
    for *each quadrature point* $\mathbf{x_q}\in\partial\mathcal{D}$ **do**
        Evaluate $\mathbf{B}(\mathcal{V};\mathbf{x_q},\mathbf{y_q})$ (using the pre-evaluated $T_{\mathcal{V}}(\mathbf{y_q})$);
        Evaluate $\delta_{\psi_i}\mathbf{B}(\mathcal{V};\mathbf{x_q},\mathbf{y_q})$ (using the pre-evaluated $\nabla\psi_i(\mathbf{y_q})$ and $T_{\mathcal{V}}(\mathbf{y_q})$);
        Evaluate $\mathbf{A}(u,\mathcal{V};\mathbf{y_q})\times\mathbf{B}(\mathcal{V};\mathbf{x_q},\mathbf{y_q})$ (using the pre-evaluated $\mathbf{A}(u,\mathcal{V};\mathbf{y_q})$), and add it to the corresponding term in $\mathbf{H}(u;\mathbf{x_q})$;
        Evaluate $\mathbf{A}(\phi_i,\mathcal{V};\mathbf{y_q})\times\mathbf{B}(\mathcal{V};\mathbf{x_q},\mathbf{y_q})$ (using the pre-evaluated $\mathbf{A}(\phi_i,\mathcal{V};\mathbf{y_q})$), and add it to the corresponding terms in $\mathbf{H}(\phi_i;\mathbf{x_q})$;
        Evaluate $\delta_{\psi_i}(\mathbf{A}(u,\mathcal{V})\times\mathbf{B}(\mathcal{V};\mathbf{x_q}))$ (using the pre-evaluated $\delta_{\psi_i}\mathbf{A}(u,\mathcal{V};\mathbf{y_q})$ and $\mathbf{A}(u,\mathcal{V};\mathbf{y_q})$) and add it to the corresponding terms in $\int_{\Omega_0}\delta_{\psi_i}(\mathbf{A}(u,\mathcal{V})\times\mathbf{B}(\mathcal{V};\mathbf{x_q}))$
Evaluate $\mathbf{H_t}(u;\mathbf{x_q})-\mathbf{p}(\mathbf{x_q})$;
Evaluate $(\mathbf{H_t}(u;\mathbf{x_q})-\mathbf{p}(\mathbf{x_q}))^2$;
Evaluate $(\mathbf{H_t}(u;\mathbf{x_q})-\mathbf{p}(\mathbf{x_q}))\cdot\mathbf{H_t}(\phi_i;\mathbf{x_q})$;
Evaluate $(\mathbf{H_t}(u;\mathbf{x_q})-\mathbf{p}(\mathbf{x_q}))\cdot\int_{\Omega_0}\delta_{\psi_i}(\mathbf{A}(u,\mathcal{V})\times\mathbf{B}(\mathcal{V};\mathbf{x_q}))$;
Perform reduction of the quantities above using the Gaussian weights $w_{\mathbf{x_q}}$ to integrate on $\partial\mathcal{D}$ and recover (2.1), (3.11) and (3.10);

The reason behind the intricacy of Alg. 2 lies in the effort done in order to increase efficiency. The two `for` loops on $\mathbf{x_q}$ and $\mathbf{y_q}$ have been switched to take advantage of the fact that many terms in the integrands depend purely on variable $\mathbf{y}\in\Omega_0$, and can then be pre-computed: in fact, only $\mathbf{B}$ defined in (3.7) and its derivative show dependency on $\mathbf{x}$. Furthermore, all the terms that require splines evaluation are computationally very demanding[1], and are functions solely of $\mathbf{y}$. On the other hand, this results in the need to store temporary matrices of relevant size. This and other trade-offs between memory consumption and computational effort are discussed more in detail in Sec. 6.2.

[1] This is due both to the higher order used for the spline basis functions and the additional complexity of the mappings (5.7) and (5.6) with respect to those involved in the evaluation of their FE counterparts.

An additional improvement in the code efficiency can be achieved exploiting the peculiarities of the discretization chosen for the vector field $\mathcal{V}_h$, as previously stated in Sec. 5.2. For example, considering the evaluation of

$$\begin{aligned} \delta_\psi \mathbf{A} &= \tfrac{\mu_0}{4\pi} \det(\mathbf{D}T_\mathcal{V})[\mathrm{tr}(\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}\psi)I - \mathbf{D}T_\mathcal{V}^{-T}\mathbf{D}\psi^T]\mathbf{D}T_\mathcal{V}^{-T}\nabla u \\ &= \mathrm{tr}(\mathbf{D}T_\mathcal{V}^{-1}\mathbf{D}\psi)\mathbf{A} - \mathbf{D}T_\mathcal{V}^{-T}\mathbf{D}\psi^T\mathbf{A}, \end{aligned} \tag{6.1}$$

we can notice that, since the same ψ are used to describe all three components of the vector field $\mathcal{V}_h$, we have

$$(\mathbf{D}\psi)_d = \mathbf{e}_d \nabla\psi(\mathbf{x})^T \implies$$

$$(\mathbf{D}\psi)_x = \begin{bmatrix} \nabla\psi(\mathbf{x})^T \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, (\mathbf{D}\psi)_y = \begin{bmatrix} \mathbf{0} \\ \nabla\psi(\mathbf{x})^T \\ \mathbf{0} \end{bmatrix}, (\mathbf{D}\psi)_z = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \nabla\psi(\mathbf{x})^T \end{bmatrix},$$

where $(\mathbf{D}\psi)_d$ denotes the Jacobian of ψ when used as a basis function to describe the d-component of $\mathcal{V}_h$. From this, we get that $\mathbf{D}T_\mathcal{V}^{-T}\mathbf{D}\psi^T$ is just

$$\begin{aligned} \mathbf{D}T_\mathcal{V}^{-T}(\mathbf{D}\psi)_d^T &= \mathbf{D}T_\mathcal{V}^{-T}\nabla\psi(\mathbf{x})\mathbf{e}_d^T \implies \\ \mathbf{D}T_\mathcal{V}^{-T}(\mathbf{D}\psi)_x^T &= \left[\begin{array}{ccc} \mathbf{D}T_\mathcal{V}^{-T}\nabla\psi(\mathbf{x}) & \mathbf{0} & \mathbf{0} \end{array}\right], \\ \mathbf{D}T_\mathcal{V}^{-T}(\mathbf{D}\psi)_y^T &= \left[\begin{array}{ccc} \mathbf{0} & \mathbf{D}T_\mathcal{V}^{-T}\nabla\psi(\mathbf{x}) & \mathbf{0} \end{array}\right], \\ \mathbf{D}T_\mathcal{V}^{-T}(\mathbf{D}\psi)_z^T &= \left[\begin{array}{ccc} \mathbf{0} & \mathbf{0} & \mathbf{D}T_\mathcal{V}^{-T}\nabla\psi(\mathbf{x}) \end{array}\right]. \end{aligned}$$

If we consider the properties of the trace operator:

$$\mathrm{tr}(\mathbf{D}T_\mathcal{V}^{-1}(\mathbf{D}\psi)_d) = \mathrm{tr}((\mathbf{D}\psi)_d\mathbf{D}T_\mathcal{V}^{-1}) = \mathrm{tr}(((\mathbf{D}\psi)_d\mathbf{D}T_\mathcal{V}^{-1})^T) = \mathrm{tr}(\mathbf{D}T_\mathcal{V}^{-T}(\mathbf{D}\psi)_d^T),$$

we can see how this term is given just by the d-component of the vector $\mathbf{D}T_\mathcal{V}^{-T}\mathbf{D}\psi^T$. The first term in (6.1) is then recovered just by multiplying $\mathbf{A}$ by the correct component of $\mathbf{D}T_\mathcal{V}^{-T}\nabla\psi$. Conversely, the second term is recovered by multiplying $\mathbf{D}T_\mathcal{V}^{-T}\nabla\psi$ by the correct component of $\mathbf{A}$. The corresponding part of the code that performs the evaluation described above is reported in Lst. 6.9.

```
1 //First retrieve trace
2 const rowMatrix3SP_t traceXYZ = DTnuInvT * gradPsi;
3
4 //To get first term, multiply A by the correct element in trace
```

```
5  const rowMatrix3SP_t firstX = A * traceXYZ.row(0);
6  const rowMatrix3SP_t firstY = A * traceXYZ.row(1);
7  const rowMatrix3SP_t firstZ = A * traceXYZ.row(2);
8
9  //To get second term, multiply trace by the correct element in A
10 const rowMatrix3SP_t secondX = traceXYZ * A(0);
11 const rowMatrix3SP_t secondY = traceXYZ * A(1);
12 const rowMatrix3SP_t secondZ = traceXYZ * A(2);
13
14 //Sum all together and store
15 dAnu.template block< 3, numSpPerPoint_ >( 0, 0 ) = firstX - secondX;
16 dAnu.template block< 3, numSpPerPoint_ >( 0, numSpPerPoint_ ) = firstY -
       secondY;
17 dAnu.template block< 3, numSpPerPoint_ >( 0, 2*numSpPerPoint_ ) = firstZ -
       secondZ;
18
```

LISTING 6.9: Function to retrieve derivative of term $\mathbf{A}$ (6.1).

Here, `gradPsi` is a $3 \times (N+1)^3$ matrix containing all evaluations of gradients of spline basis functions with support on the point considered[2], while `DTnuInvT` is $\mathbf{D}T_{\mathcal{V}}^{-T}$. The result `dAnu` is a vector containing the values of (6.1) for each of the spline basis functions, considered as contributions to the x- (first `numSpPerPoint_` elements), y- (second `numSpPerPoint_` elements) and z-components (last `numSpPerPoint_` elements) of $\mathcal{V}_h$.

`StateEqDerivativeEvaluator` To complete the evaluation of the gradient of $\mathcal{J}$, according to (3.15), three additional terms need to be computed and added to (3.11). This class is responsible for this. The three terms are

$$\begin{aligned} & \int_{\Omega_0 \setminus \Sigma_0} \nabla u \, \delta_\psi \mathbf{M}_{\mathcal{V}} \nabla z \, d\mathbf{x} \\ + \; & \mu \int_{\Omega_0} \det(\mathbf{D}T_{\mathcal{V}}) \mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}\psi) u \, d\mathbf{x} \\ + \; & \lambda \int_{\Omega_0} \det(\mathbf{D}T_{\mathcal{V}}) \mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}\psi) z \, d\mathbf{x}, \end{aligned} \tag{6.2}$$

and again they must be calculated for every spline basis function ψ. Here too some simplification can be applied, similar to those involving the evaluation of (3.11), in particular for the term $\delta_\psi \mathbf{M}_{\mathcal{V}}$ defined in (3.4). The integrand of the first integral in (6.2) can be split into three parts:

$$\begin{aligned} \nabla u \, \delta_\psi \mathbf{M}_{\mathcal{V}} \nabla z \;=\; & \nabla u^T \mathbf{D}T_{\mathcal{V}}^{-1} \mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}\psi) \det(\mathbf{D}T_{\mathcal{V}}) \mathbf{D}T_{\mathcal{V}}^{-T} \nabla z \\ - \; & \nabla u^T \mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}\psi \mathbf{D}T_{\mathcal{V}}^{-1} \det(\mathbf{D}T_{\mathcal{V}}) \mathbf{D}T_{\mathcal{V}}^{-T} \nabla z \\ - \; & \nabla u^T \mathbf{D}T_{\mathcal{V}}^{-1} \mathbf{D}T_{\mathcal{V}}^{-T} \mathbf{D}\psi^T \det(\mathbf{D}T_{\mathcal{V}}) \mathbf{D}T_{\mathcal{V}}^{-T} \nabla z. \end{aligned} \tag{6.3}$$

[2] As we said in Sec. 5.2, for 3D splines of order N, every point is contained in the support of $(N+1)^3$ different spline basis functions.

