Dataset Documentation

Topology Optimization of the Base Cell of a Periodic Metamaterial

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 2023

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 a dataset 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 the base cell of a periodic metamaterial is considered. Two free parameters are used to define the inverse homogenization problem: the target Poisson's ratio and the minimal Young's modulus for the homogenized metamaterial. The dataset is generated by performing 18 382 optimizations, considering unique sets of these parameters.

All optimizations are performed through Sequential Integer Linear Programming (SILP). For each iteration of each case, all results are stored: topology vectors; sensitivity vectors; displacements vectors; homogenized Poisson's ratio values; homogenized Young's modulus values; volume fraction values. Also, metadata is stored with relevant information, for example, the corresponding input parameters of each result. This dataset occupies around 277 GB of disk.

- To collaborate or report bugs, please look for the author's email address at:
  `https://www.fem.unicamp.br/~ltm/`
- All codes and documentation are publicly available in the following github repository:
  `https://github.com/Joquempo/Metamaterial-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 inverse homogenization problem is described. Then, the considered topology optimization method is presented.

In section 3, a user guide is presented, explaining how to use the provided programs to generate the dataset. Then, each script is presented and briefly explained. Some validation procedures are described, the corresponding scripts are available in the github repository, but they are not shown in this document. Lastly, an unfixed bug, found late in development, is described (it is reported in the "Issues" tab of the github repository).

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

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 dataset; the data stored in the dataset.

2 Topology Optimization

2.1 Problem Description

The problem of designing the microstructure of a bidimensional isotropic mechanical metamaterial, in plane stress state, is considered. It corresponds to an inverse homogenization problem, in which we search for a topology of the microstructure that yields specified mechanical properties for the homogenized metameterial. A hexagonal base cell with dihedral D_3 symmetry is used, which is a sufficient condition to obtain isotropic homogenized properties [1, 2, 3, 4]. Figure 1 presents the base cell in the coordinate system (s_x, s_y).

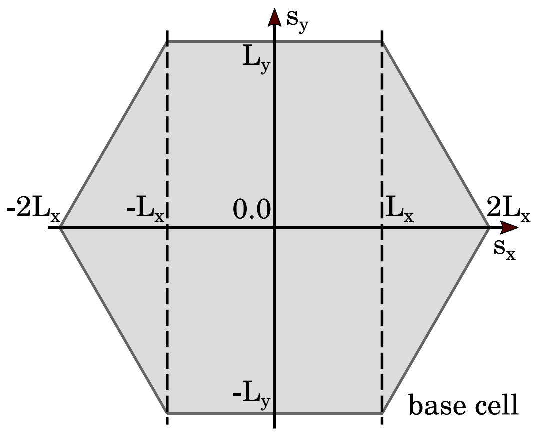

Figure 1: Coordinate system

The inverse homogenization problem can be solved through topology optimization. Since the optimization does not depend on the area of the base cell (V_Ω), or on the scale of the Young's modulus of the base material ($\breve{E}$), unitary values are considered for both of them ($V_\Omega = 1.0\,m^2$ and $\breve{E} = 1.0\,Pa$). Thus, $L_x = \left[\frac{1}{108}\right]^{\frac{1}{4}} m \approx 0.31\,m$ and $L_y = \left[\frac{1}{12}\right]^{\frac{1}{4}} m \approx 0.54\,m$.

Figure 2 presents the design domain. The six symmetries are indicated: the three rotations R_1, R_2 and R_3; and the three reflections M_1, M_2 and M_3. The material distribution over the design domain determines the material distribution over the symmetric subdomains, through rotations and reflections.

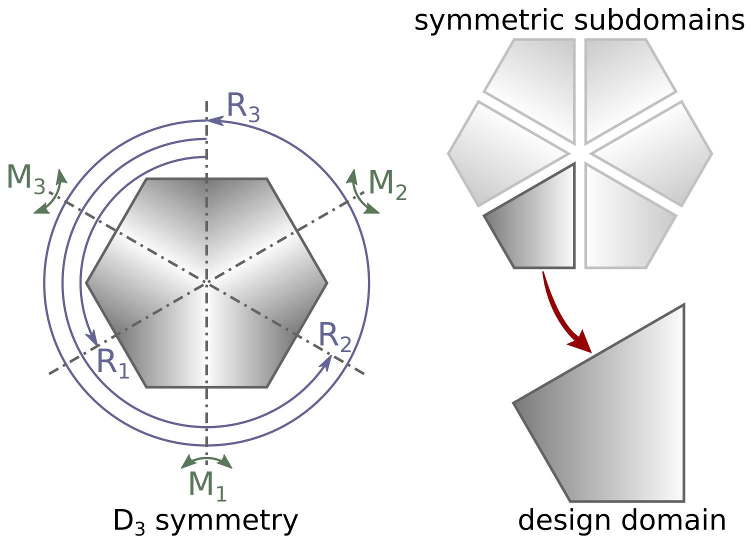

Figure 2: Design domain

For a fixed homogeneous and isotropic base material, of Poisson's ratio $\breve{\nu} = 0.3$ and Young's modulus $\breve{E} = 1.0\,Pa$, the considered problem consists in finding a distribution of the base material over the design domain that yields a specified value (ν^*) for the homogenized Poisson's ratio ($\widehat{\nu}$), while respecting a constraint of minimal homogenized Young's modulus ($\widehat{E} \geqslant E_{\min}$).

In order to express this as an optimization problem, the structure is discretized in a mesh of finite elements, then a computational homogenization [5, 6] is performed to obtain the required functions. The discretized structure is defined by the number of elements in each direction of the design domain, N_s. All elements have the same area, the length of their shorter sides is $e_x = \frac{L_x}{N_s}$ and the length of their longer sides is $e_y = \frac{L_y}{N_s}$. The number of design variables is given by $N_d = {N_s}^2$, and the number of quadrilateral finite elements in the cell is given by $N_t = 6\,N_d$.

The design variables compose a density vector that fully describes the topology of the structure: $\boldsymbol{x} \in \{0,1\}^{N_d}$. Each design variable defines the material distribution in six elements of the symmetric cell: when $x_i = 1$, all corresponding elements are solid and have the same stiffness of the base material; when $x_i = 0$, all corresponding elements are void and have nearly zero stiffness. The design variables, arranged in a matrix with proper neighborhood relations, are numbered from the leftmost column to the rightmost column and, in each column, they are numbered from bottom to top. The elements of the discretized cell are numbered from the center of the hexagon to the outer edges, following an anticlockwise spiral, as illustrated in Figure 3 for $N_s = 3$.

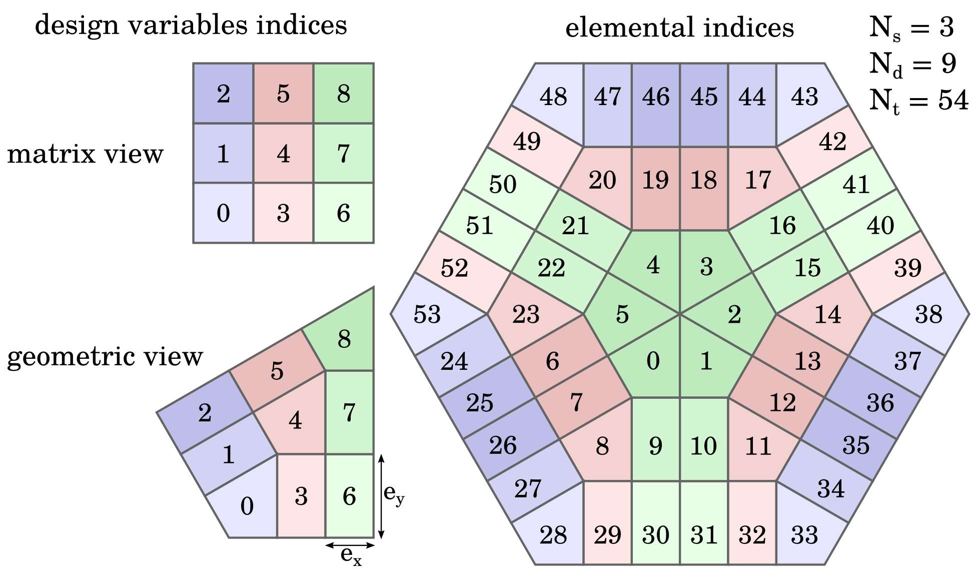

Figure 3: Elemental indices of the discretized cell

Figure 4 shows the nodal indices, which follow this same numbering rule.

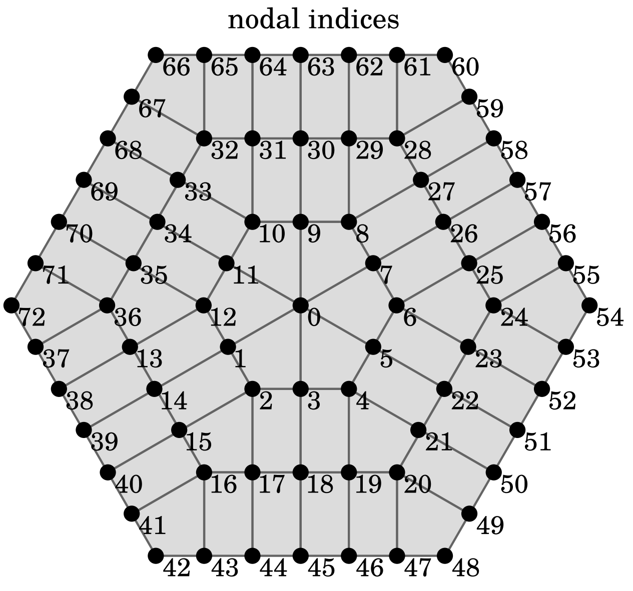

Figure 4: Nodal indices of the discretized cell

Bilinear quadrilateral elements in plane stress state are considered in the Finite Element Analysis (FEA). Small displacements and strains are considered, so linear assumptions are adopted. For each element, the local nodal indices are defined as shown in Figure 5, they are numbered anticlockwise. There are twelve different types of quadrilateral elements, according to their shape and rotation. Although some of them have the same stiffness matrix, the twelve elemental stiffness matrices are independently computed, considering that the elements are solid, through Gaussian quadrature with 2×2 points.

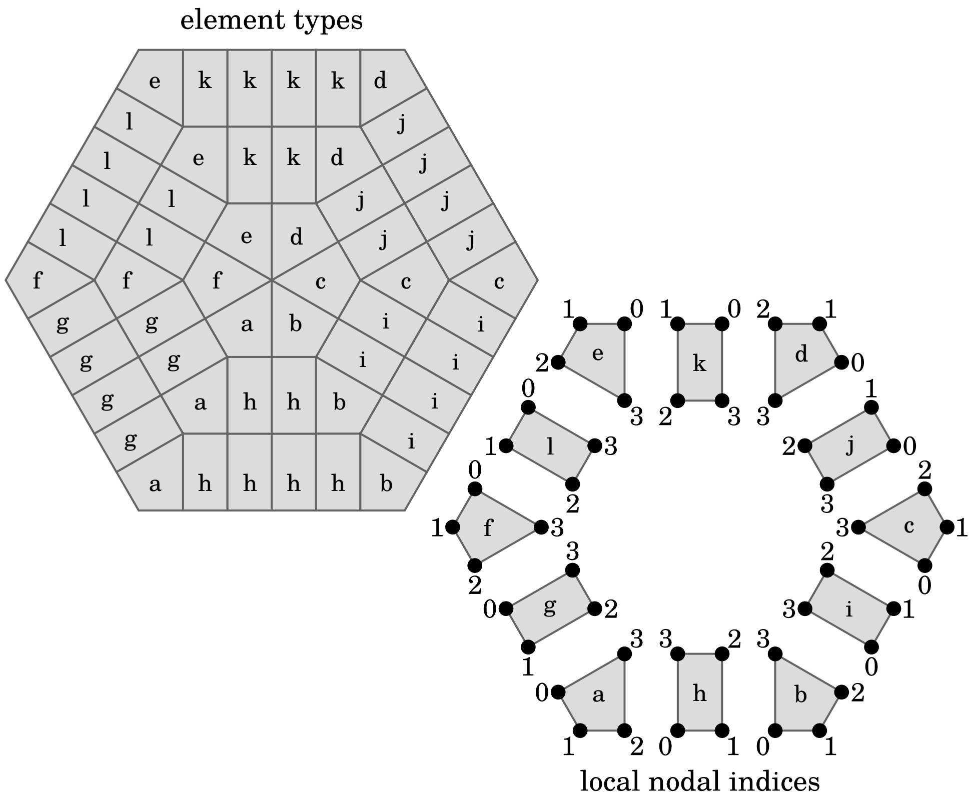

Figure 5: Local nodal indices of the bilinear quadrilateral elements

For simplicity, each sextuplet of symmetric elements (which share the same density value) can be understood as a single augmented element. Thus, for $x_i = 1$, the stiffness matrix of the ith augmented element is given by

$$\boldsymbol{K_i}^{[\mathrm{aug}]} = \sum_{w=1}^{6} \boldsymbol{K_i}^{[\boldsymbol{w}]}, \tag{1}$$

where $\boldsymbol{K_i}^{[\boldsymbol{w}]}$ is the stiffness matrix of the wth quadrilateral element corresponding to the ith design variable. The matrices of the quadrilateral elements are defined in the global system, so they assume zero values everywhere outside a small submatrix of dimensions 8×8, and this assembly can be performed as a simple summation.

A soft-kill approach is adopted to prevent singularities throughout the optimization procedure, so a small stiffness is assigned to void elements according to a small soft-kill parameter (p_k). This means that the following base stiffness is assigned to the whole structure:

$$\boldsymbol{K_0} = p_k \sum_{i=1}^{N_d} \boldsymbol{K_i}^{[\mathrm{aug}]}. \tag{2}$$

The elemental variation matrix, corresponding to the variation applied to the global stiffness matrix when the state of the ith augmented element is switched, is defined as

$$\boldsymbol{K_i} = (1 - p_k)\, \boldsymbol{K_i}^{[\mathrm{aug}]}. \tag{3}$$

Therefore, the global stiffness matrix of the base cell can be written as a function of $\boldsymbol{x}$:

$$\boldsymbol{K}(\boldsymbol{x}) = \boldsymbol{K_0} + \sum_{i=1}^{N_d} x_i\, \boldsymbol{K_i}. \tag{4}$$

To perform the homogenization procedure [5, 6], three fixed macro-displacements are imposed. For a compact notation, they are grouped in a matrix:

$$\widehat{\boldsymbol{U}} = \begin{bmatrix} \widehat{\boldsymbol{u}}_{\boldsymbol{xx}} & \widehat{\boldsymbol{u}}_{\boldsymbol{yy}} & \widehat{\boldsymbol{u}}_{\boldsymbol{xy}} \end{bmatrix} . \tag{5}$$

The total displacements is defined as

$$\boldsymbol{U}(\boldsymbol{x}) = \widehat{\boldsymbol{U}} + \widetilde{\boldsymbol{U}}(\boldsymbol{x}) , \tag{6}$$

where $\widetilde{\boldsymbol{U}}$ is a matrix storing the micro-displacement vectors. It can be written with respect to a reduced matrix $\breve{\boldsymbol{U}}$ as

$$\widetilde{\boldsymbol{U}}(\boldsymbol{x}) = \boldsymbol{P}\,\breve{\boldsymbol{U}}(\boldsymbol{x}) , \tag{7}$$

where $\boldsymbol{P}$ is a known constant matrix used to impose the required periodicity constraints over the base cell of the metamaterial.

The reduced stiffness matrix and the reduced term for the right-hand side can then be defined as

$$\breve{\boldsymbol{K}}(\boldsymbol{x}) = \boldsymbol{P}^T \boldsymbol{K}(\boldsymbol{x})\,\boldsymbol{P} \text{ and } \breve{\boldsymbol{F}}(\boldsymbol{x}) = -\boldsymbol{P}^T \boldsymbol{K}(\boldsymbol{x})\,\widehat{\boldsymbol{U}} , \tag{8}$$

resulting in a linear system with unique solution, from which $\breve{\boldsymbol{U}}$ can be obtained:

$$\breve{\boldsymbol{U}}(\boldsymbol{x}) = \left[\breve{\boldsymbol{K}}(\boldsymbol{x})\right]^{-1} \breve{\boldsymbol{F}}(\boldsymbol{x}) . \tag{9}$$

The elasticity matrix of the homogenized material is computed as follows:

$$\boldsymbol{C}(\boldsymbol{x}) = \begin{bmatrix} C_{00}(\boldsymbol{x}) & C_{01}(\boldsymbol{x}) & C_{02}(\boldsymbol{x}) \\ C_{10}(\boldsymbol{x}) & C_{11}(\boldsymbol{x}) & C_{12}(\boldsymbol{x}) \\ C_{20}(\boldsymbol{x}) & C_{21}(\boldsymbol{x}) & C_{22}(\boldsymbol{x}) \end{bmatrix} = \frac{1}{V_\Omega} \left[\boldsymbol{U}(\boldsymbol{x})\right]^T \boldsymbol{K}(\boldsymbol{x})\,\boldsymbol{U}(\boldsymbol{x}) \tag{10}$$

Since isotropy is guaranteed, it must result in the following matrix:

$$\boldsymbol{C}(\boldsymbol{x}) = \frac{\widehat{E}(\boldsymbol{x})}{1 - [\widehat{\nu}(\boldsymbol{x})]^2} \begin{bmatrix} 1 & \widehat{\nu}(\boldsymbol{x}) & 0 \\ \widehat{\nu}(\boldsymbol{x}) & 1 & 0 \\ 0 & 0 & \frac{1-\widehat{\nu}(\boldsymbol{x})}{2} . \end{bmatrix} \tag{11}$$

By comparing both expressions, the homogenized parameters can be obtained as

$$\widehat{\nu}(\boldsymbol{x}) = 1 - \frac{2\,C_{22}(\boldsymbol{x})}{C_{00}(\boldsymbol{x})} \text{ and } \widehat{E}(\boldsymbol{x}) = \frac{4\,C_{22}(\boldsymbol{x})\,[C_{00}(\boldsymbol{x}) - C_{22}(\boldsymbol{x})]}{C_{00}(\boldsymbol{x})} . \tag{12}$$

Besides the presented functions, volume and topology variation functions can be included in the problem in order to properly apply the considered optimization method. Since all elements have the same area, the number of solid augmented elements can be used as a volume function:

$$V(\boldsymbol{x}) = \|\boldsymbol{x}\|_1 = \sum_{i=1}^{N_d} x_i . \tag{13}$$

For a reference density vector $\bar{\boldsymbol{x}}$, the topology variation function can be defined as

$$D(\boldsymbol{x}) = \|\boldsymbol{\Delta x}\|_1 = \|\boldsymbol{x} - \bar{\boldsymbol{x}}\|_1 = \sum_{i=1}^{N_d} |x_i - \bar{x}_i| . \tag{14}$$

Both can be written as relative values, the volume fraction and topology variation fraction are given by:

$$V_f(\boldsymbol{x}) = \frac{V(\boldsymbol{x})}{N_d} \text{ and } D_f(\boldsymbol{x}) = \frac{D(\boldsymbol{x})}{N_d} . \tag{15}$$

Finally, for each pair of properties $(\nu^*, E_{\min})$, a topology optimization problem can be stated to solve the inverse homogenization problem.

$$\begin{gathered} \boldsymbol{x}^* = \arg\min\, [\widehat{\nu}(\boldsymbol{x}) - \nu^*]^2 \\ \text{subject to} \\ \widehat{E}(\boldsymbol{x}) \geqslant E_{\min} \end{gathered} \tag{16}$$

Each optimization problem is then solved through SILP.

2.2 Sequential Integer Linear Programming (SILP)

In this approach, all functions have to be linearized around the current topology $\bar{\mathbf{x}}$ [7, 8, 9, 6]. In order to extract the linear component from a given function of binary variables, it is firstly written in the form

$$f(\mathbf{x}) = \alpha^{\langle 0 \rangle} + \sum_{j=1}^{N_d} \boldsymbol{\alpha}^{\langle \boldsymbol{j} \rangle} (\cdot)^j (\mathbf{x} - \bar{\mathbf{x}})^j \,, \tag{17}$$

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

$$\boldsymbol{\alpha}^{\langle \boldsymbol{j} \rangle} (\cdot)^j (\mathbf{x} - \bar{\mathbf{x}})^j = \sum_{i_1=1}^{N_d} \sum_{i_2=1}^{N_d} \dots \sum_{i_j=1}^{N_d} \alpha^{\langle j \rangle}_{i_1\, i_2\, \dots\, i_j} \, (x_{i_1} - \bar{x}_{i_1}) \, (x_{i_2} - \bar{x}_{i_2}) \dots (x_{i_j} - \bar{x}_{i_j}) \,. \tag{18}$$

The scalar $\alpha^{\langle 0 \rangle}$ corresponds to $f(\bar{\mathbf{x}})$, $\boldsymbol{\alpha}^{\langle \boldsymbol{1} \rangle}$ is a vector of N_d entries corresponding to the variations of f 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 $f(\mathbf{x})$, denoted by $f^{\text{lin}}(\mathbf{x})$, is given by

$$f^{\text{lin}}(\mathbf{x}) = \alpha^{\langle 0 \rangle} + \boldsymbol{\alpha}^{\langle \boldsymbol{1} \rangle} \cdot [\mathbf{x} - \bar{\mathbf{x}}] = f(\bar{\boldsymbol{x}}) + \boldsymbol{\alpha}^{[\boldsymbol{f}]} \cdot [\boldsymbol{x} - \bar{\boldsymbol{x}}] \,. \tag{19}$$

The vector $\boldsymbol{\alpha}^{\langle \boldsymbol{1} \rangle}$ is referred to as $\boldsymbol{\alpha}^{[\boldsymbol{f}]}$. It can be understood as the sensitivity of the linearized function with respect to each design variable, its entries are given by

$$\alpha_i^{[f]} = f(\bar{\mathbf{x}}, x_i = 1) - f(\bar{\mathbf{x}}, x_i = 0) \,, \tag{20}$$

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.

The considered approach consists in solving a sequence of linearized subproblems, using the branch-and-bound method coupled with simplex algorithm to solve each linear integer subproblem [10, 11]. In the special case in which the Young's modulus is unconstrained ($E_{\min} = 0.0\, Pa$), the BESO algorithm [12, 13] can be used instead, which is a much more efficient approach for problems with simple constraints.

The functions V_f and D_f are additively separable functions of binary variables, which means that they are already linear functions ($V_f = V_f^{\text{lin}}$ and $D_f = D_f^{\text{lin}}$) whose sensitivity values are given by

$$\alpha_i^{[V_f]} = \frac{1}{N_d} \quad \text{and} \quad \alpha_i^{[D_f]} = \frac{1 - 2\,\bar{x}_i}{N_d} \,. \tag{21}$$

On the other hand, the considered objective function, $h_\nu(\boldsymbol{x}) = [\widehat{\nu}(\boldsymbol{x}) - \nu^*]^2$, and constraint function, $g_E(\boldsymbol{x}) = E_{\min} - \widehat{E}(\boldsymbol{x}) \leqslant 0$, are nonlinear. So their linearized versions should only be used as approximations for topologies reasonably close to $\bar{\boldsymbol{x}}$.

In order to improve the accuracy of the linearizations throughout the optimization procedure, a maximal topology variation ($D_{\max}$) is imposed in each iteration. An auxiliary parameter (η_E) is included to limit the variation of $\widehat{E}$. Moreover, a volume penalization parameter (β) is used to inhibit the appearance of disconnected solid islands in the structure.

Thus, to obtain the next topology of the iterative procedure ($\bar{\boldsymbol{x}}^{(k+1)}$), the functions are linearized around the current one ($\bar{\boldsymbol{x}}^{(k)}$) and the following subproblem is solved:

$$\begin{gathered}\bar{\boldsymbol{x}}^{(k+1)} = \arg\min\ h_\nu^{\text{lin}}(\boldsymbol{x}) + \beta\, V_f(\boldsymbol{x}) \\ \text{subject to} \\ E_{\min} + \eta_E - \widehat{E}^{\text{lin}}(\boldsymbol{x}) \leqslant 0 \\ D_f(\boldsymbol{x}) \leqslant D_{\max}\end{gathered} \tag{22}$$

The heuristic SILP approach consists in exploring the domain of feasible topologies by successively solving these linearized subproblems, starting from a given initial topology $\bar{\boldsymbol{x}}^{(0)}$. The stopping criterion is given in terms of a patience parameter (P): 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.

Usually, the most costly task of this algorithm is to perform the sensitivity analysis, that is, to compute $\boldsymbol{\alpha}^{[\boldsymbol{h_\nu}]}$ and $\boldsymbol{\alpha}^{[\boldsymbol{E}]}$. It is performed as follows.

The direct homogenization procedure is performed for the current topology ($\bar{\boldsymbol{x}}$), so all current matrices $\boldsymbol{K}$, $\breve{\boldsymbol{U}}$, $\boldsymbol{U}$ and $\boldsymbol{C}$ are known, as well as the scalars $\breve{\nu}$ and $\breve{E}$. For a compact notation, the parameters $\bar{\gamma}$ and $\bar{\bar{\gamma}}$ are defined as

$$\bar{\gamma} = \frac{C_{22}}{C_{00}} \text{ and } \bar{\bar{\gamma}} = \frac{C_{22} + \Delta C_{22}}{C_{00} + \Delta C_{00}}\,, \tag{23}$$

where ΔC_{00} and ΔC_{22} are diagonal terms of the matrix $\boldsymbol{\Delta C}$, which corresponds to the variation of $\boldsymbol{C}$ when the state of the ith augmented is switched. Then, the variation of each elasticity property can be obtained as

$$\Delta\widehat{\nu} = \frac{2\,[\bar{\gamma}\,\Delta C_{00} - \Delta C_{22}]}{C_{00} + \Delta C_{00}} \text{ and } \Delta\widehat{E} = 4\,[1 - \bar{\bar{\gamma}}]\,\Delta C_{22} + 2\,C_{22}\,\Delta\widehat{\nu}\,. \tag{24}$$

Thus, the sensitivity values are obtained as

$$\alpha_i^{[h_\nu]} = \begin{cases} \;\;\,2\,[\widehat{\nu} - \nu^*]\,\Delta\widehat{\nu} + \Delta\widehat{\nu}^2\,, & \text{if } \bar{x}_i = 0\,, \\ -\,2\,[\widehat{\nu} - \nu^*]\,\Delta\widehat{\nu} + \Delta\widehat{\nu}^2\,, & \text{if } \bar{x}_i = 1\,, \end{cases} \tag{25}$$

and

$$\alpha_i^{[E]} = \begin{cases} \;\;\,\Delta\widehat{E}\,, & \text{if } \bar{x}_i = 0\,, \\ -\,\Delta\widehat{E}\,, & \text{if } \bar{x}_i = 1\,, \end{cases} \tag{26}$$

The Woodbury Sensitivity (WS) approach [9, 6] can be used to compute the exact $\boldsymbol{\Delta C}$ for each augmented element. Firstly, the corresponding elemental variation matrix is factorized as

$$\boldsymbol{K_i} = \boldsymbol{H_i}\,\boldsymbol{H_i}^T\,, \tag{27}$$

where $\boldsymbol{H_i}$ is a rectangular matrix with 30 columns at most, and with no more than 48 nonzero rows. Constrained matrices can be defined as

$$\breve{\boldsymbol{H}}_{\boldsymbol{i}} = \boldsymbol{P}^T\boldsymbol{H_i} \text{ and } \breve{\boldsymbol{K}}_{\boldsymbol{i}} = \boldsymbol{P}^T\boldsymbol{K_i}\boldsymbol{P} = \breve{\boldsymbol{H}}_{\boldsymbol{i}}\,\breve{\boldsymbol{H}}_{\boldsymbol{i}}^T\,. \tag{28}$$

These are used to define the matrices

$$\boldsymbol{A_i} = \breve{\boldsymbol{H}}_{\boldsymbol{i}}^T\breve{\boldsymbol{K}}^{-1}\breve{\boldsymbol{H}}_{\boldsymbol{i}} \text{ and } \boldsymbol{V_i} = \boldsymbol{H_i}^T\boldsymbol{U} = \boldsymbol{H_i}^T\widehat{\boldsymbol{U}} + \breve{\boldsymbol{H}}_{\boldsymbol{i}}^T\breve{\boldsymbol{U}}\,. \tag{29}$$

Once $\boldsymbol{A_i}$ and $\boldsymbol{V_i}$ are known, the variation of $\boldsymbol{C}$ can be obtained as

$$\boldsymbol{\Delta C} = \begin{cases} \dfrac{1}{V_\Omega}\, \boldsymbol{V_i}^T \left[\boldsymbol{I} + \boldsymbol{A_i}\right]^{-1} \boldsymbol{V_i}\,, & \text{if } \bar{x}_i = 0\,, \\ -\dfrac{1}{V_\Omega}\, \boldsymbol{V_i}^T \left[\boldsymbol{I} - \boldsymbol{A_i}\right]^{-1} \boldsymbol{V_i}\,, & \text{if } \bar{x}_i = 1\,. \end{cases} \tag{30}$$

Since the dimensions of $\boldsymbol{A_i}$ are at most 30×30, the inverse of $[\boldsymbol{I} \pm \boldsymbol{A_i}]$ can be easily computed. The Cholesky factorization is used to solve the linear systems from the homogenization procedure. Although this factorization can be reused to compute $\breve{\boldsymbol{K}}^{-1}\breve{\boldsymbol{H}}_{\boldsymbol{i}}$, this has to be performed for each of the N_d augmented elements, thus, the computation of the exact sensitivity vectors is an expensive task.

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

$$\alpha_i^{[h_\nu]} \approx 4\left[\hat{\nu} - \nu^*\right]\left[\frac{\bar{\gamma}\, C_{00}^\delta - C_{22}^\delta}{C_{00}}\right] \tag{31}$$

and

$$\alpha_i^{[E]} \approx 4\left[\bar{\gamma}^2\, C_{00}^\delta + \hat{\nu}\, C_{22}^\delta\right]\,, \tag{32}$$

where C_{00}^δ and C_{22}^δ are diagonal terms of the matrix

$$\boldsymbol{C^\delta} = \frac{1}{V_\Omega}\, \boldsymbol{U}^T \boldsymbol{K_i}\, \boldsymbol{U}\,. \tag{33}$$

However, by using the Conjugate Gradient Sensitivity (CGS) approach [9, 6], more accurate approximations can be obtained.

