Structural and Multidisciplinary Optimization
https://doi.org/10.1007/s00158-020-02552-0

EDUCATIONAL PAPER

A MATLAB code for topology optimization using the geometry projection method

Hollis Smith [1] · Julián A. Norato [1]

Received: 19 December 2019 / Revised: 29 January 2020 / Accepted: 14 February 2020

Abstract
This work introduces a MATLAB code to perform the topology optimization of structures made of bars using the geometry projection method. The primary objective of this code is to make available to the structural optimization community a simple implementation of the geometry projection method that illustrates the formulation and makes it possible to easily and efficiently reproduce results. A guiding principle in writing the code is modularity, so that researchers can easily modify the program for their own purposes. Another goal is efficiency, for which extensive use of vectorization is made. This paper details the formulation of the geometry projection, discusses implementation aspects of the code, and demonstrates some of its capabilities by presenting several 2D and 3D compliance minimization examples.

Keywords Topology optimization · Geometry projection

1 Introduction

The prevalent techniques to perform topology optimization of continua are the density-based and the level-set methods (Sigmund and Maute 2013). These techniques produce organic designs that are highly efficient. In some cases, a design that closely follows the optimal topology can be manufactured using, for example, additive manufacturing or casting techniques. However, when the most economical fabrication process consists of joining stock material such as bars or plates, it can be very difficult to translate the optimal topology into a design that conforms to that process. This difficulty has motivated the development of topology optimization techniques that produce designs exclusively made of geometric primitives.

One of the topology optimization techniques that render designs made of geometric components is the geometry projection method (GPM) (Bell et al. 2012; Norato et al. 2004, 2015). There exist other techniques to perform topology optimization using geometric components, such as the moving morphable components method (Guo et al. 2014; Zhang et al. 2016b); a review of these techniques is outside of the scope of this paper; and we refer the reader to the recent review by Wein et al. (2019). The purpose of this work is to introduce a MATLAB code to illustrate the GPM for the topology optimization of 2D and 3D structures made of bars. The GPM is described in detail in Section 2. In particular, we aim to demonstrate how the geometry mapping can be performed in an efficient manner using vectorized operations. Unlike other educational codes published in this journal, we do not attempt to fit the code into a relatively low number of lines and include it in the manuscript. Although this would be in principle possible, we believe in our case it would sacrifice clarity and therefore hinder the objective of explaining the inner workings of the geometry projection. Instead, this manuscript provides an overview of the program, and the code is released for free and made available through GitHub at https://github.com/jnorato/GPTO. This approach allows us to better modularize the code but also to add various functionalities and options that users can experiment with, which we believe will be beneficial to the research community. The code is released under a Creative Commons CC-BY-NC 4.0 license, which means it is free for non-commercial use and that appropriate credit must be given. The program is not an open source

This work was supported by the U.S. Office of Naval Research, Grant Number N00014-17-1-2505.

Responsible Editor: Xu Guo

✉ Julián A. Norato
julian.norato@uconn.edu

[1] The University of Connecticut, 191 Auditorium Road, U-3139, Storrs, CT 06269, USA

Published online: 23 March 2020

program in the sense that a repository to which users can contribute to further development of the code is not available. Nevertheless, the authors welcome any suggestions that users may have for future improvements.

The rest of the manuscript is structured as follows. Section 2 presents formulation of the geometry projection method. In Section 3, details of the implementation are provided. Some usage examples are presented in Section 4, and we draw conclusions of this work in Section 5.

2 Geometry projection formulation

The basic idea of the geometry projection is to take a high-level parametric description of a geometric component ω_b and map it onto a pseudo-density field over a design region $\Omega \supseteq \omega_b$. This field is subsequently discretized via a fixed mesh for analysis. The mapping must be differentiable so that design sensitivities of the optimization functions are well defined, and thus efficient gradient-based optimizers can be employed. This section describes the formulation of the geometry projection and of the sensitivities.

2.1 Projected, penalized, and combined densities

We first consider the projection of a single geometric component. The projected density at a point x is the volume fraction of the intersection between the ball B_x^r of radius r centered at x and the component ω_b (Norato et al. 2004), i.e.,

$$\rho(x, z_b, r) := \frac{|B_x^r \cap \omega_b(z_b)|}{|B_x^r|} \tag{1}$$

where z_b is the vector of geometric parameters that describe ω_b. For arbitrary shapes of the component ω_b, computing exactly this intersection is not straightforward. Therefore, we seek an approximation of (1) that is easy to compute and is differentiable. If we assume ω_b to be a smooth closed manifold and r to be sufficiently small, then $\partial\omega_b \cap B_x^r$ can be approximated as a line segment in 2D and a circle in 3D (since B_x^r is a circle and a sphere, respectively). With that assumption, the projected density can be approximated as the area fraction of the circular segment of height $h = r - \phi_b$ in 2D (or volume fraction of the spherical cap of the same height in 3D), where $\phi_b(x, z_b)$ is the signed distance from x to $\partial\omega_b$ (see Fig. 1).[1] The projected density is thus given by

$$\rho_b\left(\frac{\phi_b(x, z_b)}{r}\right) := \begin{cases} 0, & \text{if } \phi_b/r < -1 \\ \widetilde{H}(\phi_b/r), & \text{if} -1 \le \phi_b/r \le 1 \\ 1, & \text{if } \phi_b/r > 1, \end{cases} \tag{2}$$

[1] Unlike previous works, here the signed distance is positive inside ω_b and negative outside, to be consistent with most of the literature.

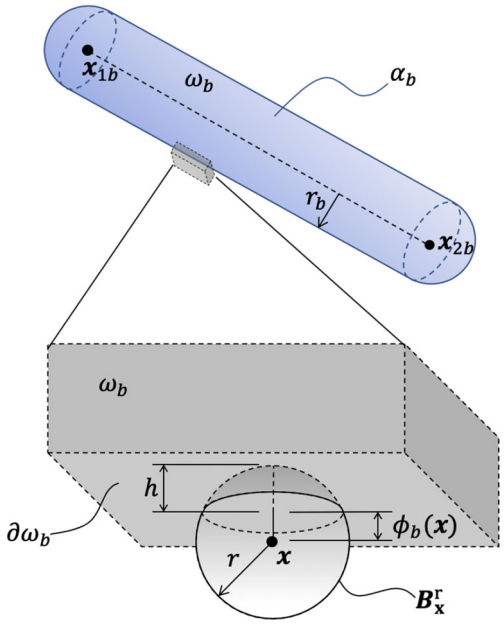

Fig. 1 Geometry projection

where

$$\widetilde{H}(x) = \begin{cases} 1 - \dfrac{\arccos x + x\sqrt{1-x^2}}{\pi}, & \text{in 2D} \\ \dfrac{1}{2} + \dfrac{3x}{4} - \dfrac{x^3}{4}, & \text{in 3D.} \end{cases} \tag{3}$$

The notation $\widetilde{H}$ is used here to bring attention to the fact that this is a regularized Heaviside; however, we emphasize that it corresponds to the area (volume) fraction calculation of the circular (spherical) cap. The size r of the sample window is fixed throughout the optimization; hence, it is hereafter omitted as an argument. We also omit arguments on the right-hand side of equations for brevity. The calculation of ϕ_b depends on the particular representation of the geometric components. In the case of the offset bars considered in this work, it will be detailed in Section 2.2.

A key ingredient of the geometry projection method is that, in addition to the parameters that describe the shape of each component, a size variable $\alpha_b \in [0, 1]$ is ascribed to each component b. This size variable is penalized in the same spirit of penalization schemes employed in density-based topology optimization, so that a value of $\alpha_b = 1$ indicates the geometric component must be part of the structure, while a value $\alpha_b = 0$ indicates the component must be removed from the design. This feature makes it easier for the optimizer to remove geometric components and modify the topology. The penalization is achieved via the definition of a *penalized density* $\widehat{\rho}_b$ that incorporates the size variable α_b as

$$\widehat{\rho}_b(x, z_b, q) := \mu(\alpha_b \rho_b, q), \tag{4}$$

where μ is a penalization function, q is the penalization parameter, and we note that $\alpha_b \in z_b$. For example, for a power law penalization, as in the solid isotropic material penalization (SIMP) commonly used in density-based topology optimization, we have that $\mu(\alpha_\mathrm{b}\rho_\mathrm{b}, q) = (\alpha_\mathrm{b}\rho_\mathrm{b})^q$. Importantly, for the penalization to be effective, it is required that the volume of the structure be computed with the *unpenalized* density (e.g., $q = 1$ for SIMP).

Here we note that, unlike our previous works, the projected density ρ_b is also penalized, as otherwise the material interpolation is linear in ρ_b when $\alpha_b = 1$, which renders a physically unrealistic material wherever intermediate-density material appears (i.e., along the boundaries of geometric components), as demonstrated in Bendsøe and Sigmund (1999). We also observe slightly better convergence with this modification, and this strategy produces less gaps in between components in the final design that are likely due to unrealistically stiff gray regions.

