Dataset Documentation

Topology Optimization of a Cantilever Beam

Laboratory of Topology Optimization and Multiphysics Analysis
Department of Computational Mechanics
School of Mechanical Engineering
University of Campinas (Brazil)

Author : Daniel Candeloro Cunha
Version : 1.0
Date : May 2022

Contents

1 Introduction

1.1 General Description

This work is part of the PhD thesis entitled *Análise de Sensibilidade de Variação Finita assistida por Redes Neurais Artificiais para Concepção de Metamateriais* (Finite Variation Sensitivity Analysis assisted by Artificial Neural Networks for Designing Metamaterials). The author Daniel Candeloro Cunha and his supervisor Professor Renato Pavanello are researchers at the Laboratory of Topology Optimization and Multiphysics Analysis, at the University of Campinas (Brazil).

The objective of the presented programs is to generate datasets that will be used to train artificial neural networks. The purpose of such networks is to improve the performance of standard topology optimization programs, by reducing computational costs, making the procedures more stable, or more accurate.

The topology optimization of a cantilever beam is considered. Four free parameters are used to define the boundary conditions of the problem. For the specified mesh, there are 296 208 unique sets of parameters, to generate the complete datasets, all the 148 240 non-redundant cases must be optimized. The complementary cases can be obtained through simple operations described in this document (but not implemented in this version).

Each case is optimized twice, using different optimization methods. Therefore, $296\,480 = 2 \times 148\,240$ optimizations are performed to generate the two datasets (one for each optimization method). For each iteration of each case, all results are stored: topology vectors; sensitivity vectors; displacements vectors; objective function values; volume fraction values; gray level values. Also, metadata is stored with relevant information, for example, the corresponding input parameters of each result. Together, both datasets occupy around 980 GB of disk.

- To collaborate or report bugs, you may contact the author: **cunhadc [at] unicamp [dot] br**
- All codes and documentation are publicly available in the following github repository: https://github.com/Joquempo/Cantilever-Dataset

If you use the presented programs (or the data generated by it) in your work, the developer would be grateful if you would cite the indicated references. They are listed in the "CITEAS" file available in the github repository.

1.2 Overview

In section 2, the optimization problem is described, as well as the considered free parameters that define the boundary conditions. Then, both optimization methods are presented.

In section 3, a user guide is presented, explaining how to use the provided programs to generate the datasets. Then, each script is presented and briefly explained. Lastly, some validation procedures are described, the corresponding scripts are available in the github repository, but they are not shown in this document.

In section 4, some samples are presented and discussed, to illustrate the data that composes the datasets.

In section 5, a summary is presented with information about: the main parameters of the programs; the execution order of the scripts to generate the datasets; the data stored in the datasets.

2 Topology Optimization

2.1 Problem Description

The considered topology optimization problem is the standard structural compliance minimization of a bidimensional cantilever beam. The design domain is positioned in the coordinate system (s_x, s_y) as presented in Figure 1. The height of the cantilever is denoted by L_y and its length is denoted by L_x.

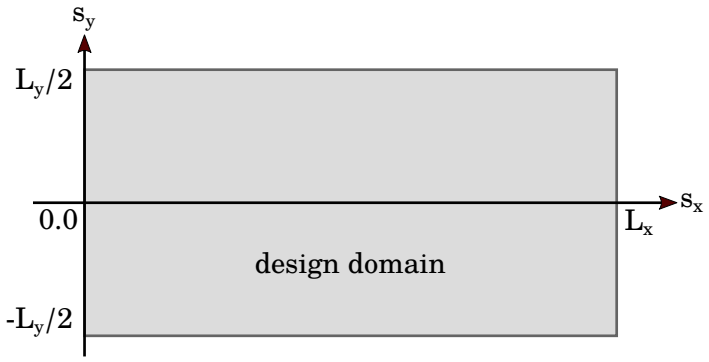

Figure 1: Coordinate system

Figure 2 presents a family of boundary conditions, defined according to the following rules. There is a single clamped region on the left surface of the design domain, which is defined by its center position, p_{bc}, and by its half-length, r_{bc}. And there is a single loaded region on the right surface of the design domain, which is defined by its center position, p_{ld}, and by its half-length, r_{ld}.

Figure 2: Boundary conditions

As shown in Equation 1, the load distribution is uniform and the total load is unitary. It should be noted that f is given in N/m because a bidimensional structure is considered, this means that $1\,N$ per unit width is applied to the cantilever beam.

$$
\begin{aligned}
q(L_x, s_y) &= -\frac{1}{2\,r_{ld}} N/m^2\,,\; s_y \in [p_{ld} - r_{ld}, p_{ld} + r_{ld}] \\
f &= \int_{p_{ld}-r_{ld}}^{p_{ld}+r_{ld}} q(L_x, s_y)\, ds_y = -1\,N/m
\end{aligned}
\tag{1}
$$

Each unique valid configuration of the quadruplet $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$ defines a unique topology optimization problem. In the continuous domain, this would result in a family of infinite distinct problems. Here, the domain is discretized, and additional considerations are included so that a finite number of problems is defined.

The discretized domain is defined by the number of elements in the horizontal direction, N_x, and by the number of elements in the vertical direction, N_y. In order to generate a structured grid mesh with square elements, the parameter L_x is set as

$$
L_x = N_x \frac{L_y}{N_y} = N_x\, e_s, \tag{2}
$$

where e_s is the side length of the square elements.

The elements are numbered from the leftmost column to the rightmost column and, in each column, they are numbered from bottom to top, as illustrated in Figure 3 for $N_x = 8$ and $N_y = 4$.

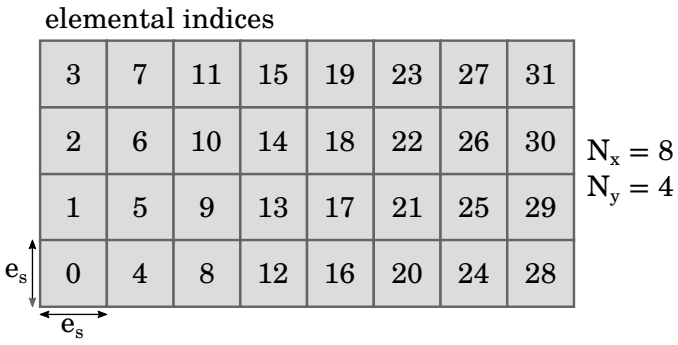

Figure 3: Elemental indices of the discretized domain

Figure 4 shows the nodal indices, which follow this same numbering rule, for $N_x = 8$ and $N_y = 4$.

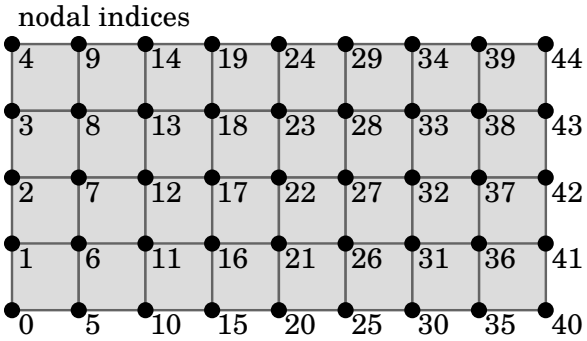

Figure 4: Nodal indices of the discretized domain

Bilinear square elements in plane stress state are considered in the Finite Element Analysis (FEA). A homogeneous isotropic material is considered with Young's modulus of $E_y = 1.0\,Pa$ and Poisson's coefficient of $\nu = 0.3$. Small displacements and strains are considered, so linear assumptions are adopted.

It should be noted that, although this Young's modulus is unrealistic, in the compliance minimization problem with linear analysis, changing the values of the load magnitude and of the Young's modulus results in multiplying the objective function by a positive constant factor. This means that the solution of the optimization problem is the same, no matter what values are considered for those parameters. Therefore, unitary values are used to improve numerical accuracy.

For each element, the local nodal indices are given as presented in Figure 5, they are numbered anticlockwise, starting at the bottom-left node.

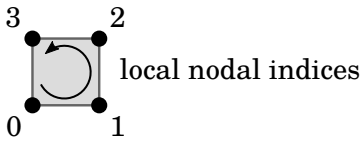

Figure 5: Local nodal indices of the bilinear square element

The local indices are used to consistently define the elemental stiffness matrix, $\mathbf{K_i}$. Under the presented setting, it is approximately given by

$$\mathbf{K_i} \approx \begin{bmatrix} 0.4945 & 0.1786 & -0.3022 & -0.0137 & -0.2473 & -0.1786 & 0.0549 & 0.0137 \\ 0.1786 & 0.4945 & 0.0137 & 0.0549 & -0.1786 & -0.2473 & -0.0137 & -0.3022 \\ -0.3022 & 0.0137 & 0.4945 & -0.1786 & 0.0549 & -0.0137 & -0.2473 & 0.1786 \\ -0.0137 & 0.0549 & -0.1786 & 0.4945 & 0.0137 & -0.3022 & 0.1786 & -0.2473 \\ -0.2473 & -0.1786 & 0.0549 & 0.0137 & 0.4945 & 0.1786 & -0.3022 & -0.0137 \\ -0.1786 & -0.2473 & -0.0137 & -0.3022 & 0.1786 & 0.4945 & 0.0137 & 0.0549 \\ 0.0549 & -0.0137 & -0.2473 & 0.1786 & -0.3022 & 0.0137 & 0.4945 & -0.1786 \\ 0.0137 & -0.3022 & 0.1786 & -0.2473 & -0.0137 & 0.0549 & -0.1786 & 0.4945 \end{bmatrix} . \tag{3}$$

In the discretized domain, the family of boundary conditions from Figure 2 can be reduced to a finite number of unique configurations. In order to do so, it is established that the boundary conditions can only be applied to whole elements, never to only part of their edges. In other words, it is imposed that both the lower and upper limits of the clamped region coincide with nodes of the mesh. And that both the lower and upper limits of the loaded region coincide with nodes of the mesh. At least two nodes must be clamped, so the structural analysis has a unique solution. Point loads are allowed, so the loaded region can contain a single node. In order to allow a point load to be applied in the center of the cantilever beam, N_y is set to be an even number.

The topology vector is defined as $\mathbf{x} \in \{0,1\}^N$, where $N = N_x \times N_y$ is the total number of elements in the design domain. The element of index i is solid when $x_i = 1$ and is void when $x_i = 0$. Thus, the global stiffness matrix, $\mathbf{K}$, can be written as

$$\mathbf{K}(\mathbf{x}) = \sum_{i=0}^{N-1} \left[x_i \, \mathbf{R_i}^T \, \mathbf{K_i} \, \mathbf{R_i} \right] , \tag{4}$$

where $\mathbf{R_i}$ is a known constant matrix that transfer the local matrix $\mathbf{K_i}$ to the global system, reordering rows and columns according to the global indices, removing constrained degrees of freedom, and filling with zeroes all rows and columns corresponding to degrees of freedom that are not related to the i-th element.

A soft-kill approach is used so that $\mathbf{K}$ does not become singular throughout the optimization procedure. In order to do so, a small stiffness, $\varepsilon_k \, \mathbf{K_i}$, is attributed to void elements, where ε_k is a small positive value (e.g., 1×10^{-6}). Therefore, $\mathbf{K}$ is redefined as

$$\mathbf{K}(\mathbf{x}) = \varepsilon_k \sum_{i=0}^{N-1} \left[\mathbf{R_i}^T \, \mathbf{K_i} \, \mathbf{R_i} \right] + [1 - \varepsilon_k] \sum_{i=0}^{N-1} \left[x_i \, \mathbf{R_i}^T \, \mathbf{K_i} \, \mathbf{R_i} \right] . \tag{5}$$

The volume of material, V, in the structure is given by

$$V(\mathbf{x}) = V_e \sum_{i=0}^{N-1} x_i \, , \tag{6}$$

where $V_e = w \times e_s^2$ is the elemental volume (w is the width of the structure).

The displacements vector, $\mathbf{u}$, is given by

$$\mathbf{u}(\mathbf{x}) = \mathbf{K}^{-1}(\mathbf{x}) \, \mathbf{f} \, , \tag{7}$$

where $\mathbf{f}$ is the known constant load vector.

And the structural compliance, C, is given by

$$C(\mathbf{x}) = \mathbf{u}(\mathbf{x})^T \, \mathbf{K}(\mathbf{x}) \, \mathbf{u}(\mathbf{x}) = \mathbf{f}^T \, \mathbf{K}^{-1}(\mathbf{x}) \, \mathbf{f} \, . \tag{8}$$

Finally, for specified inputs (N_x, N_y, L_y) and for a given valid configuration of the quadruplet $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$, from which the constraint matrices $\mathbf{R_i}$ and the load vector $\mathbf{f}$ are obtained, the topology optimization problem can be stated as follows.

$$\begin{aligned} &\min_{\mathbf{x}} C(\mathbf{x}) \\ &s.t. \\ &V(\mathbf{x}) = V^* \end{aligned} \tag{9}$$

Since the structural compliance always decreases when each x_i increases, the volume constraint is required to avoid the trivial solution of a fully solid topology, V^* is a user-defined volume for the optimized topology.

Here, this problem of binary variables is solved through two different iterative approaches. Firstly, through a heuristic Sequential Integer Linear Programming (SILP) algorithm. Then, through a gradient-based method coupled with a continuous relaxation of the problem, using the Solid Isotropic Material with Penalization (SIMP) scheme.

2.2 Optimization Methods

2.2.1 Sequential Integer Linear Programming (SILP)

In this approach, the discrete objective function has to be linearized around the current topology $\bar{\mathbf{x}}$ [1, 2, 3]. In order to extract the linear component from a function of binary variables, it is firstly written in the form

$$C(\mathbf{x}) = \alpha^{\langle 0 \rangle} + \sum_{j=1}^{N} \boldsymbol{\alpha}^{\langle \boldsymbol{j} \rangle} (\cdot)^j (\mathbf{x} - \bar{\mathbf{x}})^j \,, \tag{10}$$

where the jth-order tensor $(\mathbf{x} - \bar{\mathbf{x}})^j$ corresponds to the outer product between j vectors $(\mathbf{x} - \bar{\mathbf{x}})$ and the $(\cdot)^j$-product represents the operation given by

$$\boldsymbol{\alpha}^{\langle \boldsymbol{j} \rangle} (\cdot)^j (\mathbf{x} - \bar{\mathbf{x}})^j = \sum_{i_1=0}^{N-1} \sum_{i_2=0}^{N-1} \dots \sum_{i_j=0}^{N-1} \alpha^{\langle j \rangle}_{i_1 \, i_2 \, \dots \, i_j} \left(x_{i_1} - \bar{x}_{i_1}\right) \left(x_{i_2} - \bar{x}_{i_2}\right) \dots \left(x_{i_j} - \bar{x}_{i_j}\right) . \tag{11}$$

The scalar $\alpha^{\langle 0 \rangle}$ corresponds to $C(\bar{\mathbf{x}})$, $\boldsymbol{\alpha}^{\langle \mathbf{1} \rangle}$ is a vector of N entries corresponding to the variations of C when the state of a single element of $\bar{\mathbf{x}}$ is switched (from solid to void, or from void to solid), and, for $j > 1$, $\boldsymbol{\alpha}^{\langle \boldsymbol{j} \rangle}$ are strictly upper triangular tensors of order j, so $\alpha_{i_1 \, i_2 \, \dots \, i_j}$ only assumes non-zero values when $i_1 < i_2 < \dots < i_j$. The j-th order tensor, $\boldsymbol{\alpha}^{\langle \boldsymbol{j} \rangle}$, is related to the combined effect of simultaneously switching the state of j elements of $\bar{\mathbf{x}}$.

Thus, the linear truncation of $C(\mathbf{x})$, denoted by $\hat{C}(\mathbf{x})$, is given by

$$\hat{C}(\mathbf{x}) = \alpha^{\langle 0 \rangle} + \boldsymbol{\alpha}^{\langle \mathbf{1} \rangle} \cdot [\mathbf{x} - \bar{\mathbf{x}}] \,. \tag{12}$$

The vector $\boldsymbol{\alpha}^{\langle \mathbf{1} \rangle}$ can be understood as the sensitivity of the linearized objective function with respect to each design variable, its entries are given by

$$\alpha_i^{\langle 1 \rangle} = C(\bar{\mathbf{x}}, x_i = 1) - C(\bar{\mathbf{x}}, x_i = 0) \,, \tag{13}$$

where the arguments $(\bar{\mathbf{x}}, x_i = 1)$ and $(\bar{\mathbf{x}}, x_i = 0)$ denote vectors that are equal to $\bar{\mathbf{x}}$ except at their ith term, which assumes the explicitly defined value.

Let $\mathbf{x}^{(k)}$ be the topology vector of the k-th iteration and $\hat{C}(\mathbf{x})$ the objective function linearized around $\mathbf{x}^{(k)}$, then, the following subproblem can be stated:

$$\begin{gathered} \min_{\mathbf{x}} \hat{C}(\mathbf{x}) \\ s.t. \\ V(\mathbf{x}) - V(\mathbf{x}^{(k)}) = VV^{(k)} \\ \|\mathbf{x} - \mathbf{x}^{(k)}\|_1 \leq TV^{(k)} \,, \end{gathered} \tag{14}$$

where $VV^{(k)}$ is the volume variation imposed for the k-th iteration and $TV^{(k)}$ is the maximal topological variation allowed in the k-th iteration. It is important to limit the maximal topological variation per iteration in order to improve the accuracy of the linear approximation $\hat{C}(\mathbf{x})$, and also to improve the stability of the iterative procedure. Once the topological variation is limited, it is necessary to define a proper gradual progression from $V(\mathbf{x}^{(0)})$ to V^*, through imposed volume variations $VV^{(k)}$. Otherwise, the feasible region of the subproblem could become empty depending on $\mathbf{x}^{(k)}$, V^* and $TV^{(k)}$.

It should be noted that any additively separable function of binary variables is an affine function, therefore, both constraints of the considered subproblems are linear.

The heuristic SILP approach consists in exploring the domain of feasible topologies by successively solving these linearized subproblems, starting from a given initial topology $\mathbf{x}^{(0)}$. The stopping criterion is given in terms of a patience parameter, $P \in \mathbb{N}^*$: throughout the optimization procedure, the best topology thus far is stored, when P consecutive iterations are performed without obtaining a better topology, the procedure stops.

For this pair of constraints, the Bidirectional Evolutionary Structural Optimization (BESO) algorithm [4, 5, 6] can be used to quickly solve each Integer Linear Programming subproblem. For the problem linearized around $\mathbf{x}^{(k)}$, the next topology, $\mathbf{x}^{(k+1)}$, can by obtained as follows: the elements are ordered by their sensitivity values, then the elements with higher values are turned into voids while the ones with lower values are turned into solids, ensuring that the volume variation and topological variation constraints are both satisfied.

The only costly task of the BESO algorithm is to perform the sensitivity analysis, that is, to compute $\boldsymbol{\alpha}^{\langle \mathbf{1} \rangle}$. Its exact expression, obtained through the Woodbury Sensitivity (WS) approach [3], is given by

$$\alpha_i^{\langle 1 \rangle} = \begin{cases} -\,\mathbf{u}^T\,\mathbf{H_i}\left[\mathbf{I} + \mathbf{H_i}^T\,\mathbf{K}^{-1}\,\mathbf{H_i}\right]^{-1}\mathbf{H_i}^T\,\mathbf{u} \;\;, \text{ if } x_i = 0 \\ -\,\mathbf{u}^T\,\mathbf{H_i}\left[\mathbf{I} - \mathbf{H_i}^T\,\mathbf{K}^{-1}\,\mathbf{H_i}\right]^{-1}\mathbf{H_i}^T\,\mathbf{u} \;\;, \text{ if } x_i = 1\,, \end{cases} \tag{15}$$

where $\mathbf{H_i}$ is a rectangular matrix such that

$$\mathbf{H_i}\,\mathbf{H_i}^T = [1 - \varepsilon_k]\,\mathbf{R_i}^T\,\mathbf{K_i}\,\mathbf{R_i}\,. \tag{16}$$

To compute $\mathbf{H_i}$, the constrained elemental stiffness matrix is factorized in a matrix, $\mathbf{V}$, of orthonormal eigenvectors and a diagonal matrix, $\mathbf{D}$, of eigenvalues

$$[1 - \varepsilon_k]\,\mathbf{R_i}^T\,\mathbf{K_i}\,\mathbf{R_i} = \mathbf{V}\,\mathbf{D}\,\mathbf{V}^T\,. \tag{17}$$

Since $\mathbf{D}$ has only non-negative values, $\mathbf{H_i}$ can be defined as

$$\mathbf{H_i} = \mathbf{V}\,\sqrt{\mathbf{D}}\,. \tag{18}$$

It should be noted that, for bilinear square elements, each $\mathbf{H_i}$ has at most 5 columns, so the inverse of $[\mathbf{I} \pm \mathbf{H_i}^T\mathbf{K}^{-1}\mathbf{H_i}]$ can be easily computed.

The Cholesky factorization is used to solve the linear system from structural analysis and obtain $\mathbf{u}$. Although this factorization can be reused to compute $\mathbf{K}^{-1}\,\mathbf{H_i}$, this has to be performed for each of the N elements of the mesh, thus, the computation of this exact sensitivity vector is an expensive task.

Alternatively, approximated expressions can be used to reduce computational costs. The standard approximation is given by

$$\alpha_i^{\langle 1 \rangle} \approx -[1 - \varepsilon_k]\,\mathbf{u_i}^T\,\mathbf{K_i}\,\mathbf{u_i}\,, \tag{19}$$

where $\mathbf{u_i} = \mathbf{R_i}\,\mathbf{u}$ is a local displacements vector for the i-th element.

However, by using the Conjugate Gradient Sensitivity (CGS) approach [3], more accurate approximations can be obtained.

For a given element of index i, let the vector $\mathbf{b}$ be defined as

$$\mathbf{b} = \begin{cases} \;\;\mathbf{R_i}^T\,\mathbf{K_i}\,\mathbf{u_i} \;\;, \text{ if } x_i = 0 \\ -\,\mathbf{R_i}^T\,\mathbf{K_i}\,\mathbf{u_i} \;\;, \text{ if } x_i = 1 \end{cases} \tag{20}$$

the matrix $\mathbf{M_K}$ be defined as

$$\mathbf{M_K} = \begin{cases} \mathbf{K} + [1 - \varepsilon_k]\,\mathbf{R_i}^T\,\mathbf{K_i}\,\mathbf{R_i} \;\;, \text{ if } x_i = 0 \\ \mathbf{K} - [1 - \varepsilon_k]\,\mathbf{R_i}^T\,\mathbf{K_i}\,\mathbf{R_i} \;\;, \text{ if } x_i = 1 \end{cases} \tag{21}$$

and the diagonal matrix $\mathbf{M_D}$ can be defined as

$$\mathbf{M_D} = \mathbf{diag}(\mathbf{M_K})\,. \tag{22}$$

Let the coefficients $\langle b\rangle_0$ and $\langle b\rangle_1$ be defined as

$$\begin{aligned} \langle b\rangle_0 &= \quad \mathbf{b}^T\,\mathbf{v_M} \\ \langle b\rangle_1 &= \mathbf{v_K}^T\,\mathbf{v_M}\,, \end{aligned} \tag{23}$$

where the vectors $\mathbf{v_M}$ and $\mathbf{v_K}$ are given by

$$\begin{aligned} \mathbf{v_M} &= \mathbf{M_D}^{-1}\,\mathbf{b} \\ \mathbf{v_K} &= \mathbf{M_K}\,\mathbf{v_M}\,. \end{aligned} \tag{24}$$

Then, the CGS approximation with 1 step can be written as

$$\alpha_i^{\langle 1\rangle} \approx \begin{cases} -\left[[1-\varepsilon_k]\,\mathbf{u_i}^T\,\mathbf{K_i}\,\mathbf{u_i} - \dfrac{\langle b\rangle_0{}^2}{\langle b\rangle_1}\right] & ,\ \text{if } x_i = 0 \\ -\left[[1-\varepsilon_k]\,\mathbf{u_i}^T\,\mathbf{K_i}\,\mathbf{u_i} + \dfrac{\langle b\rangle_0{}^2}{\langle b\rangle_1}\right] & ,\ \text{if } x_i = 1\,. \end{cases} \tag{25}$$

Using 2 steps, a more accurate CGS approximation can be obtained. Let the coefficients $\langle b\rangle_2$ and $\langle b\rangle_3$ be defined as

$$\begin{aligned} \langle b\rangle_2 &= \mathbf{v_K}^T\,\mathbf{v_L} \\ \langle b\rangle_3 &= \mathbf{v_R}^T\,\mathbf{v_L}\,, \end{aligned} \tag{26}$$

where the vectors $\mathbf{v_L}$ and $\mathbf{v_R}$ are given by

$$\begin{aligned} \mathbf{v_L} &= \mathbf{M_D}^{-1}\,\mathbf{v_K} \\ \mathbf{v_R} &= \mathbf{M_K}\,\mathbf{v_L}\,. \end{aligned} \tag{27}$$

Then, the CGS approximation with 2 steps can be written as

$$\alpha_i^{\langle 1\rangle} \approx \begin{cases} -\left[[1-\varepsilon_k]\,\mathbf{u_i}^T\,\mathbf{K_i}\,\mathbf{u_i} - \dfrac{\langle b\rangle_0{}^2\,\langle b\rangle_3 - 2\,\langle b\rangle_0\,\langle b\rangle_1\,\langle b\rangle_2 + \langle b\rangle_1{}^3}{\langle b\rangle_1\,\langle b\rangle_3 - \langle b\rangle_2{}^2}\right] & ,\ \text{if } x_i = 0 \\ -\left[[1-\varepsilon_k]\,\mathbf{u_i}^T\,\mathbf{K_i}\,\mathbf{u_i} + \dfrac{\langle b\rangle_0{}^2\,\langle b\rangle_3 - 2\,\langle b\rangle_0\,\langle b\rangle_1\,\langle b\rangle_2 + \langle b\rangle_1{}^3}{\langle b\rangle_1\,\langle b\rangle_3 - \langle b\rangle_2{}^2}\right] & ,\ \text{if } x_i = 1\,. \end{cases} \tag{28}$$

The sensitivity vector expressed in Equation 15 is referred to as WS expression; the one expressed in Equation 19 is referred to as CGS-0 approximation; the one expressed in Equation 25 is referred to CGS-1 approximation; and the one expressed in Equation 28 is referred to as CGS-2 approximation.

The WS expression corresponds to the exact sensitivity vector, the CGS-0, CGS-1 and CGS-2 are approximations with increasing accuracy. Evidently, the more accurate expressions are also more expensive to compute.

Instead of using the raw sensitivity vector to linearize the objective function and solve the optimization subproblem through a SILP algorithm, two procedures are included to improve the stability of the iterative procedure and the quality of the solutions. The sensitivity map is smoothed through a filtering procedure, this is done in order to deal with the checkerboard problem and mesh dependency [7, 8]. Moreover, a momentum method is included, so the previous values of the sensitivity vector throughout the iterations (with proper weighting factors) are added to the value of the current sensitivity vector [9]. This inhibits oscillations between consecutive iterations and favors a more extensive exploration of the domain of feasible topologies.

Here, a simple conical filter is considered, it is defined by its radius, $r_{\max}$, which is a geometric parameter that indirectly controls the minimal thickness of the structural components of the beam, independently of the mesh.

Let $r(i,j)$ be defined as the distance between centers of the i-th and the j-th elements. Then, the filter weights W_{ij} are given by

$$W_{ij} = \frac{\max\left(r_{\max} - r(i,j)\,,\,0\right)}{\sum_{k=0}^{N-1} \max\left(r_{\max} - r(i,k)\,,\,0\right)} \tag{29}$$

and the filtered sensitivity vector $\boldsymbol{\alpha}^{[f]}$ is given by

$$\alpha_i^{[f]} = \sum_{j=0}^{N-1} W_{ij}\, \alpha_j^{\langle 1\rangle}\,. \tag{30}$$

Only elements within the range of the filter radius are considered when filtering the sensitivity value of each element. Furthest elements are disregarded since their contributions would be $\max\left(r_{\max} - r(i,j)\,,\,0\right) = 0$. By definition, the conical filter weights decrease linearly with the distance from the central element. The filtering procedure is a linear transformation and the weights can be stored in a sparse matrix $\mathbf{W}$. Every row of $\mathbf{W}$ adds up to 1, so there is no scaling factor.

Here, equal weights are used for the momentum method. The vectors are normalized so that eventual sensitivity peaks do not overly pollute the optimization process [10]. Thus, for the k-th iteration the normalized filtered vector $\tilde{\boldsymbol{\alpha}}^{[f]}$ is given by

$$\tilde{\boldsymbol{\alpha}}^{[f]} = \frac{\boldsymbol{\alpha}^{[f]}}{\|\boldsymbol{\alpha}^{[f]}\|_\infty} = \frac{\boldsymbol{\alpha}^{[f]}}{\max\limits_i |\alpha_i^{[f]}|}\,. \tag{31}$$

Then, the momentum is applied as follows:

$$\boldsymbol{\alpha}^{[m](k)} = \tilde{\boldsymbol{\alpha}}^{[f](k)} + \tilde{\boldsymbol{\alpha}}^{[m](k-1)}\,. \tag{32}$$

Finally, the sensitivity vector used in the SILP algorithm is given by

$$\tilde{\boldsymbol{\alpha}}^{[m]} = \frac{\boldsymbol{\alpha}^{[m]}}{\|\boldsymbol{\alpha}^{[m]}\|_\infty} = \frac{\boldsymbol{\alpha}^{[m]}}{\max\limits_i |\alpha_i^{[m]}|}\,. \tag{33}$$

To perform this procedure consistently in the first iteration, it is defined that $\tilde{\boldsymbol{\alpha}}^{[m](-1)} = \mathbf{0}$.

One last, very important remark must be made for this optimization method. Any solid element which is the sole element connecting an imposed load to a constrained part of the structure will have an arbitrarily high sensitivity absolute value, according to the soft-kill parameter ε_k. To avoid working with exploding values, which may hamper the whole optimization through the filtering and momentum procedures, critical elements are taken out of the design domain, that is, their state is permanently set to be solid and they are considered to be in a non-design domain.

Here, this is done for all directly loaded elements: all elements of the right extremity of the design domain which intersect the loaded region. Their raw sensitivity values are computed normally, but they are disregarded in the filtering procedure (their weights are set to 0 when filtering nearby elements), their values in $\boldsymbol{\alpha}^{[m](k)}$ are set to 0 (after computing Equation 32), and their values in $\tilde{\boldsymbol{\alpha}}^{[m]}$ are set to $-\infty$ (after computing Equation 33).

2.2.2 Solid Isotropic Material with Penalization (SIMP)

In this approach, a continuous relaxation of the problem is considered. The design variables are redefined in a continuous domain, $\mathbf{x} \in [0,1]^N$, and an interpolation function, $\gamma(x_i)$, is used to describe the elemental stiffness for intermediary density values (between 0 and 1). It can be any function with the following properties:

$\gamma(0) = 0$, $\gamma(1) = 1$, γ is differentiable and monotonic in $[0, 1]$. Thus, the global stiffness matrix can be redefined as

$$\mathbf{K}(\mathbf{x}) = \varepsilon_k \sum_{i=0}^{N-1} \left[\mathbf{R_i}^T \mathbf{K_i} \mathbf{R_i} \right] + [1 - \varepsilon_k] \sum_{i=0}^{N-1} \left[\gamma(x_i) \mathbf{R_i}^T \mathbf{K_i} \mathbf{R_i} \right] . \tag{34}$$

Now, all $\mathbf{K}(\mathbf{x})$, $\mathbf{u}(\mathbf{x})$, $C(\mathbf{x})$ and $V(\mathbf{x})$ have become differentiable functions and the relaxed optimization problem can be solved by any gradient-based method.

In order to inhibit solutions with intermediary density values, the SIMP scheme [11, 12] is used, in which γ is defined as

$$\gamma(x_i) = {x_i}^p , \tag{35}$$

where p is the penalization exponent. Using this interpolation function, the derivative of $C(\mathbf{x})$ can be obtained as

$$\frac{\partial C(\mathbf{x})}{\partial x_i} = -p \, {x_i}^{p-1} \, [1 - \varepsilon_k] \, \mathbf{u_i}^T \mathbf{K_i} \, \mathbf{u_i} . \tag{36}$$

Considering the compliance minimization problem with a volume constraint, it can be shown that, for a sufficiently high value of p, the solution of the relaxed problem corresponds to a solution of the original discrete problem, that is, all variables in the solution assume extreme values (0 or 1) [13, 14]. Nonetheless, numerical issues may occur if p is too high. Thus, in the implementation, p is progressively increased throughout the optimization process, starting from $p = 1$, until a sufficient value is reached.

The Method of Moving Asymptotes (MMA) is used to solve the relaxed optimization problem [15]. This method uses the current point $\bar{\mathbf{x}}$, the current values of the objective and constraint functions and the current values of their first derivatives to generate a subproblem in which all functions are convex and separable. For points close to $\bar{\mathbf{x}}$, this subproblem is a good approximation of the original problem. It is then solved by a dual method.

