Structural and Multidisciplinary Optimization (2021) 63:935–954
https://doi.org/10.1007/s00158-020-02719-9

EDUCATIONAL PAPER

A 101-line MATLAB code for topology optimization using binary variables and integer programming

Renato Picelli[1] · **Raghavendra Sivapuram**[2] · **Yi Min Xie**[3]

Received: 19 May 2020 / Revised: 5 August 2020 / Accepted: 10 August 2020 / Published online: 27 September 2020

Abstract

This paper presents a MATLAB code with the implementation of the Topology Optimization of Binary Structures (TOBS) method first published by Sivapuram and Picelli (Finite Elem Anal Des 139: pp. 49–61, 2018). The TOBS is a gradient-based topology optimization method that employs binary design variables and formal mathematical programming. Besides its educational purposes, the 101-line code is provided to show that topology optimization with integer linear programming can be efficiently carried out, contrary to the previous reports in the literature. Compliance minimization subject to a volume constraint is first solved to highlight the main features of the TOBS method. The optimization parameters are discussed. Then, volume minimization subject to a compliance constraint is solved to illustrate that the method can efficiently deal with different types of constraints. Finally, simultaneous volume and displacement constraints are investigated in order to expose the capabilities of the optimizer and to serve as a tutorial of multiple constraints. The 101-line MATLAB code and some simple enhancements are elucidated, keeping only the integer programming solver unmodified so that it can be tested and extended to other numerical examples of interest.

Keywords Topology optimization · Binary variables · Integer linear programming · Educational code

1 Introduction

Topology optimization (TO) has exceeded the level of maturity to start pushing the engineering design beyond its previous capabilities. A considerable contribution in the dissemination of the TO methods can be granted to educational papers and codes, such as the well-known 99-line work by Sigmund (2001a). Educational papers help to ease the learning curve and attract new practitioners to the field. Other methods are also documented in the form of educational papers or published compact codes (Allaire 2009; Challis 2010; Zuo and Xie 2015; Zhang et al. 2016; Ansola et al. 2018; Xia et al. 2018; Wei et al. 2018; Liang and Cheng 2020), shedding light on TO knowledge. This paper aims to advocate the use of TO using binary design variables and integer linear programming, a rare combination in the TO literature which presents potential advantages in several applications.

The idea of TO is to solve a material distribution problem via numerical optimization. When it comes to the design of elastic structures, the aim is to find optimized layouts by defining a set of binary {0, 1} design variables, where 0 means the absence of material (void) and 1 represents the location of solid material. The binary problem has been considered very hard or even impossible to be solved using formal mathematical programming (Deaton and Grandhi 2014). Hence, the idea of relaxing the binary constraint by allowing intermediate densities became very popular with the Solid Isotropic Material with Penalization (SIMP) formulation (Rozvany et al. 1992). The SIMP method usually starts with intermediate (grayscale) densities and during optimization it steers the solution to a nearly

Responsible Editor: Ming Zhou

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00158-020-02719-9) contains supplementary material, which is available to authorized users.

✉ Renato Picelli
rpicelli@usp.br

[1] Department of Mining and Petroleum Engineering, University of São Paulo, Praça Narciso de Andrade s/n, Vila Mathias, Santos - SP, 11013-560, Brazil

[2] Structural Engineering Department, University of California San Diego, 9500 Gilman Drive, La Jolla, CA, 92093, USA

[3] Center for Innovative Structures and Materials, School of Engineering, RMIT University, Melbourne, 3001, Australia

solid/void design, or actual black and white solutions with the aid of projection schemes (Guest et al. 2004).

The first attempt to carry out TO of continuum structures using binary variables was made by Xie and Steven (1993). The idea is that, starting from a full solid design, inefficient material can be systematically removed from the structure using sensitivity information. This method was called Evolutionary Structural Optimization (ESO). The ESO method was based only on the removal of material. A few attempts were made to develop a bi-directional version of the ESO method (BESO), in which material could also be added to the structure (Querin et al. 1998). The BESO method became what it is today after convergent and mesh-independent solutions were presented by Huang and Xie (2007). In the standard BESO algorithm, the structure is usually (but not necessarily) first considered as a full solid design and a target volume is used to quantify the amount of removed/added material until convergence, only allowing $\{0, 1\}$ variables. This volume-based update scheme, however, implies that a volume constraint should be always present in the optimization formulation. This precludes problems such as mass minimization or multiple constraints to be solved in a schematic way. Therefore, how to include formal mathematical programming in the binary TO framework is still a research topic.

Beckers (1999) first introduced formal mathematical programming to solve a structural topology optimization problem with binary $\{0, 1\}$ variables. Beckers' method was based on sequential approximate programming and a Lagrangian dual-method optimizer. A perimeter constraint was used to make the problem well-posed and the use of sensitivity filtering was also investigated, although it was reported to be expensive back at that time. Beckers' solutions were convergent and mesh independent, showing examples with up to 30,000 elements. Svanberg and Werme (2005) found that when only one element is completely removed from the structure, the new global stiffness matrix is a low-rank modification of the old one, which validates the use of discrete sensitivities. Relying on that, Svanberg and Werme (2006) introduced integer linear programming to solve a sequentially approximate problem with binary variables. However, their method needs to hierarchically refine the finite element mesh, so only a few layers of material are removed at each step, leading to solutions that are highly dependent on the mesh used. Beckers (1999) and Svanberg and Werme (2006) were great contributors to the idea of doing TO with binary $\{0, 1\}$ variables with mathematical programming, being the topic rarely explored since then. To the best of our knowledge, we presented the first convergent and mesh-independent solutions for TO with $\{0, 1\}$ variables and integer programming in Sivapuram and Picelli (2018). The method so-called Topology Optimization of Binary Structures (TOBS) combines four numerical ingredients: sequential problem linearization, constraints' relaxation, sensitivity filtering, and an integer programming solver. These well-known numerical ingredients build up a more general scheme for binary TO than Beckers (1999) and Svanberg and Werme (2006). The TOBS method was extended to 3D microstructural optimization problems with multiple (up to 6) non-volume constraints by Sivapuram et al. (2018). One year later, Liang and Cheng (2019) proposed a method for binary TO using the Canonical Duality Theory (CDT) originally developed by Gao and Zhang (2010) to solve the integer problem. In contrast, the TOBS method uses a branch-and-bound solver (Williams 2009). Liang and Cheng (2020) recently published further elaborations on their work, including a compact code with their implementation. It is also important to mention that Liang and Cheng (2020) present the mathematical validity for using discrete sensitivities when turning a solid element into void. In comparison with TOBS, the CDT-based method involves some assumptions which make the topologies obtained at each iteration not perfectly discrete, i.e., some elements can have intermediate densities or even densities greater than unity. This might make their method retain the disadvantages of SIMP-based methods for e.g. design-dependent load problems. On the other hand, the CDT-based solver seems to be as fast as MMA and other solvers in SIMP, while commercial branch-and-bound solvers as used in TOBS are slower. The control of constraints via move limits in the TOBS method is not present in Liang and Cheng (2020). For further comparisons with Liang and Cheng (2020) and other methods, including direct numerical results, the reader is referred to Picelli and Sivapuram (2019). Sivapuram and Picelli (2020) extended TOBS to design topologies in the presence of design-dependent pressure loads and thermal loads acting simultaneously. Sivapuram and Picelli (2020) also presented a computational time study that shows the time needed to solve the integer programming problems via a branch-and-bound algorithm increases much slowly with the mesh size as compared to the time needed by the FEA solver. Therefore, FEA is the main computational bottleneck in topology optimization using TOBS.

One advantage of using binary $\{0, 1\}$ variables in TO relies on having explicitly defined structural boundaries. In contrast with methods with intermediate densities, the location of the boundaries at each step of the optimization is directly defined by the $\{0, 1\}$ configuration. This is advantageous in problems where the structure interacts with other physics via its boundaries, such as acoustic-structure (Vicente et al. 2015) and fluid-structure interaction (Picelli et al. 2020). In such cases, interface conditions can be modeled straightforwardly. Another advantage can be in avoiding issues due to numerical interpolation. Although a

material model (e.g., SIMP) can be used in TOBS to aid the sensitivity analysis, the finite element model is only built with the $\{0, 1\}$ variables by which interpolation does not change the physics of the problem. This might be important when dealing with problems involving design-dependent body loads, such as thermoelastic design (Sivapuram and Picelli 2020), or in problems with complex numerical analysis such as fluid flow optimization (Yoon 2020). The Level Set Topology Optimization (LSTO) represents another class of methods that inherently produce crisply defined structural boundaries (Emmendoerfer et al. 2018; Picelli et al. 2019). However, the boundaries still require extra modelling techniques when extending LSTO to design-dependent physics problems as they are approximate and interpolated from a rational level-set function (Feppon et al. 2019; Neofytou et al. 2020). Besides, LSTO is generally more complex than density-based and binary methods.

The 101-line code provided in this paper borrows the efficient finite element and sensitivity analysis implementation from Andreassen et al. (2011), who provided a modification of the original 99-line code by Sigmund (2001a). This paper aims to give in detail all the steps of the TOBS method, both mathematically and numerically. Although one of the main advantages in using the binary framework from TOBS is in handling design-dependent and multiphysics loads, the present work focuses on the compliance and volume minimization problems with fixed point loads in single and multiple constraints examples for educational purposes. For an implementation of hydrostatic pressure loads, the reader is referred to www.github.com/renatopicelli/tobs. The present 101-line code uses the built-in MATLAB function `intlinprog` for solving the integer programming subproblems. This solver is chosen so the reader can run the code without installing external libraries. However, an example of a more efficient commercial solver implementation is also suggested. The remainder of the paper is organized as follows. Section 2 describes the mathematical background of the TOBS method. Section 3 presents the MATLAB implementation in detail. Section 4 presents numerical results and discussions and Section 5 concludes the manuscript.

2 Topology Optmization framework

The TOBS method is formulated in order to align the use of binary design variables with formal mathematical programming. The following sections describe the theoretical background of the topology optimization with integer programming.

2.1 The TOBS method

A generic binary optimization problem with inequality constraints is given by:

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{Minimize}} \quad & f(\boldsymbol{x}), \\ \text{Subject to} \quad & g_i(\boldsymbol{x}) \leq \overline{g}_i, \ i \in [1, N_g], \\ & x_j \in \{0, 1\}, \ j \in [1, N_d], \end{aligned} \tag{1}$$

where $\boldsymbol{x}$ is the vector of design variables (densities in case of topology optimization) of size N_d, f is the objective function, g_i is the i^{th} inequality constraint, $\overline{g}_i$ is the associated upper bound, and N_g is the number of inequality constraints in the optimization problem.

