# A LEVEL SET-BASED STRUCTURAL OPTIMIZATION CODE USING FENICS

ANTOINE LAURAIN

ABSTRACT. This paper presents an educational code written using FEniCS, based on the level set method, to perform compliance minimization in structural optimization. We use the concept of distributed shape derivative to compute a descent direction for the compliance, which is defined as a shape functional. The use of the distributed shape derivative is facilitated by FEniCS, which allows to handle complicated partial differential equations with a simple implementation. The code is written for compliance minimization in the framework of linearized elasticity, and can be easily adapted to tackle other functionals and partial differential equations. We also provide an extension of the code for compliant mechanisms. We start by explaining how to compute shape derivatives, and discuss the differences between the distributed and boundary expressions of the shape derivative. Then we describe the implementation in details, and show the application of this code to some classical benchmarks of topology optimization. The code is available at `http://antoinelaurain.com/compliance.htm`, and the main file is also given in the appendix.

## 1. INTRODUCTION

The popular "99 line" Matlab code by Sigmund published in 2001 [41] has started a trend of sharing and publishing educational codes for structural optimization. Since then, an upgrade of the "99 line" code has been published, improving speed and reducing the code size to 88 lines; see [7]. The codes of [7, 41] are written for Matlab and are based on the solid isotropic microstructure with penalty (SIMP) approach [10, 51]. Various other codes have been published using different approaches and/or other platforms than Matlab. We review here several categories of approaches to tackle this problem.

In the SIMP approach the material is allowed to have intermediate values, and the optimization variables are the material densities of the mesh elements. The intermediate values are also penalized using a power law to enforce $0-1$ values. Using filtering techniques, it provides feasible designs. Considering SIMP approaches as in [41], Talischi et al. have introduced `PolyMesher` [45] and `PolyTop` [46] to provide a MATLAB implementation of topology optimization using a general framework for finite element discretization and analysis.

Another category of approaches for topology optimization which has emerged after the SIMP approach are level set methods. They consist in representing the boundary of the moving domain $\Omega$ as the zero level set of a function $\phi$. Level set methods were introduced by Osher and Sethian [34] in the context of the mean curvature flow to facilitate the modelization of topological changes during curve evolution. Since then, they have been applied to many shape optimization and boundary perturbations problems. There is already a substantial literature for level set methods applied to structural optimization, see [3, 4, 36, 40, 48] for the pioneering works using this approach, and [47] for a review. Early references for level set approaches include a code in FEMLAB [29] by Liu et al. in 2005, a Matlab code [13] in the spirit of the "99 line" code, by Challis in 2010, and a a 88 lines Matlab code [37] using a reaction-diffusion equation by Otomori et al in 2014.

Other approaches to structural topology optimization include phase-field methods [49], level-set methods without Hamilton-Jacobi equations [8], and an algorithm based on the notion of topological derivative [6]. We also mention an early FreeFem++ code [2] by Allaire and Pantz in 2006, implementing the boundary variation method and the homogenization. For a critical comparison of four different level-set approaches and one phase-field approach, see [22].

The code presented in the present paper enters the category of level set methods. In the usual approach, the concept of shape derivative [17, 42] is used to compute the sensitivity of the objective functional. Is is known that the shape derivative is a distribution on the boundary of the domain, and algorithms are usually based on this property. This means that the shape derivative is expressed as a boundary integral, and then extended to the entire domain or to a narrow band for use in the level set method; see [3, 4, 13, 20, 21, 24, 48] for applications of this approach. The shape derivative can also be written as a domain integral, which is called *distributed*, *volumetric* or *domain expression* of the shape derivative; see [11, 18, 26, 28], and [25, 32, 44] for applications.

From a numerical point of view, the distributed expression is often easier to implement than the boundary expression as it is a volume integral. Other advantages of the distributed expression are presented in [11, 26]. In [11], it is shown that the discretization and the shape differentiation processes commute for the volume expression but not for the boundary expression; i.e., a discretization of the boundary expression does not generally lead to the same expression as the shape derivative computed after the problem is discretized. In [26], the authors conclude

that "volume based expressions for the shape gradient often offer better accuracy than the use of formulas involving traces on boundaries". See also [1] for a discussion about the difficulty to use the boundary expression in the multi-material setting. In the present paper, the main focus is the compact yet efficient implementation of the level set method for structural optimization allowed by the distributed shape derivative. We also show that it is useful to handle the ersatz material approach. Combining these techniques, we obtain a straightforward and general way of solving the shape optimization problem, from the rigorous calculation of the shape derivative to the numerical implementation.

The choice of FEniCS for the implementation is motivated by its ability to facilitate the implementation of complicated variational formulations, thanks to a near-mathematical notation. This is appropriate in our case since the expression of the distributed shape derivative is usually lengthy. The FEniCS Project (`https://fenicsproject.org/`) is a collaborative project with a particular focus on automated solution of differential equations by finite element methods; see [5, 30].

The paper is structured as follows. In Section 2, we recall the definition of the shape derivative, and show the relation between its distributed and boundary expression. In Section 3, we compute the shape derivative in distributed and boundary form for a general functional in linear elasticity using a Lagrangian approach, and we discuss the particular cases of compliance and compliant mechanisms. In Section 4, we show how to obtain descent directions. In section 5, we explain the level set method used in the present paper, which is a variation of the usual level set method suited for the distributed shape derivative. In this section we also describe the discretization and reinitialization procedures. In section 6, we explain in details the numerical implementation. In section 7, we show numerical results for several classical benchmarks. Finally, in section 8, we discuss the computation time and the influence of the initialization on the optimal design. In the appendix, we give the code for the main file `compliance.py`.

## 2. Volume and boundary expressions of the shape derivative

In this section we recall basic notions about the shape derivative, the main tool used in this paper. Let $\mathcal{P}(\mathcal{D})$ be the set of subsets of $\mathcal{D}$, where the so-called *universe* $\mathcal{D} \subset \mathbb{R}^m$ is assumed to be a piecewise smooth open and bounded set, and $\mathbb{P}$ be a subset of $\mathcal{P}(\mathcal{D})$. In our numerical application, $\mathcal{D}$ is a rectangle. Let $k \geq 1$ be an integer, $C_c^k(\mathbb{R}^m, \mathbb{R}^m)$ be the set of $k$-times continuously differentiable vector-valued functions with compact support. Let $L \subset \partial\mathcal{D}$ be the set of points where the normal $n$ is not defined, i.e. the set of singular points of $\partial\mathcal{D}$, such as the corners of a rectangle. Define

$$\Theta^k(\mathcal{D}) = \{\theta \in C_c^k(\mathbb{R}^m, \mathbb{R}^m) | \theta \cdot n_{|\partial\mathcal{D}\setminus L} = 0 \text{ and } \theta_{|L} = 0\}$$

equipped with the topology induced by $C_c^k(\mathbb{R}^m, \mathbb{R}^m)$. Consider a vector field $\theta \in \Theta^k(\mathcal{D})$ and the associated flow $T_t^\theta : \mathbb{R}^m \to \mathbb{R}^m$, $t \in [0, \tau]$ defined for each $x_0 \in \mathbb{R}^m$ as $T_t^\theta(x_0) := x(t)$, where $x : [0, \tau] \to \mathbb{R}$ solves

$$\dot{x}(t) = \theta(x(t)) \quad \text{for } t \in [0, \tau], \quad x(0) = x_0. \tag{1}$$

We use the simpler notation $T_t = T_t^\theta$ when no confusion is possible. Let $\Omega \in \mathcal{P}(\mathcal{D})$ and denote $n$ the outward unit normal vector to $\Omega$. We consider the family of perturbed domains

$$\Omega_t := T_t^\theta(\Omega). \tag{2}$$

The choice of $\Theta^k(\mathcal{D})$ guarantees that $T_t^\theta$ maps $\overline{\mathcal{D}}$ onto $\overline{\mathcal{D}}$, so that $\overline{\Omega_t} \subset \overline{\mathcal{D}}$; see [42, Theorem 2.16].

**Definition 1.** *Let $J : \mathbb{P} \to \mathbb{R}$ be a shape function.*

(i) *The Eulerian semiderivative of $J$ at $\Omega$ in direction $\theta \in \Theta^k(\mathcal{D})$, when the limit exists, is defined by*

$$dJ(\Omega; \theta) := \lim_{t \searrow 0} \frac{J(\Omega_t) - J(\Omega)}{t}. \tag{3}$$

(ii) *$J$ is shape differentiable at $\Omega$ if it has a Eulerian semiderivative at $\Omega$ for all $\theta \in \Theta^k(\mathcal{D})$ and the mapping*

$$dJ(\Omega) : \Theta^k(\mathcal{D}) \to \mathbb{R},\ \theta \mapsto dJ(\Omega; \theta)$$

*is linear and continuous, in which case $dJ(\Omega)$ is called the shape derivative at $\Omega$.*

When the shape derivative is computed as a volume integral, it is convenient to write it in the following particular form.

**Definition 2.** *Let $\Omega \in \mathbb{P}$ be open. A shape differentiable function $J$ admits a tensor representation of order $1$ if there exist tensors $S_l \in L^1(\mathcal{D}, \mathcal{L}^l(\mathbb{R}^m, \mathbb{R}^m))$, $l = 0, 1$, such that*

$$dJ(\Omega; \theta) = \int_{\mathcal{D}} S_1 : D\theta + S_0 \cdot \theta \, dx, \tag{4}$$

*for all $\theta \in \Theta^k(\mathcal{D})$. Here $\mathcal{L}^l(\mathbb{R}^m, \mathbb{R}^m)$ denotes the space of multilinear maps from $(\mathbb{R}^m)^l$ to $\mathbb{R}^m$.*

Expression (4) is called distributed, volumetric, or domain expression of the shape derivative. Under natural regularity assumptions, the shape derivative only depends on the restriction of the normal component $\theta \cdot n$ to the interface $\partial\Omega$. This fundamental result is known as the Hadamard-Zolésio structure theorem in shape optimization; see [17, pp. 480-481]. From the tensor representation (4), one immediately obtains such structure of the shape derivative as follows.

**Proposition 1.** *Let $\Omega \in \mathbb{P}$ and assume $\partial\Omega$ is $C^2$. Suppose that $dJ(\Omega)$ has the tensor representation* (4)*. If $S_l$, $l = 0, 1$ are of class $W^{1,1}$ in $\Omega$ and $\mathcal{D} \setminus \overline{\Omega}$, then we obtain the so-called boundary expression of the shape derivative:*

$$dJ(\Omega)(\theta) = \int_{\partial\Omega} g\, \theta \cdot n \, ds, \tag{5}$$

*with $g := [(S_1^+ - S_1^-)n] \cdot n$, where $+$ and $-$ denote the restrictions of the tensor to $\Omega$ and $\mathcal{D} \setminus \overline{\Omega}$, respectively.*

See [28] for a proof of Proposition 1 in a more general case. Usually the boundary expression (5) is used to devise level set-based numerical methods, but in this paper we present an alternative approach based on the volume expression (4), which allows a simple implementation. We use a Lagrangian approach to compute the tensor representation (4).

Further, we sometimes denote the distributed expression (4) by $dJ^{\mathtt{vol}}(\Omega; \theta)$, and the boundary expression (5) by $dJ^{\mathtt{surf}}(\Omega; \theta)$ when we compare them. Note that if the domain is $C^2$, Proposition 1 shows that

$$dJ^{\mathtt{vol}}(\Omega; \theta) = dJ^{\mathtt{surf}}(\Omega; \theta).$$

When $\Omega$ is less regular than $C^2$, it may happen that $dJ(\Omega)(\theta)$ cannot be written in the form (5). Note that even in this case, $dJ^{\mathtt{vol}}(\Omega; \theta)$ is a distribution with support on the boundary, even if written as a domain integral.

## 3. Shape derivatives in the framework of linear elasticity

3.1. **Shape derivative of the volume.** We introduce a parameterized domain $\Omega_t = T_t^\theta(\Omega)$ as in (2). We start with the simple case of the volume

$$\mathcal{V}(\Omega_t) := \int_{\Omega_t} 1 \, dx,$$

which is useful to become familiar with the computation of shape derivatives. Using the change of variable $x \mapsto T_t(x)$, we get

$$\mathcal{V}(\Omega_t) := \int_{\Omega} \xi(t) \, dx,$$

where $\xi(t) := |\det DT_t| = \det DT_t$ for $t$ small enough. We have $\xi'(0) = \frac{d}{dt} \det DT_t|_{t=0} = \operatorname{div} \theta$; see for instance [17, Theorem 4.1, pp. 182]. Thus the distributed expression of the shape derivative of the volume is given by

$$d\mathcal{V}^{\mathtt{vol}}(\Omega; \theta) = \int_{\Omega} \operatorname{div} \theta = \int_{\Omega} I_d : D\theta,$$

where $I_d$ is the identity matrix. We have obtained the distributed expression (4) of the shape derivative of the volume with $S_1 = I_d$ and $S_0 = 0$.

Applying Proposition 1, assuming $\partial\Omega$ is $C^2$, we get the usual boundary expression of the shape derivative

$$d\mathcal{V}^{\mathtt{surf}}(\Omega; \theta) = \int_{\partial\Omega} \theta \cdot n,$$

which, in this case, is the same as applying Stokes' theorem.

3.2. **The ersatz material approach.** We use the framework of the ersatz material, which is common in level set-based topology optimization of structures; see for instance [4, 48]. It is convenient as it allows to work on a fixed domain $\mathcal{D}$ instead of the variable domain $\Omega$, but also can create instability issues as pointed out in [15]. The idea of the ersatz material method is that the fixed domain $\mathcal{D}$ is filled with two homogeneous materials with different Hooke elasticity tensors $A_0$ and $A_1$ defined by

$$A_i \xi = 2\mu_i \xi + \lambda_i (\mathrm{Tr}\xi) I_d, \quad i = 0, 1.$$

with Lamé moduli $\lambda_i$ and $\mu_i$ for $i = 0, 1$, $I_d$ is the identity matrix and $\xi$ is a matrix. The first material lays in the open subset $\Omega$ of $\mathcal{D}$ and the background material fills the complement so that Hooke's law is written in $\mathcal{D}$ as

$$A_\Omega = A_0 \chi_\Omega + \epsilon A_0 \chi_{\mathcal{D}\setminus\Omega}, \tag{6}$$

where $\chi_\Omega$ denotes the indicator function of $\Omega$, and $\epsilon$ is a given small parameter. Hence the region $\mathcal{D} \setminus \Omega$ represents a "weak phase" whereas $\Omega$ is the "strong phase". The optimization is still performed with respect to the variable set $\Omega$, but here $\Omega$ is embedded in the fixed, larger set $\mathcal{D}$.

Note that we compute the shape derivative for the PDE including the ersatz material, unlike what is usually done in the literature; see Section 3.7 for a more detailed discussion of this point.

