精读

Advances in Engineering Software 85 (2015) 1–11

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

# A simple and compact Python code for complex 3D topology optimization

Zhi Hao Zuo, Yi Min Xie [*]

*Centre for Innovative Structures and Materials, School of Civil, Environmental and Chemical Engineering, RMIT University, GPO Box 2476, Melbourne 3001, Australia*

ARTICLE INFO

*Article history:*
Received 13 December 2014
Accepted 11 February 2015
Available online 4 March 2015

*Keywords:*
Topology optimization
BESO
Python
Abaqus
NumPy
Large deflection

ABSTRACT

This paper presents a 100-line Python code for general 3D topology optimization. The code adopts the Abaqus Scripting Interface that provides convenient access to advanced finite element analysis (FEA). It is developed for the compliance minimization with a volume constraint using the Bi-directional Evolutionary Structural Optimization (BESO) method. The source code is composed of a main program controlling the iterative procedure and five independent functions realizing input model preparation, FEA, mesh-independent filter and BESO algorithm. The code reads the initial design from a model database (.cae file) that can be of arbitrary 3D geometries generated in Abaqus/CAE or converted from various widely used CAD modelling packages. This well-structured code can be conveniently extended to various other topology optimization problems. As examples of easy modifications to the code, extensions to multiple load cases and nonlinearities are presented. This code is useful for researchers in the topology optimization field and for practicing engineers seeking automated conceptual design tools. With further extensions, the code could solve sophisticated 3D conceptual design problems in structural engineering, mechanical engineering and architecture practice. The complete code is given in the appendix section and can also be downloaded from the website: www.rmit.edu.au/research/cism/.

## 1. Introduction

Structural topology optimization is a topic concerning the best locations for cavities in a structure design domain. This topic has been intensively investigated in the past three decades. During this time, several educational articles were published aimed at introducing fundamentals of various algorithms by presenting computer program implementations, such as the Matlab codes by Sigmund [19] for the Solid Isotropic Material with Penalization (SIMP) method [2,25] and Challis [5] for the level-set method [16,21]. Also an 88-line Matlab code was developed by Andreassen et al. [1] as an extension to Sigmund's 99-line Matlab code with improved efficiency. These works demonstrated basic 2D compliance minimization based on a rectangular design domain discretized into linear 4-noded elements and served the educational purposes well. Huang and Xie [10] also published a Matlab code for 2D compliance minimization using the Bi-directional Evolutionary Structural Optimization (BESO) method [7,11]. Recently Liu and Tovar [15] presented a Matlab implementation using a modified SIMP model for 3D topology optimization for linear structures with regular 8-noded elements. Inspired by these works, this paper presents a Python code for 3D topology optimization using the BESO method. The Python code is developed based on the Abaqus environment that provides a broad range of linear/nonlinear static/dynamic FEA capacities and meshing techniques. The code can be well extended to practical applications. Selected extensions are presented in the paper to enhance computational efficiency and to consider multiple load cases and geometrical nonlinearity. All the Python source codes for the basic implementation together with the selected extensions are given in the appendix section.

The BESO method is developed from the Evolutionary Structural Optimization (ESO) method [22,23]. The original ESO is a heuristic method based on removing lowly-stressed materials. Through years of development, the current form of BESO has become a gradient-based mathematical optimization method with convergent and mesh-independent algorithms [7]. Recent applications of BESO have addressed a range of macro structural optimization problems including compliance minimization [9], frequency maximization [14], displacement constraints [12,27], as well as inverse homogenization [18] problems such as material moduli maximization [24,17] and concurrent designs [13,26].

The Python code provided in this paper is based on the "soft-kill" BESO scheme [9]. Like the SIMP method, BESO is based finite

* Corresponding author. Tel.: +61 3 99253655; fax: +61 3 96390138.
*E-mail address:* mike.xie@rmit.edu.au (Y.M. Xie).

http://dx.doi.org/10.1016/j.advengsoft.2015.02.006
0965-9978/© 2015 Elsevier Ltd. All rights reserved.

element analysis (FEA) and treats the relative densities of elements as the design variables. For a discrete 0–1 design problem, a design variable takes values of either 1.0 indicating the presence of the element (solid status), or a prescribed small number such as 0.001 (usually denoted as $x_{\min}$), indicating the absence of the element (void status). The term "soft-kill" is used in contrast to "hard-kill" where elements are completely removed to create cavities. In the soft-kill scheme, the removal of an element is realized by switching the relative density from 1.0 to $x_{\min}$ instead of a complete deletion. The Python code provides detailed insights into realizing topology optimization through a line-to-line presentation. For a good programming practice as well as for a clear explanation, the global flow of execution is coded as the main program, with the detailed implementations being coded into five separate functions: one for preparing the input model for optimization, one for running FEA, one for updating design variables using the BESO algorithm, and two for performing the mesh-independence filter scheme including pre-mapping related elements and averaging sensitivities based on the mapping.

Abaqus [6] is a general purpose commercial software package widely used in aerospace, civil, mechanical and automotive industries. Note that a student edition of Abaqus with a maximum model size limited to 1000 nodes is also available from Dassault Systemes for educational purposes. The Python code takes advantage of the advanced FEA capacities of the Abaqus software and employs the Abaqus Scripting Interface (ASI) to communicate with Abaqus. ASI is an extension of the Python language. As a standard component of the Abaqus software, ASI provides a convenient interface to the models and results. With access to the sophisticated modelling capability provided by Abaqus/CAE or other CAD packages, the Python code is able to deal with arbitrarily-shaped 2D/3D geometries; in other words, the present code is not limited to fixed simple 2D geometries as in the previous Matlab codes [19,5,10]. Through ASI, FEA can be easily performed with a defined function that is only nines line long; preparing the input model is performed by another 12-line function defined for formatting the material properties according to the soft-kill scheme. The rest of the code is for a standard BESO algorithm that is only 58 lines long excluding header and comments. The extensions to the code are mainly made in the FEA and model preparation functions, without changing the BESO part.

The remainder of the paper is organized as follows: Section 2 gives an introduction of the topology optimization algorithms, Section 3 explains the Python code in detail, Section 4 presents several extensions to the basic code, Section 5 draws the conclusions, and the complete Python codes are given in the appendix section.