Since general topology optimization problems are highly non-linear and non-convex, the TOBS method generates approximate integer linear suboptimization problems sequentially and solves the generated integer linear programs. Using Taylor's series approximation, the objective and constraint functions can be written as:

$$\begin{aligned} f(\boldsymbol{x}) &= f(\boldsymbol{x}^k) + \frac{\partial f(\boldsymbol{x}^k)}{\partial \boldsymbol{x}} \cdot \Delta \boldsymbol{x}^k + O(\left|\left|\Delta \boldsymbol{x}^k\right|\right|_2^2), \\ g_i(\boldsymbol{x}) &= g_i(\boldsymbol{x}^k) + \frac{\partial g_i(\boldsymbol{x}^k)}{\partial \boldsymbol{x}} \cdot \Delta \boldsymbol{x}^k + O(\left|\left|\Delta \boldsymbol{x}^k\right|\right|_2^2), \end{aligned} \tag{2}$$

where $(\cdot)^k$ indicates the value of quantity $(\cdot)$ at the iteration k and $O(\left|\left|\Delta \boldsymbol{x}^k\right|\right|_2^2)$ corresponds to superlinear terms. TOBS employs truncated linear approximations of the objective and constraint functions:

$$\begin{aligned} f(\boldsymbol{x}) &\approx f(\boldsymbol{x}^k) + \frac{\partial f(\boldsymbol{x}^k)}{\partial \boldsymbol{x}} \cdot \Delta \boldsymbol{x}^k, \\ g_i(\boldsymbol{x}) &\approx g_i(\boldsymbol{x}^k) + \frac{\partial g_i(\boldsymbol{x}^k)}{\partial \boldsymbol{x}} \cdot \Delta \boldsymbol{x}^k, \end{aligned} \tag{3}$$

with the truncation error being $O(\left|\left|\Delta \boldsymbol{x}^k\right|\right|_2^2)$. One can use $\Delta \boldsymbol{x}^k$ as the vector of change in design variables in order to solve the optimization problem in (1). For instance, in structural topology optimization, $x_j = 1$ represents a solid element. In this case, one can choose $\Delta x_j \in \{-1, 0\}$ to prescribe that the element j either turns void ($x_j = 0$) or remains solid. The same goes for void elements: one can choose $\Delta x_j \in \{0, 1\}$ prescribing that the element j either turns solid or remains void after solving the integer linear subproblem. This guarantees that the structural topology is clear ($\{0, 1\}$) at any given iteration k. The set Δx_j is fed to the optimizer as a bound constraint, e.g.:

$$\begin{cases} 0 \leq \Delta x_j^k \leq 1 & \text{if } x_j^k = 0, \\ -1 \leq \Delta x_j^k \leq 0 & \text{if } x_j^k = 1, \end{cases} \tag{4}$$

or, in the unified form,

$$\Delta x_j^k \in \{-x_j^k, 1 - x_j^k\}. \tag{5}$$

Thus, the optimizer picks optimal Δx_j^k while satisfying the problem constraints and also satisfying integer-only constraints.

In order to maintain the linear approximation (3) valid, the truncation error $O(\left|\left|\Delta \boldsymbol{x}^k\right|\right|_2^2)$ cannot be large. The truncation error is controlled by adding an extra constraint that restricts the number of flips $\Delta \boldsymbol{x}^k$ from 1 to 0 and vice versa. This constraint can be written as:

$$\left|\left|\Delta \boldsymbol{x}^k\right|\right|_1 \leq \beta N_d. \tag{6}$$

For topology optimization, this means that the number of elements evolving from solid to void and vice versa in each iteration is restrained to a β fraction of the total number of design variables N_d. Using small values of the control parameter β ensure that the number of flips remains low at each iteration k, thereby keeping the truncation error small.

Using the sequential linear approximations from (3) in the original optimization problem (1) and the extra constraints from (5) and (6), one can write the approximate integer linear subproblem as:

$$\begin{aligned}
\underset{\Delta \boldsymbol{x}^k}{\text{Minimize}} \quad & \frac{\partial f(\boldsymbol{x}^k)}{\partial \boldsymbol{x}} \cdot \Delta \boldsymbol{x}^k, \\
\text{Subject to} \quad & \frac{\partial g_i(\boldsymbol{x}^k)}{\partial \boldsymbol{x}} \cdot \Delta \boldsymbol{x}^k \leq \overline{g}_i - g_i\left(\boldsymbol{x}^k\right) := \Delta g_i^k, \ i \in [1, N_g], \\
& \left|\left|\Delta \boldsymbol{x}^k\right|\right|_1 \leq \beta N_d, \\
& \Delta x_j^k \in \{-x_j^k, 1 - x_j^k\}, \ j \in [1, N_d].
\end{aligned} \tag{7}$$

Equation (7) expresses the sequential suboptimization problems in the standard form solved by TOBS. The term $f(\boldsymbol{x}^k)$ from the linearization (3) is dropped out in (7) since scalar addition to the objective function does not alter the optimum design variables. The same is not valid for the constraints and $g_i(\boldsymbol{x}^k)$ is used to compute the right-hand side of the constraint Δg_i^k. The truncation error constraint (6) restrains the topology from undergoing great changes. This might lead to the infeasibility of some of the constraints g_i in the current iteration k when the bound $\Delta g_i^k = \overline{g}_i - g_i\left(\boldsymbol{x}^k\right)$ is used. This can be avoided by modifying the upper bounds of constraints Δg_i^k so that the suboptimization problems yield feasible solutions. This also helps in generating feasible subproblems when the initial solution given to the optimizer is far from being feasible, for instance, starting with a fully solid design domain and having a small volume constraint in the problem. The constraint bounds are modified using:

$$\Delta g_i^k = \begin{cases} -\epsilon_i g_i\left(\boldsymbol{x}^k\right) & : \overline{g}_i < (1 - \epsilon_i) g_i\left(\boldsymbol{x}^k\right), \\ \overline{g}_i - g_i\left(\boldsymbol{x}^k\right) & : \overline{g}_i \in [(1 - \epsilon_i) g_i\left(\boldsymbol{x}^k\right), (1 + \epsilon_i) g_i\left(\boldsymbol{x}^k\right)], \\ \epsilon_i g_i\left(\boldsymbol{x}^k\right) & : \overline{g}_i > (1 + \epsilon_i) g_i\left(\boldsymbol{x}^k\right), \end{cases} \tag{8}$$

where ϵ_i is the relaxation parameter corresponding to constraint g_i. These parameters are selected such that the suboptimization problems obtained through linearization yield feasible solutions. The modifications are made such that after the linearized subproblem is solved, the constraint value is expected to remain close enough to the constraint value before solving the subproblem. For instance, when $g_i\left(\boldsymbol{x}^k\right)$ is far from $\overline{g}_i$ and approaches it from above, the upper bound of the linearized constraint g_i is $-\epsilon g_i\left(\boldsymbol{x}^k\right)$. This means the change of $g_i\left(\boldsymbol{x}^k\right)$ at the iteration k is a fraction ϵ_i of $g_i\left(\boldsymbol{x}^k\right)$ and should enforce a decrease in $g_i\left(\boldsymbol{x}^k\right)$ so that it gradually comes close to $\overline{g}_i$. This will be demonstrated and further discussed with numerical examples.

The integer suboptimization problems generated using sequential linearization (7) can be solved using Integer Linear Programming (ILP). An ILP problem is the same as a Linear Programming (LP) problem with additional constraints that the design variables can only have integer solutions. This leads to ILP-based solutions being suboptimal with respect to the LP-based solutions. Nevertheless, topology optimization requires a $\{0, 1\}$ solution and so integer programming is a sensible option. In this work, the ILP problem is solved using the branch-and-bound algorithm implemented in the MATLAB built-in `intlinprog` function, which solves mixed integer linear problems. The branch-and-bound method is an algorithm based on the heap data structure. The ILP is first solved without any integer constraints using some linear optimization techniques, e.g., Simplex method. Then branches of LPs are created with additional inequality constraints on the design variables, with a motive that integer solutions are finally yielded, as illustrated in Fig. 1. The integer solutions of subproblems $\boldsymbol{R}_n$ are stored for further evaluation of the optimal point; therefore, recursive investigation of the optimization tree can be carried out (Land and Doig 1960; Vanderbei 2014). In the TOBS method, we use the ILP solver as a black box with some selected options for better performance.

2.2 Topology optimization: formulation

The minimum compliance problem subject to a volume constraint is used to illustrate structural topology optimization. The corresponding optimization problem P1 is formulated

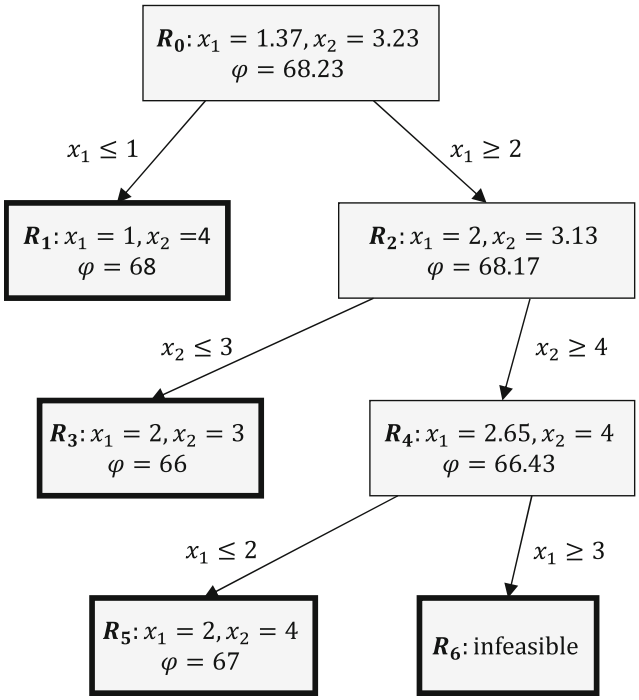


Fig. 1 Illustration of an optimization problem with objective function φ and $\{x_1, x_2\}$ design variables solved with the branch-and-bound algorithm. Boxes with bold line indicate end nodes (also called *leaf*) of the optimization tree where no more subdivisions of the $\boldsymbol{R}_n$ problems are possible

as follows:

$$\begin{aligned} \text{P1:} \quad & \underset{\boldsymbol{x}}{\text{Minimize}} && C(\boldsymbol{x}) = \boldsymbol{u}^T \boldsymbol{K} \boldsymbol{u}, \\ & \text{Subject to} && \frac{V(\boldsymbol{x})}{V_0} \leq \overline{V}, \\ & && \boldsymbol{K}\boldsymbol{u} = \boldsymbol{F}, \\ & && x_j \in \{0, 1\}, \ j \in [1, N_d], \end{aligned} \tag{9}$$

where $C(\boldsymbol{x})$ is the structural compliance function, $\boldsymbol{K}$ is the global stiffness matrix, $\boldsymbol{F}$ is the vector of applied loads, $\boldsymbol{u}$ is the structural displacements field, $V(\boldsymbol{x})$ is the volume of the structure, V_0 is the volume of the initial fully solid design domain, and $\overline{V}$ is the upper bound specified on the structural volume fraction. Finite Element Analysis (FEA) is used to solve for the state variables $\boldsymbol{u}$ under the structural equilibrium condition $\boldsymbol{K}\boldsymbol{u} = \boldsymbol{F}$.

