



# FEniTop: a simple FEniCSx implementation for 2D and 3D topology optimization supporting parallel computing

Yingqi Jia<sup>1</sup> · Chao Wang<sup>1</sup> · Xiaojia Shelly Zhang<sup>1,2,3</sup>

Received: 22 November 2023 / Revised: 3 March 2024 / Accepted: 15 May 2024  
© The Author(s) 2024, corrected publication 2024

## Abstract

Topology optimization has emerged as a versatile design tool embraced across diverse domains. This popularity has led to great efforts in the development of education-centric topology optimization codes with various focuses, such as targeting beginners seeking user-friendliness and catering to experienced users emphasizing computational efficiency. In this study, we introduce FEniTop, a novel 2D and 3D topology optimization software developed in Python and built upon the open-source FEniCSx library, designed to harmonize usability with computational efficiency and post-processing for fabrication. FEniTop employs a modular architecture, offering a unified input script for defining topology optimization problems and six replaceable modules to streamline subsequent optimization tasks. By enabling users to express problems in the weak form, FEniTop eliminates the need for matrix manipulations, thereby simplifying the modeling process. The software also integrates automatic differentiation to mitigate the intricacies associated with chain rules in finite element analysis and sensitivity analysis. Furthermore, FEniTop provides access to a comprehensive array of readily available solvers and preconditioners, bolstering flexibility in problem-solving. FEniTop is designed for scalability, furnishing robust support for parallel computing that seamlessly adapts to diverse computing platforms, spanning from laptops to distributed computing clusters. It also facilitates effortless transitions for various spatial dimensions, mesh geometries, element types and orders, and quadrature degrees. Apart from the computational benefits, FEniTop facilitates the automated exportation of optimized designs, compatible with open-source ParaView software for post-processing. This functionality allows for visualizing optimized designs across diverse mesh geometries and element shapes, automatically smoothing 3D designs, and converting smoothed designs into STereoLithography (STL) files for 3D printing. To illustrate the capabilities of FEniTop, we present five representative examples showcasing topology optimization across 2D and 3D geometries, structured and unstructured meshes, solver switching, and complex boundary conditions. We also assess the parallel computational efficiency of FEniTop by examining its performance across diverse computing platforms, process counts, problem sizes, and solver configurations. Finally, we demonstrate a physical 3D-printed model utilizing the STL file derived from the design optimized by FEniTop. These examples showcase not only FEniTop's rich functionality but also its parallel computing performance. The open-source FEniTop is given in [Appendix B](#) and will be available to download at <https://github.com/missionlab/fenitop>.

**Keywords** Topology optimization · Parallel computing · Automatic differentiation · Weak form · FEniCSx · Python · 3D printing

Responsible Editor: Helder Rodrigues.

✉ Xiaojia Shelly Zhang  
zhangxs@illinois.edu

<sup>1</sup> Department of Civil and Environmental Engineering, University of Illinois Urbana-Champaign, Urbana, IL 61801, USA

<sup>2</sup> Department of Mechanical Science and Engineering, University of Illinois Urbana-Champaign, Urbana, IL 61801, USA

<sup>3</sup> National Center for Supercomputing Applications, Urbana, USA

## 1 Introduction

Topology optimization (Bendsøe and Kikuchi 1988; Bendsoe and Sigmund 2003) has emerged as a robust design tool for optimizing material distribution to minimize a specified objective function while adhering to defined constraints. This approach has found widespread utility across diverse domains, including mechanical engineering (Wang et al. 2023; Li et al. 2023), aerospace engineering (Aage et al. 2017), and structural engineering (Beghini et al. 2014; Zargham et al. 2016; Jia et al. 2023). To implement topology optimization, substantial progress has been made in the

development of educational codes within the structural and multidisciplinary optimization (SMO) community, such as Sigmund (2001), Andreassen et al. (2011), Talischi et al. (2012), Duarte et al. (2015), Sanders et al. (2018), Ferrari and Sigmund (2020), Giraldo-Londoño and Paulino (2021a), Smit et al. (2021) and Chandrasekhar and Suresh (2021). Readers are referred to Wang et al. (2021) for a comprehensive review of the educational efforts in SMO. These codes have greatly benefited beginners entering the optimization field and aided experienced researchers in completing optimization tasks within their respective domains.

Educational codes for topology optimization primarily originate from codes designed for classical topology optimization problems involving structural compliance minimization (i.e., maximizing structural stiffness). These codes, including `top99` (Sigmund 2001), `top88` (Andreassen et al. 2011), `top3d` (Liu and Tovar 2014), and `PolyTop` (Talischi et al. 2012), are primarily implemented in the commercial software, `Matlab`, offering immediate usability following software installation. Their exceptional readability and user-friendliness have played a pivotal role in facilitating the entry of newcomers into the topology optimization field.

Building upon these topology optimization educational codes, researchers have developed field-specific software for handling complex optimization tasks that go beyond maximizing structural stiffness with a single material and a material volume constraint. To give a few examples, Xia and Breitkopf (2015) present topology optimization for designing unit cells of periodic structures using an energy-based homogenization approach. `PolyMat` (Sanders et al. 2018) presents topology optimization for compliance minimization with multiple candidate materials and multiple volume constraints. `PolyStress` (Giraldo-Londoño and Paulino 2021b) implements topology optimization with local stress constraints based on the augmented Lagrangian method. `PolyDyna` (Giraldo-Londoño and Paulino 2021a) optimizes mean structural compliance under dynamic loading using the HHT- $\alpha$  method and a variation of the ZPR design variable update scheme. Homayouni-Amlashi et al. (2021) propose topology optimization codes for designing piezoelectric actuators and energy harvesters, considering density and polarization direction as optimization variables. These field-specific educational codes allow researchers outside the optimization field to conveniently perform topology optimization tasks, therefore promoting the application of topology optimization in diverse fields.

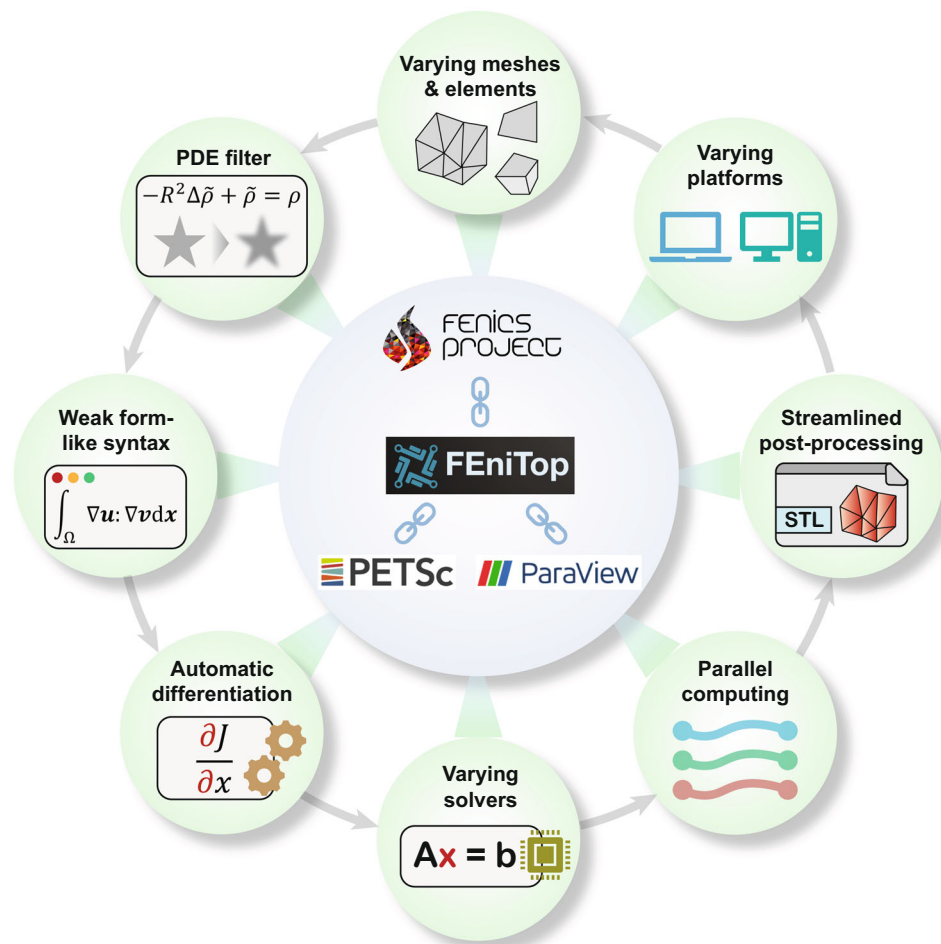
As topology optimization spreads rapidly across various fields, more educational codes, such as Duarte et al. (2015), Aage et al. (2015), Ferrari and Sigmund (2020), Zhang et al. (2021), Träff et al. (2023), are contributing to improved computational efficiency. The necessity to reduce the computational costs arises from the complexity of the topology optimization process, which involves solving

equilibrium equations in finite element analysis (FEA) and adjoint equations in sensitivity analysis across numerous iterations. To address the high computational demands, recent educational codes employ advanced techniques such as computationally efficient programming languages (e.g., C/C++ and Fortran), robust solver algorithms (both nonlinear and linear solvers, along with preconditioners), improved density filtering and solution update schemes, and CPU-based or GPU-based parallel computing. For instance, `PolyTop++` (Duarte et al. 2015) leverages robust solvers developed with the C++ programming language and CUDA extension for parallel computing. `top99neo` (Ferrari and Sigmund 2020) employs a convolution operation to expedite density filtering and utilizes the proposed “`fsparse`” subroutine to accelerate matrix assembly in FEA. These computational-efficiency-centric educational codes further enable the application of topology optimization to large-scale and complex nonlinear optimization problems.

With the significant progress made by educational codes for topology optimization, a comparison of these educational codes reveals a trade-off between ease of use/extensibility for complex problems and computational efficiency. For instance, educational codes designed for readability may be less computationally efficient for large-scale problems. Computational-efficiency-centric codes are more compact and require users to delve into programming intricacies (e.g., data transferring between CPU and GPU, control flow optimization, and memory coalescing). To better balance the ease of use and computational efficiency, this paper introduces `FEniTop` (Fig. 1), an innovative topology optimization software that combines user-friendliness with computational efficiency. `FEniTop` is built upon the open-source `FEniCSx` software (Scroggs et al. 2022), an evolution of the `FEniCS` library (Langtangen and Logg 2016). `FEniCSx` serves as a versatile platform for solving partial differential equations (PDEs), enabling users to effortlessly translate mathematical models into efficient finite element codes that support parallel computing, spanning from personal laptops to high-performance computing clusters.