The left and right factors of (6.3) are common to each of the terms above and can then be precomputed. Let $\mathbf{a} = \mathbf{D}T_{\mathcal{V}}^{-T}\nabla u$ and $\mathbf{b} = \det(\mathbf{D}T_{\mathcal{V}})\mathbf{D}T_{\mathcal{V}}^{-T}\nabla z$: we can rewrite the integrand as

$$\begin{aligned}\nabla u\,\delta_\psi \mathbf{M}_{\mathcal{V}}\nabla z &= \mathbf{a}^T\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\psi)\mathbf{b} - \mathbf{a}^T\mathbf{D}\psi\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{b} - \mathbf{a}^T\mathbf{D}T_{\mathcal{V}}^{-T}\mathbf{D}\psi^T\mathbf{b}.\\ &= \mathbf{a}^T\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\psi)\mathbf{b} - \mathbf{b}^T\mathbf{D}T_{\mathcal{V}}^{-T}\mathbf{D}\psi^T\mathbf{a} - \mathbf{a}^T\mathbf{D}T_{\mathcal{V}}^{-T}\mathbf{D}\psi^T\mathbf{b}.\end{aligned}$$

We have already discussed how $\mathrm{tr}(\mathbf{D}T_{\mathcal{V}}^{-1}\mathbf{D}\psi)$ can be simplified: as a consequence, the first term reduces to multiply the trace by the result of the scalar product between $\mathbf{a}$ and $\mathbf{b}$. The last two terms are very similar, and given the particular form of $\mathbf{D}T_{\mathcal{V}}^{-T}\mathbf{D}\psi^T$, they both reduce to perform first a scalar product involving the vector on the left ($\mathbf{a}^T\cdot\nabla\psi$ or $\mathbf{b}^T\cdot\nabla\psi$) and then multiply the result by the correct element of the vector on the right (depending on whether we are considering contribution of ψ to x-, y- or z-components of $\mathcal{V}_h$). The actual code responsible for this is presented in Lst. 6.10

```
1  //Evaluate trace
2  const rowMatrix3SP_t traceXYZ = DTnuInvT * gradPsi;
3
4  //multiply by a*b to get first term
5  const rowMatrix3SP_t firstXYZ = traceXYZ * DTinvTdetDTgradZ.dot( gradUDTinv
       );
6
7  //To get second and third terms, first scalar product with vector on the
       left
8  const rowVectorSP_t temp1 = gradUDTinv.transpose() * traceXYZ;
9  const rowVectorSP_t temp2 = DTinvTdetDTgradZ.transpose() * traceXYZ;
10 //then multiply by element of vector on the right
11 const rowMatrix3SP_t secondXYZ = DTinvTdetDTgradZ * temp1;
12 const rowMatrix3SP_t thirdXYZ = gradUDTinv * temp2;
13
14 //store contributions for first integrand
15 const rowMatrix3SP_t allTogether = firstXYZ - secondXYZ - thirdXYZ;
16 dStiff.template segment< numSpPerPoint_ >(0) = allTogether.row(0);
17 dStiff.template segment< numSpPerPoint_ >(numSpPerPoint_) = allTogether.row
       (1);
18 dStiff.template segment< numSpPerPoint_ >(2*numSpPerPoint_) = allTogether.
       row(2);
19
20 const rowMatrix3SP_t allTogetherU = traceXYZ * detDTUw;
21 dConstrU.template segment< numSpPerPoint_ >(0) = allTogetherU.row( 0 );
22 dConstrU.template segment< numSpPerPoint_ >(numSpPerPoint_) = allTogetherU.
       row(1);
23 dConstrU.template segment< numSpPerPoint_ >( 2*numSpPerPoint_ ) =
       allTogetherU.row(3);
24
25 const rowMatrix3SP_t allTogetherZ = traceXYZ * detDTZw;
```

```
26 dConstrZ.template segment< numSpPerPoint_ >(0) = allTogetherZ.row( 0 );
27 dConstrZ.template segment< numSpPerPoint_ >(numSpPerPoint_) = allTogetherZ.
       row(1);
28 dConstrZ.template segment< numSpPerPoint_ >( 2*numSpPerPoint_ ) =
       allTogetherZ.row(3);
29
```

LISTING 6.10: Function to recover the terms in (6.2).

As the names suggest, `gradUDTinv` is $\mathbf{a} = \mathbf{D}T_{\mathcal{V}}^{-T} \nabla u$, while `DTinvTdetDTgradZ` is $\mathbf{b} = \det(\mathbf{D}T_{\mathcal{V}})\mathbf{D}T_{\mathcal{V}}^{-T} \nabla z$; `dStiff` contains evaluations of the first integrand, while `dConstrU` and `dConstrZ` refer to the second and third ones in (6.2).

6.2 Memory/computation trade-off

In this section, the trade-off between computational load and memory usage regarding the most important functions described in Sec. 6.1 is discussed. In particular, we focus our attention on Alg. 2 and on the functions responsible for the splines evaluations.

For the way Alg. 2 is implemented, it needs to store matrices of size $N_{\mathbf{x_q}} \times n$ and $N_{\mathbf{x_q}} \times m$ for the partial derivatives (3.10) and (3.11) respectively, being $N_{\mathbf{x_q}}$ the number of quadrature points on the surface $\partial\mathcal{D}$, n the number of FE basis functions ϕ, m the number of spline basis functions ψ. These dimensions grow linearly in the number of nodes composing the meshes defined on $\partial\mathcal{D}$ and on Ω_0 or the splines grid. Furthermore, the matrices themselves are dense, because of the nature of the non-local operator that appear in $\mathcal{J}$[3]. This by far represents the largest bottleneck of the code with regards to memory consumption. The other matrices of considerable size that need to be assembled are in fact (5.3) and (5.10). Although their dimensions depend on the number of FE and spline basis functions respectively, still they present a sparse pattern [26, Chap. 4], so this dependency is only linear. This is due to the limited support of both the FE and spline basis functions. A less memory-consuming but more computationally demanding alternative would consist in performing the double `for` loop described in Alg. 2 in the classical manner: the external one sweeping through the quadrature points $\mathbf{x_q} \in \partial\mathcal{D}$ and the internal through the $\mathbf{y_q} \in \Omega_0$. This would allow to perform the outer integral as the loop proceeds, without the need to gather every single contribution first, hence reducing the storage requirement to a simple vector of size n for (3.10) or m for (3.11). On the other hand, all the simplifications described in Sec. 6.1 would no longer be feasible, ending up in a considerable increase in the computational cost.

[3] We remind that due to Biot-Savart law, the magnetic field in one quadrature point $\mathbf{x_q} \in \partial\mathcal{D}$ depends on the results of an integral defined on the whole domain Ω_0. Hence, every FE and spline basis function contributes actively to the final evaluation of $\mathbf{H}$ and its derivatives.

Chapter 7

Validation

Since no analytical solution is available for benchmarking, the following tests have been conducted in order to check the correctness of the implementation.

7.1 Solution of parametric state equation

As a first control routine, the code regarding the construction of the linear system representing the approximated mapped state equation (2.6) has been verified, together with its solution procedure. The verification routine has been conducted as follows:

- Consider a discretized reference domain Ω_0.
- Interpolate an arbitrary analytic vector field using spline basis functions, thus recovering a discretized map $T_{\mathcal{V}_h}$.
- Assemble matrix (5.3), using FE basis functions defined on Ω_0, and by considering the effect of the map $T_{\mathcal{V}_h}$.
- Solve the linear system associated to (2.6), and recover u_h.
- Modify the mesh according to the map $\Omega = T_{\mathcal{V}_h}(\Omega_0)$.
- Retrieve the canonical isotropic stiffness matrix $(\hat{\mathbf{A}})_{i,j} = \int_{\Omega} \hat{\nabla}\hat{\phi}_j \hat{\nabla}\hat{\phi}_i \, d\hat{\mathbf{x}}$, as well as the contributions $\int_{\Omega} \hat{\phi}_i \, d\hat{\mathbf{x}}$, using FE basis functions (cut-off included) defined on the mapped mesh representing Ω.
- Assemble and solve the linear system associated to (2.5) and (2.4) to recover $\hat{u_h}$.
- Compare the values of some functionals that depend on u evaluated directly, with those of the same functionals defined through the mapping $T_{\mathcal{V}_h}$.

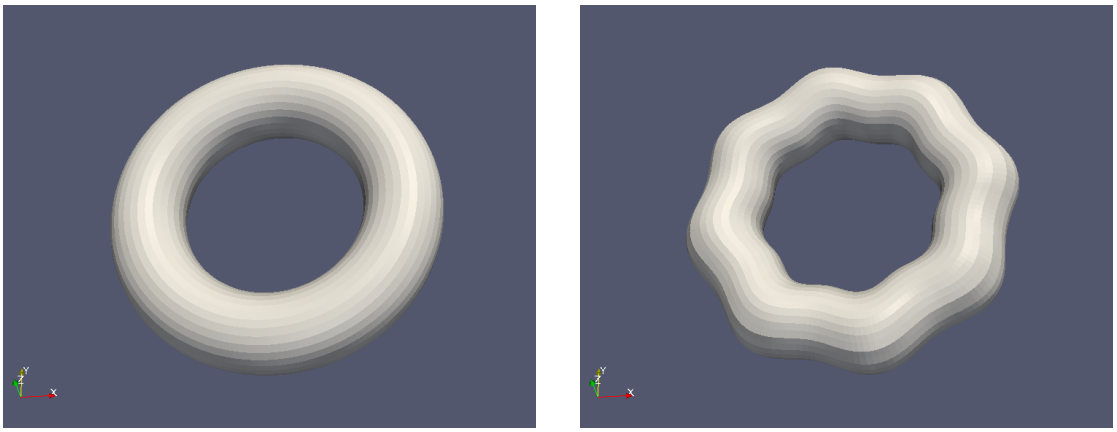

FIGURE 7.1: Comparison between reference domain Ω_0 (on the left) and mapped domain $\Omega = T_{\mathcal{V}}(\Omega_0)$, with $T_{\mathcal{V}}$ obtained interpolating the function (7.1) using splines.

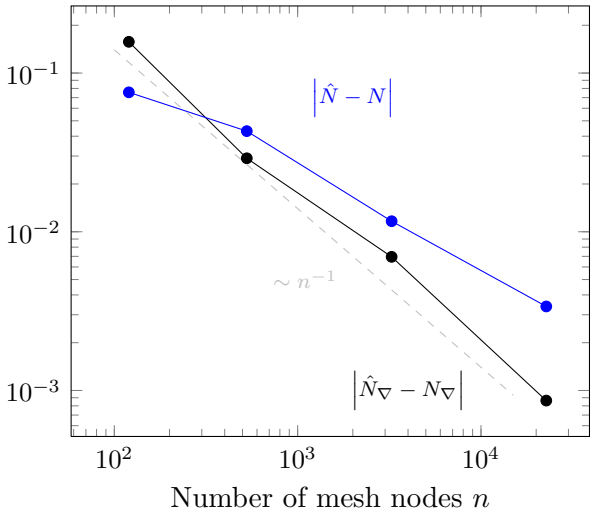

FIGURE 7.2: Logarithmic plot of the absolute values of the differences between $\hat{N} = \sqrt{\int_\Omega \hat{u}^2 \, d\hat{\mathbf{x}}}$ and $N = \sqrt{\int_{\Omega_0} \det \mathbf{D}T_{\mathcal{V}} \, u^2 \, d\mathbf{x}}$ (blue) and between $\hat{N}_\nabla = \sqrt{\int_\Omega \hat{\nabla}\hat{u} \cdot \hat{\nabla}\hat{u} \, d\hat{\mathbf{x}}}$ and $N_\nabla = \sqrt{\int_{\Omega_0} \nabla u \, \mathbf{M}_{\mathcal{V}} \, \nabla u \, d\mathbf{x}}$ (black), for meshes with $n = 120, 528, 3256, 22704$ number of nodes.