In order to illustrate the handling of different types of constraints using TOBS, a volume minimization problem subject to a compliance constraint is used. The corresponding optimization problem P2 is given as:

$$\begin{aligned} \text{P2:} \quad & \underset{\boldsymbol{x}}{\text{Minimize}} && \frac{V(\boldsymbol{x})}{V_0}, \\ & \text{Subject to} && C(\boldsymbol{x}) \leq \overline{C}, \\ & && \boldsymbol{K}\boldsymbol{u} = \boldsymbol{F}, \\ & && x_j \in \{0, 1\}, \ j \in [1, N_d], \end{aligned} \tag{10}$$

where $\overline{C}$ is the specified upper bound for compliance.

Finally, in order to illustrate the handling of multiple constraints, the problem of compliance minimization subject to volume and displacement constraints is implemented. A problem P3 is formulated as follows:

$$\begin{aligned} \text{P3:} \quad & \underset{\boldsymbol{x}}{\text{Minimize}} && C(\boldsymbol{x}) = \boldsymbol{u}^T \boldsymbol{K} \boldsymbol{u}, \\ & \text{Subject to} && \frac{V(\boldsymbol{x})}{V_0} \leq \overline{V}, \\ & && u^* \leq \bar{u}, \\ & && \boldsymbol{K}\boldsymbol{u} = \boldsymbol{F}, \\ & && x_j \in \{0, 1\}, \ j \in [1, N_d], \end{aligned} \tag{11}$$

where u^* is the displacement at a prescribed point of the structure and $\bar{u}$ is the allowed displacement of that point.

2.3 Sensitivity analysis

The TOBS method is a gradient-based optimization algorithm. Therefore, the sensitivities (derivatives) of the objective and constraint functions are required. In fact, these sensitivities are explicitly present in the linearized suboptimization problem in (7).

The compliance sensitivities with respect to the structural densities x_j can be derived using the adjoint method (Haftka and Gürdal 1991). First, an extended function is written:

$$\mathcal{L} = \boldsymbol{u}^T \boldsymbol{K} \boldsymbol{u} + \lambda^T (\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f}), \tag{12}$$

where λ is a vector of arbitrary multipliers. Deriving $\mathcal{L}$ is equivalent of deriving $C(\boldsymbol{x})$ as $\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f} = 0$. The derivative of $\mathcal{L}$ can be expressed as:

$$\frac{\partial \mathcal{L}}{\partial x_j} = 2\boldsymbol{u}^T \boldsymbol{K} \frac{\partial \boldsymbol{u}}{\partial x_j} + \boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial x_j} \boldsymbol{u} + \lambda^T \frac{\partial \boldsymbol{K}}{\partial x_j} \boldsymbol{u} + \lambda^T \boldsymbol{K} \frac{\partial \boldsymbol{u}}{\partial x_j}. \tag{13}$$

The derivative of $\boldsymbol{f}$ in this case is 0 since the load does not change with the element densities x_j. The derivatives of the state variables $\dfrac{\partial \boldsymbol{u}}{\partial x_j}$ in (13) are usually expensive to be numerically computed. Adding the equilibrium equation into (12) works as such it helps in getting rid of this term in the analytical expression. Grouping the terms depending on $\dfrac{\partial \boldsymbol{u}}{\partial x_j}$ and making them 0, one can write:

$$2\boldsymbol{u}^T \boldsymbol{K} \frac{\partial \boldsymbol{u}}{\partial x_j} + \lambda^T \boldsymbol{K} \frac{\partial \boldsymbol{u}}{\partial x_j} = 0. \tag{14}$$

In this case, we can conclude $\lambda = -2\boldsymbol{u}$ so (14) is satisfied. This substitution is possible to be carried out as the structural compliance function is self-adjoint. For more complex problems, e.g., stress, the expression in (14) might yield a system of equations required to be solved. Choosing $\lambda = -2\boldsymbol{u}$ and knowing the terms depending on $\dfrac{\partial \boldsymbol{u}}{\partial x_j}$ are now

canceled out, (13) can be rewritten as:

$$\frac{\partial \mathcal{L}}{\partial x_j} = \boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial x_j}\boldsymbol{u} - 2\boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial x_j}\boldsymbol{u}, \tag{15}$$

and, finally,

$$\frac{\partial C(\boldsymbol{x})}{\partial x_j} = \frac{\partial \mathcal{L}}{\partial x_j} = -\boldsymbol{u}^T \frac{\partial \boldsymbol{K}}{\partial x_j}\boldsymbol{u}. \tag{16}$$

In order to analytically compute the term $\frac{\partial \boldsymbol{K}}{\partial x_j}$ in (16), we interpolate the structural stiffness as:

$$\boldsymbol{K} = \sum_{j=1}^{N_d} E(x_j)\boldsymbol{k}_0, \tag{17}$$

where $\boldsymbol{k}_0$ is the element stiffness matrix of a solid element and $E(x_j)$ is an interpolation function on Young's modulus. This function can be expressed via the modified SIMP approach (Sigmund 2007) as:

$$E(x_j) = E_{\min} + x_j^p(E_0 - E_{\min}), \tag{18}$$

where E_0 is the stiffness of the solid material, $E_{\min}$ is a very small stiffness assigned to void regions in order to prevent singularities in the global stiffness matrix, and p is a positive penalization factor. Deriving (17) with respect to x_j and substituting its value in (16), the final expression for the compliance sensitivities can be obtained:

$$\frac{\partial C}{\partial x_j} = -\frac{1}{2} p x_j^{p-1}(E_0 - E_{\min})\boldsymbol{u}_j^T \boldsymbol{k}_0 \boldsymbol{u}_j, \tag{19}$$

where $\boldsymbol{u}_j$ is the vector of element displacements.

The expression in (18) can also be called material model. With that, the compliance function could be rewritten as a function of design variables:

$$C(\boldsymbol{x}) = \sum_{j=1}^{N_d} E(x_j)\boldsymbol{u}_j^T \boldsymbol{k}_0 \boldsymbol{u}_j. \tag{20}$$

Although the TOBS method is restricted to integer variables, the material model is employed to aid the derivation of sensitivities. The penalty factor however does not affect FEA computations because of the use of binary variables. When, $x_j = 1$, $E(x_j) = E_0$, and when $x_j = 0$, $E(x_j) = E_{\min}$, thus independent of penalty p. The use of binary variables naturally avoids intermediate densities that create numerical issues when dealing with more complex physics, e.g., fluid flow optimization or multiphysics interactions (Vicente et al. 2015). We advocate that any sensitivity analysis method can be used to find the required sensitivities by the TOBS method, as long as they are only evaluated at the $\{0, 1\}$ bounds. Herein, the penalty factor p does not affect the FEA but it is present in the compliance sensitivity computations in (19). In this particular case, $p \geq 2$ so the element density x_j is not canceled out in (19). It is important to point out that Liang and Cheng (2020) provide mathematical validation of discrete sensitivities when compared with sensitivities computed via SIMP-based interpolation.

Similarly to the compliance case, the adjoint problem and sensitivity of the displacement from problem P3 (11) with respect to x_j are, respectively, given by:

$$\boldsymbol{K}\boldsymbol{\lambda} = -\boldsymbol{P}, \tag{21}$$

and,

$$\frac{\partial u^*}{\partial x_j} = p x_j^{p-1}(E_0 - E_{\min})\boldsymbol{\lambda}_j^T \boldsymbol{k}_0 \boldsymbol{u}_j, \tag{22}$$

where $\boldsymbol{\lambda}$ is the vector of adjoint variables, and has the same size as that of the global displacement vector $\boldsymbol{u}$, $\boldsymbol{P}$ is the vector having only one non-zero entry 1 at the location corresponding to displacement u^*, and $\boldsymbol{\lambda}_j$ is the vector of adjoint variables corresponding to finite element j.

The sensitivities of the volume fraction function are also required to be computed. In this work, the semi-analytical method via the following finite difference operation:

$$\frac{\partial g}{\partial x_j} = \frac{g(x_j = 1) - g(x_j = 0)}{1 - 0}, \tag{23}$$

is employed, yielding:

$$\frac{\partial g}{\partial x_j} = \frac{1}{V_0}\frac{V(x_j = 1) - V(x_j = 0)}{(1 - 0)} = \frac{V_j}{V_0}, \tag{24}$$

where V_j is the volume of the finite element j.

2.4 Filtering

Numerical filtering is employed in density-based topology optimization approaches to avoid the well-known checkerboard problem and obtain mesh-independent solutions. In topology optimization with binary $\{0, 1\}$ design variables, it makes more sense to apply sensitivity filtering instead of other common types of filtering, such as density filtering. Although feature size control is not fully guaranteed, sensitivity filtering sufficiently provides some control over the length of the structural members in TOBS similarly as in other methods. Furthermore, as the density field is always $\{0, 1\}$, no projection methods are required and sensitivity filtering is practically the only filtering scheme used in the binary topology optimization.

The filtered sensitivities for an element j are obtained using a weighted average of element sensitivities over the neighborhood of the finite element j defined by a filter radius $r_{\min}$ (see Fig. 2). The filtered sensitivity field $\widetilde{\frac{\partial f}{\partial x_j}}$ is given as:

$$\widetilde{\frac{\partial f}{\partial x_j}} = \frac{1}{\sum_{m \in N_m} H_{jm}} \sum_{m \in N_m} H_{jm}\frac{\partial f}{\partial x_m}, \tag{25}$$

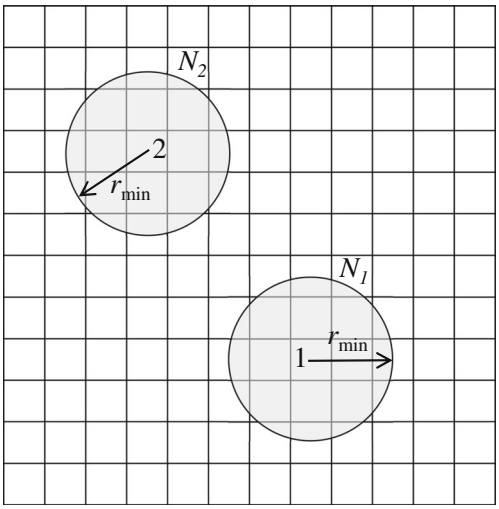


Fig. 2 Spatial filter scheme illustration

where N_m is the set of elements m for which the center-to-center distance $\texttt{dist}(\boldsymbol{x}_j, \boldsymbol{x}_m)$ from element j is smaller than the filter radius $r_{\min}$ and H_{jm} is a weight factor given by:

$$H_{jm} = \texttt{max}(0, r_{\min} - \texttt{dist}(\boldsymbol{x}_e, \boldsymbol{x}_m)) \tag{26}$$

The weights are defined such that elements closer to an element j contribute larger to the filtered sensitivities of element j compared with elements farther away from element j. The filtered sensitivities $\widetilde{\frac{\partial f}{\partial x_j}}$ and $\widetilde{\frac{\partial g}{\partial x_j}}$ are used in place of $\frac{\partial f}{\partial x_j}$ and $\frac{\partial g}{\partial x_j}$ respectively in the linearized suboptimization problem in (7).