The features of `FEniCSx` include user-friendliness, high-performance computation, open-source accessibility, and an active community, rendering it a promising resource in a variety of applications. These appealing capabilities of the `FEniCSx` library stems from its four primary components: `DOLFINx` (Logg and Wells 2010) for computational environments, `Basix` (Scroggs et al. 2022) for finite element definition and tabulation, `UFL` (Alnæs et al. 2014) for declaring finite element discretizations of variational forms, and finally `FFCx` (Kirby and Logg 2006) for compiling these variational forms into low-level C code. `FEniCSx` fosters seamless cooperation among these components and offers

**Fig. 1** Illustration of the innovative FEniTop software for facilitating streamlined topology optimization



interfaces in both C++ and high-level Python programming languages.

In this work, we capitalize on the Python interface of FEniCSx to introduce FEniTop for 2D and 3D topology optimization supporting parallel computing. FEniTop presents several key advantages (Fig. 1), including (1) seamless transitions to various spatial dimensions (2D and 3D), mesh geometries (structured and unstructured), element geometries (e.g., triangles, quadrilaterals), element types (e.g., Lagrange, Crouzeix–Raviart), element orders, and quadrature degrees; (2) the expression of PDEs in weak form, bypassing the need for tedious matrix manipulation; (3) the automatic differentiation that can mitigate the intricacies of chain rules in FEA and sensitivity analysis; (4) the access to readily available linear and nonlinear solvers and preconditioners through the PETSc backend (Balay et al. 2019) via `petsc4py` (Python interface to PETSc) (Dalcin et al. 2011); (5) scalable parallel computing compatible with various platforms from laptops to distributed computing clusters. Based on these attractive features, the proposed FEniTop allows users to conveniently develop computationally effective topology optimization codes.

Besides the above strengths synergized from FEniCSx, the proposed FEniTop also enables the following features to ensure streamlined topology optimization (Fig. 1). (1) FEniTop includes a Helmholtz-type PDE filter (Lazarov and Sigmund 2011) tailored for large-scale problems in parallel computing, particularly emphasizing its efficacy for large filter radii. (2) FEniTop enables parallel implementations for both the Optimality Criteria (OC) optimizer (Hassani and Hinton 1998) and the Method of Moving Asymptotes (MMA) (Svanberg 1987) to update the design variables. Incorporating these optimizers allows the users to handle complex design problems beyond compliance minimization. (3) FEniTop automatically exports optimized designs into eXtensible Data Model and Format (XDMF) files, enabling seamless importation into open-source ParaView software (Ayachit 2015) for post-processing. This process facilitates visualization of optimized designs, filtering of low-density regions, smoothing of jagged boundaries, and conversion of smoothed designs into STereoLithography (STL) files for 3D printing. We remark that FEniTop is fully implemented in parallel while maintaining code readability. In addition, FEniTop is highly modularized and can be extended to

tackle more sophisticated design problems, as exemplified in topology optimization for nonlinear coupled-field fracture problems in Jia et al. (2023).

In this study, we demonstrate the utilization of FEniTop for topology optimization of linear elastic structures with maximized static stiffness and output displacement, and the paper is organized as follows. Section 2 provides a concise overview of density-based topology optimization. Section 3 delves into the detailed rationale and coding techniques behind FEniTop. Section 4 presents five illustrative examples that demonstrate topology optimization leveraging FEniTop, showcasing 2D and 3D geometries, structured and unstructured meshes, solver switching, and complex boundary conditions. This section also explores the computational efficiency of FEniTop by examining its performance for various computing platforms, numbers of processes, mesh sizes, and solvers. Finally, Sect. 5 offers several concluding remarks. The paper is complemented by two appendices. Appendix A elucidates the sensitivity analysis through automatic differentiation, and Appendix B presents the complete implementation of FEniTop.

## 2 Fundamentals of density-based topology optimization

In this section, we briefly review the fundamental theories of density-based topology optimization that will be implemented using FEniTop in Sect. 3. Specifically, we focus on two classical design problems: compliance minimization and compliant mechanisms (Bendsøe and Kikuchi 1988; Bendsoe and Sigmund 2003). The compliance minimization problem can be formulated as

$$\begin{cases} \text{minimize : } C(\bar{\rho}, \mathbf{u}(\bar{\rho})) = \int_{\Omega} \boldsymbol{\sigma}(\bar{\rho}, \mathbf{u}(\bar{\rho})) : \boldsymbol{\epsilon}(\mathbf{u}(\bar{\rho})) d\mathbf{X}, \\ \text{subjected to : } V(\bar{\rho}) = \frac{1}{|\Omega|} \int_{\Omega} \bar{\rho} d\mathbf{X} - \bar{V} \leq 0, \\ \text{with : } \int_{\Omega} \boldsymbol{\sigma}(\bar{\rho}, \mathbf{u}(\bar{\rho})) : \boldsymbol{\epsilon}(\mathbf{v}) d\mathbf{X} = \int_{\Omega} \bar{\mathbf{b}} \cdot \mathbf{v} d\mathbf{X} \\ \quad + \int_{\partial\Omega_N} \bar{\mathbf{t}} \cdot \mathbf{v} d\mathbf{X}. \end{cases} \quad (1)$$

In this formulation,  $\rho \in [0, 1]$  represents the dimensionless design variable, which corresponds to the unfiltered density field, while  $C$  denotes the structural compliance aimed to be minimized. The variable  $\bar{\rho} \in [0, 1]$  signifies the physical density field, where  $\bar{\rho} = 1$  denotes solid material and  $\bar{\rho} = 0$  indicates void. The variable  $\mathbf{u}$  is the displacement field, and the symbol  $\Omega$  denotes the design domain. The variables  $\boldsymbol{\sigma}$  and  $\boldsymbol{\epsilon} = (\nabla \mathbf{u} + \nabla \mathbf{u}^T)/2$  stand for the infinitesimal stress and strain tensors, respectively. The symbol  $V$  represents the material volume constraint, with  $\bar{V} \in (0, 1]$  specifying the

allowable material volume fraction. The vector  $\mathbf{v}$  represents the test displacement field, and the variables  $\bar{\mathbf{b}}$  and  $\bar{\mathbf{t}}$  denote the applied body force defined in the domain ( $\Omega$ ) and the traction defined on the domain's Neumann boundary ( $\partial\Omega_N$ ), respectively.

As for the compliant mechanism problem, it is expressed as

$$\begin{cases} \text{minimize : } U(\mathbf{u}(\bar{\rho})) = \mathbf{U}(\mathbf{u}(\bar{\rho})) \cdot \mathbf{L}, \\ \text{subjected to : } C(\bar{\rho}, \mathbf{u}(\bar{\rho})) - \bar{C} \leq 0, \\ \quad V(\bar{\rho}) = \frac{1}{|\Omega|} \int_{\Omega} \bar{\rho} d\mathbf{X} - \bar{V} \leq 0, \\ \text{with : } \int_{\Omega} \boldsymbol{\sigma}(\bar{\rho}, \mathbf{u}(\bar{\rho})) : \boldsymbol{\epsilon}(\mathbf{v}) d\mathbf{X} = \int_{\Omega} \bar{\mathbf{b}} \cdot \mathbf{v} d\mathbf{X} \\ \quad + \int_{\partial\Omega_N} \bar{\mathbf{t}} \cdot \mathbf{v} d\mathbf{X}. \end{cases} \quad (2)$$

Here, the variable  $U$  represents the summation of the output displacement, while  $\mathbf{U}$  denotes the global displacement vector. The variable  $\mathbf{L}$  stands for a prescribed output displacement indicator, where  $L_i = 1$  signifies that the degree-of-freedom (DOF)  $i$  is controlled, and  $L_i = 0$  otherwise. The variable  $\bar{C}$  represents the upper bound of the structural compliance, while the remaining parameters are consistent with (1). We note that the compliance constraint introduced in (2) serves to ensure the minimum stiffness of the design and to avoid small hinges. It is primarily utilized for demonstrating multiple constraints and is optional for designing a compliant mechanism.

Further, the physical density field in (1) and (2),  $\bar{\rho}$ , can be obtained from the Heaviside projection (Wang et al. 2011), which is expressed as

$$\bar{\rho} = \frac{\tanh(\beta\theta) + \tanh(\beta(\tilde{\rho} - \theta))}{\tanh(\beta\theta) + \tanh(\beta(1 - \theta))} \quad (3)$$

where the coefficients  $\beta$  and  $\theta = 0.5$  are the sharpness parameter and the projection threshold value, respectively. The variable  $\tilde{\rho} \in [0, 1]$  in (3) is the filtered density field and can be further derived from the Helmholtz-type PDE filter (Lazarov and Sigmund 2011) with a homogeneous Neumann boundary condition. The weak form of this PDE filter can be expressed as

$$\int_{\Omega} \left( R^2 \nabla \tilde{\rho} \cdot \nabla v + \tilde{\rho} v \right) d\mathbf{X} = \int_{\Omega} \rho v d\mathbf{X}. \quad (4)$$

Here,  $R$  represents the filter radius, and  $v$  denotes the test function belonging to an appropriate function space.

In formulations (1) and (2), the infinitesimal stress tensor can be computed as (Hughes 2012)

$$\boldsymbol{\sigma}(\boldsymbol{\epsilon}(\mathbf{u})) = 2\mu\boldsymbol{\epsilon} + \lambda\text{tr}(\boldsymbol{\epsilon})\mathbf{I}, \quad (5)$$

where  $\lambda$  and  $\mu$  are Lamé constants and can be calculated as

$$\lambda(\bar{\rho}) = \frac{E(\bar{\rho})\nu(\bar{\rho})}{(1+\nu(\bar{\rho}))(1-2\nu(\bar{\rho}))} \quad \text{and} \quad \mu(\bar{\rho}) = \frac{E(\bar{\rho})}{2(1+\nu(\bar{\rho}))}. \quad (6)$$

The variables  $E$  and  $\nu$  in (6) are the Young's modulus and the Poisson's ratio interpolated by the modified Solid Isotropic Material with Penalization (SIMP) scheme in Sigmund (2007), which can be expressed as

$$E(\bar{\rho}) = E^0[\varepsilon + (1 - \varepsilon)\bar{\rho}^p] \quad \text{and} \quad \nu(\bar{\rho}) = \nu^0. \quad (7)$$

Here,  $E^0$  and  $\nu^0$  are Young's modulus and Poisson's ratio of a solid material, respectively. The parameter  $\varepsilon = 10^{-6}$  is a sufficiently small positive number to avoid the singularity of the global stiffness matrix when performing FEA. The parameter  $p = 3$  is used to penalize the intermediate physical density field (i.e.,  $\bar{\rho} \in (0, 1)$ ).