The approximation of the structural compliance is given by

$$C(\mathbf{x}) \approx c - \sum_{i=0}^{N-1} \frac{[\bar{x}_i - L_i]^2 \, \frac{\partial C}{\partial x_i}(\bar{x}_i)}{x_i - L_i} , \tag{37}$$

where c is a constant such that the value of the approximation at $\bar{\mathbf{x}}$ corresponds to the exact $C(\bar{\mathbf{x}})$. For each design variable, a parameter L_i is defined. It corresponds to the position of a vertical asymptote of the convex separable approximation. The variation of each variable can be indirectly limited by controlling the values of these parameters throughout the iterative procedure.

In the globally-convergent version on the method [16], after obtaining the candidate for $\mathbf{x}^{(k+1)}$ by solving the subproblem generated at $\mathbf{x}^{(k)}$, the method verifies some conditions to measure if the obtained approximation was really adequate to perform the step from $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$, if the conditions are not met, a more conservative approximation is generated at $\mathbf{x}^{(k)}$ and the process is repeated.

It should be noted that globally convergent does not mean that a global minimum is obtained. It means that the method converge to a local minimum for any feasible initial point.

In each iteration, the gray level, $g(\mathbf{x})$, of the current topology is computed as

$$g(\mathbf{x}) = \frac{4}{N} \sum_{i=0}^{N-1} x_i \, [1 - x_i] . \tag{38}$$

It is defined so that $g(\mathbf{0}) = g(\mathbf{1}) = \min_{\mathbf{x}} g(\mathbf{x}) = 0$ and $g(\mathbf{0.5}) = \max_{\mathbf{x}} g(\mathbf{x}) = 1$. The gray level is an inverse measure of the topology discreteness. Low values of g imply that the penalization exponent p should be increased in order to obtain a solution which is more coherent with the original discrete problem.

The stopping criterion used for this algorithm is based on the variation of the design variables between iterations, when no design variable varies more than a specified value (e.g., 10^{-2}) between iterations, the

procedure stops. Moreover, alternative convergence criteria can be defined, based on the variation of the objective function and of the gray level of the topology. A maximal number of inner evaluations can be defined: if the method retries too many times to obtain a sufficiently conservative approximation without success, it perform the step anyway. Also, a maximal number of outer evaluations can be defined, which is used to interrupt the optimization procedure if too many iterations are performed without reaching the stopping criteria.

In this method, the sensitivity of the objective function with respect to each design variable is given by the gradient of the objective function. Instead of using the raw sensitivity vector to generate the subproblem, the sensitivity map is firstly smoothed through a filtering procedure. Once again, this is done in order to deal with the checkerboard problem and mesh dependency. Here, the same conical filter is considered, defined by its radius, $r_{\max}$.

3 Dataset Generation

3.1 Fixed Properties

The dimensions of the design domain are set to $L_x = 2.0\,m$ and $L_y = 1.0\,m$. The numbers of elements in the each direction are set to $N_x = 64$ and $N_y = 32$, so the total number of elements in the mesh is $N = 2048$. The target volume V^*, is specified as 50% of the design domain volume, so it corresponds to 1024 elements. The sensitivity filter radius, $r_{\max}$, is set to $0.125\,m$ (1/8 of L_y) and the soft-kill parameter, ε_k, is set to 1×10^{-6}.

For the SILP-BESO approach, $VV^{(k)}$ and $TV^{(k)}$ are the same for all iterations, their respective values are 1.5625% (1/64) and 3.1250% (1/32), related to the whole volume of the design domain. The patience parameter used as stopping criterion is set to 20.

For the SIMP-MMA approach, 16 iterations are performed with no penalization ($p = 1$), then 16 iterations are performed with $p = 3$, lastly the penalization exponent is set to $p = 6$ and the method iterates until the stopping criteria are achieved.

3.2 User Guide

All programs were developed in Python and Cython. Anaconda was used to manage packages through conda.

This short guide describes how to setup the conda environment, and how to properly execute the provided scripts in order to generate the datasets. Everything was developed in Linux (Ubuntu 20.04 LTS). There is a chance that other operating systems do not support some of the external packages used in the programs, if it is the case, users will have to adapt the codes according to their need.

3.2.1 Setup Conda Environment

To install Anaconda, go to https://www.anaconda.com/distribution and download the latest stable version. Alternatively, the Anaconda bash script may be downloaded using curl:

```
cd /tmp
curl -O https://repo.anaconda.com/archive/Anaconda3-2022.05-Linux-x86_64.sh
```

Then, run the bash script:

```
bash Anaconda3-2022.05-Linux-x86_64.sh
```

Follow the instructions to progress. Accept the license terms (if you agree with them) and specify your preferable location to install Anaconda. After the installation is finished, you will be asked if the installer should initialize Anaconda, write "yes". Lastly, activates the installation:

```
source ~/.bashrc
```

Next, choose a location in your machine (<location_in_your_machine>) and download everything from the github repository (https://github.com/Joquempo/Cantilever-Dataset). Finally, run the provided "cantilever.sh" bash script in order to setup the conda environment:

```
cd <location_in_your_machine>/source
bash ./cantilever.sh
```

This script will: update conda; create a new Python-3.8 environment named "cantilever"; add the channel conda-forge; set channel_priority as strict; install numpy [17], scipy [18], matplotlib [19], cython [20], scikit-sparse [21] and nlopt [22]; and build all Cython codes in "./cython" directory.

3.2.2 Generate Datasets

Before executing the scripts, be warned that the BESO dataset occupies around 617 GB of disk, and the SIMP dataset occupies around 363 GB of disk. So around **980 GB** of free space is needed to generate both.

To generate the datasets, activate the cantilever environment and go to the directory with the provided Python codes:

```
conda activate cantilever
cd <location_in_your_machine>/source/python
```

Then, run the script "input_str.py", which will create the "./input" directory and store 148 240 files in it, containing all possible non-redundant input parameters. This will occupy around 600 MB of disk.

```
python ./input_str.py
```

Although some procedures are embarrassingly parallelizable, it has been decided to keep the programs serialized and call multiple parallel executions, using multiple processors of the machine. Run the following command to list the processors in your machine. It is recommended to perform some tests to obtain the optimal number of parallel executions. In my machine, I used 4 processors, of indices 0, 1, 2 and 3 which are all different physical cores.

```
cat /proc/cpuinfo  | egrep "processor|core id"
```

Then, go to "<location_in_your_machine>/source/python/BESO" and open the "structural_beso.py" script in your preferable text editor. Redefine the values of the parameters "fid_ini" and "fid_lim" to select how many cases will be optimized. Create one copy of this script for each processor, with complementary values for the parameters "fid_ini" and "fid_lim". For example: "fid_ini=0" and "fid_lim=1600" in "structural_beso_0.py"; "fid_ini=1600" and "fid_lim=3200" in "structural_beso_1.py"; "fid_ini=3200" and "fid_lim=4800" in "structural_beso_2.py"; and "fid_ini=4800" and "fid_lim=6400" in "structural_beso_3.py". This will run the first 6400 cases using 4 processors.

The script prints information about which case is being optimized, so it is recommended that each script be executed in a different terminal window. Open the first terminal and execute the first script, setting it to the desired processor:

```
taskset -c 0 python ./BESO/structural_beso_0.py
```

Open the next terminal and repeat for the next processor:

```
taskset -c 1 python ./BESO/structural_beso_1.py
```

```
taskset -c 2 python ./BESO/structural_beso_2.py
```

```
taskset -c 3 python ./BESO/structural_beso_3.py
```

The "./BESO/output" directory will be created and each script will create a subfolder in it, to store the generated data. When the executions are concluded, verify the generated log files (in each subfolder), they present the input values corresponding to each input file, and the execution time of each performed optimization.

Keep redefining the values of the parameters "fid_ini" and "fid_lim" (always with complementary values, so no redundant data is generated) and executing the scripts, until all 148 240 cases are optimized. Be warned that the WS expression (Equation 15), which is very costly, is computed in each iteration, so each BESO optimization may take a few minutes (even for this coarse mesh). **Weeks may be necessary to generate the whole BESO dataset, depending on your computer**.

The exact same procedure is repeated to generate the SIMP dataset. Go to "<location_in_your_machine>/source/python/SIMP" and open the "structural_simp.py" script in your

preferable text editor. Redefine the values of the parameters "fid_ini" and "fid_lim" to select how many cases will be optimized. Create one copy of this script for each processor, with complementary values for the parameters "fid_ini" and "fid_lim". For example: "fid_ini=0" and "fid_lim=6400" in "structural_simp_0.py"; "fid_ini=6400" and "fid_lim=12800" in "structural_simp_1.py"; "fid_ini=12800" and "fid_lim=19200" in "structural_simp_2.py"; and "fid_ini=19200" and "fid_lim=25600" in "structural_simp_3.py". This will run the first 25 600 cases using 4 processors.

As before, the script prints information about which case is being optimized, so it is recommended that each script be executed in a different terminal window. Open the first terminal and execute the first script, setting it to the desired processor:

```
taskset -c 0 python ./SIMP/structural_simp_0.py
```

Open the next terminal and repeat for the next processor:

```
taskset -c 1 python ./SIMP/structural_simp_1.py
```

```
taskset -c 2 python ./SIMP/structural_simp_2.py
```

```
taskset -c 3 python ./SIMP/structural_simp_3.py
```

The "./SIMP/output" directory will be created and each script will create a subfolder in it, to store the generated data. When the executions are concluded, verify the generated log files (in each subfolder).

Keep redefining the values of the parameters "fid_ini" and "fid_lim" (always with complementary values, so no redundant data is generated) and executing the scripts, until all 148 240 cases are optimized. Here, there is no operation as costly as computing the WS expression, so each SIMP optimization only takes a few seconds. Considering the time to generate the BESO dataset, it should be around 100 times faster to generate the whole SIMP dataset.

After performing all $296\,480 = 2 \times 148\,240$ optimizations, run the script "generate_str.py" to conclude the generation of the datasets:

```
python ./generate_str.py
```

In the <location_in_your_machine>, this script will create the directories: "./dataset/BESO" and "./dataset/SIMP". Finally, it will relocate all generated data to these directories in an organized manner.

3.3 Implementation – Python

3.3.1 ./source/python/input_str.py

This script generates the input data for the structural optimization problems to be solved. For a given number of elements in the vertical direction, N_y, all valid configurations of the quadruplets $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$ are generated and stored in separate files.

Firstly, the directory where the input data will be stored is created; the parameters N_y and L_y are defined; and e_s, denoted by "esize", is computed.

input_str

```
1  import os, pickle
2  import numpy as np
3  # check directories
4  if not os.path.exists('./input'):
5      os.mkdir('./input')
6  Ny = 32        # number of elements in y-axis
7  Ly = 1.0       # cantilever height
8  esize = Ly/Ny  # element size
```

Then, all possible constraint configurations, given by (p_{bc}, r_{bc}), are generated and stored in a pair of arrays: "bc_pos_list" and "bc_rad_list". The number of possible constraints is given by $\frac{N_y\,(N_y+1)}{2}$, which corresponds to 528 possibilities for the considered mesh.

input_str

```
9  bc_pos_list_unique = 0.5*(esize*(np.arange(2*Ny-1)+1) - Ly)
10 bc_num = (Ny*(Ny+1))//2
11 bc_pos_list = np.zeros(bc_num)
```

```
bc_rad_list = np.zeros(bc_num)
c0 = 0
for k in range(Ny):
    c1 = c0 + k//2 + 1
    bc_pos_list[c0:c1] = bc_pos_list_unique[k]
    bc_rad_list[c0:c1] = np.arange(k//2+1) + 1
    if c0 == 0:
        bc_pos_list[-c1:] = bc_pos_list_unique[-1-k]
        bc_rad_list[-c1:] = np.arange(k//2+1) + 1
    else:
        bc_pos_list[-c1:-c0] = bc_pos_list_unique[-1-k]
        bc_rad_list[-c1:-c0] = np.arange(k//2+1) + 1
    c0 = c1
bc_rad_list = esize*bc_rad_list
```

Likewise, all possible load configurations, given by (p_{ld}, r_{ld}), are generated and stored in a pair of arrays: "ld_pos_list" and "ld_rad_list". The number of possible constraints is given by $\frac{(N_y+1)\,(N_y+2)}{2}$, which corresponds to 561 possibilities for the considered mesh. There are more load configurations than constraint configurations because at least 2 nodes must be constrained, while point loads are allowed.

input_str

```
ld_pos_list_unique = 0.5*(esize*(np.arange(2*Ny+1)) - Ly)
ld_num = ((Ny+1)*(Ny+2))//2
ld_pos_list = np.zeros(ld_num)
ld_rad_list = np.zeros(ld_num)
c0 = 0
for k in range(Ny+1):
    c1 = c0 + k//2 + 1
    ld_pos_list[c0:c1] = ld_pos_list_unique[k]
    ld_rad_list[c0:c1] = np.arange(k//2+1) + k%2
    if c0 == 0:
        ld_pos_list[-c1:] = ld_pos_list_unique[-1-k]
        ld_rad_list[-c1:] = np.arange(k//2+1) + k%2
    else:
        ld_pos_list[-c1:-c0] = ld_pos_list_unique[-1-k]
        ld_rad_list[-c1:-c0] = np.arange(k//2+1)  + k%2
    c0 = c1
ld_rad_list = esize*ld_rad_list
```

Lastly, all constraint configuration are combined with all load configurations to generate all possible quadruplets $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$. The total number is given by $\frac{N_y\,(N_y+1)^2\,(N_y+2)}{4}$, which corresponds to $296\,208 = 528 \times 561$ possibilities for the considered mesh. To reduce processing and storage costs, redundant entries are ignored. If the optimization results for $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$ are known, then the results for $(-p_{bc}, r_{bc}, -p_{ld}, r_{ld})$ can be easily obtained: the load is inverted and the new displacements vectors are obtained through the linearity property; then all topology, sensitivity and displacements vectors are mirrored over the horizontal axis; the values of the objective and volume functions are not altered. This reduce the number of non-redundant possibilities to 148 240.

Only entries with non-positive p_{bc} are considered; moreover, when $p_{bc} = 0$, only entries with non-positive p_{ld} are considered. For each possibility, a file containing N_y, p_{bc}, r_{bc}, p_{ld} and r_{ld} (all in 4-bytes formats) is stored in disk. The files are stored in the input folder and they are named as: "inp_000000.pckl", "inp_000001.pckl", ..., "inp_148239.pckl". Each file contains 20 bytes of data, therefore all 148 240 of them correspond to around 2.8 MB of data.

It should be noted that generating 148 240 separate files is a highly non-optimal storage strategy, since much more than 20 bytes will be allocated for each one of them, according to the block size of the file system. With this implementation, the 2.8 MB of data can actually end up occupying around 600 MB of disk. This should be improved in future versions of these codes. Nonetheless, the whole datasets are composed by around 980 GB of data, so 600 MB is not a substantial loss. Moreover, these input files are temporary and they can be deleted after the datasets are generated, so one should not be overly concerned about this issue.

input_str

```
print('generating input files')
total_num = bc_num*ld_num
fid = 0
for kbc in range(bc_num):
    print(': {:06d} / {:06d} :'.format(kbc*ld_num+1,total_num))
    for kld in range(ld_num):
        input_num = kbc*ld_num + kld
        if bc_pos_list[kbc] > 0.1*esize:
            pass
        else:
            if (bc_pos_list[kbc] > -0.1*esize) and (ld_pos_list[kld] > 0.1*esize):
                pass
            else:
                Ny_uint32 = np.uint32(Ny)              # number of elements in y-axis
```

```
                bc_pos = np.float32(bc_pos_list[kbc])  # center of the restricted area
                bc_rad = np.float32(bc_rad_list[kbc])  # half-length (radius) of the restricted area
                ld_pos = np.float32(ld_pos_list[kld])  # center of the loaded area
                ld_rad = np.float32(ld_rad_list[kld])  # half-length (radius) of the loaded area
                finp = open('./input/inp_{:06d}.pckl'.format(fid),'wb')
                pickle.dump([Ny_uint32,bc_pos,bc_rad,ld_pos,ld_rad],finp)
                finp.close()
                fid = fid + 1
print(': {:06d} / {:06d} :'.format(input_num+1,total_num))
print('[ {:d} input files generated ]'.format(fid))
print('done!')
```

3.3.2 ./source/python/BESO/structural_beso.py

This script performs the SILP-BESO optimization procedure for the selected input data. Two text files are written. The input-output log ("io_log.txt") lists what this program generates as output data and presents each considered input data, together with dates and times that inform when each optimization procedure was performed. The time log ("time_log.txt") presents the execution times of each task of the optimization procedures.

The output data is composed by numpy arrays with: the input files indices ("fid.npy"); the input data ("inp.npy"); the optimized topologies ("top_opt.npy"); the optimized objective function values ("obj_opt.npy"); pointers relating each input with the corresponding iterations of the optimization processes ("ptr2opt.npy"); pointers relating each iteration of the optimization processes with the corresponding inputs ("ptr2inp.npy"); all generated topology vectors ("top.npy"); the displacements vectors corresponding to each generated topology ("dis.npy"); the CGS-0 approximations for the sensitivity vector of each topology ("sen_0.npy"); the CGS-1 approximations ("sen_1.npy"); the CGS-2 approximations ("sen_2.npy"); the exact sensitivity vectors, obtained through the WS expression ("sen_w.npy"); the relative volume values of each topology ("vol.npy"); the objective function values of each topology ("obj.npy"); the execution times of each task of the optimization processes, together with the number of performed iterations ("tim.npy").

Except for the density values, which are stored as single bits, all data is stored in 4-bytes format. For a more efficient storage of the generated data, each file stores the results from a group of 16 optimizations problems (this value is defined through the parameter "noptf"). Thus, each "fid.npy" file contains 64 bytes of data; each "inp.npy" file contains 256 bytes of data; each "top_opt.npy" file contains 4.0 kB of data; each "obj_opt.npy" file contains 64 bytes of data; each "ptr2opt.npy" file contains 68 bytes of data; and each "tim.npy" file contains 896 bytes of data. The other files depend on the number of iterations performed in each optimization process. Considering the average value of 82 iterations (so that each one of these lists has 83 entries), each "ptr2inp.npy" file would contain 5.2 kB of data; each "top.npy" file would contain 332.0 kB of data; each "dis.npy" file would contain 21.8 MB of data; each "sen_0.npy" would contain 10.4 MB of data; each "sen_1.npy" would contain 10.4 MB of data; each "sen_2.npy" would contain 10.4 MB of data; each "sen_w.npy" would contain 10.4 MB of data; each "vol.npy" file would contain 5.2 kB of data; and each "obj.npy" file would contain 5.2 kB of data. These add up to around 64 MB of data.

The 148 240 optimization problems will generate 9 265 of each of these files, which would result in around 579 GB of data. However, according to the Disk Usage Analyzer, a tool for analyzing disk usage for GNOME, the BESO dataset occupies around **617 GB** of disk (7% more than the amount of useful data).

Firstly, the necessary modules are imported and all used-defined parameters are set.

structural_beso

```
import os, sys, gc, pickle
import numpy as np
from time import time
from datetime import datetime
from scipy.sparse import coo_matrix
from sksparse.cholmod import analyze
sys.path.append('../../cython/')
from structural_bsens import str_cgs
from structural_filter import str_filter
VV = 0.015625     # maximal volume variation
TV = 0.031250     # maximal topology variation
rmax = 0.125      # sensitivity filter radius
patience = 20     # patience stop criterion
momentum = 0.50   # sensitivity momentum
Ey = 1.0          # Young's modulus
nu = 0.3          # Poisson's coefficient
epsk = 1e-6       # soft-kill parameter
Ly = 1.0          # cantilever height
small = 1e-14     # small value to compare float numbers
noptf   = 16      # number of optimizations to be stored in the same file
fid_ini = 0       # initial input index |run from input 0
fid_lim = 148240  # input index limit   |up to input 148239
```

The stiffness matrix for the bilinear square element in plane stress state is defined as “Ke” and its vectorized data is stored in “Kevec”. The stiffness variation matrix is defined as “dKe”, which corresponds to the stiffness change in the global matrix when the state of one element is switched (from solid to void, or from void to solid). Each possible non-zero block of the $\mathbf{H_i}$ matrices (Equation 18) is computed and stored as “H”, “H_01”, “H_67” and “H_0167”. The first one corresponds to an element which has no clamped degree of freedom; the second one corresponds to an element whose 0th and 1st degrees of freedom are clamped; the third one corresponds to an element whose 6th and 7th degrees of freedom are clamped; and the fourth one corresponds to an element whose 0th, 1st, 6th and 7th degrees of freedom are clamped (which are all possible configurations for the considered cantilever beam problems).

structural_beso

```
# Elemental Matrix (Quad4) - Plane Stress State
kk = (Ey/(1-nu**2))*np.array([ 1/2-nu/6 ,  1/8+nu/8, -1/4-nu/12, -1/8+3*nu/8,
                              -1/4+nu/12, -1/8-nu/8,      nu/6 ,  1/8-3*nu/8])
Ke = np.array([[kk[0],kk[1],kk[2],kk[3],kk[4],kk[5],kk[6],kk[7]],
               [kk[1],kk[0],kk[7],kk[6],kk[5],kk[4],kk[3],kk[2]],
               [kk[2],kk[7],kk[0],kk[5],kk[6],kk[3],kk[4],kk[1]],
               [kk[3],kk[6],kk[5],kk[0],kk[7],kk[2],kk[1],kk[4]],
               [kk[4],kk[5],kk[6],kk[7],kk[0],kk[1],kk[2],kk[3]],
               [kk[5],kk[4],kk[3],kk[2],kk[1],kk[0],kk[7],kk[6]],
               [kk[6],kk[3],kk[4],kk[1],kk[2],kk[7],kk[0],kk[5]],
               [kk[7],kk[2],kk[1],kk[4],kk[3],kk[6],kk[5],kk[0]]])
Kevec = Ke.ravel()
dKe = (1.0-epsk)*Ke  # stiffness variation of a topological change
# Elemental Matrix Factorizations
D,V = np.linalg.eigh(dKe)
mask = abs(D) > small
D = D[mask]
V = V[:,mask]
H = V*np.sqrt(D)
D,V = np.linalg.eigh(dKe[:,[2,3,4,5,6,7]][[2,3,4,5,6,7],:])
mask = abs(D) > small
D = D[mask]
V = V[:,mask]
H_01 = V*np.sqrt(D)
D,V = np.linalg.eigh(dKe[:,[0,1,2,3,4,5]][[0,1,2,3,4,5],:])
mask = abs(D) > small
D = D[mask]
V = V[:,mask]
H_67 = V*np.sqrt(D)
D,V = np.linalg.eigh(dKe[:,[2,3,4,5]][[2,3,4,5],:])
H_0167 = V*np.sqrt(D)
```

If there is no input data, a standard input is created. The directory where the output data will be stored is created. Both input-output log and time log files are opened to be written.

structural_beso

```
# check directories
if not os.path.exists('../input'):
    os.mkdir('../input')
if not os.path.exists('./output'):
    os.mkdir('./output')
if not os.path.exists('./output/run_{:06d}_{:06d}'.format(fid_ini,fid_lim-1)):
    os.mkdir('./output/run_{:06d}_{:06d}'.format(fid_ini,fid_lim-1))
# check input
if not os.path.exists('../input/inp_000000.pckl'):
    Ny = np.uint32(32)          # number of elements in y-axis
    bc_pos = np.float32(0.0)    # center of the restricted area
    bc_rad = np.float32(0.5)    # half-length (radius) of the restricted area
    ld_pos = np.float32(0.0)    # center of the loaded area
    ld_rad = np.float32(0.125)  # half-length (radius) of the loaded area
    finp = open('../input/inp_000000.pckl','wb')
    pickle.dump([Ny,bc_pos,bc_rad,ld_pos,ld_rad],finp)
    finp.close()
# open log files
if not os.path.exists('./output/run_{:06d}_{:06d}/logs'.format(fid_ini,fid_lim-1)):
    os.mkdir('./output/run_{:06d}_{:06d}/logs'.format(fid_ini,fid_lim-1))
iolog = open('./output/run_{:06d}_{:06d}/logs/io_log.txt'.format(fid_ini,fid_lim-1),'a')
tlog = open('./output/run_{:06d}_{:06d}/logs/time_log.txt'.format(fid_ini,fid_lim-1),'a')
iolog.truncate(0)
tlog.truncate(0)
```

The headers of the log files are written.

structural_beso

```
iolog.write('DISCRETE STRUCTURAL OPTIMIZATION (IO LOG)\n')    # write in IO log
iolog.write('=============================================================================================\n')
iolog.write('= OUTPUT :               input file id :             fid.npy                          =\n')
iolog.write('= ------ :                  input data :             inp.npy                          =\n')
iolog.write('= ------ :           optimized topology :          top_opt.npy                         =\n')
iolog.write('= ------ : optimized objective function :          obj_opt.npy                         =\n')
```

```
iolog.write('= ------ : pointer input > optimization :         ptr2opt.npy                                    =\n')
iolog.write('= ------ : pointer optimization > input :         ptr2inp.npy                                    =\n')
iolog.write('= ------ :             topology vectors :             top.npy                                    =\n')
iolog.write('= ------ :        displacements vectors :             dis.npy                                    =\n')
iolog.write('= ------ :    CGS-0 sensitivity vectors :           sen_0.npy                                    =\n')
iolog.write('= ------ :    CGS-1 sensitivity vectors :           sen_1.npy                                    =\n')
iolog.write('= ------ :    CGS-2 sensitivity vectors :           sen_2.npy                                    =\n')
iolog.write('= ------ :       WS sensitivity vectors :           sen_w.npy                                    =\n')
iolog.write('= ------ :                volume array :             vol.npy                                    =\n')
iolog.write('= ------ :     objective function array :             obj.npy                                    =\n')
iolog.write('= ------ :                  time array :             tim.npy                                    =\n')
iolog.write('================================================================================================\n')
iolog.write('       INPUT || ELEM Y :  BC POS : BC RAD :  LD POS : LD RAD ||            BEGIN :               END\n')
tlog.write('DISCRETE STRUCTURAL OPTIMIZATION (TIME LOG)\n')   # write in time log
tlog.write('=============================================================================================\n')
tlog.write('       INPUT ||    FILES :     MESH :   B-COND : ASSEMBLY :    PRE-S :   SOLVER :   POST-S ||----------\n')
tlog.write('-------------||           (  IT x ):   M-SENS : M-UPDATE :  M-PRE-S : M-SOLVER : M-POST-S ||     TOTAL\n')
```

The loop to go through all the selected input files is started. The output data lists are initialized. A nested loop is started in order to store together the results of each block of "noptf" optimization problems.

structural_beso

```
file = 0  # file counter
fid = fid_ini
while (fid < fid_lim) and (os.path.exists('../input/inp_{:06d}.pckl'.format(fid))):
    if not os.path.exists('./output/run_{:06d}_{:06d}/file_{:05d}'.format(fid_ini,fid_lim-1,file)):
        os.mkdir('./output/run_{:06d}_{:06d}/file_{:05d}'.format(fid_ini,fid_lim-1,file))
    list_fid     = []
    list_inp     = []
    list_top_opt = []
    list_obj_opt = []
    list_ptr2opt = []
    list_ptr2inp = []
    list_top     = []
    list_dis     = []
    list_sen_0   = []
    list_sen_1   = []
    list_sen_2   = []
    list_sen_w   = []
    list_obj     = []
    list_vol     = []
    list_tim     = []
    ptr = 0  # pointer to input
    for counter in range(noptf):
        if (fid >= fid_lim) or (not os.path.exists('../input/inp_{:06d}.pckl'.format(fid))):
            break
        print('running : {:06d} : setup'.format(fid))
        inp_file = 'inp_{:06d}.pckl'.format(fid)
        iolog.write('> ' + inp_file[:-5] + ' ||')
        tlog.write('> ' + inp_file[:-5] + ' ||')
```

The current input file is read and the input variables are cast to 8-bytes precision. The input index is appended to "list_fid" and the input data is appended to "list_inp". The input data is written in the input-output log. The parameters "Nx", "N" and "esize" are computed based on "Ny". The topology vector is initialized as "x", the initial topology corresponds to a fully solid structure. The time to read the input file and to setup these variables is stored in "time_array[0]".

structural_beso

```
        t0 = time()
        # read input file
        finp = open('../input/'+inp_file,'rb')
        Ny,bc_pos,bc_rad,ld_pos,ld_rad = pickle.load(finp)
        Ny = int(Ny)
        bc_pos = float(bc_pos)
        bc_rad = float(bc_rad)
        ld_pos = float(ld_pos)
        ld_rad = float(ld_rad)
        finp.close()
        list_fid = list_fid + [fid]
        list_inp = list_inp + [[bc_pos,bc_rad,ld_pos,ld_rad]]
        # write in log
        iolog.write(' {:6d} :'.format(Ny))
        iolog.write(' {:7.4f} :'.format(bc_pos))
        iolog.write(' {:6.4f} :'.format(bc_rad))
        iolog.write(' {:7.4f} :'.format(ld_pos))
        iolog.write(' {:6.4f} ||'.format(ld_rad))
        iolog.write(datetime.now().strftime(' %y/%m/%d-%H:%M:%S :'))
        # input properties
        Nx = 2*Ny                  # number of elements in x-axis
        N = Nx*Ny                  # total number of elements
        esize = Ly/Ny              # element size
        x = np.ones(N,dtype=bool)  # set initial topology
        # write in log
        time_array = np.zeros(14)  # initialize time array
```

```
155         t1 = time()
156         time_array[0] = t1 - t0
157         tlog.write(' {:6.3f} s :'.format(time_array[0]))
```

The mesh is generated by creating the matrix "coor" with the coordinates of each node of the mesh, and the matrix "inci" that relates each element to its nodes. The variable "G" stores the number of degrees of freedom of the unconstrained system. The time to generate the mesh is stored in "time_array[1]".

structural_beso

```
158         t0 = time()
159         # coordinates matrix
160         xcoor = (Ny+1)*[list(range(Nx+1))]
161         xcoor = np.ravel(xcoor,'F')
162         ycoor = (Nx+1)*[list(range(Ny+1))]
163         ycoor = np.ravel(ycoor,'C')
164         coor = esize*np.array([xcoor,ycoor]).T
165         coor[:,1] = coor[:,1] - 0.5*Ly
166         # incidence matrix
167         N = Nx*Ny
168         G = 2*(Nx+1)*(Ny+1)
169         inci = np.ndarray([N,4],dtype=int)
170         elem_ids = np.arange(N)
171         inci[:,0] = elem_ids + elem_ids//Ny
172         inci[:,1] = inci[:,0] + Ny + 1
173         inci[:,2] = inci[:,0] + Ny + 2
174         inci[:,3] = inci[:,0] + 1
175         # write in log
176         t1 = time()
177         time_array[1] = t1 - t0
178         tlog.write(' {:6.3f} s :'.format(time_array[1]))
```

The boundary conditions are processed. The boolean array "freeDofs" is created, it has "G" entries, they are *True* for the unconstrained degrees of freedom and *False* for the clamped degrees of freedom. The array "bc_lim" stores the indices of the first and the last clamped nodes. The number of degrees of freedom of the constrained system is stored in the variable "sys_size". The array "fg" is the external load vector of the unconstrained system and the array "fr" is the external load vector of the constrained system. The array "ld_lim" stores the indices of the first and last elements that are connected to the loaded region. The time to process the boundary conditions is stored in "time_array[2]".

structural_beso

```
179         t0 = time()
180         # free DOFs
181         bc_ycoor = coor[:Ny+1,1]
182         bc_mask = abs(bc_ycoor - bc_pos*Ly) < bc_rad*Ly + small
183         if sum(bc_mask) < 2:
184             print('insufficient constraint')
185             iolog.close()
186             tlog.close()
187             sys.exit()
188         bc_ids = np.arange(0,Ny+1)
189         bc_ids = bc_ids[bc_mask]
190         bc_lim = np.array([bc_ids[0],bc_ids[-1]],dtype="int64")
191         bc = np.concatenate((2*bc_ids,2*bc_ids+1))
192         freeDofs = np.ones(G,dtype=bool)
193         freeDofs[bc] = False
194         sys_size = sum(freeDofs)
195         # load vector
196         fg = np.zeros(G)
197         ld_ycoor = coor[Nx*(Ny+1):,1]
198         ld_mask = abs(ld_ycoor - ld_pos*Ly) < ld_rad*Ly + small
199         ld_mask_ele = ld_mask[1:] | ld_mask[:-1]
200         ld_ele = np.arange((Nx-1)*Ny,Nx*Ny)
201         ld_ele = ld_ele[ld_mask_ele]
202         ld_lim = np.array([ld_ele[0],ld_ele[-1]],dtype="int64")
203         ld_num = sum(ld_mask)
204         ld_ids = np.arange(Nx*(Ny+1),(Nx+1)*(Ny+1))
205         ld_ids = ld_ids[ld_mask]
206         if ld_num == 0:
207             pass
208         elif ld_num == 1:
209             fg[2*ld_ids+1] = -1.0
210         else:
211             ld_val = -1.0/(ld_num-1)
212             fg[2*ld_ids[1:-1]+1] = ld_val
213             fg[2*ld_ids[[0,-1]]+1] = 0.5*ld_val
214         fr = fg[freeDofs]
215         # write in log
216         t1 = time()
217         time_array[2] = t1 - t0
218         tlog.write(' {:6.3f} s :'.format(time_array[2]))
```

The matrix assembly is performed through the "scipy.sparse.coo_matrix" function. The variable "Kg_coo" is the COO (Coordinate list) unconstrained global stiffness matrix; "Kg_csc" is the CSC (Compressed Sparse