When multiple bars overlap, we combine the penalized densities for all bars into a *combined density* given by

$$\rho(x, Z, p) := \begin{cases} \rho_{\min}, & \text{if } \widehat{\rho}_b = 0, \text{for } b = 1, \cdots, n_b \\ \widetilde{\max}\limits_{b} \widehat{\rho}_b, & \text{otherwise,} \end{cases} \tag{5}$$

where n_b is the number of geometric components, $\widetilde{\max}$ denotes a smooth approximation of the maximum function, $Z := \left[z_1^T, \cdots, z_{n_b}^T\right]^T$ is the vector of design variables for all components, and $0 < \rho_{\min} \ll 1$ is a positive lower bound to prevent an ill-posed analysis. An example of a function that embodies (5) is the modified p-norm described in Zhang et al. (2016a):1

$$\rho(x, Z, p) := \left[\rho_{\min}^p + \left(1 - \rho_{\min}^p\right) \sum_{b=1}^{n_b} \widehat{\rho}_b^{\,p}\right]^{\frac{1}{p}}. \tag{6}$$

This modified p norm renders $\rho = \rho_{\min}$, if $\widehat{\rho}_b = 0 \forall b$ and $\rho = 1$ if $\widehat{\rho}_b = 1 \forall b$, regardless of the number of geometric components. Finally, the combined density is reflected in the analysis by using an ersatz material, with the elasticity tensor modified as

$$\mathbb{C}(x, Z) := \rho \mathbb{C}_0 \tag{7}$$

where $\mathbb{C}_0$ is the elasticity tensor for the material the geometric components are made of.

The foregoing formulation can be used for geometric components of any shape that are made of a single, isotropic material, so long as a signed distance and its derivatives with respect to the geometric parameters can be computed. For instance, this scheme has been employed to design structures made of bars (Norato et al. 2015), flat plates (Zhang et al. 2016a), curved plates bent along a circular arc (Zhang et al. 2018), and supershapes (Norato 2018), which are a generalization of hyperellipses with variable symmetry. The case of multi-material structures, where components can be made of one out of a given set of isotropic materials, and where the optimizer simultaneously determines the optimal components layout and the best material for each component, requires a different strategy to combine the geometric primitives and to interpolate the properties of the different materials (cf. Watts and Tortorelli (2017) and Kazemi et al. (2018)).

2.2 Distance

We now describe the computation of the signed distance ϕ_b in 1. Here, as in some of our previous works, we represent bars as offset surfaces (Norato et al. 2015). Specifically, the boundary of the bar is given by the set of all points at a distance r_b of the line segment with endpoints $x_{1\mathrm{b}}$ and $x_{2\mathrm{b}}$ (cf. Figure 1). This definition represents bars as rectangles with semicircular caps in 2D and cylinders with semispherical caps in 3D. The user must define an initial design made of a set of bars; bars can be removed from the design during the optimization (e.g., by setting its size variable to zero) or reintroduced (by increasing a zero size variable), but in this code, bars cannot be introduced during the optimization.

The vector of design parameters for bar b in the offset surface representation is therefore given by $z_\mathrm{b} = \{x_{1\mathrm{b}}, x_{2\mathrm{b}}, r_\mathrm{b}, \alpha_\mathrm{b}\}$. Our formulation allows for the case where endpoints are shared by two or more bars, thus they remain "connected" at these endpoints throughout the optimization (much like ground structure approaches). The total number of design variables is $2(n_\mathrm{d} n_\mathrm{p} + n_\mathrm{b})$, where $n_\mathrm{d} = 2$, 3 in 2D and 3D, respectively, and n_p is the total number of endpoints. Alternatively, a bar can be defined to have its own medial axis endpoints, thus it can "float" within the design region independently of other bars. If all bars are "floating," then $n_\mathrm{p} = 2n_\mathrm{b}$.

The advantage of using the offset surface representation is that the signed distance to the boundary of the bar can simply be computed as the distance to the medial axis (d_b) minus the bar radius as

$$\phi_b(x, z_\mathrm{b}) = d_\mathrm{b}(x, z_\mathrm{b}) - r_\mathrm{b}. \tag{8}$$

To compute the distance to the bar's boundary, it is therefore only necessary to compute the distance to the bar's medial axis. Moreover, as we show in the sequel, the distance d_b and its design sensitivities can be computed in closed form as a function of the bar's design parameters z_b.

Though not strictly necessary, an element-uniform combined density (5) is employed for the computation of the ersatz material of (7). Therefore, the combined density and consequently the signed distance are computed at the element centroid, which we denote as x_e (cf. Figure 2). We also use the notation

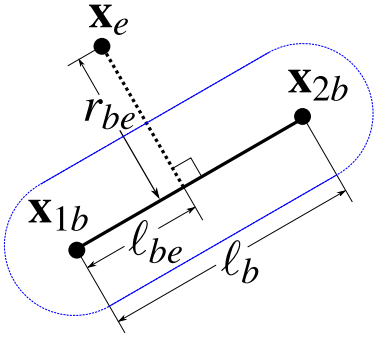

Fig. 2 Definition of vectors and quantities for distance computation

$$x_{\alpha/\beta} := x_\alpha - x_\beta. \tag{9}$$

The length of bar b is given by

$$\ell_{\mathrm{b}} = \|x_{2\mathrm{b}/1\mathrm{b}}\|, \tag{10}$$

and the unit vector along the bar's medial axis is given by

$$a_{\mathrm{b}} = \frac{x_{2\mathrm{b}/1\mathrm{b}}}{\ell_{\mathrm{b}}}. \tag{11}$$

Consider a Cartesian coordinate system for each bar in which $x_{1\mathrm{b}}$ is the origin and a_{b} is the first axis. On this coordinate system, ℓ_{be} and r_{be} are, respectively, the parallel and perpendicular components of $x_{\mathrm{e}/1\mathrm{b}}$ given by

$$\ell_{\mathrm{be}} = a_{\mathrm{b}} \cdot x_{\mathrm{e}/1\mathrm{b}} \tag{12}$$

$$r_{\mathrm{be}} = \|r_{\mathrm{be}}\| = \|x_{e/1b} - \ell_{\mathrm{be}} \mathbf{a_b}\|. \tag{13}$$

The distance from the medial segment of bar b to the centroid of element e is given by

$$d_{be} = \begin{cases} \|x_{e/1b}\|, & \text{if } \ell_{be} \le 0 \\ \|x_{e/2b}\|, & \text{if } \ell_{be} < \ell_b \\ r_{be}, & \text{otherwise} \end{cases} \tag{14}$$

2.3 Optimization problem

The optimization problem solved by default in our code is the compliance minimization problem given by

$$\min_Z \; c(u(Z)) := \int_{\Gamma_t} u(Z) \cdot t \, \mathrm{d}\Gamma \tag{15}$$

subject to

$$v_f := \frac{1}{|\Omega|} \int_\Omega \rho \, \mathrm{d}\Omega \le v_f^* \tag{16}$$

$$\mathsf{a}(u(Z), v) = \mathsf{l}(\mathrm{v}), \forall \mathrm{v} \in \mathcal{U}_0 \tag{17}$$

$$x_{1b}, x_{2b} \in \Omega \text{ for } b = 1, \cdots, n_b \tag{18}$$

$$r_{b_{min}} \le r_b \le r_{b_{max}} \tag{19}$$

$$0.0 \le \alpha_b \le 1.0, \tag{20}$$

where c is the compliance, v_f is the volume fraction and v_f^* its limit, and ρ is the combined density of (5). The domain Ω denotes the region occupied by the design envelope, and Γ_t and Γ_u denote the portions of $\partial\Omega$ where traction t and displacement boundary conditions u are imposed, respectively, with $\Gamma_t \cup \Gamma_u = \partial\Omega$, $\Gamma_t \cap \Gamma_u = \varnothing$ as usual. We assume there are no body loads and the applied traction is design-independent. The admissible sets for the trial displacement functions u and the test functions v are given by $\mathcal{U} := \left\{ u \middle| u \in H^1(\Omega), u\big|_{\Gamma_u} = u \right\}$ and $\mathcal{U}_0 := \left\{ \mathrm{v} \middle| \mathrm{v} \in H^1(\Omega), v\big|_{\Gamma_u} = 0 \right\}$, respectively. The energy bilinear form a and the load linear form l in (17) are computed as

$$\mathsf{a}(u, v) := \int_\Omega \nabla_S v \cdot \mathbb{C} \nabla_S u \, \mathrm{d}\Omega \tag{21}$$

with $\mathbb{C}$ the ersatz elasticity tensor of (7), ∇_S the symmetric gradient operator and

$$\mathsf{l}(v) := \int_{\Gamma_t} \nabla_S v \cdot t \, \mathrm{d}\Gamma. \tag{22}$$

2.4 Sensitivities

In this section, we present the expressions required to compute analytical sensitivities of the compliance and volume fraction. For an optimization function θ that depends on the design exclusively and implicitly through the displacements, its sensitivity with respect to a design variable z_i is given by

$$\frac{\partial \theta}{\partial z_i} = \sum_{e=1}^{n_e} \frac{\partial \theta}{\partial \rho_e} \frac{\partial \rho_e}{\partial z_i}, \tag{23}$$

where n_e is the number of elements. Using adjoint analysis, and as customary in density-based topology optimization, for the compliance (i.e., $\theta \equiv c$) we have

$$\frac{\partial c}{\partial \rho_e} = -\frac{1}{\rho_e} u^{\mathrm{e}^{\mathrm{T}}} K^{\mathrm{e}} u^{\mathrm{e}} = -u^{\mathrm{e}^{\mathrm{T}}} K_0^{\mathrm{e}} u^{\mathrm{e}}, \tag{24}$$

where u^{e} is the vector of nodal displacements for element e, K^{e} is the element stiffness matrix computed using the ersatz material of (7), and K_0^{e} is the "fully solid" element stiffnes matrix corresponding to $\rho_e = 1$. It should be noted that K_0^{e} only needs to be computed once and for all in the optimization and stored in memory.