We note that the single projection scheme in (3) is employed solely for demonstration purposes and may not ensure the minimum length scale if the filter radius,  $R$ , is not sufficiently large. To guarantee the minimum length scale, the multiple-projection scheme outlined in Wang et al. (2011) can be utilized without incurring significant additional computational cost in the compliance minimization problem in (1). Furthermore, compared to classical methods such as Andreassen et al. (2011) and convolution-based approaches like Ferrari and Sigmund (2020), the PDE filter described in (4) is more suitable here. This preference arises because the PDE filter requires minimal information exchange across different processes, making it naturally well-suited for parallel computing environments. Additionally, the PDE filter necessitates less memory usage, particularly for large filter radii. Moreover, it can be easily adapted to mitigate artificial boundary effects, as discussed in Wallin et al. (2020).

### 3 FEniTop implementation for topology optimization

In this section, we present the numerical implementation of topology optimization in Sect. 2 by leveraging the FEniTop software. FEniTop comprises one input script and six modules (Fig. 2): `topopt` for the core topology optimization process, `parameterize` for design space parameterization, `fem` for the finite element analysis, `sensitivity` for the sensitivity analysis, `optimize` for the solution update with optimizer, and `utility` that offers useful auxiliary functions.

Generally, users only need to specify the topology optimization problem in the input script, and this script can

automatically import the related modules to perform topology optimization. In this section, we focus on introducing the input script and the two important modules: `topopt` and `fem`. The remaining modules, `parameterize`, `sensitivity`, and `optimize`, closely align with the algorithms in the literature that can be easily found. For example, the `parameterize` module implements the PDE filter in Lazarov and Sigmund (2011) and the Heaviside projection in Wang et al. (2011). The `sensitivity` module incorporates sensitivity analysis based on the adjoint method detailed in Bendsoe and Sigmund (2003), and the sensitivity analysis through automatic differentiation is introduced in Appendix A. Meanwhile, the `optimize` module furnishes the OC (Hassani and Hinton 1998) and MMA (Svanberg 1987) optimizers. As for the `utility` module, it provides auxiliary functions that typically remain unchanged, serving as an area for interested users to explore. We emphasize that all modules in FEniTop are implemented in parallel, facilitating rapid computational speed. In addition, due to its high modularity, FEniTop enables users to substitute specific modules with external packages if desired. The complete implementation of FEniTop is available in Appendix B.

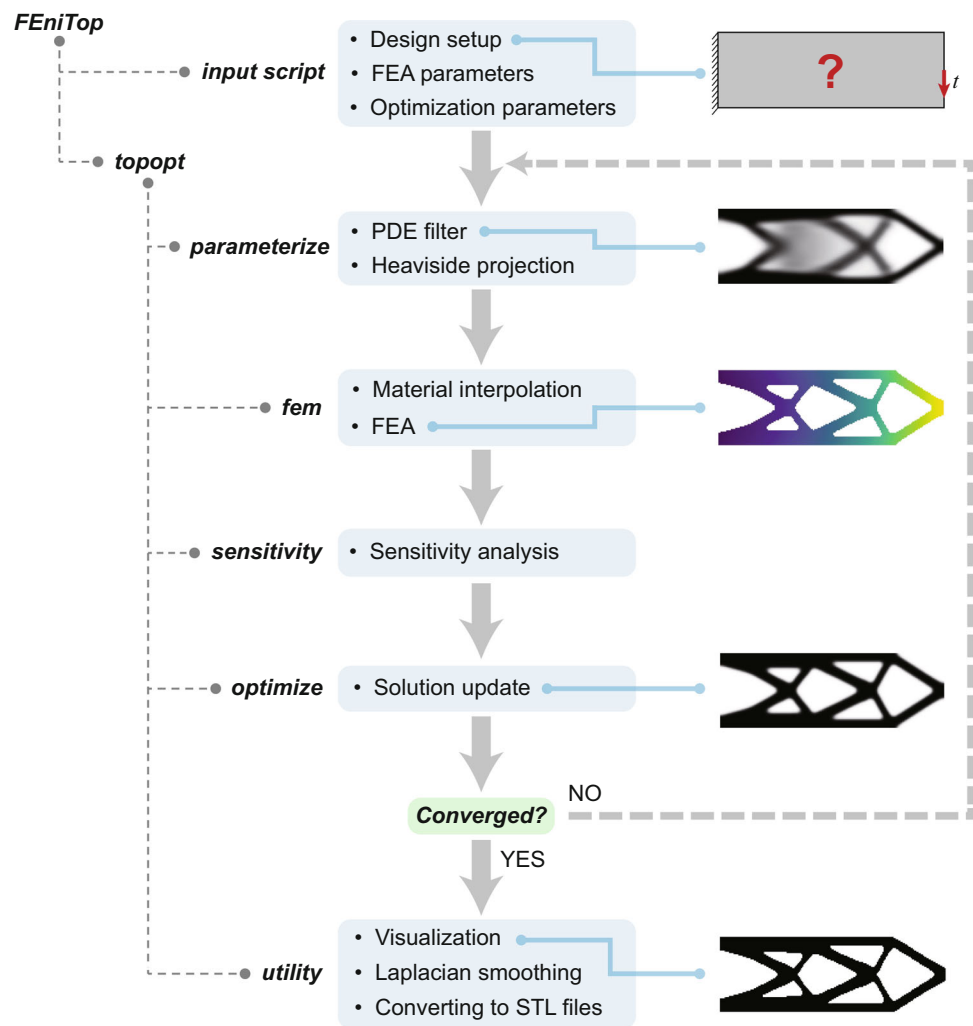
#### 3.1 Input script for defining a topology optimization problem

In the FEniTop package, we provide the users with five sample input scripts (`beam_2d.py`, `disk_2d.py`, `beam_3d.py`, `shell_3d.py`, and `mechanism_2d.py`) corresponding to the five examples in Sect. 4. In these input scripts, we define the topology optimization problem of interest with two dictionary variables, `fem` and `opt`, which specify the parameters for the FEA and topology optimization, respectively. Table 1 shows the keys, values, and explanations of the input dictionary variable `fem` by taking `beam_2d.py` as an example.

##### 3.1.1 Defining an FEA mesh

As shown in Table 1, we need to provide two FEniCSx meshes for the variables `fem["mesh"]` and `fem["mesh_serial"]`, respectively. The former is a parallel (distributed) mesh used for computation, and the latter is a serial mesh only defined in the root process and used for plotting the optimized design. These two meshes can be easily defined with the command as

**Fig. 2** Illustration of the FEniTop implementation for topology optimization



```

1 mesh = create_rectangle(MPI.COMM_WORLD, [[0, 0], [60, 20]],
2                               [200, 60],
3                               CellType.quadrilateral)
4 if MPI.COMM_WORLD.rank == 0:
5     mesh_serial = create_rectangle(
6         MPI.COMM_SELF, [[0, 0], [60, 20]],
7         [200, 60], CellType.quadrilateral)
8 else:
9     mesh_serial = None
  
```

Here `create_rectangle` represents creating a rectangular mesh. `MPI.COMM_WORLD` and `MPI.COMM_SELF` are communicators employed for parallel computing coordination. The argument `[[0, 0], [60, 20]]` specifies the starting `([0, 0])` and ending `([60, 20])` vertices of the rectangular domain. The argument `[200, 60]` determines the number of elements in  $x$ - and  $y$ -directions. `CellType.quadrilateral` indicates the geometry of finite elements. The FEniCSx library at the core of FEniTop offers a range of methods for adapting mesh and element geometries to meet diverse requirements.

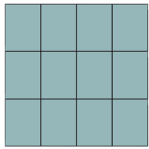
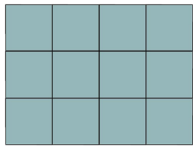
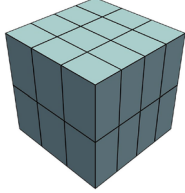
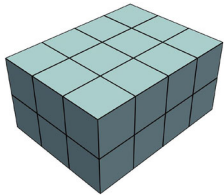
When dealing with structured meshes, users can seamlessly leverage the built-in functions provided by FEniCSx (Table 2). For unstructured meshes, a comprehensive discussion is available in Sect. 4.2.

One noteworthy feature of the FEniCSx library is its separation of mesh and element geometries. As shown in Table 3, 2D meshes offer support for both triangle and quadrilateral element geometries, whereas 3D meshes accommodate tetrahedron and hexahedron element geometries. We stress that in topology optimization, the choice of simplex elements such as triangle and tetrahedron may be preferable for certain irregular domains (Jia et al. 2024a). These elements can provide benefits such as achieving uniform finite element discretization, enabling the use of specialized element types, or facilitating local mesh refinement. However, in our current study, adhering to conventional topology optimization practices, we find it sufficient to employ quadrilateral and hexahedron element geometries when defining the mesh.