Jacobi preconditioning is used, so the preconditioner matrix $\boldsymbol{M}$ corresponds to the diagonal of $\left[\breve{\boldsymbol{K}} + \boldsymbol{\Delta}\breve{\boldsymbol{K}}\right]$, where $\boldsymbol{\Delta}\breve{\boldsymbol{K}}$ is given by

$$\boldsymbol{\Delta}\breve{\boldsymbol{K}} = \begin{cases} \breve{\boldsymbol{K}}_{\boldsymbol{i}}\,, & \text{if } \bar{x}_i = 0\,, \\ -\,\breve{\boldsymbol{K}}_{\boldsymbol{i}}\,, & \text{if } \bar{x}_i = 1\,. \end{cases} \tag{34}$$

The vector $\boldsymbol{z_h}$ is used to compute the CGS approximations, when estimating ΔC_{00}, it is defined with respect to the first column of $\boldsymbol{U}$ $(\boldsymbol{u_{xx}})$:

$$\boldsymbol{z_h} = \boldsymbol{P}^T \boldsymbol{K_i}\, \boldsymbol{u_{xx}}\,. \tag{35}$$

When estimating ΔC_{22}, $\boldsymbol{z_h}$ is defined with respect to the third column of $\boldsymbol{U}$ $(\boldsymbol{u_{xy}})$:

$$\boldsymbol{z_h} = \boldsymbol{P}^T \boldsymbol{K_i}\, \boldsymbol{u_{xy}}\,. \tag{36}$$

By computing, for each case, the vectors

$$\boldsymbol{z_m} = \boldsymbol{M}^{-1}\boldsymbol{z_h} \;\text{ and }\; \boldsymbol{z_k} = \left[\breve{\boldsymbol{K}} + \boldsymbol{\Delta}\breve{\boldsymbol{K}}\right]\boldsymbol{z_m}\,, \tag{37}$$

and the coefficients

$$\omega_{hm} = \boldsymbol{z_h}^T \boldsymbol{z_m}\;\;,\;\; \omega_{mk} = \boldsymbol{z_m}^T \boldsymbol{z_k} \;\text{ and }\; \phi_{m1} = \frac{\omega_{hm}}{\omega_{mk}}\,, \tag{38}$$

the CGS approximations with 1 step (CGS-1) can be obtained as

$$\Delta C_{00} \approx \begin{cases} C_{00}^\delta - \dfrac{1}{V_\Omega}\left[\phi_{m1}\, \omega_{hm}\right], & \text{if } \bar{x}_i = 0\,, \\ -\,C_{00}^\delta - \dfrac{1}{V_\Omega}\left[\phi_{m1}\, \omega_{hm}\right], & \text{if } \bar{x}_i = 1\,, \end{cases} \tag{39}$$

and

$$\Delta C_{22} \approx \begin{cases} C_{22}^{\delta} - \dfrac{1}{V_\Omega} \left[\phi_{m1}\, \omega_{hm} \right] , & \text{if } \bar{x}_i = 0 \, , \\ -\, C_{22}^{\delta} - \dfrac{1}{V_\Omega} \left[\phi_{m1}\, \omega_{hm} \right] , & \text{if } \bar{x}_i = 1 \, . \end{cases} \tag{40}$$

Using 2 steps, more accurate CGS expressions can be obtained. By computing, for each case, the vectors

$$\boldsymbol{z_\eta} = \boldsymbol{M}^{-1} \boldsymbol{z_k} \ \text{ and } \ \boldsymbol{z_\xi} = \left[\breve{\boldsymbol{K}} + \boldsymbol{\Delta} \breve{\boldsymbol{K}} \right] \boldsymbol{z_\eta} \, , \tag{41}$$

and the coefficients

$$\omega_{k\eta} = \boldsymbol{z_k}^T \boldsymbol{z_\eta} \ , \ \omega_{\eta\xi} = \boldsymbol{z_\eta}^T \boldsymbol{z_\xi} \ , \ \phi_{m2} = \frac{\omega_{hm}\, \omega_{\eta\xi} - \omega_{mk}\, \omega_{k\eta}}{\omega_{mk}\, \omega_{\eta\xi} - {\omega_{k\eta}}^2} \ \text{ and } \ \phi_{\eta 2} = \frac{{\omega_{mk}}^2 - \omega_{hm}\, \omega_{k\eta}}{\omega_{mk}\, \omega_{\eta\xi} - {\omega_{k\eta}}^2} \, , \tag{42}$$

the CGS-2 approximations can be obtained as

$$\Delta C_{00} \approx \begin{cases} C_{00}^{\delta} - \dfrac{1}{V_\Omega} \left[\phi_{m2}\, \omega_{hm} + \phi_{\eta 2}\, \omega_{mk} \right] , & \text{if } \bar{x}_i = 0 \, , \\ -\, C_{00}^{\delta} - \dfrac{1}{V_\Omega} \left[\phi_{m2}\, \omega_{hm} + \phi_{\eta 2}\, \omega_{mk} \right] , & \text{if } \bar{x}_i = 1 \, , \end{cases} \tag{43}$$

and

$$\Delta C_{22} \approx \begin{cases} C_{22}^{\delta} - \dfrac{1}{V_\Omega} \left[\phi_{m2}\, \omega_{hm} + \phi_{\eta 2}\, \omega_{mk} \right] , & \text{if } \bar{x}_i = 0 \, , \\ -\, C_{22}^{\delta} - \dfrac{1}{V_\Omega} \left[\phi_{m2}\, \omega_{hm} + \phi_{\eta 2}\, \omega_{mk} \right] , & \text{if } \bar{x}_i = 1 \, . \end{cases} \tag{44}$$

Instead of using the raw sensitivity vectors to linearize the functions 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 [14, 15]. Moreover, a momentum method is included, so the previous values of the sensitivity vectors throughout the iterations (with proper weighting factors) are added to the value of the current sensitivity vectors [16]. This inhibits oscillations between consecutive iterations and favors a more extensive exploration of the domain of feasible topologies.

In order to properly filter the sensitivity map, the sensitivity values of the augmented elements are assigned to all corresponding symmetric quadrilateral elements in a extended mesh. The extended mesh includes parts of some neighboring periodic cells, and it is composed of $10\, N_d$ elements, as illustrated in Figure 6 for $N_s = 3$.

Here, a simple conical filter is considered, it is defined by its radius, r_s, which is a geometric parameter that indirectly controls the minimal thickness of the structural components, independently of the mesh.

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

$$W_{ij} = \frac{\max\left(r_s - r(i,j) \, , \, 0\right)}{\displaystyle\sum_{k=1}^{10\, N_d} \max\left(r_s - r(i,k) \, , \, 0\right)} \tag{45}$$

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

$$\left[\alpha_i^{[f]}\right]^{\text{fil}} = \sum_{j=1}^{10\, N_d} W_{ij}\, \alpha_j^{[f]} \, . \tag{46}$$

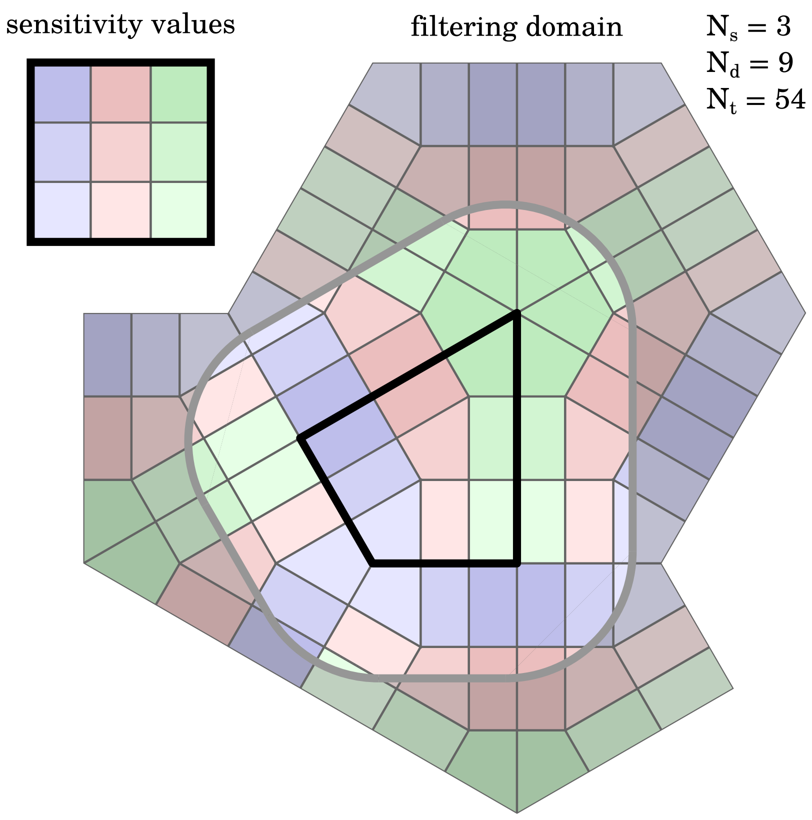

Figure 6: Extended mesh

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 $\max\left(r_s - r(i,j)\,,\,0\right) = 0$. 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.

For the momentum method, 25% is used for the objective function, and 0% (no momentum) is used for the constraint function. The sensitivity vector of the objective function is normalized so that eventual sensitivity peaks do not overly pollute the optimization process [17].

Thus, for the k-th iteration the normalized filtered vector $\tilde{\boldsymbol{\alpha}}^{[\boldsymbol{h}_\nu]}$ is given by

$$\tilde{\boldsymbol{\alpha}}^{[\boldsymbol{h}_\nu]} = \frac{\left[\boldsymbol{\alpha}^{[\boldsymbol{h}_\nu]}\right]^{\text{fil}}}{\left\|\left[\boldsymbol{\alpha}^{[\boldsymbol{h}_\nu]}\right]^{\text{fil}}\right\|_\infty} = \frac{\left[\boldsymbol{\alpha}^{[\boldsymbol{h}_\nu]}\right]^{\text{fil}}}{\max\limits_i \left|\left[\alpha_i^{[h_\nu]}\right]^{\text{fil}}\right|}\,. \tag{47}$$

Then, the momentum is applied as follows:

$$\left[\boldsymbol{\alpha}^{[\boldsymbol{h}_\nu]}\right]^{\text{mom}\,(k)} = 0.75\,\tilde{\boldsymbol{\alpha}}^{[\boldsymbol{h}_\nu](k)} + 0.25\,\left[\tilde{\boldsymbol{\alpha}}^{[\boldsymbol{h}_\nu]}\right]^{\text{mom}\,(k-1)}. \tag{48}$$

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

$$\left[\tilde{\boldsymbol{\alpha}}^{[\boldsymbol{h}_\nu]}\right]^{\text{mom}} = \frac{\left[\boldsymbol{\alpha}^{[\boldsymbol{h}_\nu]}\right]^{\text{mom}}}{\left\|\left[\boldsymbol{\alpha}^{[\boldsymbol{h}_\nu]}\right]^{\text{mom}}\right\|_\infty} = \frac{\left[\boldsymbol{\alpha}^{[\boldsymbol{h}_\nu]}\right]^{\text{mom}}}{\max\limits_i \left|\left[\alpha_i^{[h_\nu]}\right]^{\text{mom}}\right|}\,. \tag{49}$$

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

After solving the linearized subproblem, some additional procedures are performed. They correspond to reasonable small perturbations that are applied in each iteration to improve explorability and to bias the results towards simplified structures (with no disconnected solid islands, and with larger and smoother void cavities and solid components). In each iteration, an opening morphological operator is applied to the solution of the linearized subproblem. Then, solid islands are identified. If the islands are composed by less than $N_d \times D_{\max}$ augmented elements, they are completely removed (turned into void); otherwise, the $N_d \times D_{\max}$ most insensitive augmented elements are removed, with respect to $\left[\boldsymbol{\alpha}^{[\boldsymbol{E}]}\right]^{\text{fil}}$. If the topology is kept unaltered after all these procedures (solving the linearized subproblem; applying the opening operator; and removing elements from solid islands), an erosion operator is applied to the topology. Lastly, if the constraint is not respected in the end of an iteration, the topology from the last iteration is recovered and a dilation operator is applied to it. When the optimization is concluded, remaining solid islands are removed from the final solution.

The morphological operators [18] are defined their radius r_m, which is a geometric parameter that controls the minimal thickness of the structural components. The operator weights B_{ij} are given by

$$B_{ij} = \begin{cases} 1\,, & \text{if } r(i,j) \leqslant r_m\,, \\ 0\,, & \text{if } r(i,j) > r_m\,. \end{cases} \tag{50}$$

As before, these distances are computed in the extended mesh, so that proper neighborhood relationships are considered for elements close to the boundaries of the design domain.

The eroded density vector is given by

$$[x_i]^{\text{ero}} = 1 - \max_j \left(B_{ij}\,[1 - x_j]\right)\,, \tag{51}$$

the dilated density vector is given by

$$[x_i]^{\text{dil}} = \max_j \left(B_{ij}\,x_j\right)\,, \tag{52}$$

and the opened density vector is given by

$$[x_i]^{\text{ope}} = \max_j \left(B_{ij}\,\,[x_j]^{\text{ero}}\right)\,. \tag{53}$$

These operations can be efficiently performed by storing the required information in a sparse matrix $\boldsymbol{B}$.

In order to remove the disconnected structural components from the base cell, a depth-first search is performed to visit all solids connected to a reference element (arbitrarily chosen). Elements are considered to be connected if there is a path connecting them through a sequence of directly connected solid neighbors. Neighboring solid elements are considered to be directly connected if they have a common edge (elements that share a single node are not directly connected). To avoid identifying an island in place of the actual structure, depth-first searches are performed (altering the reference element) until the identified group of connected elements corresponds to at least 50% of the solids of the current topology. Then, all solid elements that are not in this group are eligible to be removed from the structure.

Once again, the extended mesh is used, so that proper neighborhood relationships are considered for paths going through the boundaries of the design domain.

3 Dataset Generation

3.1 Fixed Properties

The area of the base cell is set to $V_\Omega = 1.0\,m^2$, so the dimensions of the domain are $L_x = \left[\frac{1}{108}\right]^{\frac{1}{4}}\ m \approx 0.31\,m$ and $L_y = \left[\frac{1}{12}\right]^{\frac{1}{4}}\ m \approx 0.54\,m$. The number of elements in each direction of the design domain is set to $N_s = 32$, so the number of design variables is $N_d = 1024$, and the number of quadrilateral elements in the mesh is $N_t = 6144$. The sensitivity filter radius is set to $r_s = 0.024\,m$; and the radius used for the morphological operators is set to $r_m = 0.018\,m$.

The soft-kill parameter is set to $p_k = 1 \times 10^{-9}$; the volume penalization factor is set to $\beta = 0.05$; the maximal topology variation is set to $D_{\max} = 1.5625\%$ (1/64); η_E is defined in each iteration so that a maximal decrease of $0.05\,Pa$ is imposed for the Young's modulus in each linearized subproblem; and the patience parameter used as stopping criterion is set to $P = 30$.

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/Anaconda3-2022.10-Linux-x86_64.sh
```

Then, run the bash script:

```
bash Anaconda3-2022.10-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/Metamaterial-Dataset). Finally, run the provided "metamaterial.sh" bash script in order to setup the conda environment:

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

This script will: update conda; create a new Python-3.8 environment named "metamaterial"; add the channel conda-forge; set channel_priority as strict; install numpy [19], scipy [20], matplotlib [21], cython [22], scikit-sparse [23] and pulp [24]; and build all Cython codes in "./cython" directory.

3.2.2 Generate Datasets

Before executing the scripts, be warned that the complete dataset occupies around **277 GB** of disk.

To generate the dataset, activate the metamaterial environment and go to the directory with the provided Python codes:

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

Then, run the script "input_metamat.py", which will create the "./input" directory and generate the input file "inputmat.npy", it contains a set of 18 382 unique pairs of parameters (ν^*, $E_{\min}$). This will occupy around 150 kB of disk.

```
python ./input_metamat.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/SILP" and open the "basecell_silp.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=700" in "basecell_silp_0.py"; "fid_ini=700" and "fid_lim=1400" in "basecell_silp_1.py"; "fid_ini=1400" and "fid_lim=2100" in "basecell_silp_2.py"; and "fid_ini=2100" and "fid_lim=2800" in "basecell_silp_3.py". This will run the first 2800 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 ./SILP/basecell_silp_0.py
```

If it is the first time the program is being executed, the files "sfil_data.npy", "sfil_row.npy", "sfil_col.npy", "mfil_data.npy", "mfil_row.npy", "mfil_col.npy" and "neighbors.npy" will be generated. This will occupy around 290 kB of disk. The file "neighbors.npy" contains a matrix of dimensions $N_d \times 4$ with the indices of all directly connected neighbors for each design variable. The other files contain the matrices (in COO format) of the sensitivity filter and morphological operators. It is recommended that no new parallel execution be started before all these files are generated. Then, open the next terminal and repeat for the next processor:

```
taskset -c 1 python ./SILP/basecell_silp_1.py
```

```
taskset -c 2 python ./SILP/basecell_silp_2.py
```

```
taskset -c 3 python ./SILP/basecell_silp_3.py
```

The "./SILP/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 problem, the obtained mechanical properties for the optimized metamaterials, and the execution time of the main tasks for 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 18 382 cases are optimized. Be warned that the WS expression (Equation 30), which is very costly, is computed in each iteration, so each optimization may take a few minutes (even for this coarse mesh). **Weeks may be necessary to generate the whole dataset, depending on your computer**.

After performing all 18 382 optimizations, run the script "generate_metamat.py" to conclude the generation of the dataset:

```
python ./generate_metamat.py
```

In the <location_in_your_machine>, this script will create the directory "./dataset/SILP". Finally, it will relocate all generated data to this directory in an organized manner.

3.3 Implementation – Python

3.3.1 ./source/python/input_metamat.py

This script generates the input data for the inverse homogenization problems to be solved. For given properties of the base material ($\breve{E}$ and $\breve{\nu}$), 18 382 unique pairs of prescribed properties (ν^*, E_{min}) are defined and stored in a single file.

Firstly, the directory where the input data will be stored is created; and the parameters $\breve{E}$ and $\breve{\nu}$ are defined, denoted by "Ey" and "nu".

input_metamat

```python
import os
import numpy as np
# check directories
if not os.path.exists('./input'):
    os.mkdir('./input')
Ey = 1.00  # Young's modulus of the base material
nu = 0.30  # Poisson's ratio of the base material
```

Then, all considered values for the target Poisson's ratio (ν*) are generated and stored in the array "nuval_list". Likewise, all considered values for the minimal Young's modulus ($E_{\min}$) are generated and stored in the array "Eymin_list".

The target Poisson's ratio range from -1.00 to 1.00 in steps of 0.01, but values close to the Poisson's ratio of the base material (values in $]0.20, 0.40[$) are disregarded, which results in 182 unique values; the minimal Young's modulus ranges from 0.0% to 50.0% in steps of 0.5%, relative to the Young's modulus of the base material, which results in 101 unique values.

input_metamat
```
nuval_list = np.concatenate((np.linspace(-1.0,nu-0.1,1+int(100*(nu+0.9))),np.linspace(nu+0.1,1.0,1+int(100*(0.9-nu)))))
nu_num = len(nuval_list)
Eymin_list = np.linspace(0,0.5*Ey,101)
Ey_num = len(Eymin_list)
```

Lastly, all values of target Poisson's ratio are combined with all values of minimal Young's modulus to generate all possible pairs ($\nu^*, E_{\min}$), the total number of combinations is $18\,382 = 182 \times 101$. All pairs of properties are stored in the rows of the matrix "inputmat". It is stored in disk (in 4-bytes format), in the input folder, named as "inputmat.npy". It corresponds to around 145 kB of data.

input_metamat
```
print('generating input files')
total_num = nu_num*Ey_num
inputmat = np.ndarray([total_num,2],dtype=np.float32)
fid = 0
for knu in range(nu_num):
    print(': {:05d} / {:05d} :'.format(knu*Ey_num+1,total_num))
    for kEy in range(Ey_num):
        input_num = knu*Ey_num + kEy
        nuval = np.float32(nuval_list[knu])  # target Poisson's ratio
        Eymin = np.float32(Eymin_list[kEy])  # minimal Young's modulus
        inputmat[fid,0] = nuval
        inputmat[fid,1] = Eymin
        fid = fid + 1
print(': {:05d} / {:05d} :'.format(input_num+1,total_num))
np.save('./input/inputmat.npy',inputmat)
print('[ input file generated ]')
print('done!')
```

3.3.2 ./source/python/SILP/basecell_silp.py

This script performs the SILP 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, presents each considered input data and the obtained mechanical properties of each optimized metamaterial, together with dates and times that inform when each optimization procedure was performed. The time log ("time_log.txt") presents the execution times of the main tasks 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 Poisson's ratio of the optimized topology ("nu_opt.npy"); the Young's modulus of the optimized topology ("Ey_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_xx.npy", "dis_yy.npy" and "dis_xy.npy"); the CGS-0 approximations for each topology ("dC00_0.npy", "dC11_0.npy" and "dC22_0.npy"); the CGS-1 approximations ("dC00_1.npy", "dC11_1.npy" and "dC22_1.npy"); the CGS-2 approximations ("dC00_2.npy", "dC11_2.npy" and "dC22_2.npy"); the exact variations, obtained through the WS expression ("dC00_w.npy", "dC11_w.npy" and "dC22_w.npy"); the Poisson's ratio of each topology ("nu.npy"); the Young's modulus of each topology ("Ey.npy"); the relative volume values of each topology ("vol.npy"); the execution times of the main tasks of the optimization processes, together with the number of performed iterations ("tim.npy").

It should be noted that the arrays "dC00_0", "dC00_1", "dC00_2", "dC00_w", "dC11_0", "dC11_1", "dC11_2", "dC11_w", "dC22_0", "dC22_1", "dC22_2" and "dC22_w" store the **variations** of the diagonal terms of $\boldsymbol{C}$ when the state of each augmented element is switched. They are sometimes referred to as sensitivity vectors because their absolute values correspond to the sensitivity values of these diagonal terms. However, to obtain proper sensitivity values, the signs of the values corresponding to solid augmented elements would have to be reversed.

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 7 optimizations problems (this value is defined through the parameter "noptf"). Thus, each "fid.npy" file contains 28 bytes of data; each "inp.npy" file contains 56 bytes of data; each "top_opt.npy" file contains 896 bytes of data; each "nu_opt.npy" file contains 28 bytes of data; each "Ey_opt.npy" file contains 28 bytes of data; each "ptr2opt.npy" file contains 32 bytes of data; and each "tim.npy" file contains 196 bytes of data. The other files depend on the number of iterations performed in each optimization process. Considering the average value of 74 iterations (so that each one of these lists has 75 entries), each "ptr2inp.npy" file would contain 2.1 kB of data; each "top.npy" file would contain 65.6 kB of data; each "dis_xx.npy" file would contain 25.4 MB of data; each "dis_yy.npy" file would contain 25.4 MB of data; each "dis_xy.npy" file would contain 25.4 MB of data; each "dC00_0.npy" would contain 2.1 MB of data; each "dC11_0.npy" would contain 2.1 MB of data; each "dC22_0.npy" would contain 2.1 MB of data; each "dC00_1.npy" would contain 2.1 MB of data; each "dC11_1.npy" would contain 2.1 MB of data; each "dC22_1.npy" would contain 2.1 MB of data; each "dC00_2.npy" would contain 2.1 MB of data; each "dC11_2.npy" would contain 2.1 MB of data; each "dC22_2.npy" would contain 2.1 MB of data; each "dC00_w.npy" would contain 2.1 MB of data; each "dC11_w.npy" would contain 2.1 MB of data; each "dC22_w.npy" would contain 2.1 MB of data; each "nu.npy" file would contain 2.1 kB of data; each "Ey.npy" file would contain 2.1 kB of data; and each "vol.npy" file would contain 2.1 kB of data. These add up to around 101 MB of data.

The 18 382 optimization problems will generate 2 626 of each of these files, which would result in around 260 GB of data. However, according to the Disk Usage Analyzer, a tool for analyzing disk usage for GNOME, the dataset occupies around **277 GB** of disk (7% more than the amount of useful data).

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

basecell_silp

```python
import os, sys, gc
import numpy as np
from time import perf_counter
from datetime import datetime
from scipy.sparse import coo_matrix
from sksparse.cholmod import analyze
from mesh import get_mesh, get_fmesh
from elem import get_emat, get_augmat
from filters import get_sfil, get_mope
from rem_islands import visit, get_neighbors
from topopt import update, ws
from ilp_solver import solve_ILP, solve_BESO
sys.path.append('../../cython/')
from silp_sens import cgs
Ns = 32          # number of elements in each side of the design domain
Eyvar = 0.05     # maximal decrease in Young's modulus per iteration
nuvar = 2.0      # maximal variation in Poisson's ratio per iteration
Dmax  = 0.015625 # maximal topology variation
rsen  = 0.024    # sensitivity filter radius
rmor  = 0.018    # morphology filter radius
patience = 30    # patience stop criterion
momentum = 0.25  # sensitivity momentum
beta = 0.05      # volume penalization factor
Ey = 1.00        # Young's modulus of the base material
nu = 0.30        # Poisson's ratio of the base material
pk = 1e-9        # soft-kill parameter
small = 1e-14    # small value to compare float numbers
noptf   = 7      # number of optimizations to be stored in the same file
fid_ini = 0      # initial input index |run from input 0
fid_lim = 18382  # input index limit   |up to input 18381
```

Some geometrical parameters are defined according to N_s (denoted by "Ns"), considering that the base cell is a regular hexagon of unitary area. The number of elements in the design domain (N_d) is stored in the variable "N", and the number of quadrilateral elements in the mesh (N_t) is stored in the variable "Nt". The variable "M" stores the number of nodes in the base cell, and the variable "G" stores the number of degrees of freedom of the unconstrained system. 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, this is performed by the function "mesh.get_mesh". The array "etype" stores, for each quadrilateral element, the pointer to its corresponding stiffness matrix in "Ket" (defined in the next snippet of code), the matrix "sym" stores, for each design variable, the indices of its six corresponding quadrilateral elements. The python script "mesh.py" is detailed in a following section.

basecell_silp

```python
Lx = 1.0/(108**0.25)        # design domain shorter side
Ly = np.sqrt(3)*Lx          # design domain longer side
Lex = Lx/Ns                 # element shorter side
Ley = np.sqrt(3)*Lex        # element longer side
N = Ns**2                   # number of elements in the design domain
Nt = 6*N                    # number of elements in the base cell
M = 1 + 6*Ns*(Ns+1)         # number of nodes in the base cell
G = 2*M                     # number of degrees of freedom in the base cell
```

```
dXmax = int(round(N*Dmax))  # maximal topology variation (number of elements)
# Generate Mesh
coor, inci, etype, sym = get_mesh(Ns, Lex, Ley)
```

The function “elem.get_emat” is used to compute the all the elemental stiffness matrices, for the twelve considered types of bilinear quadrilateral elements, in plane stress state. They are stored in the tensor “Ket”, of dimensions $12 \times 8 \times 8$. Its vectorized data is stored in “Ketvec”. The stiffness variation matrices are stored in “dKe”, which correspond to the stiffness changes in the global matrix when the state of one element is switched (from solid to void, or from void to solid). The function “elem.get_augmat” is used to compute all the six different local stiffness variation matrices for the augmented elements, they are stored in the list “dKelist”. The array “aug_etype” stores, for each design variable, the pointer to its corresponding local matrix in “dKelist”. And the list “Hlist” stores the factorizations of each matrix in “dKelist” (Equation 27). The python script “elem.py” is detailed in a following section.

basecell_silp

```
# Element Matrices (Quad4) - Plane Stress State
Ket = get_emat(Ey,nu)
Ketvec = np.ndarray((12,64))
dKe = np.ndarray((12,8,8))
for ek in range(12):
    Ketvec[ek,:] = Ket[ek,:,:].ravel()
    dKe[ek,:,:] = (1.0-pk)*Ket[ek,:,:]  # stiffness variation of a topological change
# augmented element matrices (6xQuad4)
aug_etype, Hlist, dKelist = get_augmat(Ns,inci,etype,sym,dKe)
```

The initial topology is defined and stored in “x_init”. It is a mostly solid structure, with three rhombus-shaped cavities placed on different symmetry axes, each cavity is centered between the center of the hexagonal cell and one of its edges, each rhombus has a larger diagonal of $\frac{1}{16}$ of the height of the base cell and a smaller diagonal of $\frac{1}{16}$ of the side of the base cell. The symmetric density vector, which defines the density values for all the quadrilateral elements, is stored in “xt”.

basecell_silp

```
# Initial Topology
x_init = np.ones(N,dtype=bool)    # design variables
Ntotal = Ns//16
Nhole = Ntotal
while Nhole > 0:
    x_init[(Ns-1-(Ntotal-Nhole))*Ns+(Ns//2-Nhole):(Ns-1-(Ntotal-Nhole))*Ns+(Ns//2+Nhole)] = False
    Nhole = Nhole - 1
xt = np.ndarray((Nt),dtype=bool)  # symmetric density vector
for k in range(N):
    xt[sym[k,:]] = x_init[k]
```

If the file “sfil_data.npy” (“sfil_row.npy” and “sfil_col.npy” are not verified) is not in the “./source/python/SILP” directory, the sensitivity filter matrix is computed and stored in disk. If the file “mfil_data.npy” (“mfil_row.npy” and “mfil_col.npy” are not verified) does not exist in the directory, the morphological filter matrix is computed and stored in disk. Likewise, if the file “neighbors.npy” does not exist in the directory, the neighbors matrix is computed and stored in disk. If these files already exist, the corresponding matrices are loaded from disk. The function “mesh.get_fmesh” is used to generate the extended mesh, that is required to compute the sensitivity filter matrix, the morphological filter matrix and the neighbors matrix. The matrix “Q” is used to transform a vector of the design domain into a symmetric vector in the extended mesh. The function “filters.get_sfil” computes the sensitivity filter matrix, according to the defined radius r_s (denoted by “rsen”). The function “filters.get_mope” computes the morphological filter matrix, according to the defined radius r_m (denoted by “rmor”). The function “rem_islands.get_neighbors” computes the neighbors matrix, used to identify disconnected structural components. All these matrices occupy around 290 kB of disk. The python scripts “mesh.py”, “filters.py” and “rem_islands.py” are detailed in following sections.