Other types of filters can be used. One example is the BESO filter, which employs nodal averaging of the sensitivity field before using a spatial filtering to recover element sensitivity field from the nodal sensitivity field (Huang and Xie 2007). Different from the standard sensitivity filter in SIMP-based topology optimization, the element densities do not take part in the spatial filtering procedure (25) as the TOBS method uses binary densities. Filtering sensitivities plays an important role in binary topology optimization as it smoothens out the original sensitivity field and extrapolates it to void regions. This increases the chances of void elements to return to solid state, especially near highly stressed solid regions, although it might lead to inaccurate assessment of the sensitivities in the void regions. One way to mitigate this is to employ time stabilization for the sensitivity field as proposed by Huang and Xie (2007). In practice, the filtered sensitivity field is averaged over two consecutive iterations as:

$$\widetilde{\frac{\partial f}{\partial x_j}}^{k} \longleftarrow \frac{\widetilde{\frac{\partial f}{\partial x_j}}^{k} + \widetilde{\frac{\partial f}{\partial x_j}}^{k-1}}{2}. \tag{27}$$

3 MATLAB implementation

The 101-line MATLAB code with the implementation of the TOBS method is provided in the Appendix and as a supplementary material to the manuscript. The code can be called from the MATLAB command window using:

```
tobs101(nelx,nely,gbar,epsilons, ...
 beta,rmin)
```

where `nelx` and `nely` are the number of elements in the horizontal and vertical directions, respectively, `gbar` includes the prescribed constrained values, `epsilons` includes the move limits for the constraint functions, `beta` is the truncation error constraint parameter, and `rmin` is the filter radius (divided by the element size). The 101-line code consists in four parts: the FEA, the sensitivity filter matrix preparation, the optimization loop, and the TOBS solver. The FEA and the sensitivity evaluation are carried out using the implementation presented in the 88-line code by Andreassen et al. (2011). The ILP solver is the key function in the code and its details are given throughout this manuscript.

3.1 Finite element analysis

The code starts with the definition of the material properties (line 4), where `E0` is the Young modulus E_0 of the solid material, `Emin` is the artificial stiffness $E_{\min}$ assigned to void regions, and ν is the Poisson ratio. The penalty factor p for the material model is chosen same as that of standard SIMP, $\texttt{penal} = 3$.

The design domain is considered to be rectangular and discretized with unit square elements. Figure 3 illustrates an example consisting of 12 elements with four nodes per element and two degrees of freedom (DoFs) per node. Both nodes and elements are numbered column-wise from top to bottom, and left to right. The DoFs $2n-1$ and $2n$ correspond to the horizontal and vertical displacements of node n, respectively.

Lines 6–10 compute the element stiffness matrix $\boldsymbol{k}_0$ with E_0 Young's modulus. This matrix is denoted as `KE` in the code. As all elements have the same size and shape, their stiffness matrix `KE` can be computed only once. Lines 11–13 build the matrix `edofMat` whose j^{th} row contains the eight DoF numbers corresponding to the j^{th} element. This allows

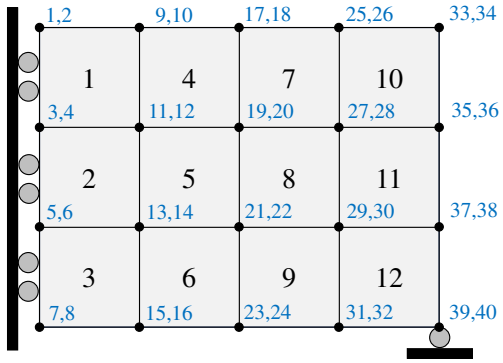


Fig. 3 Example of a 4 × 3 design domain with 12 elements and 40 DoFs

for an efficient assembly of the stiffness matrix later in the optimization loop using the MATLAB function `sparse`. Lines 14–15 prepare the row and column index vectors, `iK` and `jK` respectively to assemble stiffness matrix. The `sparse` function requires three inputs, `iK`, `jK` and the entries `sK` of the sparse stiffness matrix. These entries are computed in line 48 and depend on the design (density) variables x_j and the material model. The element stiffness matrices `KE` are reshaped to obtain column vectors and are multiplied with the corresponding Young moduli $E(x_j)$.

Lines 17–21 define and prepare the boundary conditions. The 101-line code is written to optimize the MBB beam depicted in Fig. 4. Extensions to other examples are explained in the numerical results. The point load is prescribed using the DoF along which the load is acting. Line 17 creates the force vector `F` with `2*(nely+1)*(nelx+1)` entries and sparse assembly is made with a point load at the `2*(nely+1)` DoF, the bottom left corner of the design domain, with load −1 as:

```
F = sparse(2*(nely+1),1,-1, ...
2*(nely+1)*(nelx+1),1);
```

Supports are implemented by eliminating fixed DoFs from the linear equations. The MATLAB function `setdiff` is used in line 21 to identify the free DoFs after prescribing the fixed ones, which for the MBB beam are given by:

```
fixeddofs = union([1:2:2*(nely+1)], ...
[2*(nelx+1)*(nely+1)]);
```

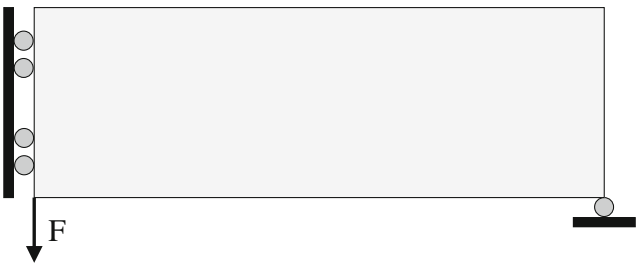


Fig. 4 MBB beam design problem

As the boundary conditions do not change in the studied cases of this work, their command lines can be placed outside the optimization loop. The structural equilibrium equation is solved to yield displacements; line 50 calls a standard MATLAB linear solver as:

```
U(freedofs) = ...
K(freedofs,freedofs)\F(freedofs);
```

where `freedofs` is the vector of unrestrained DoFs. For more details on this FEA implementation, the reader is directed to Sigmund (2001b) and Andreassen et al. (2011).

3.2 Sensitivity analysis and filtering

The sensitivity filtering in (25) is applied which is a weighted average over sensitivities of neighboring elements, and thus is a linear operator. It can be efficiently implemented as a matrix product of a coefficient matrix `H` and a vector containing the original sensitivities $\frac{\partial f}{\partial x_j}$. The matrix `H` can be assembled before the optimization loop by computing the weights H_{jm} in (26). This is done in lines 23–39. The matrix `H` is of size (`nelx` × `nely`) × (`nelx` × `nely`) containing non-zero entries only at some positions as the weights are computed only for small neighboring regions defined by $r_{\min}$. Consequently, `H` is sparse and can also be assembled using the `sparse` function. Line 40 computes the term $\sum_{m \in N_m} H_{jm}$ of the sensitivity filter which is the denominator in (25). This term is stored as `Hs` in the code and it is the sum of the filter weights which divides the weighted sum of sensitivities in order to not change the dimension of the sensitivities. After solving the equilibrium equation, the sensitivities of the structural compliance (19) and volume fraction (24) are computed in lines 52–54 and 55, respectively. The volume of each element is unity. The compliance sensitivities are filtered when multiplied by `H` and divided by `Hs`, as seen in line 57. Both original and filtered sensitivities are stored in the variable `dc`, rewriting the variable during optimization. The volume sensitivities `dv` in this case do not need to be filtered as they have a uniform distribution.

Following the sensitivity computation and filtering, time stabilization is employed in the 101-line code. This is done

in line 58 when the k^{th} iteration (variable `loop` in the code) is higher than 1. In the first iteration (`loop` $= 1$), line 59 stores the filtered sensitivities `dc` for the next iteration's averaging.

3.3 Optimization loop

The optimization loop is built similar to any standard topology optimization algorithm. It starts by computing the entries of the stiffness matrix depending on the current design variables x_j, followed by the assembly operation. After solving for the displacements, sensitivity analysis and filtering are carried out, herein followed by the sensitivity time stabilization. The objective function and constraint function histories are stored in the variables `obj` and `gi`, respectively. The ILP solver is then called to solve the integer linear suboptimization problem in (7) and update the binary design variables $\boldsymbol{x}$. The optimization loop is repeated until a convergence parameter `change` is lower than a small value, prescribed to be 10^{-4} as default in this code. The variable `change` is computed in line 65 by evaluating the change in the objective function along 10 consecutive iterations as proposed by (Huang and Xie 2007). Finally, the information about iteration, objective function, constraint function, and convergence are printed, and the current topology is plotted.

3.4 ILP function

The ILP function is the core of the present work and presents an implementation of the method described in Section 2.1. The function is called inside the optimization loop as:

```
[x] = ILP(dfdx,dgdx,gbar,gi,epsilons, ...
 beta,x)
```

where `dfdx` and `dgdx` are the objective and constraint functions sensitivities, respectively, `gbar` includes the prescribed upper bounds for the constraint functions, `epsilons` includes the constraints move limit parameters and `beta` is the truncation error constraint parameter. The size of `dfdx` must be $N_d \times 1$, while `dgdx` is of size $N_d \times N_g$. The vector sizes for `gbar` and `epsilons` are both $1 \times N_g$. Therefore, multiple constraints are included by adding N_g columns in the corresponding variables when necessary. The ILP function also receives as input the set of binary design variables `x` and outputs the updated `x`.