The sensitivity of the combined density, $\partial \rho_e / \partial z_i$ in (23), is obtained from (5) as

$$\frac{\partial \rho_e}{\partial z_i} = \sum_{b=1}^{n_b} \frac{\partial \rho_e}{\partial \widehat{\rho}_{be}} \frac{\partial \widehat{\rho}_{be}}{\partial z_i} = \sum_{b=1}^{n_b} \frac{\partial}{\partial \widehat{\rho}_{be}} \left(\widetilde{\max}_b \, \widehat{\rho}_{be} \right) \frac{\partial \widehat{\rho}_{be}}{\partial z_i}, \tag{25}$$

where $\widehat{\rho}_{be}$ is the penalized density for element e and bar b of (4). It should be noted that $\partial \widehat{\rho}_{be} / \partial z_i$ is zero when z_i does not

correspond to one of the design variables of bar b. The difference between "floating" and "connected" bars is that in the former case $\partial\widehat{\rho}_{be}/\partial z_i$ is non-zero for (at most) one bar and only the summand for the corresponding bar needs to be computed. The actual expression for the derivative of the smooth maximum depends of course on the particular function chosen.

The sensitivities of the penalized density $\widehat{\rho}_{be}$ are obtained from (4) as

$$\frac{\partial\widehat{\rho}_{be}}{\partial z_i} = \frac{\partial\mu}{\partial(\alpha_b\rho_{be})}\left(\alpha_{\mathrm{b}}\frac{\partial\rho_{\mathrm{be}}}{\partial z_{\mathrm{i}}} + \rho_{be}\frac{\partial\alpha_{\mathrm{b}}}{\partial z_{\mathrm{i}}}\right). \tag{26}$$

If z_i corresponds to one of the components of x_{1b} or x_{2b}, then $\partial\alpha_b/\partial z_i = 0$ and the second term on the right-hand side of (26) vanishes. Conversely, if $z_i \equiv \alpha_b$, then $\partial\rho_{be}/\partial z_i = 0$ and $\partial\alpha_b/\partial z_i = 1$, thus the first term on the right-hand side of (26) vanishes. Finally, if z_i does not correspond to one of the design parameters of bar b, then the entire expression is zero. Consequently, each of the terms in the right-hand side of (26) is computed separately for each bar, and the sensitivities are subsequently "assembled" into a matrix.

From (2), the sensitivity of the projected density ρ_{be} is given by

$$\frac{\partial\rho_{be}}{\partial z_i} = \begin{cases} \dfrac{1}{r}\dfrac{\partial\widetilde{H}}{\partial(\phi_{be}/r)}\dfrac{\partial\phi_{be}}{\partial z_i} & \text{if} -1 \le \phi_{be}/r \le 1 \\ 0 & \text{otherwise.} \end{cases} \tag{27}$$

This expression highlights the fact that the sensitivities of the projected density are non-zero only in a band of width $2r$ along the bar boundary. From (3), we find

$$\frac{\partial\widetilde{H}}{\partial x} = \begin{cases} 2\sqrt{1-x^2}/\pi & \text{in 2D} \\ 3\left(1-x^2\right)/4 & \text{in 3D,} \end{cases} \tag{28}$$

with $x = \phi_{be}/r$. The sensitivities of the signed distance of (8) are given by

$$\frac{\partial\phi_{be}}{\partial z_i} = \begin{cases} \dfrac{\partial d_{be}}{\partial z_i} & \text{if } z_i \equiv (x_{1b})_k \text{ or } (x_{2b})_k, k = 1, \cdots, n_d \\ -1 & \text{if } z_i \equiv r_b \\ 0 & \text{otherwise,} \end{cases} \tag{29}$$

Finally, the sensitivities of the distance d_{be} with respect to the bar endpoints x_{sb}, $s \in \{1, 2\}$ are given by

$$\frac{\partial d_{be}}{\partial \mathbf{x}_{sb}} = \frac{1}{d_{be}}\begin{cases} -x_{e/1b}\delta_1^s, & \text{if } \ell_{be} \le 0 \\ -x_{e/2b}\delta_2^s, & \text{if } \ell_{be} < \ell_b \\ -r_{be}\left(\delta_1^s + \dfrac{\ell_{be}}{\ell_b}\left(\delta_2^s - \delta_1^s\right)\right) & \text{otherwise,} \end{cases} \tag{30}$$

where $\delta_k^s := \{1 \text{ if } s = k, 0 \text{ otherwise}\}$ is the Kronecker delta.

It can be noted from (30) that $\partial d_{be}/\partial x_{sb}$ is undefined when $d_{be} = 0$ (Norato et al. 2015; Zhang et al. 2016a). In this case, an element centroid x_e lies exactly on the medial axis of bar b. This situation can be circumvented by making sure the sample window size is smaller than the bar's width, i.e., $r < r_b$, since an infinitesimal perturbation of the bar's medial axis endpoints will leave the projected density of any point on the medial axis unchanged ($\rho_{be} = 1$), and thus $\partial\rho_{be}/\partial x_{sb}$ in (27) must necessarily be zero.

When the bar's axis collapses to a point (so that $\ell_b = 0$), the sensitivities are still well defined, since the first branch of (30) applies. However, since the code still computes a_b/ℓ_b, we check ℓ_b against a small tolerance to prevent a division by zero.

The sensitivities of the volume fraction of (16) (i.e., $\theta \equiv v_f$) are computed as

$$\frac{\partial v_f}{\partial z_i} = \frac{1}{V}\sum_{e=1}^{n_e} v_e\frac{\partial\rho_e}{\partial z_i}, \tag{31}$$

where v_e is the volume of element e and $V = \sum_{e=1}^{n_e} v_e$ is the volume of the design region. The term $\partial\rho_e/\partial z_i$ is computed as before using (25–30), however with the clarification that the volume fraction must be computed with the unpenalized density. Consequently, $\partial\mu/\partial(\alpha_b\rho_{be}) = 1$ in (26).

Most of the equation numbers for this section have been added to comments in the code so that the user can readily find them (for instance, as shown in Appendix A).

2.5 Algorithm

We complete the presentation of the geometry projection formulation with the pseudo-code shown in Fig. 3. This algorithm presents a *conceptual* (but logically correct) flow of the steps taken to perform the optimization. However, as will be mentioned in the following section, an efficient implementation of the code does not exactly follow this pseudo-code. In Fig. 3, $\widehat{z}$ denotes the vector of design variables after they have been scaled so that they lie in the range [0, 1]. This scaling and the imposition of move limits on each design update are discussed in detail in Section 3.5.

3 Implementation

In this section, we discuss general aspects of the implementation, particularly in terms of functionality of the code. We also discuss in some detail the calculation of the geometry projection described in the previous section. The code was developed and tested using MATLAB, version R2018b. It was tested in the Red Hat Enterprise Linux 7.4, macOS Mojave, and Windows 10 operative systems. Because we make extensive use of vectorization, the code performs better in multi-core machines. The code is executed by running the script GPTO.m (without any arguments). To check that the program

Algorithm Topology Opt. via Geometry Projection

$k \leftarrow 0$ ▷ Iteration counter
$\hat{\mathbf{z}}^{(0)} \leftarrow \hat{\mathbf{z}}_0$ ▷ Initial design
repeat
  for $b = 1, \ldots, n_b$ **do**
    for $e = 1, \ldots, n_e$ **do**
      Compute signed distance ϕ_{be} from $\mathbf{x}_e$ to bar b ▷ (8)
      Compute projected density ρ_{be} ▷ (2)
      Compute penalized density $\hat{\rho}_{be}$ ▷ (4)
    end for
  end for
  for $e = 1, \ldots, n_e$ **do**
    Compute combined density ρ_e ▷ (5)
    Compute element stiffness matrix $\mathbf{K}^e$ using $\mathbb{C}(\mathbf{z})$ of (7)
    Assemble $\mathbf{K}^e$ into global stiffness matrix $\mathbf{K}$
  end for
  Solve $\mathbf{K}(\hat{\mathbf{z}})\mathbf{u}(\hat{\mathbf{z}}) = \mathbf{f}$ for $\mathbf{u}(\hat{\mathbf{z}})$ ▷ (17)
  Compute $c(\mathbf{u}(\hat{\mathbf{z}}))$, $\nabla_{\mathbf{z}} c(\mathbf{u}(\hat{\mathbf{z}}))$ ▷ (15), (24)
  Compute $v_f(\hat{\mathbf{z}})$, $\nabla_{\mathbf{z}} v_f(\hat{\mathbf{z}})$ ▷ (16), (31)
  Impose move limits and update $\hat{\mathbf{z}}_{low}$ and $\hat{\mathbf{z}}_{upp}$ ▷ (32), (33)
  $\hat{\mathbf{z}}^{(k+1)} \leftarrow \texttt{opt}(\hat{\mathbf{z}}^{(k)}, c, \nabla_{\mathbf{z}} c, v_f, \nabla_{\mathbf{z}} v_f, \hat{\mathbf{z}}_{low}, \hat{\mathbf{z}}_{upp})$ ▷ Update design
  $k \leftarrow k + 1$
until Any of the stopping criteria is satisfied ▷ §3.5

Fig. 3 Algorithm for topology optimization using geometry projection

is running, we suggest the user simply runs this script, which executes the optimization of a 2D cantilever beam (not described in this work).

It should be noted that performance will suffer drastically on non-Intel CPUs because Intel's math kernel library (MKL) disables by default all but the most basic vectorization on non-Intel CPUs. However, this can be circumvented by setting an environment variable: MKL_DEBUG_CPU_TYPE = 5.