The functionals whose values are compared are the norms of $\hat{u}$ and of its gradient, respectively: $\|\hat{u}\|^2 = \int_\Omega \hat{u}^2 \, d\hat{\mathbf{x}} = \int_{\Omega_0} \det(\mathbf{D}T_{\mathcal{V}}) \, u^2 \, d\mathbf{x}$ and $\|\nabla \hat{u}\|^2 = \int_\Omega \hat{\nabla}\hat{u} \cdot \hat{\nabla}\hat{u} \, d\hat{\mathbf{x}} = \int_{\Omega_0} \nabla u \, \mathbf{M}_{\mathcal{V}} \, \nabla u \, d\mathbf{x}$. Different levels of mesh refinements on Ω_0 (and consequently on $\Omega = T_{\mathcal{V}}(\Omega_0)$) have been employed, in order to check for convergence. The analytical function interpolated via splines is

$$f(\mathbf{x}) = \left(c_1 \sin\left(c_2 \tan^{-1}\left(\frac{\mathbf{x}_2}{\mathbf{x}_1} \right) \right) + c_3 \right) \mathbf{x}, \tag{7.1}$$

and its effect once applied to the original domain Ω_0 can be seen in Fig. 7.1. The convergence results for four different levels of mesh refinement are reported in Fig. 7.2.

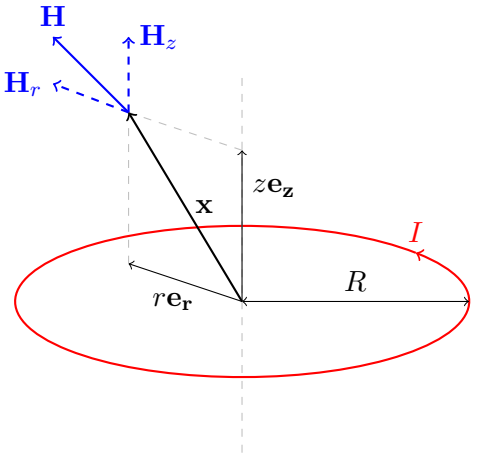

FIGURE 7.3: Sketch of the decomposition of $\mathbf{H}$ into z- and radial components. Thanks to the high symmetry of the problem, an analytical expression for the solution is available (7.2), and it depends only on coordinates z and $r = \sqrt{x^2 + y^2}$. Here, $\mathbf{x} = (x, y, z)^T$.

The results from Fig. 7.2 show how the difference between the two alternative evaluations for the norm of u, $\sqrt{\int_\Omega \hat{u}^2 \, d\hat{\mathbf{x}}}$ and $\sqrt{\int_{\Omega_0} \det \mathbf{D}T_\mathcal{V} \, u^2 \, d\mathbf{x}}$, as well as those for the norm of ∇u, $\sqrt{\int_\Omega \hat{\nabla}\hat{u} \cdot \hat{\nabla}\hat{u} \, d\hat{\mathbf{x}}}$ and $\sqrt{\int_{\Omega_0} \nabla u \, \mathbf{M}_\mathcal{V} \, \nabla u \, d\mathbf{x}}$, tend to zero, as the mesh is refined. Since the available routines from BETL2 are given for exact, this represents a satisfactory hint for the correctness of the implementation of this part of the code.

7.2 Evaluation of magnetic field

Another test has been conducted to verify the functions responsible for the magnetic field evaluation. In the case of a circular current loop, a non-trivial analytical formula involving elliptic integrals of first and second kind [29] is available to compute the magnetic field also off the central axis[1] [30]:

$$\begin{aligned} \mathbf{H}_z &= H_0 \frac{1}{\pi\sqrt{Q}} \left(E(k) \frac{1-\alpha^2-\beta^2}{Q-4\alpha} + K(k) \right) \mathbf{e_z} \\ \mathbf{H}_r &= H_0 \frac{\gamma}{\pi\sqrt{Q}} \left(E(k) \frac{1+\alpha^2+\beta^2}{Q-4\alpha} - K(k) \right) \mathbf{e_r}, \end{aligned} \tag{7.2}$$

being $\mathbf{H}_z$ and $\mathbf{H}_r$ the z- and radial component of $\mathbf{H}$ respectively, which give $\mathbf{H} = \mathbf{H}_z + \mathbf{H}_r$. See Fig. 7.3 for a sketch of these components. In (7.2), H_0 is the magnetic field intensity at the center of the loop of radius R and with circulating current I, $H_0 = \frac{I\mu_0}{2R}$; we then have: $\alpha = r/R$, $\beta = z/R$, $\gamma = z/r$, $Q = (1+\alpha)^2 + \beta^2$, $k = \sqrt{4\alpha/Q}$, with r the radial component, $r = \sqrt{x^2 + y^2}$; finally, $K(k)$ and $E(k)$ are the elliptic integrals of first and

[1] From a simple application of Biot-Savart law, the magnetic field along the central axis can be easily calculated as $\mathbf{H}(z) = \frac{\mu_0}{4\pi} \frac{2\pi R^2 I}{\sqrt{(z^2+R^2)^3}} \mathbf{e_z}$.

second kind, whose definition is in (7.3).

$$\begin{aligned} K(k) &= \int_0^{\pi/2} \frac{1}{\sqrt{1-k\sin^2(\theta)}}\, d\theta \\ E(k) &= \int_0^{\pi/2} \sqrt{1-k\sin^2(\theta)}\, d\theta. \end{aligned} \tag{7.3}$$

The test set-up is as follows:

- Consider a very thin toroidal mesh (radius of cross section $\lll$ external radius).
- Solve state equation (2.6) to recover u_h.
- Evaluate magnetic field $\mathbf{H}$ on the surface of an internal object $\partial\mathcal{D}$ using own implementation of Biot-Savart law.
- Evaluate magnetic field $\mathbf{H}_{ref}$ on the surface of an internal object $\partial\mathcal{D}$ using (7.2).
- Evaluate $\|\mathbf{H}-\mathbf{H}_{ref}\| = \sqrt{\int_{\partial\mathcal{D}} (\mathbf{H}(\mathbf{x})-\mathbf{H}_{ref}(\mathbf{x}))^2\, d\mathbf{x}}$ and thus compare $\mathbf{H}$ with $\mathbf{H}_{ref}$.

The code used for the implementation of (7.2) has been written by Oded Stein for his master thesis [31]. The evaluation has been conducted for different levels of refinement of the meshes on the toroidal domain Ω_0 and on $\partial\mathcal{D}$, as well as for different sizes of the torus itself: a representation of this is given in Fig. 7.4.

The convergence results reported on Fig. 7.5 show how, after having reached a certain number of nodes, further refinement of the meshes has no longer an impact on the convergence of $\|\mathbf{H}-\mathbf{H}_{ref}\|$. This is because $\mathbf{H}_{ref}$ is given for a line current, while the torus has an inner radius larger than 0. The effect of reducing the inner radius, however, is significant in diminishing $\|\mathbf{H}-\mathbf{H}_{ref}\|$. All these considerations again make us confident about the reliability of this particular routine.

7.3 Shape derivative evaluation

As a final validation, the evaluation of the gradient of the cost functional (3.6) is tested, in the following manner:

- Start from a reference shape $\Omega = T_{\mathcal{V}}(\Omega_0) = \Omega_0 + \mathcal{V}(\Omega_0)$.
- Evaluate cost functional $\mathcal{J}(\mathcal{V})$.
- Evaluate derivative of cost functional along a direction $\tilde{\mathcal{V}}$, $\left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle$.
- Update the map $T_\nu = \mathcal{I} + \mathcal{V} + h\tilde{\mathcal{V}}$.

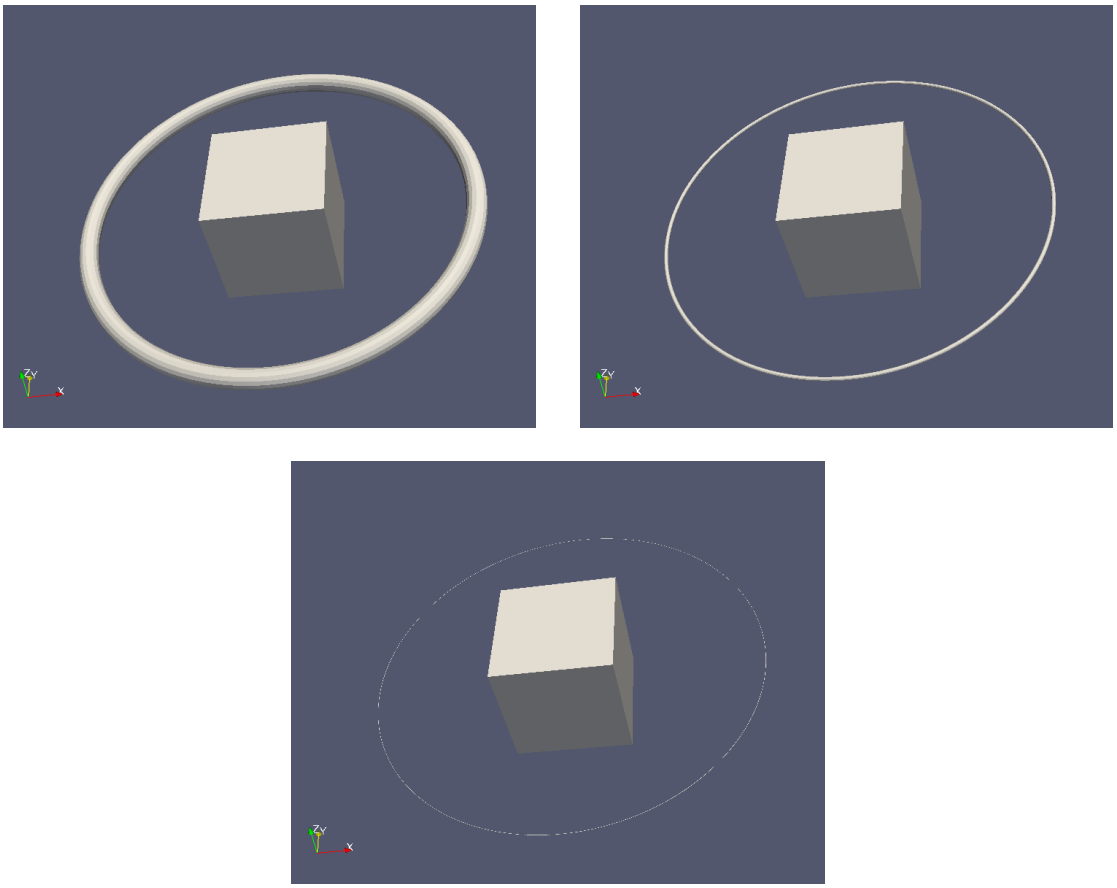

FIGURE 7.4: Representations of the three different sizes of the torus Ω_0 used in the test described in Sec. 7.2, as well as the internal object $\partial\mathcal{D}$. The external radius of the torus is always 1, while the radius of the torus cross section (internal radius) is 0.05 (top left), 0.01 (top right) and 0.001 (bottom). Object $\partial\mathcal{D}$ is a cube of side 0.4.

- Evaluate cost functional in new position $\mathcal{J}(\mathcal{V} + h\tilde{\mathcal{V}})$.
- Compare the values of $\frac{\mathcal{J}(\mathcal{V}+h\tilde{\mathcal{V}})-\mathcal{J}(\mathcal{V})}{h}$ and $\left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle$ as $h \to 0$.

Also the results of this experiment have been checked for various refinement levels of the mesh on Ω_0, and for different directions $\tilde{\mathcal{V}}$. One of these is reported in Fig. 7.6. It is clear that, as the step size h reduces, the value of the derivative approximated with finite-differences approaches the one computed using our own routine for the evaluation of the shape derivative as $\sim h^{-1}$.

The three validations presented above, together with many others of minor relevance conducted throughout the writing of the code, allow us to be confident with regards to the correctness of the implementation, and to proceed with the collection of some results from actual experiments, which are presented next.