basecell_silp

```
# Generate Extended Mesh
if (not os.path.exists('./sfil_data.npy')) or (not os.path.exists('./mfil_data.npy')) or (not os.path.exists('./neighbors.npy')):
    coor_lb, coor_bot, inci_lb, inci_bot, sym_lb, sym_bot = get_fmesh(Ns, Lx, Ly, Lex, Ley)
# Sensitivity and Morphology Filters Matrices
if os.path.exists('./sfil_data.npy') and os.path.exists('./mfil_data.npy'):
    data = np.load('./sfil_data.npy')
    row  = np.load('./sfil_row.npy')
    col  = np.load('./sfil_col.npy')
    Sf = coo_matrix((data,(row,col)),shape=(N,N))
    Sf = Sf.tocsr()
    data = np.load('./mfil_data.npy')
    row  = np.load('./mfil_row.npy')
    col  = np.load('./mfil_col.npy')
    Mf = coo_matrix((data,(row,col)),shape=(N,N))
    Mf = Mf.tocsr()
else:
    # extended mesh
    fcoor = np.vstack((coor,coor_lb,coor_bot))
    finci = np.vstack((inci,inci_lb,inci_bot))
    fsym = np.hstack((sym,sym_lb,sym_bot))
    elepos = 0.25*fcoor[finci].sum(axis=1)
    row = fsym.ravel('C')
    col = np.repeat(np.arange(N),10)
    data = np.ones(10*N)
    Q = coo_matrix((data,(row,col)),shape=(10*N,N))
    Q = Q.tocsc()
    # sensitivity filter matrix
    Sf = get_sfil(N,sym,elepos,Q,rsen)
    Sfcoo = Sf.tocoo()
    np.save('./sfil_data.npy',Sfcoo.data)
    np.save('./sfil_row.npy',Sfcoo.row)
    np.save('./sfil_col.npy',Sfcoo.col)
    # morphology filter matrix
    Mf = get_mope(N,sym,elepos,Q,rmor)
    Mfcoo = Mf.tocoo()
    np.save('./mfil_data.npy',Mfcoo.data)
    np.save('./mfil_row.npy',Mfcoo.row)
    np.save('./mfil_col.npy',Mfcoo.col)
# Get Neighbors
if os.path.exists('./neighbors.npy'):
    neighbors = np.load('./neighbors.npy')
else:
    neighbors = get_neighbors(Ns,inci,inci_lb,inci_bot,sym,sym_lb,sym_bot)
    np.save('./neighbors.npy',neighbors)
```

The matrix of periodic boundary conditions is stored in “P”, it is used to impose zero-displacement at the node of index 0 (in the center of the hexagon), and to impose proper relations between displacements and loads of opposing edges. The macro-displacements vectors are computed and stored in the matrix “Uhat”.

basecell_silp

```
# constraint matrix
Gb = 4*Ns
Gd = G - 2 - 6*Gb
Gr = Gd + 3*Gb - 2
ivec = np.arange(2,G)
j0 = np.arange(0,Gd)
j1 = np.arange(Gd,Gd+3*Gb-2)
v1=np.arange(Gd+Gb-2,Gd-1,-2)
v2=np.arange(Gd+Gb-1,Gd,-2)
j2=np.vstack((v1,v2)).ravel('F')
v1=np.arange(Gd+2*Gb-2,Gd+Gb-3,-2)
v2=np.arange(Gd+2*Gb-1,Gd+Gb-2,-2)
j3=np.vstack((v1,v2)).ravel('F')
v1=np.arange(Gd+3*Gb-4,Gd+2*Gb-3,-2)
v2=np.arange(Gd+3*Gb-3,Gd+2*Gb-2,-2)
j4=np.vstack((v1,v2)).ravel('F')
jvec = np.concatenate((j0,j1,j2,j3,j4))
avec = np.ones(G-2)
P = coo_matrix((avec,(ivec,jvec)),shape=(G,Gr)).tocsr()
# macro-strain tensors
eps_xx = np.array([[1,0],[0,0]])
eps_yy = np.array([[0,0],[0,1]])
eps_xy = np.array([[0,0.5],[0.5,0]])
# macro-displacements vectors
uhat_xx = np.ravel(coor @ eps_xx, 'C')
uhat_yy = np.ravel(coor @ eps_yy, 'C')
uhat_xy = np.ravel(coor @ eps_xy, 'C')
Uhat = np.vstack((uhat_xx,uhat_yy,uhat_xy)).T
```

If there is no input data, a standard input is created and stored in disk. The input matrix (denoted by “inputmat”) is loaded from disk. The directory where the output data will be stored is created. Both input-output log and time log files are opened to be written.

basecell_silp

```
# 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_{:05d}_{:05d}'.format(fid_ini,fid_lim-1)):
    os.mkdir('./output/run_{:05d}_{:05d}'.format(fid_ini,fid_lim-1))
# check input
if not os.path.exists('../input/inputmat.npy'):
    nuval = np.float32(0.00)  # target Poisson's ratio
    Eymin = np.float32(0.10)  # minimal Young's modulus
    inputmat = np.array([[nuval,Eymin]])
    np.save('../input/inputmat.npy',inputmat)
# read input file
```

```
inputmat = np.load('../input/inputmat.npy')
# open log files
if not os.path.exists('./output/run_{:05d}_{:05d}/logs'.format(fid_ini,fid_lim-1)):
    os.mkdir('./output/run_{:05d}_{:05d}/logs'.format(fid_ini,fid_lim-1))
iolog = open('./output/run_{:05d}_{:05d}/logs/io_log.txt'.format(fid_ini,fid_lim-1),'a')
tlog = open('./output/run_{:05d}_{:05d}/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.

```
basecell_silp
iolog.write('BASE CELL 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 Poisson\'s ratio :           nu_opt.npy                     =\n')
iolog.write('= ------ :      optimized Young\'s modulus :           Ey_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('= ------ :       xx-displacements vectors :           dis_xx.npy                     =\n')
iolog.write('= ------ :       yy-displacements vectors :           dis_yy.npy                     =\n')
iolog.write('= ------ :       xy-displacements vectors :           dis_xy.npy                     =\n')
iolog.write('= ------ : dC00_CGS-0 sensitivity vectors :           dC00_0.npy                     =\n')
iolog.write('= ------ : dC00_CGS-1 sensitivity vectors :           dC00_1.npy                     =\n')
iolog.write('= ------ : dC00_CGS-2 sensitivity vectors :           dC00_2.npy                     =\n')
iolog.write('= ------ :    dC00_WS sensitivity vectors :           dC00_w.npy                     =\n')
iolog.write('= ------ : dC11_CGS-0 sensitivity vectors :           dC11_0.npy                     =\n')
iolog.write('= ------ : dC11_CGS-1 sensitivity vectors :           dC11_1.npy                     =\n')
iolog.write('= ------ : dC11_CGS-2 sensitivity vectors :           dC11_2.npy                     =\n')
iolog.write('= ------ :    dC11_WS sensitivity vectors :           dC11_w.npy                     =\n')
iolog.write('= ------ : dC22_CGS-0 sensitivity vectors :           dC22_0.npy                     =\n')
iolog.write('= ------ : dC22_CGS-1 sensitivity vectors :           dC22_1.npy                     =\n')
iolog.write('= ------ : dC22_CGS-2 sensitivity vectors :           dC22_2.npy                     =\n')
iolog.write('= ------ :    dC22_WS sensitivity vectors :           dC22_w.npy                     =\n')
iolog.write('= ------ :          Poisson\'s ratio array :               nu.npy                     =\n')
iolog.write('= ------ :          Young\'s modulus array :               Ey.npy                     =\n')
iolog.write('= ------ :                   volume array :              vol.npy                     =\n')
iolog.write('= ------ :                     time array :              tim.npy                     =\n')
iolog.write('======================================================================================\n')
iolog.write('      INPUT || NUVAL : EYMIN >> NUOPT : EYOPT ||             BEGIN :               END\n')
tlog.write('BASE CELL OPTIMIZATION (TIME LOG)\n')   # write in time log
tlog.write('============================================================================================\n')
tlog.write('      INPUT || (  IT x ):    M-ILP : M-SOLVER :    M-CGS :     M-WS :   M-POST ||     TOTAL\n')
```

The analysis for the initial topology is performed. The matrix assembly is performed through the "scipy.sparse.coo_matrix" function. The variable "Kg_coo_init" is the COO (Coordinate list) unconstrained global stiffness matrix; "Kg_cgs" is the CSC (Compressed Sparse Column) unconstrained matrix; and "Kr" is the CSC constrained matrix. Since "scipy" automatically removes zero entries from the matrix after performing the constraining operations, a maneuver is performed to preserve the nonzero pattern of "Kr". By preserving the nonzero pattern, the Cholesky factorization can be more efficiently performed throughout the optimization procedure. The variable "Fr" is the right-hand side of the linear system to be solved.

```
basecell_silp
# Assembly
pen = np.ones(Nt)
pen[~xt] = pk
data = np.ndarray((64*Nt))
for et in range(Nt):
    ek = etype[et]
    data[64*et:64*et+64] = pen[et]*Ketvec[ek,:]
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_init = coo_matrix((data,(row,col)),shape=(G,G))
Kg_csc = Kg_coo_init.tocsc()
Kr = P.T @ Kg_csc @ P
# maneuver to fix the pattern of non-zero entries
Z_coo = coo_matrix((np.ones(64*Nt),(row,col)),shape=(G,G))
Z_csc = Z_coo.tocsc()
Zr = P.T @ Z_csc @ P
Zr.sort_indices()
shift = 10*np.amax(abs(Ket))
```

```
Kr = Kr + shift*Zr
Kr.sort_indices()
Kr.data = Kr.data - shift*Zr.data
# right-hand side
Fr = -P.T @ Kg_csc @ Uhat
```

The optimal fill-reducing permutation is computed for “Kr” using the “sksparse.cholmod.analyze” function. The reduced displacements matrix ($\breve{\boldsymbol{U}}$) is computed and stored in “Ur”. Then, the total displacements matrix ($\boldsymbol{U}$) is stored in “Ug_init”.

basecell_silp

```
# Solve System
factor = analyze(Kr)
factor.cholesky_inplace(Kr)
Ur = factor(Fr)
Ug_init = Uhat + P @ Ur
```

The elasticity matrix of the homogenized material, the corresponding mechanical properties and the volume fraction are computed for the initial topology.

basecell_silp

```
# Effective Properties Matrix
Ch_init = Ug_init.T @ Kg_csc @ Ug_init
gamma_init = Ch_init[2,2]/Ch_init[0,0]
nuhat_init = 1-2*Ch_init[2,2]/Ch_init[0,0]
Eyhat_init = 4*Ch_init[2,2]*(Ch_init[0,0]-Ch_init[2,2])/Ch_init[0,0]
vol_init = sum(x_init)/N
```

The arrays that will store the sensitivity values of $\widehat{E}$ and h_ν are initialized. The sensitivity values of the diagonal terms of $\boldsymbol{C}$ are computed and stored. The function “silp_sens.cgs” is used to compute the CGS-0, CGS-1 and CGS-2 approximations, and the function “topopt.ws” is used to compute the exact sensitivity values through WS approach. The cython script “silp_sens.pyx” and the python script “topopt.py” are detailed in following sections.

basecell_silp

```
s_Ey = np.ndarray((N))
s_obj = np.ndarray((N))
dC00_0_init = np.ndarray((N))
dC11_0_init = np.ndarray((N))
dC22_0_init = np.ndarray((N))
dC00_1_init = np.ndarray((N))
dC11_1_init = np.ndarray((N))
dC22_1_init = np.ndarray((N))
dC00_2_init = np.ndarray((N))
dC11_2_init = np.ndarray((N))
dC22_2_init = np.ndarray((N))
cgs(dC00_0_init,dC11_0_init,dC22_0_init,dC00_1_init,dC11_1_init,dC22_1_init,
    dC00_2_init,dC11_2_init,dC22_2_init,x_init,N,sym,etype,aug_etype,inci,Ug_init,dKe,P,Kr,dKelist)
dC00_w_init, dC11_w_init, dC22_w_init = ws(x_init,aug_etype,sym,P,factor,inci,Ug_init,Hlist)
```

The loop to go through all the selected input values 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.

basecell_silp

```
file = 0  # file counter
fid = max([0,fid_ini])
while fid < min([fid_lim,inputmat.shape[0]]):
    if not os.path.exists('./output/run_{:05d}_{:05d}/file_{:04d}'.format(fid_ini,fid_lim-1,file)):
        os.mkdir('./output/run_{:05d}_{:05d}/file_{:04d}'.format(fid_ini,fid_lim-1,file))
    list_fid     = []
    list_inp     = []
    list_top_opt = []
    list_nu_opt  = []
    list_Ey_opt  = []
    list_ptr2opt = []
    list_ptr2inp = []
    list_top     = []
    list_dis_xx  = []
    list_dis_yy  = []
    list_dis_xy  = []
    list_dC00_0  = []
    list_dC00_1  = []
    list_dC00_2  = []
    list_dC00_w  = []
    list_dC11_0  = []
    list_dC11_1  = []
    list_dC11_2  = []
    list_dC11_w  = []
    list_dC22_0  = []
    list_dC22_1  = []
    list_dC22_2  = []
    list_dC22_w  = []
    list_nu      = []
    list_Ey      = []
    list_vol     = []
    list_tim     = []
    ptr = 0  # pointer to input
    for counter in range(noptf):
        if fid >= min([fid_lim,inputmat.shape[0]]):
            break
        print('running : {:05d} : setup'.format(fid))
        inp_id = 'inp_{:05d}'.format(fid)
        iolog.write('> ' + inp_id + ' ||')
        tlog.write('> ' + inp_id + ' ||')
```

The input values, in 4-bytes format, are read from "inputmat". They are automatically cast to 8-bytes precision whenever they are used for comparisons or arithmetic operations. 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 topology vector, stiffness matrix, displacements matrix, elasticity matrix, mechanical properties, volume fraction, and sensitivity values of the diagonal terms of $\boldsymbol{C}$ are recovered from the analysis performed for the initial topology.

basecell_silp

```
        nuval = inputmat[fid,0]  # target Poisson's ratio
        Eymin = inputmat[fid,1]  # minimal Young's modulus
        list_fid += [fid]
        list_inp += [[nuval,Eymin]]
        # write in log
        iolog.write(' {:5.2f} :'.format(nuval))
        iolog.write(' {:5.3f} >>'.format(Eymin))
        begin = datetime.now().strftime(' %y/%m/%d-%H:%M:%S :')
        # get initial data
        x      = x_init.copy()
        Kg_coo = Kg_coo_init.copy()
        Ug     = Ug_init.copy()
        Ch     = Ch_init.copy()
        gamma  = gamma_init
        nuhat  = nuhat_init
        Eyhat  = Eyhat_init
        vol    = vol_init
        dC00_0 = dC00_0_init.copy()
        dC00_1 = dC00_1_init.copy()
        dC00_2 = dC00_2_init.copy()
        dC00_w = dC00_w_init.copy()
        dC11_0 = dC11_0_init.copy()
        dC11_1 = dC11_1_init.copy()
        dC11_2 = dC11_2_init.copy()
        dC11_w = dC11_w_init.copy()
        dC22_0 = dC22_0_init.copy()
        dC22_1 = dC22_1_init.copy()
        dC22_2 = dC22_2_init.copy()
        dC22_w = dC22_w_init.copy()
```

According to the maximal variation in Poisson's ratio per iteration ("nuvar"), the parameter η_ν is computed. In the considered dataset, "nuvar" is set to 2.0, so η_ν is always 0.0 and no limitation imposed on the variation of $\widehat{\nu}$. The function h_ν is computed and stored in "fnu_g". The constraint function (that must be greater than or equal to zero) is computed and stored in "fEy". According to the maximal decrease in Young's modulus per iteration ("Eyvar"), to the constraint ("Eymin") and to the Young's modulus of the current topology ("Eyhat"), the parameter η_E is computed.

basecell_silp

```
        # target Poisson's ratio (limited variation)
        if nuhat > nuval:
            eta_nu = max([nuhat - nuval - nuvar, 0.0])
        else:
            eta_nu = min([nuhat - nuval + nuvar, 0.0])
        nu_g = nuval + eta_nu
        fnu_g = (nuhat-nu_g)**2
        # minimal Young's modulus (limited variation)
        fEy = Eyhat-Eymin
        eta_Ey = max([fEy - Eyvar, 0.0])
```

The sensitivity analysis is performed for the initial topology. The raw sensitivity values of the $\widehat{E}$ and h_ν are stored in "s_Ey" and "s_obj". The raw sensitivity values of the objective function (which includes the volume penalization) are stored in "raw_obj". The objective function for the current topology is computed and stored in "obj". The filtered sensitivity values ("fil_Ey" and "fil_obj") are computed. The final sensitivity vector for the objective function is stored in “mom_obj”, it is obtained by applying momentum from previous iterations (zero values in the first iteration) and by normalizing the vector with respect to its maximal absolute value.

basecell_silp
```
        # sensitivity analysis
        for e in range(N):
            ggamma = (Ch[2,2]+dC22_w[e])/(Ch[0,0]+dC00_w[e])
            dnu = 2.0*(gamma*dC00_w[e]-dC22_w[e])/(Ch[0,0]+dC00_w[e])
            dEy = 4.0*(1.0-ggamma)*dC22_w[e] + 2.0*Ch[2,2]*dnu
            dobj = 2.0*(nuhat-nu_g)*dnu + dnu*dnu
            if x[e]:
                s_Ey[e]  = -dEy
                s_obj[e] = -dobj
            else:
                s_Ey[e]  =  dEy
                s_obj[e] =  dobj
        raw_obj = s_obj + beta/N
        obj = fnu_g + beta*vol
        fil_Ey  = Sf @ s_Ey
        fil_obj = Sf @ raw_obj
        mom_obj = np.zeros(N)
        mom_obj = momentum*mom_obj + (1.0-momentum)*fil_obj/max(abs(fil_obj))
        mom_obj = mom_obj/max(abs(mom_obj))
```

The list of pointers “list_ptr2opt” appends the value that relates the current input to the index of the current optimization in the block of “noptf” processes. 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 vectors, sensitivity values of the diagonal terms of $\boldsymbol{C}$, mechanical properties and volume fraction are appended to their corresponding lists. For the current optimization, the best topology obtained thus far is stored in “top_opt”, the values of its mechanical properties and objective function are stored in “nu_opt”, “Ey_opt” and “obj_opt”.

basecell_silp
```
        # store data
        size_list = len(list_ptr2inp)
        list_ptr2opt += [size_list]
        list_ptr2inp += [ptr]
        list_top     += [x.copy()]
        list_dis_xx  += [Ug[:,0].copy()]
        list_dis_yy  += [Ug[:,1].copy()]
        list_dis_xy  += [Ug[:,2].copy()]
        list_dC00_0  += [dC00_0.copy()]
        list_dC00_1  += [dC00_1.copy()]
        list_dC00_2  += [dC00_2.copy()]
        list_dC00_w  += [dC00_w.copy()]
        list_dC11_0  += [dC11_0.copy()]
        list_dC11_1  += [dC11_1.copy()]
        list_dC11_2  += [dC11_2.copy()]
        list_dC11_w  += [dC11_w.copy()]
        list_dC22_0  += [dC22_0.copy()]
        list_dC22_1  += [dC22_1.copy()]
        list_dC22_2  += [dC22_2.copy()]
        list_dC22_w  += [dC22_w.copy()]
        list_nu      += [nuhat]
        list_Ey      += [Eyhat]
        list_vol     += [vol]
        # optimized topology thus far
        top_opt = x.copy()
        nu_opt = nuhat
        Ey_opt = Eyhat
        obj_opt = obj
```

The time array is initialized and the optimization loop is started. Elements that are certain to keep their state are disregarded in the current iteration, the array “selection” stores to the boolean mask that identifies the elements that may change their state. If there is no Young’s modulus constraint, the “ilp_solver.solve_BESO” function is used to solve the linear subproblem with the BESO algorithm. Otherwise, the “ilp_solver.solve_ILP” function is used to solve it with branch-and-bound simplex. After solving the linear subproblem, the opening morphological operator is applied to the topology vector. Then, disconnected solid elements are removed. If the topology is kept unaltered, the erosion operator is applied to it. The time to perform these procedures is stored in “time_array[0]”. The python script “ilp_solver.py” is detailed in a following section.

basecell_silp
```
        time_array = np.zeros(7)  # initialize time array
        keep_going = True
        waiting = 0
        it = 0
        while keep_going:
            it = it + 1
            print('running : :05d : :5d'.format(fid,it))
            # solve ILP
            t0 = perf_counter()
            selection = (x & (mom_obj>-small)) | ((~x) & (mom_obj<small)) | (x & (fil_Ey<small)) | ((~x) & (fil_Ey>-small))
            Nsel = sum(selection)
            y = x.copy()
            if Nsel > 0:
                if (Eymin + eta_Ey) < small:
                    ysel = solve_BESO(Nsel,x[selection],mom_obj[selection],dXmax)
                    y[selection] = ysel
                else:
                    ysel = solve_ILP(Nsel,x[selection],mom_obj[selection],fil_Ey[selection],fEy,eta_Ey,dXmax,sense_h='G')
                    y[selection] = ysel
            # open operator (erode + dilate)
            y[Mf[~y,:].indices] = False
            y[Mf[y,:].indices] = True
            # remove islands
            voly = sum(y)/N
            continent = np.zeros(N,dtype=bool)
            continent_vol = 0.0
            for e in (list(range(0,Ns))+list(range(Ns,N,Ns))):
                if y[e] and (not continent[e]):
                    continent = np.zeros(N,dtype=bool)
                    visit(e,y,continent,neighbors)
                    continent_vol = sum(continent)/N
                if continent_vol > 0.50*voly:
                    break
            if continent_vol > 0.50*voly:
                islands = np.argwhere(y!=continent).ravel()
                if len(islands) > dXmax:
                    sortedargs = np.argsort(abs(fil_Ey[islands]))
                    y[islands[sortedargs[:dXmax]]] = False
                else:
                    y[islands] = False
            # erode if nothing has been changed
            if all(x==y):
                print('--- erode ---')
                y[Mf[~y,:].indices] = False
            t1 = perf_counter()
            time_array[0] += (t1-t0)
```

The function “topopt.update” is used to update the topology vector, the stiffness matrix and the displacements matrix. The new mechanical properties, volume fraction, constraint function and objective function are computed. If the new topology breaks the Young’s modulus constraint, the previous topology is recovered, the dilation operator is applied to it, and everything is computed again for this new topology. The time to perform these procedures is stored in “time_array[1]”. The python script “topopt.py” is detailed in a following section.

```
basecell_silp
            # update topology
            t0 = perf_counter()
            if any(x!=y):
                elist = list(np.argwhere(x!=y)[:,0])
                Ug, Kr = update(x,etype,sym,pk,Ketvec,P,Kg_coo,Zr,shift,Uhat,factor,elist)
            # compute homogenized properties
            Kg_csc = Kg_coo.tocsc()
            Ch = Ug.T @ Kg_csc @ Ug
            gamma = Ch[2,2]/Ch[0,0]
            nuhat = 1-2*Ch[2,2]/Ch[0,0]
            Eyhat = 4*Ch[2,2]*(Ch[0,0]-Ch[2,2])/Ch[0,0]
            if nuhat > nuval:
                eta_nu = max([nuhat - nuval - nuvar, 0.0])
            else:
                eta_nu = min([nuhat - nuval + nuvar, 0.0])
            nu_g = nuval + eta_nu
            fnu_g = (nuhat-nu_g)**2
            vol = sum(x)/N
            obj = fnu_g + beta*vol
            fEy_test = Eyhat-Eymin
            eta_Ey_test = max([fEy_test - Eyvar, 0.0])
            # go back to last topology and dilate if constraint is broken
            if fEy_test < 0.0:
                # go back to last topology
                update(x,etype,sym,pk,Ketvec,P,Kg_coo,Zr,shift,Uhat,factor,elist,solve_sys=False)
                # dilate
                print('--- dilate ---')
                y = x.copy()
                y[Mf[y,:].indices] = True
                elist = list(np.argwhere(x!=y)[:,0])
                # compute homogenized properties
                Ug, Kr = update(x,etype,sym,pk,Ketvec,P,Kg_coo,Zr,shift,Uhat,factor,elist)
                Kg_csc = Kg_coo.tocsc()
                Ch = Ug.T @ Kg_csc @ Ug
                gamma = Ch[2,2]/Ch[0,0]
                nuhat = 1-2*Ch[2,2]/Ch[0,0]
                Eyhat = 4*Ch[2,2]*(Ch[0,0]-Ch[2,2])/Ch[0,0]
                if nuhat > nuval:
```

```
                eta_nu = max([nuhat - nuval - nuvar, 0.0])
            else:
                eta_nu = min([nuhat - nuval + nuvar, 0.0])
            nu_g = nuval + eta_nu
            fnu_g = (nuhat-nu_g)**2
            vol = sum(x)/N
            obj = fnu_g + beta*vol
            fEy = Eyhat-Eymin
            eta_Ey = max([fEy - Eyvar, 0.0])
        else:
            fEy = fEy_test
            eta_Ey = eta_Ey_test
        t1 = perf_counter()
        time_array[1] += (t1-t0)
```

The CGS and WS sensitivity values for the diagonal terms of $\boldsymbol{C}$ are computed with the functions "silp_sens.cgs" and "topopt.ws". The time to perform the CGS analysis is stored in "time_array[2]" and the time to perform the WS analysis is stored in "time_array[3]". The cython script "silp_sens.pyx" and the python script "topopt.py" are detailed in following sections.

basecell_silp

```
        # CGS analysis for dCh
        t0 = perf_counter()
        cgs(dC00_0,dC11_0,dC22_0,dC00_1,dC11_1,dC22_1,dC00_2,dC11_2,dC22_2,
            x,N,sym,etype,aug_etype,inci,Ug,dKe,P,Kr,dKelist)
        t1 = perf_counter()
        time_array[2] += (t1-t0)
        # WS analysis for dCh
        t0 = perf_counter()
        dC00_w, dC11_w, dC22_w = ws(x,aug_etype,sym,P,factor,inci,Ug,Hlist)
        t1 = perf_counter()
        time_array[3] += (t1-t0)
```

The sensitivity analysis is performed for the current topology. The data of the current iteration is appended to the output lists. If the Young's modulus constraint is respected, and if the Poisson's ratio is better than the best result obtained thus far, the best topology obtained thus far is updated. In order to improve explorability, if the objective function has decreased, even if the Poisson's ratio is worse than the best result obtained thus far, the counter used to check convergence ("waiting") is set to zero. If the limit number of iterations without improvements ("patience") has been achieved, the boolean variable "keep_going" is set to *False*, which will end the optimization loop. The time to perform these procedures is stored in "time_array[4]".

basecell_silp

```
        # post-procedures
        t0 = perf_counter()
        # sensitivity analysis for obj and Ey
        for e in range(N):
            ggamma = (Ch[2,2]+dC22_w[e])/(Ch[0,0]+dC00_w[e])
            dnu = 2.0*(gamma*dC00_w[e]-dC22_w[e])/(Ch[0,0]+dC00_w[e])
            dEy = 4.0*(1.0-ggamma)*dC22_w[e] + 2.0*Ch[2,2]*dnu
            dobj = 2.0*(nuhat-nu_g)*dnu + dnu*dnu
            if x[e]:
                s_Ey[e]  = -dEy
                s_obj[e] = -dobj
            else:
                s_Ey[e]  =  dEy
                s_obj[e] =  dobj
        raw_obj = s_obj + beta/N
        obj = fnu_g + beta*vol
        fil_Ey  = Sf @ s_Ey
        fil_obj = Sf @ raw_obj
        mom_obj = momentum*mom_obj + (1.0-momentum)*fil_obj/max(abs(fil_obj))
        mom_obj = mom_obj/max(abs(mom_obj))
        # store data
        list_ptr2inp += [ptr]
        list_top     += [x.copy()]
        list_dis_xx  += [Ug[:,0].copy()]
        list_dis_yy  += [Ug[:,1].copy()]
        list_dis_xy  += [Ug[:,2].copy()]
        list_dC00_0  += [dC00_0.copy()]
        list_dC00_1  += [dC00_1.copy()]
        list_dC00_2  += [dC00_2.copy()]
        list_dC00_w  += [dC00_w.copy()]
        list_dC11_0  += [dC11_0.copy()]
        list_dC11_1  += [dC11_1.copy()]
        list_dC11_2  += [dC11_2.copy()]
        list_dC11_w  += [dC11_w.copy()]
        list_dC22_0  += [dC22_0.copy()]
        list_dC22_1  += [dC22_1.copy()]
        list_dC22_2  += [dC22_2.copy()]
        list_dC22_w  += [dC22_w.copy()]
        list_nu      += [nuhat]
        list_Ey      += [Eyhat]
```

```
            list_vol     += [vol]
            # stopping criterion
            if ((obj<(1.0-small)*obj_opt) or (abs(nuhat-nuval)<(1.0-small)*abs(nu_opt-nuval))) and (Eyhat>Eymin-small):
                waiting = 0
                obj_opt = obj
                if (abs(nuhat-nuval)<(1.0-small)*abs(nu_opt-nuval)):
                    # update optimized topology
                    top_opt = x.copy()
                    nu_opt  = nuhat
                    Ey_opt  = Eyhat
            else:
                waiting += 1
                # check convergence
                if waiting == patience:
                    keep_going = False
            t1 = perf_counter()
            time_array[4] += (t1-t0)
```

After concluding the current optimization process, all remaining disconnected solid elements are removed from the optimized topology. This produces the unfixed bug presented in subsection 3.7.

```
basecell_silp
        # remove remaining islands from optimized topology
        y = top_opt.copy()
        voly = sum(y)/N
        continent = np.zeros(N,dtype=bool)
        continent_vol = 0.0
        for e in (list(range(0,Ns))+list(range(Ns,N,Ns))):
            if y[e] and (not continent[e]):
                continent = np.zeros(N,dtype=bool)
                visit(e,y,continent,neighbors)
                continent_vol = sum(continent)/N
            if continent_vol > 0.50*voly:
                break
        if continent_vol > 0.50*voly:
            islands = np.argwhere(y!=continent).ravel()
            y[islands] = False
        if any(top_opt!=y):
            if any(x!=y):
                elist = list(np.argwhere(x!=y)[:,0])
                Ug, Kr = update(x,etype,sym,pk,Ketvec,P,Kg_coo,Zr,shift,Uhat,factor,elist)
                Kg_csc = Kg_coo.tocsc()
                Ch = Ug.T @ Kg_csc @ Ug
                gamma = Ch[2,2]/Ch[0,0]
                nuhat = 1-2*Ch[2,2]/Ch[0,0]
                Eyhat = 4*Ch[2,2]*(Ch[0,0]-Ch[2,2])/Ch[0,0]
            top_opt = x.copy()
            nu_opt = nuhat
            Ey_opt = Eyhat
```

The total time to perform the current optimization process is stored in “time_array[5]”. 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[6]”. The execution times are written in the time log. The mechanical properties of the optimized topology are written in the input-output log. The optimized topology and the corresponding mechanical properties are appended to “list_top_opt”, “list_nu_opt” and “list_Ey_opt”. 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.