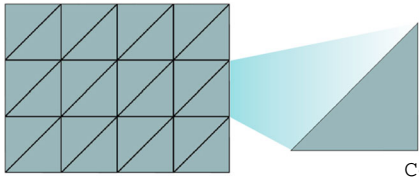
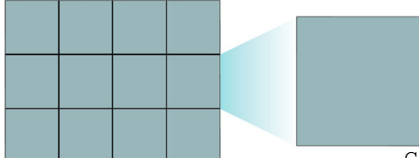
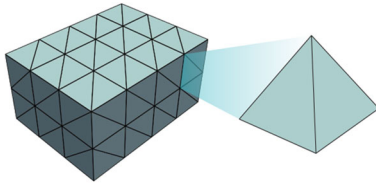
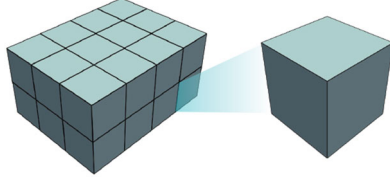
**Table 1** Keys, values, and explanations of the input dictionary variable `fem`

| Keys                | Values          | Explanations                                                                                          |
|---------------------|-----------------|-------------------------------------------------------------------------------------------------------|
| "mesh"              | FEniCSx mesh    | Parallel mesh used for computation                                                                    |
| "mesh_serial"       | FEniCSx mesh    | Serial mesh used for plotting the design                                                              |
| "young's modulus"   | 100             | Young's modulus of the solid material ( $E^0$ )                                                       |
| "poisson's ratio"   | 0.25            | Poisson's ratio of the solid material ( $\nu^0$ )                                                     |
| "disp_bc"           | lambda function | Locator of displacement boundaries ( $\partial\Omega_D = \partial\Omega \setminus \partial\Omega_N$ ) |
| "traction_bcs"      | List of Lists   | Traction values ( $\bar{\mathbf{t}}$ ) and locators of traction boundaries ( $\partial\Omega_N$ )     |
| "body_force"        | (0, 0)          | Prescribed body force ( $\bar{\mathbf{b}}$ )                                                          |
| "quadrature_degree" | 2               | Quadrature degree used in numerical integration                                                       |
| "petsc_options"     | dictionary      | Information of the linear solver and preconditioner                                                   |

**Table 2** Mesh geometries supported by FEniTop through the FEniCSx library

| Design domains                                                                      | Commands                                                                                    | Descriptions                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | <code>create_unit_square(MPI.COMM_WORLD, nx, ny, cell_type)</code>                          | <ul style="list-style-type: none"> <li>• Create a 2D unit square</li> <li>• nx, ny: the element numbers in x- and y-directions, respectively</li> <li>• cell_type: the element geometry provided in Table 3</li> </ul>                                                                                    |
|   | <code>create_rectangle(MPI.COMM_WORLD, [[0, 0], [lx, ly]], [nx, ny], cell_type)</code>      | <ul style="list-style-type: none"> <li>• Create a 2D rectangle</li> <li>• lx, ly: the edge lengths in x- and y-directions, respectively</li> <li>• nx, ny: the element numbers in x- and y-directions, respectively</li> <li>• cell_type: the element geometry provided in Table 3</li> </ul>             |
|  | <code>create_unit_cube(MPI.COMM_WORLD, nx, ny, nz, cell_type)</code>                        | <ul style="list-style-type: none"> <li>• Create a 3D unit cube</li> <li>• nx, ny, nz: the element numbers in x-, y-, and z-directions, respectively</li> <li>• cell_type: the element geometry provided in Table 3</li> </ul>                                                                             |
|  | <code>create_box(MPI.COMM_WORLD, [[0, 0, 0], [lx, ly, lz]], [nx, ny, nz], cell_type)</code> | <ul style="list-style-type: none"> <li>• Create a 3D box</li> <li>• lx, ly, lz: the edge lengths in x-, y-, and z-directions, respectively</li> <li>• nx, ny, nz: the element numbers in x-, y-, and z-directions, respectively</li> <li>• cell_type: the element geometry provided in Table 3</li> </ul> |

**Table 3** Element geometries supported by FEniTop through the FEniCSx library

| Element geometries                                                                | Commands                            | Descriptions          |
|-----------------------------------------------------------------------------------|-------------------------------------|-----------------------|
|  | <code>CellType.triangle</code>      | Triangular element    |
|  | <code>CellType.quadrilateral</code> | Quadrilateral element |
|  | <code>CellType.tetrahedron</code>   | Tetrahedral element   |
|  | <code>CellType.hexahedron</code>    | Hexahedral element    |

### 3.1.2 Defining material properties and boundary conditions

Upon the definition of a FEniCSx mesh, the subsequent step involves specifying material properties, denoted as  $E^0$  and  $\nu^0$  in (7), for solid materials. Following this step, we define the boundary conditions for FEA as follows. As for the displacement boundary condition, we locate the target boundaries ( $\partial\Omega_D = \partial\Omega \setminus \partial\Omega_N$ ) by defining a lambda function in `fem["disp_bc"]` as

```
1 lambda x: np.isclose(x[0], 0)
```

This function identifies all nodes within a given FEniCSx mesh that meet the specified condition ( $x$ -coordinates are equal to zero herein). In this command, `x[0]` represents the  $x$ -coordinates of all nodes. Similarly, we can use `x[1]` and `x[2]` to represent  $y$ - and  $z$ -coordinates of all nodes, respectively. For simplicity, this work exclusively considers displacement boundaries with zero values, akin to pins and rollers.

For traction boundary conditions, we use a nested List structure within the variable `fem["traction_bcs"]`, which is organized as

```
1 [[Traction value 0, Locator 0], [
   Traction value 1, Locator 1], ...]
```

In this command, each inner List contains a traction value (represented as a tuple of two scalars in 2D and three

scalars in 3D) and the corresponding traction boundary segment, designated as a lambda function. All this traction boundary information is encapsulated within the outer List. This structure allows us to specify different traction values on distinct parts of the traction boundaries ( $\partial\Omega_N$ ), as detailed in Sect. 4.4. In the current example, we define a single traction boundary condition as

```
1 [(0, -0.02), lambda x: (np.isclose(x[0], 60) & np.greater(x[1], 8)
2                               & np.less(x[1], 12)))]
```

Here, we prescribe the traction value  $(0, -0.02)$  to be applied on the boundary segment of  $\{(x, y) | x = 60 \text{ and } y \in (8, 12)\}$  with  $x$  and  $y$  representing nodal coordinates. Next, for the body force, we define it as a tuple in `fem["body_force"]`, which will be applied to the entire design domain ( $\Omega$ ). In this case, we set `fem["body_force"]` to  $(0, 0)$  to indicate a scenario without any applied body force.

### 3.1.3 Defining the quadrature degree and solver configurations

Once the boundary conditions are defined, we can move forward to specify the quadrature degree for numerical integration in the variable `fem["quadrature_degree"]`.

The last piece of information in the fem configuration is the "petsc\_options" variable, which is of type dictionary and provides details about the linear solver and preconditioner settings. This information is used with the PETSc backend for solving linear system equations in PDE filtering, FEA, and sensitivity analysis. By default, we define "petsc\_options" as

```
1 { "ksp_type": "cg", "pc_type": "gamg"
  }
```

In this command, we specify the linear solver as the preconditioned conjugate gradient (CG) method (Hestenes and Stiefel 1952) by setting the "ksp\_type" option as "cg". Additionally, we specify the geometric algebraic multigrid (AMG) preconditioner (Stüben 2001) by setting the "pc\_type" option as "gamg". See more in-depth discussion about the solver configurations in Sect. 4.3.

After defining the dictionary variable fem for FEA, we proceed to define another important dictionary variable, opt, for topology optimization. Table 4 shows the keys, values, and explanations of the input dictionary variable opt by taking beam\_2d.py as an example. Notably, the opt variable specifies the passive solid and void regions with lambda functions. Additionally, it determines which solver (OC or MMA) will be utilized and which topology optimization problem (compliance minimization or compliant mechanism) will be solved.

After defining dictionary variables fem and opt, the input script can automatically initiate topology optimization by calling the toptopt module with the command as

```
1 if __name__ == "__main__":
2     toptopt(fem, opt)
```

Here \_\_name\_\_ == "\_\_main\_\_" serves as the entry point of the code, and the toptopt(fem, opt) module is invoked to execute the topology optimization process. Detailed information about the toptopt module is provided in Sect. 3.2. The separation of the input script and the toptopt module enables the users to exploit the same modules to address diverse topology optimization problems by simply adjusting the input script.

### 3.2 toptopt module for topology optimization

The toptopt module serves as the central component for conducting topology optimization and is invoked by the pre-defined input script. This module further imports and reconciles the remaining five modules (parameterize, fem, sensitivity, optimize, and utility) in the FEniTop software. In this subsection, we present the implementation of the toptopt module consisting of initialization and body parts as follows.

#### 3.2.1 Initialization

Prior to the topology optimization iterations, several initialization steps need to be undertaken as follows. We begin by invoking the form\_fem function to substantiate a class variable linear\_problem, which will be used later for repeatedly solving the equilibrium equation in FEA and the adjoint equation in sensitivity analysis. We also define several important field variables, the displacement field (u\_field), adjoint variable field (lambda\_field), unfiltered density field (rho\_field), and physical density field (rho\_phys\_field). Similarly, we invoke the DensityFilter, Heaviside, and Sensitivity functions to define three class variables, density\_filter for the PDE filter in (4), heaviside for the Heaviside projection in (3), and sens\_problem for sensitivity analysis, respectively. Next, we initialize two variables, S\_comm and plotter, with the Communicator and Plotter functions, respectively. These two functions will be used to plot the optimized design. Finally, we compute the number of constraint functions (num\_consts) as well as the number of finite elements in the current process (num\_elems). Additionally, we initialize several quantities (rho\_old1, rho\_old2, low, and upp) used for the MMA optimizer. The corresponding commands are as follows.

```
1 # Initialization
2 comm = MPI.COMM_WORLD
3 linear_problem, u_field,
4 lambda_field, rho_field,
5 rho_phys_field = form_fem(fem, opt)
6 density_filter = DensityFilter(comm,
7 rho_field, rho_phys_field, opt["
8 filter_radius"], fem["petsc_options
9 "])
10 heaviside = Heaviside(rho_phys_field)
11 sens_problem = Sensitivity(comm, opt,
12 linear_problem, u_field,
13 lambda_field, rho_phys_field)
14 S_comm = Communicator(rho_phys_field
15 .function_space, fem["mesh_serial"
16 ])
17 if comm.rank == 0:
18     plotter = Plotter(fem["
19 mesh_serial"])
20 num_consts = 1 if opt["
21 opt_compliance"] else 2
22 num_elems = rho_field.vector.array.
23 size
24 if not opt["use_oc"]:
25     rho_old1, rho_old2, = np.zeros(
26 num_elems), np.zeros(num_elems)
27 low, upp = None, None
```

Subsequently, we apply the passive solid and void regions to the design domain. This step involves computing the element centers (centers) and identifying the solid and void