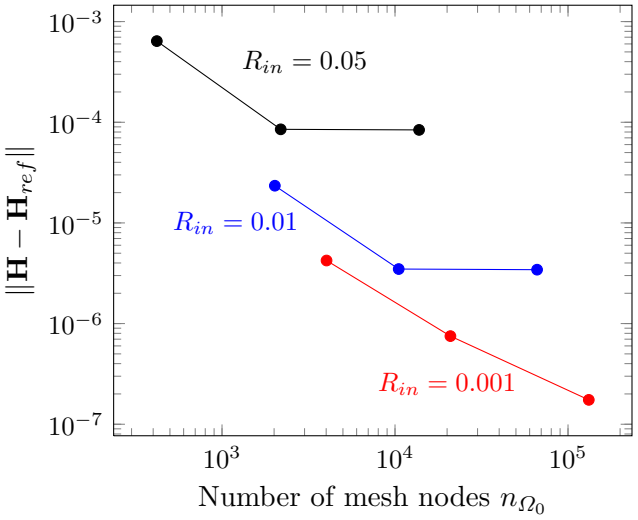

FIGURE 7.5: Logarithmic plot of the norm of the error $\|\mathbf{H} - \mathbf{H}_{ref}\|$, for meshes defined on Ω_0 with number of nodes $n_{\Omega_0} = 420, 2184, 13776$ (black), $n_{\Omega_0} = 2020, 10504, 66256$ (blue), $n_{\Omega_0} = 4020, 20904, 131856$ (red). External radius of the torus $R = 1$ for all cases; internal radius $R_{in} = 0.05$ (black), $R_{in} = 0.01$ (blue), $R_{in} = 0.001$ (red). Tests have been carried out for meshes defined on $\partial\mathcal{D}$ with number of nodes $n_{\partial\mathcal{D}} = 42, 320, 642$, but the difference in the results for this set of meshes is minimal. It is clear how refining meshes after a certain level has no effect, since the analytical solution refers to a line current. A great impact can instead be seen on the convergence when the internal radius of the torus is reduced.

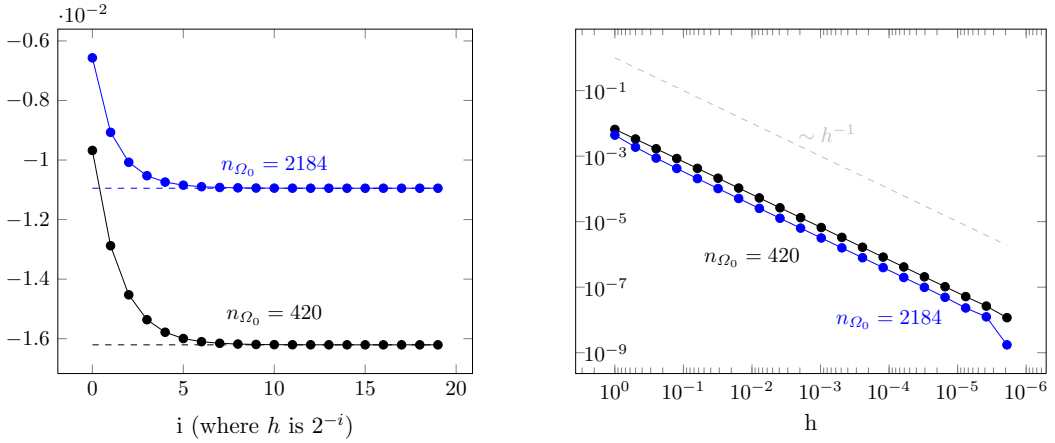

FIGURE 7.6: Plots showing the convergence of $(\mathcal{J}(\mathcal{V} + h\tilde{\mathcal{V}}) - \mathcal{J}(\mathcal{V}))/h$ to $\left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle$ as $h \to 0$, for meshes defined on Ω_0 with number of nodes $n_{\Omega_0} = 420$ (black), and $n_{\Omega_0} = 2184$ (blue). On the left, the values of $(\mathcal{J}(\mathcal{V} + h\tilde{\mathcal{V}}) - \mathcal{J}(\mathcal{V}))/h$ (full lines with dots), and $\left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle$ (dashed lines) are reported. On the right, a log-log plot of the difference $(\mathcal{J}(\mathcal{V} + h\tilde{\mathcal{V}}) - \mathcal{J}(\mathcal{V}))/h - \left\langle \mathcal{J}'(\mathcal{V}), \tilde{\mathcal{V}} \right\rangle$ is given. It is possible to notice how the error scales as $\sim h^{-1}$, as expected from a finite-difference approximation (at least until saturation is reached). Here, the direction $\tilde{\mathcal{V}}$ is chosen as the H^1 representative of $\mathcal{J}'$ (5.12).

Chapter 8

Results

Experiments have been conducted to analyze the evolution of the cost functional as the optimization algorithm proceeds. The collected results have been compared for different refinement levels of the meshes used, and for different choices of the object $\partial\mathcal{D}$. For all the tests, the chosen target function $\mathbf{p}$ is defined as

$$\mathbf{p}(\mathbf{x}) = -\frac{c}{(\sqrt{z^2+1})^3}\mathbf{e_z}, \tag{8.1}$$

of which only the tangential component is considered.

As $\partial\mathcal{D}$, a cube of side 0.6 and a prism with elliptical basis of semi-axes 0.2 and 0.1 have been used. A representation of both these objects is shown in Fig. 8.1. In Fig. 8.2 the corresponding evolution of the cost functional is reported. We can see how saturation tends to be reached quite soon, and the optimization algorithm struggles to reduce the value of (2.8), which indicates that the initial configuration Ω_0 is already relatively

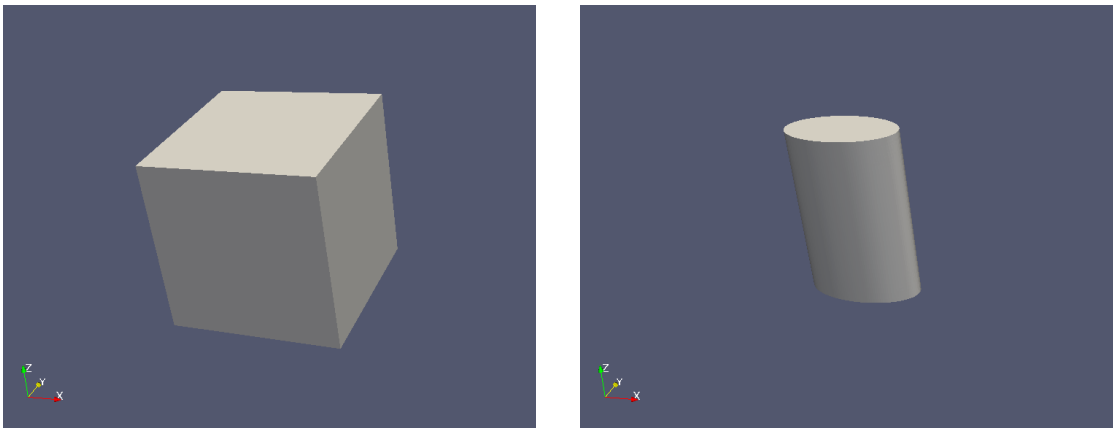

FIGURE 8.1: Internal objects $\partial\mathcal{D}$ used in the experiments. The cube on the left has side 0.4, while the prism with elliptic basis on the right has semi-axes 0.1 and 0.2, and height 0.6. Both of them are centered with respect to the torus.

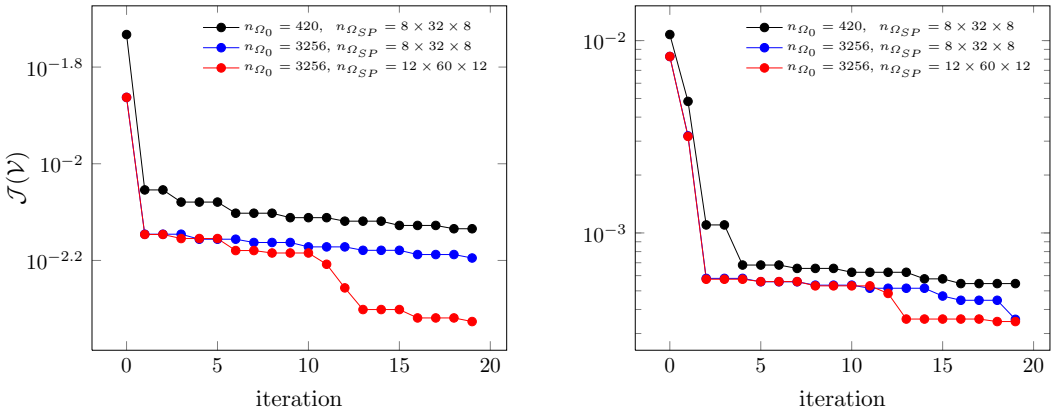

FIGURE 8.2: Semi-Logarithmic plot of the values of $\mathcal{J}(\mathcal{V})$ for meshes defined on Ω_0 with number of nodes $n_{\Omega_0} = 420$ (black) and $n_{\Omega_0} = 3256$ (blue and red), at each iteration of the optimization algorithm. The results on the left refers to $\partial\mathcal{D}$ taken as the cube in Fig. 8.1, while on the right there are those for the prism. The splines grid has respectively $8 \times 32 \times 8$ nodes (black and blue) and $12 \times 60 \times 12$ nodes (red). The effects of meshes refinement are more pronounced in the case of the cube, probably because the shape is much less regular and presents wilder angles.

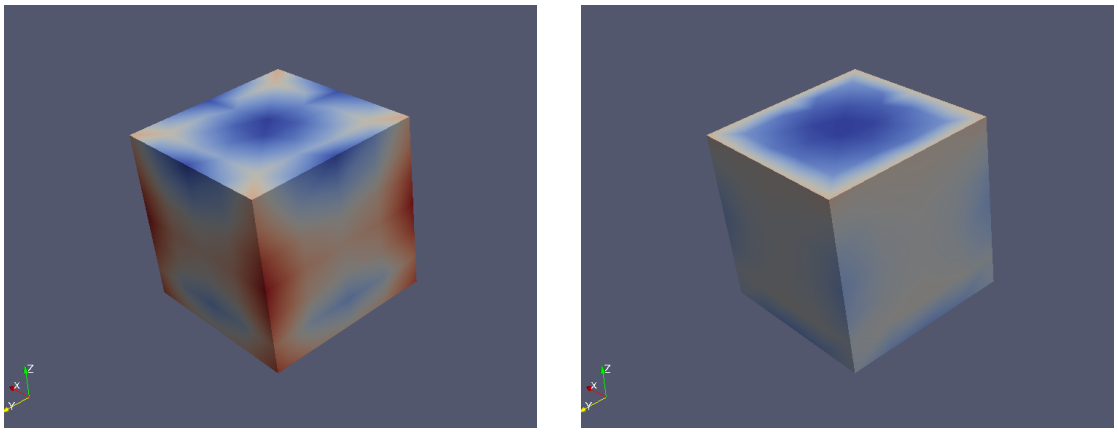

FIGURE 8.3: Norm of the tangential component of the difference between magnetic field and target function $\|(\mathbf{H} - \mathbf{p})_t\|_{\partial\mathcal{D}}$, before and after the optimization process.

close to optimal. Refining the mesh on Ω_0 can help to improve the result, to a certain extent. To overcome saturation, the splines grid can be refined as well: an example of the consequent improvement is also reported in Fig. 8.2. The reason why in some steps $\mathcal{J}$ is stationary lies in the application of Armijo rule: the flat regions in the graph represent iterations where the test (4.1) has failed.

In Figg. 8.4 and 8.5, the actual evolution of the shape of the torus Ω_0 is presented for both cases. We can notice how the conductor tends to embrace the shape of the internal object, to better align the magnetic field to the target function (8.1). A plot of the norm of $(\mathbf{H} - \mathbf{p})_t$ on the cube before and after the optimization process is shown in Fig. 8.3.