```
basecell_silp
        # write in log
        tlog.write(' ({:4d} x ):'.format(it))
        time_array[5] = sum(time_array[:5])
        time_array[:5] = time_array[:5]/it
        time_array[6] = (1+small)*it
        tlog.write(' {:6.3f} s : {:6.3f} s : {:6.3f} s : {:6.3f} s : {:6.3f} s ||'.format(
                    time_array[0],time_array[1],time_array[2],time_array[3],time_array[4]))
        tlog.write(' {:7.1f} s\n'.format(time_array[5]))
        iolog.write(' {:5.2f} :'.format(nu_opt))
        iolog.write(' {:5.3f} ||'.format(Ey_opt))
        iolog.write(begin)
        iolog.write(datetime.now().strftime(' %y/%m/%d-%H:%M:%S\n'))
        # store data
        list_top_opt += [top_opt.copy()]
        list_nu_opt  += [nu_opt]
        list_Ey_opt  += [Ey_opt]
        list_tim     += [time_array.copy()]
        # update pointer
        ptr += 1
        # prepare to open next input file
        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.

basecell_silp
```
    size_list = len(list_ptr2inp)
    list_ptr2opt += [size_list]
    # save files
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/fid.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_fid,dtype=np.uint32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/inp.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_inp,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/top_opt.npy'.format(
        fid_ini,fid_lim-1,file),np.packbits(np.array(list_top_opt),axis=1))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/nu_opt.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_nu_opt,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/Ey_opt.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_Ey_opt,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/ptr2opt.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_ptr2opt,dtype=np.uint32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/ptr2inp.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_ptr2inp,dtype=np.uint32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/top.npy'.format(
        fid_ini,fid_lim-1,file),np.packbits(np.array(list_top),axis=1))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dis_xx.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dis_xx,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dis_yy.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dis_yy,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dis_xy.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dis_xy,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC00_0.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC00_0,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC00_1.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC00_1,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC00_2.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC00_2,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC00_w.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC00_w,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC11_0.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC11_0,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC11_1.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC11_1,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC11_2.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC11_2,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC11_w.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC11_w,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC22_0.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC22_0,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC22_1.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC22_1,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC22_2.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC22_2,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/dC22_w.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_dC22_w,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/nu.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_nu,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/Ey.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_Ey,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/vol.npy'.format(
        fid_ini,fid_lim-1,file),np.array(list_vol,dtype=np.float32))
    np.save('./output/run_{:05d}_{:05d}/file_{:04d}/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_nu_opt, list_Ey_opt, list_ptr2opt, list_ptr2inp, list_top
    del list_dis_xx, list_dis_yy, list_dis_xy, list_dC00_0, list_dC00_1, list_dC00_2, list_dC00_w
    del list_dC11_0, list_dC11_1, list_dC11_2, list_dC11_w, list_dC22_0, list_dC22_1, list_dC22_2, list_dC22_w
    del list_nu, list_Ey, list_vol, list_tim
    gc.collect()
    # prepare to write next output file
    file += 1
```

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

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

3.3.3 ./source/python/mesh.py

This script generates the hexagonal mesh for the base cell, and the extended mesh used to perform filtering operations and to identify directly connected neighbors. Firstly, the numpy module is imported.

mesh
```
import numpy as np
```

The function “get_mesh” is defined. It generates the matrix “coor” with the coordinates of each node; the matrix “inci” with the nodes corresponding to each quadrilateral element; the array “etype” with the types of each quadrilateral element; and the matrix “sym” which relates each design variable with its six corresponding quadrilateral elements.

mesh

```python
def get_mesh(Ns, Lex, Ley):
    N = Ns**2
    Nt = 6*N
    M = 1 + 6*Ns*(Ns+1)
    # coordinates matrix
    coor = np.ndarray((M,2))
    coor[0,:] = [0.0,0.0]
    for k in range(Ns):
        num = 1 + 6*k*(k+1)
        pts = np.array([[-2*k*Lex-1.5*Lex ,       -0.5*Ley],
                        [      -(k+1)*Lex ,     -(k+1)*Ley],
                        [          -k*Lex ,     -(k+1)*Ley],
                        [       (k+1)*Lex ,     -(k+1)*Ley],
                        [ k*Lex + 1.5*Lex , -k*Ley-0.5*Ley],
                        [     2*(k+1)*Lex ,            0.0],
                        [ 2*k*Lex+1.5*Lex ,        0.5*Ley],
                        [       (k+1)*Lex ,      (k+1)*Ley],
                        [           k*Lex ,      (k+1)*Ley],
                        [      -(k+1)*Lex ,      (k+1)*Ley],
                        [  -k*Lex-1.5*Lex ,  k*Ley+0.5*Ley],
                        [    -2*(k+1)*Lex ,            0.0]])
        for kk in range(6):
            coor[num+2*kk*(k+1):num+2*(kk+1)*(k+1),0] = np.linspace(pts[2*kk,0],pts[2*kk+1,0],2*(k+1))
            coor[num+2*kk*(k+1):num+2*(kk+1)*(k+1),1] = np.linspace(pts[2*kk,1],pts[2*kk+1,1],2*(k+1))
    # incidence matrix
    inci = np.ndarray((Nt,4),dtype=np.uint32)
    etype = np.ndarray((Nt),dtype=np.uint8)
    for k in range(Ns):
        for kk in range(6):
            ek = 6*(k**2) + 2*(kk+1)*k + kk
            etype[ek] = kk
            inci[ek,0] = (2*kk+1) + (14+2*kk)*k + 6*k*(k-1)
            inci[ek,1] = (2*kk+2) + (14+2*kk)*k + 6*k*(k-1)
            if kk == 5:
                inci[ek,2] = (2*kk+3) + (14+2*kk)*(k-1) + 6*(k-1)*(k-2)
            else:
                inci[ek,2] = (2*kk+3) + (14+2*kk)*k + 6*k*(k-1)
            inci[ek,3] = (2*kk+2) + (14+2*kk)*(k-1) + 6*(k-1)*(k-2)
    for k in range(1,Ns):
        for kk in range(6):
            ek = 6 + 18*(k-1) + 6*(k-2)*(k-1) + (2*k+1)*kk
            etype[ek:ek+2*k] = 6 + kk
            kkk = (kk-1)%6
            if kkk == 5:
                inci[ek:ek+2*k,0] = (2*kkk+3) + (14+2*kkk)*(k-1) + 6*(k-1)*(k-2)
                inci[ek:ek+2*k,1] = (2*kkk+4) + (14+2*kkk)*(k-1) + 6*(k-1)*(k-2)
                inci[ek:ek+2*k,2] = (2*kkk+3) + (14+2*kkk)*(k-2) + 6*(k-2)*(k-3)
                inci[ek:ek+2*k,3] = (2*kkk+2) + (14+2*kkk)*(k-2) + 6*(k-2)*(k-3)
            else:
                inci[ek:ek+2*k,0] = (2*kkk+3) + (14+2*kkk)*k + 6*k*(k-1)
                inci[ek:ek+2*k,1] = (2*kkk+4) + (14+2*kkk)*k + 6*k*(k-1)
                inci[ek:ek+2*k,2] = (2*kkk+3) + (14+2*kkk)*(k-1) + 6*(k-1)*(k-2)
                inci[ek:ek+2*k,3] = (2*kkk+2) + (14+2*kkk)*(k-1) + 6*(k-1)*(k-2)
            inci[ek:ek+2*k,:] = inci[ek:ek+2*k,:] + np.arange(2*k).reshape(2*k,1)
        ek = 6 + 18*(k-1) + 6*(k-2)*(k-1)
        inci[ek,3] = 12 + 24*(k-1) + 6*(k-1)*(k-2)
    # D3-symmetry map
    sym = np.ndarray((N,6),dtype=np.uint32)
    for k in range(Ns):
        for kk in range(6):
            ek = 6*(Ns-k-1)**2 + 2*(kk+1)*(Ns-k-1) + kk
            kkk = 1-2*(kk%2)
            sym[k*(Ns+1)::Ns,kk] = np.array(range(ek,ek+kkk*(Ns-k),kkk))
            if kk == 5:
                ek = 6*(Ns-k-2)**2 + 2*(kk+1)*(Ns-k-2) + kk
            sym[k*(Ns+1)+1:(k+1)*Ns,kk] = np.array(range(ek-kkk,ek-kkk*(Ns-k),-kkk))
    return coor, inci, etype, sym
```

The function “get_fmesh” is defined. It generates the matrix “coor_lb” with the coordinates of the nodes of the cell located at the left-bottom of the main cell; the matrix “coor_bot” with the coordinates of the nodes of the cell located at the bottom of the main cell; the matrix “inci_lb” with the nodes corresponding to each quadrilateral element of the left-bottom cell; the matrix “inci_bot” with the nodes corresponding to each quadrilateral element of the bottom cell; the matrix “sym_lb” which relates each design variable with its two corresponding quadrilateral elements of the left-bottom cell; and the matrix “sym_bot” which relates each design variable with its two corresponding quadrilateral elements of the bottom cell.

mesh

```python
def get_fmesh(Ns, Lx, Ly, Lex, Ley):
    N = Ns**2
    M = 1 + 6*Ns*(Ns+1)
    # left-bottom cell
    Mlb = 2*N + Ns
    coor_lb = np.ndarray((Mlb,2))
    coor_lb[0,:] = [0.0,0.0]
    for k in range(Ns-1):
        num = 1 + 5*k + 2*(k-1)*k
        pts = np.array([[   1.5*(k+1)*Lex , -0.5*(k+1)*Ley],
                        [     2*(k+1)*Lex ,            0.0],
                        [ 2*k*Lex+1.5*Lex ,        0.5*Ley],
                        [       (k+1)*Lex ,      (k+1)*Ley],
                        [           k*Lex ,      (k+1)*Ley],
                        [             0.0 ,      (k+1)*Ley]])
        coor_lb[num      :num+  k+2,0] = np.linspace(pts[0,0],pts[1,0],  k+2)
        coor_lb[num      :num+  k+2,1] = np.linspace(pts[0,1],pts[1,1],  k+2)
        coor_lb[num+  k+2:num+3*k+4,0] = np.linspace(pts[2,0],pts[3,0],2*k+2)
        coor_lb[num+  k+2:num+3*k+4,1] = np.linspace(pts[2,1],pts[3,1],2*k+2)
        coor_lb[num+3*k+4:num+4*k+5,0] = np.linspace(pts[4,0],pts[5,0],  k+1)
        coor_lb[num+3*k+4:num+4*k+5,1] = np.linspace(pts[4,1],pts[5,1],  k+1)
    k = Ns-1
    num = 1 + 5*k + 2*(k-1)*k
    pts = np.array([[   1.5*(k+1)*Lex , -0.5*(k+1)*Ley],
                    [ 2*k*Lex+1.5*Lex ,       -0.5*Ley],
                    [           k*Lex ,      (k+1)*Ley],
                    [             0.0 ,      (k+1)*Ley]])
    coor_lb[num    :num+  k+1,0] = np.linspace(pts[0,0],pts[1,0], k+1)
    coor_lb[num    :num+  k+1,1] = np.linspace(pts[0,1],pts[1,1], k+1)
    coor_lb[num+k+1:num+2*k+2,0] = np.linspace(pts[2,0],pts[3,0], k+1)
    coor_lb[num+k+1:num+2*k+2,1] = np.linspace(pts[2,1],pts[3,1], k+1)
    coor_lb = coor_lb - np.array([3*Lx,Ly])
    inci_lb = np.ndarray((2*N,4),dtype=np.uint32)
    etype_lb = np.ndarray((2*N),dtype=np.int8)
    for k in range(Ns-1):
        for kk in range(2):
            ek = 2*((k+kk)**2) + k - kk*(2*k+1)
            etype_lb[ek] = 2+kk
            id1 = 1 + 5*(k+kk) + 2*(k-1+kk)*(k+kk) + (k+1) - 2*kk*(k+1) - kk
            id3 = 1 + 5*(k-1+kk) + 2*(k-2+kk)*(k-1+kk) + k - 2*kk*k - kk
            inci_lb[ek,0] = id1-1
            inci_lb[ek,1] = id1
            inci_lb[ek,2] = id1+1
            inci_lb[ek,3] = id3
    k = Ns-1
    for kk in range(2):
        ek = 2*((k+kk)**2) + k - kk*(2*k+1)
        etype_lb[ek] = 2+kk
        if kk == 0:
            inci_lb[ek,0] = 2*(k**2) + 4*k + 1
            inci_lb[ek,1] = 6*k*(k+1) + 2*Ns - M
            inci_lb[ek,2] = 6*k*(k+1) + 2*Ns - 1 - M
            inci_lb[ek,3] = 2*(k**2)
        else:
            inci_lb[ek,0] = 6*k*(k+1) + 1 - M
            inci_lb[ek,1] = 6*(k+1)*(k+2) - M
            inci_lb[ek,2] = 2*(k**2) + 4*k + 3
            inci_lb[ek,3] = 2*(k**2) + 2*k
    for k in range(1,Ns-1):
        kkk = np.tile(np.arange(4*k),(2,1))
        kkk[0,  k:] = kkk[0,  k:] + 2
        kkk[0,3*k:] = kkk[0,3*k:] + 2
        for kk in range(3):
            ek = 2*(k**2) + kk + k*((kk*(kk+1))//2)
            etype_lb[ek:ek+((2-kk)*kk+1)*k] = 8 + kk
            n0 = 1 + 5*k + 2*(k-1)*k
            n1 = n0 + 1
            n2 = 2 + 5*(k-1) + 2*(k-2)*(k-1)
            n3 = n2 - 1
            inci_lb[ek:ek+((2-kk)*kk+1)*k,0] = n0 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
            inci_lb[ek:ek+((2-kk)*kk+1)*k,1] = n1 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
            inci_lb[ek:ek+((2-kk)*kk+1)*k,2] = n2 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
            inci_lb[ek:ek+((2-kk)*kk+1)*k,3] = n3 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
    k = Ns-1
    kkk = np.tile(np.arange(4*k),(2,1))
    kkk[0,k:3*k] = np.arange(0,-2*Ns+2,-1)
    kkk[0,3*k:] = kkk[0,3*k:] - 2*k + 1
    for kk in range(3):
        ek = 2*(k**2) + kk + k*((kk*(kk+1))//2)
        etype_lb[ek:ek+((2-kk)*kk+1)*k] = 8 + kk
        if kk == 1:
            n0 = 1 + 6*k*(k+1) + 2*(Ns-1) - M
            n1 = n0 - 1
        else:
            n0 = 1 + 5*k + 2*(k-1)*k
            n1 = n0 + 1
        n2 = 2 + 5*(k-1) + 2*(k-2)*(k-1)
        n3 = n2 - 1
        inci_lb[ek:ek+((2-kk)*kk+1)*k,0] = n0 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
        inci_lb[ek:ek+((2-kk)*kk+1)*k,1] = n1 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
        inci_lb[ek:ek+((2-kk)*kk+1)*k,2] = n2 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
```

```
            inci_lb[ek:ek+((2-kk)*kk+1)*k,3] = n3 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
        inci_lb = inci_lb + M
        sym_lb = np.ndarray((N,2),dtype=np.uint32)
        for k in range(Ns):
            for kk in range(2):
                ek = 2*((k+kk)**2) + k - kk*(2*k+1)
                sym_lb[N-1-k*(Ns+1):N-k*Ns,kk] = np.arange(ek,ek+(2*kk-1)*(k+1),2*kk-1)
        for k in range(Ns-1):
            for kk in range(2):
                sym_lb[(k+1)*Ns+k::Ns,kk] = np.arange(sym_lb[k*Ns+k,kk]+1-2*kk,sym_lb[k*Ns+k,kk]+(1-2*kk)*(Ns-k),1-2*kk)
        sym_lb = sym_lb + 6*N
        # bottom cell
        Mbot = 2*N
        coor_bot = np.ndarray((Mbot,2))
        coor_bot[0,:] = [0.0,0.0]
        for k in range(Ns-1):
            num = 1 + 5*k + 2*(k-1)*k
            pts = np.array([[  1.5*(k+1)*Lex , 0.5*(k+1)*Ley],
                            [      (k+1)*Lex ,     (k+1)*Ley],
                            [          k*Lex ,     (k+1)*Ley],
                            [     -(k+1)*Lex ,     (k+1)*Ley],
                            [ -k*Lex-1.5*Lex , k*Ley+0.5*Ley],
                            [ -1.5*(k+1)*Lex , 0.5*(k+1)*Ley]])
            coor_bot[num      :num+  k+2,0] = np.linspace(pts[0,0],pts[1,0],  k+2)
            coor_bot[num      :num+  k+2,1] = np.linspace(pts[0,1],pts[1,1],  k+2)
            coor_bot[num+  k+2:num+3*k+4,0] = np.linspace(pts[2,0],pts[3,0],2*k+2)
            coor_bot[num+  k+2:num+3*k+4,1] = np.linspace(pts[2,1],pts[3,1],2*k+2)
            coor_bot[num+3*k+4:num+4*k+5,0] = np.linspace(pts[4,0],pts[5,0],  k+1)
            coor_bot[num+3*k+4:num+4*k+5,1] = np.linspace(pts[4,1],pts[5,1],  k+1)
        k = Ns-1
        num = 1 + 5*k + 2*(k-1)*k
        pts = np.array([[   1.5*(k+1)*Lex , 0.5*(k+1)*Ley],
                        [ k*Lex + 1.5*Lex , k*Ley+0.5*Ley]])
        coor_bot[num:num+k+1,0] = np.linspace(pts[0,0],pts[1,0], k+1)
        coor_bot[num:num+k+1,1] = np.linspace(pts[0,1],pts[1,1], k+1)
        coor_bot = coor_bot - np.array([0.0,2*Ly])
        inci_bot = np.ndarray((2*N,4),dtype=np.uint32)
        etype_bot = np.ndarray((2*N),dtype=np.int8)
        for k in range(Ns-1):
            for kk in range(2):
                ek = 2*((k+kk)**2) + k - kk*(2*k+1)
                etype_bot[ek] = 3+kk
                id1 = 1 + 5*(k+kk) + 2*(k-1+kk)*(k+kk) + (k+1) - 2*kk*(k+1) - kk
                id3 = 1 + 5*(k-1+kk) + 2*(k-2+kk)*(k-1+kk) + k - 2*kk*k - kk
                inci_bot[ek,0] = id1-1
                inci_bot[ek,1] = id1
                inci_bot[ek,2] = id1+1
                inci_bot[ek,3] = id3
        k = Ns-1
        for kk in range(2):
            ek = 2*((k+kk)**2) + k - kk*(2*k+1)
            etype_bot[ek] = 3+kk
            if kk == 0:
                inci_bot[ek,0] = 2*(k**2) + 4*k + 1
                inci_bot[ek,1] = 6*k*(k+1) + 4*Ns - M - Mlb
                inci_bot[ek,2] = 6*k*(k+1) + 4*Ns - 1 - M - Mlb
                inci_bot[ek,3] = 2*(k**2)
            else:
                inci_bot[ek,0] = 6*k*(k+1) + 2*Ns + 1 - M - Mlb
                inci_bot[ek,1] = 6*k*(k+1) + 2*Ns - M - Mlb
                inci_bot[ek,2] = 2*(k**2) + 4*k + 1 - Mlb
                inci_bot[ek,3] = 2*(k**2) + 2*k
        for k in range(1,Ns-1):
            kkk = np.tile(np.arange(4*k),(2,1))
            kkk[0,  k:] = kkk[0,  k:] + 2
            kkk[0,3*k:] = kkk[0,3*k:] + 2
            for kk in range(3):
                ek = 2*(k**2) + kk + k*((kk*(kk+1))//2)
                etype_bot[ek:ek+((2-kk)*kk+1)*k] = 9 + kk
                n0 = 1 + 5*k + 2*(k-1)*k
                n1 = n0 + 1
                n2 = 2 + 5*(k-1) + 2*(k-2)*(k-1)
                n3 = n2 - 1
                inci_bot[ek:ek+((2-kk)*kk+1)*k,0] = n0 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
                inci_bot[ek:ek+((2-kk)*kk+1)*k,1] = n1 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
                inci_bot[ek:ek+((2-kk)*kk+1)*k,2] = n2 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
                inci_bot[ek:ek+((2-kk)*kk+1)*k,3] = n3 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
        k = Ns-1
        kkk = np.tile(np.arange(4*k),(2,1))
        kkk[0,k:3*k] = np.arange(0,-2*Ns+2,-1)
        kkk[0,3*k:] = np.arange(k,0,-1) - Mlb
        for kk in range(3):
            ek = 2*(k**2) + kk + k*((kk*(kk+1))//2)
            etype_bot[ek:ek+((2-kk)*kk+1)*k] = 9 + kk
            if kk == 1:
                n0 = 4*Ns -1 + 6*k*(k+1) - M - Mlb
                n1 = n0 - 1
            else:
                n0 = 1 + 5*k + 2*(k-1)*k
                n1 = n0 + 1 - kk
            n2 = 2 + 5*(k-1) + 2*(k-2)*(k-1)
            n3 = n2 - 1
            inci_bot[ek:ek+((2-kk)*kk+1)*k,0] = n0 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
            inci_bot[ek:ek+((2-kk)*kk+1)*k,1] = n1 + kkk[0,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
            inci_bot[ek:ek+((2-kk)*kk+1)*k,2] = n2 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
            inci_bot[ek:ek+((2-kk)*kk+1)*k,3] = n3 + kkk[1,((kk*(kk+1))//2)*k:((-(kk**2)+5*kk+2)//2)*k]
        inci_bot = inci_bot + M + Mlb
        sym_bot = np.ndarray((N,2),dtype=np.uint32)
        for k in range(Ns):
            for kk in range(2):
                ek = 2*((k+kk)**2) + k - kk*(2*k+1)
                sym_bot[N-1-k*(Ns+1):N-k*Ns,kk] = np.arange(ek,ek+(1-2*kk)*(k+1),1-2*kk)
        for k in range(Ns-1):
            for kk in range(2):
                sym_bot[(k+1)*Ns+k::Ns,kk] = np.arange(sym_bot[k*Ns+k,kk]+2*kk-1,sym_bot[k*Ns+k,kk]+(2*kk-1)*(Ns-k),2*kk-1)
        sym_bot = sym_bot + 8*N
        return coor_lb, coor_bot, inci_lb, inci_bot, sym_lb, sym_bot
```

3.3.4 ./source/python/elem.py

This script generates the local elemental stiffness matrices for each type of quadrilateral element, and for each type of augmented element. Firstly, the numpy module is imported.

elem

```
import numpy as np
```

The function "getJ" is defined. Considering an isoparametric bilinear quadrilateral element, whose coordinates are stored in the matrix "sv", for given coordinates "xi" and "eta" (defined in $[-1,1] \times [-1,1]$), the Jacobian matrix, its inverse and its determinant are computed.

elem

```
def getJ(xi,eta,sv):
    dN = 0.25*np.array([[-(1-eta), (1-eta), (1+eta), -(1+eta)],
                        [ -(1-xi), -(1+xi),  (1+xi),   (1-xi)]])
    J = dN @ sv
    Jdet = J[0,0]*J[1,1] - J[0,1]*J[1,0]
    Jinv = (1/Jdet) * np.array([[ J[1,1], -J[0,1]],
                                [-J[1,0],  J[0,0]]])
    return Jinv, Jdet
```

The function "getB" is defined. Considering the quadrilateral element defined by the nodal coordinates "sv", for given coordinates "xi" and "eta", the matrix "Bmat" is computed. When it is applied on the nodal displacements vector, it yields the strain vector (in Voigt notation) at the position defined by "xi" and "eta".

elem

```
def getB(xi,eta,sv):
    Jinv, Jdet = getJ(xi,eta,sv)
    Ai = np.array([[1,0,0,0],
                   [0,0,0,1],
                   [0,1,1,0]])
    Aj = np.zeros((4,4))
    Aj[0:2,0:2] = Jinv
    Aj[2:4,2:4] = Jinv
    Ak = 0.25*np.array([[-(1-eta), 0.0, (1-eta), 0.0, (1+eta), 0.0, -(1+eta), 0.0],
                        [ -(1-xi), 0.0, -(1+xi), 0.0,  (1+xi), 0.0,   (1-xi), 0.0],
                        [0.0, -(1-eta), 0.0, (1-eta), 0.0, (1+eta), 0.0, -(1+eta)],
                        [0.0,  -(1-xi), 0.0, -(1+xi), 0.0,  (1+xi), 0.0,   (1-xi)]])
    Bmat = Ai @ Aj @ Ak
    return Bmat, Jdet
```

The function "get_emat" is defined. Given the Young's modulus ("Ey") and the Poisson's ratio ("nu") of an isotropic material in plane stress state, the corresponding constitutive matrix is stored in "Ce". Since the elemental stiffness matrix does no depend on the scale of the element, elements with shorter sides of $1.0\,m$ and longer sides of $\sqrt{3}\,m$ are considered. The stiffness matrices, for each of the twelve types of quadrilateral elements, are computed through Gaussian quadrature with 2×2 points. They are stored in the tensor "Ket", of dimensions $12 \times 8 \times 8$.

elem

```
def get_emat(Ey,nu):
    # constitutive matrix
    Ce = (Ey/(1-nu**2))*np.array([[1,nu,0],[nu,1,0],[0,0,(1-nu)/2]])
    # elements nodes
    sv = np.ndarray((12,4,2))
    s3 = np.sqrt(3)
    sv[0,:,:] = np.array([[-0.5,  0.5*s3],
                          [ 0.0,  0.0   ],
                           [ 1.0,  0.0   ],
                           [ 1.0,      s3]])
    sv[1,:,:]  = np.array([[-1.0,  0.0   ],
                           [ 0.0,  0.0   ],
                           [ 0.5,  0.5*s3],
                           [-1.0,      s3]])
    sv[2,:,:]  = np.array([[-0.5, -0.5*s3],
                           [ 0.0,  0.0   ],
                           [-0.5,  0.5*s3],
                           [-2.0,  0.0   ]])
    sv[3,:,:]  = np.array([[ 0.5, -0.5*s3],
                           [ 0.0,  0.0   ],
                           [-1.0,  0.0   ],
                           [-1.0,     -s3]])
    sv[4,:,:]  = np.array([[ 1.0,  0.0   ],
                           [ 0.0,  0.0   ],
                           [-0.5, -0.5*s3],
                           [ 1.0,     -s3]])
    sv[5,:,:]  = np.array([[ 0.5,  0.5*s3],
                           [ 0.0,  0.0   ],
                           [ 0.5, -0.5*s3],
                           [ 2.0,  0.0   ]])
    sv[6,:,:]  = np.array([[ 0.0,  0.0   ],
                           [ 0.5, -0.5*s3],
                           [ 2.0,  0.0   ],
                           [ 1.5,  0.5*s3]])
    sv[7,:,:]  = np.array([[ 0.0,  0.0   ],
                           [ 1.0,  0.0   ],
                           [ 1.0,      s3],
                           [ 0.0,      s3]])
    sv[8,:,:]  = np.array([[ 0.0,  0.0   ],
                           [ 0.5,  0.5*s3],
                           [-1.0,      s3],
                           [-1.5,  0.5*s3]])
    sv[9,:,:]  = np.array([[ 0.0,  0.0   ],
                           [-0.5,  0.5*s3],
                           [-2.0,  0.0   ],
                           [-1.5, -0.5*s3]])
    sv[10,:,:] = np.array([[ 0.0,  0.0   ],
                           [-1.0,  0.0   ],
                           [-1.0,     -s3],
                           [ 0.0,     -s3]])
    sv[11,:,:] = np.array([[ 0.0,  0.0   ],
                           [-0.5, -0.5*s3],
                           [ 1.0,     -s3],
                           [ 1.5, -0.5*s3]])
    # stiffness matrices
    Ket = np.ndarray((12,8,8))
    gpoint = s3/3
    for ek in range(12):
        B0, J0 = getB(-gpoint, -gpoint, sv[ek,:,:])
        B1, J1 = getB( gpoint, -gpoint, sv[ek,:,:])
        B2, J2 = getB( gpoint,  gpoint, sv[ek,:,:])
        B3, J3 = getB(-gpoint,  gpoint, sv[ek,:,:])
        Ket[ek,:,:] = J0 * B0.T @ Ce @ B0 + J1 * B1.T @ Ce @ B1 + J2 * B2.T @ Ce @ B2 + J3 * B3.T @ Ce @ B3
        Ket[ek,:,:] = 0.5*(Ket[ek,:,:]+Ket[ek,:,:].T)
    return Ket
```

The function “get_augmat” is defined. It computes the stiffness variation matrices, stored in “dKelist”, and their factorizations (Equation 27), stored in “Hlist”, for the six different types of augmented elements.

elem

```
def get_augmat(Ns,inci,etype,sym,dKe,small=1e-14):
    N = Ns**2  # number of elements in the design domain
    # different unconstrained augmented elements
    aug_etype = np.ndarray((N),dtype=np.uint8)
    aug_etype[0:N-1:Ns+1]                 = 0
    for k in range(Ns-2):
        aug_etype[k*(Ns+1)+1:(k+1)*Ns-1]    = 1
        aug_etype[(k+1)*(Ns+1)-1:N-Ns-1:Ns] = 2
    aug_etype[Ns-1:N-1:Ns]                = 3
    aug_etype[N-Ns:N-1]                   = 4
    aug_etype[N-1]                        = 5
    # factorization
    Hlist = []
    dKelist = []
    eledofs = np.ndarray((6,8),dtype=np.uint32)
    locdofs = np.ndarray((6,8),dtype=np.uint32)
    for e in [0,1,Ns,Ns-1,N-Ns,N-1]:
        for k in range(6):
            et = sym[e,k]
            n0 = inci[et,0]
            n1 = inci[et,1]
            n2 = inci[et,2]
            n3 = inci[et,3]
            eledofs[k,:] = np.array([2*n0,2*n0+1,2*n1,2*n1+1,2*n2,2*n2+1,2*n3,2*n3+1])
        size = len(np.unique(eledofs))
        elebool = np.ones((6,8),dtype=bool)
        kkk = 0
        for k in range(6):
            for kk in range(8):
                if elebool[k,kk]:
                    mask = (eledofs==eledofs[k,kk])
                    locdofs[mask] = kkk
                    elebool[mask] = False
                    kkk = kkk + 1
        aug_dKe = np.zeros((size,size))
        for k in range(6):
            et = sym[e,k]
            ek = etype[et]
            grid = np.ix_(locdofs[k,:],locdofs[k,:])
            aug_dKe[grid] = aug_dKe[grid] + dKe[ek,:,:]
        D,V = np.linalg.eigh(aug_dKe)
        mask = abs(D) > small
        D = D[mask]
        V = V[:,mask]
        He = V*np.sqrt(D)
        Hlist = Hlist + [He]
        dKelist = dKelist + [aug_dKe]
    return aug_etype, Hlist, dKelist
```