**Table 4** Keys, values, and explanations of the input dictionary variable `opt`

| Keys             | Values          | Explanations                                                                                 |
|------------------|-----------------|----------------------------------------------------------------------------------------------|
| "max_iter"       | 400             | Maximum optimization iterations ( $n^{\max}$ )                                               |
| "opt_tol"        | 1e-5            | Allowable tolerance of the maximum change of design variables ( $\varepsilon^{\text{opt}}$ ) |
| "vol_frac"       | 0.5             | Upper bound of the material volume fraction ( $\bar{V}$ )                                    |
| "solid_zone"     | lambda function | Locator of passive solid regions                                                             |
| "void_zone"      | lambda function | Locator of passive void regions                                                              |
| "penalty"        | 3.0             | Penalty parameter ( $p$ ) used in the modified SIMP rule in (7)                              |
| "epsilon"        | 1e-6            | Sufficiently small positive number ( $\varepsilon$ ) used in the modified SIMP rule in (7)   |
| "filter_radius"  | 1.2             | Radius ( $R$ ) in the PDE filter in (4)                                                      |
| "beta_interval"  | 50              | Interval for doubling the Heaviside sharpness parameter ( $\beta^{\text{int}}$ )             |
| "beta_max"       | 128             | Maximum value of the Heaviside sharpness parameter ( $\beta^{\max}$ )                        |
| "use_oc"         | True or False   | Use the OC (True) or MMA (False) optimizer                                                   |
| "move"           | 0.05            | Move limit ( $m$ ) of the design variable                                                    |
| "opt_compliance" | True or False   | Optimize compliance (True) or the output displacement (False)                                |

elements (solid and void, respectively). We then assign the initial values (`rho_ini`) and bounds (`rho_min` and `rho_max`) of the unfiltered density field (`rho_field`). The corresponding commands are as follows.

```

1  # Apply passive zones
2  centers = rho_field.function_space.
   tabulate_dof_coordinates()[ :
   num_elems].T
3  solid, void = opt["solid_zone"](
   centers), opt["void_zone"](centers)
4  rho_ini = np.full(num_elems, opt["
   vol_frac"])
5  rho_ini[solid], rho_ini[void] =
   0.995, 0.005
6  rho_field.vector.array[:] = rho_ini
7  rho_min, rho_max = np.zeros(
   num_elems), np.ones(num_elems)
8  rho_min[solid], rho_max[void] =
   0.99, 0.01

```

we apply the Heaviside projection in (3) to update the physical density field (`rho_phys_field`) by utilizing the `heaviside.forward(beta)` function. The corresponding commands are as follows.

```

1  # Start topology optimization
2  opt_iter, beta, change = 0, 1, 2*opt
   ["opt_tol"]
3  while opt_iter < opt["max_iter"] and
   change > opt["opt_tol"]:
4      opt_start_time = time.
   perf_counter()
5      opt_iter += 1
6
7      # Density filter and Heaviside
   projection
8      density_filter.forward()
9      if opt_iter % opt["beta_interval"]
   == 0 and beta < opt["beta_max"]:
10         beta *= 2
11         change = opt["opt_tol"] * 2
12     heaviside.forward(beta)

```

### 3.2.2 Body parts of topology optimization

Following the necessary initialization steps, we proceed to the body parts of topology optimization presented in a `while`-loop. The body parts consist of density filter and Heaviside projection, FEA and sensitivity analysis, as well as design variable update and post-processing, which are laid out as follows.

**Density filter and Heaviside projection** The first segment within this `while`-loop entails the application of the density filter and Heaviside projection. Here, we apply the PDE filter in (4) on the unfiltered density field (`rho_field`) by invoking the `density_filter.forward()` function. After necessary updates of the Heaviside sharpness parameter (`beta`) and the change of design variables (`change`),

**FEA and sensitivity analysis** After implementing the density filter and Heaviside projection, we perform the FEA with a simple command as

```

1  # Solve FEM
2  linear_problem.solve_fem()

```

Subsequently, we proceed to evaluate the values and sensitivities of the objective and constraint functions. We begin by invoking the `sens_problem.evaluate()` function to compute the function values, including compliance (`C_value`), volume (`V_value`), and output displacement (`U_value`), as well as the sensitivities with respect to the physical density field (`sensitivities`). Next, we compute the sensitivities concerning the unfiltered density field (`dCdrho`, `dVdrho`, and `dUdrho`) by applying chain

rules to (3) and (4). This step is done by invoking the `heaviside.backward(sensitivities)` and `density_filter.backward(sensitivities)` functions. Finally, we rearrange the computed quantities into the constraint function values (`g_vec`) as well as the sensitivities of the objective (`dJdrho`) and constraint (`dgdrho`) functions. The corresponding commands are as follows.

```

1  # Compute function values and
2  sensitivities
3  [C_value, V_value, U_value],
4  sensitivities = sens_problem.
5  evaluate()
6  heaviside.backward(sensitivities)
7  [dCdrho, dVdrho, dUdrho] =
8  density_filter.backward(
9  sensitivities)
10 if opt["opt_compliance"]:
11     g_vec = np.array([V_value -
12     opt["vol_frac"]])
13     dJdrho, dgdrho = dCdrho, np.
14     vstack([dVdrho])
15 else:
16     g_vec = np.array([V_value -
17     opt["vol_frac"], C_value - opt["
18     compliance_bound"]])
19     dJdrho, dgdrho = dUdrho, np.
20     vstack([dVdrho, dCdrho])

```

**Design variable update and post-processing** Following the sensitivity analysis, we proceed to employ the OC (optimality\_criteria) or MMA (mma\_optimizer) optimizer to compute the updated values (`rho_new`) and the maximum change (`change`) of the design variable. After that, the information of the current optimization iteration will be output. The corresponding commands are as follows.

```

1  # Update the design variables
2  rho_values = rho_field.vector.
3  array.copy()
4  if opt["opt_compliance"] and opt
5  ["use_oc"]:
6      rho_new, change =
7      optimality_criteria(
8      rho_values, rho_min,
9      rho_max, g_vec, dJdrho, dgdrho[0],
10     opt["move"])
11 else:
12     rho_new, change, low, upp =
13     mma_optimizer(
14     num_consts, num_elems,
15     opt_iter, rho_values, rho_min,
16     rho_max,
17     rho_old1, rho_old2,
18     dJdrho, g_vec, dgdrho, low, upp,
19     opt["move"])
20     rho_old2 = rho_old1.copy()
21     rho_old1 = rho_values.copy()
22     rho_field.vector.array = rho_new
23     .copy()

```

```

14 # Output the histories
15 opt_time = time.perf_counter() -
16 opt_start_time
17 if comm.rank == 0:
18     print(f"opt_iter: {opt_iter
19     }, opt_time: {opt_time:.3g} (s), "
20     f"beta: {beta}, C: {
21     C_value:.3f}, V: {V_value:.3f}, "
22     f"U: {U_value:.3f},
23     change: {change:.3f}", flush=True)

```

The procedures outlined within the while-loop will be repeated until the maximum iteration is reached or the design converges. Upon obtaining a converged design, we use the `S_comm.gather` function to gather the distributed physical design variables from all processes to the root process and use the `plotter.plot` function to plot the optimized design. Alternatively, we can invoke the `save_xdmf` function to directly save the optimized design in parallel with a simple command given as

```

1  values = S_comm.gather(
2  rho_phys_field)
3  if comm.rank == 0:
4      plotter.plot(values)
5  save_xdmf(fem["mesh"],
6  rho_phys_field)

```

The optimized design is saved as an XDMF file, which can be seamlessly imported into ParaView software for post-processing tasks such as visualization, filtering low-density regions, Laplacian smoothing, and conversion to STL files for 3D printing.

### 3.3 fem module for constructing an FEA problem

The `fem` module is designed to construct linear problems that will be repeatedly solved in each optimization iteration. In this module, we also define several pertinent quantities for the sensitivity analysis of topology optimization. The basic components of the `fem` module are as follows.

#### 3.3.1 Defining function spaces and functions

In the `fem` module, we start by defining necessary function spaces and functions (field variables). Specifically, we define a vector function space, `V`, for the displacement field. Additionally, we define two scalar function spaces, `S0` and `S`, for the unfiltered and physical density fields, respectively. Here, ("CG", 1) denotes the first-order Lagrange finite element with DOFs defined at the element vertices, while ("DG", 0) represents the discontinuous Lagrange finite element (element-wise constants). Based on the specified function spaces, we define the trial (`u`) and test (`v`) functions of the displacement field for solving linear system equations. Furthermore, we define four crucial field variables: the displacement field (`u_field`), adjoint variable field

(`lambda_field`), unfiltered density field (`rho_field`), and physical density field (`rho_phys_field`). The corresponding commands are as follows.

```
1 def form_fem(fem, opt):
2     """Form an FEA problem."""
3     # Function spaces and functions
4     mesh = fem["mesh"]
5     V = VectorFunctionSpace(mesh, ("CG",
6     1))
7     S0 = FunctionSpace(mesh, ("DG", 0))
8     S = FunctionSpace(mesh, ("CG", 1))
9     u, v = ufl.TrialFunction(V), ufl.
10    TestFunction(V)
11    u_field = Function(V) #
12    Displacement field
13    lambda_field = Function(V) #
14    Adjoint variable field
15    rho_field = Function(S0) # Density
16    field
17    rho_phys_field = Function(S) #
18    Physical density field
```

We highlight that thanks to the separation of the element geometries and element orders within the FEniCSx library, the users can conveniently access higher-order elements through a simple modification such as

```
1 V = VectorFunctionSpace(mesh, ("CG",
2     2))
```

In this command, we effortlessly employ the second-order Lagrange elements by substituting ("CG", 1) with ("CG", 2), and the remaining codes keep unchanged. Furthermore, the FEniCSx library also offers extensive support for a diverse range of element types as documented in the link.<sup>1</sup> Users can readily access these distinct element types by executing a command such as

```
1 V = VectorFunctionSpace(mesh, ("CR",
2     1))
```

Within this command, the adoption of the Crouzeix–Raviart finite element merely entails replacing ("CG", 1) with ("CR", 1), while preserving the entirety of the remaining code unchanged. For the current linear elastostatics problem, it suffices to employ the first-order Lagrange finite element, denoted as ("CG", 1).