Let $\Omega \subset \mathcal{D} \subset \mathbb{R}^m$, $m = 2, 3$, where $\mathcal{D}$ is a fixed domain whose boundary $\partial\mathcal{D}$ is partitioned into four subsets $\Gamma_d$, $\Gamma_n$, $\Gamma_s$ and $\Gamma$. A homogeneous Dirichlet (respectively Neumann) boundary condition is imposed on $\Gamma_d$ (resp. $\Gamma$). On $\Gamma_n$, a non-homogeneous Neumann condition is imposed, which represents a given surface load $g \in H^{-1/2}(\Gamma_n)^m$. The free interface between the weak and strong phase is $\partial\Omega$. A spring with stiffness $k_s$ is attached on the boundary $\Gamma_s$, which corresponds to a Robin boundary condition; this condition is used for mechanisms. Let $H^1_d(\mathcal{D})^m$ be the space of vector fields in $H^1(\mathcal{D})^m$ which satisfy the homogeneous Dirichlet boundary conditions on $\Gamma_d$.

We define a parameterized domain $\Omega_t = T_t^\theta(\Omega)$ as in (2), and we assume additionally that $T_t^\theta = \mathrm{id}$ on $\Gamma_d \cup \Gamma_n \cup \Gamma_s \cup \Gamma_m$, where id is the identity.

In the ersatz material approach, the displacement field $u \in H^1_d(\mathcal{D})^m$ is the solution of the linearized elasticity system

$$-\operatorname{div} A_\Omega e(u) = 0 \text{ in } \mathcal{D}, \tag{7}$$

$$u = 0 \text{ on } \Gamma_d, \tag{8}$$

$$A_\Omega e(u)n = g \text{ on } \Gamma_n, \tag{9}$$

$$A_\Omega e(u)n = 0 \text{ on } \Gamma, \tag{10}$$

$$A_\Omega e(u)n = -k_s u \text{ on } \Gamma_s, \tag{11}$$

where the symmetrized gradient is $e(u) = (Du + Du^\mathsf{T})/2$, and $Du^\mathsf{T}$ denotes the transpose of $Du$. We consider the following functional

$$J(\Omega) := c_1 \int_\mathcal{D} A_\Omega e(u(x)) : e(u(x))\, dx + c_2 \int_\mathcal{D} F_\mathcal{D}(x, u(x))\, dx + c_3 \int_{\Gamma_m} F_\Gamma(x, u(x))\, ds_x. \tag{12}$$

We assume that $F_\mathcal{D}$ and $F_\Gamma$ are smooth functions of $u$, that $F_\mathcal{D}$ is $C^1$ with respect to the first argument, and $\Gamma_m \subset \partial\mathcal{D}$. The set $\Gamma_m \subset \partial\mathcal{D}$ is a region where the shape displacements are monitored and is used for mechanisms only; it is set to $\Gamma_m = \emptyset$ in other cases. This general functional covers several important cases such as the compliance and certain functionals used for compliant mechanisms. The case $(c_1, c_2, c_3) = (1, 0, 0)$ corresponds to the compliance. The case of compliant mechanisms may be achieved by an appropriate choice of $F_\mathcal{D}$ and $F_\Gamma$, see Section 3.6.

We denote by $u_t$ the solution of (7)-(11) with $\Omega$ substituted by $\Omega_t$. Defining $u^t := u_t \circ T_t$ and using the chain rule we have the relation

$$Du^t = D(u_t \circ T_t) = (Du_t) \circ T_t\, DT_t, \tag{13}$$

and consequently

$$E(t, u^t) := e(u_t) \circ T_t = (Du^t DT_t^{-1} + DT_t^{-\mathsf{T}}(Du^t)^\mathsf{T})/2. \tag{14}$$

The variational formulation of the PDE is to find $u_t \in H^1_d(\mathcal{D})^m$ such that

$$\int_\mathcal{D} A_{\Omega_t} e(u_t) : e(v_t) + \int_{\Gamma_s} k_s u_t \cdot v_t = \int_{\Gamma_n} g \cdot v_t, \tag{15}$$

for all $v_t \in H^1_d(\mathcal{D})^m$. We proceed with the change of variable $x \mapsto T_t(x)$ in (15) which yields

$$\int_\mathcal{D} [A_{\Omega_t} e(u_t)] \circ T_t : e(v_t) \circ T_t\, \xi(t) + \int_{\Gamma_s} k_s u_t \cdot v_t = \int_{\Gamma_n} g \cdot v_t, \text{ for all } v_t \in H^1_d(\mathcal{D})^m, \tag{16}$$

where $\xi(t) := |\det DT_t|$ is the Jacobian of the transformation $x \mapsto T_t(x)$. Note that neither the Jacobian nor $T_t$ needs to appear in the integrals on $\Gamma_n$ and $\Gamma_s$, since we have assumed $T_t = \mathrm{id}$ on $\Gamma_d \cup \Gamma_n \cup \Gamma_s \cup \Gamma_m$. In view of (14), we can rewrite (16) as

$$\int_\mathcal{D} A_\Omega E(t, u^t) : E(t, v)\, \xi(t) + \int_{\Gamma_s} k_s u^t \cdot v = \int_{\Gamma_n} g \cdot v, \tag{17}$$

for all $v \in H^1_d(\mathcal{D})^m$. In a similar way, we have

$$J(\Omega_t) = c_1 \int_\mathcal{D} A_{\Omega_t} e(u_t) : e(u_t) + c_2 \int_\mathcal{D} F_\mathcal{D}(x, u_t(x))\, dx + c_3 \int_{\Gamma_m} F_\Gamma(x, u_t(x))\, ds_x,$$

and using the change of variable $x \mapsto T_t(x)$ yields

$$
\begin{aligned}
J(\Omega_t) = c_1 \int_{\mathcal{D}} A_\Omega E(t,u^t) : E(t,u^t)\, \xi(t) + c_2 \int_{\mathcal{D}} F_{\mathcal{D}}(T_t(x), u^t(x)) \xi(t)(x)\, dx \\
+ c_3 \int_{\Gamma_m} F_\Gamma(x, u^t(x))\, ds_x.
\end{aligned} \tag{18}
$$

3.3. **Shape derivative using a Lagrangian approach.** To compute the shape derivative of $J(\Omega)$, we use the averaged adjoint method, a Lagrangian-type method introduced in [43]. Formally, the Lagrangian $G$ is obtained by summing the expression (18) of the cost functional and the variational formulation (17) of the PDE constraint, and $(u^t, v)$ is replaced with the variables $(\varphi, \psi)$; see [28, 43] for a rigorous mathematical presentation and detailed explanations. Writing $A$ instead of $A_\Omega$ for simplicity, this yields

$$
\begin{aligned}
G(t,\varphi,\psi) :=& c_1 \int_{\mathcal{D}} AE(t,\varphi) : E(t,\varphi)\, \xi(t) + c_2 \int_{\mathcal{D}} F_{\mathcal{D}}(T_t(x), \varphi(x)) \xi(t)(x)\, dx + c_3 \int_{\Gamma_m} F_\Gamma(x, \varphi(x))\, ds_x \\
&+ \int_{\mathcal{D}} AE(t,\varphi) : E(t,\psi)\, \xi(t) + \int_{\Gamma_s} k_s \varphi \cdot \psi - \int_{\Gamma_n} g \cdot \psi.
\end{aligned}
$$

In view of (17) and (18), we have $J(\Omega_t) = G(t, u^t, \psi)$ for all $\psi \in H^1_d(\mathcal{D})^m$. Thus the shape derivative can be computed as

$$
dJ(\Omega;\theta) = \frac{d}{dt}(G(t,u^t,\psi))|_{t=0}. \tag{19}
$$

The advantage of the Lagrangian is that, under suitable assumptions, one can show that

$$
\frac{d}{dt}(G(t,u^t,\psi))|_{t=0} = \partial_t G(0,u^0,p^0). \tag{20}
$$

which essentially means that it is not necessary to compute the derivative of $u^t$ to compute $dJ(\Omega;\theta)$. In this paper we assume for simplicity that (20) is true for the problem under consideration, but note that this result can be made mathematically rigorous using the averaged adjoint method; see [23, 28, 43].

The adjoint is given as the solution of the following first-order optimality condition

$$
\partial_\varphi G(0,u,p)(\hat{\varphi}) = 0 \text{ for all } \hat{\varphi} \in H^1_d(\mathcal{D})^m,
$$

which yields, using $A = A^{\mathsf{T}}$,

$$
\begin{aligned}
&2c_1 \int_{\mathcal{D}} AE(0,u) : E(0,\hat{\varphi}) + c_2 \int_{\mathcal{D}} \partial_u F_{\mathcal{D}}(x,u(x)) \cdot \hat{\varphi}(x)\, dx + c_3 \int_{\Gamma_m} \partial_u F_\Gamma(x,u(x)) \cdot \hat{\varphi}(x)\, ds_x \\
&+ \int_{\Gamma_s} k_s \hat{\varphi} \cdot p + \int_{\mathcal{D}} AE(0,\hat{\varphi}) : E(0,p) = 0, \text{ for all } \hat{\varphi} \in H^1_d(\mathcal{D})^m.
\end{aligned}
$$

Since $E(0,v) = e(v)$ for $v \in H^1_d(\mathcal{D})^m$, and $A = A^{\mathsf{T}}$, we get the adjoint equation

$$
\begin{aligned}
&\int_{\mathcal{D}} Ae(p) : e(\hat{\varphi}) + \int_{\Gamma_s} k_s p \cdot \hat{\varphi} = -2c_1 \int_{\mathcal{D}} Ae(u) : e(\hat{\varphi}) - c_2 \int_{\mathcal{D}} \partial_u F_{\mathcal{D}}(x,u(x)) \cdot \hat{\varphi}(x)\, dx \\
&- c_3 \int_{\Gamma_m} \partial_u F_\Gamma(x,u(x)) \cdot \hat{\varphi}(x)\, ds_x \text{ for all } \hat{\varphi} \in H^1_d(\mathcal{D})^m.
\end{aligned} \tag{21}
$$

Using (19) and (20), we obtain

$$
\begin{aligned}
dJ(\Omega;\theta) =& c_1 \int_{\mathcal{D}} A\partial_t E(0,u) : E(0,u) + c_1 \int_{\mathcal{D}} AE(0,u) : \partial_t E(0,u) + AE(0,u) : E(0,u) \operatorname{div} \theta \\
&+ c_2 \int_{\mathcal{D}} \partial_x F_{\mathcal{D}}(x,u(x)) \cdot \theta(x) + F_{\mathcal{D}}(x,u(x)) \operatorname{div} \theta(x)\, dx \\
&+ \int_{\mathcal{D}} A\partial_t E(0,u) : E(0,p) + \int_{\mathcal{D}} AE(0,u) : \partial_t E(0,p) + AE(0,u) : E(0,p) \operatorname{div} \theta
\end{aligned}
$$

We also compute, using $E(0,v) = e(v)$ for $v \in H^1_d(\mathcal{D})^m$,

$$
\partial_t E(0,v) = (-DvD\theta - D\theta^{\mathsf{T}} Dv^{\mathsf{T}})/2.
$$

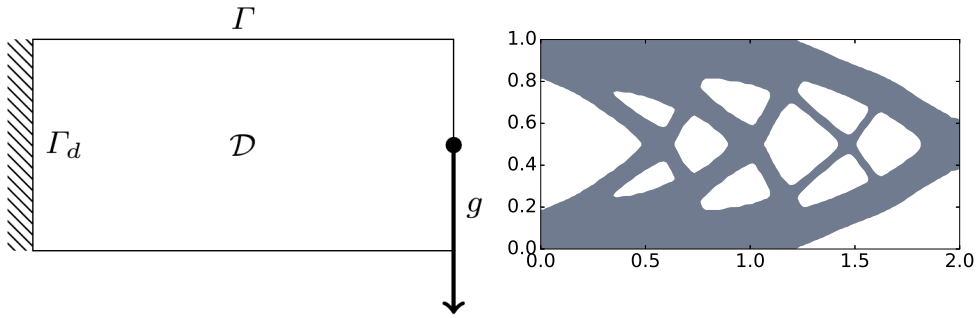



FIGURE 1. Design domain of cantilever (left) and example of optimal design (right).

Using $A = A^\mathsf{T}$ we obtain

$$\begin{aligned} dJ(\Omega;\theta) = &- \int_\mathcal{D} \frac{1}{2} Du^\mathsf{T}(Ae(p) + (Ae(p))^\mathsf{T}) : D\theta - \int_\mathcal{D} \frac{1}{2} Dp^\mathsf{T}(Ae(u) + (Ae(u))^\mathsf{T}) : D\theta \\ &+ \int_\mathcal{D} (Ae(u) : e(p) + c_1 Ae(u) : e(u)) \operatorname{div}\theta - c_1 \int_\mathcal{D} (Du^\mathsf{T} Ae(u) + Du^\mathsf{T}(Ae(u))^\mathsf{T}) : D\theta \\ &+ c_2 \int_\mathcal{D} \partial_x F_\mathcal{D}(x, u(x)) \cdot \theta(x) + F_\mathcal{D}(x, u(x)) \operatorname{div}\theta(x)\, dx. \end{aligned}$$

Using $(Ae(v))^\mathsf{T} = Ae(v)$ we get

$$dJ^{\mathtt{vol}}(\Omega;\theta) = \int_\mathcal{D} S_1 : D\theta + S_0 \cdot \theta, \tag{22}$$

with

$$\begin{aligned} S_1 = &- Du^\mathsf{T} A_\Omega e(p) - Dp^\mathsf{T} A_\Omega e(u) - 2c_1 Du^\mathsf{T} A_\Omega e(u) \\ &+ (A_\Omega e(u) : e(p) + c_1 A_\Omega e(u) : e(u) + c_2 F_\mathcal{D}(\cdot, u)) I_d, \end{aligned} \tag{23}$$

$$S_0 = c_2 \partial_x F_\mathcal{D}(\cdot, u). \tag{24}$$

Formula (22) is convenient for the numerics as it can be implemented in a straightforward way in FEniCS.

3.4. **Compliance.** The case of the compliance is obtained by setting $(c_1, c_2, c_3) = (1, 0, 0)$. This yields $S_0 \equiv 0$, $p = -2u$ and (23) becomes

$$S_1 = 2Du^\mathsf{T} A_\Omega e(u) - A_\Omega e(u) : e(u) I_d. \tag{25}$$

See Figure 1 for an example of design domain, boundary conditions, and optimal design for minimization of the compliance.

3.5. **Multiple load cases.** For multiple load cases and compliance, we consider the set of forces $\{g_i\}_{i\in I}$ and the compliance is the sum of the compliances associated to each force $g_i$:

$$J(\Omega) := \sum_{i\in I} \int_{\Gamma_n} g_i \cdot u_i,$$

where $u_i$ is the solution of the linearized elasticity system corresponding to $g_i$. The shape derivative is in this case

$$dJ^{\mathtt{vol}}(\Omega;\theta) = \int_\mathcal{D} S_1 : D\theta, \tag{26}$$

with $S_1 := \sum_{i\in I} 2Du_i^\mathsf{T} A_\Omega e(u_i) - A_\Omega e(u_i) : e(u_i) I_d$.