3.1 Organization

Since we do not attempt to write the code in as concise a manner as possible, we are free to modularize it as much as possible, which we have attempted to do. Similar functions are placed under the same subfolder—for instance, those related to the finite element analysis, geometry projection, optimization, plotting, etc. This allows the user to more easily navigate the code, examine and focus on particular portions of the implementation, and make changes. We have adhered to the rule that one routine corresponds to one MATLAB script (i.e., no script has multiple functions declared in it). The modular structure also facilitates having multiple options for, e.g., the penalization function μ of (4), the aggregation function of (5), the mesh input or generation, etc. The root folder only contains the main script GPTO.m and another script to invoke the input functions (get_inputs.m); all other scripts are inside subfolders.

Another important aspect of our implementation is that we use three MATLAB structures, named FE, GEOM, and OPT, to store all information related to the finite element analysis, geometric components, and method parameters, respectively. These structures are declared as global variables in any script where they are needed. Having only a few structures and declaring them as global variables avoids having to pass long lists of arguments to the functions that need the information stored in them. If a new field is added to one of these structures, it becomes immediately available to any routine invoking the structure as a global variable. Using global variables is in general discouraged in modern compiled languages because they may not be thread-safe; however, since we are only using MATLAB's own vectorized operations, we assume that potential problems such as race conditions are managed by MATLAB (and we have to date not observed any issue related to this in our numerical experiments). Also, the use of global variables makes the code slightly more efficient, since local copies of the structures (some of which may be relatively large, such as FE) need not be created in each routine that uses them.

3.2 Inputs

The user must provide all of the inputs using MATLAB scripts. This strategy circumvents having to parse text files, allows in providing inputs in any order, facilitates the incorporation of comments, and makes it easier for the user to customize the code. Although not strictly necessary, we have placed all the input files for the sample problems into the subfolder \texttt (in fact, we created additional subfolders inside this subfolder for each one of the examples). To switch from running one example to another, the user needs only to update the single line with the run command in the inputs.m script located in the root folder to indicate the location of the master input file for the run. Having a script that calls the master input files makes it easy to keep different inputs for different runs. Note that the \input_files subfolder is not added to the search path out of precaution to avoid potential name conflicts and because it is not necessary for the code to run.

Since there are quite a few options in the program, the easiest way to create input files for a new run is to copy and modify the files for one of the sample problems. The input files for all problems are well commented to make it easy to modify them. All of the inputs are used to initialize the aforementioned three data structures that pertain to the finite element analysis, the bars that make up the initial design, and all parameters related to the geometry projection, the optimization problem, the optimizer, and the finite difference check of sensitivities (if requested).

3.3 Finite element analysis

The parameters relating to the finite element mesh are placed in the FE.mesh_input structure. Three options are available to create or read a mesh, which are indicated in the type field. The 'generate' option creates a mesh on the fly for rectangular

and cuboid design regions in 2D and 3D, respectively. The user must specify the dimensions of the region and the number of elements along each dimension using the box_dimensions and elements_per_side vectors, respectively. The second option, 'read-home-made', can be used to read a mesh that has been previously created (e.g., using the makemesh script provided in the code) and saved to a MATLAB .mat file, whose path must be provided in the field mesh_filename. The option 'read-gmsh' allows the user to read a mesh created with the open source software Gmsh (Geuzaine and Remacle 2009). The user must create a mesh made of quadrilateral or hexahedral elements in 2D or 3D, respectively (a transfinite mesh in Gmsh parlance); export it to MATLAB format; and indicate the path of this file in the gmsh_filename field. This third option is very convenient for design regions that are not cuboid-shaped.

The mesh only contains the node locations and the element connectivity. Displacement boundary conditions and loads must be specified in a separate MATLAB script that the user must create and whose path must be specified in the bcs_file field. This separation facilitates using the same mesh for different problems with different boundary conditions. As before, it is easier to copy the file for one of the sample problems and modify it. For cuboid-shaped meshes, the compute_predefined_node_sets function provides a very convenient utility to retrieve node sets corresponding to prescribed points, edges, and faces of the domain, which can be subsequently used to impose displacement boundary conditions or to apply loads. These node sets are stored in the FE.node_set structure. For example, for a 2D rectangular domain, the set BR_PT contains the node in the bottom-right corner of the domain, and the set L_edge contains all nodes on the left-hand side edge of the domain. Needless to say, the user must ensure that the problem is sufficiently restrained to get a well-posed analysis.

The code for the finite element analysis closely follows the sparse data structures and vectorized operations presented in Andreassen et al. (2011). It provides the option to use a direct or an iterative solver through the parameter FE.analysis.solver.type. In the former case, we simply use MATLAB's matrix left division operator "\", which uses Cholesky factorization. In the latter case, which may be useful for larger systems, we use the preconditioned conjugate gradient solver with an incomplete Cholesky preconditioner (using MATLAB's pcg and ichol functions, respectively). For the iterative solver, the user must specify the convergence tolerance and the maximum number of iterations in the tol and maxit fields, respectively.

The code also has an option to use a GPU card to solve the system of linear equations, by setting the parameter FE.analysis.solver.use_gpu to true, which can be used if the system has an NVIDIA GPU card. In this case, only the iterative solver can be used, and a Jacobi preconditioner is used instead of the incomplete Cholesky preconditioner, since using the latter incurs a high data transfer cost to the GPU memory and is inefficient. The trade-off is that the iterative solution requires more iterations with the Jacobi preconditioner; nevertheless, for large meshes, it is still faster than using the iterative solver with CPUs.

Table 1 shows the time it takes to perform one optimization iteration using CPUs and GPUs in a system with an NVIDIA GTX 1070 Max-Q GPU card and an Intel core i7 8750H (with 6 cores) running on Ubuntu 19.10. The times correspond to the average iteration time of the first two iterations for the optimization of the default problem in the program (a 2D cantilever beam with eight bars). These times thus include not only the finite element analysis but also the geometry projection and the design update by the optimizer. Although the Jacobi preconditioner requires far more PCG iterations to converge, it still outperforms the use of CPUs for these mesh sizes (in this system). Also, it should be noted that the time and number of iterations will vary throughout the optimization, since different designs will lead to different condition numbers for the system matrix. As the table shows, it can be very convenient to use GPUs in this code. For instance, a mesh with half a million elements requires about one and a half minutes per iteration, therefore, 100 iterations of the optimization can be completed in approximately two and a half hours, which is—presently—very good for performing topology optimization in MATLAB on an individual workstation (recalling, however, that this is a 2D model).

3.4 Initial design

Through the GEOM.initial_design.path field in the master input file, the user must specify the path of the script containing the specification of the bars that make up the initial design. This script contains two arrays that describe the initial layout of bars. The point_matrix array has as many rows as points, with the first column containing an integer identifier for the point, and the remaining columns containing the spatial coordinates of the point. The bar_matrix array has as many rows as

Table 1 Average optimization iteration times for different size meshes of a 2D problem

Device	Mesh size	t (s)	n_{PCG}
cpu	320,000	61.6	1.2 k
gpu	320,000	71.9	6 k
cpu	500,000	115.2	1.5 k
gpu	500,000	90.2	7 k
cpu	749,088	198.4	1.8 k
gpu	749,088	152.9	8 k
cpu	999,698	301.7	2 k
gpu	999,698	223.2	9.5 k

n_{PCG} is an approximate average number of PCG iterations to convergence

bars. Its first column corresponds to an integer identifier for the bar; the next two columns have the IDs of the points in the point_matrix array that correspond to the endpoints of the bar's medial axis; and the fourth and fifth columns contain the initial values of the bar's size variable and its radius, respectively.

It should be noted that this specification of points and bars allows for two possibilities: bars can be "floating" or "connected," as detailed in Section 2.2. The computation of sensitivities (cf. (25)) in the code accounts for both situations.

As noted in Section 2.4, for sensitivities of the projected density of (2) to be well defined, it is necessary that $r < r_b$. Since we typically consider a sample window that at least covers each finite element, as detailed in Section 3.9, this requirement imposes a minimum element size. For example, if $r_{b_{min}} = 1$, and the sample window radius r is chosen to be twice the size of the element radius r_e, then $r_e < 1/2$ and therefore the element size h_e should be at most $\sqrt{2}/2$ in 2D and $\sqrt{3}/2$ in 3D. To avoid undefined sensitivities in the initial design, it is also necessary that the length of each bar is not zero, i.e., $x_{1b} \neq x_{2b}$. Therefore, the user should not create initial designs with zero-length bars.

When running the optimization, the code saves the current design (i.e., the arrays of points and bars) to a MATLAB .mat file. If the flag GEOM.initial_design.restart is set to true, the code reads the initial design from this file. This is a useful feature to start an optimization from the design obtained by another optimization run or to perform a finite difference check of the sensitivities (as detailed in Section 3.7) on the final design of a run. We note, however, that this does not constitute a clean continuation of a previous optimization run, since gradient-based optimizers (certainly the ones used in this code) use information from two or more consecutive iterations to construct approximations of the optimization functions.

3.5 Optimization

Although the code in its current form is written to solve the optimization problem of (15)–(20), it is also structured so that (a) multiple constraints can be imposed and (b) any function can be chosen as the objective. This is achieved by indicating in the master input file the name of the objective function (e.g., OPT.functions.objective = 'compliance';) and the names and limits of the constraint functions (e.g., OPT.functions.constraints={'volume fraction', 'my constraint'}; and OPT.functions.constraint_limit = [0.3 1.0];). In this example, the user has to implement the function named 'my constraint' and add it to the list of available functions in the script init_optimization.m. This should facilitate the modification of our code to address other optimization problems.