## 2. Topology optimization algorithms behind the Python implementation

### 2.1. Problem statement

The topology optimization problem considered in this paper is that of compliance minimization subject to a volume constraint. The mathematical description of this problem is as follows.

$$\min_{\mathbf{X}} : \quad C(\mathbf{X}) = \mathbf{F}^T\mathbf{U} = \mathbf{U}^T\mathbf{K}\mathbf{U} \tag{1}$$

$$\text{subject to}: \quad \mathbf{X} = \{x_e\},\ x_e = 1 \ or\ x_{\min} \quad \forall\, e = 1,\ldots,N \tag{2}$$

$$\mathbf{F} = \mathbf{K}\mathbf{U} \tag{3}$$

$$V(\mathbf{X}) = \sum_{\mathbf{X}} x_e v_e = V^* \tag{4}$$

where the compliance $C$ is the objective function; $\mathbf{X}$ is the vector of the elemental relative densities and thus vector of binary design variables as in a common discrete problem; $x_e$ is the $e$th design variable with candidate values of either 1 for solid element (presence) or prescribed $x_{\min}$ (0.001 in this paper) for void element (absence); $N$ is the total number of elements; $\mathbf{F}$ and $\mathbf{U}$ are the global force vector and displacement vector respectively; $\mathbf{K}$ is the global stiffness matrix; $V$ is the total volume of the structure with $v_e$ being the elemental volume; $V^*$ is the imposed value of the volume constraint. Besides the volume constraint in Eq. (4), the constraint defined in Eq. (3) ensures the equilibrium of the structure.

### 2.2. Optimization algorithms

The BESO method is employed in the Python code to solve the above optimization problem. As a gradient-based method, the design variable update is based on the element sensitivity $\alpha_e$ obtained by differentiating the objective function $C$.

$$\alpha_e = \frac{\partial C}{\partial x_e} \tag{5}$$

Detailed calculation of the above differentiation under linear elasticity can be performed based on the SIMP material model [3,9] which yields

$$\alpha_e = -p x_e^{p-1}\mathbf{u}_e^T\mathbf{k}_0\mathbf{u}_e = -\frac{p}{x_e}x_e^p\mathbf{u}_e^T\mathbf{k}_0\mathbf{u}_e = -p\frac{E_e}{x_e} \tag{6}$$

where $p$ is the penalty exponent [3] and has a constant value of 3 in this paper, $\mathbf{u}_e$ is the elemental displacement vector and $\mathbf{k}_0$ is the element stiffness matrix in a solid status (i.e. $x_e = 1$). It is noted that the sensitivity contains a term $x^p\mathbf{u}_e^T\mathbf{k}_0\mathbf{u}_e$ which is exactly the element strain energy $E_e$ that can be directly obtained from FEA. The relevant Python implementation of the sensitivity is defined in the function *FEA*.

This raw sensitivity is usually processed in order to produce a mesh-independent solution [20]. A blurring filter can be used for this purpose such as the filter scheme in Huang and Xie [7]. Without losing the solution accuracy, a simplified elemental sensitivity filtering scheme is used in this paper as follows.

$$\hat{\alpha}_e = \frac{\sum_j w(r_{ej})\alpha_j}{\sum_j w(r_{ej})} = \sum_j\left(\frac{w(r_{ej})}{\sum_j w(r_{ej})}\alpha_j\right) = \sum_j \eta_j\alpha_j \tag{7}$$

$$w(r_{ej}) = \max(0, r_{\min} - r_{ej}) \tag{8}$$

where $r_{ej}$ is the distance between the centres of elements $e$ and $j$; $w$ is a weight function for averaging the raw sensitivities; $r_{\min}$ is the filter radius. It is noted that the weight factor $\eta_j$ is independent of the sensitivity values and can be calculated beforehand. The Python implementation of this filter scheme is divided into two functions: *preFilt* that calculates only once and saves the fixed weight factors, and *fltAe* that applies the filter in each iteration.

In order to achieve a convergent solution, it is recommended to further average the sensitivity with its historical information for discrete methods such as BESO [7]. This is realized by simply averaging the sensitivity of the current iteration with that of the previous iteration.

$$\tilde{\alpha}_e = \frac{\hat{\alpha}_e^k + \hat{\alpha}_e^{k-1}}{2} \tag{9}$$

where $k$ is the current iteration number. In the Python code, this is easily implemented with a one-line expression in the main program.

BESO usually starts from a full design and reduces the structural volume iteratively by switching element status. In the iteration, the target volume of next iteration $V^{k+1}$ is determined based on the current volume $V^k$ and an Evolutionary Ratio *ert*.

$$V^{k+1} = V^k(1 \pm ert) \tag{10}$$

```
Input: sensitivities ae, design variables xe,
        target volume tv
Output: threshold th
lo = min(ae), hi = max(ae)
While (hi - lo)/hi > 0.00001
    th = (lo + hi)/2
    For i = 1 to N
        If ae[i] > th: xe[i] = 1
        Else: xe[i] = 0.001
    If(sum(xe) – tv > 0): lo = th
    Else: hi = th
End while
```

**Fig. 1.** Pseudocode of the bisection algorithm used in *BESO* for finding the sensitivity threshold.

Then the element update is based on the optimality criteria for soft-kill BESO [10] that the sensitivities of solid elements ($x_e = 1$) are always lower than those of void elements ($x_e = x_{\min}$), for a minimization problem. The update scheme is then devised according to the target volume and the sensitivities: a threshold *th* shall be determined so that elements with a sensitivity lower than the threshold are switched to solid (if void) and elements with a sensitivity higher than the threshold are switched to void (if solid), the result of which achieves the target volume. The threshold can be determined using a simple bisection method that changes the threshold progressively as shown in Fig. 1. In the Python code, this update algorithm is defined in the function *BESO*.

## 3. Python implementation

This section explains the basic form of the Python code which is given in Appendix A in full. The code is to be called in Abaqus/CAE through the menu command File->Run Script.

The Python code makes no assumption on the geometry of the design domain. The design domain and required parameters are transferred to the program as input arguments through two prompt dialogues: the first dialogue allows the user to input the predefined parameters for the target final volume fraction, the filter radius and the evolutionary volume ratio; the second dialogue lets the user to specify a model database file (.cae) that contains the model of the design domain (also as the initial structure) of an arbitrary geometry. Fig. 2 shows a 3D cantilever as an example with a filter radius of 4 and target volume fraction of 10%. Note that the input model database shall be prepared such that it contains a model named ‘Model-1’ with a dependent part named ‘Part-1’ and a static analysis step named ‘Step-1’.