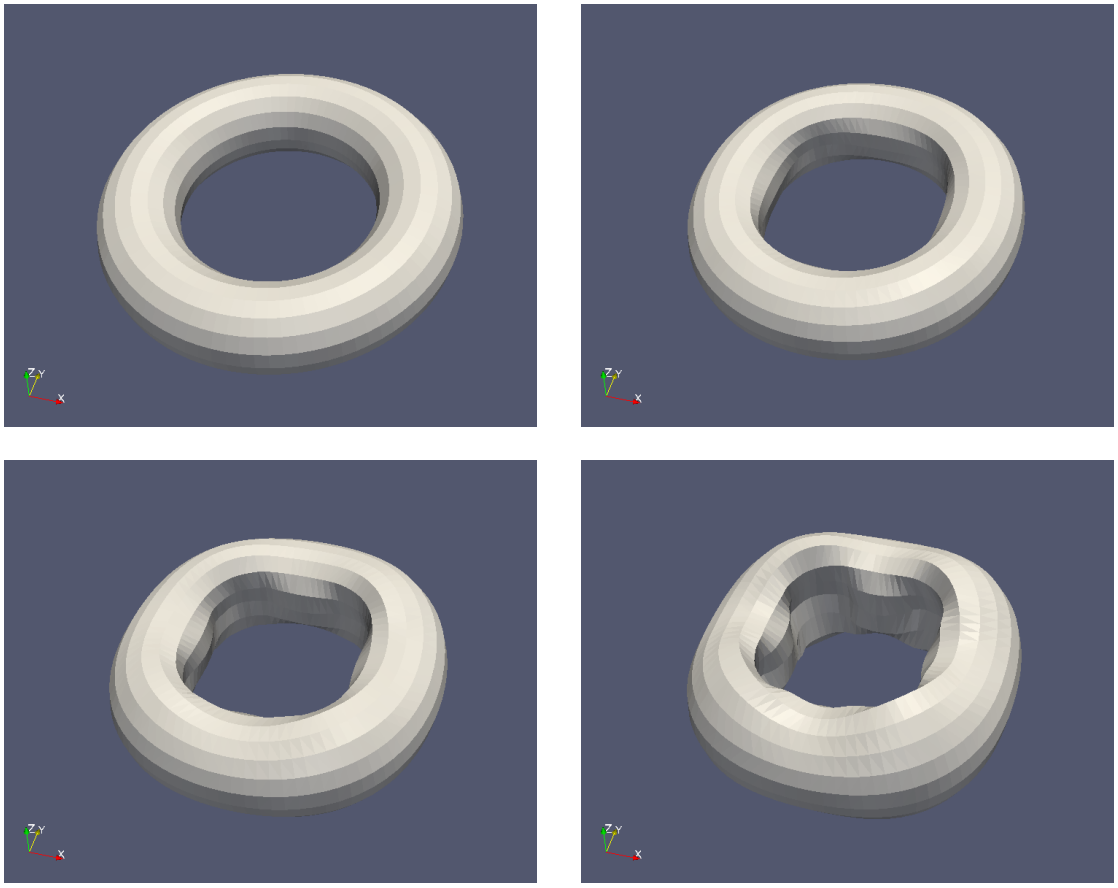

FIGURE 8.4: Evolution of the shape $T_{\nu_h}(\Omega_0)$ for the case in which the internal object $\partial\mathcal{D}$ is the cube in Fig. 8.1. The reference domain Ω_0 is on the top left.

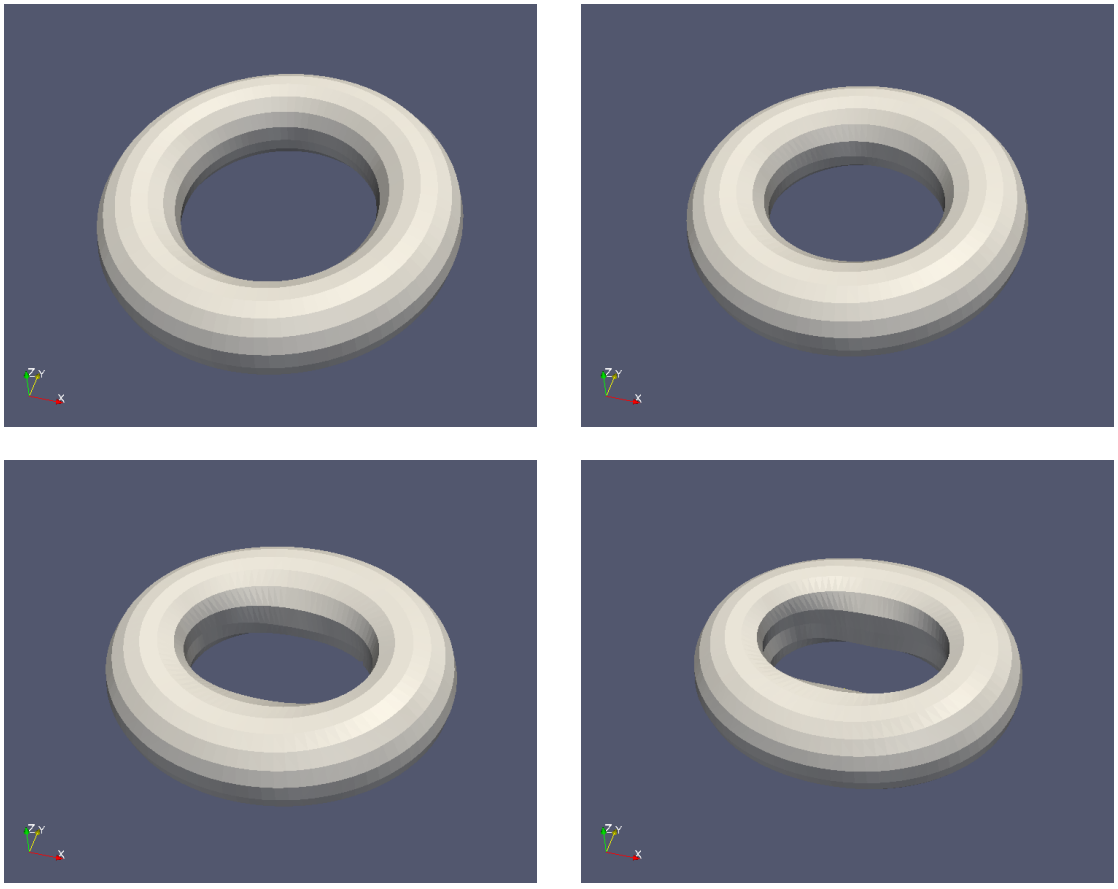

FIGURE 8.5: Evolution of the shape $T_{\nu_h}(\Omega_0)$ for the case in which the internal object $\partial\mathcal{D}$ is the prism with elliptic basis in Fig. 8.1. The reference domain Ω_0 is on the top left.

Chapter 9

Further work

9.1 Parallelization

As already stated in Chap. 6, Alg. 2 represents the bottleneck of the whole program in terms of computational time required for execution. Due to its nature, however, it also offers many possibilities in terms of parallelization: each of the iterations of the `for` loops over the integration points $\mathbf{y_q}$ and $\mathbf{x_q}$ is in fact independent of any of the others. The easiest way of exploiting this is by directly parallelizing the outer `for` loop. A fine grain parallelization can be applied instead to the routines for the splines evaluation, Lst. 6.3, Lst. 6.4 and Lst. 6.5, which are often called throughout the code and represent another part that is at the same time easily parallelizable and expensive in terms of computational cost.

9.2 Pre-evaluation of spline basis functions

In Sec. 6.1 and in Sec. 9.1, we pointed out how demanding the functions for splines evaluations are. Combining the algorithms described in Lst. 6.3, Lst. 6.4 and Lst. 6.5, we can see how, in order to recover the values of the splines with support on a point $\mathbf{y}$ and their gradients, we first need to apply to $\mathbf{y}$ the inverses of the maps (5.6) and (5.7); then, the 1D splines must be evaluated; finally, the 3D splines can be computed from them. This amounts to the application of some non-linear functions (mapping), plus the evaluation of $3(N+1)$ polynomials of order N (1D splines) and as many of order $(N-1)$ (derivatives), plus $4(N+1)^3$ additional multiplications (3D splines and derivatives). However, the points where the splines values need to be calculated, (*i.e.*, the quadrature points $\mathbf{y_q} \in \Omega_0$), are always the same throughout the whole algorithm. This suggests that these values might actually be pre-computed and stored, thus drastically

reducing the amount of mathematical operations that is necessary to perform in various parts of the code, and in Alg. 2 in particular. The additional memory required to store these values is proportional to $4N_{\mathbf{y_q}}(N+1)^3$, being $N_{\mathbf{y_q}}$ the number of quadrature points $\mathbf{y_q} \in \Omega_0$. This size is independent on m, it grows linearly with the number of nodes composing the mesh on Ω_0, and is hence comparable to the memory used for storing the system matrix (5.3). Still, it is in general[1] much smaller than that used for the temporary matrices in Alg. 2.

9.3 Higher order information for descent direction

More advanced alternatives to the steepest descent algorithm described in Alg. 1 are available in order to improve convergence. These in general require more complex information coming from higher order derivatives of the cost functional (2.8). For example, if the second order derivative $\mathcal{J}''(\mathcal{V})$ of (2.8) was to be evaluated along the direction of the gradient representative $\tilde{\mathcal{V}}$ chosen with (5.12), a line-optimization problem could be solved to recover an optimal step size σ_{opt}. This is a more effective choice than that of a fixed $\overline{\sigma}$ employed by the classical steepest descent algorithm, and it can be achieved by solving

$$\begin{aligned} \sigma_{opt} \quad &\coloneqq \quad \operatorname{argmin}_\sigma \tfrac{1}{2}\sigma^2 \left\langle \mathcal{J}''(\mathcal{V};\tilde{\mathcal{V}}),\tilde{\mathcal{V}} \right\rangle + \sigma \left\langle \mathcal{J}'(\mathcal{V}),\tilde{\mathcal{V}} \right\rangle \\ &\implies \quad \sigma_{opt} = -\frac{\langle \mathcal{J}'(\mathcal{V}),\tilde{\mathcal{V}}\rangle}{\langle \mathcal{J}''(\mathcal{V};\tilde{\mathcal{V}}),\tilde{\mathcal{V}}\rangle}. \end{aligned} \tag{9.1}$$

Furthermore, if the whole operator $\mathcal{J}''(\mathcal{V})$ was available, then even a *Newton method* [3, Chap. 2] could be implemented. This method employs a linearization of the optimality condition $\left\langle \mathcal{J}'(\mathcal{V}),\tilde{\mathcal{V}} \right\rangle = 0$ in order to recover a more precise descent direction $\tilde{\mathcal{V}}$. This translates into finding the solution of the system

$$\left\langle \mathcal{J}''(\mathcal{V}),\tilde{\mathcal{V}} \right\rangle = -\mathcal{J}'(\tilde{\mathcal{V}}). \tag{9.2}$$

Although both these techniques should show some improvements in terms of convergence, it is also true that they do require more complicated evaluations, and might consequently significantly increase the computational effort needed.

[1] Increasing the order of the spline basis functions influences with cubic proportionality this amount, so this might not be true if splines of very high order are used. Furthermore, we assume the number of quadrature points $\mathbf{x_q} \in \partial\mathcal{D}$ and $\mathbf{y_q} \in \Omega_0$ to be comparable. If this is not the case, and $N_{\mathbf{x_q}} \lll N_{\mathbf{y_q}}$, again this statement might not be true anymore.

Chapter 10

Conclusion

An application of a shape optimization problem in the field of magnetic heat induction was implemented and discussed. The problem consisted in finding the domain Ω of a toroidal conductor traversed by a current, so that the induced magnetic field $\mathbf{H}$ on the surface of an object $\partial\mathcal{D}$ is as close as possible to a given target function $\mathbf{p}$. The cost functional $\mathcal{J}$ we were interested in minimizing considers the squared norm of the difference between $\mathbf{H}$ and $\mathbf{p}$.