### 3.3.2 Defining interpolated material properties

Based on the interpolation rules in (6) and (7), we define the interpolated material properties with the commands as

```
1 # Material interpolation
2 E0, nu = fem["young's modulus"], fem
3 ["poisson's ratio"]
4 p, eps = opt["penalty"], opt["
5     epsilon"]
```

<sup>1</sup> <https://docs.fenicsproject.org/basix/main>.

```
4 E = (eps + (1-eps)*rho_phys_field**p
5     ) * E0
6 _lambda, mu = E*nu/(1+nu)/(1-2*nu),
7 E/(2*(1+nu)) # Lamé constants
```

We highlight that the values of these interpolated material properties (`_lambda` and `mu`) can be automatically computed once updates are made to the values of the physical density field (`rho_phys_field`).

### 3.3.3 Defining kinematic quantities

We define the infinitesimal strain (epsilon) and stress (sigma) tensors with the commands as

```
1 # Kinematics
2 def epsilon(u):
3     return ufl.sym(ufl.grad(u))
4
5 def sigma(u): # 3D or plane strain
6     return 2*mu*epsilon(u) + _lambda
7     *ufl.tr(epsilon(u))*ufl.Identity(
8     len(u))
```

Notably, the programming syntax employed in these commands closely mirrors their mathematical representations in (5). This alignment between the code and the mathematical formulation is an important feature offered by the underlying FEniCSx library of FEniTop, and this feature streamlines the user's focus towards the core aspects of the physical problem while abstracting away intricate programming details.

### 3.3.4 Setting the boundary conditions

We define the displacement (Dirichlet) boundary condition as follows. Initially, we identify the mesh dimension as `dim` and the facet dimension as `fdim`, where a facet corresponds to an edge in 2D and a face in 3D. Given the FEniCSx mesh (`mesh`), facet dimension (`fdim`) and displacement boundary locator (`fem["disp_bc"]`), we can determine the controlled facet as `disp_facets` with the `locate_entities_boundary` function. Finally, we construct an object (`bc`) for specifying the displacement boundary conditions through the `dirichletbc` function with three inputs: the boundary condition values (we set  $\bar{\mathbf{u}} = \mathbf{0}$  as previously mentioned), the controlled DOFs, and the function space associated with the displacement field.

```
1 # Boundary conditions
2 dim = mesh.topology.dim
3 fdim = dim - 1
4 disp_facets =
5     locate_entities_boundary(mesh, fdim,
6     fem["disp_bc"])
7 bc = dirichletbc(Constant(mesh, np.
8     full(dim, 0.0)),
9     locate_dofs_topological(V, fdim,
10    disp_facets), V)
```

Subsequently, we establish the traction (Neumann) boundary conditions with the following commands.

```

1  tractions, facets, markers = [], [],
   []
2  for marker, (traction, traction_bc)
   in enumerate(fem["traction_bcs"]):
3      tractions.append(Constant(mesh,
4      np.array(traction, dtype=float)))
5      current_facets =
   locate_entities_boundary(mesh, fdim,
6      traction_bc)
7      facets.extend(current_facets)
8      markers.extend([marker,]*len(
   current_facets))
9      facets = np.array(facets, dtype=np.
   int32)
10     markers = np.array(markers, dtype=np.
   int32)
11     _, unique_indices = np.unique(facets
   , return_index=True)
12     facets, markers = facets[
   unique_indices], markers[
   unique_indices]
13     sorted_indices = np.argsort(facets)
14     facet_tags = meshtags(mesh, fdim,
   facets[sorted_indices], markers[
   sorted_indices])
15
16     metadata = {"quadrature_degree": fem
   ["quadrature_degree"]}
17     dx = ufl.Measure("dx", metadata=
   metadata)
18     ds = ufl.Measure("ds", domain=mesh,
   metadata=metadata, subdomain_data=
   facet_tags)
19     b = Constant(mesh, np.array(fem["
   body_force"], dtype=float))

```

We initiate this process by iterating through the provided traction boundaries (`fem["traction_bcs"]`) using a for-loop. During this iteration, we collect information pertaining to the markers (markers), traction values (tractions), and controlled facets (facets). Here, facets with identical traction values are assigned the same marker for clarity. Next, we integrate this gathered information into the `facet_tags` variable. Additionally, we define a metadata variable to specify the quadrature degree for numerical integration. Based on the defined variables, `facet_tags` and `metadata`, we construct the differential variables for the domain (`dx`) and boundary (`ds`). Finally, the body force is represented as a Constant variable denoted as `b`.

### 3.3.5 Establishing an FEA problem

Built on the defined boundary conditions, we establish the equilibrium equation and formulate an FEA problem with the commands as

```

1  # Establish the equilibrium and
   adjoint equations

```

```

2  lhs = ufl.inner(sigma(u), epsilon(v))
   *dx
3  rhs = ufl.dot(b, v)*dx
4  for marker, t in enumerate(tractions)
   ):
5      rhs += ufl.dot(t, v)*ds(marker)
6  if opt["opt_compliance"]:
7      spring_vec = opt["l_vec"] = None
8  else:
9      spring_vec, opt["l_vec"] =
   create_mechanism_vectors(
10     V, opt["in_spring"], opt["
   out_spring"])
11  linear_problem = LinearProblem(
   u_field, lambda_field, lhs, rhs,
   opt["l_vec"], spring_vec, [bc], fem
   ["petsc_options"])

```

Here we start by defining the left-hand side (lhs) and the right-hand side (rhs) of the equilibrium equation in FEA. Thanks to the FEniCSx library, the programming of the equilibrium equation with the weak form is similar to its mathematical expression in (1) and (2). Additionally, we define the spring stiffness (`spring_vec`) and the displacement indicator (`opt["l_vec"]`) vectors, which will be utilized in the compliant mechanism example in Sect. 4.5. With the defined variables, we can initialize `linear_problem` to solve the linear system equations in FEA and sensitivity analysis.

### 3.3.6 Defining other quantities related to topology optimization

In the remainder of the `fem` module, we define several quantities of interest for topology optimization. Specifically, at the weak form level, we define the internal force (`opt["f_int"]`), structural compliance (`opt["compliance"]`), actual material volume (`opt["volume"]`), and the total material volume (`opt["total_volume"]`). The code snippet is as follows.

```

1  # Define optimization-related
   variables
2  opt["f_int"] = ufl.inner(sigma(
   u_field), epsilon(v))*dx
3  opt["compliance"] = ufl.inner(sigma(
   u_field), epsilon(u_field))*dx
4  opt["volume"] = rho_phys_field*dx
5  opt["total_volume"] = Constant(mesh,
   1.0)*dx

```

Utilizing the input script and modules introduced previously, FEniTop offers a powerful and convenient framework for defining a range of topology optimization problems with various configurations. Moreover, FEniTop harnesses parallel computation capabilities to solve these optimization problems efficiently (see performance tests in Sect. 4). This combination of flexibility and parallel computing makes

FEniTop a promising tool for tackling complex topology optimization tasks.

## 4 Sample examples and performance investigation

In this section, we demonstrate the exploitation of FEniTop in addressing the benchmark compliance minimization problem in (1) and the complaint mechanism problem in (2) through the examination of five representative examples as summarized in Table 5. We also employ these examples to investigate the computational efficiency of FEniTop across various computing platforms, process numbers, problem sizes, and solver configurations. Specifically, Sect. 4.1 studies a 2D cantilever beam example with a structured mesh, where we evaluate the computational time and speedup of FEniTop for various process numbers on diverse computing platforms (see detailed tested configurations in Table 6). Next, Sect. 4.2 generalizes the topology optimization problem to a 2D disk domain with an unstructured mesh. Subsequently, Sect. 4.3 transitions from 2D to 3D topology optimization, focusing on a spatial cantilever beam. Within this context, we provide a detailed examination of the utilization of different solvers and preconditioning techniques offered by the PETSc library integrated into the FEniTop software. After that, Sect. 4.4 showcases topology optimization for a 3D spherical shell with an unstructured mesh. Here, we not only demonstrate the implementation of intricate boundary conditions but also demonstrate the post-processing capabilities enriched by ParaView software. Finally, Sect. 4.5 illustrates leveraging FEniTop to address the complex compliant mechanism design problem. This problem entails applying passive solid and void zones, incorporating artificial springs, sensitivity analysis of non-self-adjoint functions, and managing multiple constraints using the MMA optimizer.

In this section, we maintain consistent FEA and topology optimization parameters for all examples unless stated otherwise. Specifically, we set Young's modulus and Poisson's ratio of the solid material as  $E^0 = 100 \text{ N/mm}^2$  and  $\nu^0 = 0.25$ , respectively. We vanish the body force as  $\bar{\mathbf{b}} = \mathbf{0}$  and set the quadrature degree as 2 for the numerical integration. We set the maximum number of optimization iterations as  $n^{\max} = 400$  and the tolerance for changes in design variables as  $\varepsilon^{\text{opt}} = 10^{-5}$ . As for the material property interpolation rules in (7), we set the penalty parameter as  $p = 3$  and the sufficiently small positive parameter as  $\varepsilon = 10^{-6}$ . In addition, we define the interval for doubling the Heaviside sharpness parameter as  $\beta^{\text{int}} = 50$ , while the maximum sharpness parameter is constrained to  $\beta^{\max} = 128$ .

Moreover, in conducting all performance tests within this section, we leverage the FEniTop software built upon

FEniCSx of version 0.7.3 and employ version 3.10.12 in the 64-bit variant for the Python interpreter. To access the FEniTop software on the laptop, desktop, and workstation in Table 6, we use the Docker Desktop software (Anderson 2015) to substantiate the pertinent containers and execute FEniTop inside these containers within the Windows Subsystem for Linux (WSL2). To enable the FEniTop on the cluster in Table 6, we utilize the Singularity software (Kurtzer et al. 2017) by following a similar procedure. For performance assessment, we rely on the Python built-in module `time.perf_counter()` to measure computational time and utilize the `psutil` package<sup>2</sup> to gauge memory usage.

### 4.1 A simple demonstration and performance tests

In this subsection, we employ the FEniTop software to maximize the structural stiffness of a 2D cantilever beam, as illustrated in Fig. 3. Figure 3a provides an overview of the design domain and boundary conditions. Specifically, we fix the left edge of the beam while applying a downward traction loading, denoted as  $\bar{\mathbf{t}} = (0, -0.2)$  in units of  $\text{N/mm}^2$ . To conduct FEA and topology optimization, we discretize the design domain using a structured mesh composed of first-order quadrilateral Lagrange finite elements. In addition, we utilize an iterative solver, CG (Hestenes and Stiefel 1952), with the AMG preconditioner (Stüben 2001), facilitated by the PETSc library. The solver configurations are explicitly defined as

```
1  "petsc_options": {
2      "ksp_type": "cg",
3      "pc_type": "gamg",
4  }
```

These configurations are specified within the input variable `fem`. Concerning topology optimization parameters, we set the upper bound for the material volume fraction as  $\bar{V} = 0.5$  and establish a density filter radius of  $R = 1.2 \text{ mm}$ . Additionally, we impose a move limit of  $m = 0.05$  in the OC optimizer.