Column) unconstrained matrix; and “Kr” is the CSC constrained matrix. The time to perform the assembly is stored in “time_array[3]”.

structural_beso

```
        t0 = time()
        # COO data
        pen = np.ones(N)
        pen[~x] = epsk
        pen = pen.repeat(64)
        data = pen*np.tile(Kevec,N)
        # COO indices
        dof0 = 2*inci[:,0]
        dof1 = dof0 + 1
        dof2 = 2*inci[:,1]
        dof3 = dof2 + 1
        dof4 = 2*inci[:,2]
        dof5 = dof4 + 1
        dof6 = 2*inci[:,3]
        dof7 = dof6 + 1
        eledofs = np.array([dof0,dof1,dof2,dof3,dof4,dof5,dof6,dof7])
        row = eledofs.repeat(8,axis=0).ravel('F')
        col = eledofs.T.repeat(8,axis=0).ravel('C')
        # stiffness matrix
        Kg_coo = coo_matrix((data,(row,col)),shape=(G,G))
        Kg_csc = Kg_coo.tocsc()
        Kr = Kg_csc[freeDofs,:][:,freeDofs]
        # write in log
        t1 = time()
        time_array[3] = t1 - t0
        tlog.write(' :6.3f s :'.format(time_array[3]))
```

The unconstrained displacements vector is initialized as “ug”. The optimal fill-reducing permutation is computed for “Kr” using the “sksparse.cholmod.analyze” function. This is used for solving the linear systems through Cholesky factorization. The time to perform this task is stored in “time_array[4]”.

structural_beso

```
        t0 = time()
        # initialize displacements vector
        ug  = np.zeros(G)
        # analyze sparse matrix
        factor = analyze(Kr)
        # write in log
        t1 = time()
        time_array[4] = t1 - t0
        tlog.write(' {:6.3f} s :'.format(time_array[4]))
```

The constrained global stiffness matrix is factorized, then the linear system is solved and the displacements vector is obtained. The time to solve the linear system is stored in “time_array[5]”.

structural_beso

```
        t0 = time()
        # call solver
        factor.cholesky_inplace(Kr)
        ug[freeDofs] = factor(fr)
        # write in log
        t1 = time()
        time_array[5] = t1 - t0
        tlog.write(' {:6.3f} s :'.format(time_array[5]))
```

Before beginning the optimization iterations, the arrays that will store the sensitivity values are initialized, together with some auxiliary variables. The current values for the volume and objective functions are computed and appended to “list_vol” and “list_obj”. The value that relates the current input to the index of the current optimization in the block of “noptf” processes is appended to the list of pointers “list_ptr2opt”. The time to perform these initializations is stored in “time_array[6]”.

structural_beso

```
        t0 = time()
        # optimization setup
        alpha_0 = np.zeros(N)          # CGS-0
        alpha_1 = np.zeros(N)          # CGS-1
        alpha_2 = np.zeros(N)          # CGS-2
        alpha_r = np.zeros(N)          # raw sensitivity vector
        alpha_f = np.zeros(N)          # filtered sensitivity vector
        alpha_m = np.zeros(N)          # filtered sensitivity vector with momentum
        fe = np.zeros((sys_size,5))    # auxiliary matrix for WS approach
        Vt = int(N/2)                  # target volume
        dVmax = max([1,int(VV*N)])     # maximal volume change
        dXmax = max([2,TV*N])          # maximal topological change
        vol = sum(x)                   # volume
```

```
        list_vol = list_vol + [vol/N]  # volume progression
        obj = np.dot(ug,fg)            # objective function
        list_obj = list_obj + [obj]    # objective function progression
        obj_opt = np.infty
        keep_going = True
        waiting = 0
        it = 0
        size_list = len(list_ptr2inp)
        list_ptr2opt += [size_list]
        # write in log
        t1 = time()
        time_array[6] = t1 - t0
        tlog.write(' {:6.3f} s ||----------\n'.format(time_array[6]))
```

The optimization loop is started. The "structural_bsens.str_cgs" function is used to compute the CGS-0 (Equation 19), CGS-1 (Equation 25) and CGS-2 (Equation 28) approximations, which are stored in the arrays "alpha_0", "alpha_1" and "alpha_2". The cython script "structural_bsens.pyx" is detailed in a following section. The exact sensitivity expression (Equation 15) is computed and stored in the array "alpha_r".

structural_beso

```
        while keep_going:
            it = it + 1
            print('running : {:06d} :  {:4d}'.format(fid,it))
            # sensitivity analysis
            t0 = time()
            ur = ug[freeDofs]
            str_cgs(alpha_0, x, Kg_csc, bc_lim, dKe, ug, Nx, Ny, steps=0)
            str_cgs(alpha_1, x, Kg_csc, bc_lim, dKe, ug, Nx, Ny, steps=1)
            str_cgs(alpha_2, x, Kg_csc, bc_lim, dKe, ug, Nx, Ny, steps=2)
            for e in range(N):
                n0 = e + (e // Ny)
                n1 = n0 + Ny + 1
                n2 = n1 + 1
                n3 = n0 + 1
                nodes = np.array([n0,n1,n2,n3])
                freeNodes = (nodes < bc_lim[0]) | (nodes > bc_lim[1])
                nodes = nodes[freeNodes]
                mask = nodes > bc_lim[1]
                nodes[mask] = nodes[mask] - (bc_lim[1]-bc_lim[0]+1)
                gv = np.repeat(2*nodes,2)
                gv[1::2] = gv[1::2] + 1
                rank = 5
                if all(freeNodes):
                    He = H
                elif (not freeNodes[0]) and (freeNodes[-1]):
                    He = H_01
                elif (freeNodes[0]) and (not freeNodes[-1]):
                    He = H_67
                else:
                    He = H_0167
                    rank = 4
                Ai = np.zeros((rank,rank))
                fe[gv,:rank] = He
                aux = factor.solve_L(factor.apply_P(fe[:,:rank]),use_LDLt_decomposition=False)
                fe[gv,:rank] = 0.0
                Ai = aux.T @ aux
                vi = He.T @ ur[gv]
                Ii = np.identity(rank)
                if x[e]:
                    alpha_r[e] = -vi @ np.linalg.inv(Ii-Ai) @ vi
                else:
                    alpha_r[e] = -vi @ np.linalg.inv(Ii+Ai) @ vi
```

The list of pointers "list_ptr2inp" appends the value that relates the index of the current iteration in the block of "noptf" optimization processes to the current input. The current topology, displacements and sensitivity vectors are appended to "list_top", "list_dis", "list_sen_0", "list_sen_1", "list_sen_2" and "list_sen_w". The "structural_filter.str_filter" function is used to smooth the sensitivity map. The cython script "stuctural_filter.pyx" is detailed in a following section. The momentum is applied and the final sensitivity vector is stored in "alpha_m". The arrays "ld_lim" and "ld_ele" are used to ignore loaded elements in the sensitivity analysis, since their density values are considered non-design variables. The total time spent performing sensitivity analyses for the current optimization problem is stored in "time_array[7]".

structural_beso

```
            list_ptr2inp += [ptr]
            list_top     += [x.copy()]
            list_dis     += [ug.copy()]
            list_sen_0   += [alpha_0.copy()]
            list_sen_1   += [alpha_1.copy()]
            list_sen_2   += [alpha_2.copy()]
            list_sen_w   += [alpha_r.copy()]
            str_filter(alpha_r, alpha_f, rmax, esize, Nx, Ny, load_lim=ld_lim)
            alpha_m[ld_ele] = 0.0
```

```
            alpha_m = momentum*alpha_m + (1.0-momentum)*(alpha_f/max(abs(alpha_f)))
            alpha_m = alpha_m/max(abs(alpha_m))
            alpha_m[ld_ele] = -np.infty
            t1 = time()
            time_array[7] = time_array[7] + (t1-t0)
```

Using the sensitivity vector "alpha_m", the current volume value "vol", the target volume "Vt", the maximal volume variation "dVmax" and the maximal topological variation "dXmax", the BESO algorithm is used to solve the SILP subproblem and update the topology. For coarse meshes, it is faster to directly update the factor matrices after altering the state of each element, than it is to update the whole topology vector, alter the original matrix, then factorize it from scratch. The total time spent performing the BESO algorithm for the current optimization problem is stored in "time_array[8]".

structural_beso

```
            # update topology
            t0 = time()
            solid = np.argwhere(x)[:,0]
            void = np.argwhere(~x)[:,0]
            sorted_solid = np.argsort(alpha_m[solid])
            sorted_void = np.argsort(alpha_m[void])
            # changing volume
            count = 0
            for i in range(min([vol-Vt,dVmax])):
                es = solid[sorted_solid[-1-i]]
                x[es] = False
                Kg_coo.data[64*es:64*(es+1)] = epsk*Kevec
                # update Cholesky factor (faster for coarse meshes)
                n0 = es + (es // Ny)
                n1 = n0 + Ny + 1
                n2 = n1 + 1
                n3 = n0 + 1
                nodes = np.array([n0,n1,n2,n3])
                freeNodes = (nodes < bc_lim[0]) | (nodes > bc_lim[1])
                nodes = nodes[freeNodes]
                mask = nodes > bc_lim[1]
                nodes[mask] = nodes[mask] - (bc_lim[1]-bc_lim[0]+1)
                gv = np.repeat(2*nodes,2)
                gv[1::2] = gv[1::2] + 1
                lgv = len(gv)
                rank = 5
                if all(freeNodes):
                    hdata = H.ravel()
                elif (not freeNodes[0]) and (freeNodes[-1]):
                    hdata = H_01.ravel()
                elif (freeNodes[0]) and (not freeNodes[-1]):
                    hdata = H_67.ravel()
                else:
                    hdata = H_0167.ravel()
                    rank = 4
                hrow  = np.repeat(gv,rank)
                hcol  = np.tile(np.arange(rank),lgv)
                H_coo = coo_matrix((hdata,(hrow,hcol)),shape=(sys_size,rank))
                H_csc = H_coo.tocsc()
                factor.update_inplace(H_csc, subtract=True)
                count = count + 1
            # constant volume
            for i in range(min([len(sorted_void),int((dXmax-count)/2)])):
                es = solid[sorted_solid[-1-i-count]]
                ev = void[sorted_void[i]]
                if alpha_m[es] < alpha_m[ev]:
                    break
                x[es] = False
                x[ev] = True
                Kg_coo.data[64*es:64*(es+1)] = epsk*Kevec
                Kg_coo.data[64*ev:64*(ev+1)] = Kevec
                # update Cholesky factor (faster for coarse meshes)
                n0 = es + (es // Ny)
                n1 = n0 + Ny + 1
                n2 = n1 + 1
                n3 = n0 + 1
                nodes = np.array([n0,n1,n2,n3])
                freeNodes = (nodes < bc_lim[0]) | (nodes > bc_lim[1])
                nodes = nodes[freeNodes]
                mask = nodes > bc_lim[1]
                nodes[mask] = nodes[mask] - (bc_lim[1]-bc_lim[0]+1)
                gv = np.repeat(2*nodes,2)
                gv[1::2] = gv[1::2] + 1
                lgv = len(gv)
                rank = 5
                if all(freeNodes):
                    hdata = H.ravel()
                elif (not freeNodes[0]) and (freeNodes[-1]):
                    hdata = H_01.ravel()
                elif (freeNodes[0]) and (not freeNodes[-1]):
                    hdata = H_67.ravel()
                else:
```

```
                    hdata = H_0167.ravel()
                    rank = 4
                hrow  = np.repeat(gv,rank)
                hcol  = np.tile(np.arange(rank),lgv)
                H_coo = coo_matrix((hdata,(hrow,hcol)),shape=(sys_size,rank))
                H_csc = H_coo.tocsc()
                factor.update_inplace(H_csc, subtract=True)
                n0 = ev + (ev // Ny )
                n1 = n0 + Ny + 1
                n2 = n1 + 1
                n3 = n0 + 1
                nodes = np.array([n0,n1,n2,n3])
                freeNodes = (nodes < bc_lim[0]) | (nodes > bc_lim[1])
                nodes = nodes[freeNodes]
                mask = nodes > bc_lim[1]
                nodes[mask] = nodes[mask] - (bc_lim[1]-bc_lim[0]+1)
                gv = np.repeat(2*nodes,2)
                gv[1::2] = gv[1::2] + 1
                lgv = len(gv)
                rank = 5
                if all(freeNodes):
                    hdata = H.ravel()
                elif (not freeNodes[0]) and (freeNodes[-1]):
                    hdata = H_01.ravel()
                elif (freeNodes[0]) and (not freeNodes[-1]):
                    hdata = H_67.ravel()
                else:
                    hdata = H_0167.ravel()
                    rank = 4
                hrow  = np.repeat(gv,rank)
                hcol  = np.tile(np.arange(rank),lgv)
                H_coo = coo_matrix((hdata,(hrow,hcol)),shape=(sys_size,rank))
                H_csc = H_coo.tocsc()
                factor.update_inplace(H_csc, subtract=False)
            t1 = time()
            time_array[8] = time_array[8] + (t1-t0)
```

The updated stiffness matrix is re-assembled and the new displacements vector is computed. The current values of the volume and objective functions are appended to "list_vol" and "list_obj". If a better candidate for the optimized topology is obtained, "x_opt" and "obj_opt" are updated to store the new candidate topology and the corresponding objective function value. If the stopping criterion is achieved, the variable "keep_going" is switched to *False*, so the optimization loop is broken. The total times spent re-assembling the stiffness matrix, computing the displacements vector and performing the post-solver tasks are respectively stored in "time_array[9]", "time_array[10]" and "time_array[11]".

structural_beso

```
            # assembly
            t0 = time()
            Kg_csc = Kg_coo.tocsc()
            t1 = time()
            time_array[9] = time_array[9] + (t1-t0)
            # solver
            t0 = time()
            ug[freeDofs] = factor(fr)
            t1 = time()
            time_array[10] = time_array[10] + (t1-t0)
            # post-solver
            t0 = time()
            vol = sum(x)
            list_vol = list_vol + [vol/N]
            obj = np.dot(ug,fg)
            list_obj = list_obj + [obj]
            if vol == Vt:
                # update optimized topology
                if obj < (1.0-small) * obj_opt:
                    x_opt = x.copy()
                    obj_opt = obj
                    waiting = 0
                else:
                    waiting = waiting + 1
                    # check convergence
                    if waiting == patience:
                        keep_going = False
            t1 = time()
            time_array[11] = time_array[11] + (t1-t0)
```

After concluding the current optimization process, the sensitivity analysis is performed for the topology of the last iteration.

structural_beso

```
        # sensitivity analysis
        ur = ug[freeDofs]
        str_cgs(alpha_0, x, Kg_csc, bc_lim, dKe, ug, Nx, Ny, steps=0)
```

```
        str_cgs(alpha_1, x, Kg_csc, bc_lim, dKe, ug, Nx, Ny, steps=1)
        str_cgs(alpha_2, x, Kg_csc, bc_lim, dKe, ug, Nx, Ny, steps=2)
        for e in range(N):
            n0 = e + (e // Ny)
            n1 = n0 + Ny + 1
            n2 = n1 + 1
            n3 = n0 + 1
            nodes = np.array([n0,n1,n2,n3])
            freeNodes = (nodes < bc_lim[0]) | (nodes > bc_lim[1])
            nodes = nodes[freeNodes]
            mask = nodes > bc_lim[1]
            nodes[mask] = nodes[mask] - (bc_lim[1]-bc_lim[0]+1)
            gv = np.repeat(2*nodes,2)
            gv[1::2] = gv[1::2] + 1
            rank = 5
            if all(freeNodes):
                He = H
            elif (not freeNodes[0]) and (freeNodes[-1]):
                He = H_01
            elif (freeNodes[0]) and (not freeNodes[-1]):
                He = H_67
            else:
                He = H_0167
                rank = 4
            Ai = np.zeros((rank,rank))
            fe[gv,:rank] = He
            aux = factor.solve_L(factor.apply_P(fe[:,:rank]),use_LDLt_decomposition=False)
            fe[gv,:rank] = 0.0
            Ai = aux.T @ aux
            vi = He.T @ ur[gv]
            Ii = np.identity(rank)
            if x[e]:
                alpha_r[e] = -vi @ np.linalg.inv(Ii-Ai) @ vi
            else:
                alpha_r[e] = -vi @ np.linalg.inv(Ii+Ai) @ vi
```

The total time to perform the current optimization process is stored in "time_array[12]". The time array is updated to store the average times of the tasks performed in the optimization loop. The number of iterations performed in the current optimization process is stored in "time_array[13]". The execution times are written in the time log. The optimized topology and the corresponding objective function value are appended to "list_top_opt" and "list_obj_opt". The results corresponding to the last iteration of the current optimization process are appended to "list_ptr2inp", "list_top", "list_dis", "list_sen_0", "list_sen_1", "list_sen_2" and "list_sen_w". The "time_array" is appended to "list_tim". The pointer variable "ptr" and the input index variable "fid" are updated so the optimization process for the next input can start.

structural_beso

```
        # write in log
        tlog.write('-------------||            ({:4d} x ):'.format(it))
        time_array[12] = sum(time_array[:12])
        time_array[7:12] = time_array[7:12]/it
        time_array[13] = (1+small)*it
        tlog.write(' {:6.3f} s : {:6.3f} s : {:6.3f} s : {:6.3f} s : {:6.3f} s ||'.format(
            time_array[7],time_array[8],time_array[9],time_array[10],time_array[11]))
        tlog.write(' {:7.1f} s\n'.format(time_array[12]))
        iolog.write(datetime.now().strftime(' %y/%m/%d-%H:%M:%S\n'))
        list_top_opt += [x_opt.copy()]
        list_obj_opt += [obj_opt]
        list_ptr2inp += [ptr]
        list_top     += [x.copy()]
        list_dis     += [ug.copy()]
        list_sen_0   += [alpha_0.copy()]
        list_sen_1   += [alpha_1.copy()]
        list_sen_2   += [alpha_2.copy()]
        list_sen_w   += [alpha_r.copy()]
        list_tim     += [time_array.copy()]
        # update pointer
        ptr = ptr + 1
        # prepare to open next input file
        fid = fid + 1
```

After performing "noptf" optimization processes, the corresponding output data is written in disk. The last value of "list_ptr2opt" is appended. Then, each output list is saved in an independent file. Afterward, the output variables are deleted and "gc.collect()" is called to ensure that the RAM be freed. The file counter "file" is updated so the next block of "noptf" optimization processes can start.

structural_beso

```
    size_list = len(list_ptr2inp)
    list_ptr2opt += [size_list]
    # save files
    np.save('./output/run_{:06d}_{:06d}/file_{:05d}/fid.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_fid,dtype=np.uint32))
    np.save('./output/run_{:06d}_{:06d}/file_{:05d}/inp.npy'.format(
```

```
            fid_ini,fid_lim-1,file),np.array(list_inp,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/top_opt.npy'.format(
            fid_ini,fid_lim-1,file),np.packbits(np.array(list_top_opt),axis=1))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/obj_opt.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_obj_opt,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/ptr2opt.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_ptr2opt,dtype=np.uint32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/ptr2inp.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_ptr2inp,dtype=np.uint32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/top.npy'.format(
            fid_ini,fid_lim-1,file),np.packbits(np.array(list_top),axis=1))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/dis.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_dis,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/sen_0.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_sen_0,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/sen_1.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_sen_1,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/sen_2.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_sen_2,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/sen_w.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_sen_w,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/obj.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_obj,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/vol.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_vol,dtype=np.float32))
        np.save('./output/run_{:06d}_{:06d}/file_{:05d}/tim.npy'.format(
            fid_ini,fid_lim-1,file),np.array(list_tim,dtype=np.float32))
        del list_fid, list_inp, list_top_opt, list_obj_opt, list_ptr2opt, list_ptr2inp, list_top, list_dis
        del list_sen_0, list_sen_1, list_sen_2, list_sen_w, list_obj, list_vol, list_tim
        gc.collect()
        # prepare to write next output file
        file = file + 1
```

When all selected optimization processes are done, the log files are closed and the program terminates.

structural_beso

```
iolog.close()
tlog.close()
print('done!')
```

3.3.3 ./source/python/SIMP/structural_simp.py

This script performs the SIMP-MMA optimization procedure for the selected input data. Two text files are written. The input-output log ("io_log.txt") lists what this program generates as output data and presents each considered input data, together with the dates and times that each optimization procedure began and ended their execution. The time log ("time_log.txt") presents the execution times of each task of the optimization procedures.

In order to generate consistent data, all topology vectors are stored considering that $p = 1$, that is, considering that there is no SIMP penalization. The adjusted density vector, $\mathbf{z}$, is given by: $z_i = {x_i}^p$. Thus, it represents the actual stiffness distribution over the mesh. The same is done when storing the sensitivity vectors, volume array and gray level array. The adjusted sensitivity vector is given by $\beta_i = \frac{\partial C(\mathbf{x}(\mathbf{z}))}{\partial z_i}$. The adjusted volume value is given by $V(\mathbf{z})$. And the adjusted gray level is given by $g(\mathbf{z})$.

The output data is composed by numpy arrays with: the input files indices ("fid.npy"); the input data ("inp.npy"); the optimized topologies ("top_p1_opt.npy"); the discretized optimized topologies ("dtop_opt.npy"); the optimized objective function values ("obj_opt.npy"); the objective values of the discretized solutions ("dobj_opt.npy"); pointers relating each input with the corresponding iterations of the optimization processes ("ptr2opt.npy"); pointers relating each iteration of the optimization processes with the corresponding inputs ("ptr2inp.npy"); all generated topology vectors ("top_p1.npy"); the displacements vectors corresponding to each generated topology ("dis.npy"); the sensitivity vector of each topology ("sen_p1.npy"); the relative volume values of each topology ("vol_p1.npy"); the gray level values of each topology ("gra_p1.npy"); the objective function values of each topology ("obj.npy"); the execution times of each task of the optimization processes, together with the number of performed iterations ("tim.npy").

Except for the density values of the discretized solutions, which are stored as single bits, all data is stored in 4-bytes format. For a more efficient storage of the generated data, each file stores the results from a group of 16 optimizations problems (this value is defined through the parameter "noptf"). Thus, each "fid.npy" file contains 64 bytes of data; each "inp.npy" file contains 256 bytes of data; each "top_p1_opt.npy" file contains 128.0 kB of data; each "dtop_opt.npy" file contains 4.0 kB of data; each "obj_opt.npy" file contains 64 bytes of data; each "dobj_opt.npy" file contains 64 bytes of data; each "ptr2opt.npy" file contains 68 bytes of data; and each "tim.npy" file contains 576 bytes of data. The other files depend on the number of iterations performed in each optimization process. Considering the average value of 72 iterations (so that each one of these lists has

73 entries), each “ptr2inp.npy” file would contain 4.6 kB of data; each “top_p1.npy” file would contain 9.1 MB of data; each “dis.npy” file would contain 19.1 MB of data; each “sen_p1.npy” would contain 9.1 MB of data; each “vol_p1.npy” file would contain 4.6 kB of data; each “gra_p1.npy” file would contain 4.6 kB of data; and each “obj.npy” file would contain 4.6 kB of data. These add up to around 38 MB of data.

The 148 240 optimization problems will generate 9 265 of each of these files, which would result in around 344 GB of data. However, according to the Disk Usage Analyzer, a tool for analyzing disk usage for GNOME, the SIMP dataset occupies around **363 GB** of disk (6% more than the amount of useful data).

Firstly, the necessary modules are imported and all used-defined parameters are set.

structural_simp

```
import os, sys, gc, pickle
import numpy as np
from time import time
from datetime import datetime
from scipy.sparse import coo_matrix
from sksparse.cholmod import analyze
from structural_mma import solve_mma
sys.path.append('../../cython/')
from structural_ssens import str_ssens
p    = [1.0, 3.0, 6.0]  # penalization exponents
meva = [ 16,  16,   0]  # max number of evaluations
miev = [  1,   1,   5]  # max number of inner evaluations
rmax = 0.125            # sensitivity filter radius
Ey = 1.0                # Young's modulus
nu = 0.3                # Poisson's coefficient
epsk = 1e-6             # soft-kill parameter
Ly = 1.0                # cantilever height
small = 1e-14           # small value to compare float numbers
noptf   = 16            # number of optimizations to be stored in the same file
fid_ini = 0             # initial input index |run from input 0
fid_lim = 148240        # input index limit   |up to input 148239
```

The stiffness matrix for the bilinear square element in plane stress state is defined as “Ke” and its vectorized data is stored in “Kevec”. The stiffness variation matrix is defined as “dKe”, which corresponds to the stiffness change in the global matrix when the state of one element is switched (from solid to void, or from void to solid).

structural_simp

```
# Elemental Matrix (Quad4) - Plane Stress State
kk = (Ey/(1-nu**2))*np.array([ 1/2-nu/6 ,  1/8+nu/8, -1/4-nu/12, -1/8+3*nu/8,
                              -1/4+nu/12, -1/8-nu/8,      nu/6 ,  1/8-3*nu/8])
Ke = np.array([[kk[0],kk[1],kk[2],kk[3],kk[4],kk[5],kk[6],kk[7]],
               [kk[1],kk[0],kk[7],kk[6],kk[5],kk[4],kk[3],kk[2]],
               [kk[2],kk[7],kk[0],kk[5],kk[6],kk[3],kk[4],kk[1]],
               [kk[3],kk[6],kk[5],kk[0],kk[7],kk[2],kk[1],kk[4]],
               [kk[4],kk[5],kk[6],kk[7],kk[0],kk[1],kk[2],kk[3]],
               [kk[5],kk[4],kk[3],kk[2],kk[1],kk[0],kk[7],kk[6]],
               [kk[6],kk[3],kk[4],kk[1],kk[2],kk[7],kk[0],kk[5]],
               [kk[7],kk[2],kk[1],kk[4],kk[3],kk[6],kk[5],kk[0]]])
Kevec = Ke.ravel()
dKe = (1.0-epsk)*Ke  # stiffness variation of a topological change
```

If there is no input data, a standard input is created. The directory where the output data will be stored is created. Both input-output log and time log files are opened to be written.

structural_simp

```
# check directories
if not os.path.exists('../input'):
    os.mkdir('../input')
if not os.path.exists('./output'):
    os.mkdir('./output')
if not os.path.exists('./output/run_{:06d}_{:06d}'.format(fid_ini,fid_lim-1)):
    os.mkdir('./output/run_{:06d}_{:06d}'.format(fid_ini,fid_lim-1))
# check input
if not os.path.exists('../input/inp_000000.pckl'):
    Ny = np.uint32(32)          # number of elements in y-axis
    bc_pos = np.float32(0.0)    # center of the restricted area
    bc_rad = np.float32(0.5)    # half-length (radius) of the restricted area
    ld_pos = np.float32(0.0)    # center of the loaded area
    ld_rad = np.float32(0.125)  # half-length (radius) of the loaded area
    finp = open('../input/inp_000000.pckl','wb')
    pickle.dump([Ny,bc_pos,bc_rad,ld_pos,ld_rad],finp)
    finp.close()
# open log files
if not os.path.exists('./output/run_{:06d}_{:06d}/logs'.format(fid_ini,fid_lim-1)):
    os.mkdir('./output/run_{:06d}_{:06d}/logs'.format(fid_ini,fid_lim-1))
iolog = open('./output/run_{:06d}_{:06d}/logs/io_log.txt'.format(fid_ini,fid_lim-1),'a')
tlog = open('./output/run_{:06d}_{:06d}/logs/time_log.txt'.format(fid_ini,fid_lim-1),'a')
iolog.truncate(0)
tlog.truncate(0)
```

The headers of the log files are written.

```
                                                                    structural_simp
iolog.write('CONTINUOUS STRUCTURAL OPTIMIZATION (IO LOG)\n')    # write in IO log
iolog.write('=============================================================================================================\n')
iolog.write('= OUTPUT :                input file id :             fid.npy                                          =\n')
iolog.write('= ------ :                   input data :             inp.npy                                          =\n')
iolog.write('= ------ :           optimized topology :       top_p1_opt.npy                                         =\n')
iolog.write('= ------ :            discrete solution :         dtop_opt.npy                                         =\n')
iolog.write('= ------ : optimized objective function :          obj_opt.npy                                         =\n')
iolog.write('= ------ :  discrete solution objective :         dobj_opt.npy                                         =\n')
iolog.write('= ------ : pointer input > optimization :          ptr2opt.npy                                         =\n')
iolog.write('= ------ : pointer optimization > input :          ptr2inp.npy                                         =\n')
iolog.write('= ------ :             topology vectors :           top_p1.npy                                         =\n')
iolog.write('= ------ :        displacements vectors :              dis.npy                                         =\n')
iolog.write('= ------ :          sensitivity vectors :           sen_p1.npy                                         =\n')
iolog.write('= ------ :                 volume array :           vol_p1.npy                                         =\n')
iolog.write('= ------ :              gray level array :           gra_p1.npy                                         =\n')
iolog.write('= ------ :     objective function array :              obj.npy                                         =\n')
iolog.write('= ------ :                  time array :              tim.npy                                         =\n')
iolog.write('=============================================================================================================\n')
iolog.write('       INPUT || ELEM Y :  BC POS : BC RAD :  LD POS : LD RAD ||             BEGIN :                END\n')
tlog.write('CONTINUOUS STRUCTURAL OPTIMIZATION (TIME LOG)\n')   # write in time log
tlog.write('=================================================================================\n')
tlog.write('       INPUT ||    FILES :     MESH :   B-COND : ASSEMBLY :    OPT-S ||----------\n')
tlog.write('-------------|| (  IT x ):    M-OPT :          :          :   POST-P ||     TOTAL\n')
```

The loop to go through all the selected input files is started. The output data lists are initialized. The variable "lists" is created to be used as input of the "structural_mma.solve_mma" function, which performs the MMA optimization procedure. A nested loop is started in order to store together the results of each block of "noptf" optimization problems.

```
                                                                    structural_simp
file = 0  # file counter
fid = fid_ini
while (fid < fid_lim) and (os.path.exists('../input/inp_{:06d}.pckl'.format(fid))):
    if not os.path.exists('./output/run_{:06d}_{:06d}/file_{:05d}'.format(fid_ini,fid_lim-1,file)):
        os.mkdir('./output/run_{:06d}_{:06d}/file_{:05d}'.format(fid_ini,fid_lim-1,file))
    list_fid         = []
    list_inp         = []
    list_top_p1_opt  = []
    list_dtop_opt    = []
    list_obj_opt     = []
    list_dobj_opt    = []
    list_ptr2opt     = []
    list_ptr2inp     = []
    list_top_p1      = []
    list_dis         = []
    list_sen_p1      = []
    list_obj         = []
    list_gra_p1      = []
    list_vol_p1      = []
    list_tim         = []
    lists            = [list_ptr2inp, list_top_p1, list_dis, list_sen_p1, list_obj, list_gra_p1, list_vol_p1]
    ptr = 0  # pointer to input
    for counter in range(noptf):
        if (fid >= fid_lim) or (not os.path.exists('../input/inp_{:06d}.pckl'.format(fid))):
            break
        print('running : {:06d} : setup'.format(fid))
        inp_file = 'inp_{:06d}.pckl'.format(fid)
        iolog.write('> ' + inp_file[:-5] + ' ||')
        tlog.write('> ' + inp_file[:-5] + ' ||')
```

The current input file is read and the input variables are cast to 8-bytes precision. The input index is appended to "list_fid" and the input data is appended to "list_inp". The input data is written in the input-output log. The parameters "Nx", "N" and "esize" are computed based on "Ny". The topology vector is initialized as "x", the initial topology corresponds to a perfectly gray structure, in which all density values are 0.5. The time to read the input file and to setup these variables is stored in "time_array[0]".

```
                                                                    structural_simp
        t0 = time()
        # read input file
        finp = open('../input/'+inp_file,'rb')
        Ny,bc_pos,bc_rad,ld_pos,ld_rad = pickle.load(finp)
        Ny = int(Ny)
        bc_pos = float(bc_pos)
        bc_rad = float(bc_rad)
        ld_pos = float(ld_pos)
        ld_rad = float(ld_rad)
        finp.close()
        list_fid += [fid]
        list_inp += [[bc_pos,bc_rad,ld_pos,ld_rad]]
```

```
123         # write in log
124         iolog.write(' {:6d} :'.format(Ny))
125         iolog.write(' {:7.4f} :'.format(bc_pos))
126         iolog.write(' {:6.4f} :'.format(bc_rad))
127         iolog.write(' {:7.4f} :'.format(ld_pos))
128         iolog.write(' {:6.4f} ||'.format(ld_rad))
129         iolog.write(datetime.now().strftime(' %y/%m/%d-%H:%M:%S :'))
130         # input properties
131         Nx = 2*Ny                 # number of elements in x-axis
132         N = Nx*Ny                 # total number of elements
133         esize = Ly/Ny             # element size
134         x = 0.5*np.ones(N)        # set initial topology
135         # write in log
136         time_array = np.zeros(9)  # initialize time array
137         t1 = time()
138         time_array[0] = t1 - t0
139         tlog.write(' {:6.3f} s :'.format(time_array[0]))
```

The mesh is generated by creating the matrix "coor" with the coordinates of each node of the mesh, and the matrix "inci" that relates each element to its nodes. The variable "G" stores the number of degrees of freedom of the unconstrained system. The time to generate the mesh is stored in "time_array[1]".