To evaluate the resulting magnetic field, first a scalar Laplace equation was solved to recover the electric scalar potential on Ω. This was done using a Finite-Element method, tailored to consider fixed jump discontinuities by expanding the canonical piecewise linear Lagrangian basis functions set with a cut-off basis function. Then, the magnetic field on $\partial\mathcal{D}$ was computed applying Biot-Savart law.

The shape Ω was expressed via a parametrization that employs a 3D tensor product of spline basis functions. These were defined on a Cartesian grid encircling the conductor. The whole shape optimization problem was cast in parametric form using a change of coordinates from the initial reference domain Ω_0 to the updated one Ω. This identifies the shape optimization scheme used as a fixed-mesh method.

The optimization algorithm that was chosen belongs to the class of steepest descent methods, where the solution is approached by progressively following the direction indicated by the H^1 representative of the gradient of $\mathcal{J}$. The admissibility of the step size for each iteration was guaranteed by the application of Armijo rule.

For the actual implementation of the code, the library BETL2 was widely used. This was conducted with a focus both on the computational effort and on the amount of memory necessary to run the algorithms, well aware of the trade-off between the two. C++ was used as a programming language, and the code was structured using basic

patterns of object-oriented design, when it could be done without affecting the overall performance.

Validation tests were applied to confirm the correctness of the implementation. The most relevant ones covered the routines for the evaluation of the solution of the mapped state equation, of the resulting magnetic field, and of the shape derivative of the cost functional. All of them provided satisfactory results that were in line with the theoretical ones.

Some of the results obtained from the various experiments conducted were presented, with particular attention to the evolution of the cost functional throughout the optimization algorithm, and to the resulting modification of the actual shape, in order to adapt to different choices of the object $\partial\mathcal{D}$. Some effects related to mesh refinements and saturation were also discussed.

Further work should be directed towards the implementation of parallelization techniques, since the most expensive algorithm in the code would be perfectly suited for them, because of its very nature. This should provide a noticeable speed-up at a relatively low overhead cost. Other improvements could be applied to the optimization algorithm: considering second order shape derivatives should allow for faster convergence, although they would be more demanding in terms of computational cost.

Appendix A

Additional extracts from the code

In this appendix, other relevant parts of the implementation that were not presented in Chap. 6 are reported and commented. The aim of this section is to provide a more complete description of the code for future reference.

The methods here described are divided according to the classes they belong to.

A.1 SplinesGrid

In Lst. A.1 are reported the functions responsible to map coordinates from Cartesian to cylindrical to local coordinates, as defined in (5.7) and (5.6).

```
1  //NB: this also translates the z component so that every polar coordinate
       is always taken positive
2  void global2CylindricalCoord( const vector3_t& globalCoord,
3                vector3_t& cylindricalCoord) const {
4
5    numeric_t rho   = sqrt(  globalCoord(0) * globalCoord(0)
6                           + globalCoord(1) * globalCoord(1) );
7    numeric_t theta = atan2( globalCoord(1),  globalCoord(0) );
8
9    //NB all theta are taken as positive:
10   if(theta < 0){
11     theta += 2 * PI;
12   }
13   numeric_t z = globalCoord(2) + sideHalfLength_;
14
15   cylindricalCoord <<   rho, theta, z;
16 }
17
18
```

```
19
20 //NB: in order to work, all the coordinates in cylindricalCoord MUST BE
     POSITIVE
21 void cylindrical2LocalCoord(const vector3_t& cylindricalCoord,
22                 vector3_t& localCoord) const {
23
24   numeric_t scaledRho   = ( cylindricalCoord(0)-(radius_-sideHalfLength_)
     )/refRho_;
25   numeric_t scaledTheta =   cylindricalCoord(1)/refTheta_;
26   numeric_t scaledZ     =   cylindricalCoord(2)/refZ_;
27
28
29   numeric_t rhoHat    = scaledRho   - (idx_t)( scaledRho );
30   numeric_t thetaHat  = scaledTheta - (idx_t)( scaledTheta );
31   numeric_t zHat      = scaledZ     - (idx_t)( scaledZ );
32
33   localCoord << rhoHat, thetaHat, zHat;
34 }
35
```

LISTING A.1: Implementation of maps from cartesian to cylindrical (5.6) and from cylindrical to local coordinates (5.7).

Lst. A.2 evaluates the inverse transposed Jacobian of these maps, used in the computation of the gradient of spline basis functions (5.8).

```
1  void getJacobianInverseTransposed(
2                          const vector3_t& cylindricalCoord,
3                          matrix3_t& jacobian) const {
4
5    numeric_t rho = cylindricalCoord(0);
6    numeric_t theta = cylindricalCoord(1);
7
8    numeric_t st = sin(theta);
9    numeric_t ct = cos(theta);
10
11   jacobian << ct/refRho_, -st/(rho * refTheta_),  0,
12               st/refRho_, ct /(rho * refTheta_),  0,
13               0,          0,                      1/refZ_;
14 }
15
```

LISTING A.2: Function to recover inverse transposed Jacobian of map from local to global coordinates, as appears in (5.8).

A.2 SplinesBasisFunc

The routine for building the Gram matrix (5.10), is shown in Lst. A.3. This is used in the evaluation of the H^1 gradient representative, as described in Sec. 5.2.

```
template< unsigned int ORDER >
void SplinesBasisFunc< ORDER >::initializeGramMatrixH1(){

  //Retrieve quadrature info
  const auto&  wXi = hexaQuad_t::getWeights( );
  //NB: no need to rescale. The original reference hexa is defined on
    [-1,1]^3 and hence has a volume of 2^3 = 8 = scaleX, but since I'm
    remapping it to [0,1]^3, the volume = sum(weights) should be 1, as it
    is.
  //However, the coordinates for hexa integration points are defined on the
     reference domain [-1,1]^3, and need to be remapped: so, first add
    (1,1,1), then divide by 2 to get local coordinates in [0,1]^3
  const matrix_t< 3, hexaQuad_t::getNumPoints( ) > xi = 0.5 * ( hexaQuad_t
    ::getPoints( ) + matrix_t< 3, hexaQuad_t::getNumPoints( ) >::Constant
    (1.) );

  typedef Eigen::Triplet<numeric_t> triplet_t;
  typedef std::vector< triplet_t > tripletList_t;
  tripletList_t tripletList;
  //At every cell, at every integration point, every basis func that
    intersects that point is tested with any other
  tripletList.reserve( numSpWithSuppOnPoint_
                       *numSpWithSuppOnPoint_
                       *grid_.getNumberOfCells()*xi.cols() );
  //Every spline basis function support intersects numSpWithSuppOnPoint_
    other splines basis functions support
  grahmMatrix_.reserve( numSpWithSuppOnPoint_
                      *this->getNumBasisFunc()/3.);

  //Loop through all cells (actually, things change only in the radial
    direction, so it is not necessary to evaluate contributions for every
    single cell)
  for(idx_t j=0;j<grid_.getNumberOfCellsRad();j++){
    idx3_t cellIdx;
    cellIdx << j, 0, 0;
    //loop through all quadrature points (NO NEED! Consider reducing)
    for(idx_t i=0;i<xi.cols();i++){
      vector3_t cylindricalCoord;
      //map local to cylindrical coordinates
      grid_.local2cylindricalCoord( xi.col(i), cellIdx, cylindricalCoord );
      //get J^-T
      matrix3_t JacInvT;
      grid_.getJacobianInverseTransposed( cylindricalCoord, JacInvT );
```

```
33      //determinant of inverse is inverse of determinant
34      const numeric_t det = 1./( JacInvT.determinant() );
35      const numeric_t weight = wXi(i) * det;
36      //evaluate all possible splines
37      vector_t< numSpWithSuppOnPoint_ > B;
38      this->evaluateSplines3DLocal( xi.col(i), B );
39      const vector_t< numSpWithSuppOnPoint_ > Bw = B * weight;
40      //evaluate all possible spline gradients
41      matrix_t< 3, numSpWithSuppOnPoint_ > gradBtemp;
42      this->evaluateSplinesDeriv3DLocal( xi.col(i), gradBtemp );
43      const matrix_t< 3, numSpWithSuppOnPoint_ > gradB = JacInvT *
    gradBtemp;
44      const matrix_t< 3, numSpWithSuppOnPoint_ > gradBw = gradB * weight;
45      //evaluate all possible products BiBj (weight included)
46      const matrix_t< numSpWithSuppOnPoint_, numSpWithSuppOnPoint_ > BiBjw
    = B * Bw.transpose();
47      //and all possible scalar products gradBi * gradBj (weight included)
48      const matrix_t< numSpWithSuppOnPoint_, numSpWithSuppOnPoint_ >
    gradBigradBjw = gradB.transpose() * gradBw;
49      //then, put them together
50      const matrix_t< numSpWithSuppOnPoint_, numSpWithSuppOnPoint_ >
    BiBjplusgradBigradBjw = BiBjw + gradBigradBjw;
51      //now, loop through the rest of the cells
52      for(idx_t kk=0;kk<grid_.getNumberOfCellsRad();kk++ ){
53        cellIdx(2) = kk;
54        for(idx_t jj=0;jj<grid_.getNumberOfCellsTang();jj++){
55          cellIdx(1) = jj;
56          //get map from local to global spline index
57          matrix_t< 1, numSpWithSuppOnPoint_ > mapl2gSplineIdx;
58          matrix_t< numSpWithSuppOnPoint_, dim > dummy;
59          getReductionCoeffNU( cellIdx, dummy, mapl2gSplineIdx );
60          //And add contributions to the Gram matrix
61          for(idx_t iii=0;iii<numSpWithSuppOnPoint_;iii++){
62            const idx_t mapiii = mapl2gSplineIdx(iii);
63            for(idx_t jjj=0;jjj<numSpWithSuppOnPoint_;jjj++){
64              const idx_t mapjjj = mapl2gSplineIdx(jjj);
65
66              tripletList.push_back( triplet_t ( mapiii, mapjjj,
    BiBjplusgradBigradBjw(iii,jjj) ) );
67
68            }//end loop through test spline basis functions
69          }//end loop through trial spline basis functions
70        }//end loop through "theta" cells
71      }//end loop through "z" cells
72    }//end loop through quadrature points
73  }//end loop through "rho" cells
74
75  //it's finally time to assemble the Gram matrix :)
```

```
76   gramMatrix_.setFromTriplets( tripletList.begin(), tripletList.end() );
77   gramMatrix_.makeCompressed();
78
79 }
80
```

Listing A.3: Routine for assembling Gram matrix (5.10).

Another relevant function of the class `SplinesBasisFunc` is the one that computes the determinant of $\mathbf{D}T_{\mathcal{V}}$. It can be found in Lst. A.4.

```
1  void getDTnu( const vector3_t& globalCoord, matrix3_t& DTnu) const {
2    //transform coordinates:
3    vector3_t cylindricalCoord, localCoord;
4    grid_.global2CylindricalCoord( globalCoord, cylindricalCoord );
5    grid_.cylindrical2LocalCoord( cylindricalCoord, localCoord );
6
7    //get gradient of all possible splines evaluated at localCoord
8    matrix_t< dim, numSpWithSuppOnPoint_ > splinesGrad;
9    evaluateSplinesDeriv3DLocal( localCoord, splinesGrad );
10   //get index of the cell that contains the evaluation point
11   idx3_t cellIdx;
12   grid_.getCellIdx( cylindricalCoord, cellIdx );
13   //recover the coefficients that corresponds to the local evaluations of
       the splines
14   matrix_t< numSpWithSuppOnPoint_, dim > redCoeffNU_XYZ;
15   getReductionCoeffNU( cellIdx, redCoeffNU_XYZ );
16   //reduce the gradients and recover the 3x3 matrix D(rtz)V
17   matrix3_t gradRTZ = splinesGrad * redCoeffNU_XYZ;
18   //these gradients are evaluated wrt the rho, theta, z local coordinates,
       though. In order to get the proper gradient in xyz coordinates, we need
       to multiply by the jacobian of the transformation:
19   matrix3_t JacInvT;
20   grid_.getJacobianInverseTransposed( cylindricalCoord, JacInvT );
21   matrix3_t gradXYZ = JacInvT * gradRTZ;
22
23   //initialize DTnu to identity:
24   DTnu << 1, 0, 0,
25           0, 1, 0,
26           0, 0, 1;
27
28   //Add the contribution of DV to DTnu (it needs to be transposed)
29   DTnu += gradXYZ.transpose();
30 }
31
```

Listing A.4: Function that recovers $\det(\mathbf{D}T_{\mathcal{V}})$.