3.6. **Inverter mechanism.** The displacement inverter converts an input displacement on the left edge to a displacement in the opposite direction on the right edge; see [9] for a detailed description. We take $\mathcal{D} = (0,1)^2$, and an actuation force $g = (g_x, 0)$, $g_x > 0$, is applied at the input point $(0, 0.5)$. We define the output boundary $\Gamma_{out}$ and input boundary $\Gamma_{in}$ such that $\Gamma_m = \Gamma_{out} \cup \Gamma_{in}$, $\Gamma_{in} = \{0\} \times (a_0, a_1)$, $\Gamma_{out} = \Gamma_s$ and $\Gamma_s = \{1\} \times (b_0, b_1)$. An artificial spring with stiffness $k_s > 0$ is attached at the output $\Gamma_{out}$ to simulate the resistance of a workpiece. In order to maximize output displacement, while limiting the input displacement, we minimize (12) with $(c_1, c_2, c_3) = (0, 0, 1)$ and

$$F_\Gamma(x, u(x)) = \eta_{in} u_1(x) \chi_{\Gamma_{in}}(x) + \eta_{out} u_1(x) \chi_{\Gamma_{out}}(x),$$

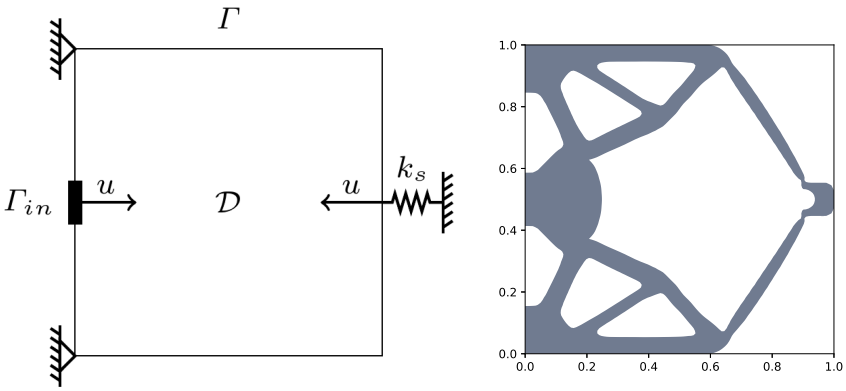



FIGURE 2. Design domain of inverter (left) and optimal design (right) for $(Nx, Ny) = (121, 121)$.

with $u = (u_1, u_2)$, $\eta_{in}, \eta_{out}$ some positive constants. Note that $u_1 > 0$ on $\Gamma_{in}$ and $u_1 < 0$ on $\Gamma_{out}$. We obtain $S_0 \equiv 0$ and (23) becomes

$$S_1 = -Du^{\mathsf{T}} A_\Omega e(p) - Dp^{\mathsf{T}} A_\Omega e(u) + A_\Omega e(u) : e(p) I_d.$$

For this functional the problem is not self-adjoint and in view of (21), the adjoint $p$ is the solution of

$$\int_{\mathcal{D}} Ae(p) : e(\hat{\varphi}) + \int_{\Gamma_s} k_s p \cdot \hat{\varphi} = -\eta_{in} \int_{\Gamma_{in}} \hat{\varphi}_1 - \eta_{out} \int_{\Gamma_{out}} \hat{\varphi}_1, \quad \forall \hat{\varphi} \in H^1_d(\mathcal{D})^m. \tag{27}$$

where $\hat{\varphi} = (\hat{\varphi}_1, \hat{\varphi}_2)$. See Figure 2 for an example of design domain, boundary conditions and optimal design for the inverter.

3.7. **Comparison of shape derivatives with and without ersatz material.** Usually the boundary expression of the shape derivative is used in level set methods, and computed for the problem without ersatz material, although the elasticity system is solved using the ersatz material in the numerics. This small mismatch is justified by the fact that the tensor of the ersatz material has a small amplitude. The reason why this mismatch is tolerated in the numerics is probably because the boundary expression of the shape derivative in that case is unpractical to handle numerically, as it requires to use the jump of the gradient across the moving interface between the strong and the weak phases; see (37).

In any case, it is more precise to use the proper shape derivative corresponding to the ersatz material framework for the numerics, in order to avoid this mismatch. Another advantage of using the exact formula for the ersatz approach is that this formula is actually valid for any value of $\epsilon$, and not only for $\epsilon$ small. This can be used for a mixture of two materials for instance. We show in this section that computing and implementing the formula of the distributed shape derivative is not more difficult for the ersatz material approach.

First we compare the distributed shape derivative without ersatz material. The elasticity system is in this case

$$-\operatorname{div} Ae(u) = 0 \text{ in } \Omega, \tag{28}$$

$$u = 0 \text{ on } \Gamma_d, \tag{29}$$

$$Ae(u)n = g \text{ on } \Gamma_n, \tag{30}$$

$$Ae(u)n = 0 \text{ on } \Gamma, \tag{31}$$

$$Ae(u)n = -k_s u \text{ on } \Gamma_s. \tag{32}$$

As in Section 3.2, $\Gamma_d$, $\Gamma_n$ and $\Gamma_s$ are fixed, but in this case $\Gamma = \partial\Omega \setminus (\Gamma_n \cup \Gamma_d \cup \Gamma_s)$ is the free boundary. We also assume that the interface between $\Gamma$ and the fixed boundaries is fixed. The Hooke elasticity tensor $A$ satisfies $A\xi = 2\mu\xi + \lambda \operatorname{tr}(\xi) I_d$, where $\xi \in \mathbb{R}^{m\times m}$, $\mu, \lambda$ are the Lamé parameters. The variational formulation of (28)-(32) consists in finding $u \in H^1_d(\Omega)^m$ such that

$$\int_\Omega Ae(u) : e(v) + \int_{\Gamma_s} k_s u \cdot v = \int_{\Gamma_n} g \cdot v \text{ for all } v \in H^1_d(\Omega)^m.$$

The cost functional is in this case

$$J_0(\Omega_t) := c_1 \int_\Omega Ae(u) : e(u) + c_2 \int_\Omega F_{\mathcal{D}}(x, u(x))\, dx + c_3 \int_{\Gamma_m} F_\Gamma(x, u(x))\, ds_x.$$

A similar calculation as in Section 3.3 yields

$$dJ_0^{\mathtt{vol}}(\Omega;\theta) = \int_\Omega S_1 : D\theta + S_0 \cdot \theta,$$

with

$$S_1 = -Du^\mathsf{T} Ae(p) - Dp^\mathsf{T} Ae(u) - 2c_1 Du^\mathsf{T} Ae(u) + (Ae(u) : e(p) + c_1 Ae(u) : e(u) + c_2 F_{\mathcal{D}}(\cdot,u))I_d, \tag{33}$$

$$S_0 = c_2 \partial_x F_{\mathcal{D}}(\cdot,u). \tag{34}$$

In the case $(c_1,c_2,c_3) = (1,0,0)$, which corresponds to the compliance, we have $S_0 \equiv 0$ and $p = -2u$, yielding

$$S_1 := 2Du^\mathsf{T} Ae(u) - Ae(u) : e(u) I_d \quad \text{ in } \Omega. \tag{35}$$

A similar formula can be found in [32, Section 2.5], for a slightly different case, and where $S_1$ is identified as the energy-momentum tensor in continuum mechanics introduced by Eshelby in [19]. Compare also (35) with the shape derivative in [44, Theorem 3.3], also in the framework of linearized elasticity but for a different functional.

Note that (35) is similar to (25), the main difference being that (25) is defined in $\mathcal{D}$ and (35) is defined in $\Omega$. Thus from a numerical point of view, (25) is not more difficult to implement than (35).

Now we compare the boundary expression of the shape derivatives with and without ersatz material, in the case of the compliance. Assuming $\Omega$ is $C^2$ and using (35) and (5) of Proposition 1, we obtain the boundary expression of the shape derivative

$$dJ_0^{\mathtt{surf}}(\Omega;\theta) = \int_{\partial\Omega} (S_1 n \cdot n)\theta \cdot n = \int_\Gamma (S_1 n \cdot n)\theta \cdot n,$$

since $\partial\Omega \setminus \Gamma$ is fixed. Then we compute

$$S_1 n \cdot n = 2Du^\mathsf{T} Ae(u) n \cdot n - Ae(u) : e(u) = 2Ae(u)n \cdot Dun - Ae(u) : e(u).$$

On $\Gamma$, we have $Ae(u)n = 0$ which yields

$$dJ_0^{\mathtt{surf}}(\Omega;\theta) = \int_\Gamma -Ae(u) : e(u)\theta \cdot n, \tag{36}$$

which is a particular case of the formula in [4, Theorem 7].

In the case of the ersatz material, applying Proposition 1 and assuming $\partial\Omega$ is $C^2$, the distributed expression (22) yields the boundary expression

$$dJ^{\mathtt{surf}}(\Omega;\theta) = \int_{\partial\Omega} [(S_1^+ - S_1^-)n] \cdot n\, \theta \cdot n,$$

with

$$[(S_1^+ - S_1^-)n] \cdot n = -[\![Ae(u) : e(u)]\!] + 2A_0 e(u)^+ n \cdot Du^+ n - 2\epsilon A_0 e(u)^- n \cdot Du^- n,$$

and where the exponents $(\cdot)^+$ and $(\cdot)^-$ denote the restrictions to $\Omega$ and $\mathcal{D}\setminus\Omega$, respectively. Also

$$[\![v]\!] := \gamma_\Omega(v) - \gamma_{\mathcal{D}\setminus\Omega}(v)$$

denotes the jump of a function $v$ across the interface $\partial\Omega$; here $\gamma_\Omega(v)$ is the trace of $v_{|\Omega}$ on $\partial\Omega$. Using the transmission condition $A_0 e(u)^+ n = \epsilon A_0 e(u)^- n$ on $\partial\Omega$, we obtain

$$dJ^{\mathtt{surf}}(\Omega;\theta) = \int_{\partial\Omega} (2\epsilon A_0 e(u^-)n \cdot [\![Du]\!]n)\theta \cdot n - \int_{\partial\Omega} [\![Ae(u) : e(u)]\!]\theta \cdot n. \tag{37}$$

The two main differences between (37) and (36) are the small perturbation term $(2\epsilon A_0 e(u^-)n \cdot [\![Du]\!]n$ and the fact that $[\![Ae(u) : e(u)]\!]$ is a jump across the interface $\partial\Omega$. We observe that (36) is easier to implement than (37) in a numerical method.

## 4. Descent direction

For the numerical method we need a descent direction $\theta$, i.e. a vector field satisfying $dJ(\Omega;\theta) < 0$. When $dJ(\Omega;\theta)$ is written using the boundary expression

$$dJ^{\mathtt{surf}}(\Omega;\theta) = \int_{\partial\Omega} G(\Omega)\, \theta \cdot n,$$

then a simple choice is to take $\theta = -G(\Omega) \cdot n$. However, this choice assumes that $G(\Omega)$ and $\partial\Omega$ are quite regular, and in practice this may yield a $\theta$ with a poor regularity and lead to an unstable behaviour of the algorithm such as

irregular or oscillating boundaries. A better choice is to find a smoother descent direction by finding $\theta \in \mathbb{H}(\partial\Omega)$ such that

$$\mathcal{B}(\theta, \xi) = -dJ^{\mathtt{surf}}(\Omega; \xi) \text{ for all } \xi \in \mathbb{H}(\partial\Omega), \tag{38}$$

where $\mathbb{H}(\partial\Omega)$ is an appropriate Sobolev space of vector fields on $\partial\Omega$ and $\mathcal{B} : \mathbb{H}(\partial\Omega) \times \mathbb{H}(\partial\Omega) \to \mathbb{R}$, $k \geq 1$, is a positive definite bilinear form on $\partial\Omega$.

In the case of the present paper we use the distributed expression (22) of the shape derivative, therefore we use a positive definite bilinear form $\mathcal{B} : \mathbb{H}(\mathcal{D}) \times \mathbb{H}(\mathcal{D}) \to \mathbb{R}$, where $\mathbb{H}(\mathcal{D})$ is an appropriate Sobolev space of vector fields on $\mathcal{D}$. Thus the problem is to find $\theta \in \mathbb{H}(\mathcal{D})$ such that

$$\mathcal{B}(\theta, \xi) = -dJ^{\mathtt{vol}}(\Omega; \xi) \text{ for all } \xi \in \mathbb{H}(\mathcal{D}), \tag{39}$$

With this choice, the solution $\theta$ of (39) is defined on all of $\mathcal{D}$ and is a descent direction since $dJ(\Omega; \theta) = -\mathcal{B}(\theta, \theta) < 0$ if $\theta \neq 0$.

It is also possible to combine the two approaches by substituting $dJ^{\mathtt{vol}}(\Omega; \theta)$ with $dJ^{\mathtt{surf}}(\Omega; \theta)$ in (39). This was done in [16] where a strong improvement of the rate of convergence of the level-set method was observed; see also [12] for a thorough discussion of various possibilities for $\mathcal{B}$. Bilinear forms defined on $\mathcal{D}$ are useful for the level set method which requires $\theta$ on $\mathcal{D}$; see Section (5).

In our algorithm we choose $\mathbb{H}(\mathcal{D}) = H^1(\mathcal{D})^m$ and

$$\mathcal{B}(\theta, \xi) = \int_{\mathcal{D}} \alpha_1 D\theta : D\xi + \alpha_2 \theta \cdot \xi, \tag{40}$$

with $\alpha_1 = 1$ and $\alpha_2 = 0.1$. We also take the boundary conditions $\theta \cdot n = 0$ on $\partial\mathcal{D}$; see Section 6.11.

## 5. Level set method

The level set method, originally introduced in [34], gives a general framework for the computation of evolving interfaces using an implicit representation of these interfaces. We refer to the monographs [33, 39] for a complete description of the level set method. The core idea of this method is to represent the boundary of the moving domain $\Omega_t \subset \mathcal{D} \in \mathbb{R}^N$ as the zero level set of a continuous function $\phi(\cdot, t) : \mathcal{D} \to \mathbb{R}$.

Let us consider the family of domains $\Omega_t \subset \mathcal{D}$ as defined in (2). Each domain $\Omega_t$ can be defined as

$$\Omega_t := \{x \in \mathcal{D},\ \phi(x, t) < 0\}, \tag{41}$$

where $\phi : \mathcal{D} \times \mathbb{R}^+ \to \mathbb{R}$ is Lipschitz continuous and called *level set function*. Indeed, if we assume $|\nabla\phi(\cdot, t)| \neq 0$ on the set $\{x \in \mathcal{D},\ \phi(x, t) = 0\}$, then we have

$$\partial\Omega_t = \{x \in \mathcal{D},\ \phi(x, t) = 0\}, \tag{42}$$

i.e. the boundary $\partial\Omega_t$ is the zero level set of $\phi(\cdot, t)$.