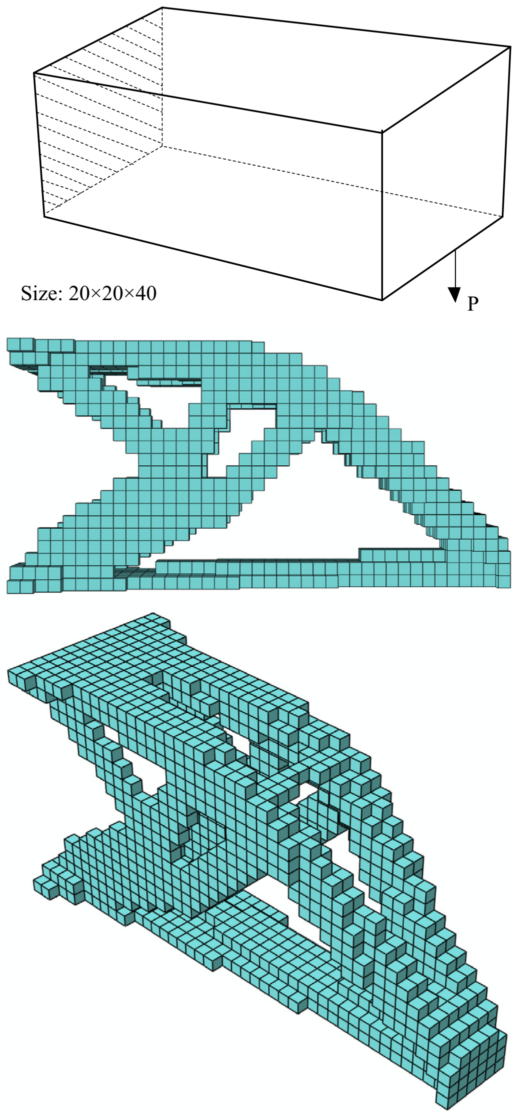

**Fig. 2.** Topology optimization of a 3D cantilever: (top) the initial full design as the design domain; (middle and bottom) side and perspective views of the optimized solution.

### *3.1. Header (lines 2–4)*

As a common programming practice, a header exists at the beginning of the script to import a number of modules and functions: the module *math* is imported for arithmetic functions such as *sqrt* and *fabs* (line 2); *customKernel* (line 2) is an Abaqus extended Python module for storing custom data in a model database; *getInput* and *getInputs* (line 3) are two functions imported from the *Abaqus* module for parsing the input parameters; and *openOdb* (line 4) is a function imported from the *odbAccess* module for accessing the FEA results in an Abaqus output database. The modules *Abaqus* and *odbAccess* are two common modules for most Python scripts employing ASI and are standard components of the Abaqus software.

### *3.2. Main program (lines 68–100)*

The main program defines the global execution flow. The main program locates at the end of the source code since the Python interpreter needs the definition of all other functions before they are called in the main program.

In the conventional Python programming fashion, the main program starts with such a line as “if __name__ == ‘__main__’:”. In the current main program, this line is followed by the function body that can be partitioned into four blocks: input acquisition, design initialization, iteration loop and result saving. In the input acquisition block, the multiple parameters are obtained (line 72) through the *getInputs* function and a preliminary verification is done (line 73) to prevent infeasible inputs such as negative final volume. A model database object is created referring to the input design model database and assigned to a variable *mddb* (line 74) using *openMdb* with the specified file name that is parsed by the *getInput* function. A model database object can contain several model objects, each of which further contains a series of member objects such as parts, materials and loads. A simplified Abaqus object model hierarchy is shown in Fig. 3 for the model database (mdb); an object model for the output database (odb) is also shown that will be employed later in function *FEA*.

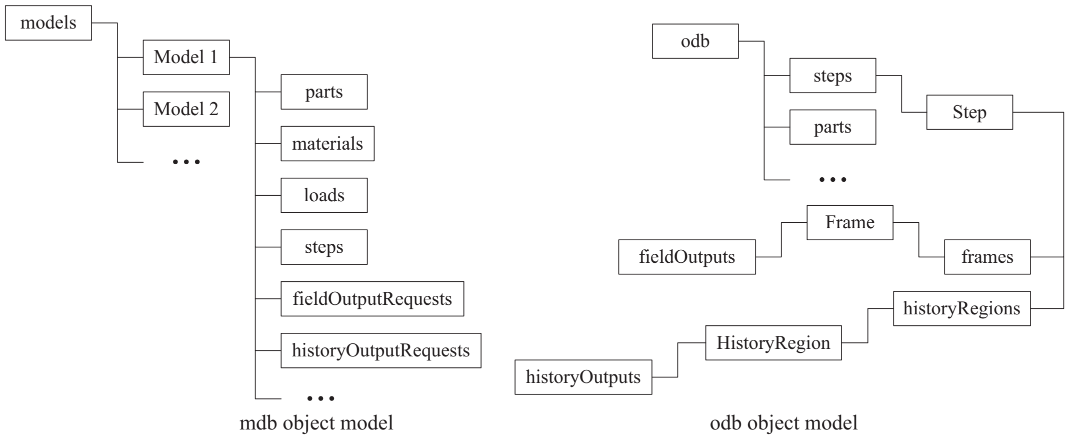

**Fig. 3.** Abaqus object model hierarchy for (left) model database – mdb; (right) output database – odb.

In the design initialization block, the initial design model base is first formatted (line 76) using a function *fmtMdb* that will be introduced in the following. Since the information in the design part, such as the elemental information, is frequently used in the program, three variables are defined referring to the objects: "Part-1" (*part*), the elements (*elmts*) and nodes (*nds*) in this part for programming simplicity. Two lists (Python data type) are defined to store the historical information for the objective function (*oh*) and volume fraction (*vh*) that will be used in verifying the solution convergence as well as in result presentation. Other initialized variables include the dictionaries (Python data type): the design variables (*xe*), the current element sensitivities (*ae*), the element sensitivities of an immediately previous iteration (*oae*) and the element mapping for the filter scheme (*fm*). Note that both list and dictionary are basic Python data types that store a series of individual objects or numbers. The function *preFlt* is called (line 82) to prepare the filter scheme if a valid filter radius $r_{\min}$ is present; note that $r_{\min} = 0$ will not trigger the filter scheme.