3.3.5 ./source/python/filters.py

This script generates the matrices used in the filtering procedures: smoothing sensitivity maps; and applying morphological operators on the density map. Firstly, the necessary modules are imported.

filters

```
import numpy as np
from scipy.sparse import coo_matrix
```

The function "get_sfil" is defined. According to the conical filter radius ("rsen"), the matrix "Sf" is computed. The filtered sensitivity vectors are obtained by applying this matrix on the raw sensitivity vectors.

filters

```
def get_sfil(N,sym,elepos,Q,rsen):
    clist = []
    csize = np.ndarray((N),dtype=np.uint32)
    for e in range(N):
        et = sym[e,0]
        c = np.argwhere(np.sum((elepos[et,:]-elepos)**2,axis=1) <= rsen**2)
        clist = clist + [c[:,0]]
        csize[e] = len(c)
    size = sum(csize)
    row = np.ndarray((size),dtype=np.uint32)
    col = np.ndarray((size),dtype=np.uint32)
    data = np.ndarray((size))
    i = 0
    for e in range(N):
        et = sym[e,0]
        c = clist[e]
        num = csize[e]
        weights = rsen - np.linalg.norm(elepos[et,:]-elepos[c,:],axis=1)
        weights = weights/sum(weights)
        row[i:i+num] = np.repeat(e,num)
        col[i:i+num] = c
        data[i:i+num] = weights
        i = i + num
    Sf = coo_matrix((data,(row,col)),shape=(N,10*N))
    Sf = Sf.tocsr()
    Sf = Sf @ Q
    return Sf
```

The function "get_mope" is defined. According to the radius "rmor", the matrix "Mf" is computed. It is used to identify elements within range, when applying the morphological operators (erosion or dilation).

filters

```
def get_mope(N,sym,elepos,Q,rmor):
    clist = []
    csize = np.ndarray((N),dtype=np.uint32)
    for e in range(N):
        et = sym[e,0]
        c = np.argwhere(np.sum((elepos[et,:]-elepos)**2,axis=1) <= rmor**2)
        clist = clist + [c[:,0]]
        csize[e] = len(c)
    size = sum(csize)
    row = np.ndarray((size),dtype=np.uint32)
    col = np.ndarray((size),dtype=np.uint32)
    data = np.ndarray((size))
    i = 0
    for e in range(N):
        et = sym[e,0]
        c = clist[e]
        num = csize[e]
        row[i:i+num] = np.repeat(e,num)
        col[i:i+num] = c
        data[i:i+num] = 1.0
        i = i + num
    Mf = coo_matrix((data,(row,col)),shape=(N,10*N))
    Mf = Mf.tocsr()
    Mf = Mf @ Q
    return Mf
```

3.3.6 ./source/python/rem_islands.py

This script is used to identify disconnected structural components in the topology. Firstly, the necessary modules are imported.

rem_islands

```
import numpy as np
import sys
```

The recursion limit is increased, so that the recursive function is not interrupted before the procedure is concluded. The recursive function "visit" is defined. According to the neighbors matrix, it performs a depth-first search, visiting all solids which are connected to a given reference element. After the search is concluded, the boolean mask "continent" identifies all solids that are connected to the reference element. While solids that are not in "continent" correspond to structural components disconnected from the reference element.

rem_islands

```
sys.setrecursionlimit(100000)
def visit(e,x,continent,neighbors):
    continent[e] = True
    for ee in neighbors[e,:]:
        if x[ee] and (not continent[ee]):
            visit(ee,x,continent,neighbors)
    return
```

The function "get_neighbors" is defined. It computes the neighbors matrix. For each design variable, it stores the indices of the four design variables that correspond to directly connected neighbors. For elements on the boundaries of the design domain, some of these indices may be repeated. This does not hamper the considered procedure, since repeated indices will be automatically ignored by the depth-first search algorithm.

rem_islands

```
def get_neighbors(Ns,inci,inci_lb,inci_bot,sym,sym_lb,sym_bot):
    N = Ns**2
    Mf = 1 + 6*Ns*(Ns+1) + Ns + 4*N
    Nf = 10*N
    finci = np.vstack((inci,inci_lb,inci_bot))
    fsym = np.hstack((sym,sym_lb,sym_bot))
    nodes = -np.ones((Mf,6),dtype=int)
    counter = np.zeros(Mf,dtype=int)
    for e in range(Nf):
        for k in range(4):
            n = finci[e,k]
            nodes[n,counter[n]] = e
            counter[n] += 1
    neighbors_extended = -np.ones((Nf,4),dtype=int)
    counter = np.zeros(Nf,dtype=int)
    for e in range(Nf):
        for k in range(4):
            n = finci[e,k]
            for kk in range(6):
                ee = nodes[n,kk]
                if (ee != e) and (ee != -1):
                    if ee not in neighbors_extended[e,:]:
                        if len(np.setdiff1d(finci[e,:],finci[ee,:],assume_unique=True)) == 2:
                            neighbors_extended[e,counter[e]] = ee
                            counter[e] += 1
    neighbors_ext = neighbors_extended.copy()
    for ef in range(Nf):
        for k in range(4):
            if neighbors_extended[ef,k] != -1:
                e = np.argwhere(fsym==neighbors_extended[ef,k])[0,0]
                neighbors_ext[ef,k] = e
    neighbors = np.ndarray((N,4),dtype=int)
    for k in range(N):
        neighbors[k,:] = neighbors_ext[fsym[k,0],:]
    return neighbors
```

3.3.7 ./source/python/topopt.py

This script defines basic operations required to perform topology optimizations: the update of the topology; and the computation of sensitivity values. Only the WS approach is considered in this script, the CGS approach is implemented in the cython script "silp_sens.pyx", detailed in a following section. Firstly, the numpy module is imported.

topopt
```
import numpy as np
```

The function "update" is defined. According to the list of elements whose states must be switched ("elist"), the density vector ("x") and the COO unconstrained global stiffness matrix ("Kg_coo") are updated. If the flag "solve_sys" is *True*, the function returns the total displacements matrix ("Ug") and the CSC constrained global stiffness matrix ("Kr") of the updated topology. Otherwise, it returns nothing.

topopt
```
def update(x,etype,sym,pk,Ketvec,P,Kg_coo,Zr,shift,Uhat,factor,elist,solve_sys=True):
    x[elist] = ~x[elist]
    for e in elist:
        if x[e]:
            for k in range(6):
                et = sym[e,k]
                ek = etype[et]
                Kg_coo.data[64*et:64*et+64] = Ketvec[ek,:]
        else:
            for k in range(6):
                et = sym[e,k]
                ek = etype[et]
                Kg_coo.data[64*et:64*et+64] = pk*Ketvec[ek,:]
    if solve_sys:
        Kg_csc = Kg_coo.tocsc()
        Kr = P.T @ Kg_csc @ P
        Kr = Kr + shift*Zr
        Kr.sort_indices()
        Kr.data = Kr.data - shift*Zr.data
        Fr = -P.T @ Kg_csc @ Uhat
        factor.cholesky_inplace(Kr)
        Ur = factor(Fr)
        Ug = Uhat + P @ Ur
        return Ug, Kr
    return
```

The function "ws" is defined. It computes the exact sensitivity values of the diagonal terms of $\boldsymbol{C}$ through WS approach.

topopt
```
def ws(x,aug_etype,sym,P,factor,inci,Ug,Hlist):
    N = len(x)
    dC00_w = np.ndarray((N))
    dC11_w = np.ndarray((N))
    dC22_w = np.ndarray((N))
    eledofs = np.ndarray((6,8),dtype=np.uint32)
    for e in range(N):
        for k in range(6):
            et = sym[e,k]
            n0 = inci[et,0]
            n1 = inci[et,1]
            n2 = inci[et,2]
            n3 = inci[et,3]
            eledofs[k,:] = np.array([2*n0,2*n0+1,2*n1,2*n1+1,2*n2,2*n2+1,2*n3,2*n3+1])
        aug_ek = aug_etype[e]
        He = Hlist[aug_ek]
        size = He.shape
        elebool = np.ones((6,8),dtype=bool)
        aug_eledofs = np.ndarray((size[0]),dtype=np.uint32)
        kkk = 0
        for k in range(6):
            for kk in range(8):
                if elebool[k,kk]:
                    mask = (eledofs==eledofs[k,kk])
                    elebool[mask] = False
                    aug_eledofs[kkk] = eledofs[k,kk]
                    kkk = kkk + 1
        Ie = np.eye(size[1])
        Ue = Ug[aug_eledofs,:]
        Pe = P[aug_eledofs,:]
        Hr = Pe.T @ He
        Ahalf = factor.solve_L(factor.apply_P(Hr),use_LDLt_decomposition=False)
        Ae = Ahalf.T @ Ahalf
        Ve = He.T @ Ue
        if x[e]:
            dCh = -Ve.T @ np.linalg.solve(Ie-Ae,Ve)
        else:
            dCh =  Ve.T @ np.linalg.solve(Ie+Ae,Ve)
        dC00_w[e] = dCh[0,0]
        dC11_w[e] = dCh[1,1]
        dC22_w[e] = dCh[2,2]
    return dC00_w, dC11_w, dC22_w
```

3.3.8 ./source/python/ilp_solver.py

This script is used to solve each linearized subproblem. If there is no Young's modulus constraint, the BESO algorithm is used, otherwise, the pulp module is used to solve the problem with branch-and-bound simplex. Firstly, the necessary modules are imported.

ilp_solver

```
import numpy as np
import pulp as pp
```

The function "solve_ILP" is defined. It receives the number of design variables ("N"); the current topology vector ("x"); the sensitivity vector of the objective function ("alpha"); the sensitivity vector of the constraint function ("alpha_h"); the current value of the constraint function ("h_bar"); the tolerance of the constraint function ("h_lim"); the maximal topology variation ("dXmax"); the sense of the optimization ("sense"); and the sense of the constraint function ("sense_h"). Then, it returns the solution of the corresponding integer linear problem. Some optional parameter can be included: "tLim" sets a maximum time for the solver (in seconds); "rErr" sets a relative gap tolerance for the solver to stop; if "verbose" is *True* the solver's log is shown.

ilp_solver

```
def solve_ILP(N,x,alpha,alpha_h,h_bar,h_lim,dXmax,sense='min',sense_h='L',tLim=np.infty,rErr=None,verbose=False):
    # Integer Linear Programming problem
    if sense == 'min':  # minimization problem
        prob = pp.LpProblem('ILP', pp.LpMinimize)
    else:               # maximization problem
        prob = pp.LpProblem('ILP', pp.LpMaximize)
    variables = [f'{e:07d}' for e in range(N)]
    xvars = pp.LpVariable.dicts('x',variables,cat='Binary')
    # Objective Function
    # f(x) = f(xbar) + alpha * [x-xbar]
    # min/max { f(x) } = min/max { alpha * x }
    alpha_obj = dict(zip(variables,alpha))
    prob += pp.lpSum([alpha_obj[v] * xvars[v] for v in variables])
    # General Constraint
    # h(x) = h(xbar) + alpha_h * [x-xbar] <= (or >=) h_lim
    # alpha_h * x <= (or >=) b_h
    b_h = h_lim - h_bar + sum(alpha_h[x])
    alpha_hcon = dict(zip(variables,alpha_h))
    if sense_h == 'L':  # constraint : <=
        prob += pp.lpSum([alpha_hcon[v] * xvars[v] for v in variables]) <= b_h
    else:               # constraint : >=
        prob += pp.lpSum([alpha_hcon[v] * xvars[v] for v in variables]) >= b_h
    # Maximal Topological Change Constraint
    # g(x) = ||x-xbar||_1 <= dXmax
    # g(x) = g(xbar) + alpha_g * [x-xbar] <= dXmax
    # alpha_g * x <= b_g
    g_bar = 0.0
    alpha_g = np.ones(N)
    alpha_g[x] = -1.0
    b_g = dXmax - g_bar + sum(alpha_g[x])
    alpha_gcon = dict(zip(variables,alpha_g))
    prob += pp.lpSum([alpha_gcon[v] * xvars[v] for v in variables]) <= b_g
    # Solve ILP
    prob.solve(solver=pp.COIN_CMD(msg=verbose, gapRel=rErr, timeLimit=tLim))
    y = np.array([v.varValue for v in prob.variables()],dtype=bool)
    if verbose:
        print('linearized objective variation = {:.1e}'.format(sum(alpha[y])-sum(alpha[x])))
    return y
```

The function "solve_BESO" is defined. It solves an integer linear problem of binary variables with a single constraint, given by the maximal topology variation ("dXmax").

ilp_solver

```
def solve_BESO(N,x,alpha,dXmax,sense='min'):
    # Integer Linear Programming problem without extra constraints
    y = x.copy()
    if sense == 'min':  # minimization problem
        mask = (x & (alpha>0.0)) | (~x & (alpha<0.0))
    else:               # maximization problem
        mask = (x & (alpha<0.0)) | (~x & (alpha>0.0))
    Nv = sum(mask)
```

```python
    if Nv == 0:
        return y
    xsub = x[mask]
    arg = np.argsort(abs(alpha[mask]))
    if Nv < dXmax:
        xsub = ~xsub
    else:
        xsub[arg[-dXmax:]] = ~xsub[arg[-dXmax:]]
    y[mask] = xsub
    return y
```

3.3.9 ./source/python/generate_metamat.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 number of optimizations stored in the same file ("noptf") is set to the same value used when generating the data, which is 7.

generate_metamat

```python
import os, sys, shutil
noptf = 7  # number of optimizations stored in the same file
```

The data generated by the program "basecell_silp" is verified. The script checks if: the output folder exists; there is at leas 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 27 arrays of data have been written in each files subfolder.

generate_metamat

```python
if not os.path.exists('./SILP/output'):
    print('no output directory')
    sys.exit()
runs = sorted(os.listdir('./SILP/output'))
if len(runs) == 0:
    print('no runs have been found')
    sys.exit()
total = 0
r2 = -1
for k in range(len(runs)):
    run_dir = './SILP/output/' + runs[k] + '/'
    r1 = int(runs[k][4:9])
    if r1 <= r2:
        print('redundant runs : ' + runs[k])
        sys.exit()
    r2 = int(runs[k][10:])
    rnum = r2 - r1 + 1
    files = sorted(os.listdir(run_dir))
    if files[-1] == 'logs':
        files = files[:-1]
    else:
        print('missing logs directory : ' + runs[k])
        sys.exit()
    if rnum % noptf != 0:
        print('number of cases is not a multiple of noptf : ' + runs[k])
        sys.exit()
    if noptf*len(files) != rnum:
        print('incoherent number of files : ' + runs[k])
        sys.exit()
    total += rnum
    for kk in range(len(files)):
        file_dir = run_dir + files[kk]
        if len(os.listdir(file_dir)) != 27:
            print('wrong number of files : ' + runs[k] + '/' + files[kk])
            sys.exit()
if total % noptf != 0:
    print('something is wrong...')
    sys.exit()
total = total // noptf
print('valid SILP dataset!')
print('{:04d} / {:04d} files in the SILP dataset ({:5.1f} %)\n'.format(
      total,18382//noptf,total*noptf*100/18382))
```

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_metamat

```
# check directories
if not os.path.exists('../../dataset'):
    os.mkdir('../../dataset')
if not os.path.exists('../../dataset/SILP'):
    os.mkdir('../../dataset/SILP')
else:
    if len(os.listdir('../../dataset/SILP')) > 0:
        print('a SILP 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. All generated data is transferred, subfolders are renamed according to the "global_id" variable, so that a unique number is attributed to each files subfolder.

generate_metamat

```
global_id = 0
runs = sorted(os.listdir('./SILP/output'))
for k in range(len(runs)):
    run_dir = './SILP/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 SILP dataset ({:5.1f} %)'.format(
                    global_id,total,global_id*100/total))
        file_dir = run_dir + files[kk] + '/'
        destination = '../../dataset/SILP/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 SILP dataset ({:5.1f} %)'.format(
      global_id,total,global_id*100/total))
print('[ SILP dataset generated ]')
```

3.4 Implementation – Cython

3.4.1 ./source/cython/cython_setup.py

This python script compiles the cython script: "silp_sens.pyx".

cython_setup

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

3.4.2 ./source/cython/silp_sens.pyx

This script performs the CGS analysis for the diagonal terms of the elasticity matrix of the homogenized metamaterial ($\boldsymbol{C}$). Firstly, some flags are set and the cython module is imported.

silp_sens

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

The function "cython_cgs" is defined. It performs the CGS-0, CGS-1 and CGS-2 sensitivity analyses for the diagonal terms of $\boldsymbol{C}$. The function returns nothing, the sensitivity vectors "dC00_0", "dC11_0", "dC22_0", "dC00_1", "dC11_1", "dC22_1", "dC00_2", "dC11_2" and "dC22_2" are received as input then edited during execution (the changes are preserved in the outer scope). The approximations are computed in sequence: CGS-0 first ("dC00_0", "dC11_0" and "dC22_0"); then CGS-1 ("dC00_1", "dC11_1" and "dC22_1"); and lastly CGS-2 ("dC00_2", "dC11_2" and "dC22_2").

The matrix "P" is used to constrain the elemental matrices. For each augmented element, the global matrix is sliced and the resulting submatrix is stored in the variable "Kbb", which can have dimensions up to 192×192.

The CGS-0 sensitivity values are easily obtained from the elemental strain energy values. The CGS-1 and CGS-2 sensitivity values are obtained after computing the required coefficients: ω_{hm} ("whm"), ω_{mk} ("wmk"), ϕ_{m1} ("pm1"), ϕ_{m2} ("pm2") and $\phi_{\eta 2}$ ("peta2").

silp_sens

```
cdef void cython_cgs(double [:] dC00_0, double [:] dC11_0, double [:] dC22_0, double [:] dC00_1, double [:] dC11_1,
                     double [:] dC22_1, double [:] dC00_2, double [:] dC11_2, double [:] dC22_2, long long [:] x, long long N,
                     long long [:,::1] sym, long long [:] etype, long long [:] aug_etype, long long [:,::1] inci, double [:,::1] Ug,
                     double [:,:,::1] dKe, long long [:] Pindices, long long [:] Pindptr, long long [:] Kindices,
                     long long [:] Kindptr, double [:] Kdata, double [:,::1] aug_dKe0, double [:,::1] aug_dKe1,
                     double [:,::1] aug_dKe2, double [:,::1] aug_dKe3, double [:,::1] aug_dKe4, double [:,::1] aug_dKe5):
    cdef long long e
    cdef long long i
    cdef long long j
    cdef long long k
    cdef long long kk
    cdef long long kkk
    cdef long long et
    cdef long long ek
    cdef long long aug_ek
    cdef long long n0
    cdef long long n1
    cdef long long n2
    cdef long long n3
    cdef long long dof
    cdef long long ptr0
    cdef long long ptr1
    cdef long long pt0
    cdef long long pt1
    cdef long long ptm
    cdef long long bvar
    cdef long long row
    cdef long long col
    cdef long long dsize
    cdef long long nPids
    cdef long long size_h
    cdef long long nKids
    cdef long long size_k
    cdef long long verylarge
    cdef long long aug_eledofs[48]
    cdef long long Peindptr[49]
    cdef long long Peindices[48]
    cdef long long Pe_h[48]
    cdef long long dofs_h[48]
    cdef long long uniquebool[864]
    cdef long long Kreindices[864]
    cdef long long dofs_k[192]
    cdef long long dofs_kk[192]
    cdef long long elebool[6][8]
    cdef long long eledofs[6][8]
    cdef double dCdx[3]
    cdef double divisor[3]
    cdef double pm1[3]
    cdef double pm2[3]
    cdef double peta2[3]
    cdef double whm[3]
    cdef double wmk[3]
    cdef double wketa[3]
    cdef double wetaxi[3]
    cdef double Minv[192]
    cdef double Kbb[192][192]
    cdef double auxmat[48][48]
    cdef double Ue[48][3]
    cdef double zh[48][3]
    cdef double zm[48][3]
    cdef double zk[192][3]
    cdef double zeta[192][3]
    cdef double zxi[192][3]
    verylarge = 9223372036854775807
    for e in range(N):
        for k in range(3):
            dCdx[k] = 0.0
        for k in range(6):
            et = sym[e][k]
            ek = etype[et]
            n0 = inci[et][0]
            n1 = inci[et][1]
            n2 = inci[et][2]
            n3 = inci[et][3]
            eledofs[k][0] = 2*n0
            eledofs[k][1] = 2*n0+1
            eledofs[k][2] = 2*n1
            eledofs[k][3] = 2*n1+1
            eledofs[k][4] = 2*n2
            eledofs[k][5] = 2*n2+1
            eledofs[k][6] = 2*n3
            eledofs[k][7] = 2*n3+1
            for kk in range(8):
                Ue[kk][0] = Ug[eledofs[k][kk]][0]
                Ue[kk][1] = Ug[eledofs[k][kk]][1]
                Ue[kk][2] = Ug[eledofs[k][kk]][2]
            for i in range(8):
                for j in range(8):
                    dCdx[0] = dCdx[0] + Ue[i][0]*dKe[ek][i][j]*Ue[j][0]
                    dCdx[1] = dCdx[1] + Ue[i][1]*dKe[ek][i][j]*Ue[j][1]
                    dCdx[2] = dCdx[2] + Ue[i][2]*dKe[ek][i][j]*Ue[j][2]
        if x[e] == 1:
            dC00_0[e] = -dCdx[0]
            dC11_0[e] = -dCdx[1]
            dC22_0[e] = -dCdx[2]
        else:
            dC00_0[e] = dCdx[0]
            dC11_0[e] = dCdx[1]
            dC22_0[e] = dCdx[2]
        aug_ek = aug_etype[e]
        if aug_ek < 3:
            dsize = 48
        elif aug_ek < 5:
            dsize = 36
        else:
            dsize = 26
        for k in range(6):
            for kk in range(8):
                elebool[k][kk] = 1
        kkk = 0
        for k in range(6):
            for kk in range(8):
                if elebool[k][kk] == 1:
                    for i in range(6):
                        for j in range(8):
                            if eledofs[i][j] == eledofs[k][kk]:
                                elebool[i][j] = 0
                    aug_eledofs[kkk] = eledofs[k][kk]
                    kkk = kkk + 1
        nPids = 0
        Peindptr[0] = 0
        for k in range(dsize):
            Ue[k][0] = Ug[aug_eledofs[k]][0]
            Ue[k][1] = Ug[aug_eledofs[k]][1]
            Ue[k][2] = Ug[aug_eledofs[k]][2]
            ptr0 = Pindptr[aug_eledofs[k]]
            ptr1 = Pindptr[aug_eledofs[k]+1]
            if ptr1 > ptr0:
                Peindptr[k+1] = Peindptr[k] + 1
                Peindices[nPids] = Pindices[ptr0]
                nPids = nPids + 1
            else:
                Peindptr[k+1] = Peindptr[k]
        for k in range(nPids):
            uniquebool[k] = 1
        size_h = 0
        for k in range(nPids):
            if uniquebool[k] == 1:
                for kk in range(nPids):
                    if Peindices[kk] == Peindices[k]:
                        uniquebool[kk] = 0
                dofs_h[size_h] = Peindices[k]
                size_h = size_h + 1
        nKids = 0
        for k in range(size_h):
            ptr0 = Kindptr[dofs_h[k]]
            ptr1 = Kindptr[dofs_h[k]+1]
            for kk in range(ptr1-ptr0):
                Kreindices[nKids+kk] = Kindices[ptr0+kk]
            nKids = nKids + (ptr1-ptr0)
        for k in range(nKids):
           uniquebool[k] = 1
        size_k = 0
        for k in range(nKids):
            if uniquebool[k] == 1:
                for kk in range(k,nKids):
                    if Kreindices[kk] == Kreindices[k]:
                        uniquebool[kk] = 0
                dofs_k[size_k] = Kreindices[k]
                size_k = size_k + 1
        for k in range(size_h):
            dofs_kk[k] = dofs_h[k]
        kkk = size_h
        for k in range(size_k):
            bvar = 1
            for kk in range(size_h):
                if dofs_k[k] == dofs_h[kk]:
                    bvar = 0
                    break
            if bvar == 1:
                dofs_kk[kkk] = dofs_k[k]
                kkk = kkk + 1
        for k in range(dsize):
            Pe_h[k] = verylarge
        row = 0
        for k in range(size_h):
            for kk in range(nPids):
                if Peindices[kk] == dofs_h[k]:
                    for kkk in range(dsize):
                        if kk < Peindptr[kkk+1]:
                            row = kkk
                            break
                    Pe_h[row] = k
        for i in range(size_h):
            for k in range(dsize):
                auxmat[i][k] = 0.0
        if aug_ek == 0:
            for j in range(dsize):
                i = Pe_h[j]
                if i != verylarge:
                    for k in range(dsize):
                        auxmat[i][k] = auxmat[i][k] + aug_dKe0[j][k]
        elif aug_ek == 1:
            for j in range(dsize):
                i = Pe_h[j]
                if i != verylarge:
                    for k in range(dsize):
                        auxmat[i][k] = auxmat[i][k] + aug_dKe1[j][k]
        elif aug_ek == 2:
            for j in range(dsize):
                i = Pe_h[j]
                if i != verylarge:
                    for k in range(dsize):
                        auxmat[i][k] = auxmat[i][k] + aug_dKe2[j][k]
        elif aug_ek==3:
            for j in range(dsize):
                i = Pe_h[j]
                if i != verylarge:
                    for k in range(dsize):
                        auxmat[i][k] = auxmat[i][k] + aug_dKe3[j][k]
        elif aug_ek==4:
            for j in range(dsize):
                i = Pe_h[j]
                if i != verylarge:
                    for k in range(dsize):
                        auxmat[i][k] = auxmat[i][k] + aug_dKe4[j][k]
        else:
            for j in range(dsize):
                i = Pe_h[j]
                if i != verylarge:
                    for k in range(dsize):
                        auxmat[i][k] = auxmat[i][k] + aug_dKe5[j][k]
        for i in range(size_h):
            zh[i][0] = 0.0
            zh[i][1] = 0.0
            zh[i][2] = 0.0
            for j in range(dsize):
                zh[i][0] = zh[i][0] + auxmat[i][j] * Ue[j][0]
                zh[i][1] = zh[i][1] + auxmat[i][j] * Ue[j][1]
                zh[i][2] = zh[i][2] + auxmat[i][j] * Ue[j][2]
        for col in range(size_k):
            ptr0 = Kindptr[dofs_kk[col]]
            ptr1 = Kindptr[dofs_kk[col]+1]
            for row in range(col,size_k):
                dof = dofs_kk[row]
                if (dof < Kindices[ptr0]) or (dof > Kindices[ptr1-1]):
                    Kbb[row][col] = 0.0
                else:
                    pt0 = ptr0
                    pt1 = ptr1
                    bvar = 1
                    while (pt1-pt0) > 1:
                        ptm = (pt0+pt1)//2
                        if dof > Kindices[ptm]:
                            pt0 = ptm
                        elif dof < Kindices[ptm]:
                            pt1 = ptm
                        else:
                            Kbb[row][col] = Kdata[ptm]
                            bvar = 0
                            break
                    if bvar == 1:
                        if dof == Kindices[pt0]:
                            Kbb[row][col] = Kdata[pt0]
                        else:
                            Kbb[row][col] = 0.0
        if x[e] == 1:
            for j in range(dsize):
                k = Pe_h[j]
                for i in range(size_h):
                    if k <= i:
                        Kbb[i][k] = Kbb[i][k] - auxmat[i][j]
        else:
            for j in range(dsize):
                k = Pe_h[j]
                for i in range(size_h):
                    if k <= i:
                        Kbb[i][k] = Kbb[i][k] + auxmat[i][j]
        for k in range(size_k):
            Minv[k] = 1.0/Kbb[k][k]
        for k in range(size_h):
            zm[k][0] = Minv[k] * zh[k][0]
            zm[k][1] = Minv[k] * zh[k][1]
            zm[k][2] = Minv[k] * zh[k][2]
            zk[k][0] = Kbb[k][k] * zm[k][0]
            zk[k][1] = Kbb[k][k] * zm[k][1]
            zk[k][2] = Kbb[k][k] * zm[k][2]
        for i in range(size_h,size_k):
            zk[i][0] = 0.0
            zk[i][1] = 0.0
            zk[i][2] = 0.0
        for i in range(size_h):
            for j in range(i):
                zk[i][0] = zk[i][0] + Kbb[i][j] * zm[j][0]
                zk[i][1] = zk[i][1] + Kbb[i][j] * zm[j][1]
                zk[i][2] = zk[i][2] + Kbb[i][j] * zm[j][2]
                zk[j][0] = zk[j][0] + Kbb[i][j] * zm[i][0]
                zk[j][1] = zk[j][1] + Kbb[i][j] * zm[i][1]
                zk[j][2] = zk[j][2] + Kbb[i][j] * zm[i][2]
        for i in range(size_h,size_k):
            for j in range(size_h):
                zk[i][0] = zk[i][0] + Kbb[i][j] * zm[j][0]
                zk[i][1] = zk[i][1] + Kbb[i][j] * zm[j][1]
                zk[i][2] = zk[i][2] + Kbb[i][j] * zm[j][2]
        for i in range(size_k):
            zeta[i][0] = Minv[i] * zk[i][0]
            zeta[i][1] = Minv[i] * zk[i][1]
            zeta[i][2] = Minv[i] * zk[i][2]
            zxi[i][0] = Kbb[i][i] * zeta[i][0]
            zxi[i][1] = Kbb[i][i] * zeta[i][1]
            zxi[i][2] = Kbb[i][i] * zeta[i][2]
        for i in range(size_k):
            for j in range(i):
                zxi[i][0] = zxi[i][0] + Kbb[i][j] * zeta[j][0]
                zxi[i][1] = zxi[i][1] + Kbb[i][j] * zeta[j][1]
                zxi[i][2] = zxi[i][2] + Kbb[i][j] * zeta[j][2]
                zxi[j][0] = zxi[j][0] + Kbb[i][j] * zeta[i][0]
                zxi[j][1] = zxi[j][1] + Kbb[i][j] * zeta[i][1]
                zxi[j][2] = zxi[j][2] + Kbb[i][j] * zeta[i][2]
        whm[0] = 0.0
        whm[1] = 0.0
        whm[2] = 0.0
        wmk[0] = 0.0
        wmk[1] = 0.0
        wmk[2] = 0.0
        for k in range(size_h):
            whm[0] = whm[0] + zh[k][0]*zm[k][0]
            whm[1] = whm[1] + zh[k][1]*zm[k][1]
            whm[2] = whm[2] + zh[k][2]*zm[k][2]
            wmk[0] = wmk[0] + zm[k][0]*zk[k][0]
            wmk[1] = wmk[1] + zm[k][1]*zk[k][1]
            wmk[2] = wmk[2] + zm[k][2]*zk[k][2]
        pm1[0] = whm[0]/wmk[0]
        pm1[1] = whm[1]/wmk[1]
        pm1[2] = whm[2]/wmk[2]
        if x[e] == 1:
            dC00_1[e] = -dCdx[0] - pm1[0]*whm[0]
            dC11_1[e] = -dCdx[1] - pm1[1]*whm[1]
            dC22_1[e] = -dCdx[2] - pm1[2]*whm[2]
        else:
            dC00_1[e] = dCdx[0] - pm1[0]*whm[0]
            dC11_1[e] = dCdx[1] - pm1[1]*whm[1]
            dC22_1[e] = dCdx[2] - pm1[2]*whm[2]
        wketa[0] = 0.0
        wketa[1] = 0.0
        wketa[2] = 0.0
        wetaxi[0] = 0.0
        wetaxi[1] = 0.0
        wetaxi[2] = 0.0
        for k in range(size_k):
            wketa[0] = wketa[0] + zk[k][0]*zeta[k][0]
            wketa[1] = wketa[1] + zk[k][1]*zeta[k][1]
            wketa[2] = wketa[2] + zk[k][2]*zeta[k][2]
            wetaxi[0] = wetaxi[0] + zeta[k][0]*zxi[k][0]
            wetaxi[1] = wetaxi[1] + zeta[k][1]*zxi[k][1]
            wetaxi[2] = wetaxi[2] + zeta[k][2]*zxi[k][2]
        divisor[0] = wmk[0]*wetaxi[0]-wketa[0]*wketa[0]
        divisor[1] = wmk[1]*wetaxi[1]-wketa[1]*wketa[1]
        divisor[2] = wmk[2]*wetaxi[2]-wketa[2]*wketa[2]
        pm2[0] = (whm[0]*wetaxi[0]-wmk[0]*wketa[0])/divisor[0]
        pm2[1] = (whm[1]*wetaxi[1]-wmk[1]*wketa[1])/divisor[1]
        pm2[2] = (whm[2]*wetaxi[2]-wmk[2]*wketa[2])/divisor[2]
        peta2[0] = (wmk[0]*wmk[0]-whm[0]*wketa[0])/divisor[0]
        peta2[1] = (wmk[1]*wmk[1]-whm[1]*wketa[1])/divisor[1]
        peta2[2] = (wmk[2]*wmk[2]-whm[2]*wketa[2])/divisor[2]
        if x[e] == 1:
            dC00_2[e] = -dCdx[0] - (pm2[0]*whm[0]+peta2[0]*wmk[0])
            dC11_2[e] = -dCdx[1] - (pm2[1]*whm[1]+peta2[1]*wmk[1])
            dC22_2[e] = -dCdx[2] - (pm2[2]*whm[2]+peta2[2]*wmk[2])
        else:
            dC00_2[e] =  dCdx[0] - (pm2[0]*whm[0]+peta2[0]*wmk[0])
            dC11_2[e] =  dCdx[1] - (pm2[1]*whm[1]+peta2[1]*wmk[1])
            dC22_2[e] =  dCdx[2] - (pm2[2]*whm[2]+peta2[2]*wmk[2])
    return
```

The "cgs" function is defined to call the cython function in the python program. It receives as input: the vectors to which the sensitivity values will be assigned, "dC00_0", "dC11_0", "dC22_0", "dC00_1", "dC11_1", "dC22_1", "dC00_2", "dC11_2" and "dC22_2"; the topology vector, "x"; the number of design variables, "N"; the symmetry matrix, "sym"; the vectors with element types, "etype" and "aug_etype"; the incidence matrix, "inci"; the unconstrained total displacements matrix, "Ug"; the elemental stiffness variation matrices, "dKe" and "dKelist"; the matrix used to apply the periodic boundary conditions, "P"; and the CSC constrained global stiffness matrix, "Kr".

silp_sens