Let $x(t)$ be the position of a moving boundary point of $\partial\Omega_t$, with velocity $\dot{x}(t) = \theta(x(t))$ according to (1). Differentiating the relation $\phi(x(t), t) = 0$ with respect to $t$ yields the Hamilton-Jacobi equation:

$$\partial_t\phi(x(t), t) + \theta(x(t)) \cdot \nabla\phi(x(t), t) = 0 \quad \text{ in } \partial\Omega_t \times \mathbb{R}^+,$$

which is then extended to all of $\mathcal{D}$ via the equation

$$\partial_t\phi(x, t) + \theta(x) \cdot \nabla\phi(x, t) = 0 \quad \text{ in } \mathcal{D} \times \mathbb{R}^+, \tag{43}$$

or alternatively to $U \times \mathbb{R}^+$ where $U$ is a neighbourhood of $\partial\Omega_t$.

When $\theta = \vartheta_n n$ is a normal vector field on $\partial\Omega_t$, noting that an extension to $\mathcal{D}$ of the unit outward normal vector $n$ to $\Omega_t$ is given by $\nabla\phi/|\nabla\phi|$, and extending $\vartheta_n$ to all of $\mathcal{D}$, one obtains from (43) the level set equation

$$\partial_t\phi + \vartheta_n|\nabla\phi| = 0 \quad \text{ in } \mathcal{D} \times \mathbb{R}^+. \tag{44}$$

The initial data $\phi(x, 0) = \phi_0(x)$ accompanying the Hamilton-Jacobi equation (43) or (44) can be chosen as the signed distance function to the initial boundary $\partial\Omega_0$ in order to satisfy the condition $|\nabla u| \neq 0$ on $\partial\Omega$, i.e.

$$\phi_0(x) = \begin{cases} d(x, \partial\Omega_0), & \text{if } x \in (\Omega_0)^c, \\ -d(x, \partial\Omega_0), & \text{if } x \in \Omega_0. \end{cases} \tag{45}$$

The fast marching method [39] and the fast sweeping method [50] are efficient methods to compute the signed distance function.

5.1. **Level set method and volume expression of the shape derivative.** In the case of the distributed shape derivative (22), we do not extend $\vartheta_n$ to $\mathcal{D}$, instead we obtain directly a descent direction $\theta$ defined in $\mathcal{D}$ by solving (39), where $dJ^{\mathtt{vol}}(\Omega;\theta)$ is given by (22). Thus, unlike the usual level set method, $\theta$ is not necessarily normal to $\partial\Omega_t$ and $\phi$ is not governed by (44) but rather by the Hamilton-Jacobi equation (43).

In shape optimization, $\vartheta_n$ usually depends on the solution of one or several PDEs and their gradient. Since the boundary $\partial\Omega_t$ in general does not match the grid nodes where $\phi$ and the solutions of the partial differential equations are defined in the numerical application, the computation and extension of $\vartheta_n$ may require the interpolation on $\partial\Omega_t$ of functions defined at the grid points only, complicating the numerical implementation and introducing an additional interpolation error. This is an issue in particular for interface problems, such as the problem of elasticity with ersatz material studied in this paper, where $\vartheta_n$ is the jump of a function across the interface, as in (37), which requires several interpolations and is error-prone. In the distributed shape derivative framework, $\theta$ only needs to be defined at grid nodes.

5.2. **Discretization of the Hamilton-Jacobi equation.** Let $\mathcal{D} = (0,1)\times(0,1)$ to simplify the presentation. For the discretization of the Hamilton-Jacobi equation (43), we first define the mesh grid corresponding to $\mathcal{D}$. We introduce the nodes $P_{ij}$ whose coordinates are given by $(i\Delta x, j\Delta y)$, $1 \le i,j \le N$ where $\Delta x$ and $\Delta y$ are the steps of the discretization in the $x$ and $y$ directions, respectively. Let us write $t^k = k\Delta t$ for the discrete time, with $k \in \mathbb{N}$ and $\Delta t$ is the time step. Denote the approximation $\phi_{ij}^k \simeq \phi(P_{ij}, t^k)$.

In the usual level set method, equation (44) is discretized using an explicit upwind scheme proposed by Osher and Sethian [33, 34, 39]. This scheme applies to the specific form (44) but is not suited to discretize (43) required for our application. Equation (43) is of the form

$$\partial_t \phi + H(\nabla\phi) = 0 \quad \text{ in } \mathcal{D}\times\mathbb{R}^+, \tag{46}$$

where $H(\nabla\phi) := \theta\cdot\nabla\phi$ is the so-called Hamiltonian. We use a Lax-Friedrichs flux, see [35], which writes in our case:

$$\hat{H}^{LF}(p^-,p^+,q^-,q^+) = H\left(\frac{p^-+p^+}{2}, \frac{q^-+q^+}{2}\right) - \frac{1}{2}(p^+-p^-)\alpha^x - \frac{1}{2}(p^+-p^-)\alpha^y,$$

where $\alpha^x = |\theta_x|$, $\alpha^y = |\theta_y|$, $\theta = (\theta_x,\theta_y)$ and

$$\begin{aligned} p^- = D_x^-\phi_{ij} = \frac{\phi_{ij}-\phi_{i-1,j}}{\Delta x}, \qquad p^+ = D_x^+\phi_{ij} = \frac{\phi_{i+1,j}-\phi_{ij}}{\Delta x},\\ q^- = D_y^-\phi_{ij} = \frac{\phi_{ij}-\phi_{i,j-1}}{\Delta y}, \qquad q^+ = D_y^+\phi_{ij} = \frac{\phi_{i,j+1}-\phi_{ij}}{\Delta y}, \end{aligned} \tag{47}$$

are the backward and forward approximations of the $x$-derivative and $y$-derivative of $\phi$ at $P_{ij}$, respectively. Using a forward Euler time discretization, the numerical scheme corresponding to (43) is

$$\phi_{ij}^{k+1} = \phi_{ij}^k - \Delta t\, \hat{H}^{LF}(p^-,p^+,q^-,q^+) \tag{48}$$

where $p^-,p^+,q^-,q^+$ are computed for $\phi_{ij}^k$.

5.3. **Reinitialization.** For numerical accuracy, the solution of the level set equation (43) should not be too flat or too steep. This is fulfilled for instance if $\phi$ is the distance function i.e. $|\nabla\phi| = 1$. Even if one initializes $\phi$ using a signed distance function, the solution $\phi$ of the level set equation (43) does not generally remain close to a distance function, thus we regularly perform a reinitialization of $\phi$; see [14].

We present here briefly the procedure for the reinitialization introduced in [38]. The reinitialization at time $t$ is performed by solving to steady state the following Hamilton-Jacobi type equation

$$\begin{aligned} \partial_\tau\varphi + S(\phi)(|\nabla\varphi|-1) = 0 \text{ in } \mathcal{D}\times\mathbb{R}^+,\\ \varphi(x,0) = \phi(x,t),\ x\in\mathcal{D}, \end{aligned}$$

where $S(\phi)$ is an approximation of the sign function

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + |\nabla\phi|^2\epsilon_s^2}}, \tag{49}$$

with $\epsilon_s = \min(\Delta x,\Delta y)$.

For the discretization we use the standard explicit upwind scheme; see [33, 34, 39],

$$\varphi_{ij}^{k+1} = \varphi_{ij}^k - \Delta t\, K(p^-,p^+,q^-,q^+), \tag{50}$$

where

$$K(p^-,p^+,q^-,q^+) = \max(S(\phi_{ij}),0)K^+ + \min(S(\phi_{ij}),0)K^-, \tag{51}$$

and

$$K^+ = \left[\max(p^-,0)^2 + \min(p^+,0)^2 + \max(q^-,0)^2 + \min(q^+,0)^2\right]^{1/2}, \tag{52}$$

$$K^- = \left[\min(p^-,0)^2 + \max(p^+,0)^2 + \min(q^-,0)^2 + \max(q^+,0)^2\right]^{1/2}, \tag{53}$$

and where $p^-, p^+, q^-, q^+$ are computed for $\phi_{ij}^k$ using (47).

## 6. Implementation

In this section we explain the implementation step by step. The code presented in this paper has been written for FEniCS 2017.1, and is compatible with FEniCS 2016.2. With a small number of modifications, the code may also run with earlier versions of FEniCS. The code can be downloaded at `http://antoinelaurain.com/compliance.htm`.

6.1. **Introduction.** We explain the code for the case of the compliance, i.e. $(c_1, c_2, c_3) = (1, 0, 0)$ in (12) and $\Gamma_s = \emptyset$, and we consider an additional volume constraint, so the functional that we minimize is

$$\mathcal{J}(\Omega) := J(\Omega) + \Lambda \mathcal{V}(\Omega), \tag{54}$$

where $\Lambda$ is a constant and $\mathcal{V}(\Omega)$ is the volume of $\Omega$.

The main file `compliance.py` can be found in the appendix, and we use a file `init.py` to initialize the data which depend on the chosen case. The user can choose between the six following cases: `half_wheel`, `bridge`, `cantilever`, `cantilever_asymmetric`, `MBB_beam`, and `cantilever_twoforces`. For instance, to run the cantilever case, the command line is

```
python compliance.py cantilever
```

An important feature of the code is that we use two separate grids. On one hand, $\mathcal{D}$ is discretized using a structured grid `mesh` made of isosceles triangles (each square is divided into four triangles), which is used to compute the solution `U` of the elasticity system, and also to compute the descent direction `th` corresponding to $\theta$. The spaces `V` and `Vvec` are spaces of scalar and vector-valued functions on `mesh`, respectively. On the other hand, we use an additional Cartesian grid, whose vertices are included in the set of vertices of `mesh`, to implement the numerical scheme (48) to solve the Hamilton-Jacobi equation, and also to perform the reinitialization (50). In `compliance.py`, the quantities defined on the Cartesian grid are matrices and therefore distinguished by the suffix `mat`. For instance `phi` is a function defined on `mesh`, while `phi_mat` is the corresponding function defined on the Cartesian grid. We need a mechanism to alternate between functions defined on `mesh` and functions defined on the Cartesian grid. This is explained in detail in Section 6.5.

In the first few lines of the code, we import the modules `dolfin`, `init`, `cm` and `pyplot` from `matplotlib`, `numpy`, `sys` and `os`. The module `matplotlib` (`http://matplotlib.org/`) is used for plotting the design. The module `dolfin` is a problem-solving environment required by FEniCS. The purpose of the line

```
9  pp.switch_backend('Agg')
```

is to use the `Agg` back end instead of the default `WebAgg` back end. With the `Agg` back end, the figures do not appear on the screen, but are saved to a file; see lines 106-113.

6.2. **Initialization of case-dependent parameters.** In this section we describe the content of the file `init.py`, which provides initial data. The outputs of `init.py` are the case-dependent variables, i.e. `Lag`, `Nx`, `Ny`, `lx`, `ly`, `Load`, `Name`, `ds`, `bcd`, `mesh`, `phi_mat`. The space `Vvec` is not case-dependent but is required to define the boundary conditions `bcd`.

The Lagrange multiplier $\Lambda$ for the volume constraint is called here `Lag`. The variable `Load` is the position of the pointwise load, for example `Load = [Point(lx, 0.5)]` for the cantilever, which means that the load is applied at the point $(l_x, 0.5)$. For the asymmetric cantilever we have `Load = [Point(lx, 0.0)]`.

The fixed domain $\mathcal{D}$ is a rectangle $\mathcal{D} = [0, l_x] \times [0, l_y]$. In `init.py` this corresponds to the variables `lx,ly`. The mesh is built using the line

```
mesh = RectangleMesh(Point(0.0,0.0),Point(lx,ly),Nx,Ny,'crossed')
```

The class `RectangleMesh` creates a mesh in a 2D rectangle spanned by two points (opposing corners) of the rectangle. The arguments `Nx,Ny` specify the number of divisions in the $x$- and $y$-directions, and the optional argument `crossed` means that each square of the grid is divided in four triangles, defined by the crossing diagonals of the square. We choose `lx,ly,Nx,Ny` with the constraint `lx Nx`$^{-1}$ $=$ `ly Ny`$^{-1}$. The choice of the argument `crossed` is necessary to have a symmetric displacement $u$ and in turn to keep a symmetric design throughout the

iterations if the problem is symmetric, for instance in the case of the cantilever. Note that to preserve the symmetry of solutions at all time, one must choose an odd number of divisions Nx or Ny, depending on the orientation of the symmetry. For instance, in the case of the symmetric cantilever, one can choose Ny$= 75$ since the symmetry axis is the line $y = 1/2$, and Nx$= 150$.

Since we chose a mesh with crossed diagonals, each square has an additional vertex at its center, where the diagonals meet. Therefore the total number of vertices is

```
37  dofsV_max =(Nx+1)*(Ny+1) + Nx*Ny
```

We also define `dofsVvec_max = 2*dofsV_max` in line 37, this represents the degrees of freedom for the vector function space `Vvec`.

The case-dependent boundary $\Gamma_d$ is defined using the class `DirBd`, and instantiated by `dirBd = DirBd()` in `init.py`. We tag `dirBd` with the number 1, the other boundaries with 0, and introduce the boundary measure `ds`. The Dirichlet boundary condition on $\Gamma_d$ is defined using

```
DirichletBC(Vvec,(0.0,0.0),boundaries,1)
```

When several types of Dirichlet boundary conditions are required, as in the case of the half-wheel for instance, the variable `bcd` is defined as a list of boundary conditions. For the cantilever, `bcd` has only one element. For the cases of the half-wheel and MBB-beam, we also define an additional class `DirBd2` to define the boundary conditions `bcd` because there are two different types of Dirichlet conditions; see Sections 7.3 and 7.5.

6.3. **Other initialization parameters.** The ersatz material coefficient $\epsilon$ is called `eps_er`; see (6). The elasticity parameters $E, \nu, \mu, \lambda$ are given lines 14-15. In lines 17-19, a directory is created to save the results. In line 21, `ls_max = 3` is the maximum number of line searches for one iteration of the main loop, `ls` is an iteration counter for the line search, and the step size used in the gradient method is `beta`, initialized as `beta0_init`. We choose `beta0_init = 0.5`. We also choose `gamma = 0.8` and `gamma2 = 0.8`, which are used to modify the step size in the line search; see Section 6.10.

The counter `It` in line 25 keeps track of the iterations of the main loop. In line 25, we also fix a maximum number of iterations `ItMax = int(1.5*Nx)`, which depends on the mesh size `Nx` due to the fact that the time step `dt` is a decreasing function of `Nx`; see Section 6.12.