The iteration loop block is entry controlled by checking the convergence error *change* of the objective function (line 85). A counter *iter* is used to count the current number of iteration. In the body of the loop, first an *FEA* is executed (line 88) to obtain the objective function and element sensitivities *ae*; then *ae* are filtered (line 90, if a valid $r_{\min}$ is present) and history-averaged (line 91); note that the current sensitivities are immediately stored to *oae* for performing next history-averaging; then the *BESO* update (line 96) is executed based on *ae* and the current target volume fraction $nv$ that is determined (line 95) according to Eq. (10); finally, a convergence check (line 97) is performed by evaluating the convergence error *change* for the entry condition of the next iteration.

In the result saving block, the historical information lists *oh* and *vh* are stored to customData (line 99) that is saved into the final design model database (line 100) named "Final_design.cae". The geometric and historical results can be manually inspected by opening this final data base in Abaqus/CAE, or can be viewed using a separate code that is introduced later in the paper.

### 3.3. Model formatting: fmtMdb (lines 5–17)

This function formats the initial model database for the soft-kill scheme. The whole model "Model-1" is stored in an Abaqus model object *mdl*. In the soft-kill scheme, the void elements are interpreted as a very soft material. Therefore two materials are defined by setting up the Young's moduli and Poisson's ratios; then these materials are linked to two sections of the design, i.e. solid section (*solidSec*) and void section (*voidSec*) respectively (lines 10–13). According to the SIMP model [3], the Young's modulus of the soft material is set to $0.001 \times 10^3$ (line 12) based on $x_{\min} = 0.001$ and $p = 3$. Then the solid section is assigned to an element set containing all the elements (line 14) as the BESO procedure will start with the initial full design. In the end, FEA outputs are requested for the element strain energies (line 16) using the Abaqus keyword "ELEDEN", and for the external work (line 17) using the keyword "ALLWK" that is equivalent to the objective function (compliance) considering no energy dissipation.

### 3.4. Finite element analysis: FEA (lines 18–27)

This function runs the finite element analysis in the iteration and outputs the element sensitivities *Ae* in the argument list and the compliance as the function return value. The program submits the analysis job to Abaqus (line 20) and waits till the analysis is finished (line 21). An output database (.odb) is automatically produced by Abaqus after the analysis and is opened by the program (line 22). The element elastic strain energies are retrieved (line 23) according to the odb object model in Fig. 3 and using the Abaqus keyword "ESEDEN". Then the element sensitivities are obtained by modifying the element strain energies (line 24) according to Eq. (6). Note that for the computational simplicity *Ae* is not multiplied with the penalty exponent *p* in the implementation since this will not affect the optimization solution [9]. Finally the objective function compliance is retrieved (line 25) using the keyword "ALLWK" and returned by the function.

### 3.5. Mesh-independent filtering: preFlt (lines 28–44) and fltAe (lines 45–50)

The function *preFlt* prepares the filter scheme and is called only once in the program to save the computational cost. For each element *e*, this function finds out the surrounding elements *j* within the sphere centred at the centroid of the element *e* with the radius $r_{\min}$; and then calculates the weight factor $\eta_j$ according to Eqs. (7) and (8). For each element *e*, the labels of elements *j* and the weight factor $\eta_j$ are stored as lists; these two lists map with the label of element *e* and form a pair in a dictionary that contains all such pairs over the design domain. This dictionary is saved and later used by *fltAe* in the iteration. In the implementation, the centre coordinates of each element is first calculated and stored in a dictionary *c0* (lines 31–34); then in a double-loop, each over all the elements, the distances between element centres and the relevant weight factors are calculated and stored in the mapping dictionary *Fm* (lines 36–44).

The function *fltAe* applies the filter scheme to the element sensitivities according to Eq. (7). In a loop over all elements, the modified sensitivities are obtained (lines 48–50) by multiplying $\eta_j$ with the raw sensitivities (line 47) based on the filter mapping dictionary *Fm*.

### 3.6. Element updating: BESO (lines 51–67)

This function implements the BESO optimizer and is the core of the Python code. This function first determines the sensitivity threshold *th* (lines 53–59) following the bisection algorithm shown in Fig. 1; during this procedure the design variables *Xe* are also set by assigning 1.0 to solid elements and 0.001 to void elements. According to the design variables, the labels of all elements are grouped into two lists: *slb* for solid elements and *vlb* for void elements (lines 61–64). Then the solid and void sections are assigned to the two element sets that are created using the lists *slb* and *vlb* respectively (lines 66–67).

It should be noted that the above BESO optimizer can be replaced by another 0–1 optimizer for test and comparison purposes. The replacement can be conveniently done, with lines 60–67 remaining in the current form.

## 4. Extensions to practical applications

This Python code is well-structured and sets up a framework for general topology optimization using BESO. Therefore, it can be easily altered to achieve other objectives with only limited modifications to the current form. A few modifications are introduced in the following as examples.

### 4.1. Increasing computational efficiency

The basic arithmetic functions of the Python language are not optimized and may therefore lead to poor performance when a large amount of computation is involved. A few separate Python libraries such as NumPy (www.numpy.org) and SciPy (www.scipy.org) optimized for scientific computation have been developed and widely used by the Python community.

The current Python code in its basic form performs well on small FE models such as a 2D model; however when dealing with large 3D models with a large number of elements, a bottleneck may occur in *preFlt* due to the large amount of arithmetic calculations in the loops. In this case, the NumPy library can be employed to significantly improve the computation speed. Note that NumPy is neither a standard component of the Python language nor included in the original Abaqus distribution; therefore the reader needs to additionally install the NumPy package (freely downloadable from www.numpy.org) for the Python version used in his/her Abaqus software. It is for this reason that the implementation involving NumPy is presented as an extension rather than in the basic code.

In this extension, the affected code is only in *preFlt*; Appendix B gives the full source code of this modified function with which the user can easily replace the basic version. NumPy is first imported and renamed as *np* for programming simplicity. Several NumPy functions are used to replace the fundamental versions, such as *np.add* replaces +, *np.subtract* replaces −, *np.divide* replaces /, *np.square* replaces *sqrt*, and *np.power* replaces *pow*; besides, *np.zero* is used to initialise a list with all zero entries and *np.sum* is used to get the summation of all entries inside a list.

NumPy can also be implemented in other functions of the Python code; the authors decide to leave this to interested readers as a programming practice.