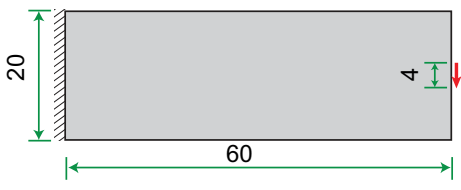
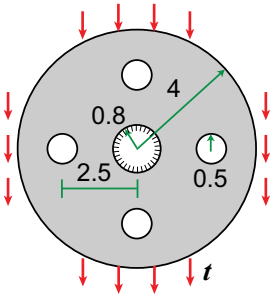
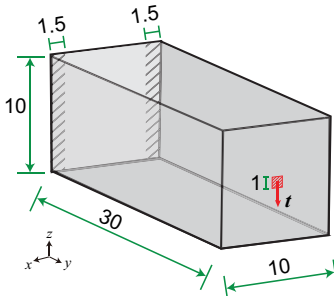
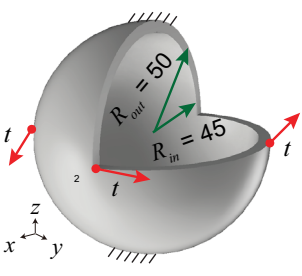
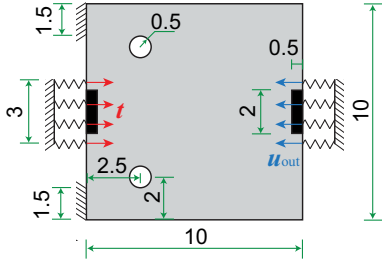
To initiate the topology optimization process, we simply need to execute the following command in the terminal as

```
1  mpirun -n 8 python3 scripts/beam_2d.py
```

Here, the `mpirun` command is utilized for program execution, either in serial or parallel, using the Message Passing Interface (MPI) (Dalcín et al. 2005). The `-n 8` argument specifies the number of processes (8 processes are invoked herein), with `-n 1` denoting serial execution. Typically, to ensure optimal computational speed, the number of processes employed should not exceed the number of physical processors. Finally, the command `python3`

<sup>2</sup> Available at <https://github.com/giampaolo/psutil>.

**Table 5** Sample examples provided by FEniTop

| Design domains                                                                      | Commands                                               | Descriptions                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------------------------------------------------------------------------|--------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | <pre>mpirun -n 8 python3 scripts/beam_2d.py</pre>      | <ul style="list-style-type: none"> <li>• <math>E = 100 \text{ MPa}</math>, <math>\nu = 0.25</math></li> <li>• <math>n^{\max} = 400</math>, <math>\varepsilon^{\text{opt}} = 10^{-5}</math></li> <li>• <math>\bar{V} = 0.5</math>, <math>R = 1.2 \text{ mm}</math></li> <li>• <math>\beta^{\text{int}} = 50</math>, <math>\beta^{\max} = 128</math></li> <li>• <math>p = 3</math>, <math>\varepsilon = 10^{-6}</math>, <math>m = 0.05</math></li> </ul>  |
|    | <pre>mpirun -n 8 python3 scripts/disk_2d.py</pre>      | <ul style="list-style-type: none"> <li>• <math>E = 100 \text{ MPa}</math>, <math>\nu = 0.25</math></li> <li>• <math>n^{\max} = 400</math>, <math>\varepsilon^{\text{opt}} = 10^{-5}</math></li> <li>• <math>\bar{V} = 0.5</math>, <math>R = 0.32 \text{ mm}</math></li> <li>• <math>\beta^{\text{int}} = 30</math>, <math>\beta^{\max} = 128</math></li> <li>• <math>p = 3</math>, <math>\varepsilon = 10^{-6}</math>, <math>m = 0.02</math></li> </ul> |
|   | <pre>mpirun -n 8 python3 scripts/beam_3d.py</pre>      | <ul style="list-style-type: none"> <li>• <math>E = 100 \text{ MPa}</math>, <math>\nu = 0.25</math></li> <li>• <math>n^{\max} = 400</math>, <math>\varepsilon^{\text{opt}} = 10^{-5}</math></li> <li>• <math>\bar{V} = 0.08</math>, <math>R = 0.6 \text{ mm}</math></li> <li>• <math>\beta^{\text{int}} = 50</math>, <math>\beta^{\max} = 128</math></li> <li>• <math>p = 3</math>, <math>\varepsilon = 10^{-6}</math>, <math>m = 0.02</math></li> </ul> |
|  | <pre>mpirun -n 8 python3 scripts/shell_3d.py</pre>     | <ul style="list-style-type: none"> <li>• <math>E = 100 \text{ MPa}</math>, <math>\nu = 0.25</math></li> <li>• <math>n^{\max} = 400</math>, <math>\varepsilon^{\text{opt}} = 10^{-5}</math></li> <li>• <math>\bar{V} = 0.12</math>, <math>R = 3.5 \text{ mm}</math></li> <li>• <math>\beta^{\text{int}} = 50</math>, <math>\beta^{\max} = 128</math></li> <li>• <math>p = 3</math>, <math>\varepsilon = 10^{-6}</math>, <math>m = 0.02</math></li> </ul> |
|  | <pre>mpirun -n 8 python3 scripts/mechanism_2d.py</pre> | <ul style="list-style-type: none"> <li>• <math>E = 100 \text{ MPa}</math>, <math>\nu = 0.25</math></li> <li>• <math>n^{\max} = 500</math>, <math>\varepsilon^{\text{opt}} = 10^{-5}</math></li> <li>• <math>\bar{V} = 0.25</math>, <math>R = 0.5 \text{ mm}</math></li> <li>• <math>\beta^{\text{int}} = 50</math>, <math>\beta^{\max} = 128</math></li> <li>• <math>p = 3</math>, <math>\varepsilon = 10^{-6}</math>, <math>m = 0.05</math></li> </ul> |

**Table 6** Hardware and software configurations of the computing platforms tested in this study

| Configurations                 | Laptop                      | Desktop                      | Workstation                       | Cluster <sup>a</sup>        |
|--------------------------------|-----------------------------|------------------------------|-----------------------------------|-----------------------------|
| CPU brand                      | Intel(R) Core(TM) i7-10875H | Intel(R) Xeon(R) Silver 4116 | AMD Ryzen Threadripper PRO 3995WX | Intel(R) Xeon(R) E5-2680 v4 |
| Number of physical processors  | 8                           | 12                           | 64                                | 14 per node                 |
| Processor base frequency (GHz) | 2.30                        | 2.10                         | 2.70                              | 2.40                        |
| Memory size (GB)               | 16                          | 64                           | 256                               | 256 per node                |
| Operating system               | Windows 11 Home 64-bit      | Windows 10 Enterprise 64-bit | Windows 10 Enterprise 64-bit      | Linux                       |
| Number of nodes                | —                           | —                            | —                                 | 15                          |

<sup>a</sup>The information provided here pertains to a specific partition within the Illinois Campus Cluster

`scripts/beam_2d.py` signifies the execution of the input script `beam_2d.py` within the `scripts` directory using the Python interpreter.

Following the execution of the aforementioned command, we conduct topology optimization for the design setup as depicted in Fig. 3a and acquire optimized structures displayed in the insets of Fig. 3b–d. In Fig. 3b–d, we present the runtime performance of FEniTop executed on four representative computing platforms: the laptop, desktop, workstation, and cluster, each with specific hardware and software configurations detailed in Table 6. To assess the effectiveness of parallel computing, we evaluate runtime performance against various process numbers,  $N_p \in \{1, 2, 4, 6, 8, 10, 12, 14\}$ ,<sup>3</sup> for three representative mesh sizes:  $200 \times 60$  for the coarse mesh,  $600 \times 200$  for the medium mesh, and  $1500 \times 500$  for the fine mesh. Correspondingly, Fig. 3e–g illustrate the speedup achieved through parallel computing in comparison to serial computing, where speedup is defined as the serial runtime divided by the parallel runtime.

Several noteworthy observations can be drawn from Fig. 3b–g as follows. First, the FEniTop software demonstrates adaptability across diverse computing platforms, accommodating various hardware and software configurations without necessitating any modifications. Second, in general, for all computing platforms and mesh sizes, there is a conspicuous reduction in runtime in Fig. 3b–d (manifested as speedup in Fig. 3e–g when comparing parallel computing with serial computing, until a substantial number of processes is reached. This observation marks the point where a balance is struck between computation cost and communication cost among different processes, a well-known phenomenon referred to as parallel slowdown. Finally, the onset of parallel

slowdown can be delayed with larger mesh sizes, resulting in an increased peak speedup. This effect arises because a larger mesh size distributes more DOFs to each process, and the computational cost predominates over communication costs.

For an in-depth analysis of the parallel computing efficiency, Fig. 4a–c provides a detailed breakdown of the total runtime on the workstation for various mesh sizes and different process numbers. The total runtime is divided into six distinct components: initialization overhead, density filter and Heaviside projection, FEA, sensitivity analysis, solution updates with the optimizer, and post-processing. Correspondingly, Fig. 4d–f present the speedup achieved by each of these individual components. Based on Fig. 4a–f, it is evident that all major components, including FEA, filter and projection, and sensitivity evaluation, exhibit noticeable speedup, particularly for medium and large mesh sizes.