The truncation error is controlled via the one-norm constraint on design variable changes (phase flips). The constraint appears non-linear, but can be written in a linear fashion:

$$\sum_i \alpha_i \Delta x_i \leq \beta N_d, \tag{28}$$

where,

$$\alpha_i = \begin{cases} 1 & : x_i = 0, \\ -1 & : x_i = 1. \end{cases} \tag{29}$$

This means that when $x_j = 0$, the positive flip (addition of a solid element) counts as 1 and when $x_j = 1$, the negative flip (removal of a solid element) also counts as 1. This is already built in the ILP function. Lines 75–77 implement this constraint as:

```
C1 = (x(:) == 0);
C2 = -(x(:) == 1);
truncation = C1 + C2;
```

Lines 79–80 normalize the constraint function sensitivities. This is done by:

```
norm = max(max(abs(dgdx)), eps);
dgdx = dgdx./repmat(norm,size(dgdx,1),1);
```

Although not always necessary, normalization is a good practice to keep the sensitivities in the same order of magnitude, which helps the optimization solvers find the solutions faster. Line 82 rewrites the constraint sensitivity vector, which will now consist of the normalized constraint sensitivities and the coefficients corresponding to the truncation error constraint:

```
dgdx = [dgdx(:)'; truncation^{\prime}];
```

The next step is the relaxation of the constraint functions by imposing relaxation. The target change for the constraint function $g(\boldsymbol{x})$ to satisfy its constrained value $\overline{g}$ is evaluated as $\Delta g(\boldsymbol{x}) = \overline{g} - g(\boldsymbol{x})$, in line 84:

```
target = (gbar - gi)./norm;
```

It can be noticed that normalization must follow the constraint computation. The current target change can be too large to yield an integer solution or, sometimes, compromise the topology optimization convergence. For instance, starting with an initial full solid design, the volume fraction constraint $g(\boldsymbol{x}) = \dfrac{V(\boldsymbol{x})}{V_0} = 1.0$. For a prescribed final volume fraction $\overline{g} = \overline{V} = 0.5$, the target change is $\Delta g(\boldsymbol{x}) = -0.5$, which means that the problem requires the

optimizer to remove 50% of solid material in one iteration to find a solution. This can make the optimization infeasible because the truncation error constraint allows only small changes to the structure in each iteration. In order to avoid this issue and to enforce gradual changes in the constraint functions toward the constrained values, the target changes are multiplied by the relaxation parameters ϵ as:

```
deltag = epsilons.*(abs(gi)./norm);
```

in line 85 of the code. After that, the constraint relaxation rule from (8) can be computed by:

```
A = (target > deltag);
B = (target < -deltag);
constlimits = (A - B).*deltag + ...
(1 - (A + B)).*target;
```

which are the new upper bounds for the constraints $g_i(\boldsymbol{x})$ in the suboptimization problem (7). To evaluate and include the constraint on the truncation error (as in (6)), line 89 calls:

```
constlimits = [constlimits; ...
 beta * length(x(:))];
```

that includes the allowed limit on flips depending on `beta`.

To complete the setup of the suboptimization problem, the upper and lower bounds on Δx_j, expressed by (4), are computed in lines 91–92 of the code as:

```
lower_limits = -1.0 * (abs(x(:) - 1) ...
 < 0.001);
upper_limits = 1.0 * (abs(x(:)) ...
 < 0.001);
```

Finally, the suboptimization problem can be solved. We set the options for the `intlinprog` function of MATLAB for superior performance and better optimal solutions from the branch-and-bound solver. The branch-and-bound optimizer imposes additional constraints on the solution space called cuts to restrict the design variables to integral solutions. The `CutGeneration` option is set to `intermediate` which uses the most cut types (Cornuéjols 2008). We use the `primal-simplex` algorithm to solve the LP branches generated in each of the branches of the ILP optimizer. The number of nodes searched by the optimizer, `HeuristicsMaxNodes`, is chosen to be 100. The decision on whether a design variable enters the basis is taken by checking if the reduced cost corresponding to the design variable exceeds a tolerance. This tolerance is set in the `LPOptimalityTolerance` option to be 10^{-10}. These options are used for all the examples in this work for superior numerical performance.

```
OptimizerOptions = optimoptions( ...
 'intlinprog', 'CutGeneration', ...
 'intermediate', 'RootLPAlgorithm', ...
 'primal-simplex', 'NodeSelection', ...
 'mininfeas', 'HeuristicsMaxNodes',
 ... 100, 'RootLPMaxIter', 60000, ...
 'MaxNodes', 1e5, 'Display', 'off');
```

The optimizer is then called via MATLAB `intlinprog` function as:

```
[deltax, ~, ~, ~] = intlinprog ...
 (dfdx(:), 1:length(x(:)), dgdx, ...
 constlimits, [], [], ...
 lower_limits, upper_limits, ...
 OptimizerOptions);
```

The function `intlinprog` solves mixed-integer linear programming problems via the branch-and-bound algorithm. It can be observed that the output of the solver is a variable called `deltax`, the solution Δx_j for the suboptimization problem in (7). Finally, the code updates the set of design variables $\boldsymbol{x}$ and obtains a new binary topology via:

```
x(:) = x(:) + deltax;
```

3.4.1 Alternative optimizer implementation

The standard 101-line code calls the built-in `intlinprog` function from MATLAB so it can be used without installing extra libraries. For a more efficient and robust branch-and-bound implementation, we advise the use of CPLEX©, a proprietary optimization package from IBM. Having the library installed, the following command lines can be used to call CPLEX in the 101-line code:

```
OptimizerOptions = ...
 cplexoptimset('cplex');
 [deltax, ObjValue, exitflag, output] ...
 = cplexmilp(dfdx(:), dgdx, ...
 constlimits, [], [], [], [], [], ...
 lower_limits, upper_limits, ...
 repmat('I', 1, length(x(:))), [], ...
 OptimizerOptions);
```

These previous commands should replace lines 94–99 in the 101-code. The path to the `cplexmilp` function of CPLEX© library is to be added before using the code.

4 Results

4.1 MBB beam

The provided 101-line code is written to solve the MBB beam example (see Fig. 4), which has been extensively studied in the topology optimization literature. The minimum compliance subject to a volume constraint problem P1 in (9) is solved. By default, the 101-line code defines the material properties to be $E = 1.0$, $E_{\min} = 1 \times 10^{-9}$ and $\nu = 0.3$. Herein, we prescribe the final volume fraction to be $\overline{V} = 0.5$. The optimization parameters are $\epsilon = 0.01$ and $\beta = 0.05$ and the filter radius is $r_{\min} = 4$. The mesh used is 120×40. This problem can be solved by executing the following command:

```
tobs101(120,40,0.5,0.01,0.05,4)
```

Some intermediate topologies yielded by the 101-line code for this problem are presented in Fig. 5.

The optimization converged in 103 iterations with clear $\{0, 1\}$ topologies guaranteed by the use of strict binary variables. Convergence is smooth, as presented by the objective function plot in Fig. 6.

In order to investigate the mesh independency of the optimized solutions, the MBB problem is solved for two other meshes, 240×80 and 480×160, with totals of 19,200 and 76,800 elements, respectively. These cases are solved using the following commands:

```
tobs101(240,80,0.5,0.01,0.05,8)

tobs101(480,160,0.5,0.01,0.05,16)
```

where 8 and 16 are the filter radii $r_{\min}$ for the respective cases. The solutions for these problems are presented in Fig. 7. Such solutions resemble those from the 88-line SIMP code by Andreassen et al. (2011). In comparison, the TOBS solutions are strictly binary and crisp without the need of projection schemes but slower to be obtained. Nevertheless, a study by Sivapuram and Picelli (2020) shows that the computational bottleneck in TOBS remains to be the FEA as the mesh size increases.

4.2 Cantilever

This example solves the short cantilever beam illustrated in Fig. 8. Changing boundary conditions in the 101-line code is carried out similarly to other educational codes in topology optimization (Andreassen et al. 2011; Ansola et al. 2018).

In order to solve the cantilever beam example, line 17 in the code must be replaced by:

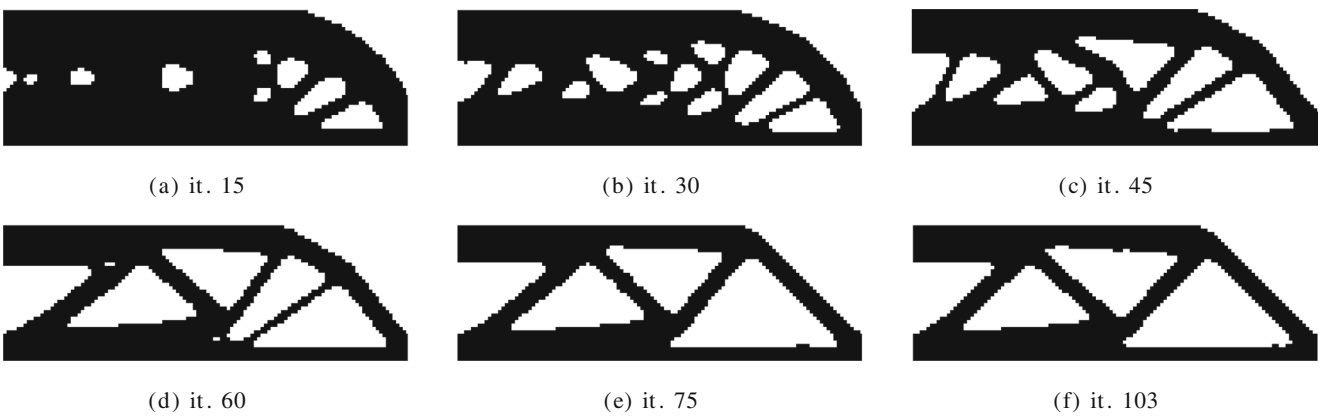
```
F = sparse(2*(nely+1)*(nelx+1),1,-1, ...
 2*(nely+1)*(nelx+1),1);
```

to prescribe the point load at the bottom right corner of the domain. Line 19 must be replaced by:

```
fixeddofs = [1:2*(nely + 1)];
```

so that it implements the cantilever support. The mesh size is set to be 160×100, the filter radius $r_{\min} = 10$ and the prescribed volume fraction $\overline{V} = 0.4$. The optimization parameters are chosen by numerical experience taking into account the nature of the constraint function. In order to illustrate the effect of the parameters, we run the cantilever design problem with $\epsilon = 0.005$, 0.01, 0.02, and 0.04. Figure 9 presents the optimized solutions obtained when each of the four ϵ's is used in topology optimization.

In the cantilever solutions in Fig. 9, the final topologies are identical for the four ϵ's considered. This might not happen for more complex problems or cases with several local minima and the choice of ϵ should be made with care. In this example, the chosen value for the constraint relaxation parameter ϵ is 0.01, for instance, implying



(a) it. 15 (b) it. 30 (c) it. 45

(d) it. 60 (e) it. 75 (f) it. 103

Fig. 5 Snapshots of the solution for the MBB beam by the 101-line code

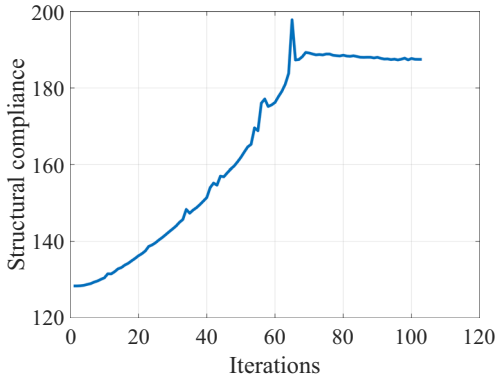


Fig. 6 Convergence history of the compliance objective function for the MBB beam example

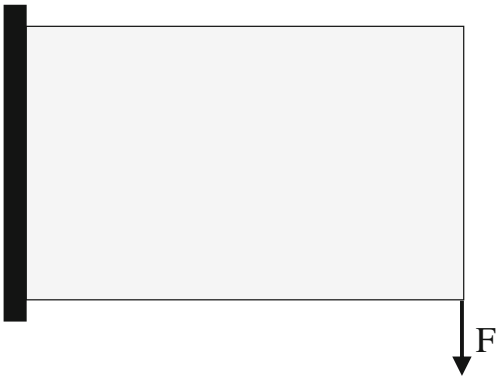


Fig. 8 Short cantilever design problem

that the constraint function $V(\boldsymbol{x})/V_0$ can decrease by a maximum of 1% of the difference between $V(\boldsymbol{x}^k)/V_0$ and $\overline{V}$ at each iteration k. This is reasonable and logical when dealing with volume fraction constraints. Using higher ϵ's leads to faster change in constraint values (material removal in case of volume) toward their prescribed bounds, however, at the risk of creating convergence issues for large ϵ's. Lower ϵ's lead to a slower change in the constraint function but helps in obtaining smoother convergence. Figure 10 presents the convergence history of the volume fraction constraint using the four different ϵ's.