### 4.2. Multiple load cases

Multiple load cases can be easily addressed by extending the basic Python code, given that the input model defines multiple load cases. For convenient reading of FEA results, the multiple analysis steps can be defined in the input model and each step corresponds to one load case. This extension requires modifications on the functions *fmtMdb* and *FEA*.

For *fmtMdb*, the modification takes place for the block of defining output requests. The program loops through all analysis steps requesting element energy densities and external work. This is realized by replacing lines 16 and 17 by the following loop.

```
016    for stp in [k for k in mdl.steps.keys() if k != 'Initial']:
         mdl.FieldOutputRequest('SEDensity'+stp,stp,
       variables=('ELEDEN',))
         mdl.HistoryOutputRequest('ExtWork'+stp,stp,
       variables=('ALLWK',))
```

Accordingly for *FEA*, the modification takes place for the block of reading the analysis outputs. Using a loop, the element sensitivities and the objective function are summed up over all analysis steps; in other words, the weight factors for all steps are identical and the objective function is defined as follows.

$$obj = \sum_i C_i \tag{11}$$

where $C_i$ is the compliance of the *i*th load case. The Python implementation is realized by replacing lines 23–25 with the following block.

```
023  obj = 0
       for k in Xe.keys(): Ae[k] = 0.0
       for stp in opdb.steps.values():
         seng = stp.frames[-1].fieldOutputs['ESEDEN'].values
         for en in seng: Ae[en.elementLabel]+=en.data/
     Xe[en.elementLabel]
         obj += stp.historyRegions['Assembly
     ASSEMBLY'].historyOutputs['ALLWK'].data[-1][1]
```

An example is given in Fig. 4 where the 3D cantilever has the identical geometry as that in Fig. 2 but includes an additional load case. A filter radius of 4 and target final volume fraction of 10% are used in this example.

### 4.3. Geometric restrictions

Geometric restrictions such as a design domain of specific shape and non-designable areas can be readily taken into account by the proposed Python implementation. Using the graphical interface Abaqus/CAE, arbitrary shaped geometries can be created and then directly taken as the design domain. It is noted that the previous Matlab implementations (e.g. [19,10,1,15]) are based on regular design domains of 2D square or 3D cube shapes with the concept of "passive elements" to create initial cavities, which is indirectly realized by changing the source codes.

Additionally, non-designable areas are solid and allow no cavities to be created inside. Since the current Python code takes only the part "Part-1" in model "Model-1" in the model database as the design domain, any other parts are automatically regarded non-designable. Therefore, a non-designable domain can be easily realized by defining an addition part besides "Part-1".

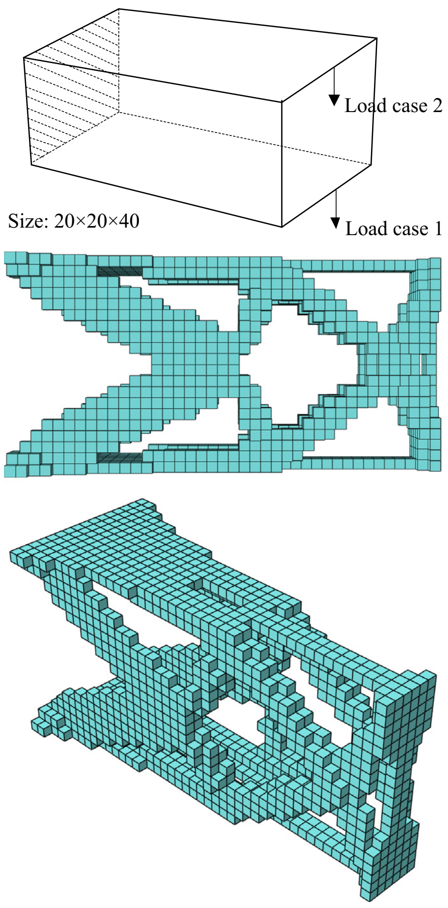

**Fig. 4.** Topology optimization of a 3D cantilever with two load cases: (top) the initial full design as the design domain; (middle and bottom) side and perspective views of the optimized solution.

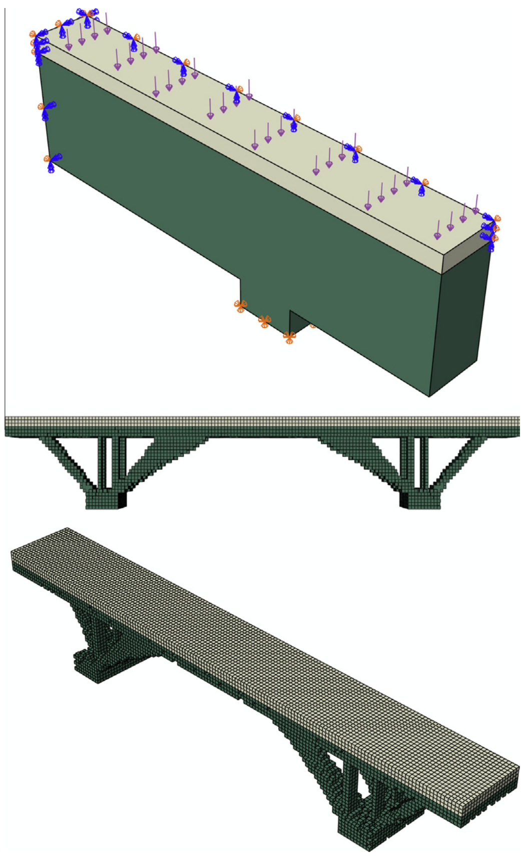

**Fig. 5.** Topology optimization with initial voids and a non-designable part: (top) the design domain (one quarter model); (middle and bottom) side and perspective views of the optimized solution of the full bridge.

Fig. 5 shows an example with both initial voids and a non-designable layer. Due to symmetry, only one quarter of the full structure is considered in Abaqus/CAE. The designable "Part-1" in green colour[1] is attached to the non-designable "Part-2" in white colour using a tie-constraint. Two voids are cut off at the bottom of "Part-1" to create an extrusive pier that is fixed at the base. A uniform pressure is applied on the top surface of "Part-2". A filter radius of 4 times the element size and a target final volume fraction of 20% are used in this example.

Fig. 6 shows an example of a gear design with a circular shaped design domain and non-designable inner- and outer-rings; the designable "Part-1" in green colour[1] is attached to the non-designable "Part-2" in white colour using a tie-constraint. Three load cases are applied as surface tractions, each being two concentrated forces of 100 N on the outer-ring. The inner ring is fixed on the internal surface. A filter radius of 2 and a target final volume fraction of 20% are used.