As opposed to density-based topology optimization, the magnitudes of the design variables in the geometry projection scheme can differ greatly. Moreover, the optimization functions can be highly nonlinear with respect to the design variables, and therefore the optimizer can take too large a design step and produce poor iterates. To improve the performance of the optimization, we thus impose move limits on the design variables. In order to impose the same relative move limit on all variables, we first scale the design variables as

$$\hat{z}_i = \frac{z_i - \underline{\hat{z}_i}}{\overline{\hat{z}_i} - \underline{\hat{z}_i}}, \tag{32}$$

where $\hat{z}_i$ is the scaled variable i and $\underline{\hat{z}_i}$ and $\overline{\hat{z}_i}$ are lower and upper bounds on the variables, respectively. For the components of the medial axis endpoints x_{1b} and x_{2b}, these bounds are given by the bounding box of the design region; for the bar radius, they correspond to the values specified by the user in GEOM.min_bar_radius and GEOM.max_bar_radius; and the size variables do not need any scaling. Variable scaling can be turned on or off using the parameter OPT.options.dv_scaling in the master input file. A move limit m is imposed on the design variables at iteration I as

$$\max\left(0, \hat{z}_i^{(I-1)} - m\right) \leq \hat{z}_i^{(I)} \leq \min\left(1, \hat{z}_i^{(I-1)} + m\right). \tag{33}$$

The move limit value is assigned with the parameters OPT.options.move_limit in the master input file.

Two optimizers can be used with our code: MATLAB's fmincon and the method of moving asymptotes (MMA) (Svanberg 1987, 2002, 2007). The choice of optimizer is made with the parameter OPT.options.optimizer in the master input file. In the case of fmincon, we employ the active-set algorithm, since it allows us to impose move limits (through the 'RelLineSrchBnd' option as a relative bound on the line search step length), and it performs well in our numerical experiments. fmincon handles well the situation where the lower bounds on a variable equal the upper bounds, and therefore a fixed bar radius can be imposed if desired by setting GEOM.min_bar_radius = GEOM.max_bar_radius.

In addition to scaling the design variables, it is important to remember that in the case of MMA, it is recommended that the constraint limits and the objective function are between 1 and 100 for reasonable values of the design variables. The volume fraction automatically satisfies this requirement, but the compliance does not. If the magnitude of the applied loading is too large and/or the elasticity modulus of the material is relatively low, the compliance may easily exceed this range. The minimum compliance design for a given volume fraction does not depend on the magnitude of the load or the elastic modulus. Therefore, the user can modify, for example, the load magnitude to make sure the compliance is within the recommended

range. Another, more general strategy is to simply scale the compliance and its sensitivities by some factor, which requires modifying the compute_compliance.m script accordingly. Ensuring proper scaling is important, as MMA may fail altogether if the magnitude of the compliance is very large.

In the case of MMA, we provide the code to call it in the script runmma.m, but the user must obtain the MATLAB version of MMA from its author and copy the files mmasub.m, subsolv.m, and kktcheck.m in the optimization folder of our code. We employed the 2007 version of MMA (Svanberg (2007)) for numerical tests with our code. Unlike fmincon, MMA does not handle well the situation where the lower and upper bounds of a variable are the same. Therefore, to approximately impose a fixed bar radius, the user should set GEOM.max_bar_radius=GEOM.min_bar_radius+delta, where delta is a small positive number.

The code employs three stopping criteria. The first criterion is satisfied if the 2-norm of the change in the vector of design variables falls below a specified value, indicated by the parameter OPT.options.step_tol. The second is satisfied if the norm of the Karush-Kuhn-Tucker optimality conditions falls below the value specified in the parameter OPT.options.kkt_tol. The third criterion is exceeding a maximum number of iterations, given by the parameter OPT.options.max_iter. If *any* of these criteria are satisfied, the optimization is stopped.

3.6 Output

The program produces several forms of output. During the course of the optimization, two MATLAB figures are created and updated at every iteration. The first is a plot of the bars drawn by using directly the geometric parameters as rectangles with semicircular ends and colored such that the transparency indicates the value of the size variable. A bar with a size variable value of unity has no transparency, and bars with a size variable value less than 0.05 are removed from the plot.

The second figure is a plot of the combined density of (5). In the case that fmincon is used as optimizer, this figure has two buttons to either pause or stop the optimization. At the end of the optimization, a third figure is produced that plots the iteration history of the objective and constraint functions. The generation of these plots can be turned off by setting the variable plot_cond to false in the master input file; this option is convenient when, for example, the code is executed in a queue in a high-performance computing system. The code also writes information on the optimization iterations to MATLAB's Command Window.

Finally, our code writes Visualization Toolkit (.vtk) files of the combined density of the design to the folder specified in the OPT.options.vtk_output_path parameter (which by default is set to 'output_files'). The user can request VTK output for none, all, or the last iteration by setting the parameter OPT.options.write_to_vtk to 'none', 'all', or 'last', respectively. These files can be visualized with Open Source tools like ParaView (Ahrens et al. 2005; Ayachit 2015), which offer wide post-processing capabilities. This is particularly useful for 3D problems.

3.7 Finite difference check

Another utility provided by our code is a forward finite difference check of the analytical sensitivities of the cost and/ or objective functions. The finite difference check is turned on by setting the parameter OPT.make_fd_check to 'true' in the master input file. If this option is chosen, the code performs the finite difference check and subsequently exits. The size of the finite difference step is specified in the parameter OPT.fd_step_size. The user can choose what function should be checked by setting the parameters check_cost_sens and check_cons_sens to 'true' or 'false'. This functionality is useful when developing new optimization functions to ensure the accuracy of the analytical sensitivities.

3.8 Distance calculation

The computation of the distance d_{be} of (14) from the centroid x_e of each element e to the medial axis of each bar b requires, conceptually, a double for loop, as shown in Fig. 3. This computation, which is of order $O(n_e n_b)$, can be quite expensive; however, it fortunately is embarrassingly parallel. In distributed memory implementations, the elements in the mesh can be divided among the available compute cores and each core calculates the distance from those elements to all of the bars (see, e.g., Zhang et al. (2016a)). This computation scales linearly with number of cores, and therefore if enough cores are employed for the distance computation (as well as the calculation of the combined density discussed in the following section), eventually the finite element analysis dominates the cost of each function evaluation as the number of elements increases, and the geometry projection only represents a small fraction of the cost.

When running on a single workstation with multiple cores, MATLAB employs shared-memory processing for many of its vectorized operations. Therefore, as discussed in Andreassen et al. (2011), one of the keys to efficient implementations of numerical methods in MATLAB is to vectorize the computations as much as possible. In particular, whenever a loop can be replaced with a vectorized operation, the improvements in performance can be drastic.

In the case of the distance calculation, the strategy we used to vectorize the double for loop is to employ three-dimensional arrays, as can be seen in the excerpt of the distance computation script shown in Appendix A. For instance, the array x_e_1b (line 36 in the code shown in the Appendix) contains all the vectors $x_{e/1b}$, used in (12–14). The first

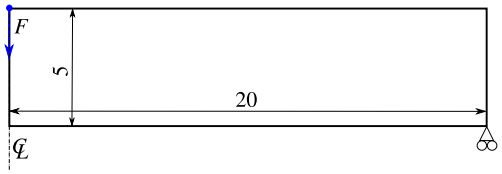

Fig. 4 MBB beam problem

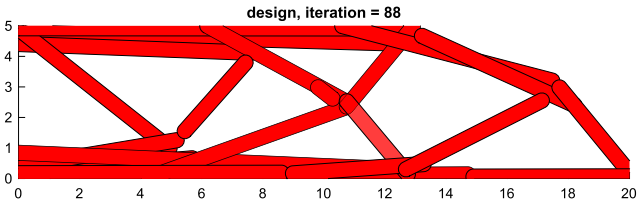

Fig. 6 Optimal design for MBB beam

dimension of the array corresponds to the spatial components in the vector (of size 2 and 3 in 2D and 3D problems, respectively); the second dimension corresponds to the bar; and the last to the element. Note that even the computation of the branched function of 2 is vectorized by using Boolean matrices of dimensions $1 \times n_b \times n_e$ whose components equal true if the condition for the corresponding branch is satisfied and false otherwise (which are equivalent to 1 and 0, respectively).

As the problem size increases, accessing three-dimensional matrices can get quite slow (particularly if they have to be stored out of memory); therefore, eventually this approach becomes inefficient. However, in our numerical experiments (and using our hardware), we have been able to run problems with meshes that have hundreds of thousands of elements and tens of bars in reasonable times, and therefore this code should in many cases be efficient enough for method development (indeed, most publications with methods to perform topology optimization with geometric components use mesh sizes well within this range).

Another important aspect of the code that is not explicitly stated in the pseudo-code of Fig. 3 is that the sensitivities of each quantity are computed in the same script where the quantity is computed, and they are then stored in the global structures or passed to the calling function so that the chain rule is used for subsequent sensitivity calculations. For instance, the derivatives of the distance d_{be} in the script of Fig. 19 with respect to the design parameters are computed in the same script. This makes for a more modular code and makes it easier to verify the correctness of the individual terms of the sensitivities. As with the distance calculation, the computation of the sensitivities makes extensive use of vectorization for efficiency.