The truncation error constraint parameter is chosen to be $\beta = 0.05$ for this cantilever optimization example. This means that the number of flips on design variables (from 1 to 0 and from 0 to 1 summed up) is constrained to be a maximum of 5% of the total number of elements N_d. In this example, this choice does not influence the optimization as the minimum compliance solution will always reach the upper bound of volume fraction prescribed using ϵ each iteration. Therefore, the only restriction when choosing ϵ and β for a single volume constraint is that $\epsilon \leq \beta$.

(a)

(b)

Fig. 7 MBB beams designed with the 101-line TOBS code for (**a**) 240×80 and (**b**) 480×160 meshes

4.3 Passive elements

Some applications of structural optimization include some non-designable regions in the design domain. This means the elements in these regions must maintain their solid or void phases from the initial solution, e.g., the deck of a bridge or the hole in a pipe. This example shows that the implementation of a non-design domain is straightforward using the 101-line code. For the cantilever example in Fig. 11(a), a circular region with radius `nelx/3` and center (`nely/2`, `nelx/3`) can be defined in the code by:

```
passive = zeros(size(x));
for ely = 1:nely
    for elx = 1:nelx
     if sqrt((ely-nely/2.)^2 + ...
      ... (elx-nelx/3.)^2) < (nely/3)
        passive(ely,elx) = 1;
        x(ely,elx) = 0;
      end
   end
end
```

where the matrix `passive` is introduced to account for passive elements, using the terminology in Sigmund (2001b) to refer to the elements that can not have their design variables flipping. This piece of code can also set void passive regions via `x(ely,elx) = 0;`. This piece of code should be written after line 42 and before the optimization loop in the original code. The ILP function remains unchanged despite the presence of passive elements. This is done by setting the objective/constraint function sensitivities of the design variables corresponding

(a) $\epsilon = 0.005$ (b) $\epsilon = 0.01$ (c) $\epsilon = 0.02$ (d) $\epsilon = 0.04$

Fig. 9 Optimized solutions for the short cantilever problem corresponding to different ϵ's

to the passive elements to be zero. The following commands should be inserted right after sensitivity filtering is done in line 57:

```
dc(find(passive)) = 0;
dv(find(passive)) = 0;
```

Figure 11 (b and c) presents the short cantilever beams designed with the modified code considering the case where the circular region must remain void (Fig. 11b) and another case where it remains solid (Fig. 11c). The design domain used is 160 × 100, and $\overline{V} = 0.5$, $\epsilon = 0.01$, $\beta = 0.05$, and $r_{\min} = 5$.

4.4 Symmetric condition

The topology optimization solution is sometimes (anti) symmetric due to symmetries in geometry and loading. This is the case of the cantilever beam depicted in Fig. 12 (a). One of the TOBS features is the formal scheme of relaxing the constraint functions so the integer programming solver can yield a feasible solution. The obtained solution can be asymmetric when the number of design variable flips at an iteration is an odd number. In such iteration, the update of the cantilever design in Fig. 12 (a) loses symmetry. Instead of manually controlling the update in order to enforce a symmetric design, we suggest to solve the problem only for the symmetric region. In order to do that, the following piece of code should replace line 62 in the original code, where the ILP function is called.

```
dc = dc(1:nely/2,:); dv = 2*dv(1:nely/2,:);
[x_new] = ILP(dc(:),dv(:),gbar, ...
gi(loop),epsilons,beta,x(1:nely/2,:));
x(1:nely/2,:) = x_new;
x(nely:-1:nely/2+1,:) = x_new;
```

In this case, the ILP function is called only with the design variables and sensitivities in the top half (`1:nely/2`) design domain of the cantilever problem. Volume sensitivities are multiplied by 2, as in `2*dv(1:nely/2,:)`, because a flip in one design variable now means the volume fraction change of two elements. A x_new set of densities are obtained and the final update is done considering symmetry. This is carried out for a horizontal line of symmetry and can be done similarly for a vertical line of symmetry. Figure 12 (b) presents the solution using the modified code with horizontal line of symmetry implementation. The design domain used is 160 × 100, and $\overline{V} = 0.5$, $\epsilon = 0.01$, $\beta = 0.05$, and $r_{\min} = 10$. The code corresponding to the force vector of this cantilever beam is:

```
F = sparse(2*(nely+1)*(nelx+1)-nely, ...
 1,-1,2*(nely+1)*(nelx+1),1);
```

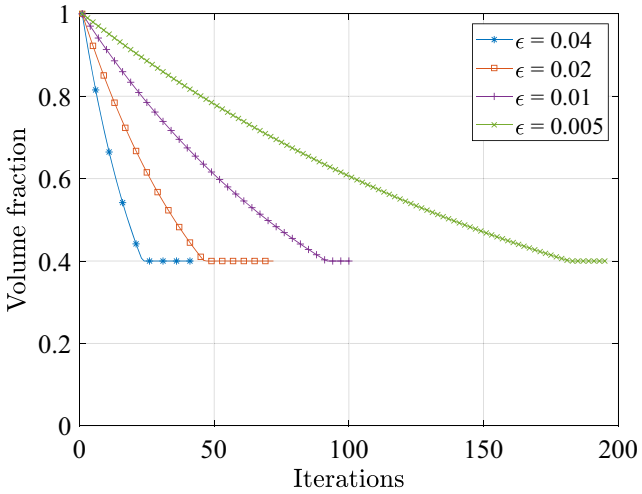


Fig. 10 Convergence histories of the volume fraction constraint functions for the cantilever example using different ϵ's

4.5 Volume minimization

The integer programming problem is generalized by using sequential linear approximation. Therefore, in the TOBS method, the constraint function does not need to be volume-based, unlike BESO. This example solves problem P2 in (10), volume minimization subject to a compliance constraint. In order to do that, the following commands should replace lines 61 and 62 in the original line code.

```
obj(loop) = mean(x(:)); gi(loop) = c;
```

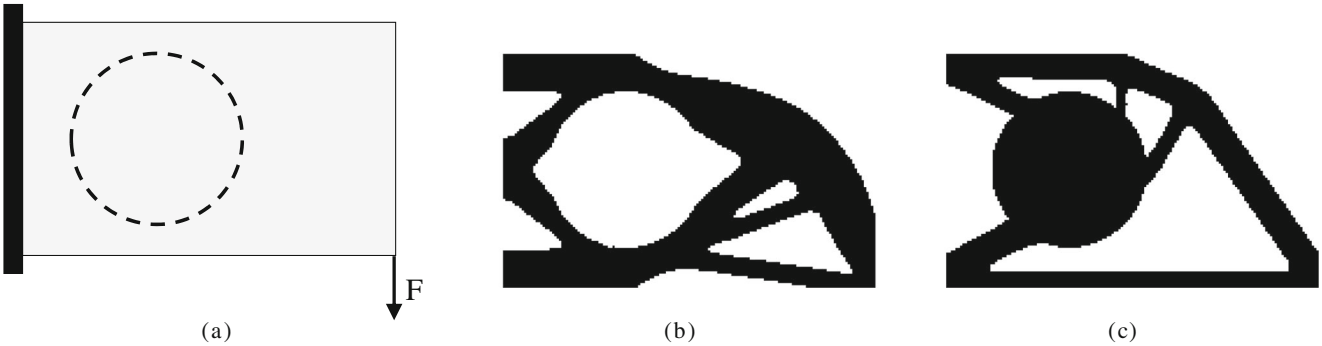
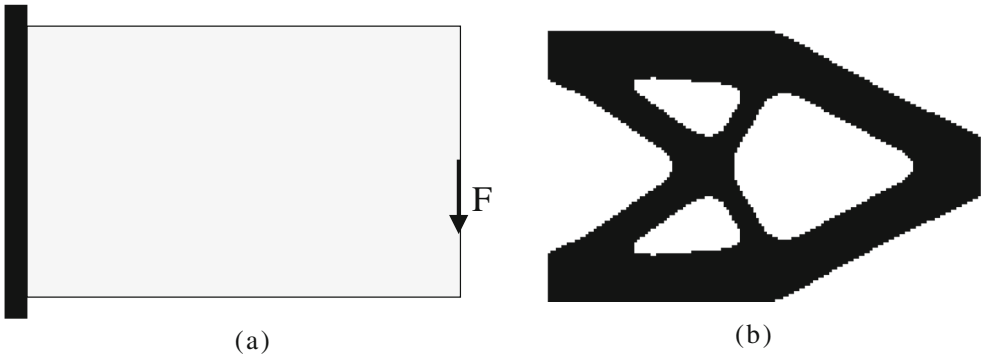


Fig. 11 Short cantilever (**a**) design problem and solutions for (**b**) void and (**c**) solid passive circular regions

Fig. 12 Short cantilever (**a**) design problem with known symmetry and (**b**) solution via the modified 101-line TOBS code considering symmetry



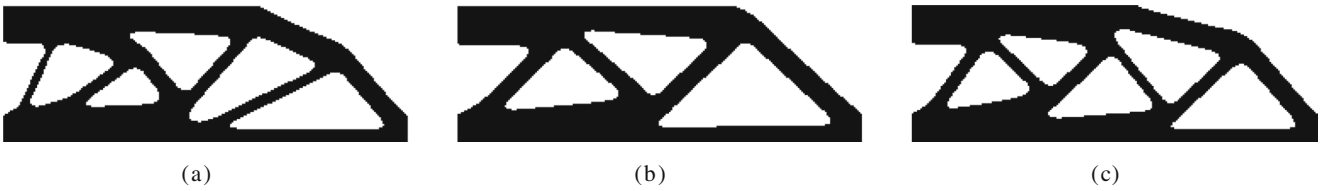


Fig. 13 Minimum volume optimized solutions for the MBB beam with $\overline{C} = 180$ using (**a**) $\epsilon = 0.01$, (**b**) $\epsilon = 0.005$, and (c) $\epsilon = 0.0025$