```
def cgs(dC00_0,dC11_0,dC22_0,dC00_1,dC11_1,dC22_1,dC00_2,dC11_2,dC22_2,x,N,sym,etype,aug_etype,inci,Ug,dKe,P,Kr,dKelist):
    cython_cgs(dC00_0, dC11_0, dC22_0, dC00_1, dC11_1, dC22_1, dC00_2, dC11_2, dC22_2, x.astype("int64"),
               N, sym.astype("int64"), etype.astype("int64"), aug_etype.astype("int64"),inci.astype("int64"), Ug, dKe,
               P.indices.astype("int64"), P.indptr.astype("int64"), Kr.indices.astype("int64"), Kr.indptr.astype("int64"), Kr.data,
               dKelist[0], dKelist[1], dKelist[2], dKelist[3], dKelist[4], dKelist[5])
    return
```

3.5 Implementation – Sampling

3.5.1 ./sample/SILP/sample.py

This script generates figures for a selected sample of the 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/SILP" directory. Plots of the Poisson's ratio, Young's modulus and volume of material throughout a given optimization can also be generated and saved as png images.

Firstly, the necessary modules are imported.

sample

```
import os, sys
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.collections as clct
from adjust import adjust
sys.path.append('../../source/python/SILP/')
from mesh import get_mesh
```

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" is *True*, figures are generated for all topologies (not only the optimized); if "fig_sen" is *True*, figures are generated for all sensitivity vectors of the diagonal terms of $\boldsymbol{C}$; if "fig_dis" is *True*, figures are generated for all displacements vectors; and if "fig_nu_Ey_vol" is *True*, figures are generated with all the plots of Poisson's ratio, Young's modulus 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 several topology, sensitivity and displacements vectors (in average, 75 topology vectors, 900 sensitivity vectors and 225 displacements vectors).

sample

```
file_ini      = 0      # initial file index |from file 0
file_lim      = 2      # file index limit   |up to file 2625
fig_top_opt   = True   # optimized topology
fig_top       = True   # topology vectors
fig_sen       = True   # sensitivity vectors
fig_dis       = True   # displacements vectors
fig_nu_Ey_vol = True   # Poisson's ratio, Young's modulus and volume
```

Some geometric properties are defined and the mesh is generated. The program checks if the "dataset" folder exists, then the folders in which the figures will be saved are created.

sample
```
# fixed properties
Ns = 32    # number of elements in each side of the design domain
N = Ns**2  # number of elements in the design domain
Nt = 6*N   # number of elements in the base cell
# area = 6*Lx*Ly = 1.0
Lx = 1.0/(108**0.25)  # design domain shorter side
Ly = np.sqrt(3)*Lx    # design domain longer side
Lex = Lx/Ns           # element shorter side
Ley = np.sqrt(3)*Lex  # element longer side
# Generate Mesh
coor, inci, etype, sym = get_mesh(Ns, Lex, Ley)
# check directories
rpath = '../../dataset/SILP/'
if not os.path.exists(rpath):
    print('missing SILP dataset')
    sys.exit()
if not os.path.exists('./top_opt'):
    os.mkdir('./top_opt')
if not os.path.exists('./top'):
    os.mkdir('./top')
if not os.path.exists('./sen'):
    os.mkdir('./sen')
if not os.path.exists('./dis'):
    os.mkdir('./dis')
if not os.path.exists('./nu_Ey_vol'):
    os.mkdir('./nu_Ey_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.

sample
```
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, Poisson's ratio values, Young's modulus values and volume values are read from the "top_opt.npy", "top.npy", "dC00_0.npy", "dC00_1.npy", "dC00_2.npy", "dC00_w.npy", "dC11_0.npy", "dC11_1.npy", "dC11_2.npy", "dC11_w.npy", "dC22_0.npy", "dC22_1.npy", "dC22_2.npy", "dC22_w.npy", "dis_xx.npy", "dis_yy.npy", "dis_xy.npy", "nu.npy", "Ey.npy" and "vol.npy" files.

sample
```
    # 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 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_sen:
        if not os.path.exists(rpath + 'f{:04d}/dC00_0.npy'.format(file)):
            print('missing : f{:04d}/dC00_0.npy'.format(file))
            sys.exit()
        list_dC00_0 = np.load(rpath + 'f{:04d}/dC00_0.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC00_1.npy'.format(file)):
            print('missing : f{:04d}/dC00_1.npy'.format(file))
            sys.exit()
        list_dC00_1 = np.load(rpath + 'f{:04d}/dC00_1.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC00_2.npy'.format(file)):
            print('missing : f{:04d}/dC00_2.npy'.format(file))
            sys.exit()
        list_dC00_2 = np.load(rpath + 'f{:04d}/dC00_2.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC00_w.npy'.format(file)):
            print('missing : f{:04d}/dC00_w.npy'.format(file))
            sys.exit()
        list_dC00_w = np.load(rpath + 'f{:04d}/dC00_w.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC11_0.npy'.format(file)):
            print('missing : f{:04d}/dC11_0.npy'.format(file))
            sys.exit()
        list_dC11_0 = np.load(rpath + 'f{:04d}/dC11_0.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC11_1.npy'.format(file)):
            print('missing : f{:04d}/dC11_1.npy'.format(file))
            sys.exit()
        list_dC11_1 = np.load(rpath + 'f{:04d}/dC11_1.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC11_2.npy'.format(file)):
            print('missing : f{:04d}/dC11_2.npy'.format(file))
            sys.exit()
        list_dC11_2 = np.load(rpath + 'f{:04d}/dC11_2.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC11_w.npy'.format(file)):
            print('missing : f{:04d}/dC11_w.npy'.format(file))
            sys.exit()
        list_dC11_w = np.load(rpath + 'f{:04d}/dC11_w.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC22_0.npy'.format(file)):
            print('missing : f{:04d}/dC22_0.npy'.format(file))
            sys.exit()
        list_dC22_0 = np.load(rpath + 'f{:04d}/dC22_0.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC22_1.npy'.format(file)):
            print('missing : f{:04d}/dC22_1.npy'.format(file))
            sys.exit()
        list_dC22_1 = np.load(rpath + 'f{:04d}/dC22_1.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC22_2.npy'.format(file)):
            print('missing : f{:04d}/dC22_2.npy'.format(file))
            sys.exit()
        list_dC22_2 = np.load(rpath + 'f{:04d}/dC22_2.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dC22_w.npy'.format(file)):
            print('missing : f{:04d}/dC22_w.npy'.format(file))
            sys.exit()
        list_dC22_w = np.load(rpath + 'f{:04d}/dC22_w.npy'.format(file))
    # displacements vectors
    if fig_dis:
        if not os.path.exists(rpath + 'f{:04d}/dis_xx.npy'.format(file)):
            print('missing : f{:04d}/dis_xx.npy'.format(file))
            sys.exit()
        list_dis_xx = np.load(rpath + 'f{:04d}/dis_xx.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dis_yy.npy'.format(file)):
            print('missing : f{:04d}/dis_yy.npy'.format(file))
            sys.exit()
        list_dis_yy = np.load(rpath + 'f{:04d}/dis_yy.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/dis_xy.npy'.format(file)):
            print('missing : f{:04d}/dis_xy.npy'.format(file))
            sys.exit()
        list_dis_xy = np.load(rpath + 'f{:04d}/dis_xy.npy'.format(file))
    # Poisson's ratio, Young's modulus and volume
    if fig_nu_Ey_vol:
        if not os.path.exists(rpath + 'f{:04d}/nu.npy'.format(file)):
            print('missing : f{:04d}/nu.npy'.format(file))
            sys.exit()
        list_nu = np.load(rpath + 'f{:04d}/nu.npy'.format(file))
        if not os.path.exists(rpath + 'f{:04d}/Ey.npy'.format(file)):
            print('missing : f{:04d}/Ey.npy'.format(file))
            sys.exit()
        list_Ey = np.load(rpath + 'f{:04d}/Ey.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 grayscale images, solid elements are represented in black and void elements are represented in gray.

sample

```
    #%% Generate figures
    print(': generating figures')
    # optimized topology
    if fig_top_opt:
        print(': : optimized topology...')
        for k in range(len(list_fid)):
            plt.figure(num=0).clear()
            fig,ax = plt.subplots(num=0)
            fid = list_fid[k]
            x = list_top_opt[k]
            x = np.unpackbits(x,axis=None).astype(float)
            xt = np.ndarray((Nt),dtype=bool)
            for k in range(N):
                xt[sym[k,:]] = x[k]
            polys = clct.PolyCollection(coor[inci],cmap='gray_r',edgecolor=(0,0,0,0))
            polys.set_array(xt+1.0)
            polys.set_clim(0.0,2.0)
            ax.add_collection(polys)
            ax.set_aspect('equal')
            ax.set_xlim([-2*Lx,2*Lx])
            ax.set_ylim([-Ly,Ly])
            ax.axis('off')
            fig.set_size_inches(8,7)
            plt.savefig('./top_opt/f{:05d}.png'.format(fid),bbox_inches='tight',pad_inches=0.05,dpi=100)
```

If "fig_top" is *True*, images of the topology vectors are saved. The topologies are represented as grayscale images.

sample
```
    # topology vectors
    if fig_top:
        print(': : topology vectors...')
        for k in range(len(list_fid)):
            fid = list_fid[k]
            j = 0
            for kk in range(list_ptr2opt[k],list_ptr2opt[k+1]):
                plt.figure(num=0).clear()
                fig,ax = plt.subplots(num=0)
                x = list_top[kk]
                x = np.unpackbits(x,axis=None).astype(float)
                xt = np.ndarray((Nt),dtype=bool)
                for k in range(N):
                    xt[sym[k,:]] = x[k]
                polys = clct.PolyCollection(coor[inci],cmap='gray_r',edgecolor=(0,0,0,0))
                polys.set_array(xt+1.0)
                polys.set_clim(0.0,2.0)
                ax.add_collection(polys)
                ax.set_aspect('equal')
                ax.set_xlim([-2*Lx,2*Lx])
                ax.set_ylim([-Ly,Ly])
                ax.axis('off')
                fig.set_size_inches(8,7)
                plt.savefig('./top/f{:05d}_{:03d}.png'.format(fid,j),bbox_inches='tight',pad_inches=0.05,dpi=100)
                j += 1
```

If "fig_sen" is *True*, images of the sensitivity vectors are saved. Each sensitivity map is represented as a single-channeled image, the "cividis" colormap is used instead of grayscale. An independent nonlinear scale is defined for each image, in order to improve contrast resolution, this scale is computed by the function "adjust.adjust". The python script "adjust.py" is detailed in the next section.

The twelve sensitivity maps are saved in a single figure, in a 3×4 grid of subplots. The first row corresponds to sensitivity maps of C_{00}; the second row to the sensitivity maps of C_{11}; and the third row to the sensitivity maps of C_{22}. The first column corresponds to the CGS-0 approximations; the second column to the CGS-1 approximations; the third column to the CGS-2 approximations; and the fourth column to the exact sensitivity maps, obtained through WS approach.

sample
```
    # sensitivity vectors
    if fig_sen:
        print(': : sensitivity vectors...')
        for k in range(len(list_fid)):
            fid = list_fid[k]
            j = 0
            for kk in range(list_ptr2opt[k],list_ptr2opt[k+1]):
                plt.figure(num=0).clear()
                fig,ax = plt.subplots(nrows=3,ncols=4,num=0)
                xmin = -2*Lx
                xmax =  2*Lx
                ymin =   -Ly
                ymax =    Ly
                dC00_0 = list_dC00_0[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC00_0,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[0,0].add_collection(polys)
                ax[0,0].set_aspect('equal')
                ax[0,0].set_xlim([xmin,xmax])
                ax[0,0].set_ylim([ymin,ymax])
                ax[0,0].axis('off')
                dC00_1 = list_dC00_1[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC00_1,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[0,1].add_collection(polys)
                ax[0,1].set_aspect('equal')
                ax[0,1].set_xlim([xmin,xmax])
                ax[0,1].set_ylim([ymin,ymax])
                ax[0,1].axis('off')
                dC00_2 = list_dC00_2[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC00_2,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[0,2].add_collection(polys)
                ax[0,2].set_aspect('equal')
                ax[0,2].set_xlim([xmin,xmax])
                ax[0,2].set_ylim([ymin,ymax])
                ax[0,2].axis('off')
                dC00_w = list_dC00_w[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC00_w,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[0,3].add_collection(polys)
                ax[0,3].set_aspect('equal')
                ax[0,3].set_xlim([xmin,xmax])
                ax[0,3].set_ylim([ymin,ymax])
                ax[0,3].axis('off')
                dC11_0 = list_dC11_0[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC11_0,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[1,0].add_collection(polys)
                ax[1,0].set_aspect('equal')
                ax[1,0].set_xlim([xmin,xmax])
                ax[1,0].set_ylim([ymin,ymax])
                ax[1,0].axis('off')
                dC11_1 = list_dC11_1[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC11_1,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[1,1].add_collection(polys)
                ax[1,1].set_aspect('equal')
                ax[1,1].set_xlim([xmin,xmax])
                ax[1,1].set_ylim([ymin,ymax])
                ax[1,1].axis('off')
                dC11_2 = list_dC11_2[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC11_2,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[1,2].add_collection(polys)
                ax[1,2].set_aspect('equal')
                ax[1,2].set_xlim([xmin,xmax])
                ax[1,2].set_ylim([ymin,ymax])
                ax[1,2].axis('off')
                dC11_w = list_dC11_w[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC11_w,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[1,3].add_collection(polys)
                ax[1,3].set_aspect('equal')
                ax[1,3].set_xlim([xmin,xmax])
                ax[1,3].set_ylim([ymin,ymax])
                ax[1,3].axis('off')
                dC22_0 = list_dC22_0[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC22_0,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[2,0].add_collection(polys)
                ax[2,0].set_aspect('equal')
                ax[2,0].set_xlim([xmin,xmax])
                ax[2,0].set_ylim([ymin,ymax])
                ax[2,0].axis('off')
                dC22_1 = list_dC22_1[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC22_1,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[2,1].add_collection(polys)
                ax[2,1].set_aspect('equal')
                ax[2,1].set_xlim([xmin,xmax])
                ax[2,1].set_ylim([ymin,ymax])
                ax[2,1].axis('off')
                dC22_2 = list_dC22_2[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC22_2,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[2,2].add_collection(polys)
                ax[2,2].set_aspect('equal')
                ax[2,2].set_xlim([xmin,xmax])
                ax[2,2].set_ylim([ymin,ymax])
                ax[2,2].axis('off')
                dC22_w = list_dC22_w[kk]
                sens_plot = np.ndarray((Nt))
                adjusted = adjust(dC22_w,N)
                for k in range(N):
                    sens_plot[sym[k,:]] = adjusted[k]
                polys = clct.PolyCollection(coor[inci],cmap='cividis',edgecolor=(0,0,0,0))
                polys.set_array(sens_plot)
                ax[2,3].add_collection(polys)
                ax[2,3].set_aspect('equal')
                ax[2,3].set_xlim([xmin,xmax])
                ax[2,3].set_ylim([ymin,ymax])
                ax[2,3].axis('off')
                fig.set_size_inches(20,13)
                plt.savefig('./sen/f{:05d}_{: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 cell are saved, considering the three imposed macro-displacements. In order to visualize the displacements field, the topologies are represented in the deformed mesh. The scale is kept the same for all figures, so the influence of each topological change can be perceived.

The three displacements fields are saved in a single figure, in a 1×3 grid of subplots. The first column corresponds to the deformed cell when $\widehat{\boldsymbol{u}}_{xx}$ is imposed; the second column to the deformed cell when $\widehat{\boldsymbol{u}}_{yy}$ is imposed; and the third column to the deformed cell when $\widehat{\boldsymbol{u}}_{xy}$ is imposed.

sample
```
    # displacements vectors
    if fig_dis:
        print(': : displacements vectors...')
        for k in range(len(list_fid)):
            fid = list_fid[k]
            j = 0
            scale = 0.10
            xmin = -2*Lx*(1.05+scale)
            xmax =  2*Lx*(1.05+scale)
            ymin =   -Ly*(1.15+scale)
            ymax =    Ly*(1.15+scale)
            for kk in range(list_ptr2opt[k],list_ptr2opt[k+1]):
                plt.figure(num=0).clear()
                fig,ax = plt.subplots(nrows=1,ncols=3,num=0)
                x = list_top[kk]
                x = np.unpackbits(x,axis=None).astype(float)
                xt = np.ndarray((Nt),dtype=bool)
                for k in range(N):
                    xt[sym[k,:]] = x[k]
                dis_xx = list_dis_xx[kk]
                umat = np.reshape(dis_xx,coor.shape)
                coor_dis = coor + scale*umat
                polys = clct.PolyCollection(coor_dis[inci],cmap='gray_r',edgecolor=(0,0,0,0))
                polys.set_array(xt+1.0)
                polys.set_clim(0.0,2.0)
                ax[0].add_collection(polys)
                ax[0].set_aspect('equal')
                ax[0].set_xlim([xmin,xmax])
                ax[0].set_ylim([ymin,ymax])
                ax[0].axis('off')
                dis_yy = list_dis_yy[kk]
                umat = np.reshape(dis_yy,coor.shape)
                coor_dis = coor + scale*umat
                polys = clct.PolyCollection(coor_dis[inci],cmap='gray_r',edgecolor=(0,0,0,0))
                polys.set_array(xt+1.0)
                polys.set_clim(0.0,2.0)
                ax[1].add_collection(polys)
                ax[1].set_aspect('equal')
                ax[1].set_xlim([xmin,xmax])
                ax[1].set_ylim([ymin,ymax])
                ax[1].axis('off')
                dis_xy = list_dis_xy[kk]
                umat = np.reshape(dis_xy,coor.shape)
                coor_dis = coor + scale*umat
                polys = clct.PolyCollection(coor_dis[inci],cmap='gray_r',edgecolor=(0,0,0,0))
                polys.set_array(xt+1.0)
                polys.set_clim(0.0,2.0)
                ax[2].add_collection(polys)
                ax[2].set_aspect('equal')
                ax[2].set_xlim([xmin,xmax])
                ax[2].set_ylim([ymin,ymax])
                ax[2].axis('off')
                fig.set_size_inches(20,8)
                fig.savefig('./dis/f{:05d}_{:03d}.png'.format(fid,j),bbox_inches='tight',pad_inches=0,dpi=100)
                j += 1
```

If "fig_nu_Ey_vol" is *True*, plots of Poisson's ratio, Young's modulus and volume are saved. The evolution of these functions throughout each optimization procedure is plotted in the same figure, in a 3×1 grid of subplots.

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

sample

```
    # Poisson's ratio, Young's modulus and volume
    if fig_nu_Ey_vol:
        print(': : Poisson\'s ratio, Young\'s modulus and volume...')
        for k in range(len(list_fid)):
            plt.figure(num=0).clear()
            fig,ax = plt.subplots(nrows=3,ncols=1,num=0)
            fid = list_fid[k]
            inp = list_inp[k]
            Eymin = inp[1]
            nu  = list_nu[list_ptr2opt[k]:list_ptr2opt[k+1]]
            Ey  = list_Ey[list_ptr2opt[k]:list_ptr2opt[k+1]]
            vol = list_vol[list_ptr2opt[k]:list_ptr2opt[k+1]]
            size = len(nu)
            delta = max(nu)-min(nu)
            miny = min(nu)-0.02*delta
            maxy = max(nu)+0.02*delta
            ax[0].plot(nu,'ok-',linewidth=2)
            ax[0].axis([-0.75, size-0.25, miny, maxy])
            ax[0].set_ylabel('Poisson\'s ratio',fontsize=18)
            ax[0].grid()
            delta = max(Ey)-min(Ey)
            miny = Eymin-0.02*delta
            maxy = max(Ey)+0.02*delta
            ax[1].plot(Ey,'ok-',linewidth=2)
            ax[1].plot([-0.75, size-0.25],[Eymin,Eymin],'k--',linewidth=2)
            ax[1].axis([-0.75, size-0.25, miny, maxy])
            ax[1].set_ylabel('Young\'s modulus [Pa]',fontsize=18)
            ax[1].grid()
            delta = max(vol)-min(vol)
            miny = min(vol)-0.02*delta
            maxy = max(vol)+0.02*delta
            ax[2].plot(vol,'ok-',linewidth=2)
            ax[2].axis([-0.75, size-0.25, miny, maxy])
            ax[2].set_ylabel('volume fraction',fontsize=18)
            ax[2].grid()
            ax[2].set_xlabel('iteration',fontsize=18)
            fig.set_size_inches(8,13)
            fig.savefig('./nu_Ey_vol/f{:05d}.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.

sample

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

3.5.2 ./sample/SILP/adjust.py

This script is used to adjust the sensitivity maps in order to improve contrast resolution of the generated figures. Firstly, the numpy module is imported.

adjust

```
import numpy as np
```

The "adjust" function is defined. The variable "delta" stores the difference between the maximal and minimal sensitivity values. The vector is shifted so that its median value becomes 0.0. The positive values are shifted up, and the nonpositive values are shifted down (the shift value is given by a fraction of "delta", defined by the variable “coef”). The absolute values of the negative part are taken, then, the logarithms of both parts are computed. Lastly, the former negative part is reflected over the horizontal axis that crosses its minimal value (“mval”). After performing all these procedures, the curve described by the sorted adjusted vector is considered. The variable “coef” is calibrated in so that the slope in the middle of the curve (“slope”) be similar to its mean slope (“ref”).

adjust

```python
def adjust(sens,N):
    slope = 1
    ref   = 0
    coef  = 1e-12
    while slope > ref:
        adjusted = sens.copy()
        delta = max(adjusted)-min(adjusted)
        adjusted = adjusted - np.median(adjusted)
        mask = (adjusted > 0.0)
        adjusted[mask]  = adjusted[mask]  + coef*delta
        adjusted[~mask] = adjusted[~mask] - coef*delta
        adjusted[mask]  = np.log(adjusted[mask])
        adjusted[~mask] = np.log(-adjusted[~mask])
        mval = min(adjusted[~mask])
        adjusted[~mask] = 2*mval - adjusted[~mask]
        adj_sorted = np.sort(adjusted)
        slope = (adj_sorted[N//2+2]-adj_sorted[N//2-3])/5
        ref = (max(adjusted[mask])-min(adjusted[~mask]))/(N-1)
        coef = 2*coef
    return adjusted
```

3.6 Validation Procedure

The validation procedure consists mainly in independent, alternative implementations to perform each task of the optimization program that generates the dataset.

The validation codes are provided in the folder “./validation”. The bash scipt “metaval.sh” can be executed in order to build the required Cython codes. The script “metamaterial_val.py” can be executed to perform the following validations.

Firstly, four different topology vectors are created. The first one corresponds to a fully solid structure; the second one corresponds to a fully void structure; the third one corresponds to the initial topology used in the optimization procedures; and the fourth one corresponds to a random structure.

The homogenization procedure is performed for all the topologies. Then, the program checks if: the obtained properties are isotropic; the loads over opposing edges are anti-symmetric; the mean stress corresponds to the obtained elasticity matrix applied on the mean strain of the base cell; the mechanical properties are withing their minimal and maximal possible values. The resultant loads are shown in a figure, so that it can be seen that there is no external load in the interior of the base cell (all nonzero loads are placed over the edges), and that the loads are anti-symmetric. The range of the loads (minimal and maximal values) are presented above each figure.

The “topopt.update” function is tested: the first topology is updated to be equal the fourth one; then, the resulting displacements and stiffness matrices are compared with the ones previously computed for the fourth topology. The error due to the shift maneuver (used to preserve the nonzero pattern of the stiffness matrix) is evaluated.

The fourth topology is used to validate the sensitivity values. The CGS values computed in the “silp_sens.pyx” script are compared with values obtained using estimations for the displacements vectors after altering each element, using the Conjugate Gradient Method. The WS values computed in the “topopt.py” script are compared with values obtained through a naive approach, using exact values for the displacements vectors after altering each element. The signs of the sensitivity values of the diagonal terms of $\boldsymbol{C}$ are verified, as well as their monotonic behavior, as more steps are considered in the CGS approach. The error of the sensitivity values for the Poisson’s ratio, Young’s modulus and objective function are evaluated. These errors are plotted for CGS-0, CGS-1 and CGS-2, so it can be seen that, for most cases, CGS-2 is substantially more accurate than CGS-0.

Lastly, through visual verification, qualitative validations are performed for the conical filter used to smooth the sensitivity maps; for the morphological operators; and for the island removal procedure. To validate the smoothing filter, the fourth topology and a toy example are filtered, so it can be verified if the smoothing procedure is working as expected. To validate the morphological operator, the erosion, dilation, opening (erosion then dilation) and closing (dilation then erosion) operators are applied to the fourth topology. To validate the island removal procedure, it is performed for the third topology, the fourth topology and a toy example, the connected and disconnected parts are shown in separate figures.

After executing the script “metamaterial_val.py”, a number of samples was selected from the generated dataset and visually verified, using the presented “sample.py” script. Some of them are shown in section 4.

Furthermore, all input-output logs were checked, in order to verify if reasonable values were obtained for the mechanical properties of each optimized topology. It was verified if every pair of redundant data truly corresponds to the same value, that is, if each array “dC00_w.npy” is equal to the corresponding array “dC11_w.npy”. All observed results are coherent and indicate that the optimization program was properly implemented.

3.7 Unfixed Bug

The removal of disconnected solids from the optimized structure may yield an unsuitable solution. This bug is reported in the “Issues” tab of the github repository.

After concluding the current optimization process, all remaining disconnected solid elements are removed from the optimized topology. In the current version of the program, the final optimized topology is stored without verifying if the removal of disconnected solids resulted in a worse value for the objective function, of if it resulted in a structure that does not respect the constraint function. In cases that different parts of the structure are only connected by single nodes (not by edges), this approach may result in unsuitable structures.

To solve this issue (in a future version), all topologies can be stored, ordered from the best one obtained thus far to the worst one. Then, when an undesirable result is obtained, the next candidate can be taken and evaluated.

This issue has affected only a small number of optimized solutions. Less than 1% of the optimized solutions, stored in the ’top_opt.npy’ files, break the constraint over the Young’s modulus. In the current version, suitable optimized solutions can be recovered using the data stored in the ’top.npy’, ’nu.npy’ and ’Ey.npy’ files. As expected, the constraint is respected for 100% of the topologies from all iterations before the final removal of disconnected solids, stored in the ’top.npy’ files.

4 Samples

The dataset stores everything generated in the optimization procedures for each one of the 18 382 unique pairs $(\nu^*, E_{\min})$.