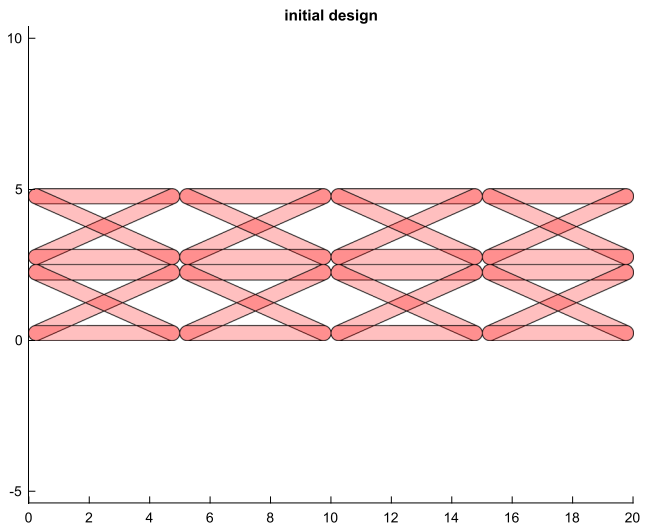

Fig. 5 Initial design for MBB beam

3.9 Combined density calculation

The computation of the combined density of (5) is also vectorized for efficiency. By default, the radius r_e of the sample window in (2) is computed as the radius of the circle (in 2D) or sphere (in 3D) that circumscribes the square or cubic element, respectively, with the same volume of the element:

$$r_e = \frac{\sqrt{d}}{2}(v_e)^{\frac{1}{d}} \tag{34}$$

where $d = 2, 3$ for 2D and 3D, respectively. This default can be overridden by uncommenting the line with the parameter OPT.parameters.elem_r in the master input file and assigning to this parameter the actual value of the sample window radius. In this case, the same radius will be used for all elements in the mesh.

The code has two options for the penalization function μ of (4), implemented in the script penalize.m: the SIMP penalization mentioned in Section 2.1 and the rational approximation of material properties (RAMP) (Stolpe and Svanberg 2001). The choice of penalization scheme is indicated by the parameter OPT.parameters.penalization_scheme in the master input file, and the penalization parameter is assigned in OPT.parameters.penalization_param.

Several combination strategies are also available to compute the smooth maximum $\widetilde{\max}$ of (5), implemented in the script smooth_max.m. These include the modified p-norm of

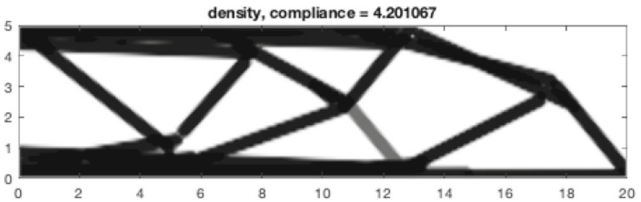

Fig. 7 Combined density of optimal design for MBB beam

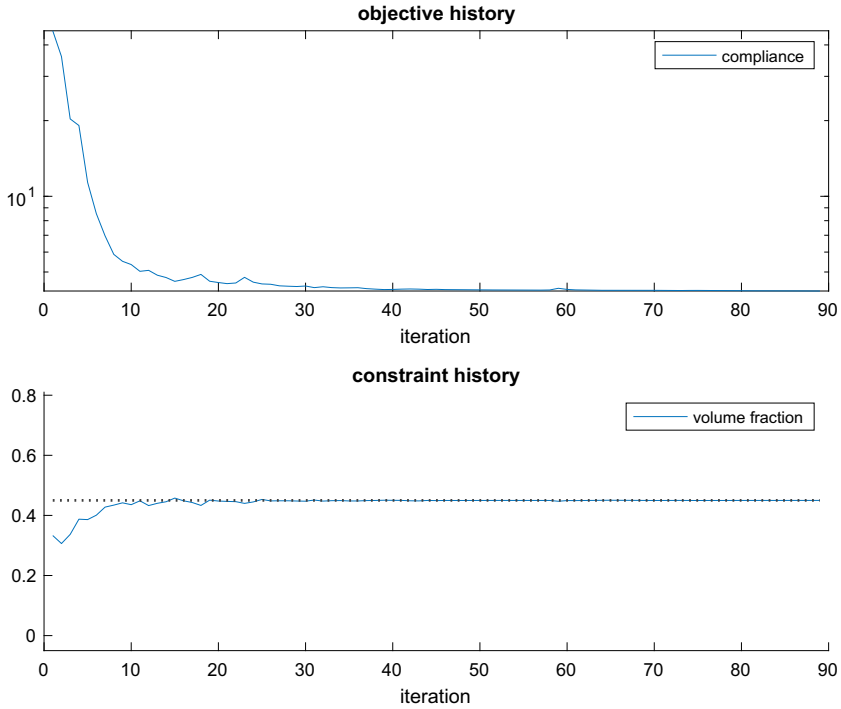

Fig. 8 Optimization history of compliance and volume fraction for MBB beam problem

(6), a modified p-mean, and lower- and upper-bound Kreisselmeier-Steinhauser approximations. The choice of combination function is indicated in the parameter OPT.parameters.smooth_max_scheme, and the aggregation parameter (e.g., p in the p-norm) is assigned in the variable OPT.parameters.smooth_max_param. Once again, we have made these functions modular so that the user can experiment with different choices or implement their own.

4 Examples

In this section, we present several examples to demonstrate some capabilities of the program. The purpose of these examples is not to demonstrate the geometry projection method but the functionality of the code. The input files for all the examples in this section are distributed with the code.

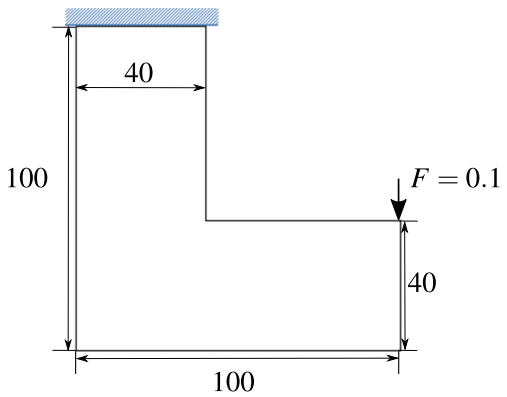

Fig. 9 2D L-bracket problem

4.1 MBB beam

The first example corresponds to the widely studied Messerschmitt-Bölkow-Blohm (MBB) beam. The dimensions, displacement boundary conditions (BCs), and load for the MBB problem are shown in Fig. 4. The magnitude of the load is $F = 0.1$. The problem is symmetric with respect to the centerline (only the right-hand side is shown in the figure). The volume fraction constraint for this problem is $v_f^* = 0.45$.

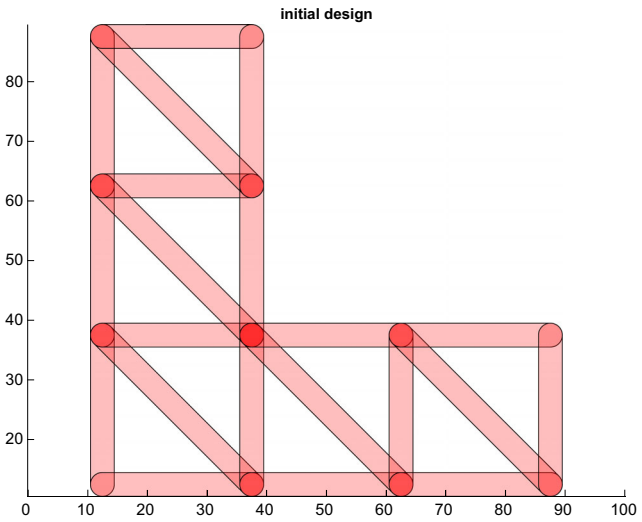

Fig. 10 Initial design for L-bracket

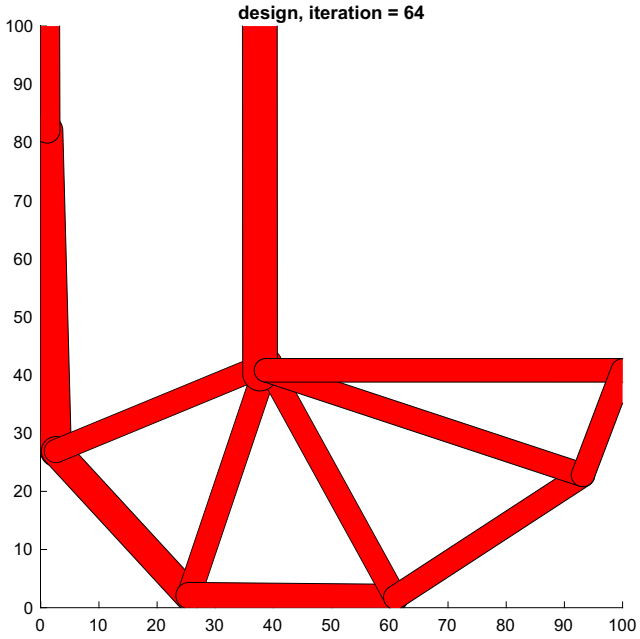

Fig. 11 Optimal design for L-bracket

The reference manual that accompanies the code uses this example as a tutorial. It provides step-by-step instructions to copy the input files of the default 2D cantilever problem and modify them to solve the MBB design problem.

For this problem, we employ a mesh of square elements of side 0.1. Bounds on the bars' radii are imposed to enforce a fixed bar radius $r_b = 0.25$. The optimization is performed using MMA. The initial design is made of 32 "floating" bars, shown in Fig. 5.