structural_simp

```
140         t0 = time()
141         # coordinates matrix
142         xcoor = (Ny+1)*[list(range(Nx+1))]
143         xcoor = np.ravel(xcoor,'F')
144         ycoor = (Nx+1)*[list(range(Ny+1))]
145         ycoor = np.ravel(ycoor,'C')
146         coor = esize*np.array([xcoor,ycoor]).T
147         coor[:,1] = coor[:,1] - 0.5*Ly
148         # incidence matrix
149         N = Nx*Ny
150         G = 2*(Nx+1)*(Ny+1)
151         inci = np.ndarray([N,4],dtype=int)
152         elem_ids = np.arange(N)
153         inci[:,0] = elem_ids + elem_ids//Ny
154         inci[:,1] = inci[:,0] + Ny + 1
155         inci[:,2] = inci[:,0] + Ny + 2
156         inci[:,3] = inci[:,0] + 1
157         # write in log
158         t1 = time()
159         time_array[1] = t1 - t0
160         tlog.write(' {:6.3f} s :'.format(time_array[1]))
```

The boundary conditions are processed. The boolean array "freeDofs" is created, it has "G" entries, they are *True* for the unconstrained degrees of freedom and *False* for the clamped degrees of freedom. The number of degrees of freedom of the constrained system is stored in the variable "sys_size". The array "fg" is the external load vector of the unconstrained system and the array "fr" is the external load vector of the constrained system. The time to process the boundary conditions is stored in "time_array[2]".

structural_simp

```
161         t0 = time()
162         # free DOFs
163         bc_ycoor = coor[:Ny+1,1]
164         bc_mask = abs(bc_ycoor - bc_pos*Ly) < bc_rad*Ly + small
165         if sum(bc_mask) < 2:
166             print('insufficient constraint')
167             iolog.close()
168             tlog.close()
169             sys.exit()
170         bc_ids = np.arange(0,Ny+1)
171         bc_ids = bc_ids[bc_mask]
172         bc = np.concatenate((2*bc_ids,2*bc_ids+1))
173         freeDofs = np.ones(G,dtype=bool)
174         freeDofs[bc] = False
175         sys_size = sum(freeDofs)
176         # load vector
177         fg = np.zeros(G)
178         ld_ycoor = coor[Nx*(Ny+1):,1]
179         ld_mask = abs(ld_ycoor - ld_pos*Ly) < ld_rad*Ly + small
180         ld_mask_ele = ld_mask[1:] | ld_mask[:-1]
181         ld_ele = np.arange((Nx-1)*Ny,Nx*Ny)
182         ld_ele = ld_ele[ld_mask_ele]
183         ld_num = sum(ld_mask)
184         ld_ids = np.arange(Nx*(Ny+1),(Nx+1)*(Ny+1))
185         ld_ids = ld_ids[ld_mask]
186         if ld_num == 0:
187             pass
188         elif ld_num == 1:
189             fg[2*ld_ids+1] = -1.0
190         else:
191             ld_val = -1.0/(ld_num-1)
192             fg[2*ld_ids[1:-1]+1] = ld_val
```

```
            fg[2*ld_ids[[0,-1]]+1] = 0.5*ld_val
        fr = fg[freeDofs]
        # write in log
        t1 = time()
        time_array[2] = t1 - t0
        tlog.write(' {:6.3f} s :'.format(time_array[2]))
```

The matrix assembly is performed through the "scipy.sparse.coo_matrix" function. The variable "Kg_coo" is the COO (Coordinate list) unconstrained global stiffness matrix, "Kg_csc" is the CSC (Compressed Sparse Column) unconstrained matrix; and "Kr" is the CSC constrained matrix. The time to perform the assembly is stored in "time_array[3]".

structural_simp

```
        t0 = time()
        # COO data
        pen = epsk + (1.0-epsk)*(x**p[0])
        pen = pen.repeat(64)
        data = pen*np.tile(Kevec,N)
        # COO indices
        dof0 = 2*inci[:,0]
        dof1 = dof0 + 1
        dof2 = 2*inci[:,1]
        dof3 = dof2 + 1
        dof4 = 2*inci[:,2]
        dof5 = dof4 + 1
        dof6 = 2*inci[:,3]
        dof7 = dof6 + 1
        eledofs = np.array([dof0,dof1,dof2,dof3,dof4,dof5,dof6,dof7])
        row = eledofs.repeat(8,axis=0).ravel('F')
        col = eledofs.T.repeat(8,axis=0).ravel('C')
        # stiffness matrix
        Kg_coo = coo_matrix((data,(row,col)),shape=(G,G))
        Kg_csc = Kg_coo.tocsc()
        Kr = Kg_csc[freeDofs,:][:,freeDofs]
        # write in log
        t1 = time()
        time_array[3] = t1 - t0
        tlog.write(' {:6.3f} s :'.format(time_array[3]))
```

The optimal fill-reducing permutation is computed for "Kr" using the "sksparse.cholmod.analyze" function. This is used for solving the linear systems through Cholesky factorization. The time to perform this task is stored in "time_array[4]".

structural_simp

```
        t0 = time()
        # analyze sparse matrix
        factor = analyze(Kr)
        # write in log
        t1 = time()
        time_array[4] = t1 - t0
        tlog.write(' {:6.3f} s ||----------\n'.format(time_array[4]))
```

The value that relates the current input to the index of the current optimization in the block of "noptf" processes is appended to the list of pointers "list_ptr2opt". The optimization loop is started. The MMA optimization procedure is performed for the selected progression of SIMP penalization exponents, defined through the variable "p". For each penalization exponent, a maximal number of evaluations and a maximal number of inner iterations are defined through the variables "meva" and "miev". The "structural_mma.solve_mma" function is detailed in the next section. This function receives a part of the output lists, stored in the variable "lists", as input. This is done in order to append the data generated throughout the iterations of the optimization procedure to the corresponding output lists. The number of MMA iterations is stored in the variable "it", the adjusted optimized topology is appended to "list_top_p1_opt" and the corresponding objective function value is appended to "list_obj_opt". The total time spent to perform the optimization loop is stored in "time_array[5]".

structural_simp

```
        t0=time()
        size_list = len(list_ptr2inp)
        list_ptr2opt += [size_list]
        i0 = len(list_obj)
        for k in range(len(p)):
            print('------- : ------ : p = {:.1f} : max_eval = {:02d}'.format(p[k],meva[k]))
            x, h = solve_mma(x, p[k], epsk, row, col, Nx, Ny, G, Kevec, dKe, freeDofs, fr, rmax, esize,
                             lists, ptr, factor, meval=meva[k], inner_meval=miev[k])
            it = len(list_obj)-i0-1
            print('------- : ------ : [{:4d} iterations]'.format(it))
        list_top_p1_opt += [x**p[-1]]
        list_obj_opt    += [h]
        t1 = time()
        time_array[5] = t1 - t0
```

After concluding the current optimization process, the optimized topology is discretized. Firstly, all elements that are connected to the loaded region are turned into solid ones, with $x_i = 1$. Then, the elements are sorted by their density values. Respecting the volume constraint, low density elements are turned into void ones, with $x_i = 0$; and high density elements are turned into solid ones, with $x_i = 1$.

structural_simp

```
245         t0=time()
246         # obtain discrete solution
247         x[ld_ele] = 1.0
248         y = np.ones(N,dtype=bool)
249         y[np.argsort(x)[:N//2]] = False
```

The displacements vector and the objective function are computed for the discretized solution. The time spent performing these post-processing tasks is stored in "time_array[6]".

structural_simp

```
250         ug = np.zeros(G)
251         pen = np.ones(N)
252         pen[~y] = epsk
253         pen = pen.repeat(64)
254         data = pen*np.tile(Kevec,N)
255         Kg_coo = coo_matrix((data,(row,col)),shape=(G,G))
256         Kg_csc = Kg_coo.tocsc()
257         Kr = Kg_csc[freeDofs,:][:,freeDofs]
258         factor.cholesky_inplace(Kr)
259         ug[freeDofs] = factor(fr)
260         h = np.dot(ug[freeDofs],fr)
261         t1 = time()
262         time_array[6] = t1 - t0
```

The total time to perform the current optimization process is stored in "time_array[7]". The time array is updated to store the average time of the tasks performed in the optimization loop. The number of iterations performed in the current optimization process is stored in "time_array[8]". The execution times are written in the time log. The discrete solution and the corresponding objective function value are appended to "list_dtop_opt" and "list_dobj_opt". The results corresponding to the discrete solution are appended to "list_obj", "list_gra_p1", "list_vol_p1", "list_ptr2inp", "list_top_p1", "list_dis" and "list_sen_p1". The "time_array" is appended to "list_tim". The pointer variable "ptr" and the input index variable "fid" are updated so the optimization process for the next input can start.

structural_simp

```
263         # write in log
264         tlog.write('-------------|| ({:4d} x ):'.format(it))
265         time_array[7] = sum(time_array[:7])
266         time_array[5] = time_array[5]/it
267         time_array[8] = (1+small)*it
268         tlog.write(' {:6.3f} s :          :          : {:6.3f} s ||'.format(time_array[5],time_array[6]))
269         tlog.write(' {:7.1f} s\n'.format(time_array[7]))
270         iolog.write(datetime.now().strftime(' %y/%m/%d-%H:%M:%S\n'))
271         list_dtop_opt += [y.copy()]
272         list_dobj_opt += [h]
273         list_obj      += [h]
274         list_gra_p1   += [4*np.mean(y*(1.0-y))]
275         list_vol_p1   += [sum(y)/N]
276         list_ptr2inp  += [ptr]
277         list_top_p1   += [y.astype(float)]
278         list_dis      += [ug.copy()]
279         alpha_r = np.zeros(N)
280         str_ssens(alpha_r, y.astype(float), dKe, ug, 1.0, Nx, Ny)
281         list_sen_p1   += [alpha_r.copy()]
282         list_tim      += [time_array.copy()]
283         # update pointer
284         ptr = ptr + 1
285         # prepare to open next input file
286         fid = fid + 1
```

After performing "noptf" optimization processes, the corresponding output data is written in disk. The last value of "list_ptr2opt" is appended. Then, each output list is saved in an independent file. Afterward, the output variables are deleted and "gc.collect()" is called to ensure that the RAM be freed. The file counter "file" is updated so the next block of "noptf" optimization processes can start.

structural_simp

```
287     size_list = len(list_ptr2inp)
288     list_ptr2opt += [size_list]
289     # save files
290     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/fid.npy'.format(
291         fid_ini,fid_lim-1,file),np.array(list_fid,dtype=np.uint32))
```

```
292     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/inp.npy'.format(
293         fid_ini,fid_lim-1,file),np.array(list_inp,dtype=np.float32))
294     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/top_p1_opt.npy'.format(
295         fid_ini,fid_lim-1,file),np.array(list_top_p1_opt,dtype=np.float32))
296     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/dtop_opt.npy'.format(
297         fid_ini,fid_lim-1,file),np.packbits(np.array(list_dtop_opt),axis=1))
298     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/obj_opt.npy'.format(
299         fid_ini,fid_lim-1,file),np.array(list_obj_opt,dtype=np.float32))
300     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/dobj_opt.npy'.format(
301         fid_ini,fid_lim-1,file),np.array(list_dobj_opt,dtype=np.float32))
302     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/ptr2opt.npy'.format(
303         fid_ini,fid_lim-1,file),np.array(list_ptr2opt,dtype=np.uint32))
304     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/ptr2inp.npy'.format(
305         fid_ini,fid_lim-1,file),np.array(list_ptr2inp,dtype=np.uint32))
306     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/top_p1.npy'.format(
307         fid_ini,fid_lim-1,file),np.array(list_top_p1,dtype=np.float32))
308     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/dis.npy'.format(
309         fid_ini,fid_lim-1,file),np.array(list_dis,dtype=np.float32))
310     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/sen_p1.npy'.format(
311         fid_ini,fid_lim-1,file),np.array(list_sen_p1,dtype=np.float32))
312     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/obj.npy'.format(
313         fid_ini,fid_lim-1,file),np.array(list_obj,dtype=np.float32))
314     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/gra_p1.npy'.format(
315         fid_ini,fid_lim-1,file),np.array(list_gra_p1,dtype=np.float32))
316     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/vol_p1.npy'.format(
317         fid_ini,fid_lim-1,file),np.array(list_vol_p1,dtype=np.float32))
318     np.save('./output/run_{:06d}_{:06d}/file_{:05d}/tim.npy'.format(
319         fid_ini,fid_lim-1,file),np.array(list_tim,dtype=np.float32))
320     del lists, list_fid, list_inp, list_top_p1_opt, list_dtop_opt, list_obj_opt, list_dobj_opt
321     del list_ptr2opt, list_ptr2inp, list_top_p1, list_dis, list_sen_p1, list_obj, list_gra_p1, list_vol_p1, list_tim
322     gc.collect()
323     # prepare to write next output file
324     file = file + 1
```

When all selected optimization processes are done, the log files are closed and the program terminates.

structural_simp

```
325 iolog.close()
326 tlog.close()
327 print('done!')
```

3.3.4 ./source/python/SIMP/structural_mma.py

This script solves the compliance minimization problem with volume constraint, through the MMA implementation from "nlopt". For a given initial topology and penalization exponent, the "solve_mma" function returns the optimized topology vector and the corresponding objective function.

Firstly, the necessary modules are imported.

structural_mma

```
1 import sys
2 import numpy as np
3 from scipy.sparse import coo_matrix
4 import nlopt
5 sys.path.append('../../cython/')
6 from structural_ssens import str_ssens
7 from structural_filter import str_filter
```

Then, the "solve_mma" function is defined. Its inputs are: the current topology vector, "dens"; the current penalization exponent, "pval"; the soft-kill parameter, "epsk"; the arrays with COO indices corresponding to the unconstrained stiffness matrix, "row" and "col"; the number of elements in each direction, "Nx" and "Ny"; the number of degrees of freedom of the unconstrained system "G"; the vectorized elemental matrix "Kevec"; the stiffness variation matrix, "dKe"; the boolean array that identifies unconstrained degrees of freedom, "freeDofs"; the constrained load vector, "fr"; the sensitivity filter radius, "rmax"; the element size, "esize"; the list of lists, in which the output data is stored, "lists"; the current value of the pointer that relates the iterations in the block of "noptf" optimization processes to the corresponding input, "ptr"; the factor object (obtained through the "sksparse.cholmod.analyze" function) that contains the optimal fill-reducing permutation of the constrained stiffness matrix, "factor" ; the maximal number of evaluations, "meval"; the maximal number of inner iterations "inner_meval"; and a flag variable to specify what the "optimize" method (from "nlopt") should print on screen, "verbose". Default values are defined for "meval", "inner_meval" and "verbose": "meval=0" means that there is no maximal number of evaluations; "inner_meval=0" means that there is no maximal number of inner evaluations; and "verbose=0" means that the "optimize" method (from "nlopt") should not print anything on screen.

The total number of elements in the mesh, "N", is computed; the unconstrained displacements vector is initialized as "ug"; the vectors "alpha_r" and "alpha_f" are initialized to store the sensitivity values before

and after the filtering procedure; the variables "last_x" and "last_h" are initialized to store the topology vector and objective function value from the last iteration thus far. The output lists are obtained from "lists". The "gra_list" is initialized as an empty list, it is used to verify the oscillation of the gray level throughout the iterations, which is one of the stopping criteria.

structural_mma

```
8  def solve_mma(dens, pval, epsk, row, col, Nx, Ny, G, Kevec, dKe, freeDofs, fr, rmax, esize,
9                lists, ptr, factor, meval=0, inner_meval=0, verbose=0):
10     N = Nx*Ny
11     ug = np.zeros(G)
12     alpha_r = np.zeros(N)
13     alpha_f = np.zeros(N)
14     last_x = np.zeros(N)
15     last_h = np.zeros(1)
16     ptr2inp_list = lists[0]
17     top_p1_list  = lists[1]
18     dis_list     = lists[2]
19     sen_p1_list  = lists[3]
20     obj_list     = lists[4]
21     gra_p1_list  = lists[5]
22     vol_p1_list  = lists[6]
23     gra_list     = []
```

The "compliance" function is defined, it computes the objective function value and its derivative with respect to the topology vector. Its inputs are: the topology vector "x"; and the vector in which the gradient of the objective function will be stored, "grad".

The matrix assembly is performed through the "scipy.sparse.coo_matrix" function. The constrained stiffness matrix, "Kr", is stored in CSC format, its Cholesky factorization is performed, and the displacements vector and objective function are computed. The output data is appended to the corresponding lists. The sensitivity analysis is performed and the filtered sensitivity vector is stored in "grad". The cython scripts "structural_ssens.pyx" and "structural_filter.pyx" are detailed in a following section.

If the penalization exponent has reached its maximal value and more than 10 iterations has been performed for this penalization exponent, an early convergence check is performed. If the oscillation of the objective function in the last 10 iterations was less than 0.1%, and if the oscillation of the gray level in the last 10 iterations was less than 0.1%, then an exception is raised and the last topology obtained is considered to be the optimized solution. Otherwise, the "compliance" function simply returns the current objective function value.

structural_mma

```
24     def compliance(x, grad):
25         # compute compliance
26         pen = epsk + (1.0-epsk)*(x**pval)
27         pen = pen.repeat(64)
28         data = pen*np.tile(Kevec,N)
29         Kg_coo = coo_matrix((data,(row,col)),shape=(G,G))
30         Kg_csc = Kg_coo.tocsc()
31         Kr = Kg_csc[freeDofs,:][:,freeDofs]
32         factor.cholesky_inplace(Kr)
33         ug[freeDofs] = factor(fr)
34         hval = np.dot(ug[freeDofs],fr)
35         # store data
36         obj_list.append(hval)
37         gra_list.append(4*np.mean(x*(1.0-x)))
38         x_p1 = x**pval
39         gra_p1_list.append(4*np.mean(x_p1*(1.0-x_p1)))
40         vol_p1_list.append(sum(x_p1)/N)
41         top_p1_list.append(x_p1.copy())
42         ptr2inp_list.append(ptr)
43         dis_list.append(ug.copy())
44         str_ssens(alpha_r, x_p1, dKe, ug, 1.0, Nx, Ny)
45         sen_p1_list.append(alpha_r.copy())
46         # sensitivity analysis
47         if grad.size > 0:
48             str_ssens(alpha_r, x.copy(), dKe, ug, pval, Nx, Ny)
49             str_filter(alpha_r, alpha_f, rmax, esize, Nx, Ny)
50             grad[:] = alpha_f
51         # early convergence
52         if (pval > 5.0) and (len(gra_list) >= 10):
53             last_x[:] = x
54             last_h[0] = hval
55             obj_var = (max(obj_list[-10:])-min(obj_list[-10:]))/min(obj_list[-10:])
56             gray_var = max(gra_list[-10:])-min(gra_list[-10:])
57             if obj_var < 0.001 and gray_var < 0.001:
58                 raise Exception('early convergence')
59         return hval
```

The "volume_const" function is defined, it computes the value of the constraint function and its derivative with respect to the topology vector.

structural_mma
```
60      def volume_const(x, grad):
61          if grad.size > 0:
62              grad[:] = 1.0
63          return sum(x)-0.5*N
```

The "nlopt.opt" method is called. The input "nlopt.LD_MMA" specifies that the MMA will be used as the optimization method, "L" denotes that it is a local optimization and "D" denotes that it is a gradient-based algorithm. The number of variables of the optimization problem is the number of elements in the mesh, "N". The box constraints ($\mathbf{x} \in [0,1]^N$) are set through the commands "set_lower_bounds" and "set_upper_bounds". The command "set_min_objective" is used to set the "compliance" function as the function that will be minimized. The command "add_inequality_constraint" is used to set the "volume_const" function as a constraint of the problem, a tolerance of $1 \times 10^{-6}\, N$ is set to this constraint. Two stopping criteria are included. The command "set_xtol_abs" specifies that the optimization stops if the maximal density variation among the design variables is lesser than 0.01. The command "set_maxeval" specifies that the optimization stops after "meval" iterations are performed. The maximal number of inner iterations and the specifications about what the "optimize" method should print on screen are set through the "set_param" command. The try-except block is used to deal with the early convergence exception. After the optimization is concluded, the "solve_mma" function returns the optimized topology vector, "xopt", and the corresponding objective function value "hopt". The data that has been appended to the output lists is kept in the outer scope.

structural_mma
```
64      opt = nlopt.opt(nlopt.LD_MMA, N)
65      opt.set_lower_bounds(np.zeros(N))
66      opt.set_upper_bounds(np.ones(N))
67      opt.set_min_objective(compliance)
68      opt.add_inequality_constraint(volume_const,1e-6*N)
69      opt.set_xtol_abs(1e-2)
70      opt.set_maxeval(meval)
71      opt.set_param('inner_maxeval', inner_meval)
72      opt.set_param('verbosity', verbose)
73      try:
74          xopt = opt.optimize(dens)
75          hopt = opt.last_optimum_value()
76      except:
77          xopt = last_x
78          hopt = last_h[0]
79      return xopt, hopt
```

3.3.5 ./source/python/generate_str.py

This script concludes the generation of the dataset. It verifies if the generated data is coherent, that is, there is no redundant data and all expected files exist. Then, the data is transferred to the "dataset" folder. Output subfolders are renamed so that a unique number is attributed to each one of them. Duplicate files are deleted, but all folders and logs are preserved.

Firstly, the necessary modules are imported. The boolean flags "conclude_BESO" and "conclude_SIMP" are defined in order to select which dataset should be concluded. The number of optimizations stored in the same file ("noptf") is set to the same value used when generating the data, which is 16.

generate_str
```
1   import os, sys, shutil
2   conclude_BESO = True  # generate BESO dataset
3   conclude_SIMP = True  # generate SIMP dataset
4   noptf         = 16    # number of optimizations stored in the same file
```

If "conclude_BESO" is *True*, the data generated by the program "structural_beso" is verified. The script verifies if: the output folder exists; there is at least one subfolder in it; no duplicated data have been generated; logs subfolders exist; the number of performed optimizations is a multiple of the "noptf" parameter (although this is not necessary, since the generation was performed in groups of exactly "noptf" cases, this is verified to make sure that everything is in order); the number of generated files is coherent; exactly 15 arrays of data have been written in each files subfolder.

generate_str
```
5   # BESO
6   if conclude_BESO:
7       if not os.path.exists('./BESO/output'):
8           print('no output directory for BESO')
9           sys.exit()
10      runs = sorted(os.listdir('./BESO/output'))
11      if len(runs) == 0:
```

```
        print('no runs for BESO')
        sys.exit()
    total_BESO = 0
    r2 = -1
    for k in range(len(runs)):
        run_dir = './BESO/output/' + runs[k] + '/'
        r1 = int(runs[k][4:10])
        if r1 <= r2:
            print('redundant runs in BESO : ' + runs[k])
            sys.exit()
        r2 = int(runs[k][11:])
        rnum = r2 - r1 + 1
        files = sorted(os.listdir(run_dir))
        if files[-1] == 'logs':
            files = files[:-1]
        else:
            print('missing logs directory in BESO : ' + runs[k])
            sys.exit()
        if rnum % noptf != 0:
            print('number of cases is not a multiple of noptf in BESO : ' + runs[k])
            sys.exit()
        if noptf*len(files) != rnum:
            print('incoherent number of files in BESO : ' + runs[k])
            sys.exit()
        total_BESO += rnum
        for kk in range(len(files)):
            file_dir = run_dir + files[kk]
            if len(os.listdir(file_dir)) != 15:
                print('wrong number of files in BESO : ' + runs[k] + '/' + files[kk])
                sys.exit()
    if total_BESO % noptf != 0:
        print('something is wrong...')
        sys.exit()
    total_BESO = total_BESO // noptf
    print('valid BESO dataset!')
    print('{:04d} / {:04d} files in the BESO dataset ({:5.1f} %)\n'.format(
        total_BESO,148240//noptf,total_BESO*noptf*100/148240))
```

If "conclude_SIMP" is *True*, the data generated by the program "structural_simp" is verified.

generate_str

```
# SIMP
if conclude_SIMP:
    if not os.path.exists('./SIMP/output'):
        print('no output directory for SIMP')
        sys.exit()
    runs = sorted(os.listdir('./SIMP/output'))
    if len(runs) == 0:
        print('no runs for SIMP')
        sys.exit()
    total_SIMP = 0
    r2 = -1
    for k in range(len(runs)):
        run_dir = './SIMP/output/' + runs[k] + '/'
        r1 = int(runs[k][4:10])
        if r1 <= r2:
            print('redundant runs in SIMP : ' + runs[k])
            sys.exit()
        r2 = int(runs[k][11:])
        rnum = r2 - r1 + 1
        files = sorted(os.listdir(run_dir))
        if files[-1] == 'logs':
            files = files[:-1]
        else:
            print('missing logs directory in SIMP : ' + runs[k])
            sys.exit()
        if rnum % noptf != 0:
            print('number of cases is not a multiple of noptf in SIMP : ' + runs[k])
            sys.exit()
        if noptf*len(files) != rnum:
            print('incoherent number of files in SIMP : ' + runs[k])
            sys.exit()
        total_SIMP += rnum
        for kk in range(len(files)):
            file_dir = run_dir + files[kk]
            if len(os.listdir(file_dir)) != 15:
                print('wrong number of files in SIMP : ' + runs[k] + '/' + files[kk])
                sys.exit()
    if total_SIMP % noptf != 0:
        print('something is wrong...')
        sys.exit()
    total_SIMP = total_SIMP // noptf
    print('valid SIMP dataset!')
    print("{:04d} / {:04d} files in the SIMP dataset ({:5.1f} %)\n".format(
        total_SIMP,148240//noptf,total_SIMP*noptf*100/148240))
```

Then, the script verifies if there is already a previously generated dataset in the "dataset" folder. In order to avoid undesired overwriting, it is expected that folders be renamed when generating multiple datasets.

generate_str

```
# check directories
if not os.path.exists('../../dataset'):
    os.mkdir('../../dataset')
if conclude_BESO:
    if not os.path.exists('../../dataset/BESO'):
        os.mkdir('../../dataset/BESO')
    else:
        if len(os.listdir('../../dataset/BESO')) > 0:
            print('a BESO dataset has already been generated, rename its directory before generating a new one')
            sys.exit()
if conclude_SIMP:
    if not os.path.exists('../../dataset/SIMP'):
        os.mkdir('../../dataset/SIMP')
    else:
        if len(os.listdir('../../dataset/SIMP')) > 0:
            print('a SIMP dataset has already been generated, rename its directory before generating a new one')
            sys.exit()
```

If everything is in order, the data is transferred to the "dataset" folder. If "conclude_BESO" is *True*, all data generated by the program "structural_beso" is transferred, subfolders are renamed according to the "global_id" variable, so that a unique number is attributed to each files subfolder.

generate_str

```
# BESO
if conclude_BESO:
    global_id = 0
    runs = sorted(os.listdir('./BESO/output'))
    for k in range(len(runs)):
        run_dir = './BESO/output/' + runs[k] + '/'
        files = sorted(os.listdir(run_dir))
        files = files[:-1]
        for kk in range(len(files)):
            if global_id % 100 == 0:
                print(': {:04d} / {:04d} : files have been moved to the BESO dataset ({:5.1f} %)'.format(
                    global_id,total_BESO,global_id*100/total_BESO))
            file_dir = run_dir + files[kk] + '/'
            destination = '../../dataset/BESO/f{:04d}'.format(global_id)
            os.mkdir(destination)
            for file in os.listdir(file_dir):
                source = file_dir + file
                shutil.move(source, destination)
            global_id += 1
    print(': {:04d} / {:04d} : files have been moved to the BESO dataset ({:5.1f} %)'.format(
        global_id,total_BESO,global_id*100/total_BESO))
    print('[ BESO dataset generated ]\n')
```

The same is done for the SIMP dataset. If "conclude_SIMP" is *True*, all data generated by the program "structural_simp" is transferred.

generate_str

```
# SIMP
if conclude_SIMP:
    global_id = 0
    runs = sorted(os.listdir('./SIMP/output'))
    for k in range(len(runs)):
        run_dir = './SIMP/output/' + runs[k] + '/'
        files = sorted(os.listdir(run_dir))
        files = files[:-1]
        for kk in range(len(files)):
            if global_id % 100 == 0:
                print(': {:04d} / {:04d} : files have been moved to the SIMP dataset ({:5.1f} %)'.format(
                    global_id,total_SIMP,global_id*100/total_SIMP))
            file_dir = run_dir + files[kk] + '/'
            destination = '../../dataset/SIMP/f{:04d}'.format(global_id)
            os.mkdir(destination)
            for file in os.listdir(file_dir):
                source = file_dir + file
                shutil.move(source, destination)
            global_id += 1
    print(': {:04d} / {:04d} : files have been moved to the SIMP dataset ({:5.1f} %)'.format(
        global_id,total_SIMP,global_id*100/total_SIMP))
    print('[ SIMP dataset generated ]')
```

3.4 Implementation – Cython

3.4.1 ./source/cython/cython_setup.py

This python script compiles the cython scripts: "structural_bsens.pyx", "structural_ssens.pyx" and "structural_filter.pyx".

cython_setup

```
from setuptools import setup
from Cython.Build import cythonize
setup(
    ext_modules = cythonize(
        ['./structural_bsens.pyx','./structural_ssens.pyx','./structural_filter.pyx'],
        compiler_directives={'language_level' : "3"},
        annotate=False)
)
```

3.4.2 ./source/cython/structural_bsens.pyx

This script performs different sensitivity analyses, which can be used in the BESO algorithm. Firstly, some flags are set and the cython module is imported.

structural_bsens

```
# cython: boundscheck=False
# cython: wraparound=False
# cython: cdivision=True
cimport cython
```

The function "cgs_0_serial" is defined. It performs the CGS-0 sensitivity analysis, given by Equation 19. The function returns nothing, the sensitivity vector "alpha_r" is received as input then edited during execution (the changes are preserved in the outer scope). For each element, its local degrees of freedom are stored in the variable "dofs". These indices are used to slice the global displacements vector "ug", obtaining the local displacements vector "ue". Then, each sensitivity value is computed.

structural_bsens

```
cdef void cgs_0_serial(double [:] alpha_r, double [:,::1] dKe, double [:] ug, long long Nx, long long Ny):
    cdef long long N
    cdef long long e
    cdef long long k1
    cdef long long k2
    cdef double sval
    cdef long long nodes[4]
    cdef long long dofs[8]
    cdef double ue[8]
    cdef double fe[8]
    N = Nx*Ny
    for e in range(N):
        nodes[0] = e + e//Ny
        nodes[1] = nodes[0] + 1 + Ny
        nodes[2] = nodes[1] + 1
        nodes[3] = nodes[0] + 1
        for k1 in range(4):
            dofs[2*k1] = 2*nodes[k1]
            dofs[2*k1+1] = dofs[2*k1] + 1
        for k1 in range(8):
            ue[k1] = ug[dofs[k1]]
        sval = 0.0
        for k1 in range(8):
            fe[k1] = 0.0
            for k2 in range(8):
                fe[k1] = fe[k1] + dKe[k1][k2]*ue[k2]
            sval = sval - ue[k1]*fe[k1]
        alpha_r[e] = sval
    return
```

The function "cgs_1_serial" is defined. It performs the CGS-1 sensitivity analysis, given by Equation 25. The function returns nothing, the sensitivity vector "alpha_r" is received as input then edited during execution (the changes are preserved in the outer scope). For each element, its global degrees of freedom are stored in the variable "dofs". The "local_ids" variable is used to relate "dofs" with the elemental local degrees of freedom. These indices are used to slice the global displacements vector "ug", obtaining the local displacements vector "ue" Then, the first term of the sensitivity value is computed.