To illustrate the size of the dataset, the homogenized mechanical properties from all the ’nu.npy’ and ’Ey.npy’ files, corresponding to topologies stored in all the ’top.npy’ files, are shown in Figure 7. There are 1 374 656 topologies stored in these files, however, around half of them have very similar (or identical) homogenized mechanical properties to other topologies from the dataset. After removing the duplicates, 616 862 topologies with unique mechanical properties are obtained, these are the ones shown in the scatter plot.

It can be seen that topologies with Young’s moduli near 100% have Poisson’s ratios near 0.30, which is the property of the base material. The range of the Poisson’s ratios increases as lower Young’s moduli are considered. When the Young’s moduli are near 0%, the obtained Poisson’s ratios range from nearly -1.0 to nearly 1.0. There is a region with sparser points that corresponds to Poisson’s values between 0.20 and 0.40. It occurs because all optimizations start from a homogenized Poisson’s ratio of around 0.30 and it is desired to obtain values that are either higher than 0.40, or lower than 0.20. So, topologies move into this region only when undesirable steps are made in the optimization procedures. There are two regions with denser points around the minimal and the maximal Poisson’s values for each Young’s modulus. They occur because, for each considered minimal Young’s modulus ($E_{\min}$) the target Poisson’s values (ν^*) range from -1.0 to 1.0. So, when $E_{\min}$ is too high to allow extreme Poisson’s ratios, all optimizations that were not able to achieve their target values yield results near the minimal or maximal possible Poisson’s ratios for that Young’s modulus constraint.

The points corresponding to the optimized results, stored in all the ’top_opt.npy’, ’nu_opt.npy’ and ’Ey_opt.npy’ files, are presented in Figure 8. Once again, topologies with very similar (or identical) homogenized mechanical properties are disregarded: only around half of the 18 382 optimized topologies are shown in the scatter plot. The same observations can be made: there is an empty region corresponding to the Poisson’s values between 0.20 and 0.40 (unsearched region); and denser regions around the minimal and maximal Poisson’s values (just above the lower unreached region and just below the upper unreached region).

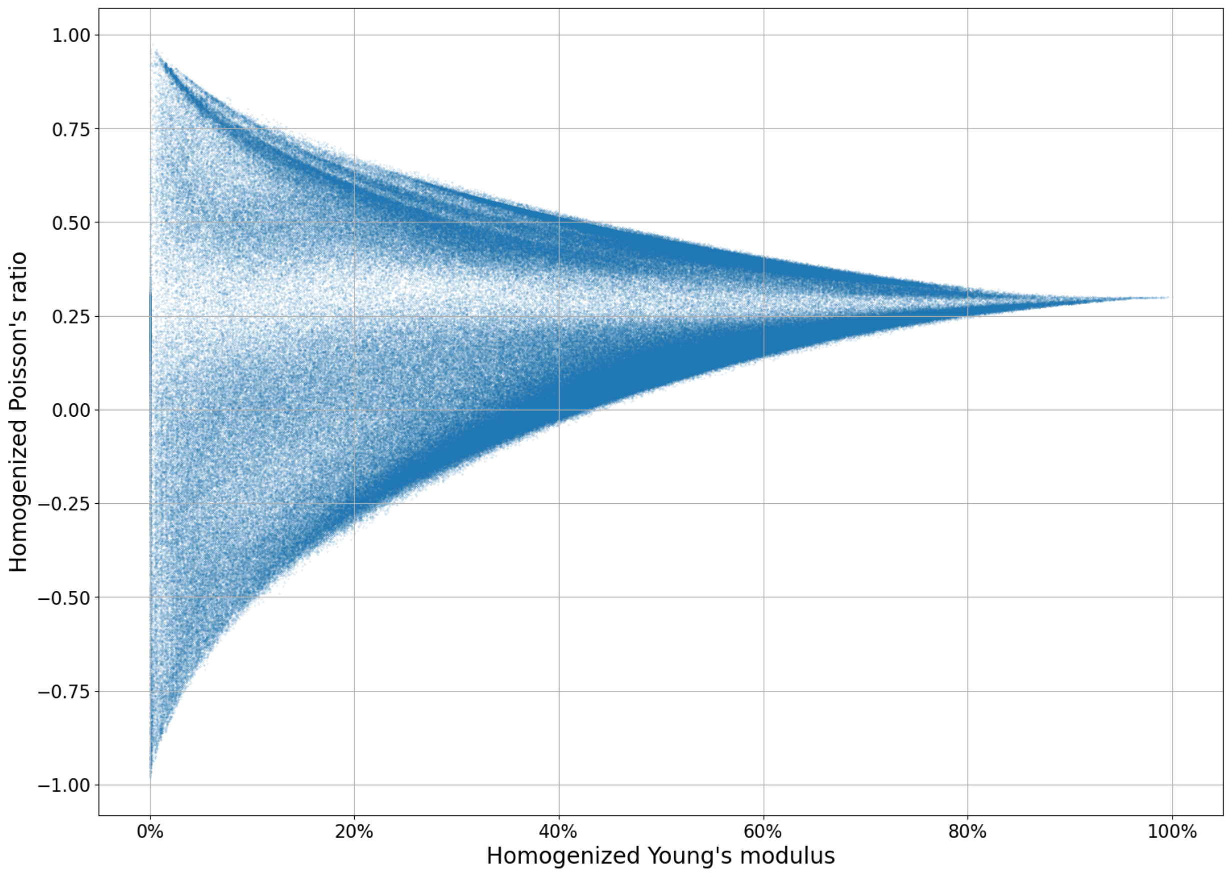

Figure 7: Properties of all generated metamaterials

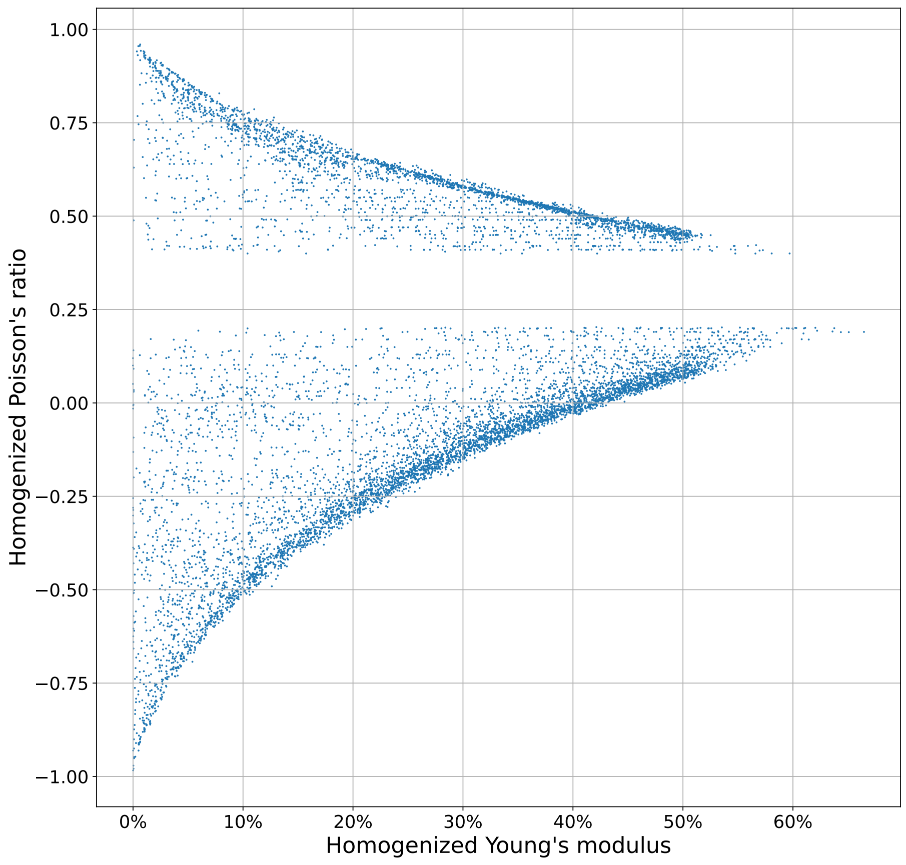

Figure 8: Properties of the optimized metamaterials

In Figure 9, a small sample of 19 optimized topologies is shown, the positions of the presented structures indicate their corresponding homogenized Poisson's ratios and Young's moduli. The solid part of the structures is represented in dark gray and the void part is represented in light gray.

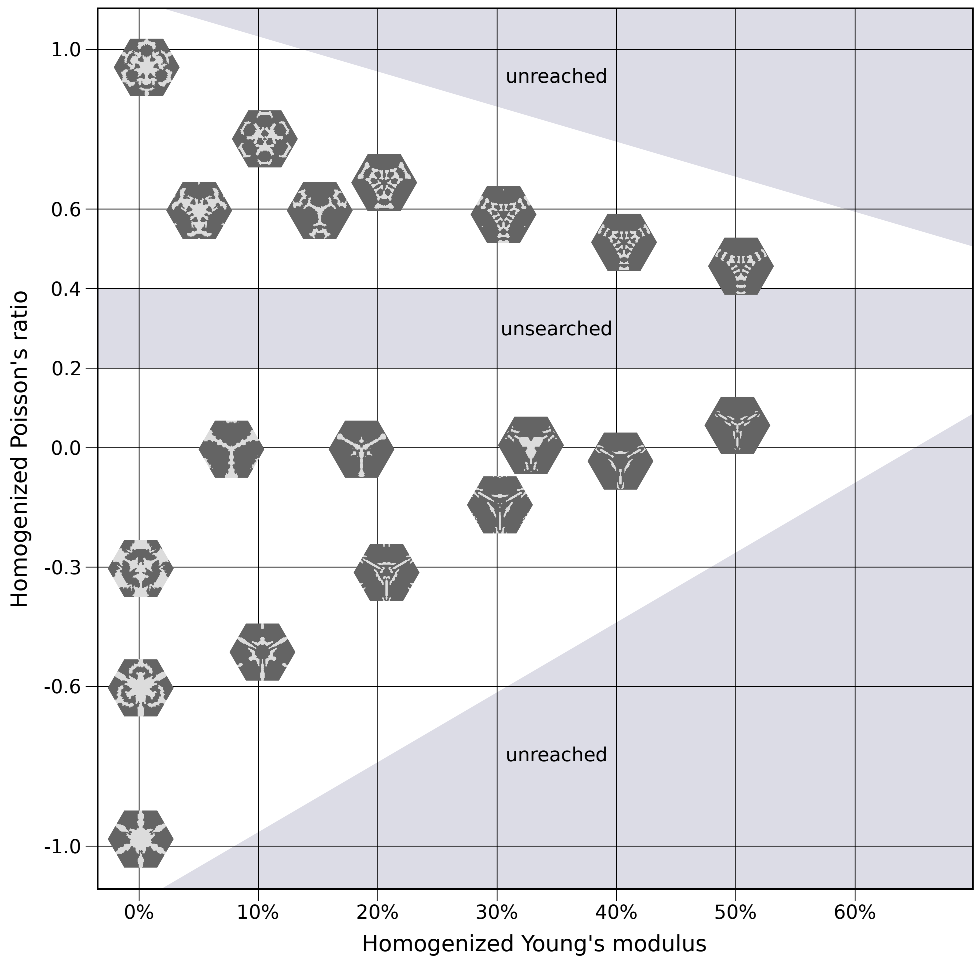

Figure 9: Optimized base cells and their homogenized properties

More samples are taken from the slices highlighted in Figure 10.

In the slice 'A', topologies with Young's moduli between 10% and 15% are taken, a sample of 28 structures with different Poisson's ratios is shown in Figure 11. Steps of 0.04 are considered. Topologies with Poisson's ratios ranging from -0.49 to 0.19 and from 0.41 to 0.77 are shown. The first row begins in lower unreached region and the second row ends in the upper unreached region. In the second row, the unsearched region corresponds to Poisson's ratios ranging from 0.20 to 0.40.

In the slice 'B', topologies with Poisson's ratios between -0.02 and 0.02 are taken, a sample of 10 structures with different Young's moduli is shown in Figure 12. Steps of at most 7.0% are considered. Topologies with Young's moduli ranging from 0.0% to 44.0% are shown. In Figure 10, it can be seen that for the considered values of Poisson's ratios, Young's moduli above 45% correspond to the lower unreached region.

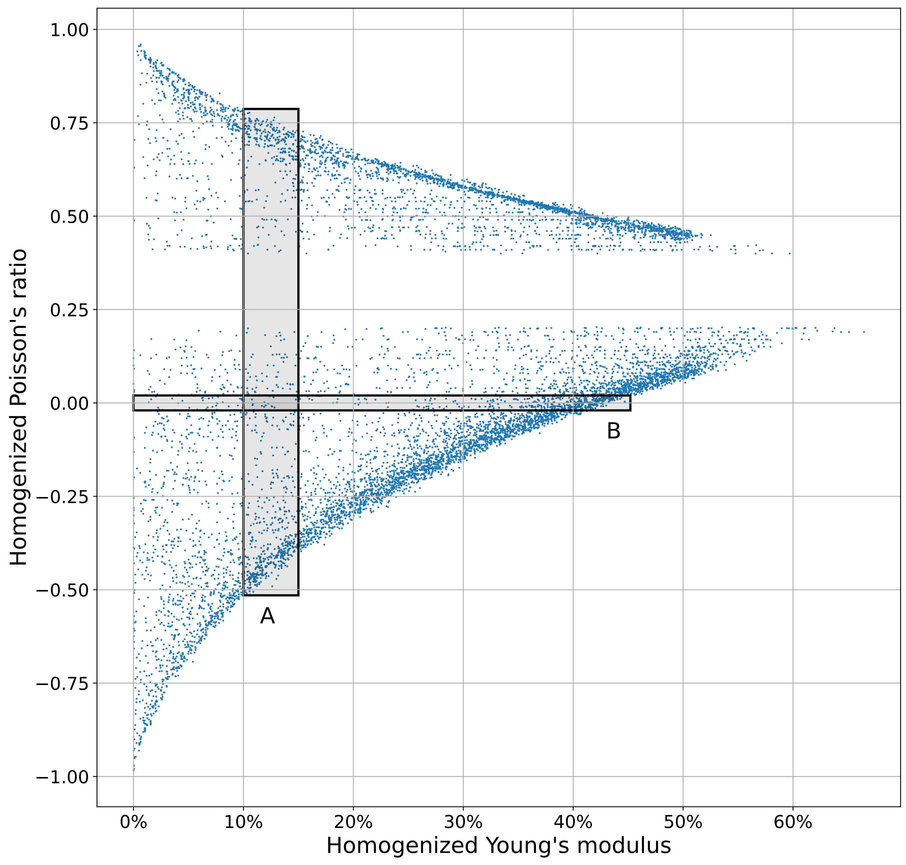

Figure 10: Slices of the scatter plot of optimized results

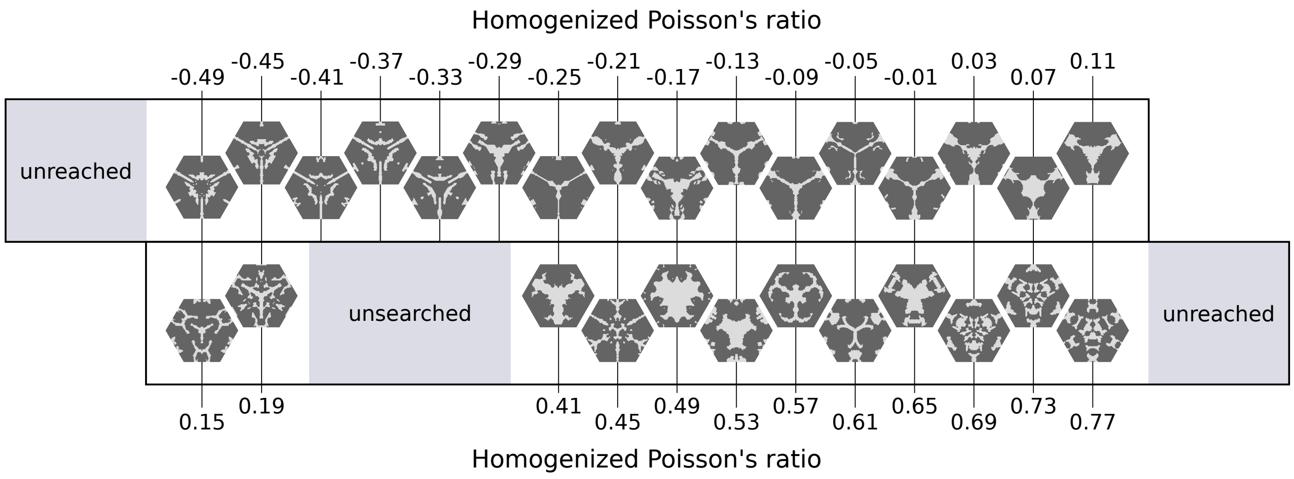

Figure 11: Optimized topologies with different Poisson's ratios (slice 'A')

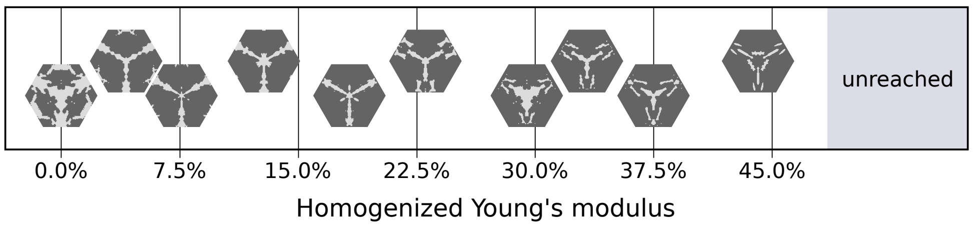

Figure 12: Optimized topologies with different Young's moduli (slice 'B')

To illustrate all the content stored in the dataset, the case with $\nu^* = -0.30$ and $E_{\min} = 20.0\%$ is considered. The obtained results are presented in Figures 13, 14, 15, 16 and 17, plotted by the "sample" script.

From Figure 13, it can be noted that the Poisson's ratio steadily improves when the Young's modulus is far from its minimal value. When the constraint is activated, the process become less stable because each time the solution of the linearized subproblem breaks the Young's modulus constraint, the structure is dilated. The dilation operations can be identified as abrupt increases in the volume fraction. The best obtained result corresponds to iteration 48, shown in Figure 14, its homogenized properties are $\widehat{\nu} = -0.29$ and $\widehat{E} = 20.4\%$. In Figure 15, all evaluated topologies are presented. Although a patience parameter of 30 was used, it took 81 iterations to conclude the procedure because the topology from iteration 51 has a smaller volume fraction than the 48-th topology, which resulted in a lower value for the objective function with volume penalization.

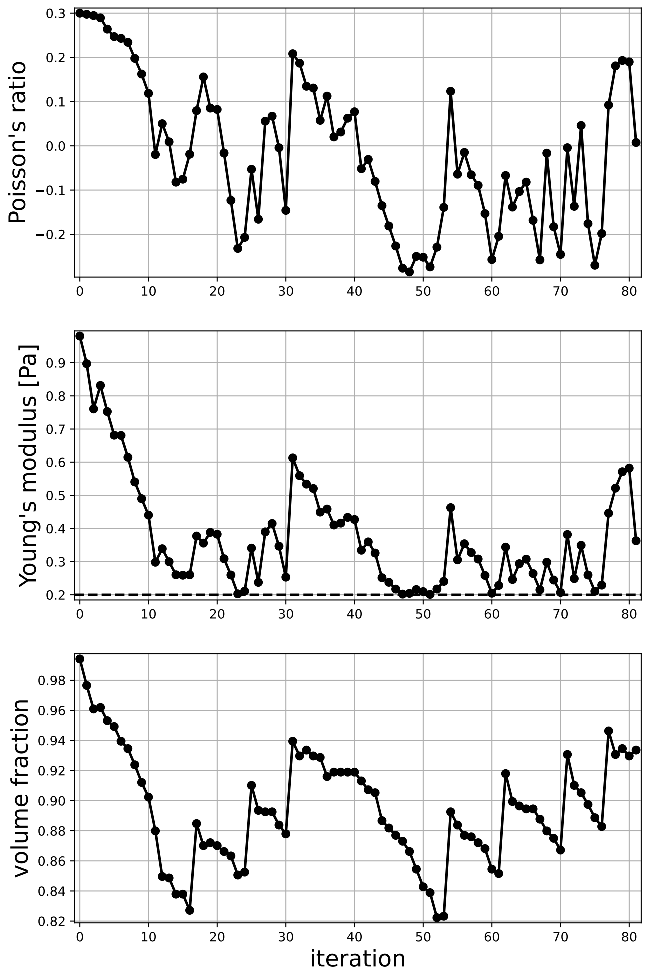

Figure 13: Poisson's ratio, Young's modulus and volume fraction for $\nu^* = -0.30$ and $E_{\text{min}} = 20.0\%$

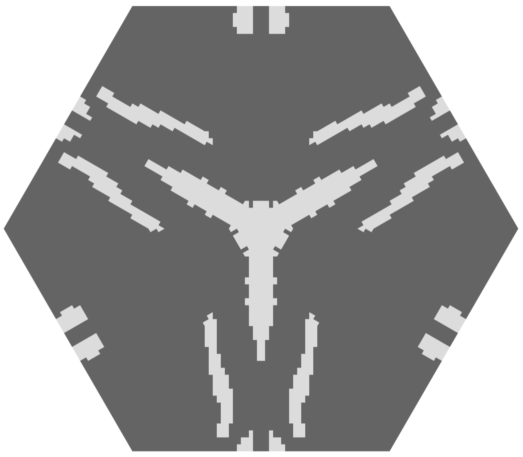

Figure 14: Optimized topology for $\nu^* = -0.30$ and $E_{\text{min}} = 20.0\%$ (iteration 48)

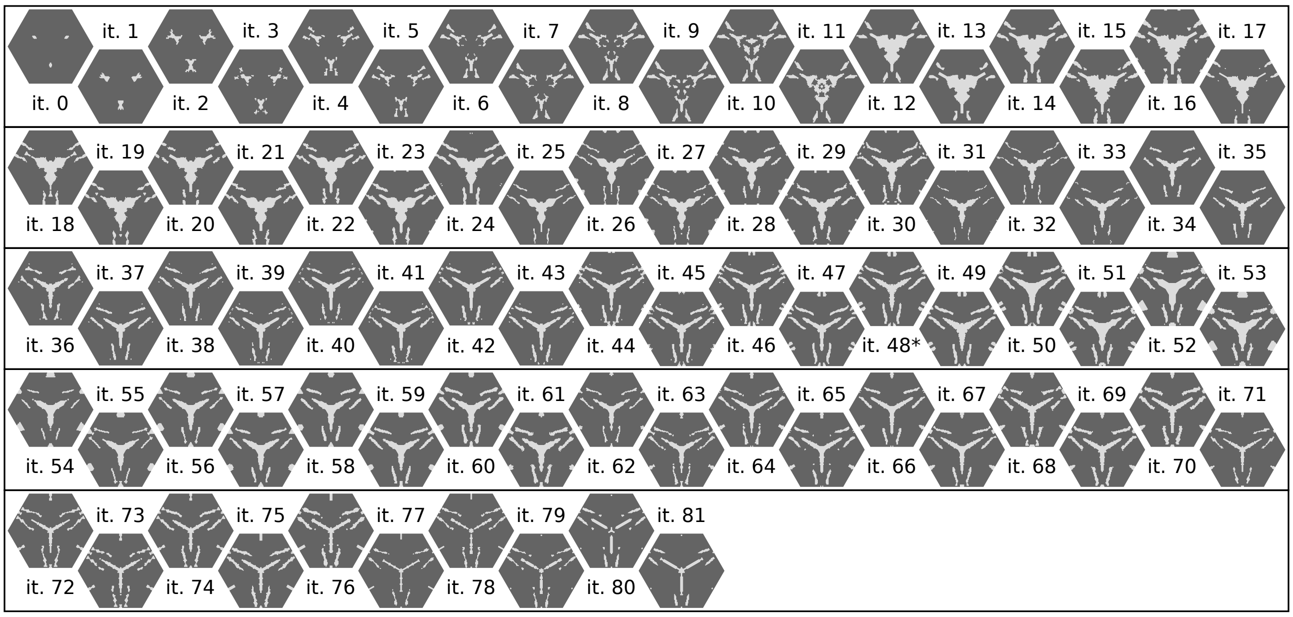

Figure 15: Topologies generated for $\nu^* = -0.30$ and $E_{\min} = 20.0\%$

Besides the topologies, homogenized properties and volume values, sensitivity and displacements vectors of each iteration are stored. Figure 16 shows the 12 sensitivity vectors stored for the optimized topology. The maps for ΔC_{00} are in the first row, the ones for ΔC_{11} are in the second row and the ones for ΔC_{22} are in the third row. The first column corresponds to the CGS-0 estimations, the second column to the CGS-1 estimations, the third column to the CGS-2 estimations and the fourth column to the exact values, computed through WS approach. An independent nonlinear scale is adjusted for each one of the 12 maps, in order to improve contrast resolution, so what is being shown is only how the sensitivity values are distributed among the different elements, not their quantitative values. Although they are referred to as sensitivity values, they correspond to the **variations** of the diagonal terms of $\boldsymbol{C}$ when the state of each augmented element is switched. To obtain proper sensitivity values, the signs of the values corresponding to solid augmented elements would have to be reversed.

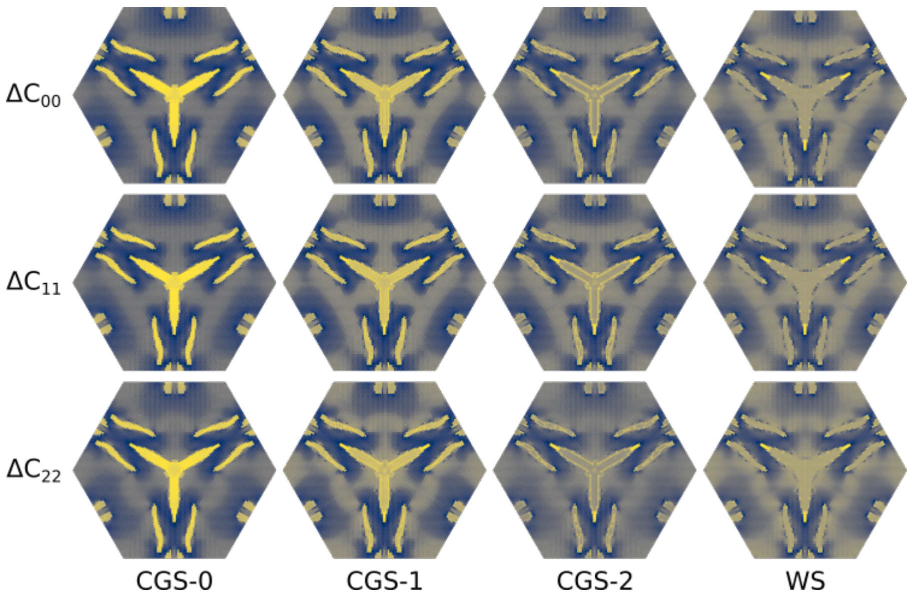

Figure 16: Sensitivity maps for the optimized topology (iteration 48)

In Figure 17, the deformed cell is presented for the three different values of macro-displacements imposed to perform the homogenization procedure. The displacements values are rescaled to improve visualization.

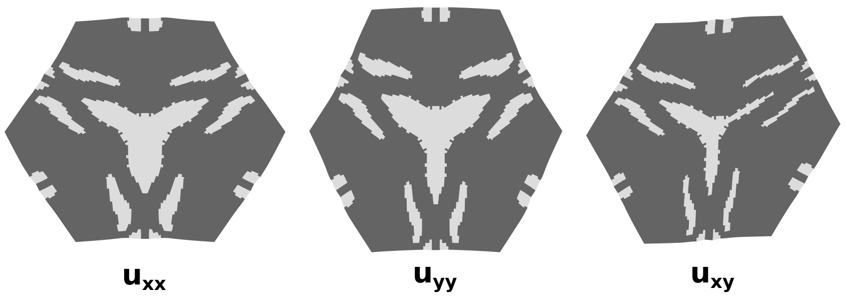

Figure 17: Deformed structures for the optimized topology (iteration 48)

To understand why there are cases in which a large number of iterations is necessary to achieve the stopping criterion, the case with maximal number of iterations is presented. The case in which $\nu^* = 0.57$ and $E_{\min} = 6.5\%$ needed 292 iterations to converge. Figure 18 presents the evolution of the Poisson's ratio, Young's modulus and volume fraction over the optimization and Figure 19 presents the topologies for some iterations. The best obtained result corresponds to iteration 262, its homogenized properties are $\widehat{\nu} = 0.57$ and $\widehat{E} = 15.3\%$.

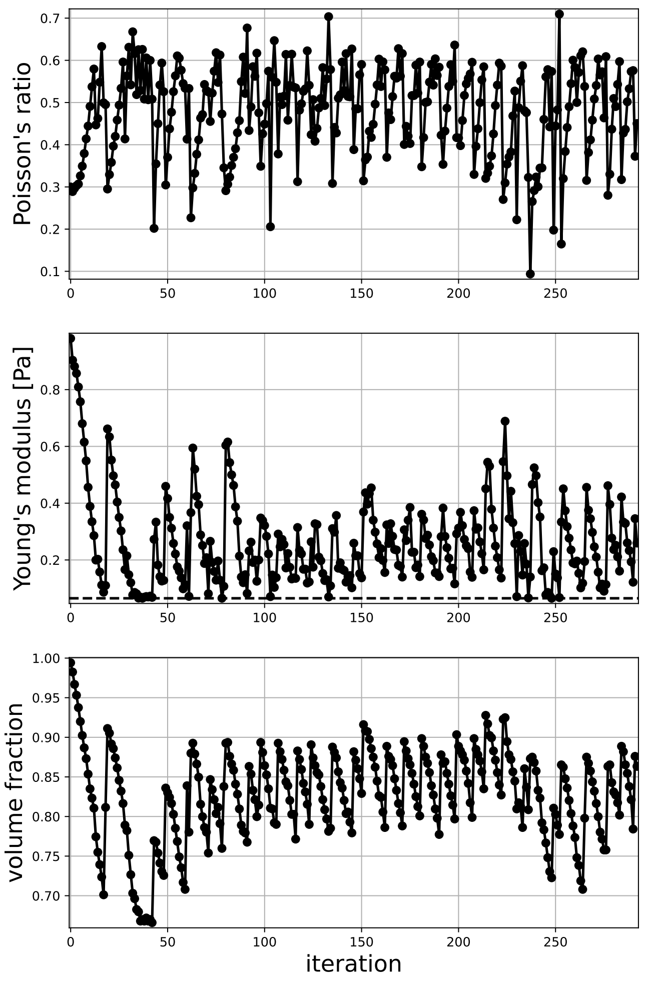

Figure 18: Poisson's ratio, Young's modulus and volume fraction for $\nu^* = 0.57$ and $E_{\min} = 6.5\%$

The procedure took so long because the dilation procedure often produced topologies that were in the basins of attraction of different local minima. Thus, more candidates were tried out throughout the iterative procedure. After iteration 262, the method was not able to keep improving the objective function, so the stopping criterion was reached.