6.4. **Finite elements.** In line 28 and in the file `init.py`, we define the following finite element spaces associated with `mesh`:

```
V = FunctionSpace(mesh, 'CG', 1)
Vvec = VectorFunctionSpace(mesh, 'CG', 1)
```

Here, `CG` is short for "continuous Galerkin", and the last argument is the degree of the element, meaning we have chosen the standard piecewise linear Lagrange elements. Note that the type of elements and degree can be easily modified using this command, and FEniCS offers a variety of them. However, the level set part of our code has been written for this particular type of elements, so changing it would require to modify other parts of the code, such as the function `_comp_lsf` line 173, so one should be aware that it would not be a straightforward modification.

6.5. **The function `_comp_lsf`.** Here we explain the mechanism to get `phi` from `phi_mat`. Indeed `phi_mat` is updated every iteration by the function `_hj` in line 100, and we need `phi` to define the new set `Omega` in lines 42-44. Observe that the set of vertices of the Cartesian grid is included in the set of vertices of `mesh`, indeed the vertices of `mesh` are precisely the vertices of the Cartesian grid, plus the vertices in the center of the squares where the diagonals meet, due to the choice of the argument `crossed` in `mesh`. Thus we compute the values of `phi` at the center of the squares using interpolation.

This is done in the function `_comp_lsf` (lines 173-182) in the following way. First of all, in lines 33-34, `dofsV` and `dofsVvec` are the coordinates of the vertices associated with the degrees of freedom. They are used in lines 35-36 to define `px`, `py` and `pxvec`, `pyvec`, which have integer values and are used by `_comp_lsf` to find the correspondence between the entries of the matrix `phi_mat` and the entries of `phi`. In `_comp_lsf`, precisely line 175, we check if the vertex associated with `px`, `py` corresponds to a vertex on the Cartesian grid. If this is the case, we set the values of `phi` in lines 176-177 to be equal to the values of `phi_mat` at the vertices which are common between `mesh` and the Cartesian grid. Otherwise, the vertex is at the center of a square, and we set the value at this vertex to be the mean value of the four vertices of the surrounding square; see lines 179-181.

Thus the output of `_comp_lsf` is the function `phi` defined on `mesh`. Note that if we had chosen squares with just one diagonal (choosing `left` or `right` instead of `crossed` in `RectangleMesh` in the file `init.py`) instead

of two, there would be an exact correspondence between `phi` and `phi_mat`, so that switching between the two would be easier.

6.6. **Initialization of the level set function.** In lines 39-40, we initialize `phi` as a function in the space `V`, and using `_comp_lsf` we determine its entries using `phi_mat`. The matrix `phi_mat` is initialized in the file `init.py`, since it is case-dependent. For instance, for the cantilever we can choose

$$\begin{aligned}\phi(x,y) = &-\cos(8\pi x/l_x)\cos(4\pi y) - 0.4 + \max(200(0.01 - x^2 - (y - l_y/2)^2), 0)\\ &+ \max(100(x + y - l_x - l_y + 0.1), 0) + \max(100(x - y - l_x + 0.1), 0),\end{aligned} \tag{55}$$

which is the initialization yielding the result in Figure 3. The coefficients inside the cosine determine the initial number of "holes" inside the domain (i.e. the number of connected components of $\mathcal{D} \setminus \Omega$). Here (55) corresponds to ten initial holes inside the domains (plus some half-holes on the boundary of $\mathcal{D}$).

The reason for the additional three max terms in (55) is specific of our approach. Since $\theta \in \Theta^k(\mathcal{D})$, we have $\theta = 0$ in the corners of the rectangle $\mathcal{D}$. Therefore the shape in a small neighbourhood of the corners will not change, and if we start with an inappropriate initialization, we will end with a small set of unwanted material in certain corners. Therefore the rôle of the max-terms in (55) is to create a small cut with the correct material in certain corners. Depending on the problem, it is easy to see what should be the correct corner material distribution for the final design.

Another problem may appear at boundary points which are on the symmetry axis for symmetric problems. Indeed, due to the smoothness of $\theta$ and the symmetry of the problem, we will get $\theta_x = 0$ or $\theta_y = 0$ at these points and the shape will not change there. For instance in the symmetric cantilever case, this problem happens at the point $[0, l_y/2]$. There is no issues at $[l_x, l_y/2]$, since this is the point where the load is applied, so it must be fixed anyway. This explains the term $\max(200(0.01 - x^2 - (y - l_y/2)^2), 0)$ in (55).

In line 46 we define the integration measure `dX` used line 50 to integrate on all of $\mathcal{D}$. In line 47 the normal vector `n` to $\mathcal{D}$ is introduced to define the boundary conditions for `av` in line 51.

6.7. **Domain update.** The main loop starts line 55. In line 57 we instantiate by `omega = Omega()`. This either initializes `omega` or updates `omega` if `phi` has been updated inside the loop. In line 59, `omega` is tagged with the number 1, and the complementary of `omega` is tagged with the number 0 in line 58. We define the integration measure for the subdomains $\Omega$ and $\mathcal{D} \setminus \Omega$ using

```
60  dx = Measure('dx')(subdomain_data=domains)
```

One assembles using `dx(1)` to integrate on $\Omega$, and using `dx(0)` to integrate on $\mathcal{D} \setminus \Omega$; see for instance line 68. For details on how to integrate on specific subdomains and boundaries, we refer to the FEniCS Tutorial [27] available at `http://www.springer.com/gp/book/9783319524610` and the FEniCS documentation [30].

6.8. **Solving the elasticity system.** Then we can compute `U`, the solution of the elasticity system in lines 61-62, using `_solve_PDE` in lines 116-127. We use a LU solver to solve the system; see lines 125-126. The surface load is applied pointwise using the function `PointSource`; see lines 122-124. Note that `U` is a list since we consider the general case of several loads. Thus the length of the list `U` is the length of `Load`; see line 30.

6.9. **Cost functional update.** In lines 64-70 we compute the compliance, the volume of `omega` and the cost functional `J` corresponding to (54). Observe that the command for the calculation of the compliance is close to the mathematical notation, i.e. it resembles the following mathematical formula:

$$J(\Omega) = \epsilon \int_{\mathcal{D}\setminus\Omega} S_1 : D\theta + \int_\Omega S_1 : D\theta, \qquad S_1 = 2\mu e(u) : e(u) + \lambda \operatorname{tr}(e(u))^2.$$

Recall that the compliance is a sum in the case of several loads, see Section 3.5, hence the `for` loop in line 65.

6.10. **Line search and stopping criterion.** The line search starts at line 72. If the criterion

```
72  J[It] > J[It-1]
```

is satisfied, then we reject the current step. In this case we reduce the step size `beta` by multiplying it by `gamma` in line 74. Also, we go back to the previous values of `phi_mat` and `phi` which were stored in `phi_mat_old` and `phi_old`, see line 75. Then we need to recalculate `phi_mat` and `phi` in lines 76-77 using the new step size `beta`.

If the step is not rejected, then we go to the next iteration starting from line 85. If the step was accepted in the first iteration of the line search, in order to speed up the algorithm we increase the reference step size `beta0` by setting

```
86  beta0 = min(beta0 / gamma2, 1)
```

to take larger steps, since `gamma2` is smaller than one. Here `beta0` is kept below 1 to stabilize the numerical scheme for the Hamilton-Jacobi equation, see the time step `dt` in line 150. We chose `gamma2 = 0.8` in our examples; see line 21.

If the maximum number of line searches `ls_max` is reached, we decrease in line 85 the reference step size `beta0` by setting

```
85  beta0 = max(beta0 * gamma2, 0.1*beta0_init)
```

We impose the lower limit `0.1*beta0_init` on `beta0` so that the step size does not become too small. Note that `beta` is reseted to `beta0` in line 88.

6.11. **Descent direction.** FEniCS uses the Unified Form Language (UFL) for representing weak formulations of partial differential equations, which results in an intuitive notation, close to the mathematical one. This can be seen in lines 49-51, where we define the matrix `av` which corresponds to the bilinear form (40). Here `theta` and `xi` are functions in `Vvec`, `xi` is the test function in (39), while `theta` corresponds to $\theta$ in (39). In our code, we use the notation `th` for $\theta$ when $\theta$ is the descent direction. The coefficient `1.0e4` in the boundary conditions for `av`:

```
51  1.0e4*(inner(dot(theta,n),dot(xi,n)) *(ds(0)+ds(1)+ds(2)))
```

forces $\theta \cdot n$ to be close to zero on $\partial\mathcal{D}$, which corresponds to the constraint $\theta \in \Theta^k(\mathcal{D})$. For cases where `dirBd2` is not defined, such as the cantilever case, the term `+ds(2)` has no effect.

We assemble the matrix for the PDE of `th` and define the LU solver in lines 50-53, before the start of the main loop. Indeed, the bilinear form $\mathcal{B}$ in (40) is independent of $\Omega$. Thus we reuse the factorization of the LU solver to solve the PDE for `th` using the parameter `reuse_factorization` in line 53. This allows to spare some calculations, but for grids larger than the ones considered in this paper, it would be appropriate to use more efficient approaches such as Krylov methods to solve the PDE. This can be done easily with FEniCS using one of the various available solvers.

In line 90, we compute the descent direction `th`. The function `_shape_der` in lines 129-139 solves the PDE for $\theta$, i.e. it implements (39)-(40) using the volume expression of the shape derivative (22) and (25). The variational formulation used in our code for the case of one load is: find $\theta \in H^k_d(\mathcal{D})^m$ such that

$$\int_{\mathcal{D}} \alpha_1 D\theta : D\xi + \alpha_2 \theta \cdot \xi = -d\mathcal{J}^{\texttt{vol}}(\Omega, \xi), \forall \xi \in H^1(\mathcal{D})^m \tag{56}$$

$$\text{where} \qquad d\mathcal{J}^{\texttt{vol}}(\Omega, \xi) = \int_{\mathcal{D}} (2Du^{\mathsf{T}} A_\Omega e(u) - A_\Omega e(u) : e(u) I_d) : D\xi + \Lambda \int_\Omega \operatorname{div} \xi,$$

and with $\alpha_1 = 1$ and $\alpha_2 = 0.1$. When several loads are applied, as in the case of `cantilever_twoforces`, the right-hand side in (56) should be replaced by a sum over the loads `u` in `u_vec`; see Section 3.5.

The right-hand side of (56) is assembled in line 136, but we need to integrate separately on $\Omega$ and $\mathcal{D} \setminus \Omega$ using `dx(1)` and `dx(0)`, respectively. The system is solved line 138 using the solver defined line 52.

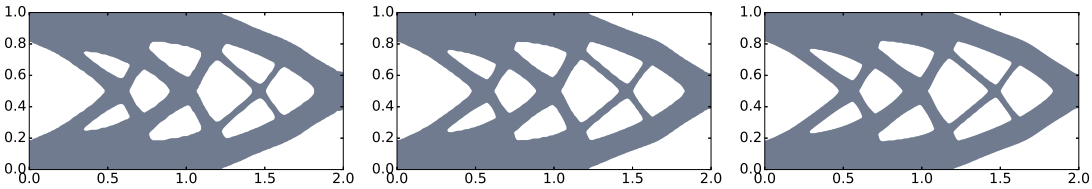
In line 90 we get the descent direction `th` in the space `Vvec`. To update `phi` we need `th` on the Cartesian grid. As we explained already, we just need to extract the appropriate values of `th` since the Cartesian grid is included in `mesh`. This is done in lines 91-97, and the corresponding function on the Cartesian grid is called `th_mat`.

6.12. **Update the level set function.** Then, we proceed to update the level set function `phi_mat` using the subfunction `_hj`. The subfunction `_hj` in lines 141-152 follows exactly the discretization procedure described in Section 5.2. In lines 143-146, the quantities `Dxm`, `Dxp`, `Dyp`, `Dym` correspond to $p^-, p^+, q^+, q^-$, respectively. In line 142, we take 10 steps of the Hamilton-Jacobi update, which is a standard, although heuristic way, to accelerate the convergence. In order to stabilize the numerical scheme, we choose the time step as

```
150  dt = beta*lx / (Nx*maxv)
```

where `maxv` is equal to $\max_\Omega(|\theta_1| + |\theta_2|)$, `lx/Nx` is the cell size, and the step size `beta` is smaller than 1 at all time in view of line 86. In line 99, we save the current versions of `phi` and `phi_mat` in the variables `phi_old`, `phi_mat_old`, for use in case the step gets rejected during the line search. In line 102, the function `phi` is extrapolated from `phi_mat` using `_comp_lsf`.

6.13. **Reinitialization of the level set function.** Every 5 iterations, we reinitialize the level set function in line 101. This is achieved by the subfunction `_reinit` in lines 154-171. The reinitialization follows the procedure described in Section 5.3. In lines 161-164, `Dxm`, `Dxp`, `Dyp`, `Dym` correspond to $p^-, p^+, q^+, q^-$, respectively. In



(A) $(Nx, Ny) = (150, 75)$. (B) $(Nx, Ny) = (202, 101)$. (C) $(Nx, Ny) = (302, 151)$.

FIGURE 3. Optimal design for the symmetric cantilever, with $\Lambda = 40$, and initialization (55).

lines 165 to 168, `Kp` and `Km` correspond to $K^+$ and $K^-$ from (52)-(53), respectively. Line 169 corresponds to (51) and line 170 to the update (50).

The function `signum`, computed in line 159, is the approximation of the sign function of $\phi$ corresponding to $S(\phi)$, defined in (49). To compute `signum`, we use `lx/Nx` for $\epsilon_s$, and $|\nabla\phi|$ is computed using symmetric finite differences $\phi$, which are given by `Dxs` and `Dys` in lines 155-158.

6.14. **Stopping criterion and saving figures.** Finally in lines 104-105, we check if the stopping criterion

```
104  if It>20 and max(abs(J[It-5:It]-J[It-1])) <2.0*J[It-1]/Nx**2:
```

is satisfied. Here, `It-1` is the current iteration. This means that the algorithm stops when the maximum difference of the value of the cost functional at the current iteration with the values of the four previous iterations is below a certain threshold. In order to take smaller steps when the grid gets finer, we have determined heuristically the threshold `2.0*J[It-1]/Nx**2` which depends on the grid size `Nx`.

Lines 107-113 are devoted to plotting the design. The filled contour of the zero level set of `phi_mat` is drawn using the `pyplot` function `contourf`; see the matplotlib documentation `http://matplotlib.org/` for details.

## 7. NUMERICAL RESULTS AND CASE-DEPENDENT PARAMETERS

In this section we discuss the case-dependent parameters in `init.py` such as boundary conditions, load position, and Lagrangian $\Lambda$.

7.1. **Symmetric cantilever.** For the symmetric cantilever the load is placed at the point $(l_x, l_y/2)$, see the parameter `Load`. The initialization for the symmetric cantilever is given by (55). See Figure 3 for the results of the symmetric cantilever for several grid sizes and $\Lambda = 40$. See also Figure 9 for a comparison of two different initializations. We observe that the optimal set is independent of the mesh size, but depends on the initialization.