The cantilever constraints are indirectly applied (some terms are set as 0, so they are disregarded throughout the operations). The global stiffness matrix is sliced. Afterward, $\mathbf{M_K}$ is obtained by adding/subtracting the stiffness variation matrix to/from it, which is stored in the variable "Kbb". The coefficients $\langle b \rangle_0$ and $\langle b \rangle_1$ are computed and stored in the variables "b0", and "b1". Lastly, the sensitivity value is obtained.

structural_bsens

```
cdef void cgs_1J_serial(double [:] alpha_r, long long [:] dens, double [:] data, long long [:] row_ind, long long [:] col_ptr,
                        long long [:] bc_lim, double [:,::1] dKe, double [:] ug, long long Nx, long long Ny):
    cdef long long N
    cdef long long e
    cdef long long k1
```

```
39      cdef long long k2
40      cdef long long c
41      cdef long long r
42      cdef long long col
43      cdef double b0
44      cdef double b1
45      cdef double sval
46      cdef long long nodes[4]
47      cdef long long dofs[8]
48      cdef long long local_ids[8]
49      cdef double ue[8]
50      cdef double fe[8]
51      cdef double mfe[8]
52      cdef double ffe[8]
53      cdef double Kbb[8][8]
54      N = Nx*Ny
55      for e in range(N):
56          col = e // Ny
57          nodes[0] = e + col
58          nodes[1] = nodes[0] + 1
59          nodes[2] = nodes[0] + 1 + Ny
60          nodes[3] = nodes[2] + 1
61          for k1 in range(4):
62              dofs[2*k1] = 2*nodes[k1]
63              dofs[2*k1+1] = dofs[2*k1] + 1
64          local_ids[0] = 0
65          local_ids[1] = 1
66          local_ids[2] = 6
67          local_ids[3] = 7
68          local_ids[4] = 2
69          local_ids[5] = 3
70          local_ids[6] = 4
71          local_ids[7] = 5
72          for k1 in range(8):
73              ue[local_ids[k1]] = ug[dofs[k1]]
74          sval = 0.0
75          for k1 in range(8):
76              ffe[k1] = 0.0
77              fe[k1] = 0.0
78              for k2 in range(8):
79                  fe[k1] = fe[k1] + dKe[k1][k2]*ue[k2]
80              sval = sval - ue[k1]*fe[k1]
81          # cantilever constraints
82          if col == 0:
83              if (e >= bc_lim[0]) and (e <= bc_lim[1]):
84                  fe[0] = 0.0
85                  fe[1] = 0.0
86              if (e+1 >= bc_lim[0]) and (e+1 <= bc_lim[1]):
87                  fe[6] = 0.0
88                  fe[7] = 0.0
89          for c in range(8):
90              r = c
91              for k1 in range(col_ptr[dofs[c]]+c,col_ptr[dofs[c]+1]):
92                  if row_ind[k1] == dofs[r]:
93                      Kbb[local_ids[r]][local_ids[c]] = data[k1]
94                      Kbb[local_ids[c]][local_ids[r]] = data[k1]
95                      if r == 7:
96                          break
97                      else:
98                          r = r + 1
99          b0 = 0.0
100         b1 = 0.0
101         if dens[e] == 0:
102             for k1 in range(8):
103                 Kbb[k1][k1] = Kbb[k1][k1] + dKe[k1][k1]
104                 mfe[k1] = fe[k1]/Kbb[k1][k1]
105                 for k2 in range(k1+1,8):
106                     Kbb[k1][k2] = Kbb[k1][k2] + dKe[k1][k2]
107                     Kbb[k2][k1] = Kbb[k1][k2]
108                 b0 = b0 + fe[k1]*mfe[k1]
109             for k1 in range(8):
110                 ffe[k1] = ffe[k1] + Kbb[k1][k1]*mfe[k1]
111                 for k2 in range(k1+1,8):
112                     ffe[k1] = ffe[k1] + Kbb[k1][k2]*mfe[k2]
113                     ffe[k2] = ffe[k2] + Kbb[k2][k1]*mfe[k1]
114                 b1 = b1 + mfe[k1]*ffe[k1]
115             sval = sval + (b0*b0)/b1
116         else:
117             for k1 in range(8):
118                 Kbb[k1][k1] = Kbb[k1][k1] - dKe[k1][k1]
119                 mfe[k1] = fe[k1]/Kbb[k1][k1]
120                 for k2 in range(k1+1,8):
121                     Kbb[k1][k2] = Kbb[k1][k2] - dKe[k1][k2]
122                     Kbb[k2][k1] = Kbb[k1][k2]
123                 b0 = b0 + fe[k1]*mfe[k1]
124             for k1 in range(8):
125                 ffe[k1] = ffe[k1] + Kbb[k1][k1]*mfe[k1]
126                 for k2 in range(k1+1,8):
127                     ffe[k1] = ffe[k1] + Kbb[k1][k2]*mfe[k2]
128                     ffe[k2] = ffe[k2] + Kbb[k2][k1]*mfe[k1]
129                 b1 = b1 + mfe[k1]*ffe[k1]
```

```
            sval = sval - (b0*b0)/b1
        alpha_r[e] = sval
    return
```

The function "cgs_2_serial" is defined. It performs the CGS-1 sensitivity analysis, given by Equation 28. The function returns nothing, the sensitivity vector "alpha_r" is received as input then edited during execution (the changes are preserved in the outer scope). For each element, the local matrices and vectors of all its immediate neighbours are used to perform this sensitivity analysis. Thus, slices are performed in the 3×3 sub-grid around each element. For each sub-grid, its global degrees of freedom are stored in the variable "dofs". The "local_ids" variable is used to relate "dofs" with the elemental local degrees of freedom. These indices are used to slice the global displacements vector "ug", obtaining the local displacements vector "ue" Then, the first term of the sensitivity value is computed.

The cantilever constraints are indirectly applied (some terms are set as 0, so they are disregarded throughout the operations). The global stiffness matrix is sliced. Afterward, $\mathbf{M_K}$ is obtained by adding/subtracting the stiffness variation matrix to/from it, which is stored in the variable "Kbb". The coefficients $\langle b \rangle_0$, $\langle b \rangle_1$, $\langle b \rangle_2$ and $\langle b \rangle_3$ are computed and stored in the variables "b0", "b1", "b2" and "b3". Lastly, the sensitivity value is obtained.

structural_bsens

```
cdef void cgs_2J_serial(double [:] alpha_r, long long [:] dens, double [:] data, long long [:] row_ind, long long [:] col_ptr,
                        long long [:] bc_lim, double [:,::1] dKe, double [:] ug, long long Nx, long long Ny):
    cdef long long N
    cdef long long e
    cdef long long k1
    cdef long long k2
    cdef long long c
    cdef long long r
    cdef long long row
    cdef long long col
    cdef double b0
    cdef double b1
    cdef double b2
    cdef double b3
    cdef double sval
    cdef long long nodes[16]
    cdef long long dofs[32]
    cdef long long local_ids[32]
    cdef double ue[8]
    cdef double fe[8]
    cdef double mfe[8]
    cdef double ffe[32]
    cdef double mffe[32]
    cdef double fffe[32]
    cdef double Kbb[32][32]
    N = Nx*Ny
    for e in range(N):
        row = e % Ny
        col = e // Ny
        nodes[5]  = e + col
        nodes[0]  = nodes[5] - 2 - Ny
        nodes[1]  = nodes[5] - 1 - Ny
        nodes[2]  = nodes[5] + 0 - Ny
        nodes[3]  = nodes[5] + 1 - Ny
        nodes[4]  = nodes[5] - 1
        nodes[6]  = nodes[5] + 1
        nodes[7]  = nodes[5] + 2
        nodes[8]  = nodes[5] + 0 + Ny
        nodes[9]  = nodes[5] + 1 + Ny
        nodes[10] = nodes[5] + 2 + Ny
        nodes[11] = nodes[5] + 3 + Ny
        nodes[12] = nodes[5] + 1 + Ny*2
        nodes[13] = nodes[5] + 2 + Ny*2
        nodes[14] = nodes[5] + 3 + Ny*2
        nodes[15] = nodes[5] + 4 + Ny*2
        for k1 in range(16):
            dofs[2*k1] = 2*nodes[k1]
            dofs[2*k1+1] = dofs[2*k1] + 1
        if row == 0:
            for k1 in range(4):
                dofs[8*k1]   = -1
                dofs[8*k1+1] = -1
        elif row == Ny-1:
            for k1 in range(4):
                dofs[8*k1+6] = -1
                dofs[8*k1+7] = -1
        if col == 0:
            for k1 in range(4):
                dofs[2*k1]   = -1
                dofs[2*k1+1] = -1
        elif col == Nx-1:
            for k1 in range(4):
                dofs[2*k1+24] = -1
```

```
                dofs[2*k1+25] = -1
        for k1 in range(10):
            local_ids[k1] = k1+8
        local_ids[10] = 0
        local_ids[11] = 1
        local_ids[12] = 6
        local_ids[13] = 7
        for k1 in range(14,18):
            local_ids[k1] = k1+4
        local_ids[18] = 2
        local_ids[19] = 3
        local_ids[20] = 4
        local_ids[21] = 5
        for k1 in range(22,32):
            local_ids[k1] = k1
        for k1 in range(10,14):
            ue[local_ids[k1]] = ug[dofs[k1]]
            ue[local_ids[k1+8]] = ug[dofs[k1+8]]
        sval = 0.0
        for k1 in range(8):
            fe[k1] = 0.0
            for k2 in range(8):
                fe[k1] = fe[k1] + dKe[k1][k2]*ue[k2]
            sval = sval - ue[k1]*fe[k1]
        # cantilever constraints
        if col == 0:
            if (e >= bc_lim[0]) and (e <= bc_lim[1]):
                fe[0] = 0.0
                fe[1] = 0.0
            if (e+1 >= bc_lim[0]) and (e+1 <= bc_lim[1]):
                fe[6] = 0.0
                fe[7] = 0.0
        for c in range(32):
            Kbb[c][c] = 0.0
            for r in range(c+1,32):
                Kbb[r][c] = 0.0
                Kbb[c][r] = 0.0
        for c in range(32):
            if dofs[c] != -1:
                for r in range(32):
                    for k1 in range(col_ptr[dofs[c]],col_ptr[dofs[c]+1]):
                        if row_ind[k1] == dofs[r]:
                            Kbb[local_ids[r]][local_ids[c]] = data[k1]
                            Kbb[local_ids[c]][local_ids[r]] = data[k1]
        b0 = 0.0
        b1 = 0.0
        b2 = 0.0
        b3 = 0.0
        if dens[e] == 0:
            for k1 in range(8):
                Kbb[k1][k1] = Kbb[k1][k1] + dKe[k1][k1]
                mfe[k1] = fe[k1]/Kbb[k1][k1]
                for k2 in range(k1+1,8):
                    Kbb[k1][k2] = Kbb[k1][k2] + dKe[k1][k2]
                    Kbb[k2][k1] = Kbb[k1][k2]
                b0 = b0 + fe[k1]*mfe[k1]
            for k1 in range(32):
                ffe[k1] = 0.0
                for k2 in range(8):
                    ffe[k1] = ffe[k1] + Kbb[k1][k2]*mfe[k2]
            for k1 in range(8):
                b1 = b1 + mfe[k1]*ffe[k1]
            # cantilever constraints
            if col == 0:
                if (row-1 >= bc_lim[0]) and (row-1 <= bc_lim[1]):
                    ffe[16] = 0.0
                    ffe[17] = 0.0
                if (row >= bc_lim[0]) and (row <= bc_lim[1]):
                    ffe[0] = 0.0
                    ffe[1] = 0.0
                if (row+1 >= bc_lim[0]) and (row+1 <= bc_lim[1]):
                    ffe[6] = 0.0
                    ffe[7] = 0.0
                if (row+2 >= bc_lim[0]) and (row+2 <= bc_lim[1]):
                    ffe[18] = 0.0
                    ffe[19] = 0.0
            elif col == 1:
                if (row-1 >= bc_lim[0]) and (row-1 <= bc_lim[1]):
                    ffe[8] = 0.0
                    ffe[9] = 0.0
                if (row >= bc_lim[0]) and (row <= bc_lim[1]):
                    ffe[10] = 0.0
                    ffe[11] = 0.0
                if (row+1 >= bc_lim[0]) and (row+1 <= bc_lim[1]):
                    ffe[12] = 0.0
                    ffe[13] = 0.0
                if (row+2 >= bc_lim[0]) and (row+2 <= bc_lim[1]):
                    ffe[14] = 0.0
                    ffe[15] = 0.0
            for r in range(32):
                k1 = local_ids[r]
```

```
                if dofs[r] != -1:
                    mffe[k1] = ffe[k1]/Kbb[k1][k1]
                    b2 = b2 + ffe[k1]*mffe[k1]
                else:
                    mffe[k1] = 0.0
            for k1 in range(32):
                fffe[k1] = 0.0
                for k2 in range(32):
                    fffe[k1] = fffe[k1] + Kbb[k1][k2]*mffe[k2]
                b3 = b3 + mffe[k1]*fffe[k1]
            sval = sval + (b0*b0*b3 - 2*b0*b1*b2 + b1*b1*b1)/(b1*b3-b2*b2)
        else:
            for k1 in range(8):
                Kbb[k1][k1] = Kbb[k1][k1] - dKe[k1][k1]
                mfe[k1] = fe[k1]/Kbb[k1][k1]
                for k2 in range(k1+1,8):
                    Kbb[k1][k2] = Kbb[k1][k2] - dKe[k1][k2]
                    Kbb[k2][k1] = Kbb[k1][k2]
                b0 = b0 + fe[k1]*mfe[k1]
            for k1 in range(32):
                ffe[k1] = 0.0
                for k2 in range(8):
                    ffe[k1] = ffe[k1] + Kbb[k1][k2]*mfe[k2]
            for k1 in range(8):
                b1 = b1 + mfe[k1]*ffe[k1]
            # cantilever constraints
            if col == 0:
                if (row-1 >= bc_lim[0]) and (row-1 <= bc_lim[1]):
                    ffe[16] = 0.0
                    ffe[17] = 0.0
                if (row >= bc_lim[0]) and (row <= bc_lim[1]):
                    ffe[0] = 0.0
                    ffe[1] = 0.0
                if (row+1 >= bc_lim[0]) and (row+1 <= bc_lim[1]):
                    ffe[6] = 0.0
                    ffe[7] = 0.0
                if (row+2 >= bc_lim[0]) and (row+2 <= bc_lim[1]):
                    ffe[18] = 0.0
                    ffe[19] = 0.0
            elif col == 1:
                if (row-1 >= bc_lim[0]) and (row-1 <= bc_lim[1]):
                    ffe[8] = 0.0
                    ffe[9] = 0.0
                if (row >= bc_lim[0]) and (row <= bc_lim[1]):
                    ffe[10] = 0.0
                    ffe[11] = 0.0
                if (row+1 >= bc_lim[0]) and (row+1 <= bc_lim[1]):
                    ffe[12] = 0.0
                    ffe[13] = 0.0
                if (row+2 >= bc_lim[0]) and (row+2 <= bc_lim[1]):
                    ffe[14] = 0.0
                    ffe[15] = 0.0
            for r in range(32):
                k1 = local_ids[r]
                if dofs[r] != -1:
                    mffe[k1] = ffe[k1]/Kbb[k1][k1]
                    b2 = b2 + ffe[k1]*mffe[k1]
                else:
                    mffe[k1] = 0.0
            for k1 in range(32):
                fffe[k1] = 0.0
                for k2 in range(32):
                    fffe[k1] = fffe[k1] + Kbb[k1][k2]*mffe[k2]
                b3 = b3 + mffe[k1]*fffe[k1]
            sval = sval - (b0*b0*b3 - 2*b0*b1*b2 + b1*b1*b1)/(b1*b3-b2*b2)
        alpha_r[e] = sval
    return
```

The "str_cgs" function is defined to call the cython functions in the python program. It receives as input: the vector to which the sensitivity values will be assigned, "alpha_r" ; the topology vector, "x"; the unconstrained global stiffness matrix in CSC format, "Kg_csc"; the array that stores the indices of the first and the last clamped nodes, "bc_lim"; the stiffness variation matrix, "dKe"; the unconstrained global displacements vector, "ug"; the number of elements in each direction, "Nx" and "Ny"; and the parameter "steps", used to select which sensitivity analysis should be performed.

structural_bsens

```
def str_cgs(alpha_r, x, Kg_csc, bc_lim, dKe, ug, Nx, Ny, steps=0):
    if steps == 0:
        cgs_0_serial(alpha_r, dKe, ug, Nx, Ny)
    else:
        dens = x.astype("int64")
        data = Kg_csc.data
        row_ind = Kg_csc.indices.astype("int64")
        col_ptr = Kg_csc.indptr.astype("int64")
        if steps == 1:
                    cgs_1J_serial(alpha_r, dens, data, row_ind, col_ptr, bc_lim, dKe, ug, Nx, Ny)
```

```
364         else:
365             cgs_2J_serial(alpha_r, dens, data, row_ind, col_ptr, bc_lim, dKe, ug, Nx, Ny)
366     return
```

3.4.3 ./source/cython/structural_ssens.pyx

This script performs the sensitivity analysis to be used in the MMA optimization procedure. Firstly, some flags are set and the cython module is imported.

structural_ssens

```
1  # cython: boundscheck=False
2  # cython: wraparound=False
3  # cython: cdivision=True
4  cimport cython
```

The function "sens_serial" is defined. It computes the derivative of $C(\mathbf{x})$, given by Equation 36. The function returns nothing, the sensitivity vector "alpha_r" is received as input then edited during execution (the changes are preserved in the outer scope). For each element, its local degrees of freedom are stored in the variable "dofs". These indices are used to slice the global displacements vector "ug", obtaining the local displacements vector "ue". Then, each sensitivity value is computed.

structural_ssens

```
5  cdef void sens_serial(double [:] alpha_r, double [:] dens, double [:,::1] dKe, double [:] ug,
6                        double p, long long Nx, long long Ny):
7      cdef long long N
8      cdef long long e
9      cdef long long k1
10     cdef long long k2
11     cdef double sval
12     cdef long long nodes[4]
13     cdef long long dofs[8]
14     cdef double ue[8]
15     cdef double fe[8]
16     N = Nx*Ny
17     for e in range(N):
18         nodes[0] = e + e//Ny
19         nodes[1] = nodes[0] + 1 + Ny
20         nodes[2] = nodes[1] + 1
21         nodes[3] = nodes[0] + 1
22         for k1 in range(4):
23             dofs[2*k1] = 2*nodes[k1]
24             dofs[2*k1+1] = dofs[2*k1] + 1
25         for k1 in range(8):
26             ue[k1] = ug[dofs[k1]]
27         sval = 0.0
28         for k1 in range(8):
29             fe[k1] = 0.0
30             for k2 in range(8):
31                 fe[k1] = fe[k1] + dKe[k1][k2]*ue[k2]
32             sval = sval - ue[k1]*fe[k1]
33         if p > 1 + 1e-9:
34             sval = sval * p * (dens[e]**(p-1.0))
35         alpha_r[e] = sval
36     return
```

The "str_ssens" function is defined to call the cython function in the python program. It receives as input: the vector to which the sensitivity values will be assigned, "alpha_r"; the topology vector, "dens"; the stiffness variation matrix, "dKe"; the unconstrained global displacements vector, "ug"; the SIMP exponent "p"; and the number of elements in each direction, "Nx" and "Ny".

structural_ssens

```
37 def str_ssens(alpha_r, dens, dKe, ug, p, Nx, Ny):
38     sens_serial(alpha_r, dens, dKe, ug, p, Nx, Ny)
39     return
```

3.4.4 ./source/cython/structural_filter.pyx

This script perform the filtering procedure to smooth the sensitivity map for both SILP-BESO and SIMP-MMA approaches. Firstly, some flags are set and the necessary modules are imported.

structural_filter

```
1  # cython: boundscheck=False
2  # cython: wraparound=False
3  # cython: cdivision=True
4  cimport cython
5  from libc.math cimport sqrt
6  from libc.stdlib cimport malloc, free
```

The function "sfilter_serial" is defined. It performs the conical filtering procedure, given by Equation 30. The function returns nothing, the filtered sensitivity vector "alpha_f" is received as input then edited during execution (the changes are preserved in the outer scope). If the filter radius is smaller than the side length of the square element, the filtered sensitivity vector is equal to the original vector "alpha_r". Otherwise, the conical filter weights are computed and the filtering procedure is performed.

structural_filter

```
cdef void sfilter_serial(double [:] alpha_r, double [:] alpha_f, double rm, long long Nx, long long Ny):
    cdef long long N
    cdef long long e
    N = Nx*Ny
    if rm <= 1.0:
        for e in range(N):
            alpha_f[e] = alpha_r[e]
        return
    cdef long long k1
    cdef long long k2
    cdef long long row
    cdef long long col
    cdef long long center
    cdef long long Rmax
    cdef long long Rbot
    cdef long long Rtop
    cdef long long Rlef
    cdef long long Rrig
    cdef double rmax_sq
    cdef double dist_sq
    cdef double weights_sum
    Rmax = <long long> rm
    center = 2*Rmax*(Rmax+1)
    rmax_sq = rm*rm
    cdef double *weights  = <double *> malloc((2*Rmax+1)*(2*Rmax+1)*sizeof(double))
    cdef double *weights0 = <double *> malloc((2*Rmax+1)*(2*Rmax+1)*sizeof(double))
    # conical filter weights
    weights_sum = 0.0
    for k1 in range(-Rmax,Rmax+1):
        for k2 in range(-Rmax,Rmax+1):
            dist_sq = k1*k1 + k2*k2
            if dist_sq > rmax_sq:
                weights[center+k1+k2*(2*Rmax+1)] = 0.0
            else:
                weights[center+k1+k2*(2*Rmax+1)] = rm - sqrt(dist_sq)
                weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
    for k1 in range((2*Rmax+1)*(2*Rmax+1)):
        weights0[k1] = weights[k1]/weights_sum
    for e in range(N):
        # submesh under the filter's influence
        row = e % Ny
        col = e // Ny
        Rbot = Rmax
        Rtop = Rmax
        Rlef = Rmax
        Rrig = Rmax
        if row < Rmax:
            Rbot = row
        if row > Ny-1-Rmax:
            Rtop = Ny-1-row
        if col < Rmax:
            Rlef = col
        if col > Nx-1-Rmax:
            Rrig = Nx-1-col
        # filtered sensitivity vector
        alpha_f[e] = 0.0
        if Rbot+Rtop+Rlef+Rrig < 4*Rmax:
            weights_sum = 0.0
            for k1 in range(-Rbot,Rtop+1):
                for k2 in range(-Rlef,Rrig+1):
                    weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
            for k1 in range(-Rbot,Rtop+1):
                for k2 in range(-Rlef,Rrig+1):
                    alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*(weights[center+k1+k2*(2*Rmax+1)]/weights_sum)
        else:
            for k1 in range(-Rbot,Rtop+1):
                for k2 in range(-Rlef,Rrig+1):
                    alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*weights0[center+k1+k2*(2*Rmax+1)]
    free(weights)
    free(weights0)
    return
```

The function "sfilter_noload_serial" is defined. It performs the conical filtering procedure, given by Equation 30. However, this function ignores directly loaded elements (all elements of the right extremity of the design domain which intersect the loaded region). The function returns nothing, the filtered sensitivity vector "alpha_f" is received as input then edited during execution (the changes are preserved in the outer scope). If the filter radius is smaller than the side length of the square element, the filtered sensitivity vector is

equal to the original vector "alpha_r" (the filtered sensitivity values of the loaded elements are set to 0). Otherwise, the conical filter weights are computed and the filtering procedure is performed.

structural_filter

```
78  cdef void sfilter_noload_serial(double [:] alpha_r, double [:] alpha_f, double rm,
79                                  long long Nx, long long Ny, long long [:] ld_lim):
80      cdef long long N
81      cdef long long e
82      cdef long long v0
83      cdef long long v1
84      N = Nx*Ny
85      v0 = ld_lim[0]
86      v1 = ld_lim[1]
87      if rm <= 1.0:
88          for e in range(N):
89              if e >= v0 and e <= v1:
90                  alpha_f[e] = 0.0
91              else:
92                  alpha_f[e] = alpha_r[e]
93          return
94      cdef long long k1
95      cdef long long k2
96      cdef long long k3
97      cdef long long ld_lim_0
98      cdef long long ld_lim_1
99      cdef long long row
100     cdef long long col
101     cdef long long center
102     cdef long long Rmax
103     cdef long long Rbot
104     cdef long long Rtop
105     cdef long long Rlef
106     cdef long long Rrig
107     cdef double rmax_sq
108     cdef double dist_sq
109     cdef double weights_sum
110     Rmax = <long long> rm
111     center = 2*Rmax*(Rmax+1)
112     rmax_sq = rm*rm
113     cdef double *weights  = <double *> malloc((2*Rmax+1)*(2*Rmax+1)*sizeof(double))
114     cdef double *weights0 = <double *> malloc((2*Rmax+1)*(2*Rmax+1)*sizeof(double))
115     # conical filter weights
116     weights_sum = 0.0
117     for k1 in range(-Rmax,Rmax+1):
118         for k2 in range(-Rmax,Rmax+1):
119             dist_sq = k1*k1 + k2*k2
120             if dist_sq > rmax_sq:
121                 weights[center+k1+k2*(2*Rmax+1)] = 0.0
122             else:
123                 weights[center+k1+k2*(2*Rmax+1)] = rm - sqrt(dist_sq)
124                 weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
125     for k1 in range((2*Rmax+1)*(2*Rmax+1)):
126         weights0[k1] = weights[k1]/weights_sum
127     ld_lim_0 = v0 % Ny
128     ld_lim_1 = v1 % Ny
129     for e in range(N):
130         if e >= v0 and e <= v1:
131             alpha_f[e] = 0.0
132         else:
133             # submesh under the filter's influence
134             row = e % Ny
135             col = e // Ny
136             Rbot = Rmax
137             Rtop = Rmax
138             Rlef = Rmax
139             Rrig = Rmax
140             if row < Rmax:
141                 Rbot = row
142             if row > Ny-1-Rmax:
143                 Rtop = Ny-1-row
144             if col < Rmax:
145                 Rlef = col
146             if col > Nx-1-Rmax:
147                 Rrig = Nx-1-col
148             # filtered sensitivity vector
149             alpha_f[e] = 0.0
150             if col >= Nx-1-Rmax:  # ignore loaded elements
151                 weights_sum = 0.0
152                 for k1 in range(-Rbot,Rtop+1):
153                     for k2 in range(-Rlef,Rrig):
154                         weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
155                 k2 = Rrig
156                 if (row+Rtop < ld_lim_0) or (row-Rbot > ld_lim_1):
157                     for k1 in range(-Rbot,Rtop+1):
158                         weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
159                 else:
160                     if row-Rbot < ld_lim_0:
161                         for k3 in range(row-Rbot,ld_lim_0):
162                             k1 = k3 - row
163                             weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
```

```
                    if row+Rtop > ld_lim_1:
                        for k3 in range(ld_lim_1+1,row+Rtop+1):
                            k1 = k3 - row
                            weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
                for k1 in range(-Rbot,Rtop+1):
                    for k2 in range(-Rlef,Rrig):
                        alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*(weights[center+k1+k2*(2*Rmax+1)]/weights_sum)
                k2 = Rrig
                if (row+Rtop < ld_lim_0) or (row-Rbot > ld_lim_1):
                    for k1 in range(-Rbot,Rtop+1):
                        alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*(weights[center+k1+k2*(2*Rmax+1)]/weights_sum)
                else:
                    if row-Rbot < ld_lim_0:
                        for k3 in range(row-Rbot,ld_lim_0):
                            k1 = k3 - row
                            alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*(weights[center+k1+k2*(2*Rmax+1)]/weights_sum)
                    if row+Rtop > ld_lim_1:
                        for k3 in range(ld_lim_1+1,row+Rtop+1):
                            k1 = k3 - row
                            alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*(weights[center+k1+k2*(2*Rmax+1)]/weights_sum)
            else:                 # no loaded elements to ignore
                if Rbot+Rtop+Rlef+Rrig < 4*Rmax:
                    weights_sum = 0.0
                    for k1 in range(-Rbot,Rtop+1):
                        for k2 in range(-Rlef,Rrig+1):
                            weights_sum = weights_sum + weights[center+k1+k2*(2*Rmax+1)]
                    for k1 in range(-Rbot,Rtop+1):
                        for k2 in range(-Rlef,Rrig+1):
                            alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*(weights[center+k1+k2*(2*Rmax+1)]/weights_sum)
                else:
                    for k1 in range(-Rbot,Rtop+1):
                        for k2 in range(-Rlef,Rrig+1):
                            alpha_f[e] = alpha_f[e] + alpha_r[e+k1+k2*Ny]*weights0[center+k1+k2*(2*Rmax+1)]
    free(weights)
    free(weights0)
    return
```

The "str_filter" function is defined to call the cython functions in the python program. It receives as input: the raw sensitivity vector, "alpha_r"; the vector to which the filtered sensitivity vector will be assigned, "alpha_f"; the conical filter radius, "rmax"; the side length of the square element, "esize"; the number of elements in each direction, "Nx" and "Ny"; and "load_lim", which stores the indices of the first and last elements that are connected to the loaded region. This last input is optional, it is used to select if loaded elements will or will not be taken into account in the filtering procedure. In the SILP-BESO approach, loaded elements should be ignored, so "load_lim" should be informed in order to call "sfilter_noload_serial". In the SIMP-MMA approach, loaded elements should not be ignored, so "load_lim" should be *None* in order to call "sfilter_serial".

structural_filter

```
def str_filter(alpha_r, alpha_f, rmax, esize, Nx, Ny, load_lim=None):
    rm = rmax/esize
    if load_lim is None:
        sfilter_serial(alpha_r, alpha_f, rm, Nx, Ny)
    else:
        sfilter_noload_serial(alpha_r, alpha_f, rm, Nx, Ny, load_lim)
    return
```

3.5 Implementation – Sampling

3.5.1 ./sample/BESO/bsample.py

This script generates figures for a selected sample of the BESO dataset. For a given selection of files, png images for the topologies, sensitivity maps and displacements fields can be generated and saved in subfolders of the "sample/BESO" directory. Plots of the objective function and the volume of material throughout a given optimization can also be generated and saved as png images.

Firstly, the necessary modules are imported.

bsample

```
import os, sys
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.collections as clct
```

Then, the user selects which files should be read through the parameters "file_ini", which specifies the index of the first file to be read, and "file_lim", which specifies the file index limit (non inclusive). Boolean flags are used to specify which figures should be generated: if "fig_top_opt" is *True*, figures are generated for all optimized topologies; if "fig_top_sen" is *True*, figures are generated for all topologies (not only the optimized)

and sensitivity vectors; if “fig_dis” is *True*, figures are generated for all displacements vectors; and if “fig_obj_vol” is *True*, figures are generated with all the plots of objective function and volume.

In order to avoid generating an unreasonable number of figures, users should be careful when defining these parameters. Each file contains the results of “noptf” optimizations, and each optimization produces dozens (83 in average) of topology, sensitivity and displacements vectors.

bsample

```
file_ini    = 0     # initial file index |from file 0
file_lim    = 4     # file index limit   |up to file 9264
fig_top_opt = True  # optimized topology
fig_top_sen = True  # topology vectors and sensitivity vectors
fig_dis     = True  # displacements vectors
fig_obj_vol = True  # objective function and volume
```

Some properties of the cantilever are defined, the program checks if the “dataset” folder exists, then the folders in which the figures will be saved are created.

bsample

```
# fixed properties
Ly = 1.0       # cantilever height
small = 1e-14  # small value to compare float numbers
Ny = 32        # number of elements in y-axis
Nx = 2*Ny      # number of elements in x-axis
N = Nx*Ny      # total number of elements
esize = Ly/Ny  # element size
# check directories
rpath = '../../dataset/BESO/'
if not os.path.exists(rpath):
    print('missing BESO dataset')
    sys.exit()
if not os.path.exists('./top_opt'):
    os.mkdir('./top_opt')
if not os.path.exists('./top_sen'):
    os.mkdir('./top_sen')
if not os.path.exists('./dis'):
    os.mkdir('./dis')
if not os.path.exists('./obj_vol'):
    os.mkdir('./obj_vol')
```

The loop through the selected files is started. The input files indices, the input data and the pointers relating each input with the corresponding iterations of the optimization processes are read from the “fid.npy”, “inp.npy” and “ptr2opt.npy” files.

bsample

```
file = file_ini
while (file < file_lim) and (os.path.exists(rpath + 'f{:04d}'.format(file))):
    #%% Read files
    print('> reading files of f{:04d}'.format(file))
    # input files id, input data and pointers to optimization
    if not os.path.exists(rpath + 'f{:04d}/fid.npy'.format(file)):
        print('missing : f{:04d}/fid.npy'.format(file))
        sys.exit()
    list_fid = np.load(rpath + 'f{:04d}/fid.npy'.format(file))
    if not os.path.exists(rpath + 'f{:04d}/inp.npy'.format(file)):
        print('missing : f{:04d}/inp.npy'.format(file))
        sys.exit()
    list_inp = np.load(rpath + 'f{:04d}/inp.npy'.format(file))
    if not os.path.exists(rpath + 'f{:04d}/ptr2opt.npy'.format(file)):
        print('missing : f{:04d}/ptr2opt.npy'.format(file))
        sys.exit()
    list_ptr2opt = np.load(rpath + 'f{:04d}/ptr2opt.npy'.format(file))
```

According to the boolean flags, the optimized topologies, topologies, sensitivity vectors, displacements vectors, objective function values and volume values are read from the “top_opt.npy”, “top.npy”, “sen_0.npy”, “sen_1.npy”, “sen_2.npy”, “sen_w.npy”, “dis.npy”, “obj.npy” and “vol.npy” files.

bsample

```
    # optimized topology
    if fig_top_opt:
        if not os.path.exists(rpath + 'f{:04d}/top_opt.npy'.format(file)):
            print('missing : f{:04d}/top_opt.npy'.format(file))
            sys.exit()
        list_top_opt = np.load(rpath + 'f{:04d}/top_opt.npy'.format(file))
    # topology vectors
    if fig_top_sen or fig_dis:
        if not os.path.exists(rpath + 'f{:04d}/top.npy'.format(file)):
            print('missing : f{:04d}/top.npy'.format(file))
            sys.exit()
```

```
        list_top = np.load(rpath + 'f{:04d}/top.npy'.format(file))
    # sensitivity vectors
    if fig_top_sen:
        if not os.path.exists(rpath + 'f{:04d}/sen_0.npy'.format(file)):
            print('missing : f{:04d}/sen_0.npy'.format(file))
            sys.exit()
        list_sen_0 = np.load(rpath + 'f{:04d}/sen_0.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/sen_1.npy'.format(file)):
            print('missing : f{:04d}/sen_1.npy'.format(file))
            sys.exit()
        list_sen_1 = np.load(rpath + 'f{:04d}/sen_1.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/sen_2.npy'.format(file)):
            print('missing : f{:04d}/sen_2.npy'.format(file))
            sys.exit()
        list_sen_2 = np.load(rpath + 'f{:04d}/sen_2.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/sen_w.npy'.format(file)):
            print('missing : f{:04d}/sen_w.npy'.format(file))
            sys.exit()
        list_sen_w = np.load(rpath + 'f{:04d}/sen_w.npy'.format(file))
    # displacements vectors
    if fig_dis:
        if not os.path.exists(rpath + 'f{:04d}/dis.npy'.format(file)):
            print('missing : f{:04d}/dis.npy'.format(file))
            sys.exit()
        list_dis = np.load(rpath + 'f{:04d}/dis.npy'.format(file))
    # objective function and volume
    if fig_obj_vol:
        if not os.path.exists(rpath + 'f{:04d}/obj.npy'.format(file)):
            print('missing : f{:04d}/obj.npy'.format(file))
            sys.exit()
        list_obj = np.load(rpath + 'f{:04d}/obj.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/vol.npy'.format(file)):
            print('missing : f{:04d}/vol.npy'.format(file))
            sys.exit()
        list_vol = np.load(rpath + 'f{:04d}/vol.npy'.format(file))
```

If "fig_top_opt" is *True*, images of the optimized topologies are saved. The optimized topologies are represented as 66×32 grayscale images. The first and last columns are used to represent the clamped and loaded regions.

bsample

```
    #%% Generate figures
    print(': generating figures')
    # optimized topology
    if fig_top_opt:
        print(': : optimized topology...')
        for k in range(len(list_fid)):
            # boundary conditions
            fid = list_fid[k]
            inp = list_inp[k]
            ycoor = esize*np.array(list(range(Ny+1)))-0.5*Ly
            mask = (ycoor > inp[0]-inp[1]-small) & (ycoor < inp[0]+inp[1]+small)
            c0 = np.zeros((32,1))
            c0[mask[1:]]  += 0.25
            c0[mask[:-1]] += 0.25
            mask = (ycoor > inp[2]-inp[3]-small) & (ycoor < inp[2]+inp[3]+small)
            c1 = np.zeros((32,1))
            c1[mask[1:]]  += 0.25
            c1[mask[:-1]] += 0.25
            # figure
            plt.figure(num=0).clear()
            fig,ax = plt.subplots(num=0)
            x = list_top_opt[k]
            x = np.unpackbits(x,axis=None).astype(float)
            xmat = np.reshape(x,(Ny,Nx),order='F')
            xmat = np.concatenate((c0,xmat,c1),axis=1)
            ax.imshow(xmat,cmap='gray_r',vmin=0,vmax=1.0,origin='lower')
            ax.axis('off')
            fig.set_size_inches(8, 4)
            plt.savefig('./top_opt/f{:06d}.png'.format(fid),bbox_inches='tight',pad_inches=0.05,dpi=100)
```

If "fig_top_sen" is *True*, images of the topology and sensitivity vectors are saved. The topologies are represented as 66×32 grayscale images. The first and last columns are used to represent the clamped and loaded regions. Each sensitivity map is represented as 64×32 single-channeled image, the "jet" colormap is used instead of grayscale. All sensitivity maps are normalized by the same factor and they are presented in the same scale, so they can be compared. Since all sensitivity values are guaranteed to be non-positive, the logarithm of the negative of the sensitivity values can be taken to improve visualization. This is necessary because some loaded and clamped elements usually have sensitivity values much larger (in modulus) than the rest of the elements. In a linear scale, this would result in most of the elements being represented by the same color, even though they have different sensitivity values.