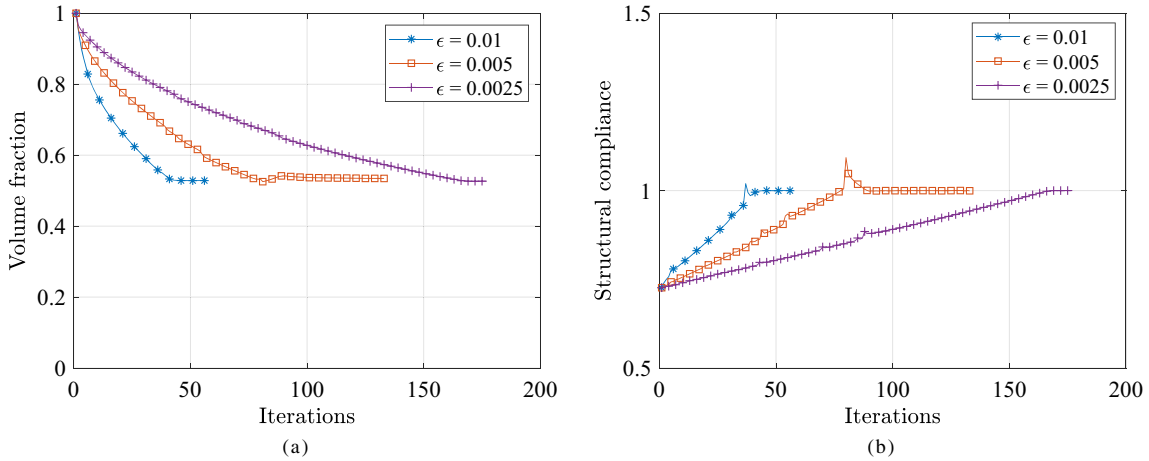


Fig. 14 Convergence history for the minimum volume problem subject to compliance constraint: (**a**) volume fraction and (**b**) structural compliance, for different ϵ's

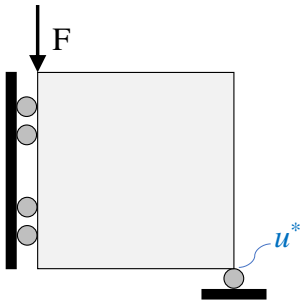


Fig. 15 Design problem considering a displacement constraint

```
[x] = ILP(dv(:),dc(:),gbar,gi(loop), ...
 epsilons,beta,x);
```

It can be noticed that, although we kept the same notation for compliance and volume variables, they have switched places when prescribing the objective and constraint functions.

This example solves the MBB beam depicted in Fig. 4. The mesh size used is 240×80, the compliance constraint $\overline{C} = 180$. We run the problem for $\epsilon = 0.01$, 0.005, and 0.0025, with $\beta = 0.05$ and $r_{\min} = 8$. Figure 13 presents the structural designs obtained using the three different ϵ's.

The optimizations converged smoothly to a final volume fraction of 0.5283, 0.5344, and 0.5267 when using $\epsilon = 0.01$, 0.005, and 0.0025, respectively. Figure 14 (a) presents the convergence of the volume fraction (objective function for this problem). For $\epsilon = 0.01$, the material was removed fast and the solution converged in 57 iterations. However, it can be mentioned that in the first iteration the volume fraction moved from 1.0 to 0.95, activating the truncation error constraint defined by $\beta = 0.05$. This shows the importance of this extra constraint. Lower ϵ dictates slower evolution but smoother convergence via finer control of the constraint change. Figure 14 (b) presents the convergence history of the normalized compliance $C/\overline{C}$ as the constraint function. The three cases converged to $C/\overline{C} = 1$. This example aims to illustrate that the present binary optimization is generalized so it can address different types of constraints, which sometimes is hard for other binary methods. This example might also illustrate that the present TOBS method approximates from density-based methods via the formal mathematical programming framework.

4.6 Multiple constraints

This example solves problem P3 from (11). The idea is to illustrate that considering multiple constraints with the present code is straightforward. In (15), the authors present the design problem considering a displacement constraint. The displacement u^* at the bottom right corner of the domain is constrained to be lower or equal than $\bar{u}$ in the horizontal direction.

For the displacement sensitivity computation, the adjoint load from (21) can be implemented in the code in a second column of the force vector as:

```
F(2*(nelx+1)*(nely+1)-1,2) = -1;
```

where `2*(nelx+1)*(nely+1)-1` is the corresponding DoF of the displacement illustrated in Fig. 15. Thus, the adjoint and structural displacement vectors $\boldsymbol{\lambda}$ and $\boldsymbol{u}$, respectively, can be obtained by solving the linear systems of equations at once. The displacement sensitivities are computed and filtered similarly to the case of compliance. The displacement constraint must be passed to the ILP solver by adding columns in the variables representing the constraint functions. Thus, the ILP function can be called via the following line:

```
[x] = ILP(dc(:),[dv(:) dc_disp(:)], ...
        gbar,gi(loop,:),epsilons,beta,x);
```

where `gi` is the value of the volume and displacement constraints, stored as:

```
gi(loop,1:2) = [mean(x(:)) ...
               U(2*(nelx+1)*(nely+1)-1,1)];
```

Problem P3 is solved for the example in Fig. 15. The mesh size used is 200×200 and the volume constraint $\overline{V} = 30\%$. The TOBS parameters are chosen to be $\epsilon = 0.01$ for the volume constraint and $\epsilon = 0.05$ for the displacement, with $\beta = 0.05$ for the truncation error constraint. The filter radius is set as $r_{\min} = 2$. The time stabilization scheme from (27) is turned off in this example. That procedure helps in reducing oscillations in the topology and stabilizing the binary optimization when solving compliance minimization subject to a volume constraint (Huang and Xie 2007). Adding the displacement constraint has a similar effect and time stabilization is not needed in this case. Furthermore, when solving multiple constrained problems, it is beneficial to use the original (not averaged) sensitivities (Sivapuram et al. 2018), as the solution space is reduced. We run the code for four different displacement cases, $\bar{u} = 10$, 12, 14, and 100. For example, with $\bar{u} = 10$ and the chosen parameters, the present code can be called (with the proper modifications) via:

```
tobs101(200,200,[0.30 10.0], ...
 [0.01 0.05],0.05,2)
```

Figure 16 presents the topology solutions for the four different displacements constraints. It can be noticed that

(a) $u^* \leq 10$ (b) $u^* \leq 12$ (c) $u^* \leq 14$ (d) $u^* \leq 100$

Fig. 16 Optimized solutions for the design problem with different displacement constraints

the displacement converges to the constrained values in the cases where $\bar{u}$ = 10, 12, and 14. By solving the problem with a very high value, i.e., the case where $\bar{u}$ = 100, the solution converged with the displacement constraint inactive (see Fig. 17) as the compliance (objective) function reached a local minimum. This can show the capabilities of the ILP procedure to accommodate active and inactive constraints. The TOBS method solves the multiple constraints problem via branch-and-bound algorithms, therefore not requiring the introduction of arbitrary multipliers in the objective function as carried out in the BESO method, for instance. This makes the consideration of more than two constraints easier if compared with BESO. For further discussions on the capabilities of the TOBS method to address multiple constraints, the reader is referred to Sivapuram et al. (2018) who investigated microstructural designs including up to six non-volume constraints.

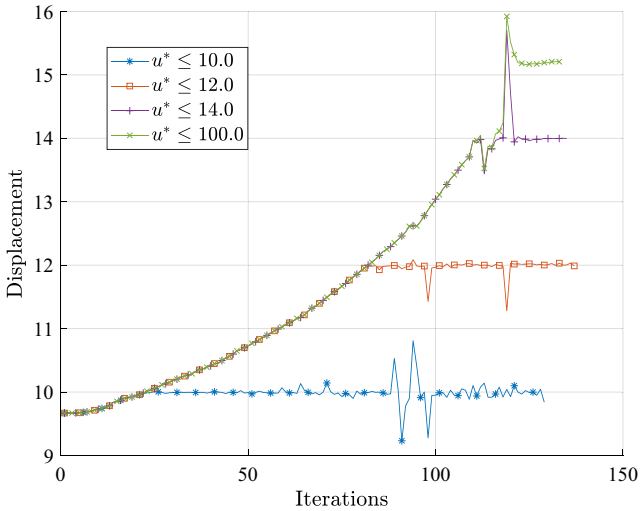


Fig. 17 Convergence of the displacement constraint when solving problem P3 for the example from Fig. 15 with the present 101-line code

5 Conclusions

We presented a 101-line code with the implementation of the TOBS method for educational purposes. The finite element analysis presented is based on existing efficient educational codes. The framework is detailed, both mathematically and numerically. We illustrated the use of integer programming in topology optimization and its practicability. The minimum compliance MBB example subject to a volume constraint is investigated. We showed that convergence of the binary topologies was smooth and the optimized solutions were mesh independent. A case with 76,800 elements is shown to illustrate that even large examples can be studied using the proposed implementation using a personal computer, although we have investigated larger cases in previous works (Sivapuram and Picelli 2020). The parameters intrinsic to the TOBS method were discussed in detail. We also presented the modifications required for passive elements and symmetric designs. A volume minimization problem subject to a compliance constraint was solved. The structural compliance and volume can be switched between objective and constraint functions by changing a few lines of the code. Finally, a compliance minimization problem subject to volume and displacements constraints was solved. This example showed that multiple constraints can be considered without further special treatments. Also important to mention, the ILP function has not been changed for any of the presented examples and it can be tested and extended for other applications.

Funding information The first author would like to thank the support of FAPESP (São Paulo Research Foundation), grants 2018/05797-8 and 2019/01685-3 under the Young Investigators Awards program.

Compliance with ethical standards

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

Replication of results The 101-line code is provided in the Appendix A. The code and its modified versions to run all examples in this paper are also provided as supplementary material. A more detailed implementation of the TOBS method is available upon request by email or for download at https://github.com/renatopicelli/tobs.