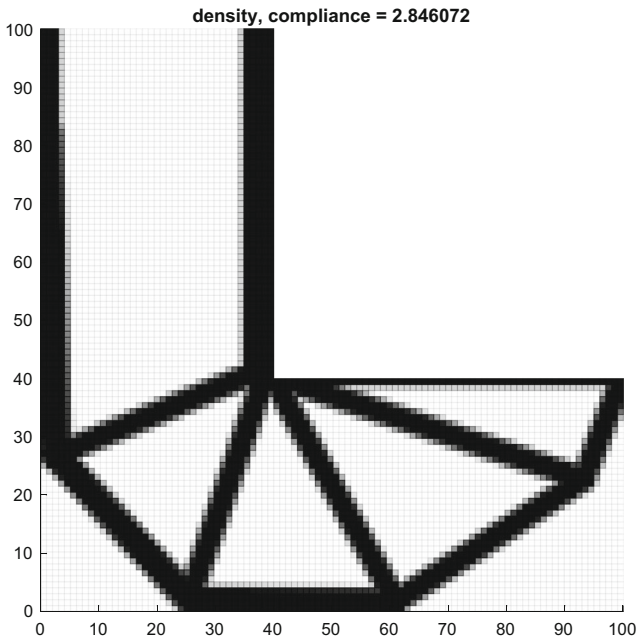

Fig. 12 Combined density of optimal design for L-bracket

The optimal design, shown in Fig. 6, is obtained in 88 iterations and in 80 s using a Mac Pro with a 3.5 GHz 6-Core Intel Xeon E5, running macOS Catalina 10.15.1, and MATLAB R2019b. The combined density of the optimal design is shown in Fig. 7. The transparency of the bars in this figures indicates their size variable: a bar not shown (fully transparent) has $\alpha_b = 0$, while a bar with no transparency has $\alpha_b = 1$. The initial design for this and all examples uses $\alpha_b = 0.5$ for all bars, and so the bars appear half transparent. The optimization history of the compliance (objective) and volume fraction (constraint) is plotted in Fig. 8. All these figures are produced by the code.

4.2 L-bracket

In this example, we minimize the compliance of the L-bracket problem shown in Fig. 9 subject to a volume fraction constraint of $v_f^* = 0.3$. The initial design for this example, shown in Fig. 10, is made of 21 *connected* bars with 12 shared endpoints. Since connected bars share the endpoints, the connectivity of the structure remains the same throughout the optimization (although bars are removed by setting their size variables to zero). Therefore, this is akin to ground structure approaches used in topology optimization with 1D elements. Bounds on the bars' radii of $2 \leq r_b \leq 3$ are imposed. MMA is used as an optimizer.

Another feature of this example is that the mesh was created with Gmsh and exported to MATLAB. The Gmsh .geo file used to create the mesh is provided in the code. When the mesh is exported to MATLAB, Gmsh creates a .m file with a structure called msh with all the mesh information. GPTO reads this file to populate the GEOM structure.

The optimal design and its corresponding combined density and the optimization history of the objective and constraint functions are shown in Figs. 11, 12, and 13, respectively. It should be noted that due to the non-convex design region in this example, it is possible that a bar lies partially outside of the design region so that the bar is "broken" into two parts. The work in Zhang et al. (2018) introduces a constraint in the optimization to prevent solid geometric components from exiting the design region. However, we do not include this constraint in this work for the sake of simplicity.

We also use this example to demonstrate another feature of our code: we perform a finite difference check of the compliance for the final design in the optimization. First, to set the design for the finite difference check to the final design of the optimization, the parameter GEOM.initial_design.restart must be set to true, and GEOM.initial_design.path must be set to the path and name of the .mat file created by the code with the same name of the initial design .m file (located in the same folder of the

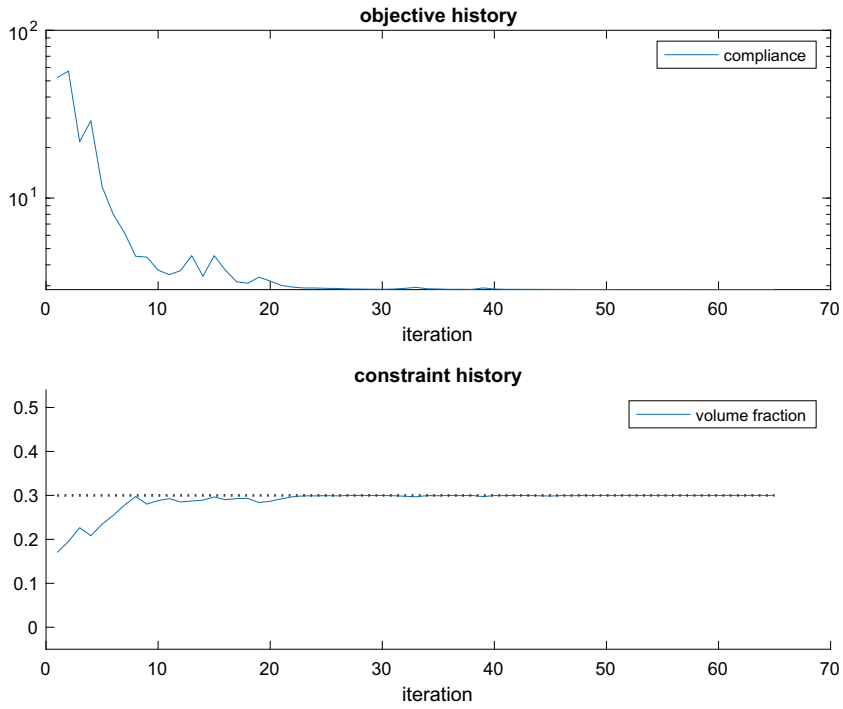

Fig. 13 Optimization history of compliance and volume fraction for L-bracket problem

master input file). Second, the parameters OPT.make_fd_check and OPT.check_cost_sens must be set to true. The perturbation size for the finite difference check is set as OPT.fd_step_size=1e-6;.

The largest absolute and relative differences between the analytical sensitivities and the finite difference sensitivities are reported to MATLAB's Command Window as − 0.0037 and − 0.0013, respectively, indicating a good agreement. They correspond to the x component of the eighth point, which is the point closest to the right-hand side corner of the top edge. The code also produces a plot comparing the two sensitivities, shown in Fig. 14.

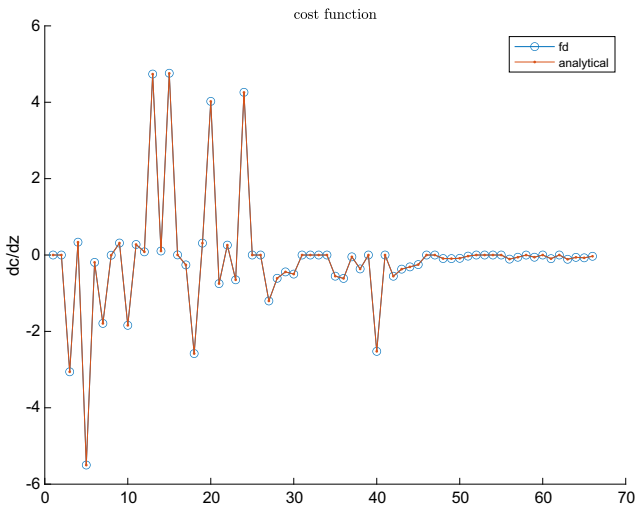

Fig. 14 Finite difference check of the compliance sensitivities for the L-bracket problem

4.3 3D cantilever beam

The last example corresponds to a 3D cantilever beam with fixed supports on all four corners at one end, and a tip downward load in the center of the opposite face, as shown in Fig. 15. The magnitude of the force is $F = 0.1$. We minimize the compliance subject to a volume fraction constraint of $v_f^* = 0.1$. The initial design for this example, shown in Fig. 16, is made of 16 floating bars with 32 points. Bounds on the bars' radii of $0.5 \leq r_b \leq 1.0$ are imposed and MMA is used as an optimizer.

The mesh for this problem is generated automatically, and it consists of $80 \times 40 \times 40 = 128000$ elements. This problem is solved using GPUs on a machine with 24 Intel Xeon CPUs at 2.2 GHz, 32 Gb of RAM, and an NVIDIA Quadro M2000 GPU card; running on Ubuntu 18.04.3 LTS; and using MATLAB R2019b.

The problem was solved in 63 min, and it took 106 iterations to convergence. The optimal design for this problem is

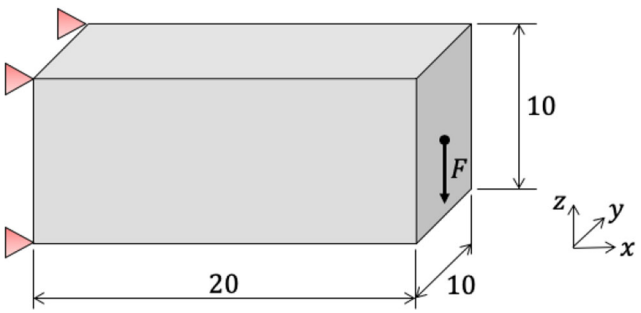

Fig. 15 Initial design for 3D cantilever beam

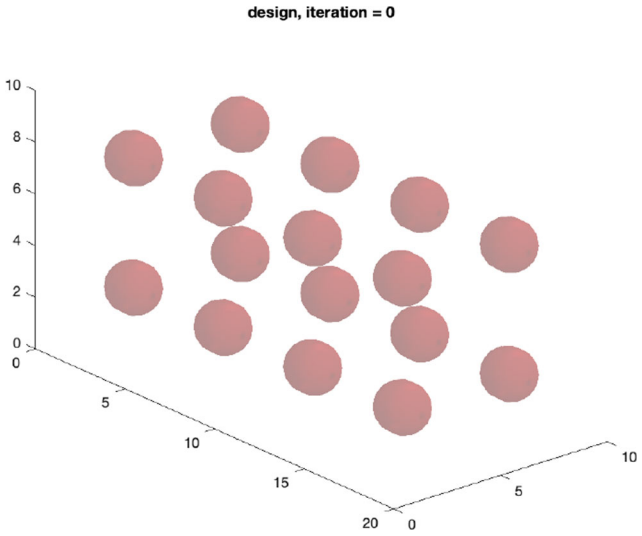

Fig. 16 Initial design for 3D cantilever beam

shown in Fig. 17. An isosurface plot of the combined density for the optimal design ($\rho = 0.5$) produced using ParaView is shown in Fig. 18. This figure is produced by opening the file dens106.vtk file created by the code and saved to the /output_files folder, subsequently applying the following filters in this order: Clean to Grid; Cell Data to Point Data; Clean to Grid; Clip, changing the type to scalar and setting the value to 0.5; Extract Surface; and Generate Surface Normal.