The topology and the four sensitivity maps are saved in a single figure, in a 3×2 grid of subplots.

bsample

```
    # topology vectors and sensitivity vectors
    if fig_top_sen:
        print(': : topology vectors and sensitivity vectors...')
        for k in range(len(list_fid)):
            # boundary conditions
            fid = list_fid[k]
            inp = list_inp[k]
            ycoor = esize*np.array(list(range(Ny+1)))-0.5*Ly
            mask = (ycoor > inp[0]-inp[1]-small) & (ycoor < inp[0]+inp[1]+small)
            c0 = np.zeros((32,1))
            c0[mask[1:]]  += 0.25
            c0[mask[:-1]] += 0.25
            mask = (ycoor > inp[2]-inp[3]-small) & (ycoor < inp[2]+inp[3]+small)
            c1 = np.zeros((32,1))
            c1[mask[1:]]  += 0.25
            c1[mask[:-1]] += 0.25
            j = 0
            for kk in range(list_ptr2opt[k],list_ptr2opt[k+1]):
                # figure
                plt.figure(num=0).clear()
                fig,ax = plt.subplots(nrows=3,ncols=2,num=0)
                x = list_top[kk]
                x = np.unpackbits(x,axis=None).astype(float)
                xmat = np.reshape(x,(Ny,Nx),order='F')
                xmat = np.concatenate((c0,xmat,c1),axis=1)
                ax[0,0].imshow(xmat,cmap='gray_r',vmin=0,vmax=1.0,origin='lower')
                ax[0,0].axis('off')
                alpha_0 = list_sen_0[kk].astype(float)
                alpha_1 = list_sen_1[kk].astype(float)
                alpha_2 = list_sen_2[kk].astype(float)
                alpha_w = list_sen_w[kk].astype(float)
                mval = max([max(abs(alpha_0)),max(abs(alpha_1)),max(abs(alpha_2)),max(abs(alpha_w))])
                alpha_0 = alpha_0/mval
                alpha_1 = alpha_1/mval
                alpha_2 = alpha_2/mval
                alpha_w = alpha_w/mval
                amat = np.reshape(np.log(-alpha_0+1e-6),(Ny,Nx),order='F')
                ax[0,1].imshow(amat,cmap='jet',vmin=np.log(1e-6),vmax=np.log(1.0+1e-6),origin='lower')
                ax[0,1].axis('off')
                amat = np.reshape(np.log(-alpha_1+1e-6),(Ny,Nx),order='F')
                ax[1,1].imshow(amat,cmap='jet',vmin=np.log(1e-6),vmax=np.log(1.0+1e-6),origin='lower')
                ax[1,1].axis('off')
                amat = np.reshape(np.log(-alpha_2+1e-6),(Ny,Nx),order='F')
                ax[2,1].imshow(amat,cmap='jet',vmin=np.log(1e-6),vmax=np.log(1.0+1e-6),origin='lower')
                ax[2,1].axis('off')
                amat = np.reshape(np.log(-alpha_w+1e-6),(Ny,Nx),order='F')
                ax[2,0].imshow(amat,cmap='jet',vmin=np.log(1e-6),vmax=np.log(1.0+1e-6),origin='lower')
                ax[2,0].axis('off')
                ax[1,0].axis('off')
                fig.set_size_inches(12, 9)
                plt.savefig('./top_sen/f{:06d}_{:03d}.png'.format(fid,j),bbox_inches='tight',pad_inches=0.05,dpi=100)
                j += 1
```

If "fig_dis" is *True*, images of the deformed structures are saved. In order to visualize the displacements field, the topologies are represented in the deformed mesh. The scale is kept the same for all figures of the same optimization procedure, so the influence of each topological change can be perceived.

bsample

```
    # displacements vectors
    if fig_dis:
        print(': : displacements vectors...')
        # coordinates matrix
        xcoor = (Ny+1)*[list(range(Nx+2+1))]
        xcoor = np.ravel(xcoor,'F')
        ycoor = (Nx+2+1)*[list(range(Ny+1))]
        ycoor = np.ravel(ycoor,'C')
        coor = esize*np.array([xcoor,ycoor]).T
        coor[:,1] = coor[:,1] - 0.5*Ly
        # incidence matrix
        N = (Nx+2)*Ny
        inci = np.ndarray([N,4],dtype=int)
        elem_ids = np.arange(N)
        inci[:,0] = elem_ids + elem_ids//Ny
        inci[:,1] = inci[:,0] + Ny + 1
        inci[:,2] = inci[:,0] + Ny + 2
        inci[:,3] = inci[:,0] + 1
        for k in range(len(list_fid)):
            # boundary conditions
            fid = list_fid[k]
            inp = list_inp[k]
            ycoor = esize*np.array(list(range(Ny+1)))-0.5*Ly
            mask = (ycoor > inp[0]-inp[1]-small) & (ycoor < inp[0]+inp[1]+small)
            c0 = np.zeros(32)
            c0[mask[1:]]  += 0.25
            c0[mask[:-1]] += 0.25
            mask = (ycoor > inp[2]-inp[3]-small) & (ycoor < inp[2]+inp[3]+small)
            c1 = np.zeros(32)
```

```
            c1[mask[1:]]  += 0.25
            c1[mask[:-1]] += 0.25
            j = 0
            for kk in range(list_ptr2opt[k],list_ptr2opt[k+1]):
                # figure
                plt.figure(num=0).clear()
                fig,ax = plt.subplots(num=0)
                ug = list_dis[kk]
                if j == 0:
                    scale = 0.50*Ly/max(abs(ug))
                ug = np.concatenate((ug[:66],ug,ug[-66:]))
                umat = np.reshape(ug,coor.shape)
                coor_dis = coor + scale*umat
                if j == 0:
                    xmax = max(coor_dis[:,0])
                    xmin = min(coor_dis[:,0])
                    ymax = max(coor_dis[:,1])
                    ymin = min(coor_dis[:,1])
                    Dx = xmax-xmin
                    Dy = ymax-ymin
                polys = clct.PolyCollection(coor_dis[inci],cmap='gray_r',edgecolor=(0,0,0,0))
                x = list_top[kk]
                x = np.unpackbits(x,axis=None).astype(float)
                x = np.concatenate((c0,x,c1))
                polys.set_array(x)
                polys.set_clim(0.0,1.0)
                ax.add_collection(polys)
                ax.set_aspect('equal')
                ax.set_xlim([xmin-0.01*Dx,xmax+0.01*Dx])
                ax.set_ylim([ymin-0.05*Dy,ymax+0.01*Dy])
                ax.axis('off')
                fig.set_size_inches(8, 6)
                fig.savefig('./dis/f{:06d}_{:03d}.png'.format(fid,j),bbox_inches='tight',pad_inches=0,dpi=100)
                j += 1
```

If "fig_obj_vol" is *True*, plots of objective function and volume are saved. The evolution of these functions throughout each optimization procedure is plotted in the same figure, in a 2×1 grid of subplots.

The file counter "file" is updated so the next images can be generated.

bsample

```
    # objective function and volume
    if fig_obj_vol:
        print(': : objective function and volume...')
        for k in range(len(list_fid)):
            fid = list_fid[k]
            plt.figure(num=0).clear()
            fig,ax = plt.subplots(nrows=2,ncols=1,num=0)
            obj = list_obj[list_ptr2opt[k]:list_ptr2opt[k+1]]
            delta = max(obj) - min(obj)
            miny = min(obj)-0.02*delta
            maxy = max(obj)+0.02*delta
            ax[0].plot(obj,'ok-',linewidth=2)
            ax[0].axis([-0.75, len(obj)-0.25, miny, maxy])
            ax[0].set_ylabel('compliance [J]',fontsize=18)
            ax[0].grid()
            vol = list_vol[list_ptr2opt[k]:list_ptr2opt[k+1]]
            ax[1].plot(vol,'ok-',linewidth=2)
            ax[1].axis([-0.75, len(obj)-0.25, -0.05, 1.05])
            ax[1].set_ylabel('volume fraction',fontsize=18)
            ax[1].grid()
            ax[1].set_xlabel('iteration',fontsize=18)
            fig.set_size_inches(8, 9)
            fig.savefig('./obj_vol/f{:06d}.png'.format(fid),bbox_inches='tight',pad_inches=0.05,dpi=100)
    # prepare to read next file
    file = file + 1
```

When all selected images are generated, the figure window is closed and the program terminates.

bsample

```
plt.close(fig=0)
print('done!')
```

3.5.2 ./sample/SIMP/ssample.py

This script generates figures for a selected sample of the SIMP dataset. For a given selection of files, png images for the topologies, sensitivity maps and displacements fields can be generated and saved in subfolders of the "sample/SIMP" directory. Plots of the objective function, volume of material and gray level throughout a given optimization can also be generated and saved as png images.

Firstly, the necessary modules are imported.

ssample
```
import os, sys
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.collections as clct
```

Then, the user selects which files should be read through the parameters "file_ini", which specifies the index of the first file to be read, and "file_lim", which specifies the file index limit (non inclusive). Boolean flags are used to specify which figures should be generated: if "fig_top_opt" is *True*, figures are generated for all optimized topologies and discretized optimized topologies; if "fig_top_sen" is *True*, figures are generated for all topologies (not only the optimized) and sensitivity vectors; if "fig_dis" is *True*, figures are generated for all displacements vectors; and if "fig_obj_vol_gra" is *True*, figures are generated with all the plots of objective function, volume and gray level.

In order to avoid generating an unreasonable number of figures, users should be careful when defining these parameters. Each file contains the results of "noptf" optimizations, and each optimization produces dozens (73 in average) of topology, sensitivity and displacements vectors.

ssample
```
file_ini        = 0     # initial file index |from file 0
file_lim        = 4     # file index limit   |up to file 9264
fig_top_opt     = True  # optimized topology and discretized optimized topology
fig_top_sen     = True  # topology vectors and sensitivity vectors
fig_dis         = True  # displacements vectors
fig_obj_vol_gra = True  # objective function, volume and gray level
```

Some properties of the cantilever are defined, the program checks if the "dataset" folder exists, then the folders in which the figures will be saved are created.

ssample
```
# fixed properties
Ly = 1.0       # cantilever height
small = 1e-14  # small value to compare float numbers
Ny = 32        # number of elements in y-axis
Nx = 2*Ny      # number of elements in x-axis
N = Nx*Ny      # total number of elements
esize = Ly/Ny  # element size
# check directories
rpath = '../../dataset/SIMP/'
if not os.path.exists(rpath):
    print('missing SIMP dataset')
    sys.exit()
if not os.path.exists('./top_opt'):
    os.mkdir('./top_opt')
if not os.path.exists('./top_sen'):
    os.mkdir('./top_sen')
if not os.path.exists('./dis'):
    os.mkdir('./dis')
if not os.path.exists('./obj_vol_gra'):
    os.mkdir('./obj_vol_gra')
```

The loop through the selected files is started. The input files indices, the input data and the pointers relating each input with the corresponding iterations of the optimization processes are read from the "fid.npy", "inp.npy" and "ptr2opt.npy" files.

ssample
```
file = file_ini
while (file < file_lim) and (os.path.exists(rpath + 'f{:04d}'.format(file))):
    #%% Read files
    print('> reading files of f{:04d}'.format(file))
    # input files id, input data and pointers to optimization
    if not os.path.exists(rpath + 'f{:04d}/fid.npy'.format(file)):
        print('missing : f{:04d}/fid.npy'.format(file))
        sys.exit()
    list_fid = np.load(rpath + 'f{:04d}/fid.npy'.format(file))
    if not os.path.exists(rpath + 'f{:04d}/inp.npy'.format(file)):
        print('missing : f{:04d}/inp.npy'.format(file))
        sys.exit()
    list_inp = np.load(rpath + 'f{:04d}/inp.npy'.format(file))
    if not os.path.exists(rpath + 'f{:04d}/ptr2opt.npy'.format(file)):
        print('missing : f{:04d}/ptr2opt.npy'.format(file))
        sys.exit()
    list_ptr2opt = np.load(rpath + 'f{:04d}/ptr2opt.npy'.format(file))
```

According to the boolean flags, the optimized topologies, discretized optimized topologies, topologies, sensitivity vectors, displacements vectors, objective function values, volume values and gray level values are read from the "top_p1_opt.npy", "dtop_opt.npy", "top_p1.npy", "sen_p1.npy", "dis.npy", "obj.npy", "vol_p1.npy" and "gra_p1.npy" files.

ssample

```
    # optimized topology and discretized optimized topology
    if fig_top_opt:
        if not os.path.exists(rpath + 'f{:04d}/top_p1_opt.npy'.format(file)):
            print('missing : f{:04d}/top_p1_opt.npy'.format(file))
            sys.exit()
        list_top_p1_opt = np.load(rpath + 'f{:04d}/top_p1_opt.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dtop_opt.npy'.format(file)):
            print('missing : f{:04d}/dtop_opt.npy'.format(file))
            sys.exit()
        list_dtop_opt = np.load(rpath + 'f{:04d}/dtop_opt.npy'.format(file))
    # topology vectors
    if fig_top_sen or fig_dis:
        if not os.path.exists(rpath + 'f{:04d}/top_p1.npy'.format(file)):
            print('missing : f{:04d}/top_p1.npy'.format(file))
            sys.exit()
        list_top_p1 = np.load(rpath + 'f{:04d}/top_p1.npy'.format(file))
    # sensitivity vectors
    if fig_top_sen:
        if not os.path.exists(rpath + 'f{:04d}/sen_p1.npy'.format(file)):
            print('missing : f{:04d}/sen_p1.npy'.format(file))
            sys.exit()
        list_sen_p1 = np.load(rpath + 'f{:04d}/sen_p1.npy'.format(file))
    # displacements vectors
    if fig_dis:
        if not os.path.exists(rpath + 'f{:04d}/dis.npy'.format(file)):
            print('missing : f{:04d}/dis.npy'.format(file))
            sys.exit()
        list_dis = np.load(rpath + 'f{:04d}/dis.npy'.format(file))
    # objective function, volume and gray level
    if fig_obj_vol_gra:
        if not os.path.exists(rpath + 'f{:04d}/obj.npy'.format(file)):
            print('missing : f{:04d}/obj.npy'.format(file))
            sys.exit()
        list_obj = np.load(rpath + 'f{:04d}/obj.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/vol_p1.npy'.format(file)):
            print('missing : f{:04d}/vol_p1.npy'.format(file))
            sys.exit()
        list_vol_p1 = np.load(rpath + 'f{:04d}/vol_p1.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/gra_p1.npy'.format(file)):
            print('missing : f{:04d}/gra_p1.npy'.format(file))
            sys.exit()
        list_gra_p1 = np.load(rpath + 'f{:04d}/gra_p1.npy'.format(file))
```

If "fig_top_opt" is *True*, images of the optimized topologies are saved. The optimized topologies and discretized optimized topologies are represented as 66×32 grayscale images. The first and last columns are used to represent the clamped and loaded regions.

The optimized topology and discretized optimized topology are saved in a single figure, in a 2×1 grid of subplots.

ssample

```
    #%% Generate figures
    print(': generating figures')
    # optimized topology and discretized optimized topology
    if fig_top_opt:
        print(': : optimized topology and discretized optimized topology...')
        for k in range(len(list_fid)):
            # boundary conditions
            fid = list_fid[k]
            inp = list_inp[k]
            ycoor = esize*np.array(list(range(Ny+1)))-0.5*Ly
            mask = (ycoor > inp[0]-inp[1]-small) & (ycoor < inp[0]+inp[1]+small)
            c0 = np.zeros((32,1))
            c0[mask[1:]]  += 0.25
            c0[mask[:-1]] += 0.25
            mask = (ycoor > inp[2]-inp[3]-small) & (ycoor < inp[2]+inp[3]+small)
            c1 = np.zeros((32,1))
            c1[mask[1:]]  += 0.25
            c1[mask[:-1]] += 0.25
            # figure
            plt.figure(num=0).clear()
            fig,ax = plt.subplots(nrows=2,ncols=1,num=0)
            x = list_top_p1_opt[k]
            xmat = np.reshape(x,(Ny,Nx),order='F')
            xmat = np.concatenate((c0,xmat,c1),axis=1)
            ax[0].imshow(xmat,cmap='gray_r',vmin=0,vmax=1.0,origin='lower')
            ax[0].axis('off')
            x = list_dtop_opt[k]
            x = np.unpackbits(x,axis=None).astype(float)
            xmat = np.reshape(x,(Ny,Nx),order='F')
            xmat = np.concatenate((c0,xmat,c1),axis=1)
            ax[1].imshow(xmat,cmap='gray_r',vmin=0,vmax=1.0,origin='lower')
            ax[1].axis('off')
            fig.set_size_inches(6, 6)
            plt.savefig('./top_opt/f{:06d}.png'.format(fid),bbox_inches='tight',pad_inches=0.05,dpi=100)
```

If "fig_top_sen" is *True*, images of the topology and sensitivity vectors are saved. The topologies are represented as 66×32 grayscale images. The first and last columns are used to represent the clamped and loaded regions. Each sensitivity map is represented as 64×32 single-channeled image, the "jet" colormap is used instead of grayscale. Since all sensitivity values are guaranteed to be non-positive, the logarithm of the negative of the sensitivity values can be taken to improve visualization. This is necessary because some loaded and clamped elements usually have sensitivity values much larger (in modulus) than the rest of the elements. In a linear scale, this would result in most of the elements being represented by the same color, even though they have different sensitivity values.

The topology and the sensitivity map are saved in a single figure, in a 2×1 grid of subplots.

ssample

```
    # topology vectors and sensitivity vectors
    if fig_top_sen:
        print(': : topology vectors and sensitivity vectors...')
        for k in range(len(list_fid)):
            # boundary conditions
            fid = list_fid[k]
            inp = list_inp[k]
            ycoor = esize*np.array(list(range(Ny+1)))-0.5*Ly
            mask = (ycoor > inp[0]-inp[1]-small) & (ycoor < inp[0]+inp[1]+small)
            c0 = np.zeros((32,1))
            c0[mask[1:]]  += 0.25
            c0[mask[:-1]] += 0.25
            mask = (ycoor > inp[2]-inp[3]-small) & (ycoor < inp[2]+inp[3]+small)
            c1 = np.zeros((32,1))
            c1[mask[1:]]  += 0.25
            c1[mask[:-1]] += 0.25
            j = 0
            for kk in range(list_ptr2opt[k],list_ptr2opt[k+1]):
                # figure
                plt.figure(num=0).clear()
                fig,ax = plt.subplots(nrows=2,ncols=1,num=0)
                x = list_top_p1[kk]
                xmat = np.reshape(x,(Ny,Nx),order='F')
                xmat = np.concatenate((c0,xmat,c1),axis=1)
                ax[0].imshow(xmat,cmap='gray_r',vmin=0,vmax=1.0,origin='lower')
                ax[0].axis('off')
                alpha = list_sen_p1[kk].astype(float)
                alpha = alpha/max(abs(alpha))
                amat = np.reshape(np.log(-alpha+1e-6),(Ny,Nx),order='F')
                ax[1].imshow(amat,cmap='jet',origin='lower')
                ax[1].axis('off')
                fig.set_size_inches(6, 6)
                plt.savefig('./top_sen/f{:06d}_{:03d}.png'.format(fid,j),bbox_inches='tight',pad_inches=0.05,dpi=100)
                j += 1
```

If "fig_dis" is *True*, images of the deformed structures are saved. In order to visualize the displacements field, the topologies are represented in the deformed mesh. The scale is kept the same for all figures of the same optimization procedure, so the influence of each topological change can be perceived.

ssample

```
    # displacements vectors
    if fig_dis:
        print(': : displacements vectors...')
        # coordinates matrix
        xcoor = (Ny+1)*[list(range(Nx+2+1))]
        xcoor = np.ravel(xcoor,'F')
        ycoor = (Nx+2+1)*[list(range(Ny+1))]
        ycoor = np.ravel(ycoor,'C')
        coor = esize*np.array([xcoor,ycoor]).T
        coor[:,1] = coor[:,1] - 0.5*Ly
        # incidence matrix
        N = (Nx+2)*Ny
        inci = np.ndarray([N,4],dtype=int)
        elem_ids = np.arange(N)
        inci[:,0] = elem_ids + elem_ids//Ny
        inci[:,1] = inci[:,0] + Ny + 1
        inci[:,2] = inci[:,0] + Ny + 2
        inci[:,3] = inci[:,0] + 1
        for k in range(len(list_fid)):
            # boundary conditions
            fid = list_fid[k]
            inp = list_inp[k]
            ycoor = esize*np.array(list(range(Ny+1)))-0.5*Ly
            mask = (ycoor > inp[0]-inp[1]-small) & (ycoor < inp[0]+inp[1]+small)
            c0 = np.zeros(32)
            c0[mask[1:]]  += 0.25
            c0[mask[:-1]] += 0.25
            mask = (ycoor > inp[2]-inp[3]-small) & (ycoor < inp[2]+inp[3]+small)
            c1 = np.zeros(32)
            c1[mask[1:]]  += 0.25
            c1[mask[:-1]] += 0.25
```

```
            j = 0
            for kk in range(list_ptr2opt[k],list_ptr2opt[k+1]):
                # figure
                plt.figure(num=0).clear()
                fig,ax = plt.subplots(num=0)
                ug = list_dis[kk]
                if j == 0:
                    scale = 0.50*Ly/max(abs(ug))
                ug = np.concatenate((ug[:66],ug,ug[-66:]))
                umat = np.reshape(ug,coor.shape)
                coor_dis = coor + scale*umat
                if j == 0:
                    xmax = max(coor_dis[:,0])
                    xmin = min(coor_dis[:,0])
                    ymax = max(coor_dis[:,1])
                    ymin = min(coor_dis[:,1])
                    Dx = xmax-xmin
                    Dy = ymax-ymin
                polys = clct.PolyCollection(coor_dis[inci],cmap='gray_r',edgecolor=(0,0,0,0))
                x = list_top_p1[kk]
                x = np.concatenate((c0,x,c1))
                polys.set_array(x)
                polys.set_clim(0.0,1.0)
                ax.add_collection(polys)
                ax.set_aspect('equal')
                ax.set_xlim([xmin-0.01*Dx,xmax+0.01*Dx])
                ax.set_ylim([ymin-0.01*Dy,ymax+0.01*Dy])
                ax.axis('off')
                fig.set_size_inches(8, 6)
                fig.savefig('./dis/f{:06d}_{:03d}.png'.format(fid,j),bbox_inches='tight',pad_inches=0,dpi=100)
                j += 1
```

If "fig_obj_vol_gra" is *True*, plots of objective function, volume and gray level are saved. The evolution of these functions throughout each optimization procedure is plotted in the same figure, in a 2×2 grid of subplots.

The file counter "file" is updated so the next images can be generated.

ssample

```
    # objective function, volume and gray level
    if fig_obj_vol_gra:
        print(': : objective function, volume and gray level...')
        for k in range(len(list_fid)):
            fid = list_fid[k]
            plt.figure(num=0).clear()
            fig,ax = plt.subplots(nrows=2,ncols=2,num=0)
            obj = list_obj[list_ptr2opt[k]:list_ptr2opt[k+1]]
            delta = max(obj) - min(obj)
            miny = min(obj)-0.02*delta
            maxy = max(obj)+0.02*delta
            ax[0,0].plot(obj,'ok-',linewidth=2)
            ax[0,0].axis([-0.75, len(obj)-0.25, miny, maxy])
            ax[0,0].set_ylabel('compliance [J]',fontsize=18)
            ax[0,0].grid()
            delta = max(obj[16:]) - min(obj)
            miny = min(obj)-0.02*delta
            maxy = max(obj[16:])+0.02*delta
            ax[1,0].plot(obj,'ok-',linewidth=2)
            ax[1,0].axis([-0.75, len(obj)-0.25, miny, maxy])
            ax[1,0].set_ylabel('compliance [J]',fontsize=18)
            ax[1,0].set_xlabel('iteration',fontsize=18)
            ax[1,0].grid()
            vol = list_vol_p1[list_ptr2opt[k]:list_ptr2opt[k+1]]
            ax[0,1].plot(vol,'ok-',linewidth=2)
            ax[0,1].axis([-0.75, len(obj)-0.25, -0.05, 1.05])
            ax[0,1].set_ylabel('volume fraction',fontsize=18)
            ax[0,1].grid()
            gra = list_gra_p1[list_ptr2opt[k]:list_ptr2opt[k+1]]
            ax[1,1].plot(gra,'ok-',linewidth=2)
            ax[1,1].axis([-0.75, len(obj)-0.25, -0.05, 1.05])
            ax[1,1].set_ylabel('gray level',fontsize=18)
            ax[1,1].set_xlabel('iteration',fontsize=18)
            ax[1,1].grid()
            fig.set_size_inches(16, 9)
            fig.savefig('./obj_vol_gra/f{:06d}.png'.format(fid),bbox_inches='tight',pad_inches=0.05,dpi=100)
    # prepare to read next file
    file = file + 1
```

When all selected images are generated, the figure window is closed and the program terminates.

ssample

```
plt.close(fig=0)
print('done!')
```

3.6 Validation Procedure

The validation procedure consists mainly in independent, alternative implementations to perform each task of the optimization programs that generate the datasets.

The validation codes are provided in the folder "./validation". The bash scipt "canval.sh" can be executed in order to setup the conda environment used for validation.

This script will: update conda; create a new Python-3.8 environment named "canval"; add the channel conda-forge; set channel_priority as strict; install numpy [17], scipy [18], matplotlib [19], fenics [23, 24], cython [20], scikit-sparse [21] and nlopt [22]; and build all Cython codes. The fenics package is used to perform an alternative FEA. So, the mesh generation; the assembly of the linear system; and the obtained displacements vectors can be validated.

The script "cantilever_val.py" can be executed to perform the following validations.

Firstly, two different topology vectors are created. The first one corresponds to a given non-trivial discrete topology, and the second one corresponds to a continuous random distribution of density values. The boundary conditions are given by: $p_{bc} = -0.250\,m$, $r_{bc} = 0.125\,m$, $p_{ld} = 0.250\,m$ and $r_{ld} = 0.125\,m$. The displacements vectors are compared using their L-∞ norms. In both discrete and continuous cases, the relative differences between displacements obtained by the programs used in the datasets generation and by the auxiliar program using fenics were below 0.0001%.

In the SILP-BESO approach, the factorized matrix is directly updated after altering the state of each element, instead of performing a new factorization from scratch in each iteration. This is faster for coarse meshes, however, it may cause numerical imprecision if too many alterations are performed throughout the optimization process. To evaluate if this procedure is reliable, considering the same discrete topology from the previous validation (in which 41.4% of the elements are void ones), all void elements are turned into solid ones, and then turned back into void ones. This is repeated 100 times and, each time, the order of the elements to be altered is randomized. After these 200 alterations of each element, the updated factorized matrix is used to compute again the displacements vector, the relative difference was below 0.0001%.

Then, the sensitivity values are validated for each method. For the SIMP-MMA approach, the sensitivity vector computed in the "structural_ssens.pyx" script is compared with a sensitivity vector obtained through exact numerical differentiation, using dual numbers. A python class is created to define the dual numbers, emulating a numeric type. Considering the same continuous topology from the previous validation, the relative difference between the sensitivity vectors was below 0.0001%.

For the SILP-BESO approach, the CGS values computed in the "structural_bsens.pyx" script are compared with sensitivity values obtained using estimations for the displacements vectors after altering each element, using the Conjugate Gradient Method. The WS expression was compared with sensitivity values obtained through a naive approach, using exact values for the displacements vectors after altering each element. Considering the same discrete topology from the previous validations, the relative differences for CGS-0, CGS-1, CGS-2 and WS were all below 0.0001%.

Lastly, through visual verification, a qualitative validation is performed for the conical filter used to smooth the sensitivity maps. Besides filtering the sensitivity maps from the previous validations, some toy examples are also included to verify if the smoothing procedures are working as expected. Both filters are applied, the one that filters the whole design domain, and the one that ignores loaded elements.

After executing the script "cantilever_val.py", a number of samples was selected from the generated datasets and visually verified, using the presented "bsample.py" and "ssample.py" scripts. Some of them are shown in the next section. All observed results are coherent and indicate that the optimization programs were properly implemented.

As already explained, although some procedures are embarrassingly parallelizable, it has been decided to keep the programs serialized and call multiple parallel executions, using multiple processors of the machine. Nonetheless, parallel versions of some functions have been implemented and they are still present in the validation Cython scripts. These parallel versions use "cython.parallel.prange" to perform parallel loops using OpenMP.

4 Samples

Both the BESO and SIMP datasets store everything generated in the optimization procedures for each one of the 148 240 non-redundant quadruplets $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$. To illustrate the content of the datasets, the usual cantilever beam, with a fully clamped left edge and a central point load at the right extremity is considered. It corresponds to $p_{bc} = 0.0\,m$, $r_{bc} = 0.5\,m$, $p_{ld} = 0.0\,m$ and $r_{ld} = 0.0\,m$.

The results obtained with the SILP-BESO approach are presented in Figures 6, 7, 8 and 9, plotted by the "bsample" script.

From Figure 6, it can be noted that the volume fraction correctly progresses from 100% to 50% in 32 iterations, in a constant rate of 1.5625%. As expected, the objective function increases while the volume is decreasing. Some peaks occur when structural components break apart, changing the beam topology. After reaching the specified volume, the objective function decreases, then oscillates until the stopping criterion is achieved. This is a well behaved case, in which 62 iterations were performed. Since a patience parameter of 20 was used, the optimized solution corresponds to the topology of the 42-th iteration.