To obtain a short symmetric cantilever, one can set `lx = ly` and `Nx = Ny`. Still, one should choose an odd number for `Nx` in order to preserve the symmetry of the problem.

7.2. **Asymmetric cantilever.** We take `Load = [Point(lx, 0.0)]` for the asymmetric cantilever, to have a load in the lower right corner. The initialization is also changed, so as to start with the material phase where the load is applied, more precisely, we choose

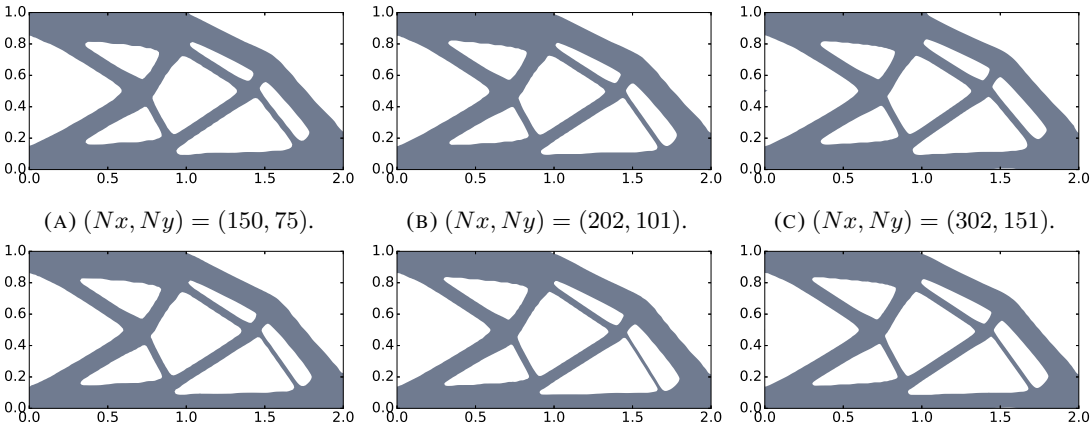
$$\phi(x, y) = -\cos(6\pi x/l_x)\cos(4\pi y) - 0.4 + \max(100(x + y - l_x - l_y + 0.1), 0).$$

See Figure 4 for the results of the asymmetric cantilever for several grid sizes, for $\Lambda = 60$ and $\Lambda = 70$.

7.3. **Half-wheel.** For the half-wheel we have `lx,ly = [2.0,1.0]`. The position of the load is given by `Load = [Point(lx/2, 0.0)]`. In the corner $(0.0, 0.0)$ we need pointwise Dirichlet conditions and rolling conditions in $(l_x, 0.0)$. For this we use the following boundaries in `init_py`:

```
class DirBd(SubDomain):
   def inside(self, x, on_boundary):
      return abs(x[0])< tol and abs(x[1])< tol
class DirBd2(SubDomain):
   def inside(self, x, on_boundary):
      return abs(x[0]-lx)<tol and abs(x[1])<tol
dirBd,dirBd2 = [DirBd(),DirBd2()]
```

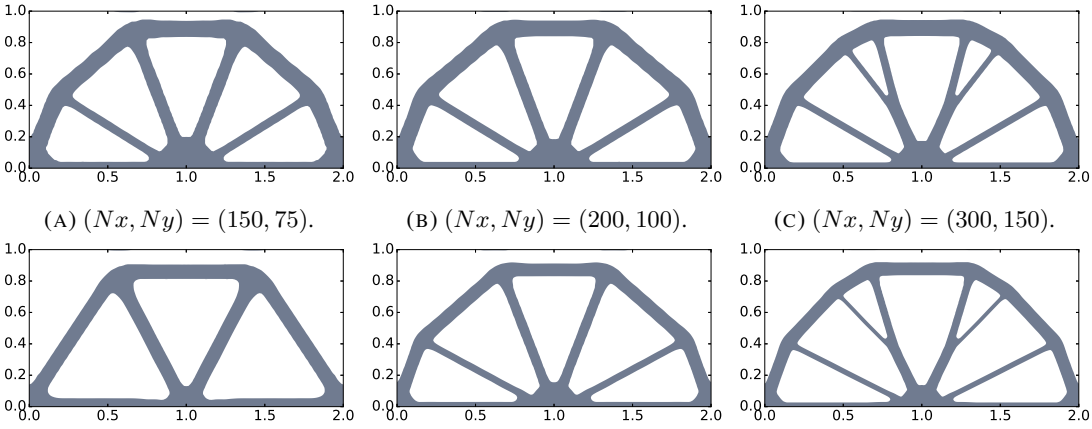
where `tol = 1E-14`. Then the two boundary parts are tagged with different numbers



(A) $(Nx, Ny) = (150, 75)$. (B) $(Nx, Ny) = (202, 101)$. (C) $(Nx, Ny) = (302, 151)$.

(D) $(Nx, Ny) = (150, 75)$. (E) $(Nx, Ny) = (202, 101)$. (F) $(Nx, Ny) = (302, 151)$.

FIGURE 4. Optimal design for the asymmetric cantilever, $\Lambda = 60$ (first row), $\Lambda = 70$ (second row).



(A) $(Nx, Ny) = (150, 75)$. (B) $(Nx, Ny) = (200, 100)$. (C) $(Nx, Ny) = (300, 150)$.

(D) $(Nx, Ny) = (150, 75)$. (E) $(Nx, Ny) = (200, 100)$. (F) $(Nx, Ny) = (300, 150)$.

FIGURE 5. Optimal design for the half-wheel, $\Lambda = 30$ (first row), $\Lambda = 50$ (second row).

```
dirBd.mark(boundaries, 1)
dirBd2.mark(boundaries, 2),
```

and we define the vector of boundary conditions as

```
bcd = [DirichletBC(Vvec, (0.0,0.0), dirBd, method='pointwise'),\
      DirichletBC(Vvec.sub(1), 0.0, dirBd2,method='pointwise')]
```

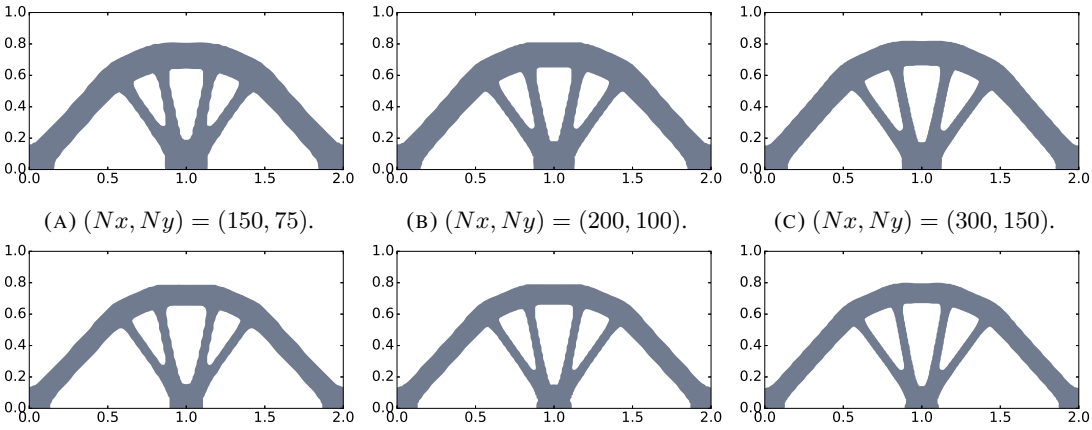
The method `pointwise` is used since `dirBd2` is a single point. Note here that the rolling boundary condition is achieved by setting the component `Vvec.sub(1)` to 0, indeed `Vvec.sub(1)` represents the $y$-component of a vector function taken in the space `Vvec` . In lines 50-51 of `compliance.py`, approximate Dirichlet conditions for $\theta$ are applied on `dirBd` and `dirBd2` as these corners should be fixed.

The initialization should also change to fit the half-wheel case. We chose

```
phi_mat = -np.cos((3.0*pi*(XX-1.0))) * np.cos(7*pi*YY) - 0.3
+ np.minimum(5.0/ly *(YY-1.0) + 4.0,0) \
+ np.maximum(100.0*(XX+YY-lx-ly+0.1),.0) + np.maximum(100.0*(-XX+YY-ly+0.1),.0)
```

In Figure 5 we compare results obtained with $\Lambda = 30$ and $\Lambda = 50$.

7.4. **Bridge.** The case of the bridge is similar to the case of the half-wheel. The main difference is the pointwise Dirichlet condition in the lower right corner, which corresponds to



(A) $(Nx, Ny) = (150, 75)$. (B) $(Nx, Ny) = (200, 100)$. (C) $(Nx, Ny) = (300, 150)$.

(D) $(Nx, Ny) = (150, 75)$. (E) $(Nx, Ny) = (200, 100)$. (F) $(Nx, Ny) = (300, 150)$.

FIGURE 6. Optimal design for the bridge, $\Lambda = 20$ (first row) and $\Lambda = 30$ (second row).

```
DirichletBC(Vvec, (0.0,0.0), dirBd2, method='pointwise')
```

Also for the initialization we take

```
phi_mat = -np.cos((4.0*pi*(XX-1.0))) * np.cos(4*pi*YY) - 0.2 \
+ np.maximum(100.0*(YY-ly+0.05),.0)
```

See Figure 6 for numerical results for the bridge, with $\Lambda = 20$ and $\Lambda = 30$.

7.5. **MBB-beam.** We define the MBB beam as in the original paper [41]. We use the symmetry of the problem to compute the solution only on the right half of the domain. Thus we impose rolling boundary condition on the left side of the computational domain $\mathcal{D}$, which corresponds to $u \cdot n = 0$. We take `lx=3.0` and `ly=1.0`, and `Nx`, `Ny` must be chosen accordingly, so as to keep a regular grid. For instance, we can choose `Nx=150`, `Ny=50`. We take `Load = [Point(0.0, 1.0)]`.

We also have pointwise rolling boundary conditions on the lower right corner of $\mathcal{D}$. In `init.py` this corresponds to the following definitions of the boundaries:

```
class DirBd(SubDomain):
   def inside(self, x, on_boundary):
      return near(x[0],.0)
class DirBd2(SubDomain):
   def inside(self, x, on_boundary):
      return abs(x[0]-lx) < tol and abs(x[1])<tol
dirBd,dirBd2 = [DirBd(),DirBd2()]
```

Then the boundaries are tagged with different numbers:

```
dirBd.mark(boundaries, 1)
dirBd2.mark(boundaries, 2)
```

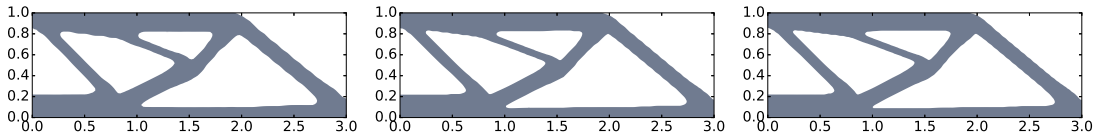
We define the boundary conditions on the two boundaries `dirBd` and `dirBd2`:

```
bcd=[DirichletBC(Vvec.sub(0),0.0,boundaries,1),\
      DirichletBC(Vvec.sub(1), 0.0, dirBd2,method='pointwise')]
```

Also, the term `+ds(2)` in lines 50-51 is active since `dirBd2` is not empty, as for the half-wheel case. We also choose an appropriate initialization

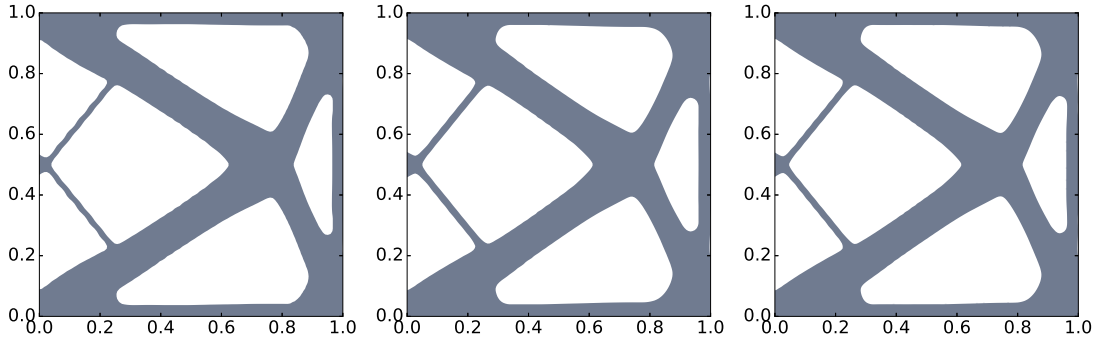
```
phi_mat = -np.cos(4.0/lx*pi*XX) *np.cos(4.0*pi*YY)-0.4\
 +np.maximum(100.0*(XX+YY-lx-ly+0.1),.0) +np.minimum(5.0/ly *(YY-1.0) + 4.0,0)
```

See Figure 7 for the MBB-beam case with $\Lambda = 130$.



(A) $(Nx, Ny) = (150, 50)$. (B) $(Nx, Ny) = (240, 80)$. (C) $(Nx, Ny) = (300, 100)$.

FIGURE 7. Optimal design for the MBB-Beam, $\Lambda = 130$.



(A) $(Nx, Ny) = (121, 121)$. (B) $(Nx, Ny) = (151, 151)$. (C) $(Nx, Ny) = (175, 175)$.

FIGURE 8. Optimal design for the cantilever with two loads, $\Lambda = 60$.

7.6. **Multiple load cases.** We have for this case that

```
Load = [Point(lx, 0.0),Point(lx, 1.0)]
```

is a list. In line 30 of `compliance.py`, `U` is thus a list with two elements corresponding to the two loads. This explains the `for` loop in `_shape_der` (see line 132).

We illustrate multiple load cases with a cantilever problem with two loads applied at the bottom-right corner and the top-right corner, both with equal intensities to get a symmetric design. Here the Lagrangian is taken as $\Lambda = 60$. The results are shown in Figure 8. We use the initialization

```
phi_mat = -np.cos(4.0*pi*(XX-0.5)) * np.cos(4.0*pi*(YY-0.5)) - 0.6 \
        -np.maximum(50.0*(YY-ly+0.1),.0)- np.maximum(50.0*(-YY+0.1),.0)
```

7.7. **Inverter.** Mechanisms require additional modifications of the code, therefore we discuss here briefly the main differences and provide the code for the inverter separately as the file `mechanism.py`. The code can be downloaded at `http://antoinelaurain.com/compliance.htm`. To run the inverter, type

```
python mechanism.py inverter
```

Unlike the compliance, the case of compliance mechanisms is not self-adjoint, therefore we need to compute an adjoint given by (27). For this we add a subfunction `_solve_adj` to compute the adjoint. The function `_solve_adj` works like `_solve_pde`, but implements the right-hand side corresponding to (27). The modification of the objective functional and of `_shape_der` follows the description of Section 3.6 in a straightforward way. In the `init.py` file, we define the boundaries `outputBd` and `inputBd` which correspond to $\Gamma_{out}$ and $\Gamma_{in}$ of Section 3.6, respectively.