### 4.4. Optimization considering nonlinearities

According to literature [4,8], the element sensitivity of nonlinear compliance minimization can be obtained as the sum of the total elastic strain energy $E^e$ and plastic strain energy $E^p$ in the element as follows.

$$\alpha_e = E_e^e + E_e^p \tag{12}$$

According to the above sensitivity calculation, the compliance minimization with both material and geometrical nonlinearities can be realized by simply requesting the total elemental energy densities (modify *fmtMdb* using Abaqus keyword "ENER") and reading from odb the elastic and plastic strain energies (modify *FEA* using keywords "SENER" and "PENER" respectively).

More simply, geometrical nonlinearity is readily activated without any change to the code, as long as a standard static analysis step in the input model is defined with the option of geometrical nonlinearity using Abaqus/CAE. Nevertheless, a nonlinear analysis may not be able to converge due to reasons such as inappropriate load increments. In order to protect the optimization procedure from such unexpected errors, the Python exception handling mechanism may be implemented in the function *FEA* by replacing lines 20 and 21 with the following block.

```
020   try:
        Mdb.Job('Design_Job'+str(Iter),'Model-1').submit()
        Mdb.jobs['Design_Job'+str(Iter)].waitForCompletion()
      except AbaqusException, message:
        print "Error occured:", message
        sys.exit(1)
```

Fig. 7 shows the topology optimization of a double-clamped beam with a filter radius of 2 and a final target volume fraction of 30% of the design domain. Both a linear solution and a

[1] For interpretation of colour in Figs. 5 and 6, the reader is referred to the web version of this article.

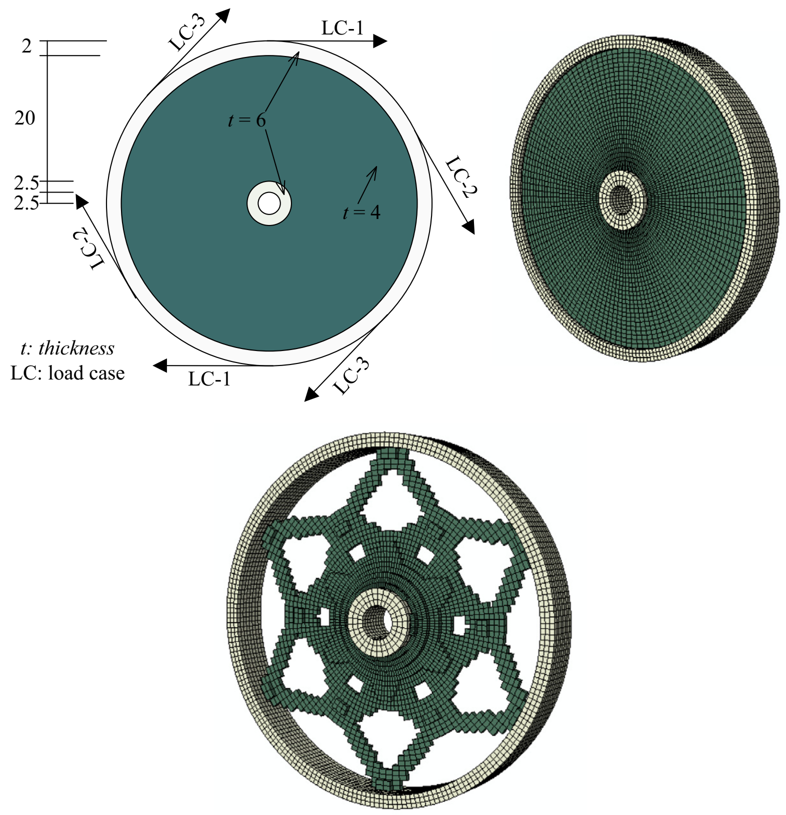

**Fig. 6.** Topology optimization with geometric restrictions: (top) the design domain and boundary conditions; (bottom) a perspective view of the optimized solution of the gear.

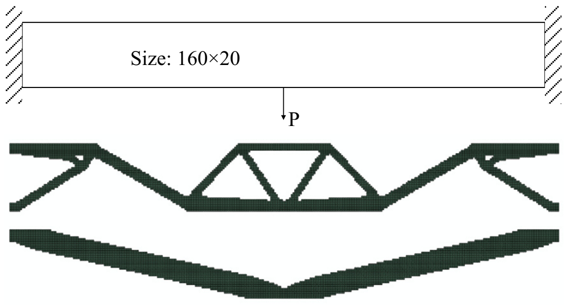

**Fig. 7.** Topology optimization of a double clamped beam: (top) the initial full design as the design domain; (middle) linear solution; (bottom) geometrically nonlinear solution.

geometrically nonlinear solution are presented for comparison. These results are similar to those in the literature (e.g. [8]).

### 4.5. Result presentation

This is a separate Python code that is written for automatic result presentation. It consists of three blocks: plotting the historical information of the volume fraction, plotting the historical information of the objective function, and displaying the final geometry. The complete code is presented in Appendix C. Although this extension is useful to some researchers and students, it is not directly related to topology optimization. Therefore the authors decide not to explain this part of the source code in detail.

## 5. Conclusions

This paper presents a simple Python code for topology optimization of general 2D and 3D structures. The compact 100-lined code is developed for the compliance minimization with a volume constraint. With simple modifications, the basic Python code has been extended to enhance computational efficiency and to consider multiple load cases and nonlinearities. More importantly, the code provides a convenient platform upon which further extensions such as new functions and different optimizers could be easily built. In doing so, the user may experiment with various algorithms and tackle a wide range of problems. The Python code is presented for educational and engineering practice purposes. Moreover, with extensions such as those presented in this paper and with the Abaqus FEA/modelling power, the Python code is capable of solving further engineering design problems using topology optimization by generating complex conceptual designs. The complete codes are given in the appendix section for users to copy and save as .py files to run with Abaqus/CAE. These codes may also be downloaded from the website: www.rmit.edu.au/research/cism/.