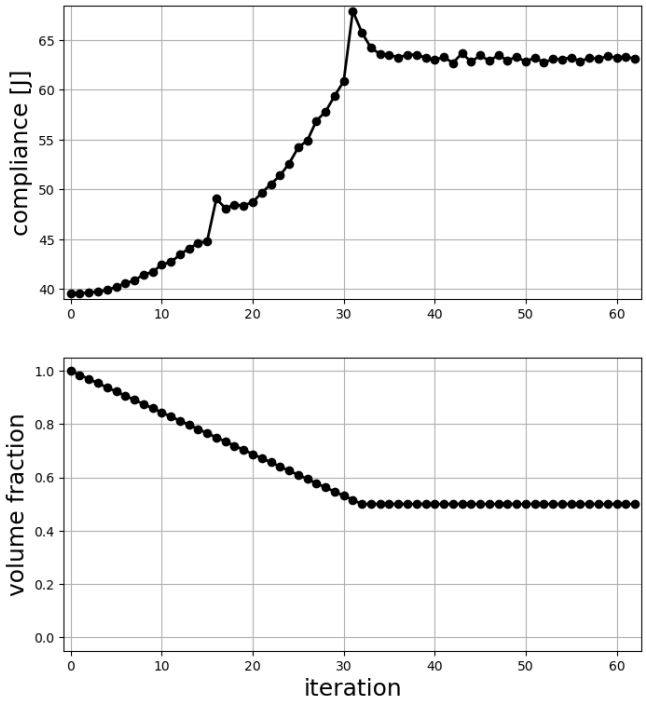

Figure 6: Objective and volume functions in the cantilever BESO-optimization

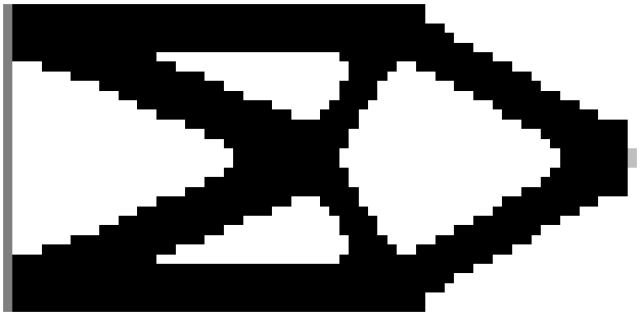

Figure 7: Optimized topology in the cantilever BESO-optimization

The optimized solution, from the 42-th iteration, is presented in Figure 7. Two extra columns have been added siding the left and the right edges of the domain, in order to inform which elements are clamped and which elements are loaded. In the left-most column, an element in white indicates that both nodes of its right edge are unrestricted, an element in light gray indicates that one node of its right edge is restricted, and an element in dark gray indicates that both nodes of its right edge are restricted. In the right-most column, an

element in white indicates that there is no load in any node of its left edge, an element in light gray indicates that one node of its left edge is loaded, and an element in dark gray indicates that both nodes of its left edge are loaded. Inside the design domain (excluding the left-most and the right-most columns), elements in white correspond to voids and elements in black correspond to solids.

Besides the objective and volume values, the topology and sensitivity vectors of each iteration are stored. Figure 8 shows the topology (top-left), the CGS-0 sensitivity map (top-right), the CGS-1 sensitivity map (middle-right), the CGS-2 sensitivity map (bottom-right) and the WS sensitivity map (bottom-left) for the 32-th iteration. The exact sensitivity values, given by the WS sensitivity map, are the ones used to update the topology in each iteration. As expected, it can be noted that the accuracy of the CGS approximation increases as more steps are considered (from 0 to 2).

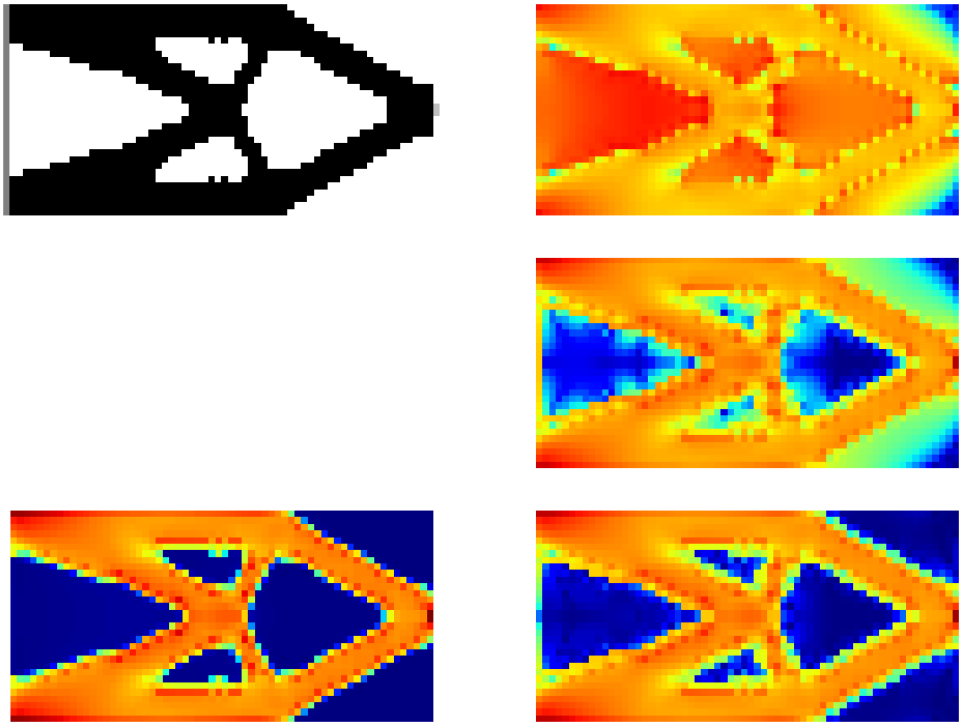

Figure 8: Topology and sensitivity maps in the cantilever BESO-optimization (it. 32)

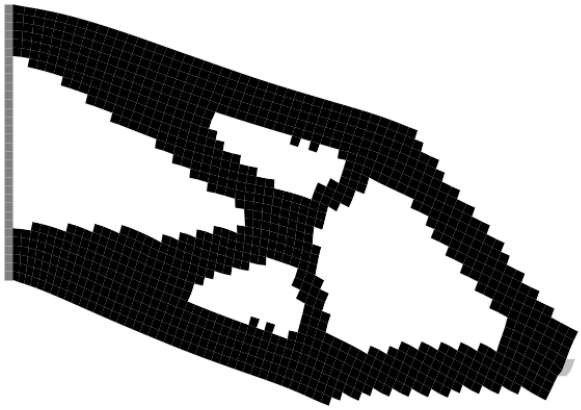

Figure 9: Deformed structure in the cantilever BESO-optimization (it. 32)

The displacements vector of each iteration is also stored. Figure 9 shows the deformed structure for the 32-th iteration. The displacements are rescaled in order to improve visualization.

The results obtained with the SIMP-MMA approach are presented in Figures 10, 11, 12 and 13, plotted by the "ssample" script.

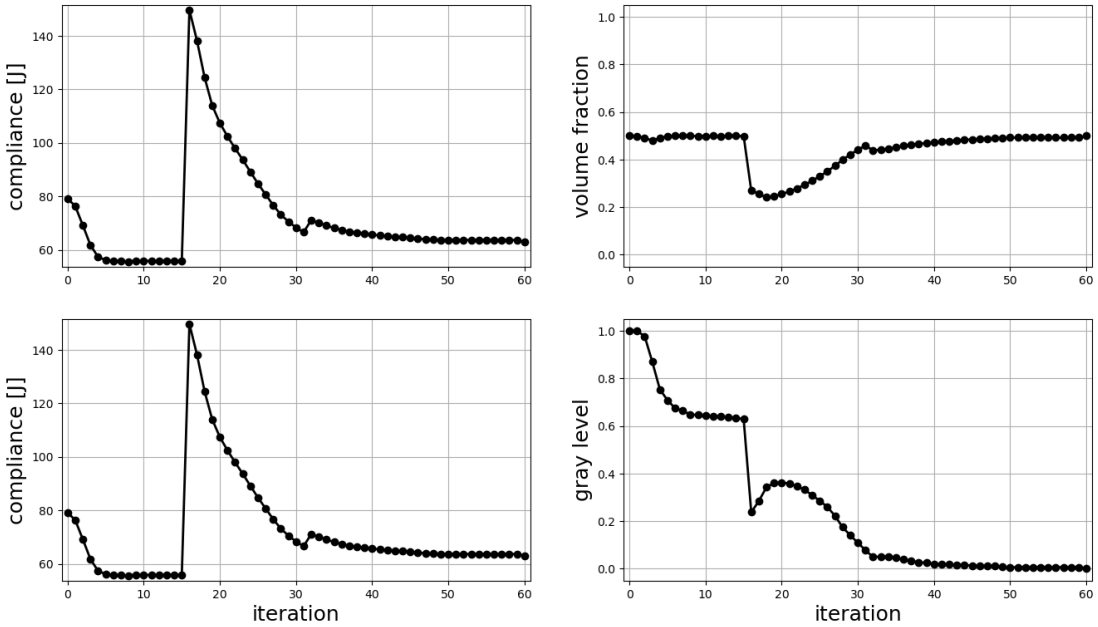

Figure 10: Objective, volume and gray level functions in the cantilever SIMP-optimization

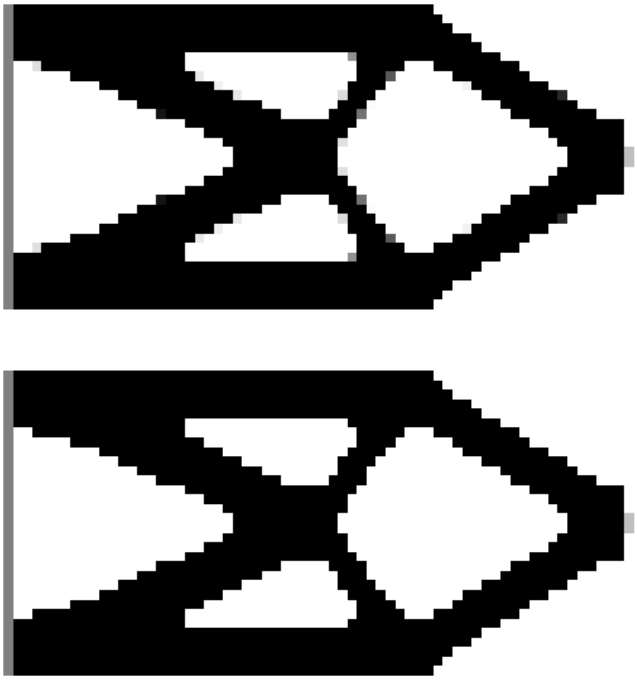

Figure 11: Optimized topology in the cantilever SIMP-optimization

Some abrupt variations can be noted in Figure 10, they occur at the 16-th iteration (when p is altered from 1 to 3) and at the 32-th iteration (when p is altered from 3 to 6). Evidently, this results in abrupt stiffness variations in the structure, which explain the behavior of the objective function. Moreover, since the dataset stores the adjusted density vectors, that represent the actual stiffness distributions over the mesh, the values of both volume and gray level are decreased when the penalization exponent is incremented. Such quick progression for the penalization exponent has been considered advantageous for the considered class of problems. Although it produces these abrupt changes in the behavior of the structure, the algorithm converges faster, while still producing efficient and coherent local minima.

The second plot of the objective function (bottom-left of Figure 10) ignores the first 16 compliance values when setting the scale of the vertical axis. This is done because, in the dataset, there are cases in which the initial topology (all elements with 50% of density) corresponds to a structure with a very high compliance value. In such cases, the behavior of the objective function can not be well perceived in the first plot, in which the

scale of the vertical axis is set according to the maximal value over all iterations. When there are no large peaks in the first 16 values, both plots are identical.

It can be noted that the volume fraction is kept near 50% in the first 16 evaluations ($p = 1$). It falls down when p is increased but, as the gray level is reduced because of the increased penalization, the volume fraction slowly increases again towards 50%. The objective function consistently decreases when the SIMP exponent p is kept constant. The stopping criteria were reached after 59 iterations. The 60-th iteration consists in the discretization of the result, so all elements are clearly classified as solids ($x_i = 1$) or voids ($x_i = 0$).

The optimized solution is presented in Figure 11. The continuous result from the MMA algorithm (iteration 59) is presented on top, and the discretized result (iteration 60) is presented on bottom. As before, two extra columns have been added to inform which elements are clamped and which elements are loaded. For the continuous result, inside the design domain, the density values are presented in grayscale, going from white ($x_i = 0$) to black ($x_i = 1$).

Besides the objective, volume and gray level values, the topology and sensitivity vectors of each iteration are stored. Figure 12 shows the topology (top) and the sensitivity map (bottom) for the 40-th iteration.

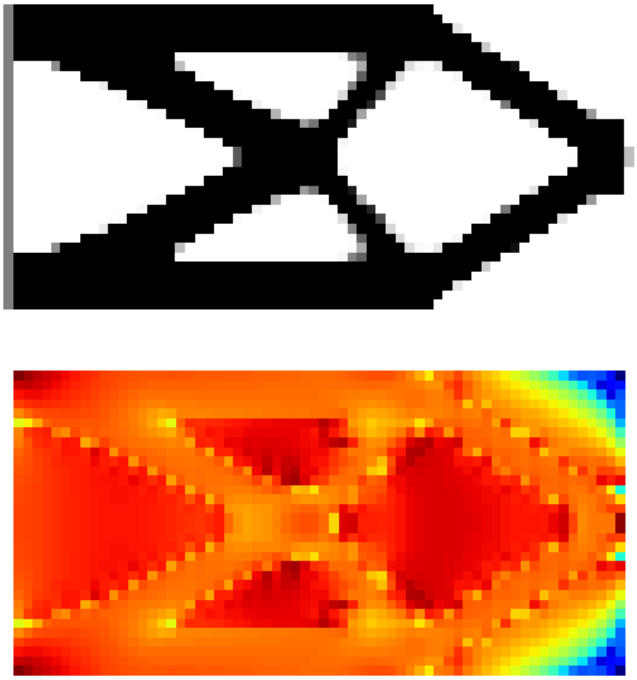

Figure 12: Topology and sensitivity map in the cantilever SIMP-optimization (it. 40)

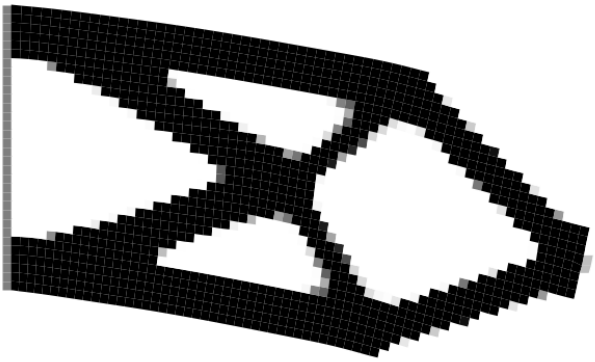

Figure 13: Deformed structure in the cantilever SIMP-optimization (it. 40)

The displacements vector of each iteration is also stored. Figure 13 shows the deformed structure for the 40-th iteration. The displacements are rescaled in order to improve visualization. It should be noted that the

scaling factor is based on the maximal displacement value of the initial topology. Since different initial topologies are considered in each approach (SILP-BESO and SIMP-MMA), Figure 9 and Figure 13 have different scaling factors.

Considering that these datasets may be used to train artificial neural networks that map the input space, given by all the quadruplets $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$, into the space of optimized topologies, it would be desirable that small input changes result in small changes in the optimized solutions. So, for this specific application, it would be desirable that similar local minima be obtained for similar boundary conditions. This property can be observed by defining three of the four input parameters, while varying the fourth one with minimal increments.

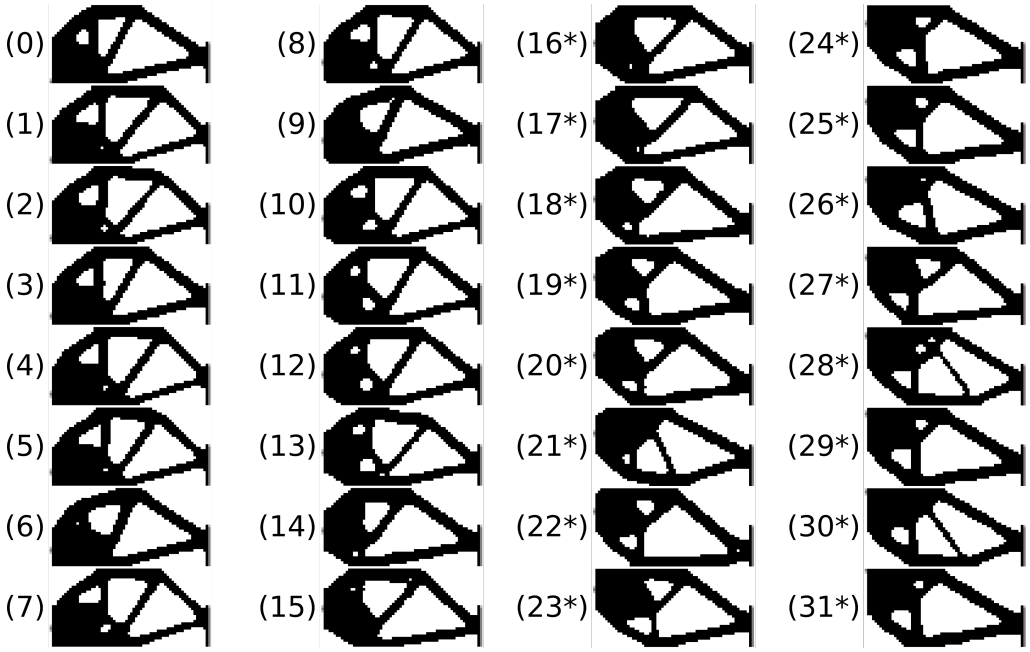

Figure 14: BESO-optimized topologies for different p_{bc} values

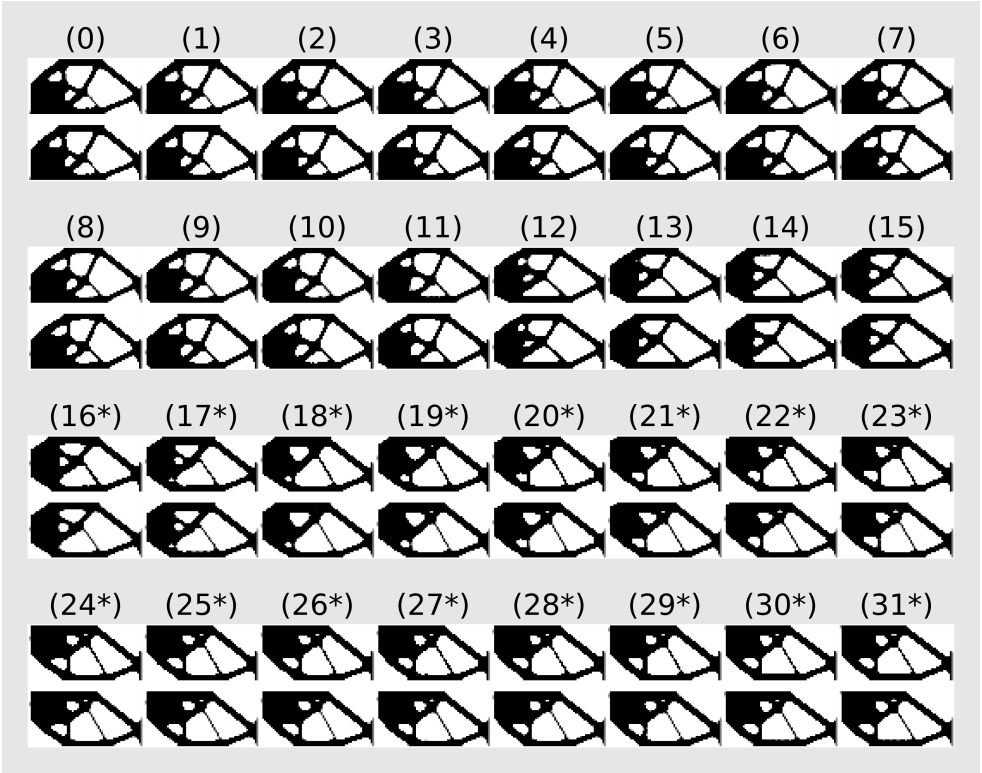

Figure 15: SIMP-optimized topologies for different p_{bc} values

For the SILP-BESO approach, the results presented in Figure 14 were obtained when varying the center position of the clamped region. A single element is clamped, so there are 32 (N_y) possible values for p_{bc}. The load is applied in the lower half of the right edge, so $p_{ld} = -0.25\,m$ and $r_{ld} = 0.25\,m$. Since only non-redundant entries are considered in the dataset, some of the presented results had to be mirrored over the horizontal axis, these are marked with an asterisk (topologies from 16* to 31*).

The same results are presented for the SIMP-MMA approach, in Figure 15. In all SIMP results, both the optimized continuous solution and the corresponding discretized topology are presented.

For a gradual change of one of the input parameters, the SIMP-MMA approach produces a set of gradually changing optimized topologies. As shown in Figure 14, most of the BESO results also have this property, however, the discrete method is more susceptible to abrupt changes through the optimization procedure. Even a small variation in the input entries may cause some structural components to break apart, taking the solution towards a different local minimum. It should be noted that, although this variability in the BESO solutions is undesirable for the mentioned application, it may be advantageous for others.

For the SILP-BESO approach, the results presented in Figure 16 were obtained when varying the half-length of the clamped region. The clamped region is centered, so $p_{bc} = 0.0\,m$ and there are 16 ($N_y/2$) possible values for r_{bc}. The load is applied in the lower half of the right edge, so $p_{ld} = -0.25\,m$ and $r_{ld} = 0.25\,m$. The same results are presented for the SIMP-MMA approach, in Figure 17.

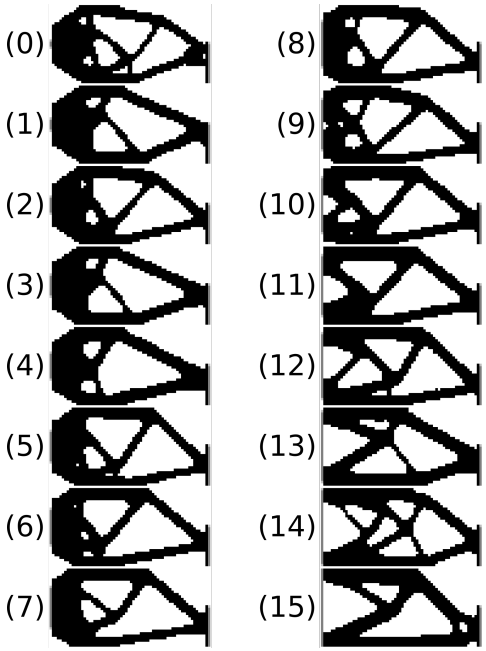

Figure 16: BESO-optimized topologies for different r_{bc} values

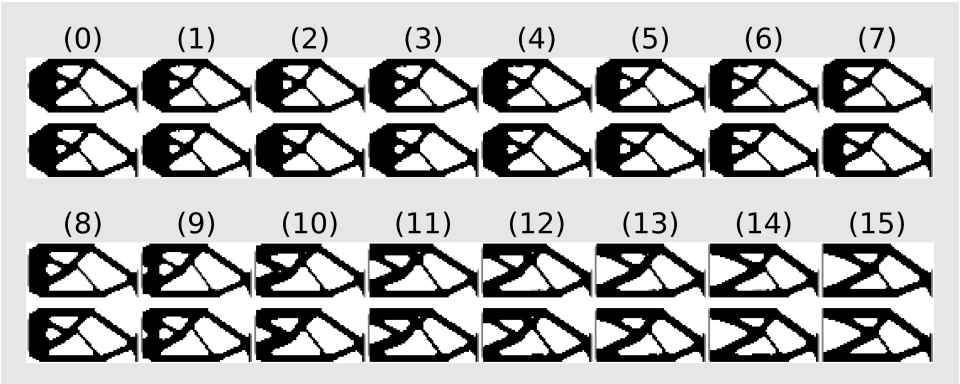

Figure 17: SIMP-optimized topologies for different r_{bc} values

For the SILP-BESO approach, the results presented in Figure 18 were obtained when varying the center position of the loaded region. When the center position is on a node, a point load is considered, when it is between nodes, both nodes are loaded. So there are 65 ($2\,N_y + 1$) possible values for p_{ld}. The mechanical

restriction is applied in the lower half of the left edge, so $p_{bc} = -0.25\,m$ and $r_{bc} = 0.25\,m$. The same results are presented for the SIMP-MMA approach, in Figure 19.

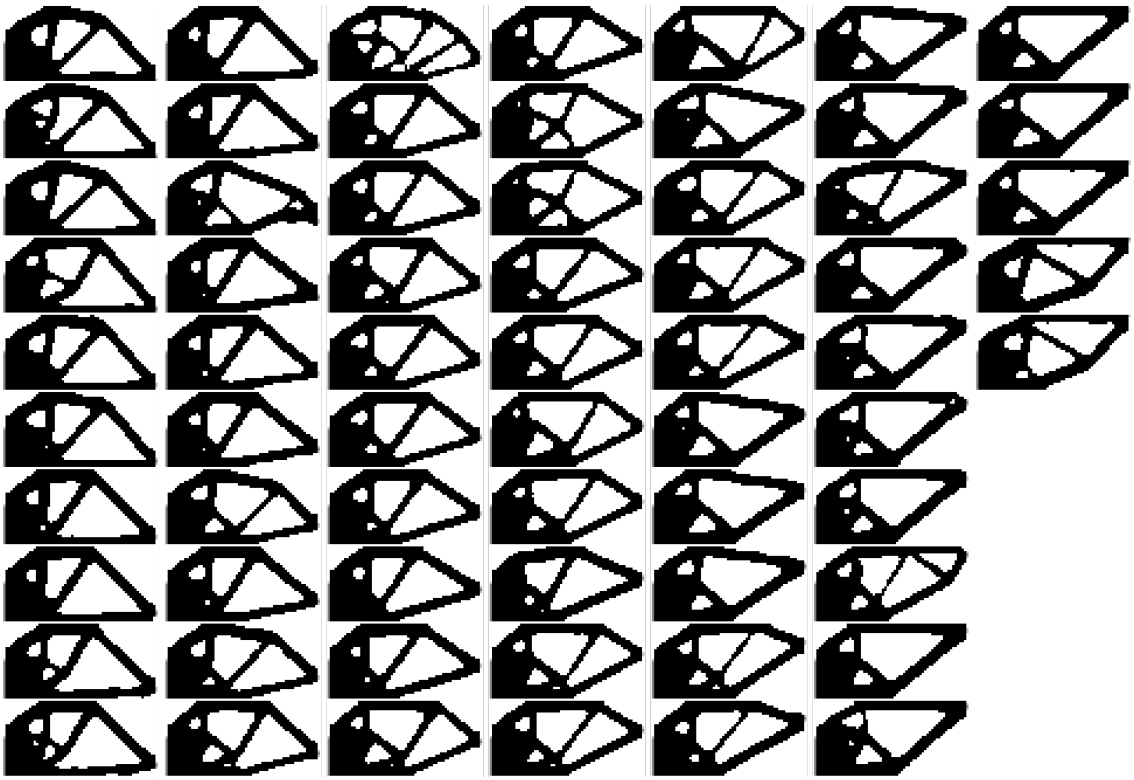

Figure 18: BESO-optimized topologies for different p_{ld} values

Figure 19: SIMP-optimized topologies for different p_{ld} values

For the SILP-BESO approach, the results presented in Figure 20 were obtained when varying the half-length of the loaded region. The loaded region is centered, so $p_{ld} = 0.0\,m$ and there are 17 $(N_y/2+1)$ possible values for r_{ld}. The mechanical restriction is applied in the lower half of the left edge, so $p_{bc} = -0.25\,m$ and $r_{bc} = 0.25\,m$. The same results are presented for the SIMP-MMA approach, in Figure 21.

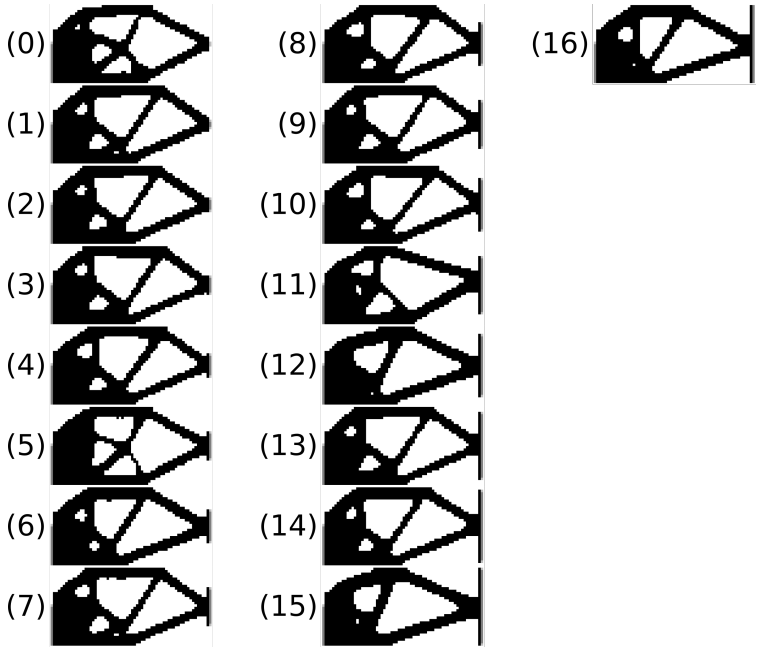

Figure 20: BESO-optimized topologies for different r_{ld} values

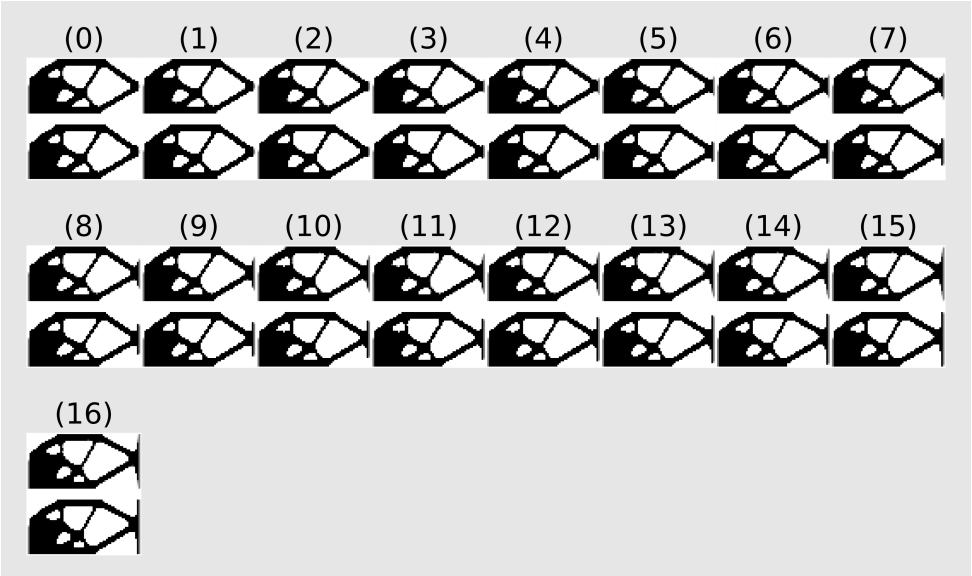

Figure 21: SIMP-optimized topologies for different r_{ld} values

To understand why there are cases in which a large number of iterations is necessary to achieve the stopping criteria, the cases with maximal number of iterations are presented.

For the SILP-BESO approach, the case in which $p_{bc} = -0.234375\,m$, $r_{bc} = 0.03125\,m$, $p_{ld} = -0.109375\,m$ and $r_{ld} = 0.3125\,m$ needed 360 iterations to converge. Figure 22 presents the evolution of the objective and volume functions over the optimization and Figure 23 presents the topology for some iterations. The optimized topology is marked with an asterisk (iteration 340*).

The large peak in the objective function plot occurs at the 31-th iteration, in which the topology was altered in very inefficient manner. At this iteration, an important structural component broke apart. Usually, when this happens, the remaining material of such component is quickly removed, since it does not contribute to the

structure stiffness anymore.