5 Conclusions

This paper presents a MATLAB code to perform topology optimization of 2D and 3D structures made of cylindrical bars by using geometry projection. The formulation of the method

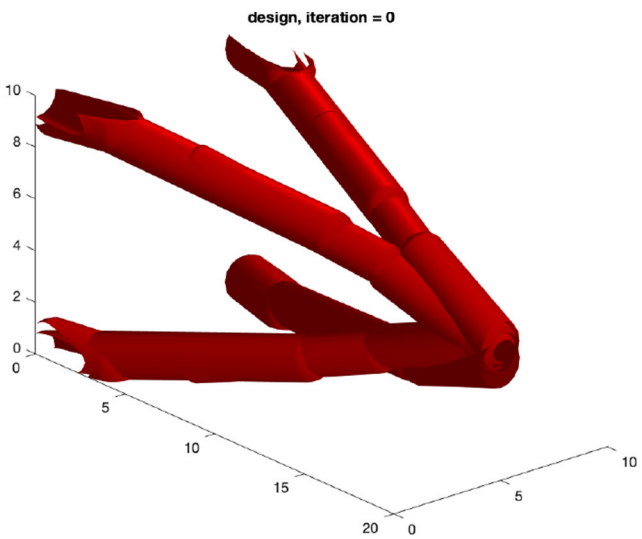

Fig. 17 Optimal design for 3D cantilever beam

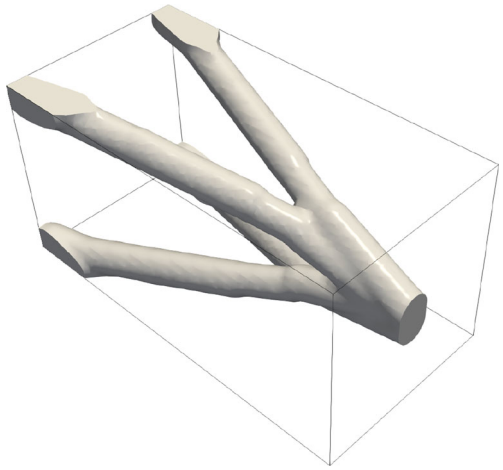

Fig. 18 Combined density isosurface for optimal design of 3D cantilever beam

is presented in full along with implementation details to aid users in understanding and using the code. The program is written in a modular manner to facilitate modification and experimentation. Vectorization is used extensively to render an efficient code. The presented examples demonstrate some of the most important features of the code. The program is accompanied by a reference manual with a step-by-step tutorial to reproduce the first example in this paper. The authors hope this is a useful contribution to the community and welcome comments on this code.

Acknowledgments The authors express their gratitude to Prof. Krister Svanberg for providing his MMA MATLAB optimizer to perform the optimization.

Funding information The authors express their gratitude to the US Office of Naval Research (Grant Number N00014-17-1-2505) for the support to conduct this work.

Compliance with ethical standards

Conflict of interests statement The authors declare that they have no conflict of interest.

Replication of results All the results presented in this work can be reproduced with the MATLAB code available from GitHub.

Appendix: code for distance calculation

Figure 19 shows the first part of the compute_bar_elem_distance.m script that computes the distance between the centroid of every element in the mesh to the medial axis of each of the bars. The latter portion of the code, which computes the derivatives of the distance with respect to the medial axes endpoint x_{1b} and x_{2b} is omitted for brevity.

```
function [dist,Ddist_Dbar_ends] = compute_bar_elem_distance()
%
% This function computes an array dist of dimensions n_bar x n_elem with
% the distance from the centroid of each element to each bar's medial axis.
%
% Ddist_Dbar_ends is a 3-dimensional array of dimensions n_bar_dofs x n_bar
% x n_elem that contains the sensitivities of the signed distances in dist
% with respect to each of the n__bar_dofs coordinates of the medial axis
% end points.

%%
global FE GEOM OPT

%% set parameters
tol = 1e-12; % tolerance on the length of a bar

n_elem = FE.n_elem;
dim = FE.dim;
n_bar = GEOM.n_bar;
n_bar_dofs = 2*dim;

% The following code is vectorized over the bars and the elements. We
% have to be consistent with the order of indices to perform element-
% wise array operations. The order of indices is (dim,bar,element)

points = GEOM.current_design.point_matrix(:,2:end).';
x_1b = points(OPT.bar_dv(1:dim,:));       % (i,b)
x_2b = points(OPT.bar_dv(dim+1:2*dim,:)); % (i,b)
x_e = permute(FE.centroids ,[1,3,2]);     % (i,1,e)

a_b = x_2b - x_1b; % Numerator of Eq. (11)
l_b = sqrt(sum(a_b.^2, 1)); % length of the bars, Eq. (10)
l_b(l_b < tol) = 1; % To avoid division by zero
a_b = a_b./l_b;  % normalize the bar direction to unit vector, Eq. (11)

x_e_1b = x_e - x_1b;                  % (i,b,e)
x_e_2b = x_e - x_2b;                  % (i,b,e)
norm_x_e_1b = sqrt(sum(x_e_1b.^2, 1)); % (1,b,e)
norm_x_e_2b = sqrt(sum(x_e_2b.^2, 1)); % (1,b,e)

l_be = sum(x_e_1b.*a_b, 1);       % (1,b,e), Eq. (12)
vec_r_be = x_e_1b - l_be.*a_b;    % (i,b,e)
r_be = sqrt(sum(vec_r_be.^2, 1)); % (1,b,e), Eq. (13)

branch1 = l_be <= 0.0;            % (1,b,e)
branch2 = l_be > l_b;             % (1,b,e)
branch3 = ~(branch1 | branch2);   % (1,b,e)

% Compute the distances, Eq. (14)
dist_tmp =  branch1.* norm_x_e_1b + ...
branch2.* norm_x_e_2b + ...
branch3.* r_be;           % (1,b,e)

dist = permute(dist_tmp,[2,3,1]); % (b,e)
```

Fig. 19 Excerpt of distance computation script (the portion with sensitivities calculation is omitted for brevity)

References

Ahrens J, Geveci B, Law C (2005) Paraview: an end-user tool for large data visualization. The visualization handbook 717

Andreassen E, Clausen A, Schevenels M, Lazarov BS, Sigmund O (2011) Efficient topology optimization in matlab using 88 lines of code. Struct Multidiscip Optim 43(1):1–16

Ayachit U (2015) The Paraview guide: a parallel visualization application. Kitware, Inc.

Bell B, Norato J, Tortorelli D (2012) A geometry projection method for continuum-based topology optimization of structures. In: 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference

Bendsøe MP, Sigmund O (1999) Material interpolation schemes in topology optimization. Arch Appl Mech 69(9–10):635–654

Geuzaine C, Remacle JF (2009) Gmsh: a 3-d finite element mesh generator with built-in pre-and post-processing facilities. Int J Numer Methods Eng 79(11):1309–1331

Guo X, Zhang W, Zhong W (2014) Doing topology optimization explicitly and geometrically—a new moving morphable components based framework. J Appl Mech 81(8)

Kazemi H, Vaziri A, Norato JA (2018) Topology optimization of structures made of discrete geometric components with different materials. J Mech Des 140(11):111,401

Norato J, Haber R, Tortorelli D, Bendsøe MP (2004) A geometry projection method for shape optimization. Int J Numer Methods Eng 60(14):2289–2312

Norato J, Bell B, Tortorelli D (2015) A geometry projection method for continuum-based topology optimization with discrete elements. Comput Methods Appl Mech Eng 293:306–327

Norato JA (2018) Topology optimization with supershapes. Struct Multidiscip Optim 58(2):415–434

Sigmund O, Maute K (2013) Topology optimization approaches. Struct Multidiscip Optim 48(6):1031–1055

Stolpe M, Svanberg K (2001) An alternative interpolation scheme for minimum compliance topology optimization. Struct Multidiscip Optim 22(2):116–124

Svanberg K (1987) The method of moving asymptotes—a new method for structural optimization. Int J Numer Methods Eng 24(2):359–373

Svanberg K (2002) A class of globally convergent optimization methods based on conservative convex separable approximations. SIAM J Optim 12(2):555–573

Svanberg K (2007) MMA and GCMMA, versions september 2007. Optim Syst Theory 104

Watts S, Tortorelli DA (2017) A geometric projection method for designing three-dimensional open lattices with inverse homogenization. Int J Numer Methods Eng 112(11):1564–1588

Wein F, Dunning P, Norato JA (2019) A review on feature-mapping methods for structural optimization. arXiv 1910.10770

Zhang S, Norato JA, Gain AL, Lyu N (2016a) A geometry projection method for the topology optimization of plate structures. Struct Multidiscip Optim 54(5):1173–1190

Zhang S, Gain AL, Norato JA (2018) A geometry projection method for the topology optimization of curved plate structures with placement bounds. Int J Numer Methods Eng 114(2):128–146

Zhang W, Yuan J, Zhang J, Guo X (2016b) A new topology optimization approach based on moving morphable components (MMC) and the ersatz material model. Struct Multidiscip Optim 53(6):1243–1260