In Lst.A.5, the routine responsible to control the positiveness of $\det(\mathbf{D}T_{\mathcal{V}})$, as described in Alg. 1, is illustrated. The check is conducted on all quadrature points of the mesh on Ω_0 used for the various integrations, as well as on those defined on the cells of the grid on Ω_{SP} for the computation of the Gram matrix. The update step size is halved only for those spline basis functions responsible for rendering the determinant too small.

```
template< typename QUADRATURE_VOL_T, typename FESPACE_T >
void updateCoeffNU( const FESPACE_T& fespace, matrix_t< Eigen::Dynamic, dim > & update ){

  const matrix_t< Eigen::Dynamic, dim > coeffNU_XYZ_old = coeffNU_XYZ_;
  //first of all, update the coefficients of V
  coeffNU_XYZ_ += update;

  //tolerance: determinant must be larger than this
  const numeric_t eps=0.001;

  //retrieve quadrature points (for both the FE and the splines cells)
  const auto&  yi = QUADRATURE_VOL_T::getPoints( );
  const matrix_t< 3, hexaQuad_t::getNumPoints( ) > xi = 0.5 * ( hexaQuad_t::getPoints( ) + matrix_t< 3, hexaQuad_t::getNumPoints( ) >::Constant(1.) );
  //initialize halving of step size
  numeric_t delta = 0.5;

  //flag to check if the determinant is too small somewhere
  bool negativeDeterminant = true;

  while( negativeDeterminant ){
    //get ready to see if the determinant is too small somewhere
    negativeDeterminant = false;
    //this will keep track of those splines that are responsible or rendering negative the determinant somewhere
    matrix_t< Eigen::Dynamic, 1 > splinesWithSmallDet( update.rows() );
    splinesWithSmallDet.setZero();

    //loop through FE volume elements
    for(const auto& E : fespace) {
      // map gp to global coordinates
      matrix_t< 3, Eigen::Dynamic > gpCoordY = E.geometry( ).global( yi );
      for(idx_t i=0;i<gpCoordY.cols();i++){
        matrix3_t DTnu;
        matrix_t< dim, numSpWithSuppOnPoint_ > dummy;
        matrix_t< 1, numSpWithSuppOnPoint_ > mapl2gSplineIdx;
        this->getGradBandDTnu( gpCoordY.col(i), dummy, DTnu, mapl2gSplineIdx );
        //if you find a very small determinant
```

```
37        if( DTnu.determinant() < eps ){
38          negativeDeterminant = true;
39          //mark the splines that are responsible for that
40          for(idx_t spIdx=0;spIdx<numSpWithSuppOnPoint_;spIdx++){
41            splinesWithSmallDet( mapl2gSplineIdx( spIdx ) ) = 1;
42          }//end loop on incriminated splines basis func
43        }//end if det small
44      }//end loop on y quadrature points
45    }//end loop on FE elements
46
47    //same thing cycling through the sp cells
48    idx3_t cellIdx;
49    cellIdx.setZero();
50
51    for(idx_t kk=0;kk<grid_.getNumberOfCellsRad();kk++){
52      cellIdx(2) = kk;
53      for(idx_t jj=0;jj<grid_.getNumberOfCellsTang();jj++){
54        cellIdx(1) = jj;
55        for(idx_t ii=0;ii<grid_.getNumberOfCellsRad();ii++){
56          cellIdx(0) = ii;
57          for(idx_t i=0;i<xi.cols();i++){
58            vector3_t globalCoord;
59            grid_.local2GlobalCoord( xi.col(i), cellIdx, globalCoord );
60            matrix3_t DTnu;
61            matrix_t< dim, numSpWithSuppOnPoint_ > dummy;
62            matrix_t< 1, numSpWithSuppOnPoint_ > mapl2gSplineIdx;
63            this->getGradBandDTnu( globalCoord, dummy, DTnu,
    mapl2gSplineIdx );
64
65            //if you find a very small determinant
66            if( DTnu.determinant() < eps ){
67              negativeDeterminant = true;
68              //mark the splines that are responsible for that
69              for(idx_t spIdx=0;spIdx<numSpWithSuppOnPoint_;spIdx++){
70                splinesWithSmallDet(mapl2gSplineIdx(spIdx))=1;
71              }//end loop on incriminated splines basis func
72            }//end if det small
73          }//end loop on x quadrature points
74        }//end loop on "rho" cells
75      }//end loop on "theta" cells
76    }//end loop on "z" cells
77
78    //if some of these splines make the determinant negative
79    if( negativeDeterminant ){
80      std::cout<<"Negative determinant found. Corresponding splines basis
    functions scaled by " << delta << std::endl;
81      //scale the corresponding sp basis functions coefficients
82      coeffNU_XYZ_ -= (delta*splinesWithSmallDet).asDiagonal() * update;
```

```
83        //reduce delta
84        delta *= 0.5;
85      }//end if small det is found
86    }//end while(det is small)
87
88    //finally, store the (eventually modified) update
89    update = coeffNU_XYZ_ - coeffNU_XYZ_old;
90  }
91
```

LISTING A.5: Function for checking positiveness of $\det(\mathbf{D}T_{\mathcal{V}})$.

A.3 CostFunctionalIntegrator

Next is reported the implementation of Alg. 2. Due to its intricacy, it has been split in a few different functions: Lst. A.6 invokes Lst. A.7 to compute the integrands, and then performs the final scalar products and reductions necessary to retrieve the values of $\mathcal{J}$ and its partial derivatives.

```
1  void compute( numeric_t& Jval, colVector_t& dJu, colVector_t& dJnu ) const{
2    //-will contain H(U) = A(U) x B. rows -> x,y,z components. cols -> x-
       integration point
3    rowMatrix3_t HUxyz;
4    //-will contain x,y,z components of H(Phi) = A(Phi) x B. rows -> phi
       basis func. cols -> x-integration point
5    matrix_t     HPhix, HPhiy, HPhiz;
6    //-will contain x,y,z components of d(AxB)(Psi). rows -> Psi basis func (
       that refers to X,Y and Z components, in order). cols -> x-integration
       point
7    matrix_t     dAxBx, dAxBy, dAxBz;
8
9    //Evaluate the internal functions (integrals over Omega (y))
10   evaluateInternalFunctions( HUxyz, HPhix, HPhiy, HPhiz, dAxBx, dAxBy,
       dAxBz );
11
12   //Prepare to perform dot products to recover J, Ju and Jnu
13   const rowMatrix3_t HUmP = (1./(4*PI) * HUxyz) - evalP_;
14   //-retrieve normal components:
15   const rowVector_t nHUmP = ((HUmP.cwiseProduct(n_)).colwise()).sum();
16   const matrix_t nHPhi  = (HPhix*((n_.row(0)).asDiagonal()))
17                          +(HPhiy*((n_.row(1)).asDiagonal()))
18                          +(HPhiz*((n_.row(2)).asDiagonal()));
19   const matrix_t ndAxB  = (dAxBx*((n_.row(0)).asDiagonal()))
20                          +(dAxBy*((n_.row(1)).asDiagonal()))
21                          +(dAxBz*((n_.row(2)).asDiagonal()));
```

```
22   //-retrieve tangential components
23   const rowMatrix3_t HUmPt= HUmP-n_*(nHUmP.asDiagonal());
24   const matrix_t HPhixt= HPhix-nHPhi*((n_.row(0)).asDiagonal());
25   const matrix_t HPhiyt= HPhiy-nHPhi*((n_.row(1)).asDiagonal());
26   const matrix_t HPhizt= HPhiz-nHPhi*((n_.row(2)).asDiagonal());
27   const matrix_t dAxBxt= dAxBx-ndAxB*((n_.row(0)).asDiagonal());
28   const matrix_t dAxByt= dAxBy-ndAxB*((n_.row(1)).asDiagonal());
29   const matrix_t dAxBzt= dAxBz-ndAxB*((n_.row(2)).asDiagonal());
30   //-perform dot products
31   const rowVector_t HUmPtHUmPt= (HUmPt.colwise()).squaredNorm();
32   const matrix_t HUmPtHPhit= HPhixt*(HUmPt.row(0)).asDiagonal()
33                              +HPhiyt*(HUmPt.row(1)).asDiagonal()
34                              +HPhizt*(HUmPt.row(2)).asDiagonal();
35   const matrix_t HUmPtdAxBt= dAxBxt*(HUmPt.row(0)).asDiagonal()
36                              +dAxByt*(HUmPt.row(1)).asDiagonal()
37                              +dAxBzt*(HUmPt.row(2)).asDiagonal();
38   //Finally, integrate over dD (x)
39   Jval = (HUmPtHUmPt.dot(weightsX_));
40   dJu  = 1./(2*PI) * (HUmPtHPhit*weightsX_);
41   dJnu = 1./(2*PI) * (HUmPtdAxBt*weightsX_);
42 }
43
```

Listing A.6: Implementation of Alg. 2 - scalar products and integration on $\partial\mathcal{D}$.