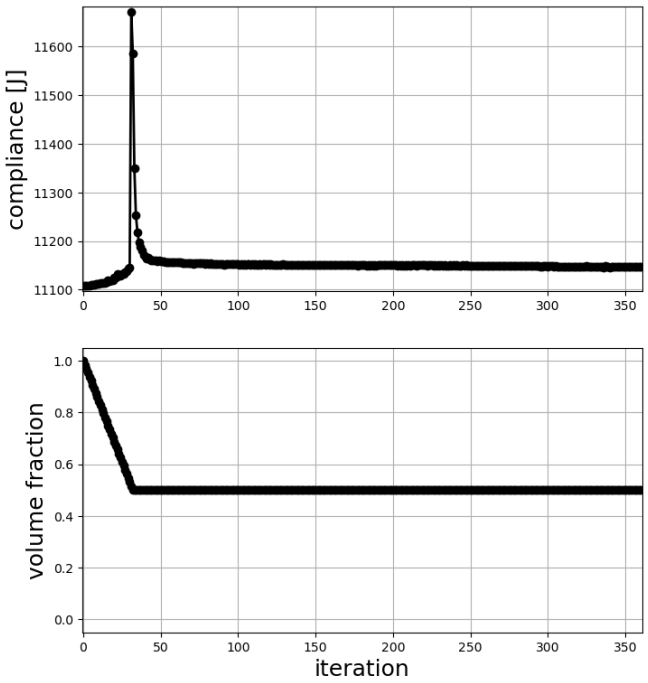

Figure 22: Objective and volume functions in the BESO-optimization (case with 360 iterations)

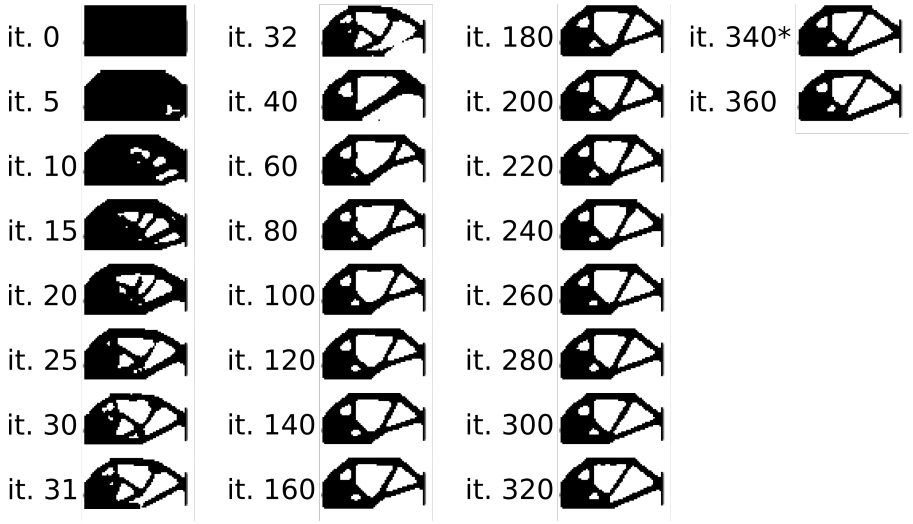

Figure 23: Topologies in the BESO-optimization (case with 360 iterations)

In this case, the abrupt change in the topology was not enough to change the local minimum that was being pursued. So the algorithm slowly recovered the broken component throughout the following iterations. By design, this method can remove material from the structure much more easily than it can add material back. Moreover, in order to have more stable and robust procedures, small $TV^{(k)}$ values are used. This explains why over 300 iterations were needed to produce the optimized result.

Although it may be computationally expensive to perform so many iterations, in the SILP-BESO approach with the considered patience stopping criterion, a large number of iterations means that, after reaching the target volume fraction, the method consistently improved the result along all the optimization procedure.

For the SIMP-MMA approach, the case in which $p_{bc} = -0.0625\,m$, $r_{bc} = 0.34375\,m$, $p_{ld} = -0.296875\,m$ and $r_{ld} = 0.1875\,m$ needed 136 iterations to converge. After convergence, one extra iteration is included, corresponding to the discretization of the solution. Figure 24 presents the evolution of the objective, volume and gray level functions over the optimization and Figure 25 presents the topology for some iterations. The

optimized continuous topology is marked with an asterisk (iteration 136*) and the discretized solution is marked with two asterisks (iteration 137**).

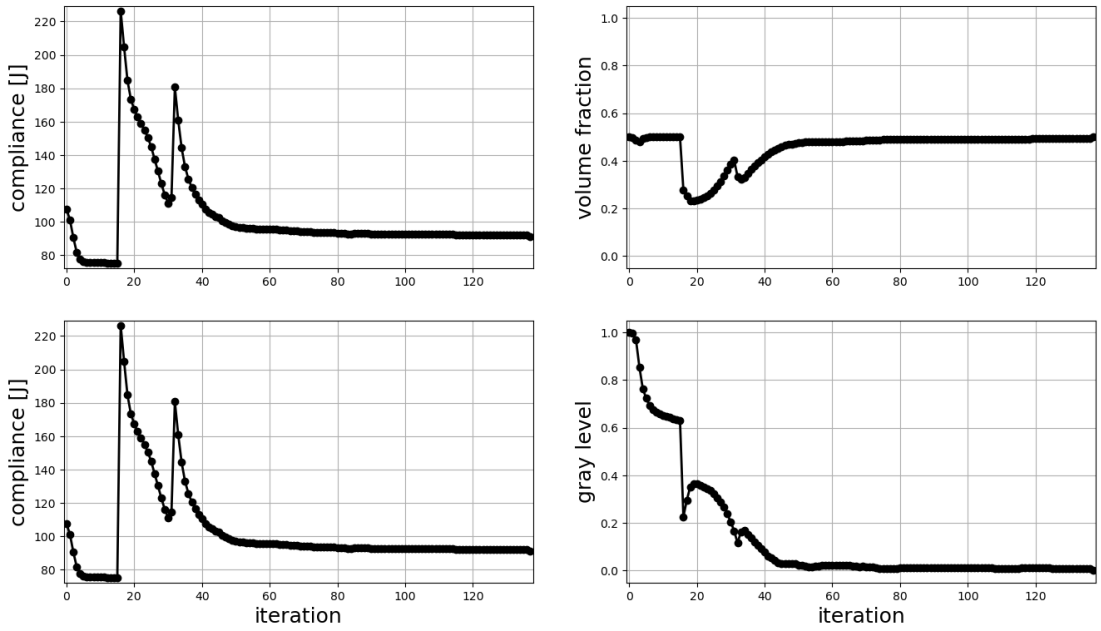

Figure 24: Objective, volume and gray level functions in the SIMP-optimization (case with 137 iterations)

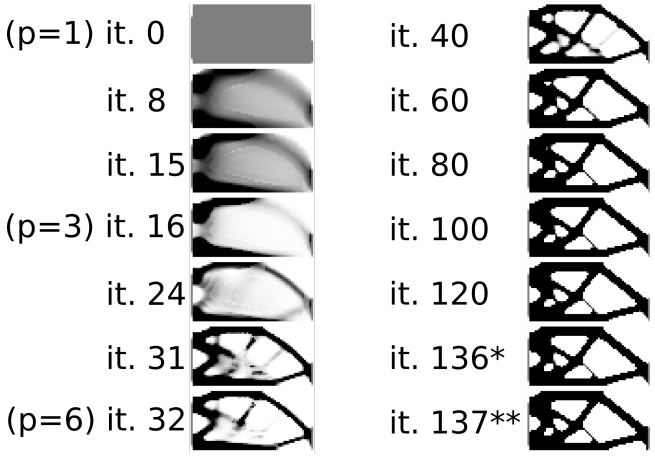

Figure 25: Topologies in the SIMP-optimization (case with 137 iterations)

It can be observed that the topology from the 80-th iteration was already very close to the solution, the method took over 50 iterations making only minor improvements. Although the convergence could be accelerated by tuning some parameters of the method, individual tuning is impracticable for the generation of such large datasets. Nonetheless, stability was considered to be more important than processing costs, therefore, these stable cases in which a large number of iterations performed are satisfactory results.

On the other hand, there are cases in which a small number of iterations is enough to achieve the stopping criteria, cases with minimal number of iterations are presented below.

For the SILP-BESO approach, the case in which $p_{bc} = -0.078125\,m$, $r_{bc} = 0.3125\,m$, $p_{ld} = 0.234375\,m$ and $r_{ld} = 0.09375\,m$ needed 52 iterations to converge. Figure 26 presents the evolution of the objective and volume functions over the optimization and Figure 27 presents the topology for some iterations. The optimized topology is marked with an asterisk (iteration 32*).

In the presented implementation, 52 iterations corresponds to the minimal possible number of iterations for convergence. It means that the topology with the lowest compliance value was achieved at the 32-th iteration, which is the first iteration in which the structure has the target volume of material. As it can be seen, the quick convergence occur because an important structural component break apart right after the target volume

is achieved.

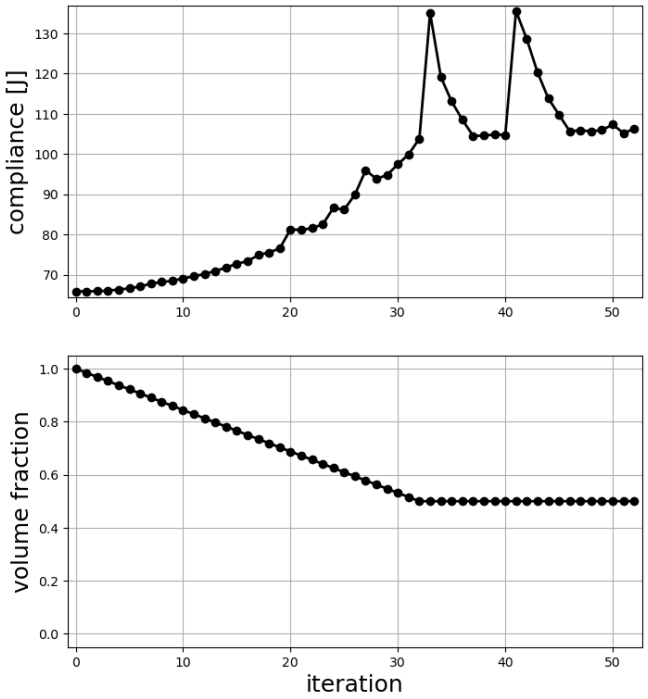

Figure 26: Objective and volume functions in the BESO-optimization (case with 52 iterations)

Figure 27: Topologies in the BESO-optimization (case with 52 iterations)

This behavior is undesirable, since the program is prevented from properly exploring the domain. The algorithm is stopped by the patience criterion, which is based solely on the objective function values. In future versions, this stopping criterion may be improved to avoid such early convergences. Nonetheless, the obtained result is reasonable and suitable for the proposed dataset.

For the SIMP-MMA approach, the case in which $p_{bc} = -0.109375\,m$, $r_{bc} = 0.1875\,m$, $p_{ld} = 0.5\,m$ and $r_{ld} = 0.0\,m$ needed 51 iterations to converge. After convergence, one extra iteration is included, corresponding to the discretization of the solution. Figure 28 presents the evolution of the objective, volume and gray level functions over the optimization and Figure 29 presents the topology for some iterations. The optimized continuous topology is marked with an asterisk (iteration 51^{*}) and the discretized solution is marked with two asterisks (iteration 52^{**}).

It can be observed that the topology from the 32-th iteration was already very close to the solution. From this topology, only 19 iterations of minor improvements were necessary to achieve the stopping criteria. In the SIMP-MMA approach, there is no issue with a quick convergence. It usually means that the considered case is well behaved and the method can progress without slowing down to avoid oscillations of the objective function.

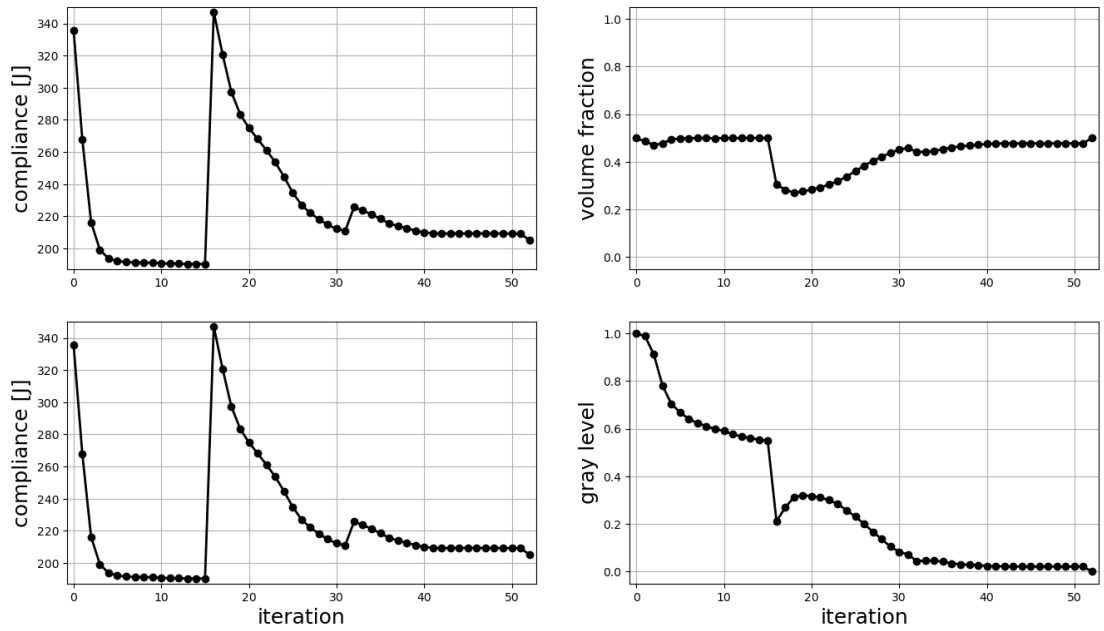

Figure 28: Objective, volume and gray level functions in the SIMP-optimization (case with 52 iterations)

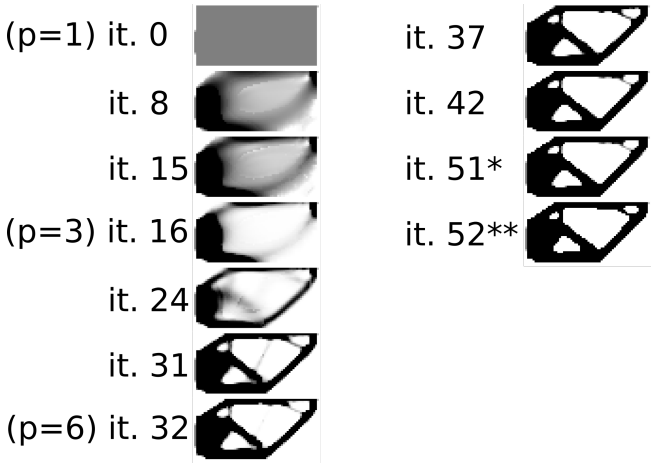

Figure 29: Topologies in the SIMP-optimization (case with 52 iterations)

5 Summary

The specified values for each fixed parameter of the considered problem are presented in Table 1. The SILP-BESO parameters are presented in Table 2 and the SIMP-MMA parameters are presented in Table 3.

Three values are shown for each SIMP-MMA parameter, this corresponds to the progression of the penalization exponent throughout the optimization procedure: 16 MMA outer iterations are performed with no penalization ($p = 1$) (with no inner MMA iterations); then, 16 iterations are performed with $p = 3$ (with no inner MMA iterations); lastly, the penalization is set to $p = 6$ and the method iterates until the stopping criteria are achieved (allowing up to 5 inner MMA iterations).

Table 1: General fixed parameters

Name (Documentation)	Name (Programs)	Value	Description
N_y	Ny	32	number of elements in y-axis
N_x	Nx	64	number of elements in x-axis
N	N	2048	total number of elements
L_y	Ly	1.0 m	cantilever height
L_x	Lx	2.0 m	cantilever length
e_s	esize	0.03125 m	side length of the square elements
E_y	Ey	1.0 Pa	Young's modulus
ν	nu	0.3	Poisson's coefficient
$r_{\max}$	rmax	0.125 m	sensitivity filter radius
ε_k	epsk	1×10^{-6}	soft-kill parameter
V^*	Vt	1024 (50%)	target volume, in number of elements
noptf	noptf	16	number of optimizations stored in the same file

Table 2: SILP-BESO parameters

Name (Documentation)	Name (Programs)	Value	Description
$VV^{(k)}$	VV	1.5625% (1/64)	maximal volume variation
$TV^{(k)}$	TV	3.1250% (1/32)	maximal topology variation
P	patience	20	patience stopping criterion
momentum	momentum	50%	sensitivity momentum

Table 3: SIMP-MMA parameters

Name (Documentation)	Name (Programs)	Value	Description
p	p	[1.0, 3.0, 6.0]	penalization exponents
meva	meva	[16, 16, 0]	maximal number of evaluations
miev	miev	[1, 1, 5]	maximal number of inner evaluations

The range of values for each input parameter is presented in Table 4. For the considered mesh, there are 528 unique possibilities for the pair (p_{bc}, r_{bc}); and there are 561 unique possibilities for the pair (p_{ld}, r_{ld}). Therefore, there are $296\,208 = 528 \times 561$ unique quadruplets $(p_{bc}, r_{bc}, p_{ld}, r_{ld})$. However, redundant cases are disregarded, so only 148 240 quadruplets are considered in the dataset generation.

Table 4: Input parameters

Name	Range of values	Description
p_{bc}	$-0.484375 \sim 0.484375\,m$	center position of the clamped region
r_{bc}	$0.031250 \sim 0.500000\,m$	half-length of the clamped region
p_{ld}	$-0.500000 \sim 0.500000\,m$	center position of the loaded region
r_{ld}	$0.000000 \sim 0.500000\,m$	half-length of the loaded region

The github repository (`https://github.com/Joquempo/Cantilever-Dataset`) is structured as presented in Figure 30. The "sample" folder contains scripts for taking samples from the generated datasets. The "source" folder contains the scripts that generate the datasets. The "validation" folder contains scripts used to validate the presented implementations. The "CITEAS" file lists indicated references for citing this work. The "LICENSE" file presents the terms of the GNU General Public License. The "README.md" file contain some key points from this documentation. And the "documentation.pdf" file corresponds to this document itself.

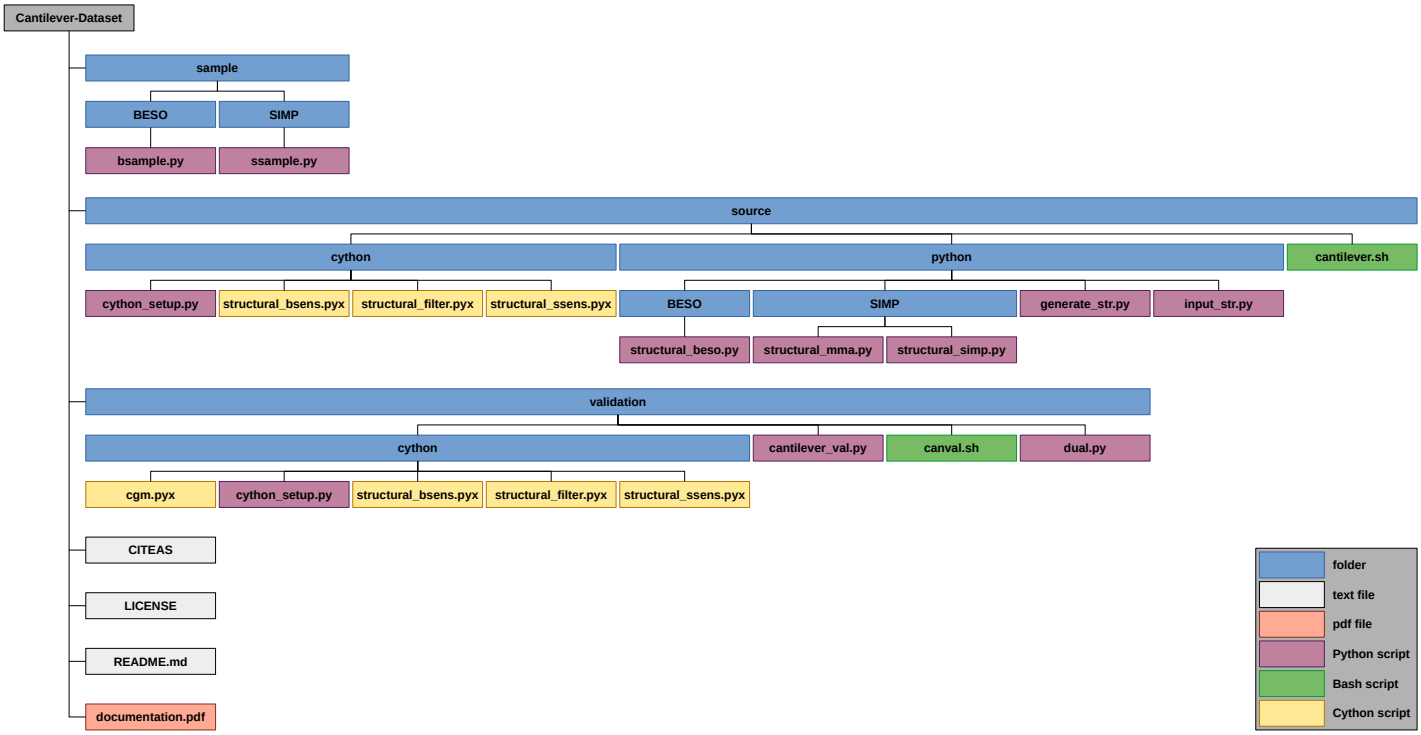

Figure 30: Cantilever-Dataset repository

Both "./source/python/BESO/structural_beso.py" and "./source/python/SIMP/structural_simp.py" scripts should be edited before executed, in order to set appropriate values for the parameters "fid_ini" and "fid_lim". For example, if "fid_ini=0" and "fid_lim=1600", the program will run the first 1600 cases, $\{0, 1, \ldots, 1599\}$; if "fid_ini=1600" and "fid_lim=6400", it will run the subsequent 4800 cases, $\{1600, 1601, \ldots, 6399\}$.

After installing and activating Anaconda, both BESO and SIMP datasets can be generated by executing the scripts ordered in the execution tree shown in Figure 31. The "Do" column specifies what is done by each script; the "Use" column specifies data and other scripts used during execution; the "Generate" column specifies what each script generates after being executed.

After generating the datasets, samples can be taken by using the scripts from the "sample" folder. To select which samples should be taken, the "./sample/BESO/bsample.py" and "./sample/SIMP/ssample.py" scripts must be edited. The indices of the files must be set, as well as the flags used to select what should be plotted. Subfolders will be created to store the generated figures.

Independently, the validation routine can be executed by using the scripts from the "validation" folder. The bash script "./validation/canval.sh" can be executed to setup the conda environment for the validation procedure. Then, by activating the created "canval" environment, the "./validation/cantilever_val.py" script can be executed to perform the procedure. Results will be printed to the terminal. An IDE is recommended in order to visualize the plots of the filtered sensitivity maps (otherwise, the script should be altered so the figures are not immediately closed after concluding the validation procedure).

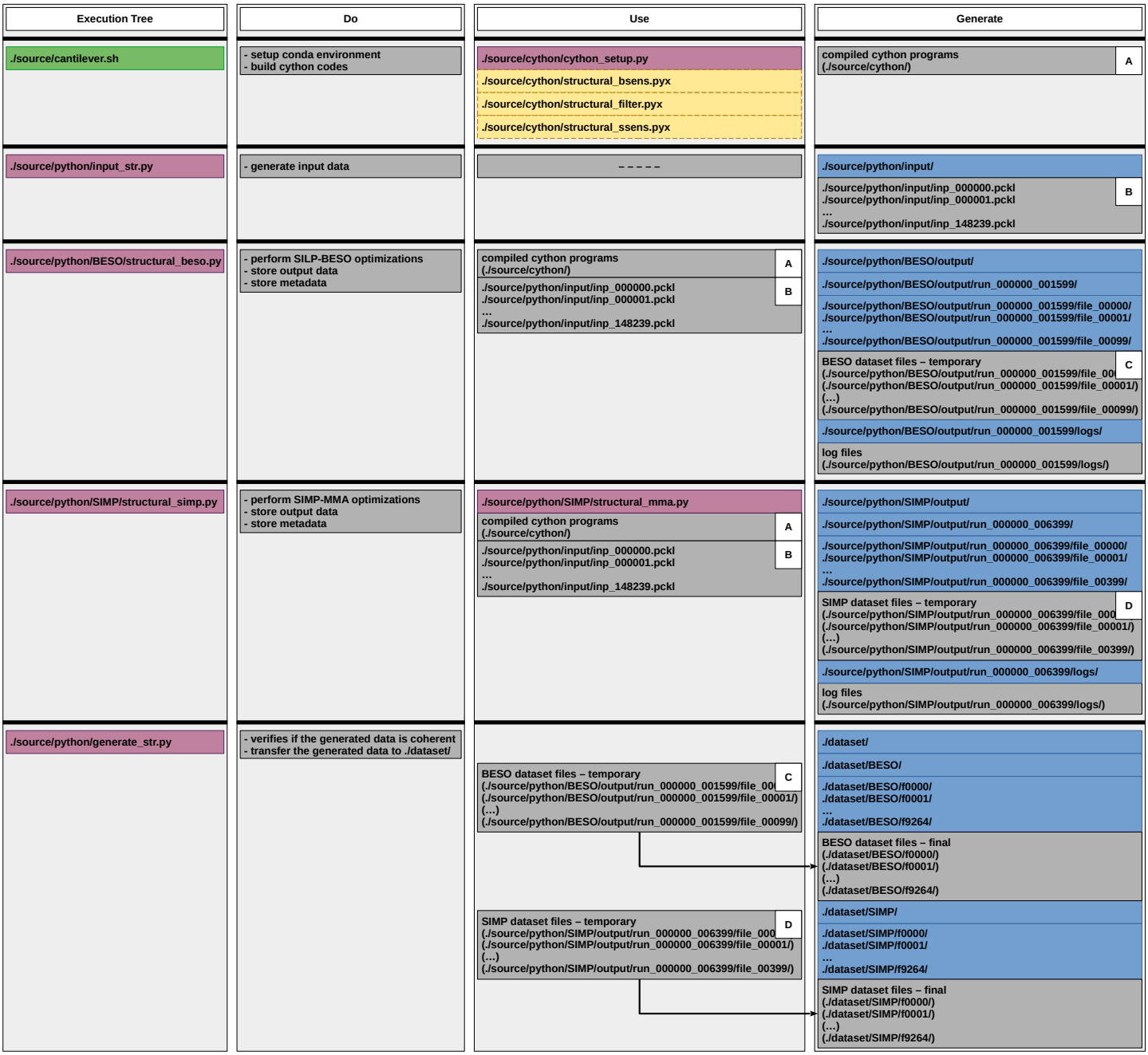

Figure 31: Execution tree

Table 5 presents the BESO dataset files. The size shown in the second column corresponds to the amount of the useful data, considering the average value of 82 iterations for the optimization procedure. The size shown in the third column corresponds to minimal and maximal observed values for the disk usage. Each of these files contains data from 16 optimization processes, so 9 265 of each one is generated after running all the 148 240 cases.

Table 6 presents the SIMP dataset files. The size shown in the second column corresponds to the amount of the useful data, considering the average value of 72 iterations for the optimization procedure. The size shown in the third column corresponds to minimal and maximal observed values for the disk usage. Each of these files contains data from 16 optimization processes, so 9 265 of each one is generated after running all the 148 240 cases.

Table 5: BESO dataset files

Name	Size (useful data)	Size (observed disk usage)	Description
dis.npy	21.8 MB	16.7 ∼ 35.6 MB	displacements vectors
fid.npy	64 bytes	4.1 ∼ 4.1 kB	input files indices
inp.npy	256 bytes	4.1 ∼ 4.1 kB	input data
obj.npy	5.2 kB	4.1 ∼ 12.3 kB	objective function values
obj_opt.npy	64 bytes	4.1 ∼ 4.1 kB	optimized objective function values
ptr2inp.npy	5.2 kB	4.1 ∼ 12.3 kB	pointers from iterations to inputs
ptr2opt.npy	68 bytes	4.1 ∼ 4.1 kB	pointers from inputs to iterations
sen_0.npy	10.4 MB	8.0 ∼ 17.0 MB	CGS-0 approximations for the sensitivity vectors
sen_1.npy	10.4 MB	8.0 ∼ 17.0 MB	CGS-1 approximations for the sensitivity vectors
sen_2.npy	10.4 MB	8.0 ∼ 17.0 MB	CGS-2 approximations for the sensitivity vectors
sen_w.npy	10.4 MB	8.0 ∼ 17.0 MB	exact sensitivity vectors
tim.npy	896 bytes	4.1 ∼ 4.1 kB	execution times and number of iterations
top.npy	332.0 kB	254.0 ∼ 532.5 kB	topology vectors
top_opt.npy	4.0 kB	8.2 ∼ 8.2 kB	optimized topologies
vol.npy	5.2 kB	4.1 ∼ 12.3 kB	relative volume values

Table 6: SIMP dataset files

Name	Size (useful data)	Size (observed disk usage)	Description
dis.npy	19.1 MB	17.1 ∼ 24.5 MB	displacements vectors
dobj_opt.npy	64 bytes	4.1 ∼ 4.1 kB	objective values of the discretized solutions
dtop_opt.npy	4.0 kB	8.2 ∼ 8.2 kB	discretized optimized topologies
fid.npy	64 bytes	4.1 ∼ 4.1 kB	input files indices
gra_p1.npy	4.6 kB	8.2 ∼ 8.2 kB	gray level values
inp.npy	256 bytes	4.1 ∼ 4.1 kB	input data
obj.npy	4.6 kB	8.2 ∼ 8.2 kB	objective function values
obj_opt.npy	64 bytes	4.1 ∼ 4.1 kB	optimized objective function values
ptr2inp.npy	4.6 kB	8.2 ∼ 8.2 kB	pointers from iterations to inputs
ptr2opt.npy	68 bytes	4.1 ∼ 4.1 kB	pointers from inputs to iterations
sen_p1.npy	9.1 MB	8.2 ∼ 11.7 MB	sensitivity vectors
tim.npy	576 bytes	4.1 ∼ 4.1 kB	execution times and number of iterations
top_p1.npy	9.1 MB	8.2 ∼ 11.7 MB	topology vectors
top_p1_opt.npy	128.0 kB	135.2 ∼ 135.2 kB	optimized topologies
vol_p1.npy	4.6 kB	8.2 ∼ 8.2 kB	relative volume values

Table 7 presents the observed disk usage for each dataset, as well as the total value.

Table 7: Disk usage

	Observed disk usage
BESO dataset	617 GB
SIMP dataset	363 GB
total	**980 GB**

Around 600 MB of disk is used to store input data in the "source" folder. It should be noted that these input files can be deleted after concluding the generation of the datasets. More importantly, it should be noted that careless sampling can quickly write a lot of data in your disk, so be cautious when running the sampling scripts.

Since all data is stored as numpy arrays, through "numpy.save(·)", everything can be easily read through "numpy.load(·)". In anyway, the sampling scripts illustrate how to read data from the generated datasets.

6 Acknowledgements

This work was supported by the São Paulo Research Foundation (FAPESP), grant numbers: 2013/08293-7 and 2019/19237-7.

Bibliography

[1] Krister Svanberg and Mats Werme. A hierarchical neighbourhood search method for topology optimization. *Structural and Multidisciplinary Optimization*, 29(5):325–340, 2005.

[2] Krister Svanberg and Mats Werme. Topology optimization by a neighbourhood search method based on efficient sensitivity calculations. *International journal for numerical methods in engineering*, 67(12):1670–1699, 2006.

[3] Daniel Candeloro Cunha, Breno Vincenzo de Almeida, Heitor Nigro Lopes, and Renato Pavanello. Finite variation sensitivity analysis for discrete topology optimization of continuum structures. *Structural and Multidisciplinary Optimization*, 64(6):3877–3909, 2021.

[4] Yi M Xie and Grant P Steven. A simple evolutionary procedure for structural optimization. *Computers & structures*, 49(5):885–896, 1993.

[5] Osvaldo M Querin, Grant P Steven, and Yi Min Xie. Evolutionary structural optimisation (eso) using a bidirectional algorithm. *Engineering computations*, 1998.

[6] Xiaodong Huang and Yi-Min Xie. A further review of eso type methods for topology optimization. *Structural and Multidisciplinary Optimization*, 41(5):671–683, 2010.

[7] Ole Sigmund and Joakim Petersson. Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Structural optimization*, 16(1):68–75, 1998.

[8] Q Li, GP Steven, and YM Xie. A simple checkerboard suppression algorithm for evolutionary structural optimization. *Structural and Multidisciplinary Optimization*, 22(3):230–239, 2001.

[9] Xiaodong Huang and YM Xie. Convergent and mesh-independent solutions for the bi-directional evolutionary structural optimization method. *Finite elements in analysis and design*, 43(14):1039–1049, 2007.

[10] EL Zhou, Yi Wu, XY Lin, et al. A normalization strategy for beso-based structural optimization and its application to frequency response suppression. *Acta Mechanica*, 232(4):1307–1327, 2021.

[11] Martin P Bendsøe. Optimal shape design as a material distribution problem. *Structural optimization*, 1(4):193–202, 1989.

[12] Martin P Bendsøe and Ole Sigmund. Material interpolation schemes in topology optimization. *Archive of applied mechanics*, 69(9):635–654, 1999.

[13] A Rietz. Sufficiency of a finite exponent in simp (power law) methods. *Structural and Multidisciplinary Optimization*, 21(2):159–163, 2001.

[14] JM Martinez. A note on the theoretical convergence properties of the simp method. *Structural and Multidisciplinary Optimization*, 29(4):319–323, 2005.

[15] Krister Svanberg. The method of moving asymptotes—a new method for structural optimization. *International journal for numerical methods in engineering*, 24(2):359–373, 1987.

[16] Krister Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM journal on optimization*, 12(2):555–573, 2002.

[17] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[18] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[19] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[20] Stefan Behnel, Robert Bradshaw, Craig Citro, et al. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[21] David Cournapeau, Nathaniel Smith, Dag Sverre Seljebotn, et al. The scikit-sparse package for sparse matrix manipulation. https://github.com/scikit-sparse/scikit-sparse.

[22] Steven G Johnson. The nlopt nonlinear-optimization package. http://github.com/stevengj/nlopt.

[23] G. N. Wells et al A. Logg, K.-A. Mardal. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[24] J. Hake A. Johansson B. Kehlet A. Logg C. Richardson J. Ring M. E. Rognes M. S. Alnaes, J. Blechta and G. N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3, 2015.