Appendix A: 101-line TOBS code

```
\%\%\%\% A 101-LINE TOPOLOGY OPTIMIZATION OF BINARY STRUCTURES CODE, 2020 \%\%\%\%\%
function tobs101(nelx,nely,gbar,epsilons,beta,rmin)
\%\% MATERIAL PROPERTIES
E0 = 1; Emin = 1e-9; nu = 0.3; penal = 3;
\%\% PREPARE FINITE ELEMENT ANALYSIS
A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12];
A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6];
B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4];
B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2];
KE = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]);
nodenrs = reshape(1:(1+nelx)*(1+nely),1+nely,1+nelx);
edofVec = reshape(2*nodenrs(1:end-1,1:end-1)+1,nelx*nely,1);
edofMat = repmat(edofVec,1,8)+repmat([0 1 2*nely+[2 3 0 1] -2 -1],nelx*nely,1);
iK = reshape(kron(edofMat,ones(8,1))',64*nelx*nely,1);
jK = reshape(kron(edofMat,ones(1,8))',64*nelx*nely,1);
\% DEFINE LOADS AND SUPPORTS (HALF MBB-BEAM)
F = sparse(2*(nely+1),1,-1,2*(nely+1)*(nelx+1),1);
U = zeros(2*(nely+1)*(nelx+1),1);
fixeddofs = union([1:2:2*(nely+1)],[2*(nelx+1)*(nely+1)]);
alldofs = [1:2*(nely+1)*(nelx+1)];
freedofs = setdiff(alldofs,fixeddofs);
\%\% PREPARE FILTER
iH = ones(nelx*nely*(2*(ceil(rmin)-1)+1)^2,1);
jH = ones(size(iH)); sH = zeros(size(iH)); k = 0;
for i1 = 1:nelx
    for j1 = 1:nely
        e1 = (i1-1)*nely+j1;
        for i2 = max(i1-(ceil(rmin)-1),1):min(i1+(ceil(rmin)-1),nelx)
            for j2 = max(j1-(ceil(rmin)-1),1):min(j1+(ceil(rmin)-1),nely)
                e2 = (i2-1)*nely+j2;
                k = k+1;
                iH(k) = e1;
                jH(k) = e2;
                sH(k) = max(0,rmin-sqrt((i1-i2)^2+(j1-j2)^2));
            end
        end
    end
end
H = sparse(iH,jH,sH);
Hs = sum(H,2);
\%\% INITIALIZE ITERATION
x = ones(nely,nelx); \% Initial variables
loop = 0; change = 1; obj = 0; gi = 0;
\% START ITERATION
while change > 1e-4
```

```
    loop = loop + 1;
    \% FE-ANALYSIS
    sK = reshape(KE(:)*(Emin+x(:)'.^penal*(E0-Emin)),64*nelx*nely,1);
    K = sparse(iK,jK,sK); K = (K+K')/2;
    U(freedofs) = K(freedofs,freedofs)\F(freedofs);
    \% OBJECTIVE FUNCTION AND SENSITIVITY ANALYSIS
    ce = reshape(sum((U(edofMat)*KE).*U(edofMat),2),nely,nelx);
    c = sum(sum((Emin+x.^penal*(E0-Emin)).*ce));
    dc = -penal*(E0-Emin)*x.^(penal-1).*ce;
    dv = ones(nely,nelx)/(nelx*nely);
    \% FILTERING/STABILIZATION
    dc(:) = H*(dc(:)./Hs);
    if (loop > 1); dc = (dc + olddc) / 2; olddc = dc;
    else; olddc = dc; end
    \% TOBS UPDATE
    obj(loop) = c; gi(loop) = mean(x(:));
    [x] = ILP(dc(:),dv(:),gbar,gi(loop),epsilons,beta,x);
    \% CONVERGENCE ANALYSIS
    if loop > 10
        change = abs(sum(obj(loop-9:loop-5))-sum(obj(loop-4:loop)))/sum(obj(loop-4:loop));
    end
    \% PRINT RESULTS
    fprintf(' It.:\%5i Comp.:\%11.4f Vol.:\%7.3f ch.:\%7.3f\n^{\prime},loop,c,mean(x(:)),
    change);
    colormap(gray); imagesc(1-x); caxis([0 1]); axis equal; axis off; drawnow;
end
end
\%\% ILP function
function [x] = ILP(dfdx,dgdx,gbar,gi,epsilons,beta,x)
\% Truncation error (for flip limits constraint)
C1 = (x(:) == 0);
C2 = -(x(:) == 1);
truncation = C1 + C2;
\% Normalization of sensitivities
norm = max(max(abs(dgdx)), eps);
dgdx = dgdx./repmat(norm,size(dgdx,1),1);
\% Constraints sensitivities
dgdx = [dgdx'; truncation'];
\% Constraint relaxation (move limit)
target = (gbar - gi)./norm;
deltag = epsilons.*(abs(gi)./norm);
A = (target > deltag);
B = (target < -deltag);
constlimits = (A - B).*deltag + (1 - (A + B)).*target;
constlimits = [constlimits'; beta * length(x(:))];
\% Variable limits
lower_limits = -1.0 * (abs(x(:) - 1) < 0.001);
upper_limits = 1.0 * (abs(x(:)) < 0.001);
\% Update with INTLINPROG
```

```
 94 OptimizerOptions = optimoptions('intlinprog', 'CutGeneration', 'intermediate',
    'RootLPAlgorithm','primal-simplex', ...
 95     'NodeSelection', 'mininfeas', 'HeuristicsMaxNodes', 100, 'RootLPMaxIter', 60000, ...
 96     'MaxNodes', 1e5, 'Display', 'off');
 97 [deltax, ~, ~, ~] = intlinprog (dfdx(:), 1:length(x(:)), ...
 98     dgdx, constlimits, [], [], ...
 99     lower_limits, upper_limits, OptimizerOptions);
100 x(:) = x(:) + deltax;
101 end
```

References

Allaire G (2009) A 2-D Scilab Code for shape and topology optimization by the level set method. http://www.cmap.polytechnique.fr/allaire/levelset_en.html

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

Ansola LR, Querin OM, Garaigordobil JA, Alonso GC (2018) A sequential element rejection and admission (SERA) topology optimization code written in Matlab. Struct Multidiscip Optim 58(3):1297–1310. https://doi.org/10.1007/s00158-018-1939-x

Beckers M (1999) Topology optimization using a dual method with discrete variables. Struct Multidiscip Optim 17:14–24

Challis VJ (2010) A discrete level-set topology optimization code written in Matlab. Struct Multidiscip Optim 41:453–464

Cornuéjols G (2008) Valid inequalities for mixed integer linear programs. Math Program B 112:3–44

Deaton JD, Grandhi RV (2014) A survey of structural and multidisciplinary continuum topology optimization: post 2000. Struct Multidiscip Optim 49:1–38

Emmendoerfer H, Fancello EA, Silva ECN (2018) Level set topology optimization for design-dependent pressure load problems. Int J Numer Methods Eng 115(7):825–848

Feppon F, Allaire G, Bordeu F, Cortial J, Dapogny C (2019) Shape optimization of a coupled thermal fluid-structure problem in a level set mesh evolution framework. SeMA J 76(3):413–458

Gao T, Zhang W (2010) Topology optimization involving thermo-elastic stress loads. Struct Multidiscip Optim 42(5):725–738

Guest JK, Prèvost JH, Belytschko T (2004) Achieving minimum length scale in topology optimization using nodal design variables. Int J Numer Meth Eng 61(2):238–254

Haftka RT, Gürdal Z (1991) Elements of structural optimization, 3rd edn. Kluwer Academic Publishers, Kluwer

Huang X, Xie YM (2007) Convergent and mesh-independent solutions for the bi-directional evolutionary structural optimization method. Finite Elem Anal Des 43:1039–1049

Land AH, Doig AG (1960) An automatic method of solving discrete programming problems. Econometrica 28:497–520

Liang Y, Cheng G (2019) Topology optimization via sequential integer programming and canonical relaxation algorithm. Comp Methods Appl Mechan Eng 348:64–96. https://doi.org/10.1016/j.cma.2018.10.050

Liang Y, Cheng G (2020) Further elaborations on topology optimization via sequential integer programming and canonical relaxation algorithm and 128-line MATLAB code. Struct Multidiscip Optim 61:411–431

Neofytou A, Picelli R, Huang TH, Chen JS, Kim HA (2020) Level set topology optimization for design-dependent pressure loads using the reproducing kernel particle method. Struct Multidiscip Optim 61:1805-1820

Picelli R, Sivapuram R (2019) Solving topology optimization with 0,1 design variables and mathematical programming: the TOBS method. In: Proceedings of the 13th world congress of structural and multidisciplinary optimization (WCSMO-13), Beijing, China

Picelli R, Neofytou A, Kim HA (2019) Topology optimization for design-dependent hydrostatic pressure loading via the level-set method. Struct Multidiscip Optim 60:1313–1326

Picelli R, Ranjbarzadeh S, Sivapuram R, Gioria RS, Silva ECN (2020) Topology optimization of binary structures under design-dependent fluid-structure interaction loads. Struct Multidiscip Optim 62:2101-2116

Querin OM, Steven GP, Xie YM (1998) Evolutionary structural optimisation (ESO) using a bidirectional algorithm. Eng Comput 15(8):1031–1048

Rozvany GIN, Zhou M, Birker T (1992) Generalized shape optimization without homogenization. Struct Opt 4:250–254

Sigmund O (2001a) A 99 line topology optimization code written in Matlab. Struct Multidiscip Optim 21:120–127

Sigmund O (2001b) Design of multiphysics actuators using topology optimization - Part i: one material structures. Comput Methods Appl Mech Eng 190:6577–6604

Sigmund O (2007) Morphology-based black and white filters for topology optimization. Struct Multidiscip Optim 33(4-5):401–424

Sivapuram R, Picelli R (2018) Topology optimization of binary structures using integer linear programming. Finite Elem Anal Des 139:49–61

Sivapuram R, Picelli R (2020) Topology design of binary structures subjected to design-dependent thermal expansion and fluid pressure loads. Struct Multidiscip Optim 61:1877–1895

Sivapuram R, Picelli R, Xie YM (2018) Topology optimization of binary microstructures involving various non-volume constraints. Comput Mater Sci 154:405–425. https://doi.org/10.1016/j.commatsci.2018.08.008

Svanberg K, Werme M (2005) A hierarchical neighbourhood search method for topology optimization. Struct Multidiscip Optim 29:325–340

Svanberg K, Werme M (2006) Topology optimization by sequential integer linear programming. In: Bendsøe MP, Olhoff N, Sigmund O (eds) IUTAM symposium on topological design optimization of structures, machines and materials. Springer, Netherlands, pp 425-436

Vanderbei RJ (2014) Linear programming: foundations and extensions, 4th edn. Springer, US

Vicente WM, Picelli R, Pavanello R, Xie YM (2015) Topology optimization of frequency responses of fluid-structure interaction systems. Finite Elem Anal Des 98:1–13

Wei P, Li Z, Li X, Wang MY (2018) An 88-line matlab code for the parameterized level set method based topology optimization using radial basis functions. Struct Multidiscip Optim 58:831–849

Williams P (2009) Integer programming. Springer, Boston, pp 25–70. https://doi.org/10.1007/978-0-387-92280-5_2

Xia L, Xia Q, Huang X, Xie YM (2018) Bi-directional evolutionary structural optimization on advanced structures and materials: a comprehensive review. Arc Comput Methods Eng 25(2):437–478

Xie YM, Steven GP (1993) A simple evolutionary procedure for structural optimization. Comput Struct 49:885–896

Yoon GH (2020) Topology optimization method with finite elements based on the $k - \varepsilon$ turbulence model. Comput Methods Appl Mechan Eng 361:112–784

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

Zuo ZH, Xie YM (2015) A simple and compact python code for complex 3D topology optimization. Adv Eng Softw 85:1–11