Lst. A.7 takes care of evaluating the integrals over Ω_0, recovering their values for each quadrature point in $\partial\mathcal{D}$ and for each basis function ϕ or ψ.

```
1  void evaluateInternalFunctions( rowMatrix3_t& HUxyz, matrix_t& HPhix,
       matrix_t& HPhiy, matrix_t& HPhiz, matrix_t& dAxBx, matrix_t& dAxBy,
       matrix_t& dAxBz ) const{
2
3    //Resize matrices adequately
4    //-dim x num of x-quad points
5    HUxyz.resize( 3, globalCoordX_.cols() );
6    HUxyz.setZero();
7    //num of FE basis functions x num of x-quad points
8    HPhix.resize( volFE_.numDofs()+1,globalCoordX_.cols());
9    HPhiy.resize( volFE_.numDofs()+1,globalCoordX_.cols());
10   HPhiz.resize( volFE_.numDofs()+1,globalCoordX_.cols());
11   HPhix.setZero(); HPhiy.setZero(); HPhiz.setZero();
12   //num of SP basis functions x num of x-quad points
13   dAxBx.resize(numSpBasisFunc_,globalCoordX_.cols());
14   dAxBy.resize(numSpBasisFunc_,globalCoordX_.cols());
15   dAxBz.resize(numSpBasisFunc_,globalCoordX_.cols());
16   dAxBx.setZero(); dAxBy.setZero(); dAxBz.setZero();
17
18   //recover info on y quad-points
```

```
19  const auto&  yi          = QUADRULE_VOL_T::getPoints( );
20  const auto&  wYi         = QUADRULE_VOL_T::getWeights( );
21  const numeric_t scaleY   = QUADRULE_VOL_T::getScale( );
22
23  //start loop over finite elements
24  for( const auto& E : volFE_ ) {
25    //map local integration points to global coordinates
26    const rowMatrix3_t globalCoordY = E.geometry().global( yi );
27
28    //evaluate integration coefficients = gaussWeight * scaleY * |detJhat|
29    const Eigen::Matrix< numeric_t, 1, gpPerVolElement_ > weightsY =
30                              scaleY * wYi.cwiseProduct( E.geometry().
    template integrationElement< gpPerVolElement_ >( yi ) ).transpose();
31
32    //  -Recover map from local to global indeces of FE basis functions
33    const auto indices = volFE_.indices( E );
34
35    //for each of the y-quad points of this element,
36    for( idx_t i=0; i<gpPerVolElement_; i++ ){
37
38      // Compute all terms that depend purely on y:
39
40      //  -evaluate term A(nu, u; y) and its derivatives wrt u and nu; also
     recover map from l2g index of SP basis function
41      //  -term dAu(nu, Phi; y) for the generic FE basis function Phi
42      //  -term dAnu(Psi, u; y) for the generic SP basis function Psi
43      vector3_t A;
44      Eigen::Matrix< numeric_t, 3, numDofsTest_ > dAu;
45      Eigen::Matrix< numeric_t, 3, 3*numSpPerPoint_ > dAnu;
46      Eigen::Matrix< numeric_t, 1, numSpPerPoint_ > mapl2gSplineIdx;
47      evaluateAandDA( globalCoordY.col( i ), yi.col( i ), E, weightsY(i), A
    , dAu, dAnu, mapl2gSplineIdx );
48
49      //  -evaluate spline basis functions Psi at y (will be used to
    evaluate B) -> quite heavy, and only depends on y, so precompute
50      Eigen::Matrix< numeric_t, 1, numSpPerPoint_ > Psi;
51      vector3_t TnuY;
52      spBasis_.mapCoordAndGetSpBasisFunc( globalCoordY.col( i ), TnuY, Psi
    );
53
54      //for each of the x-quad points
55      for( idx_t j = 0; j<globalCoordX_.cols(); j++ ){
56
57        //-evaluate term B(nu) and term dB(Psi) for the generic SP basis
    function Psi
58        vector3_t B;
59        Eigen::Matrix< numeric_t, 3, 3*numSpPerPoint_ > dB;
60        evaluateBandDB(globalCoordX_.col( j ),TnuY,Psi,B,dB);
```

```
61
62        // -evaluate cross products and store contributions:
63        //For H(U)
64        HUxyz.col( j ) += A.cross( B );
65
66        //For H(Phi)
67        for( const auto idx : indices ){
68          const vector3_t dAuCrossB = ( dAu.col( idx.local() ) ).cross( B )
      ;
69
70          const auto phiGlobalIdx = idx.global();
71          HPhix( phiGlobalIdx, j ) += dAuCrossB(0);
72          HPhiy( phiGlobalIdx, j ) += dAuCrossB(1);
73          HPhiz( phiGlobalIdx, j ) += dAuCrossB(2);
74        }//end loop over local FE basis functions
75
76
77        //For dnu(H(Psi))
78        for( idx_t xyz = 0; xyz<3; xyz++ ){
79          const idx_t lclIdxTemp = xyz*numSpPerPoint_;
80          const idx_t glbIdxTemp = xyz*numSpBasisFunc_/3;
81
82          for( idx_t ii = 0; ii<numSpPerPoint_; ii++ ){
83            const idx_t lclIdx = lclIdxTemp+ii;
84            const idx_t gblIdx = glbIdxTemp+mapl2gSplineIdx(ii);
85
86            const vector3_t dAcrossBplusACrossdB = (dAnu.col(lclIdx)).cross
      (B) + A.cross(dB.col(lclIdx));
87            dAxBx( gblIdx, j ) += dAcrossBplusACrossdB(0);
88            dAxBy( gblIdx, j ) += dAcrossBplusACrossdB(1);
89            dAxBz( gblIdx, j ) += dAcrossBplusACrossdB(2);
90          }//end loop local SP basis func
91        }//end loop over x-y-z related SP basis func
92      }//end loop x-integration points
93    }//end loop y-integration points
94  }//end loop elements}
95
```

Listing A.7: Implementation of Alg. 2 - Evaluation of integrands.

Lst. A.7 invokes two more internal functions: `evaluateAandDA` is the one responsible for evaluating term **A** in (3.7) and its derivative, and has been described in Lst. 6.9; `evaluateBandDB` computes the value of **B** in (3.7) and the corresponding derivative. The related code is reported in Lst. A.8.

```
1 void evaluateBandDB( const vector3_t& x, const vector3_t& y, const Eigen::
      Matrix< numeric_t, 1, numSpPerPoint_ >& Psi, vector3_t& B, Eigen::
      Matrix< numeric_t, 3, 3*numSpPerPoint_ >& dB ) const{
```

```
  //  -First get y-x
  B = y - x;
  //  -And a few norms that will get useful later on
  const numeric_t norm = B.norm();
  const numeric_t norm2 = norm*norm;
  const numeric_t norm3 = norm*norm*norm;

  typedef Eigen::Matrix< numeric_t, 3, numSpPerPoint_ > rowMatrix3SP_t;
  typedef Eigen::Matrix< numeric_t, 1, numSpPerPoint_ > rowVectorSP_t;

  //  -Start normalizing B
  B *= 1./norm;
  //  -And Psi: this will provide the first term of dB.
  //    -due to the choice of the SP basis functions, Psi = (Psi,0,0)' for
    x-related SP, (0,Psi,0)' for y, and so on
  const rowVectorSP_t Psireduced  = Psi / norm3;

  //  -This is useful for the second term:
  //    -the scalar product (y-x)*(Psi), is just the product of the x-x
    components (for x-related SP), of the y-y for y, and so on
  //    -this is actually -3/|y-x|^4 * (y-x)*(Psi), since I rescaled both B
     and Psi.
  const rowMatrix3SP_t scalarProducts = (-3.*B) * Psireduced;
  //    -now I need to multiply again by B (which is already normalized, so
     to recover the /|y-x|^5)
  dB.template block< 3, numSpPerPoint_ >( 0,   0              ) = B *
    scalarProducts.row(0); //for x-related SP basis func
  dB.template block< 3, numSpPerPoint_ >( 0,   numSpPerPoint_ ) = B *
    scalarProducts.row(1); //for y-related SP basis func
  dB.template block< 3, numSpPerPoint_ >( 0, 2*numSpPerPoint_ ) = B *
    scalarProducts.row(2); //for z-related SP basis func
  //    -Finally add the first term to the right rows
  dB.row(0).segment(0,numSpPerPoint_)+= Psireduced;
  dB.row(1).segment(numSpPerPoint_,numSpPerPoint_)+= Psireduced;
  dB.row(2).segment(2*numSpPerPoint_,numSpPerPoint_)+= Psireduced;
  //B should be divided by norm^3, so adjust it:
  B *= 1/norm2;
}

```

LISTING A.8: Implementation of Alg. 2 - Evaluation of **B** and its derivative (3.7).

Bibliography

[1] G. Allaire. *Conception optimale de structures*, vol. 58 of Mathématiques & Applications (Berlin) [Mathematics & Applications], Springer-Verlag, Berlin, 2007. With the collaboration of Marc Schoenauer (INRIA) in the writing of Chapter 8.

[2] W. Rudin. *Real and Complex Analysis*. McGraw Hill.

[3] M. Hinze, R. Pinnau, M. Ulbrich, S. Ulbrich. *Optimization with PDE constraints*. Springer, New York, 2009.

[4] R. E. Haimbaugh. *Practical induction heat treating*, ASM International, 2001.

[5] R. Hiptmair, A. Paganini, S. Sargheini. Comparison of approximate shape gradients. *BIT Numer Math DOI 10.1007/s10543-014-0515-z*, July 2014.

[6] R. Hiptmair, A. Paganini. Shape optimization by pursuing diffeomorphisms. *ETH SIAM, Research Report No. 2014-27*, December 2014.

[7] F. Ballarin, A. Manzoni, G. Rozza, and S. Salsa. Shape optimization by free-form deformation: existence results and numerical solution for Stokes flows. *J. Sci. Comput., 60*, December 2013.

[8] N.H. Kim, Y. Chang. Eulerian shape design sensitivity analysis and optimization with a fixed grid. *Comput. Methods Appl. Mech. Engrg. 194 3291-3314*, December 2004.

[9] Z Liu, J.G. Korvink. Structural Shape Optimization Using Moving Mesh Method. *Excerpt from the Proceedings of the COMSOL Users Conference 2007 Grenoble*, 2007.

[10] M.L. Staten, S.J. Owen, S.M. Shontz, A.G. Salinger, and T.S. Coffey. A Comparison of Mesh Morphing Methods for 3D Shape Optimization.

[11] P. Mazzoldi, M. Nigro, C. Voci. *Fisica*, vol. 2, Edises, 1998.

[12] D. P. Bertsekas, *Constrained optimization and Lagrange multipliers method*, Athena Scientific, Belmont, Massachussets, 1996.

[13] M. Benzi, G. H. Golub and J. Liesen. Numerical solution of saddle point problems. *Acta Numerica*, Vol. 14, pp. 1-137. Cambridge University Press, 2005

[14] R. A. Horn, C. R. Johnson. *Matrix Analysis*. Cambridge University Press, 2013.

[15] S. Salsa. *Equazioni a Derivate Parziali*. Springer-Verlag, Italia, Milano, 2010.

[16] H. Schwerdtfeger. *Introduction to linear algebra and the theory of matrices*. Noordhoff, 1950.

[17] S. Kumagai. An implicit function theorem: Comment. *Journal of Optimization Theory and Applications*, Vol. 31 (2), pp. 285-288, June 1980.

[18] D. F. Rogers. *An Introduction to NURBS*. Elsevier, 2001.

[19] BETL2 documentation: http://www.sam.math.ethz.ch/betl/

[20] R. Hiptmair, L. Kielhorn. BETL - A generic boundary element template library. *SAM Research Report No. 2012-36*, November 2012.

[21] C. A. Brebbia, J. C. F. Telles, L. C. Wrobel. *Boundary Element Techniques*. Springer-Verlag, Berlin, 1984.

[22] E. Gamma, R. Helm; R. Johnson, J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[23] Eigen documentation: http://eigen.tuxfamily.org/dox/

[24] P. E. Gill, W. Murray, M. A. Saunders, M. H. Wright. Maintaining LU factors of a general sparse matrix. *Linear Algebra and its Applications*. Vol. 88-89, Pag. 239-270. April, 1987.

[25] A. Quarteroni, R. Sacco, F. Salieri. *Matematica Numerica*. Springer, 2008.

[26] A. Quarteroni. *Modellistica Numerica per Problemi Differenziali*. Springer-Verlag, Italia, 2012.

[27] M. Discacciati, A. Quarteroni and S. Quinodoz. *Numerical approximation of internal discontinuity interface problems*. MATHICSE Technical Report, Nr. 09.2011, September, 2011.

[28] H. T. Rathod, B. Venkatesudu, K. V. Nagaraja. Gauss legendre quadrature formulas over a tetrahedron. *Numerical Methods for Partial Differential Equations*, Vol. 22, pp. 197-219, 2006.

[29] Wyman G. Fair and Yudell L. Luke. Rational Approximations to the Incomplete Elliptic Integrals of the First and Second Kinds. *Mathematics of Computation*, Vol. 21, No. 99, pp. 418-422, July 1967.

[30] Dragan V. Redžić. The magnetic field of a static circular current loop: a new derivation. *European Journal of Physics*, Vol. 27 , No. 5, 2006.

[31] O. Stein. *A Boundary Element Method for Eddy Currents*. Master thesis. Supervisor: Prof. Dr. Ralf Hiptmair, ETH Zürich. Zürich, April, 2015

Declaration of originality

This signed "Declaration of originality" is a required component of any written work (including any electronic version) submitted by a student during the course of studies in Environmental Sciences. For Bachelor and Master theses, a copy of this form is to be attached to the request for diploma.

I hereby declare that this written work is original work which I alone have authored and written in my own words, with the exclusion of proposed corrections.

Title of the work

Shape Optimization in Magnetostatics

Author(s)

Last name	First Name
Danieli	**Federico**

With my signature, I hereby declare:

- I have adhered to all rules outlined in the form on „Citation etiquette", www.ethz.ch/students/exams/plagiarism_s_en.pdf.
- I have truthfully documented all methods, data and operational procedures.
- I have not manipulated any data.
- I have identified all persons who have substantially supported me in my work in the acknowledgements.
- I understand the rules specified above.

I understand that the above written work may be tested electronically for plagiarism.

Zürich, 13.08.2015 Federico Danieli

Place, Date Signature*

* The signatures of all authors are required for work submitted as a group. The authors assert the authenticity of all contents of the written work submitted with their signatures.