## Appendix A. Basic Python code

```
“““General 3D topology optimization code based on BESO by Zhi Hao Zuo and Yi Min Xie. Note that the CAE file shall contain a model ‘Model-1’ with a dependent part ‘Part-1’ and a static step ’Step-1’.”””
import math,customKernel
from abaqus import getInput,getInputs
from odbAccess import openOdb
## Function of formatting Abaqus model for stiffness optimization
def fmtMdb(Mdb):
    mdl = Mdb.models[’Model-1’]
    part = mdl.parts[’Part-1’]
    # Build sections and assign solid section
    mdl.Material(’Material01’).Elastic(((1.0, 0.3),))
    mdl.HomogeneousSolidSection(’sldSec’,’Material01’)
    mdl.Material(’Material02’).Elastic(((0.001**3, 0.3),))
    mdl.HomogeneousSolidSection(’voidSec’,’Material02’)
    part.SectionAssignment(part.Set(’ss’,part.elements),’sldSec’)
    # Define output request
    mdl.FieldOutputRequest(’SEDensity’,’Step-1’,variables=(’ELEDEN’,))
    mdl.HistoryOutputRequest(’ExtWork’,’Step-1’,variables=(’ALLWK’,))
## Function of running FEA for raw sensitivities and objective function
def FEA(Iter,Mdb,Xe,Ae):
    Mdb.Job(’Design_Job’+str(Iter),’Model-1’).submit()
    Mdb.jobs[’Design_Job’+str(Iter)].waitForCompletion()
    opdb = openOdb(’Design_Job’+str(Iter)+’.odb’)
    seng = opdb.steps[’Step-1’].frames[-1].fieldOutputs[’ESEDEN’].values
    for en in seng: Ae[en.elementLabel]=en.data/Xe[en.elementLabel]
    obj=opdb.steps[’Step-1’].historyRegions[’Assembly ASSEMBLY’].historyOutputs[’ALLWK’].data[-1][1]
    opdb.close()
    return obj
## Function of preparing filter map (Fm={elm1:[[el1,el2,...],[wf1,wf2,...]],...})
def preFlt(Rmin,Elmts,Nds,Fm):
    # Calculate element centre coordinates
    c0 = {}
    for el in Elmts:
        nds = el.connectivity
        c0[el.label]=[sum([Nds[nd].coordinates[i]/len(nds) for nd in nds]) for i in range(3)]
    # Weighting factors
    for el in Elmts:
        Fm[el.label] = [[],[]]
        for em in Elmts:
            dis=math.sqrt(sum([(c0[el.label][i]-c0[em.label][i])**2 for i in range(3)]))
            if dis<Rmin:
                Fm[el.label][0].append(em.label)
                Fm[el.label][1].append(Rmin - dis)
        sm = sum(Fm[el.label][1])
        for i in range(len(Fm[el.label][0])): Fm[el.label][1][i] /= sm
## Function of filtering sensitivities
def fltAe(Ae,Fm):
    raw = Ae.copy()
    for el in Fm.keys():
        Ae[el] = 0.0
        for i in range(len(Fm[el][0])): Ae[el]+=raw[Fm[el][0][i]]*Fm[el][1][i]
## Function of optimality update for design variables and Abaqus model
def BESO(Vf,Xe,Ae,Part,Elmts):
    lo, hi = min(Ae.values()), max(Ae.values())
    tv = Vf*len(Elmts)
    while (hi-lo)/hi > 1.0e-5:
        th = (lo+hi)/2.0
        for key in Xe.keys(): Xe[key] = 1.0 if Ae[key]>th else 0.001
        if sum(Xe.values())-tv>0: lo = th
        else: hi = th
    # Label elements as solid or void
```

```
061     vlb, slb = [], []
062     for el in Elmts:
063         if Xe[el.label] == 1.0: slb.append(el.label)
064         else: vlb.append(el.label)
065     # Assign solid and void elements to each section
066     Part.SectionAssignment(Part.SetFromElementLabels('ss',slb),'sldSec')
067     Part.SectionAssignment(Part.SetFromElementLabels('vs',vlb),'voidSec')
068 ## ====== MAIN PROGRAM ======
069 if __name__ == '__main__':
070     # Set parameters and inputs
071     pars = (('VolFrac:','0.5'), ('Rmin:', '1'), ('ER:', '0.02'))
072     vf,rmin,ert = [float(k) if k!=None else 0 for k in getInputs(pars,dialogTitle='Parameters')]
073     if vf<=0 or rmin<0 or ert<=0: sys.exit()
074     mddb = openMdb(getInput('Input CAE file:',default='Test.cae'))
075     # Design initialization
076     fmtMdb(mddb)
077     part = mddb.models['Model-1'].parts['Part-1']
078     elmts, nds = part.elements, part.nodes
079     oh, vh = [], []
080     xe, ae, oae, fm = {}, {}, {}, {}
081     for el in elmts: xe[el.label] = 1.0
082     if rmin>0: preFlt(rmin,elmts,nds,fm)
083     # Optimization iteration
084     change, iter, obj = 1, -1, 0
085     while change > 0.001:
086         iter += 1
087         # Run FEA
088         oh.append(FEA(iter,mddb,xe,ae))
089         # Process sensitivities
090         if rmin>0: fltAe(ae,fm)
091         if iter > 0: ae=dict([(k,(ae[k]+oae[k])/2.0) for k in ae.keys()])
092         oae = ae.copy()
093         # BESO optimization
094         vh.append(sum(xe.values())/len(xe))
095         nv = max(vf,vh[-1]*(1.0-ert))
096         BESO(nv,xe,ae,part,elmts)
097         if iter>10: change=math.fabs((sum(oh[iter-4:iter+1])-sum(oh[iter-9:iter-4]))/sum(oh[iter-9:iter-4]))
098     # Save results
099     mddb.customData.History = {'vol':vh,'obj':oh}
100     mddb.saveAs('Final_design.cae')
```

## Appendix B. ***preFlt*** implementing NumPy (replaces lines 28–44 in Appendix A)

```
028 ## Function of preparing filter map (Fm = {elm1:[[el1,el2,...],[wf1,wf2,...]],...}) (NumPy)
    def preFlt(Rmin,Elmts,Nds,Fm):
        import numpy as np
        # Calculate element centre coordinates
        elm, c0 = np.zeros(len(Elmts)), np.zeros((len(Elmts),3))
        for i in range(len(elm)):
            elm[i] = Elmts[i].label
            nds = Elmts[i].connectivity
            for nd in nds: c0[i] = np.add(c0[i],np.divide(Nds[nd].coordinates,len(nds)))
        # Weighting factors
        for i in range(len(elm)):
            Fm[elm[i]] = [[],[]]
            for j in range(len(elm)):
                dis = np.square(np.sum(np.power(np.subtract(c0[i],c0[j]),2)))
                if dis<Rmin:
                    Fm[elm[i]][0].append(elm[j])
                    Fm[elm[i]][1].append(Rmin - dis)
            Fm[elm[i]][1] = np.divide(Fm[elm[i]][1],np.sum(Fm[elm[i]][1]))
```