We take $\Gamma_{in} = \{0\} \times (0.47, 0.53)$ and $\Gamma_{out} = \{1\} \times (0.43, 0.57)$. In order to keep the regions around $\Gamma_{out}$ and $\Gamma_{in}$ fixed, we define and tag the following small region in `mechanism.py`

```
class Fixed(SubDomain):
   def inside(self, x, on_boundary):
      return (between(x[0], (.0,.05)) and between(x[1], (.48,.52)))\
       or (between(x[0], (.9,1.0)) and between(x[1], (.43,.57)) )
fixed = Fixed()
fixed.mark(domains, 2)
```

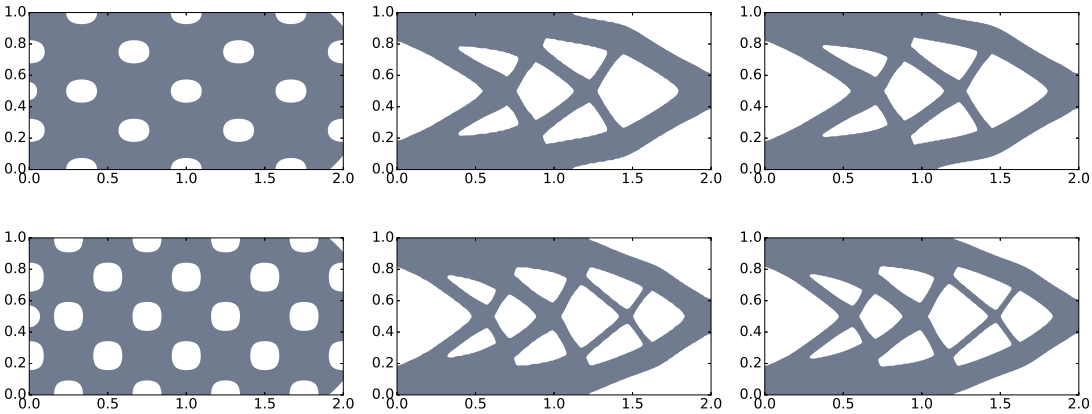


FIGURE 9. Optimal design for the symmetric cantilever, with $\Lambda = 40$ and two different initial guesses. First column: initial guess, second column: optimal design for $(Nx, Ny) = (202, 101)$, third column: optimal design for $(Nx, Ny) = (302, 151)$. The first line uses initialization (57), the second line uses initialization (55).

This is used in the following definition

```
av = assemble((inner(grad(theta),grad(xi)) +0.1*inner(theta,xi))*dx(0)\
    +1.0e5*inner(theta,xi) * dx(2)\
    +1.0e5*(inner(dot(theta,n),dot(xi,n)) * (ds(0)+ds(1)+ds(2)+ds(3))) )
```

The large coefficient `1.0e5` in the subdomain `fixed` forces `th` to be close to zero during the entire process.

We add a volume term to the objective functional with the coefficient $\Lambda = 0.01$. We choose the parameters $k_s = 0.01$, $\epsilon = 0.01$, $E = 20$, $\eta_{in} = 2, \eta_{out} = 1$, `lx=1.0`, `ly=1.0`, `beta0_init = 1.0`, `ItMax = int(2.0*Nx)` and `delta = PointSource(V.sub(0), Load, 0.05)` in function `_solve_pde`. The other parameters are the same as in `compliance.py`. For the initialization of `phi_mat` we refer to the file `init.py`. See Figure 2 and Section 3.6 for a description of the design domain, boundary conditions and optimal design.

## 8. INITIALIZATION AND COMPUTATION TIME

8.1. **Influence of initialization.** It is known that the final result may depend on the initial guess for the minimization of the compliance. We observe this phenomenon in our algorithm, as illustrated in Figure 9, where two different initializations provide two different optimal designs. We compare initialization (55) with

$$\begin{aligned}\phi(x,y) = &-\cos(6\pi x/l_x)\cos(4\pi y) - 0.6 + \max(200(0.01 - x^2 - (y - l_y/2)^2), 0)\\ &+ \max(100(x + y - l_x - l_y + 0.1), 0) + \max(100(x - y - l_x + 0.1), 0).\end{aligned} \tag{57}$$

The choice $\phi(x,y) = -\cos(6\pi x/l_x)\cos(4\pi y) - 0.6$ corresponds to a standard choice of seven holes inside the domain for the $2 \times 1$ cantilever; see [31]. It can be seen in Figure 9 that the initialization with the higher number of holes provides an optimal design $\mathcal{D} \setminus \Omega$ with more connected components.

8.2. **Computation time.** The numerical tests were run on a PC with four processors Intel Core2 Q9400, 2.66 GHz, 3.8 GB memory, with LinuxMint 17 and FEniCS 2017.1. In Tables 1 and 2 we show the computation time for the symmetric and asymmetric cantilevers. The average time for one iteration is computed by averaging over all iterations. However, it is not counting the time spent by `init.py`, and the time spent when the line search is performed, i.e. the time is recorded only for steps which are accepted.

Since we use a mesh with crossed elements, the number of elements is `dofsV_max = (Nx+1)*(Ny+1) + Nx*Ny`, see line 37. We solve two partial differential equations during each iteration (again, without counting the line search, and assuming we have only one load), one to compute `U`, and one to compute `th`. Since these are vectors, the number of degrees of freedom for solving each of these PDEs is

```
2*dofsV_max = 2*((Nx+1)*(Ny+1) + Nx*Ny)
```

For instance for the case `(Nx,Ny)=(302,151)`, as in Table 1, we get $91,658$ elements and $183,316$ degrees of freedoms. When comparing with the computational time for an algorithm such as the one described in [7], one

TABLE 1. Computation time for the asymmetric cantilever benchmark for $\Lambda = 60$.

| Mesh size | $102 \times 51$ | $202 \times 101$ | $302 \times 151$ |
|---|---|---|---|
| Number of elements | 10,558 | 41,108 | 91,658 |
| `ItMax` | 153 | 303 | 453 |
| Total iterations | 72 | 303 | 377 |
| Average time per iteration (s) | 2.02 | 7.90 | 18.00 |
| Total time (h:m:s) | 0:04:27 | 1:07:13 | 2:12:54 |

TABLE 2. Computation time for the symmetric cantilever benchmark for $\Lambda = 40$ and initialization (57).

| Mesh size | $102 \times 51$ | $202 \times 101$ | $302 \times 151$ |
|---|---|---|---|
| Number of elements | 10,558 | 41,108 | 91,658 |
| `ItMax` | 153 | 303 | 453 |
| Total iterations | 60 | 82 | 247 |
| Average time per iteration (s) | 2.00 | 8.25 | 18.20 |
| Total time (h:m:s) | 0:04:56 | 0:18:00 | 2:17:37 |

should use the number of elements as the basis for comparison. For example, a $300 \times 100$ mesh in [7] gives 30,000 elements, corresponding approximately to a grid of $170 \times 85$, which gives $29,156$ elements for our code.

The computation time is comparable with the results in [7], although slightly slower, for the same number of elements. Comparing with the educational code from [13], which is also based on the level set method, our code is significantly faster. Indeed, it was observed in [22] that the code of [13] takes a long time to converge if the mesh discretization is greater than $5,000$ elements. In [22] the authors have improved its efficiency by using a sparse matrix assembly, but they did not report on computation time.

## 9. CONCLUSION

We have presented a FEniCS code for structural optimization based on the level set method. The principal feature of the code is to rely on the notion of distributed shape derivative, which is easy to implement with FEniCS, and on the corresponding reformulation of the level set equation. We have shown how to compute the distributed shape derivative for a fairly general functional which can be used for compliance minimization and compliant mechanisms in particular. Various benchmarks of compliance minimization were tested, as well as an example of inverter mechanism.

We encourage students and newcomers to the field to experiment with new examples and parameters. The code can be used as a basis for more advanced problems. One could take advantage of the versatility of FEniCS to solve various types of PDEs, and adapt the code for multiphysics problems. An extension to three dimensions is also relatively easy using FEniCS, since the variational formulation is independent on the dimension. The main effort for extending the present code to three dimensions resides in adapting the numerical scheme for the level set part.

**Acknowledgements.** The author acknowledges the support of the Brazilian National Council for Scientific and Technological Development (Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq), through the program "Bolsa de Produtividade em Pesquisa - PQ 2015", process: 302493/2015-8. The author also acknowledges the support of FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), process: 2016/24776-6.

## 10. APPENDIX: FENICS CODE `compliance.py`

```
# ---------------------------------------------------------------------
# FEniCS 2017.1 code for level set-based structural optimization.
# Written by Antoine Laurain, 2017
# ---------------------------------------------------------------------
```

```
from dolfin import *
from init import *
from matplotlib import cm,pyplot as pp
import numpy as np, sys, os
pp.switch_backend('Agg')
set_log_level(ERROR)
# ----------------------------------------------------------------------
def _main():
    Lag,Nx,Ny,lx,ly,Load,Name,ds,bcd,mesh,phi_mat,Vvec=init(sys.argv[1])
    eps_er, E, nu = [0.001, 1.0, 0.3] # Elasticity parameters
    mu,lmbda = Constant(E/(2*(1 + nu))),Constant(E*nu/((1+nu)*(1-2*nu)))
    # Create folder for saving files
    rd = os.path.join(os.path.dirname(__file__),\
     Name +'/LagVol=' +str(np.int_(Lag))+'_Nx='+str(Nx))
    if not os.path.isdir(rd): os.makedirs(rd)
    # Line search parameters
    beta0_init,ls,ls_max,gamma,gamma2 = [0.5,0,3,0.8,0.8]
    beta0 = beta0_init
    beta = beta0
    # Stopping criterion parameters
    ItMax,It,stop = [int(1.5*Nx), 0, False]
    # Cost functional and function space
    J = np.zeros( ItMax )
    V = FunctionSpace(mesh, 'CG', 1)
    VolUnit = project(Expression('1.0',degree=2),V) # to compute volume
    U = [0]*len(Load) # initialize U
    # Get vertices coordinates
    gdim   = mesh.geometry().dim()
    dofsV  = V.tabulate_dof_coordinates().reshape((-1, gdim))
    dofsVvec = Vvec.tabulate_dof_coordinates().reshape((-1, gdim))
    px,py  = [(dofsV[:,0]/lx)*2*Nx, (dofsV[:,1]/ly)*2*Ny]
    pxvec,pyvec = [(dofsVvec[:,0]/lx)*2*Nx, (dofsVvec[:,1]/ly)*2*Ny]
    dofsV_max, dofsVvec_max =((Nx+1)*(Ny+1) + Nx*Ny)*np.array([1,2])
    # Initialize phi
    phi = Function( V )
    phi = _comp_lsf(px,py,phi,phi_mat,dofsV_max)
    # Define Omega = {phi<0}
    class Omega(SubDomain):
       def inside(self, x, on_boundary):
          return .0 <= x[0] <= lx and .0 <= x[1] <= ly and phi(x) < 0
    domains = CellFunction("size_t", mesh)
    dX = Measure('dx')
    n  = FacetNormal(mesh)
    # Define solver to compute descent direction th
    theta,xi = [TrialFunction(Vvec), TestFunction( Vvec)]
    av = assemble((inner(grad(theta),grad(xi)) +0.1*inner(theta,xi))*dX\
        + 1.0e4*(inner(dot(theta,n),dot(xi,n)) * (ds(0)+ds(1)+ds(2))) )
    solverav = LUSolver(av)
    solverav.parameters['reuse_factorization'] = True
    #---------- MAIN LOOP --------------------------------------------
    while It < ItMax and stop == False:
       # Update and tag Omega = {phi<0}, then solve elasticity system.
       omega = Omega()
       domains.set_all(0)
       omega.mark(domains, 1)
       dx = Measure('dx')(subdomain_data = domains)
       for k in range(0,len(Load)):
```

```
62            U[k] = _solve_pde(Vvec,dx,ds,eps_er,bcd,mu,lmbda,Load[k])
63         # Update cost functional
64         compliance = 0
65         for u in U:
66            eU = sym(grad(u))
67            S1 = 2.0*mu*inner(eU,eU) + lmbda*tr(eU)**2
68            compliance += assemble( eps_er*S1* dx(0) + S1*dx(1) )
69         vol = assemble( VolUnit*dx(1) )
70         J[It] = compliance + Lag * vol
71         # ------- LINE SEARCH -----------------------------------------
72         if It > 0 and J[It] > J[It-1] and ls < ls_max:
73            ls  += 1
74            beta *= gamma
75            phi_mat,phi = [phi_mat_old,phi_old]
76            phi_mat = _hj(th_mat, phi_mat, lx,ly,Nx, Ny, beta)
77            phi   = _comp_lsf(px,py,phi,phi_mat,dofsV_max)
78            print('Line search iteration : %s' % ls)
79         else:
80            print('************ ITERATION NUMBER %s' % It)
81            print('Function value : %.2f' % J[It])
82            print('Compliance     : %.2f' % compliance)
83            print('Volume fraction : %.2f' % (vol/(lx*ly)))
84            # Decrease or increase line search step
85            if ls == ls_max: beta0 = max(beta0 * gamma2, 0.1*beta0_init)
86            if ls == 0:  beta0 = min(beta0 / gamma2, 1)
87            # Reset beta and line search index
88            ls,beta,It = [0,beta0, It+1]
89            # Compute the descent direction th
90            th = _shape_der(Vvec,U,eps_er,mu,lmbda,dx,solverav,Lag)
91            th_array = th.vector().array()
92            th_mat = [np.zeros((Ny+1,Nx+1)),np.zeros((Ny+1,Nx+1))]
93            for dof in xrange(0, dofsVvec_max,2):
94               if np.rint(pxvec[dof]) %2 == .0:
95                  cx,cy= np.int_(np.rint([pxvec[dof]/2,pyvec[dof]/2]))
96                  th_mat[0][cy,cx] = th_array[dof]
97                  th_mat[1][cy,cx] = th_array[dof+1]
98            # Update level set function phi using descent direction th
99            phi_old, phi_mat_old = [phi, phi_mat]
100           phi_mat = _hj(th_mat, phi_mat, lx,ly,Nx,Ny, beta)
101           if np.mod(It,5) == 0: phi_mat = _reinit(lx,ly,Nx,Ny,phi_mat)
102           phi = _comp_lsf(px,py,phi,phi_mat,dofsV_max)
103           #------------ STOPPING CRITERION ----------------------------
104           if It>20 and max(abs(J[It-5:It]-J[It-1]))<2.0*J[It-1]/Nx**2:
105              stop = True
106           #------------ Plot Geometry ---------------------------------
107           if np.mod(It,10)==0 or It==1 or It==ItMax or stop==True:
108              pp.close()
109              pp.contourf(phi_mat,[-10.0,.0],extent = [.0,lx,.0,ly],\
110               cmap=cm.get_cmap('bone'))
111              pp.axes().set_aspect('equal','box')
112              pp.show()
113              pp.savefig(rd+'/it_'+str(It)+'.pdf',bbox_inches='tight')
114   return
115# ---------------------------------------------------------------------
116def _solve_pde(V, dx, ds, eps_er, bcd, mu, lmbda, Load):
117   u,v = [TrialFunction(V), TestFunction(V)]
118   S1 = 2.0*mu*inner(sym(grad(u)),sym(grad(v))) + lmbda*div(u)*div(v)
```

```
    A = assemble( S1*eps_er*dx(0) + S1*dx(1) )
    b = assemble( inner(Expression(('0.0', '0.0'),degree=2) ,v) * ds(2))
    U = Function(V)
    delta = PointSource(V.sub(1), Load, -1.0)
    delta.apply(b)
    for bc in bcd: bc.apply(A,b)
    solver = LUSolver(A)
    solver.solve(U.vector(), b)
    return U
#-----------------------------------------------------------------------
def _shape_der(Vvec, u_vec , eps_er, mu, lmbda, dx, solver, Lag):
    xi = TestFunction(Vvec)
    rv = 0.0
    for u in u_vec:
        eu,Du,Dxi = [sym(grad(u)),grad(u),grad(xi)]
        S1 = 2*mu*(2*inner((Du.T)*eu,Dxi) -inner(eu,eu)*div(xi))\
         + lmbda*(2*inner( Du.T, Dxi )*div(u) - div(u)*div(u)*div(xi) )
        rv += -assemble(eps_er*S1*dx(0) + S1*dx(1) + Lag*div(xi)*dx(1))
    th = Function(Vvec)
    solver.solve(th.vector(), rv)
    return th
#-----------------------------------------------------------------------
def _hj(v,psi,lx,ly,Nx,Ny,beta):
    for k in range(10):
        Dym = Ny*np.repeat(np.diff(psi,axis=0),[2]+[1]*(Ny-1),axis=0)/ly
        Dyp = Ny*np.repeat(np.diff(psi,axis=0),[1]*(Ny-1)+[2],axis=0)/ly
        Dxm = Nx*np.repeat(np.diff(psi),[2]+[1]*(Nx-1),axis=1)/lx
        Dxp = Nx*np.repeat(np.diff(psi),[1]*(Nx-1)+[2],axis=1)/lx
        g = 0.5*( v[0]*(Dxp + Dxm) + v[1]*(Dyp + Dym)) \
          - 0.5*(np.abs(v[0])*(Dxp - Dxm) + np.abs(v[1])*(Dyp - Dym))
        maxv = np.max(abs(v[0]) + abs(v[1]))
        dt = beta*lx / (Nx*maxv)
        psi = psi - dt*g
    return psi
#-----------------------------------------------------------------------
def _reinit(lx,ly,Nx,Ny,psi):
    Dxs = Nx*(np.repeat(np.diff(psi),[2]+[1]*(Nx-1),axis=1) \
         +np.repeat(np.diff(psi),[1]*(Nx-1)+[2],axis=1))/(2*lx)
    Dys = Ny*(np.repeat(np.diff(psi,axis=0),[2]+[1]*(Ny-1),axis=0)\
         +np.repeat(np.diff(psi,axis=0),[1]*(Ny-1)+[2],axis=0))/(2*ly)
    signum = psi / np.power(psi**2 + ((lx/Nx)**2)*(Dxs**2+Dys**2),0.5)
    for k in range(0,2):
        Dym = Ny*np.repeat(np.diff(psi,axis=0),[2]+[1]*(Ny-1),axis=0)/ly
        Dyp = Ny*np.repeat(np.diff(psi,axis=0),[1]*(Ny-1)+[2],axis=0)/ly
        Dxm = Nx*np.repeat(np.diff(psi),[2]+[1]*(Nx-1),axis=1)/lx
        Dxp = Nx*np.repeat(np.diff(psi),[1]*(Nx-1)+[2],axis=1)/lx
        Kp = np.sqrt((np.maximum(Dxm,0))**2 + (np.minimum(Dxp,0))**2 \
          + (np.maximum(Dym,0))**2 + (np.minimum(Dyp,0))**2)
        Km = np.sqrt((np.minimum(Dxm,0))**2 + (np.maximum(Dxp,0))**2 \
          + (np.minimum(Dym,0))**2 + (np.maximum(Dyp,0))**2)
        g = np.maximum(signum,0)*Kp + np.minimum(signum,0)*Km
        psi = psi - (0.5*lx/Nx)*(g - signum)
    return psi
#-----------------------------------------------------------------------
def _comp_lsf(px,py,phi,phi_mat,dofsV_max):
    for dof in range(0,dofsV_max):
        if np.rint(px[dof]) %2 == .0:
```

```
            cx,cy = np.int_(np.rint([px[dof]/2,py[dof]/2]))
            phi.vector()[dof] = phi_mat[cy,cx]
        else:
            cx,cy = np.int_(np.floor([px[dof]/2,py[dof]/2]))
            phi.vector()[dof] = 0.25*(phi_mat[cy,cx] + phi_mat[cy+1,cx]\
              + phi_mat[cy,cx+1] + phi_mat[cy+1,cx+1])
    return phi
# --------------------------------------------------------------------
if __name__ == '__main__':
    _main()
```