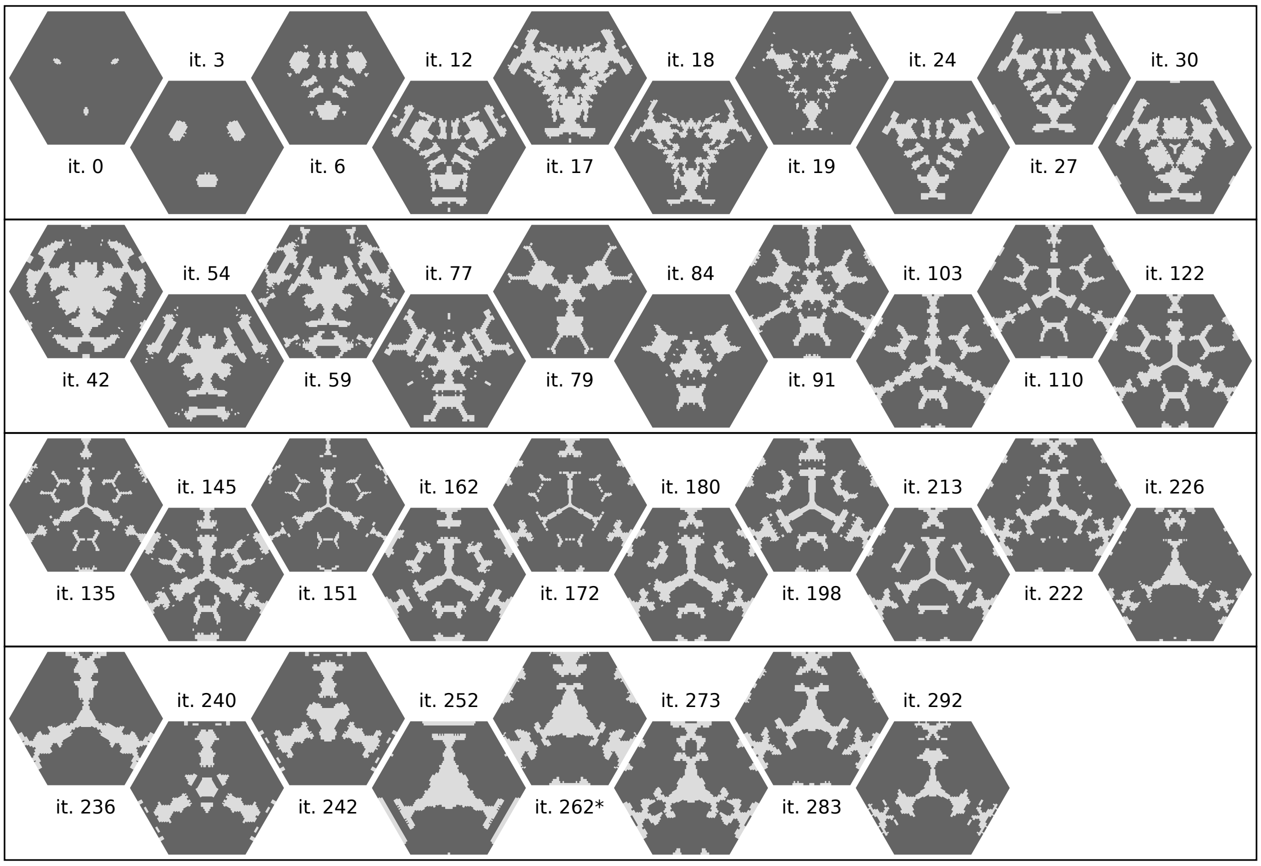

Figure 19: Topologies generated for $\nu^* = 0.57$ and $E_{\min} = 6.5\%$

Although it may be computationally expensive to perform so many iterations, in the SILP approach with the considered patience stopping criterion, a large number of iterations means that the method consistently improved the result along all the optimization procedure. Whenever there are sufficient time and computational resources, a more extensive exploration of the domain of possible solutions is advantageous, since it can yield more effective optimized structures.

On the other hand, there are cases in which a small number of iterations is enough to achieve the stopping criterion, the case with minimal number of iterations is presented. The case in which $\nu^* = -0.89$ and $E_{\min} = 40.5\%$ needed 37 iterations to converge. Figure 20 presents the evolution of the Poisson's ratio, Young's modulus and volume fraction over the optimization and Figure 21 presents the topologies for all iterations. The best obtained result corresponds to iteration 7, its homogenized properties are $\widehat{\nu} = 0.08$ and $\widehat{E} = 42.9\%$.

The high value for the minimal Young's modulus makes it easier to break the constraint, so only 7 iterations are performed before the first dilation. In this case, the perturbations from the dilation procedures were not enough to take the topologies out of the basins of attraction of the current local minimum. So, before each dilation, similar structures were obtained. Here, the first candidate ended up being better than all others obtained in subsequent iterations, so the optimization process took only 37 iterations to be concluded.

This behavior is undesirable, since the program is prevented from properly exploring the domain of possible solutions. In future versions, the stopping criterion may be improved to avoid such early convergences. Nonetheless, the obtained result is reasonable and suitable for the proposed dataset.

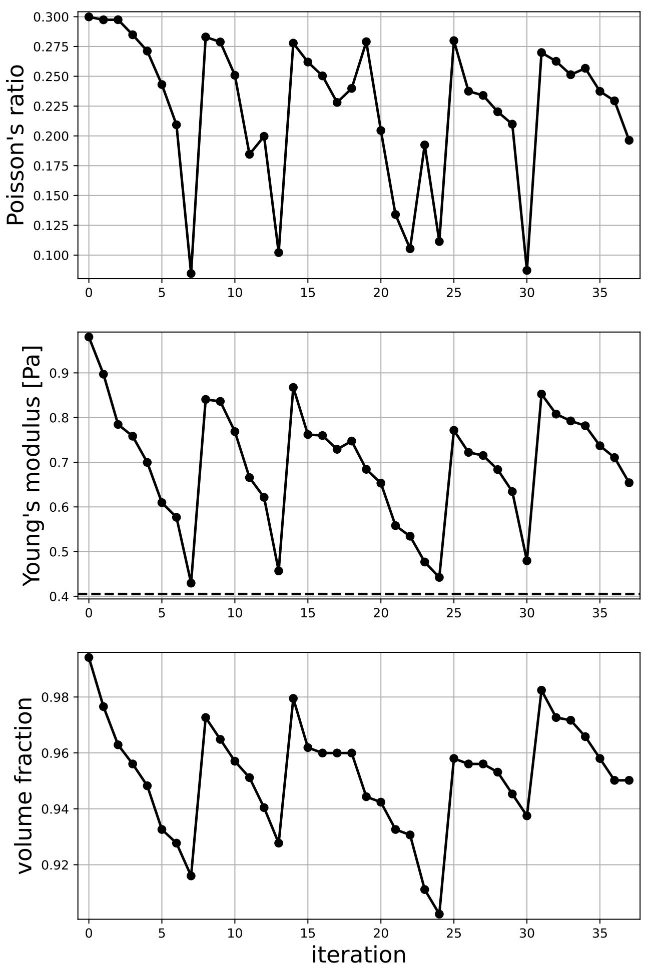

Figure 20: Poisson's ratio, Young's modulus and volume fraction for $\nu^* = -0.89$ and $E_{\min} = 40.5\%$

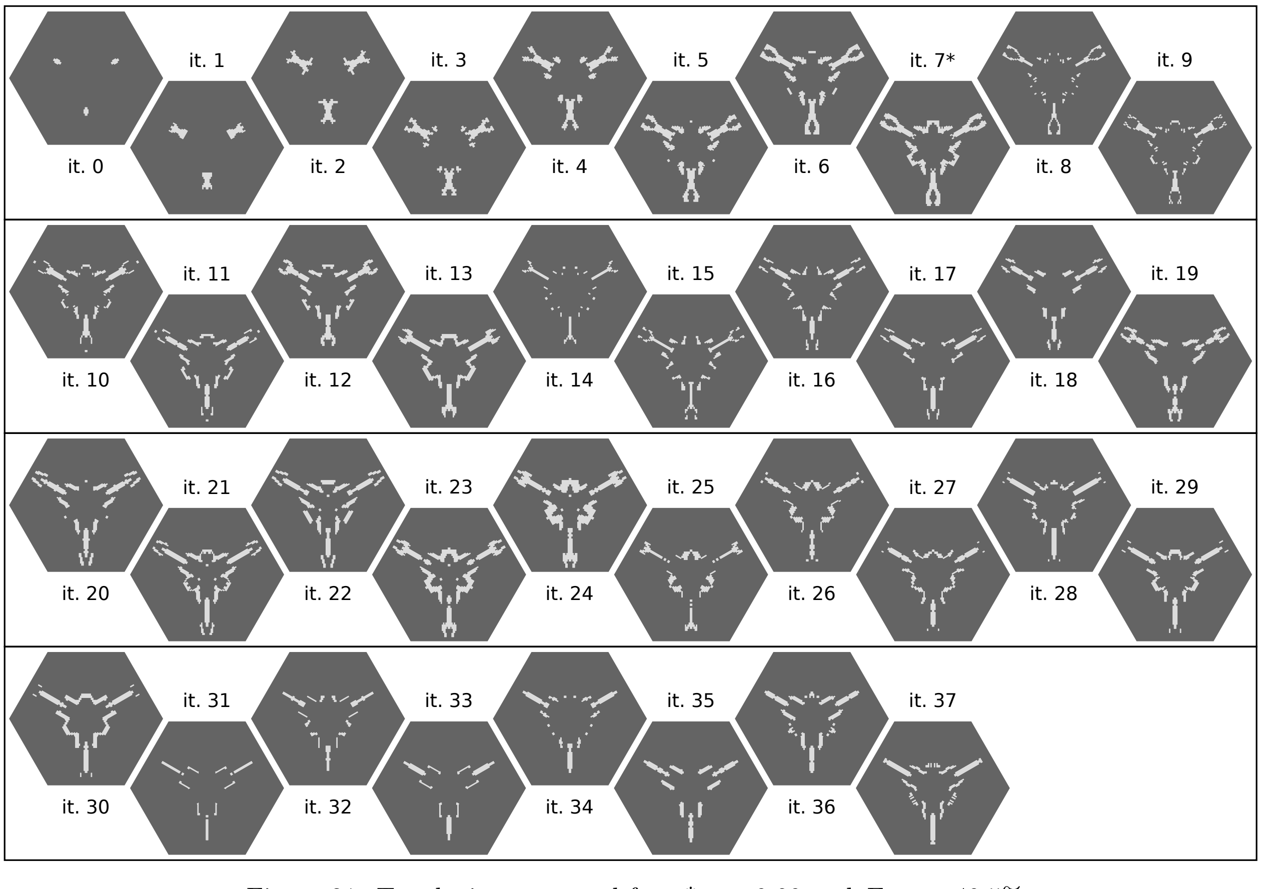

Figure 21: Topologies generated for $\nu^* = -0.89$ and $E_{\min} = 40.5\%$

It should be noted that there was a bug (already fixed) in the programs used to generate the data shown in this section. It did not compromise anything but the stability of the optimization procedures, so the presented samples are legitimate data. This only means that different results (possibly better) may be obtained when generating the dataset with the current version of the programs (presented in this document).

5 Summary

The specified values for each fixed parameter of the considered problem are presented in Table 1.

Table 1: Fixed parameters

Name (Documentation)	Name (Programs)	Value	Description
N_s	Ns	32	number of elements in each axis
N_d	N	1024	number of design variables
N_t	Nt	6144	total number of quadrilateral elements
L_x	Lx	$\approx$ 0.310 m	design domain shorter side
L_y	Ly	$\approx$ 0.537 m	design domain longer side
e_x	Lex	$\approx$0.00969 m	shorter side length of the elements
e_y	Ley	$\approx$0.01679 m	longer side length of the elements
maximal decrease of $\widehat{E}$	Eyvar	0.05 Pa	maximal decrease in Young's modulus per iteration
$D_{\max}$	Dmax	1.5625%	maximal topology variation
r_s	rsen	0.024 m	sensitivity filter radius
r_m	rmor	0.018 m	morphology filter radius
P	patience	30	patience stopping criterion
momentum	momentum	25%	momentum value for the objective function sensitivity
β	beta	0.05	volume penalization factor
$\breve{E}$	Ey	1.0 Pa	Young's modulus of the base material
$\breve{\nu}$	nu	0.3	Poisson's ratio of the base material
p_k	pk	1×10^{-9}	soft-kill parameter
noptf	noptf	7	number of optimizations stored in the same file

The range of values for each input parameter is presented in Table 2. The target Poisson's ratio ranges from -1.0 to 1.0, in steps of 0.01. Values between 0.20 and 0.40 are not considered since they are close to the property of the base material, so the values $\{0.21, 0.22, \ldots, 0.39\}$ are taken out of the set. This results in 182 possible values for ν^*. The minimal Young's modulus ranges from 0.0% to 50.0%, in steps of 0.5%. This results in 101 possible values for $E_{\min}$. Therefore, there are $18\,382 = 182 \times 101$ unique pairs $(\nu^*, E_{\min})$.

Table 2: Input parameters

Name	Range of values	Description
ν^*	$-1.00 \sim 0.20 \mid 0.40 \sim 1.00$	target Poisson's ratio
$E_{\min}$	$0.0\% \sim 50.0\%$	minimal Young's modulus

The github repository (`https://github.com/Joquempo/Metamaterial-Dataset`) is structured as presented in Figure 22. The "sample" folder contains scripts for taking samples from the generated dataset. The "source" folder contains the scripts that generate the dataset. The "validation" folder contains scripts used to validate the presented implementation. 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.

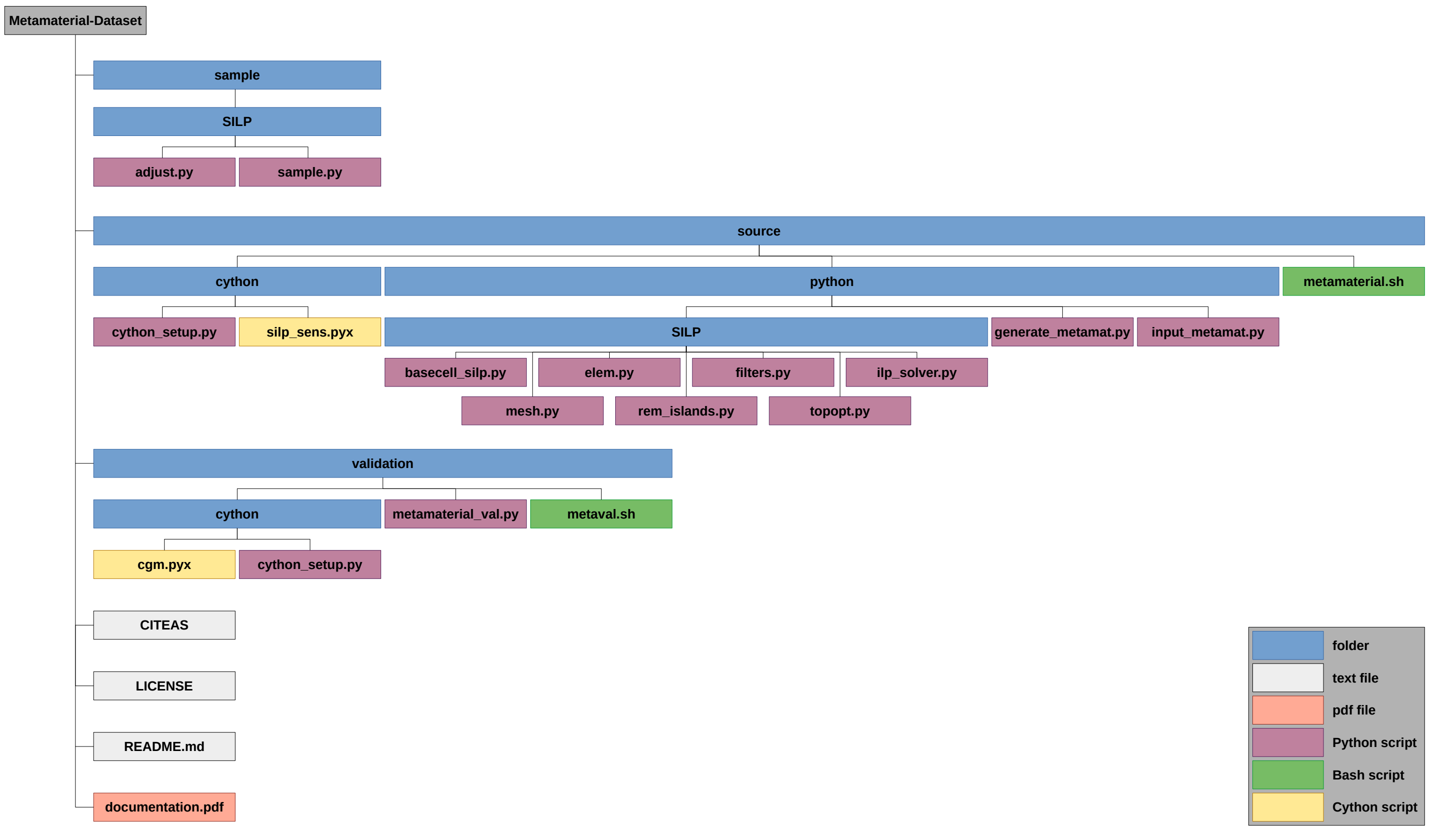

Figure 22: Metamaterial-Dataset repository

The "./source/python/SILP/basecell_silp.py" script 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=700", the program will run the first 700 cases, $\{0, 1, \ldots, 699\}$; if "fid_ini=700" and "fid_lim=2800", it will run the subsequent 2100 cases, $\{700, 701, \ldots, 2799\}$.

After installing and activating Anaconda, the dataset can be generated by executing the scripts ordered in the execution tree shown in Figure 23. 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 dataset, samples can be taken by using the scripts from the "sample" folder. To select which samples should be taken, the "./sample/SILP/sample.py" script 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/metaval.sh" can be executed to build the required cython codes. Then, after activating the "metamaterial" conda environment, the "./validation/metamaterial_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 validation figures (otherwise, the script should be altered so the figures are not immediately closed after concluding the validation procedure).

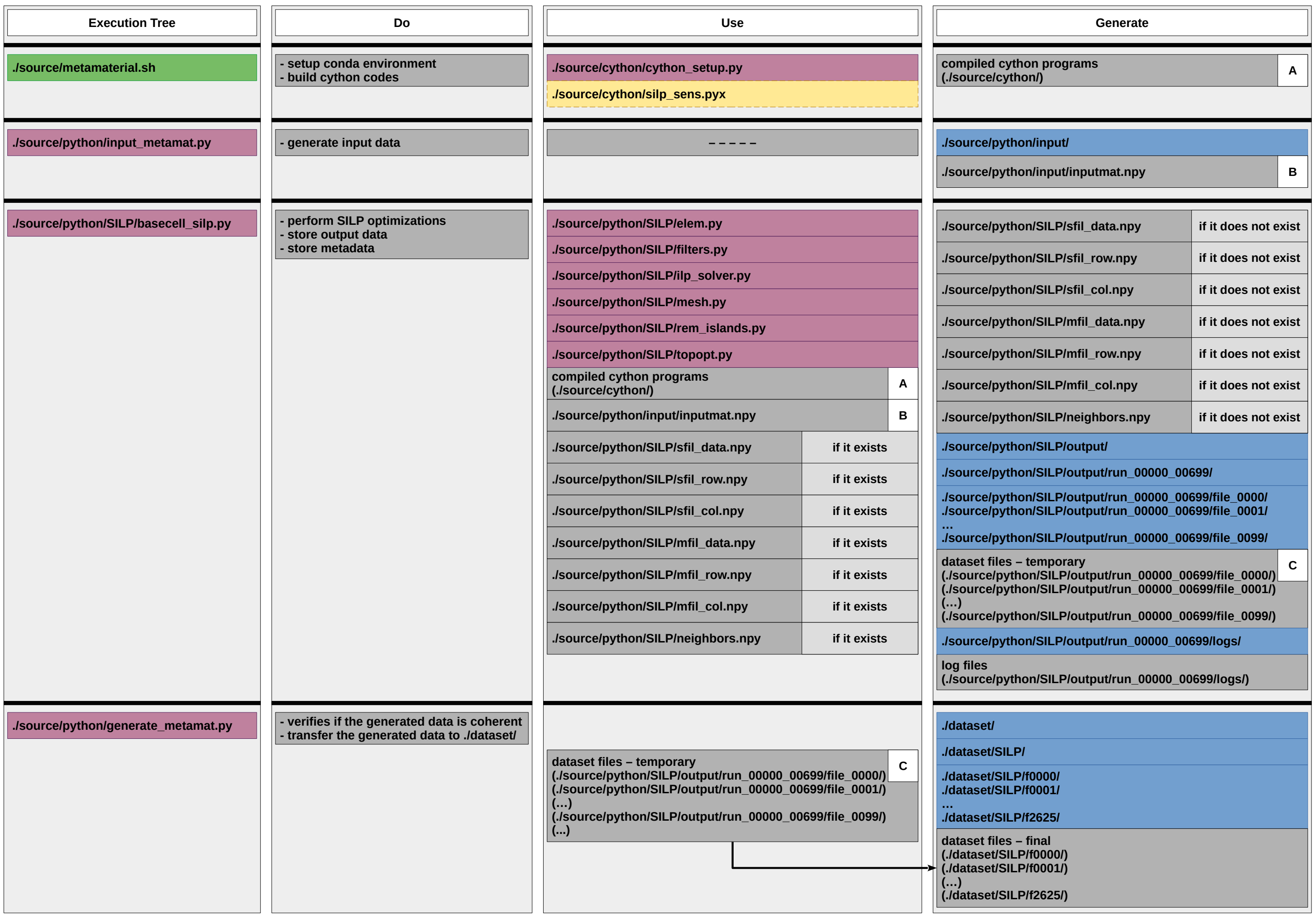
Execution Tree
Do
Use
Generate
./source/metamaterial.sh
- setup conda environment
- build cython codes
./source/cython/cython_setup.py
./source/cython/silp_sens.pyx
compiled cython programs (./source/cython/)
A
./source/python/input_metamat.py
- generate input data
– – – – –
./source/python/input/
./source/python/input/inputmat.npy
B
./source/python/SILP/basecell_silp.py
- perform SILP optimizations
- store output data
- store metadata
./source/python/SILP/elem.py
./source/python/SILP/filters.py
./source/python/SILP/ilp_solver.py
./source/python/SILP/mesh.py
./source/python/SILP/rem_islands.py
./source/python/SILP/topopt.py
compiled cython programs (./source/cython/)
A
./source/python/input/inputmat.npy
B
./source/python/SILP/sfil_data.npy
if it exists
./source/python/SILP/sfil_row.npy
if it exists
./source/python/SILP/sfil_col.npy
if it exists
./source/python/SILP/mfil_data.npy
if it exists
./source/python/SILP/mfil_row.npy
if it exists
./source/python/SILP/mfil_col.npy
if it exists
./source/python/SILP/neighbors.npy
if it exists
./source/python/SILP/sfil_data.npy
if it does not exist
./source/python/SILP/sfil_row.npy
if it does not exist
./source/python/SILP/sfil_col.npy
if it does not exist
./source/python/SILP/mfil_data.npy
if it does not exist
./source/python/SILP/mfil_row.npy
if it does not exist
./source/python/SILP/mfil_col.npy
if it does not exist
./source/python/SILP/neighbors.npy
if it does not exist
./source/python/SILP/output/
./source/python/SILP/output/run_00000_00699/
./source/python/SILP/output/run_00000_00699/file_0000/
./source/python/SILP/output/run_00000_00699/file_0001/
...
./source/python/SILP/output/run_00000_00699/file_0099/
dataset files – temporary
(./source/python/SILP/output/run_00000_00699/file_0000/)
(./source/python/SILP/output/run_00000_00699/file_0001/)
(...)
(./source/python/SILP/output/run_00000_00699/file_0099/)
C
./source/python/SILP/output/run_00000_00699/logs/
log files
(./source/python/SILP/output/run_00000_00699/logs/)
./source/python/generate_metamat.py
- verifies if the generated data is coherent
- transfer the generated data to ./dataset/
dataset files – temporary
(./source/python/SILP/output/run_00000_00699/file_0000/)
(./source/python/SILP/output/run_00000_00699/file_0001/)
(...)
(./source/python/SILP/output/run_00000_00699/file_0099/)
(...)
C
./dataset/
./dataset/SILP/
./dataset/SILP/f0000/
./dataset/SILP/f0001/
...
./dataset/SILP/f2625/
dataset files – final
(./dataset/SILP/f0000/)
(./dataset/SILP/f0001/)
(...)
(./dataset/SILP/f2625/)

Figure 23: Execution tree

Table 3 presents the dataset files. The size shown in the second column corresponds to the amount of the useful data, considering the average value of 74 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 7 optimization processes, so 2 626 of each one is generated after running all the 18 382 cases.

Table 4 presents the disk usage for the complete dataset.

Since all data is stored as numpy arrays, through “numpy.save(·)”, everything can be easily read through “numpy.load(·)”. In anyway, the sampling script illustrates how to read data from the generated dataset. As a final remark, it should be noted that careless sampling can quickly write a lot of data in your disk, so be cautious when running the sampling script.

Table 3: Dataset files

Name	Size (useful data)	Size (observed disk usage)	Description
dC00_0.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-0 finite variations of C_{00}
dC11_0.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-0 finite variations of C_{11}
dC22_0.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-0 finite variations of C_{22}
dC00_1.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-1 finite variations of C_{00}
dC11_1.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-1 finite variations of C_{11}
dC22_1.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-1 finite variations of C_{22}
dC00_2.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-2 finite variations of C_{00}
dC11_2.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-2 finite variations of C_{11}
dC22_2.npy	2.1 MB	1.3 ∼ 4.6 MB	CGS-2 finite variations of C_{22}
dC00_w.npy	2.1 MB	1.3 ∼ 4.6 MB	exact finite variations of C_{00}
dC11_w.npy	2.1 MB	1.3 ∼ 4.6 MB	exact finite variations of C_{11}
dC22_w.npy	2.1 MB	1.3 ∼ 4.6 MB	exact finite variations of C_{22}
dis_xx.npy	25.4 MB	15.9 ∼ 56.3 MB	displacements vectors used to compute C_{00}
dis_yy.npy	25.4 MB	15.9 ∼ 56.3 MB	displacements vectors used to compute C_{11}
dis_xy.npy	25.4 MB	15.9 ∼ 56.3 MB	displacements vectors used to compute C_{22}
Ey.npy	2.1 kB	4.1 ∼ 8.2 kB	Young's modulus values
Ey_opt.npy	28 bytes	4.1 ∼ 4.1 kB	optimized Young's modulus values
fid.npy	28 bytes	4.1 ∼ 4.1 kB	input files indices
inp.npy	56 bytes	4.1 ∼ 4.1 kB	input data
nu.npy	2.1 kB	4.1 ∼ 8.2 kB	Poisson's ratio values
nu_opt.npy	28 bytes	4.1 ∼ 4.1 kB	optimized Poisson's ratio values
ptr2inp.npy	2.1 kB	4.1 ∼ 8.2 kB	pointers from iterations to inputs
ptr2opt.npy	32 bytes	4.1 ∼ 4.1 kB	pointers from inputs to iterations
tim.npy	196 bytes	4.1 ∼ 4.1 kB	execution times and number of iterations
top.npy	65.6 kB	41.0 ∼ 143.4 kB	topology vectors
top_opt.npy	896 bytes	4.1 ∼ 4.1 kB	optimized topologies
vol.npy	2.1 kB	4.1 ∼ 8.2 kB	relative volume values

Table 4: Disk usage

	Useful data	Observed disk usage
Complete dataset	260 GB	**277 GB**

6 Acknowledgements

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

Bibliography

[1] Cristian Barbarosie, Daniel A Tortorelli, and Seth Watts. On domain symmetry and its use in homogenization. *Computer Methods in Applied Mechanics and Engineering*, 320:1–45, 2017.

[2] Tomasz Łukasiak. Macroscopically isotropic and cubic-isotropic two-material periodic structures constructed by the inverse-homogenization method. In *World Congress of Structural and Multidisciplinary Optimisation*, pages 1333–1348. Springer, 2017.

[3] Juan Manuel Podestá, CM Méndez, Sebastian Toro, and Alfredo Edmundo Huespe. Symmetry considerations for topology design in the elastic inverse homogenization problem. *Journal of the Mechanics and Physics of Solids*, 128:54–78, 2019.

[4] C Méndez, JM Podestá, S Toro, et al. Making use of symmetries in the three-dimensional elastic inverse homogenization problem. *International Journal for Multiscale Computational Engineering*, 17(3), 2019.

[5] Julien Yvonnet. *Computational homogenization of heterogeneous materials with finite elements*, volume 258. Springer, 2019.

[6] Daniel Candeloro Cunha and Renato Pavanello. Finite variation sensitivity analysis in the design of isotropic metamaterials through discrete topology optimization. *International Journal for Numerical Methods in Engineering*, page e7560, 2023.

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

[8] 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.

[9] 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.

[10] R Sivapuram and R Picelli. Topology optimization of binary structures using integer linear programming. *Finite Elements in Analysis and Design*, 139:49–61, 2018.

[11] Raghavendra Sivapuram, Renato Picelli, and Yi Min Xie. Topology optimization of binary microstructures involving various non-volume constraints. *Computational Materials Science*, 154:405–425, 2018.

[12] Osvaldo M Querin, Grant P Steven, and Yi Min Xie. Evolutionary structural optimisation (eso) using a bidirectional algorithm. *Engineering computations*, 15(8):1031–1048, 1998.

[13] X Y Yang, Y M Xie, G P Steven, and O M Querin. Bidirectional evolutionary method for stiffness optimization. *AIAA journal*, 37(11):1483–1488, 1999.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Ole Sigmund. Morphology-based black and white filters for topology optimization. *Structural and Multidisciplinary Optimization*, 33(4):401–424, 2007.

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

[20] 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.

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

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

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

[24] Stuart Mitchell, Michael OSullivan, and Iain Dunning. Pulp: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand*, 65, 2011.