## Appendix C. Result presentation

```
"""This is a complete script used to automatically display results generated by the Python code in this paper."""
import customKernel
from caeModules import *
## Function of plotting optimization results
def pltRslts(mddb):
   itern = len(mddb.customData.History['obj'])
   # Plot volume fraction history
   vp1 = session.Viewport('Volume history',origin=(10,10),width=150,height=100)
   xyPlot1 = session.XYPlot('Volume fraction')
   chart1 = xyPlot1.charts.values()[0]
   volDat = [(k,mddb.customData.History['vol'][k]) for k in range(itern)]
   xydv = session.XYData('Volume fraction',volDat)
   chart1.setValues(curvesToPlot=[session.Curve(xydv)])
   chart1.axes1[0].axisData.setValues(title='Iteration')
   chart1.axes2[0].axisData.setValues(title='Volume fraction')
   vp1.setValues(displayedObject=xyPlot1)
   # Plot objective function history
   vp2 = session.Viewport('Objective history',origin=(20,20),width=150,height=100)
   xyPlot2 = session.XYPlot('Objective function')
   chart2 = xyPlot2.charts.values()[0]
   objDat = [(k,mddb.customData.History['obj'][k]) for k in range(itern)]
   xydo = session.XYData('Objective function',objDat)
   chart2.setValues(curvesToPlot=[session.Curve(xydo)])
   chart2.axes1[0].axisData.setValues(title='Iteration')
   chart2.axes2[0].axisData.setValues(title='Objective function')
   vp2.setValues(displayedObject=xyPlot2)
   # Display final design
   vp3 = session.Viewport('Final design', origin=(30,30),width=150, height=100)
   p = mddb.models['Model-1'].parts['Part-1']
   vp3.setValues(displayedObject=p)
   vp3.partDisplay.setValues(mesh=ON)
   vp3.partDisplay.displayGroup.remove(leaf=dgm.LeafFromSets(sets=(p.sets['vs'],)))
## ====== MAIN PROGRAM ======
if __name__ == '__main__':
   mddb = openMdb(getInput('Input CAE file:',default='Final_Design.cae'))
   pltRslts(mddb)
```

## References

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

[2] Bendsøe MP. Optimal shape design as a material distribution problem. Struct Optim 1989;1:193–202.

[3] Bendsøe MP, Sigmund O. Topology optimization: theory, methods and applications. Berlin: Springer-Verlag; 2003.

[4] Buhl T, Pedersen CBW, Sigmund O. Stiffness design of geometrically nonlinear structures using topology optimization. Struct Multidisc Optim 2000;19:93–104.

[5] Challis VJ. A discrete level-set topology optimization code written in Matlab. Struct Multidisc Optim 2010;41:453–64.

[6] Dassault Systemes; 2014. <http://www.3ds.com/products-services/simulia/portfolio/abaqus/overview/>.

[7] Huang X, Xie YM. Convergent and mesh-independent solutions for the bi-directional evolutionary structural optimization method. Finite Elem Anal Des 2007;43:1039–49.

[8] Huang X, Xie YM. Bidirectional evolutionary topology optimization for structures with geometrical and material nonlinearities. AIAA J 2007;45(1):308–13.

[9] Huang X, Xie YM. Bi-directional evolutionary topology optimization of continuum structures with one or multiple materials. Comput Mech 2009;43:393–401.

[10] Huang X, Xie YM. A further review of ESO type methods for topology optimization. Struct Multidisc Optim 2010;41:671–83.

[11] Huang X, Xie YM. Evolutionary topology optimization of continuum structures: methods and applications. Chichester: John Wiley & Sons; 2010.

[12] Huang X, Xie YM. Evolutionary topology optimization of continuum structures with an additional displacement constraint. Struct Multidisc Optim 2010;40:409–16.

[13] Huang X, Zhou SW, Xie YM, Li Q. Topology optimization of microstructures of cellular materials and composites for macrostructures. Comput Mater Sci 2013;67:397–407.

[14] Huang X, Zuo ZH, Xie YM. Evolutionary topological optimization of vibrating continuum structures for natural frequencies. Comput Struct 2010;88:357–64.

[15] Liu K, Tovar A. An efficient 3D topology optimization code written in Matlab. Struct Multidisc Optim (online) 2014. http://dx.doi.org/10.1007/s00158-014-1107-x.

[16] Osher SJ, Sethian JA. Fronts propagating with curvature dependent speed: algorithms based on the Hamilton-Jacobi formulation. J Comput Phys 1988;79:12–49.

[17] Radman A, Huang X, Xie YM. Maximizing stiffness of functionally graded materials with prescribed variation of thermal conductivity. Comput Mater Sci 2014;82:457–63.

[18] Sigmund O. Tailoring materials with prescribed elastic properties. Mech Mater 1995;20:351–68.

[19] Sigmund O. A 99 line topology optimization code written in Matlab. Struct Multidisc Optim 2001;21:120–7.

[20] Sigmund O, Peterson J. Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. Struct Optim 1998;16:68–75.

[21] Wang MY, Wang X, Guo D. A level set method for structural topology optimization. Comput Methods Appl Mech Eng 2003;192:227–46.

[22] Xie YM, Steven GP. A simple evolutionary procedure for structural optimization. Comput Struct 1993;49(5):885–96.

[23] Xie YM, Steven GP. Evolutionary structural optimization. London: Springer; 1997.

[24] Yang XY, Huang X, Rong JH, Xie YM. Design of 3D orthotropic materials with prescribed ratios for effective Young's moduli. Comput Mater Sci 2013;67:229–37.
[25] Zhou M, Rozvany GIN. The COC algorithm, Part II: topological, geometry and generalized shape optimization. Comp Methods Appl Mech Eng 1991;89:197–224.
[26] Zuo ZH, Huang X, Rong JH, Xie YM. Multi-scale design of composite materials and structures for maximum natural frequencies. Mater Des 2013;51:1023–34.
[27] Zuo ZH, Xie YM, Huang X. Evolutionary topology optimization of structures with multiple displacement and frequency constraints. Adv Struct Eng 2012;15(2):385–98.