## REFERENCES


[1] G. Allaire, C. Dapogny, G. Delgado, and G. Michailidis. Multi-phase structural optimization *via* a level set method. *ESAIM Control Optim. Calc. Var.*, 20(2):576–611, 2014.

[2] G. Allaire and O. Pantz. Structural optimization with FreeFem++. *Struct. Multidiscip. Optim.*, 32(3):173–181, 2006.

[3] Grégoire Allaire, François Jouve, and Anca-Maria Toader. A level-set method for shape optimization. *C. R. Math. Acad. Sci. Paris*, 334(12):1125–1130, 2002.

[4] Grégoire Allaire, François Jouve, and Anca-Maria Toader. Structural optimization using sensitivity analysis and a level-set method. *J. Comput. Phys.*, 194(1):363–393, 2004.

[5] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie Rognes, and Garth Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[6] Samuel Amstutz and Heiko Andrä. A new algorithm for topology optimization using a level-set method. *J. Comput. Phys.*, 216(2):573–588, 2006.

[7] Erik Andreassen, Anders Clausen, Mattias Schevenels, Boyan S. Lazarov, and Ole Sigmund. Efficient topology optimization in matlab using 88 lines of code. *Structural and Multidisciplinary Optimization*, 43(1):1–16, 2010.

[8] T. Belytschko, S. P. Xiao, and C. Parimi. Topology optimization with implicit functions and regularization. *International Journal for Numerical Methods in Engineering*, 57(8):1177–1196, 2003.

[9] M. P. Bendsø e and O. Sigmund. *Topology optimization*. Springer-Verlag, Berlin, 2003. Theory, methods and applications.

[10] M. P. Bendsøe. Optimal shape design as a material distribution problem. *Structural optimization*, 1(4):193–202.

[11] Martin Berggren. A unified discrete-continuous sensitivity analysis method for shape optimization. In *Applied and numerical partial differential equations*, volume 15 of *Comput. Methods Appl. Sci.*, pages 25–39. Springer, New York, 2010.

[12] Martin Burger. A framework for the construction of level set methods for shape optimization and reconstruction. *Interfaces Free Bound.*, 5(3):301–329, 2003.

[13] Vivien J. Challis. A discrete level-set topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 41(3):453–464, 2009.

[14] David L. Chopp. Computing minimal surfaces via level set curvature flow. *J. Comput. Phys.*, 106(1):77–91, 1993.

[15] Marc Dambrine and Djalil Kateb. On the ersatz material approximation in level-set methods. *ESAIM Control Optim. Calc. Var.*, 16(3):618–634, 2010.

[16] Frédéric de Gournay. Velocity extension for the level-set method and multiple eigenvalues in shape optimization. *SIAM J. Control Optim.*, 45(1):343–367, 2006.

[17] M. C. Delfour and J.-P. Zolésio. *Shapes and geometries*, volume 22 of *Advances in Design and Control*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2011. Metrics, analysis, differential calculus, and optimization.

[18] Michel C. Delfour, Zoubida Mghazli, and Jean-Paul Zolésio. Computation of shape gradients for mixed finite element formulation. In *Partial differential equation methods in control and shape analysis (Pisa)*, volume 188 of *Lecture Notes in Pure and Appl. Math.*, pages 77–93. Dekker, New York, 1997.

[19] J. D. Eshelby. The elastic energy-momentum tensor. *J. Elasticity*, 5(3-4):321–335, 1975. Special issue dedicated to A. E. Green.

[20] Piotr Fulmański, Antoine Laurain, Jean-Francois Scheid, and Jan Sokołowski. A level set method in shape and topology optimization for variational inequalities. *Int. J. Appl. Math. Comput. Sci.*, 17(3):413–430, 2007.

[21] Piotr Fulmański, Antoine Laurain, Jean-François Scheid, and Jan Sokołowski. Level set method with topological derivatives in shape optimization. *Int. J. Comput. Math.*, 85(10):1491–1514, 2008.

[22] Arun L. Gain and Glaucio H. Paulino. A critical comparative assessment of differential equation-driven methods for structural topology optimization. *Structural and Multidisciplinary Optimization*, 48(4):685–710, 2013.

[23] P. Gangl, U. Langer, A. Laurain, H. Meftahi, and K. Sturm. Shape optimization of an electric motor subject to nonlinear magnetostatics. *SIAM Journal on Scientific Computing*, 37(6):B1002–B1025, 2015.

[24] Michael Hintermüller and Antoine Laurain. Multiphase image segmentation and modulation recovery based on shape and topological sensitivity. *J. Math. Imaging Vision*, 35(1):1–22, 2009.

[25] Michael Hintermüller, Antoine Laurain, and Irwin Yousept. Shape sensitivities for an inverse problem in magnetic induction tomography based on the eddy current model. *Inverse Problems*, 31(6):065006, 25, 2015.

[26] R. Hiptmair, A. Paganini, and S. Sargheini. Comparison of approximate shape gradients. *BIT*, 55(2):459–485, 2015.

[27] H.P. Langtangen and A. Logg. *Solving PDEs in Python: The FEniCS Tutorial I*. Simula SpringerBriefs on Computing. Springer International Publishing, 2017.

[28] Laurain, Antoine and Sturm, Kevin. Distributed shape derivative via averaged adjoint method and applications. *ESAIM: M2AN*, 50(4):1241–1267, 2016.

[29] Z. Liu, J.G. Korvink, and R. Huang. Structure topology optimization: fully coupled level set method via femlab. *Structural and Multidisciplinary Optimization*, 29(6):407–417, 2005.

[30] A. Logg, K.-A. Mardal, and G. N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012.
[31] Junzhao Luo, Zhen Luo, Liping Chen, Liyong Tong, and Michael Yu Wang. A semi-implicit level set method for structural shape and topology optimization. *J. Comput. Phys.*, 227(11):5561–5581, 2008.
[32] Antonio André Novotny and Jan Sokołowski. *Topological derivatives in shape optimization*. Interaction of Mechanics and Mathematics. Springer, Heidelberg, 2013.
[33] Stanley Osher and Ronald Fedkiw. *Level set methods and dynamic implicit surfaces*, volume 153 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2003.
[34] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.
[35] Stanley Osher and Chi-Wang Shu. High-order essentially nonoscillatory schemes for Hamilton-Jacobi equations. *SIAM J. Numer. Anal.*, 28(4):907–922, 1991.
[36] Stanley J. Osher and Fadil Santosa. Level set methods for optimization problems involving geometry and constraints. I. Frequencies of a two-density inhomogeneous drum. *J. Comput. Phys.*, 171(1):272–288, 2001.
[37] Masaki Otomori, Takayuki Yamada, Kazuhiro Izui, and Shinji Nishiwaki. Matlab code for a level set-based topology optimization method using a reaction diffusion equation. *Structural and Multidisciplinary Optimization*, 51(5):1159–1172, 2014.
[38] Danping Peng, Barry Merriman, Stanley Osher, Hongkai Zhao, and Myungjoo Kang. A PDE-based fast local level set method. *J. Comput. Phys.*, 155(2):410–438, 1999.
[39] J. A. Sethian. *Level set methods and fast marching methods*, volume 3 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, second edition, 1999. Evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science.
[40] J. A. Sethian and Andreas Wiegmann. Structural boundary design via level set and immersed interface methods. *J. Comput. Phys.*, 163(2):489–528, 2000.
[41] O. Sigmund. A 99 line topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 21(2):120–127, 2014.
[42] Jan Sokołowski and Jean-Paul Zolésio. *Introduction to shape optimization*, volume 16 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1992. Shape sensitivity analysis.
[43] Kevin Sturm. Minimax Lagrangian approach to the differentiability of nonlinear PDE constrained shape functions without saddle point assumption. *SIAM J. Control Optim.*, 53(4):2017–2039, 2015.
[44] Kevin Sturm, Michael Hintermüller, and Dietmar Hömberg. Distortion compensation as a shape optimisation problem for a sharp interface model. *Comput. Optim. Appl.*, 64(2):557–588, 2016.
[45] Cameron Talischi, Glaucio H. Paulino, Anderson Pereira, and Ivan F. M. Menezes. Polymesher: a general-purpose mesh generator for polygonal elements written in matlab. *Structural and Multidisciplinary Optimization*, 45(3):309–328, 2012.
[46] Cameron Talischi, Glaucio H. Paulino, Anderson Pereira, and Ivan F. M. Menezes. Polytop: a matlab implementation of a general topology optimization framework using unstructured polygonal finite element meshes. *Structural and Multidisciplinary Optimization*, 45(3):329–357, 2012.
[47] N. P. van Dijk, K. Maute, M. Langelaar, and F. van Keulen. Level-set methods for structural topology optimization: a review. *Struct. Multidiscip. Optim.*, 48(3):437–472, 2013.
[48] Michael Yu Wang, Xiaoming Wang, and Dongming Guo. A level set method for structural topology optimization. *Comput. Methods Appl. Mech. Engrg.*, 192(1-2):227–246, 2003.
[49] Michael Yu Wang and Shiwei Zhou. Phase field: a variational method for structural topology optimization. *CMES Comput. Model. Eng. Sci.*, 6(6):547–566, 2004.
[50] Hongkai Zhao. A fast sweeping method for eikonal equations. *Math. Comp.*, 74(250):603–627, 2005.
[51] M. Zhou and G.I.N. Rozvany. The COC algorithm, part II: Topological, geometrical and generalized shape optimization. *Computer Methods in Applied Mechanics and Engineering*, 89(1):309 – 336, 1991. Second World Congress on Computational Mechanics.

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA,UNIVERSIDADE DE SÃO PAULO, RUA DO MATÃO, 1010, 05508-090 - SÃO PAULO, BRAZIL

*E-mail address*: laurain@ime.usp.br