## 4.2 Extension to the unstructured mesh

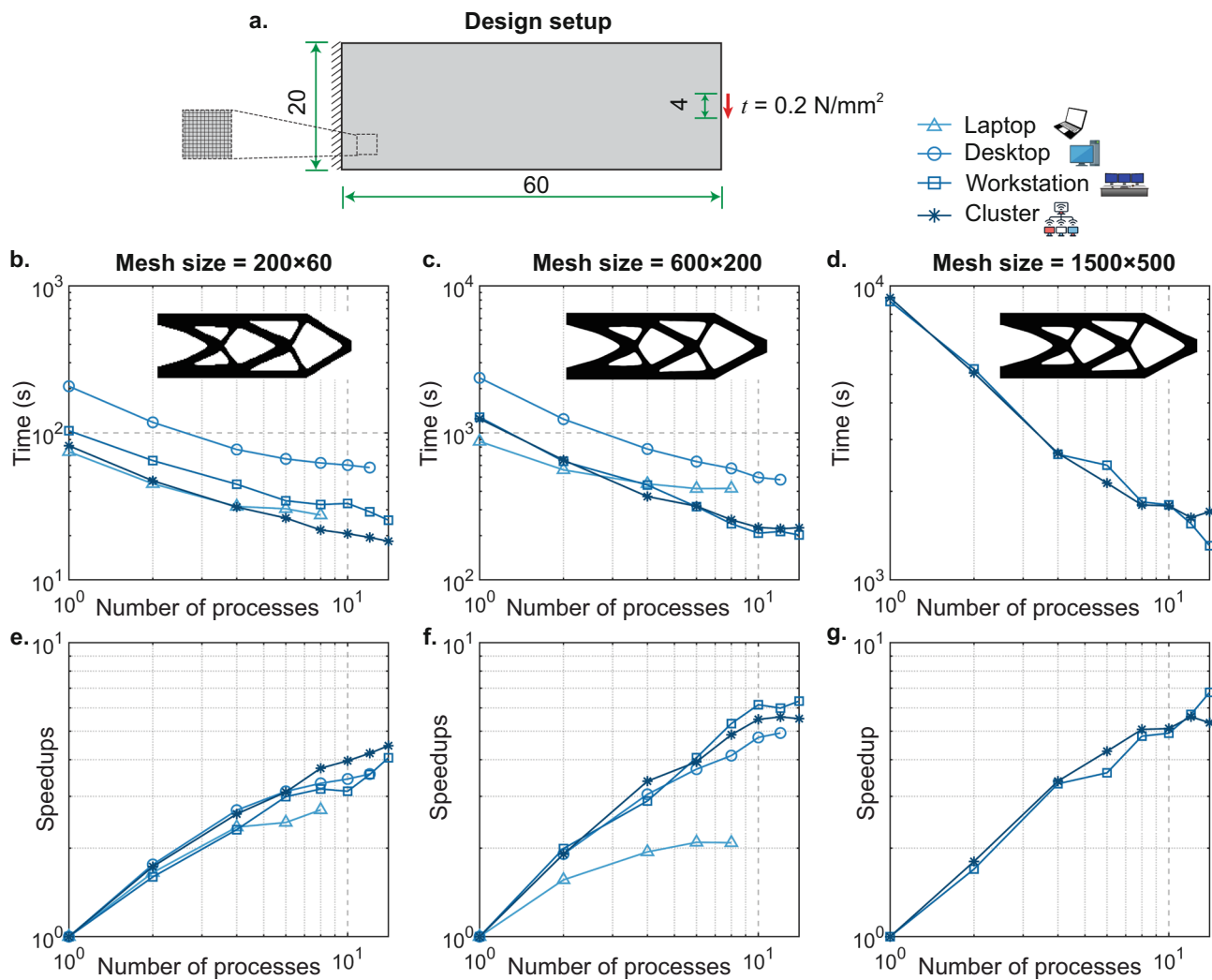
In this subsection, we harness the capabilities of FEniTop to facilitate topology optimization on unstructured meshes tailored for design domains with irregular geometries. As depicted in Fig. 5a, the design domain remains consistent with the one utilized in Sects. 4.3–4.4 of Jia et al. (2023). This domain adopts a disk shape featuring five circular holes, with the innermost hole anchored to the ground. Here, we apply a downward traction load, denoted as  $\bar{\mathbf{t}} = (0, -1)$  in units of  $\text{N/mm}^2$ , to the four corners along the outer boundary of the design domain.

To initiate the topology optimization process, we execute the input script `disk_2d.py` on the workstation in Table 6 with the following command as

```
1 mpirun -n 8 python3 scripts/disk_2d.py
```

Within this input script, we define an unstructured mesh using the following commands as

<sup>3</sup> We determine these process counts based on the problem sizes and the available processors of the hardware in Table 6. See the computational efficiency of FEA using FEniCS with larger process counts in Hoffman et al. (2016).



**Fig. 3** Parallel computing efficiency across various mesh sizes and process counts on different computing platforms. **a** Presents the design setup including the design domain and boundary conditions. The variable  $t$  is the amplitude of the applied traction (i.e.,  $t = |\bar{\mathbf{t}}|$ ). The inset demonstrates the structured mesh used in FEA and topology optimization. **b–d** Present the total computational time for three distinct

mesh sizes,  $200 \times 60$ ,  $600 \times 200$ , and  $1500 \times 500$ , respectively. Each panel displays the correlation between the total computational time and the process count, examined across four hardware configurations: laptop, desktop, workstation, and cluster. **e–g** Present the corresponding speedups

```

1 with dolfinx.io.XDMFFile(MPI.COMM_WORLD, "meshes/disk_2d.xdmf",
2   "r") as xdmf:
    mesh = xdmf.read_mesh(name="Grid")

```

These commands employ the `dolfinx.io.XDMFFile` module, which supports parallel computing, to load a pre-defined unstructured mesh. Such unstructured meshes can be readily generated using open-source libraries like `gmsh`<sup>4</sup> (its Python interface is available in FEniTop) and commer-

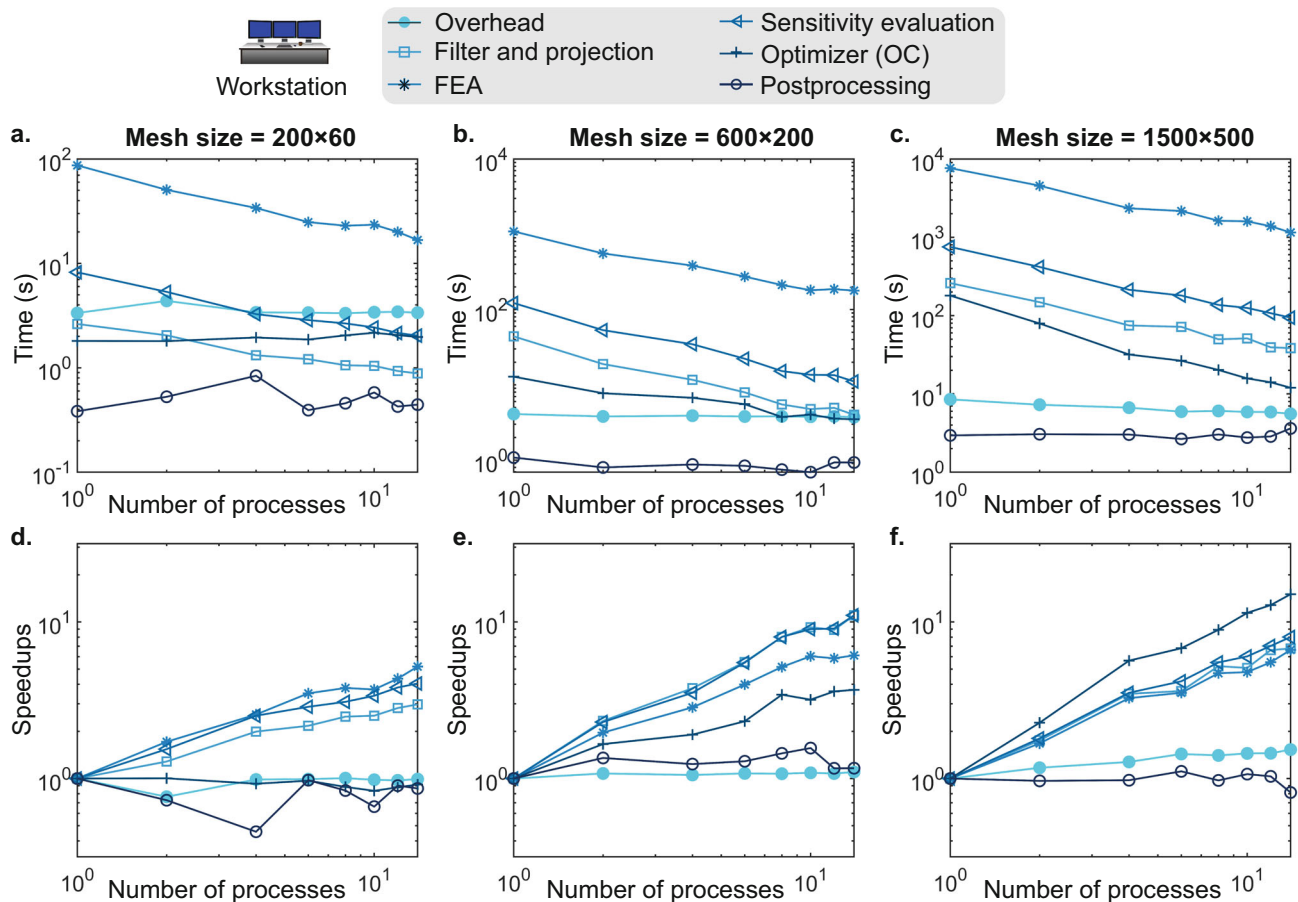
cial software like Abaqus.<sup>5</sup> Next, we can make use of the open-source package `meshio`<sup>6</sup> to seamlessly convert these unstructured meshes into XDMF files. These XDMF files can then be readily loaded by FEniTop.

In the current example, the unstructured mesh employed comprises quadrilateral finite elements, conforming to the conventions of topology optimization. As for topology optimization parameters, we set the upper bound for the material volume fraction to  $\bar{V} = 0.5$  and use a density filter radius of

<sup>4</sup> Available at <https://gmsh.info>.

<sup>5</sup> Available at <https://www.3ds.com/products-services/simulia/products/abaqus>.

<sup>6</sup> Available at <https://github.com/nschloe/meshio>.



**Fig. 4** Breakdown time and speedup of the workstation for various mesh sizes and process counts. **a–c** Present the breakdown time of the workstation for three distinct mesh sizes,  $200 \times 60$ ,  $600 \times 200$ , and  $1500 \times 500$ , respectively. Each panel displays the correlation between

the breakdown time and the process count, examined across six major components during topology optimization: overhead, filter and projection, FEA, sensitivity evaluation, OC optimizer, and postprocessing. **d–f** Present the corresponding speedups

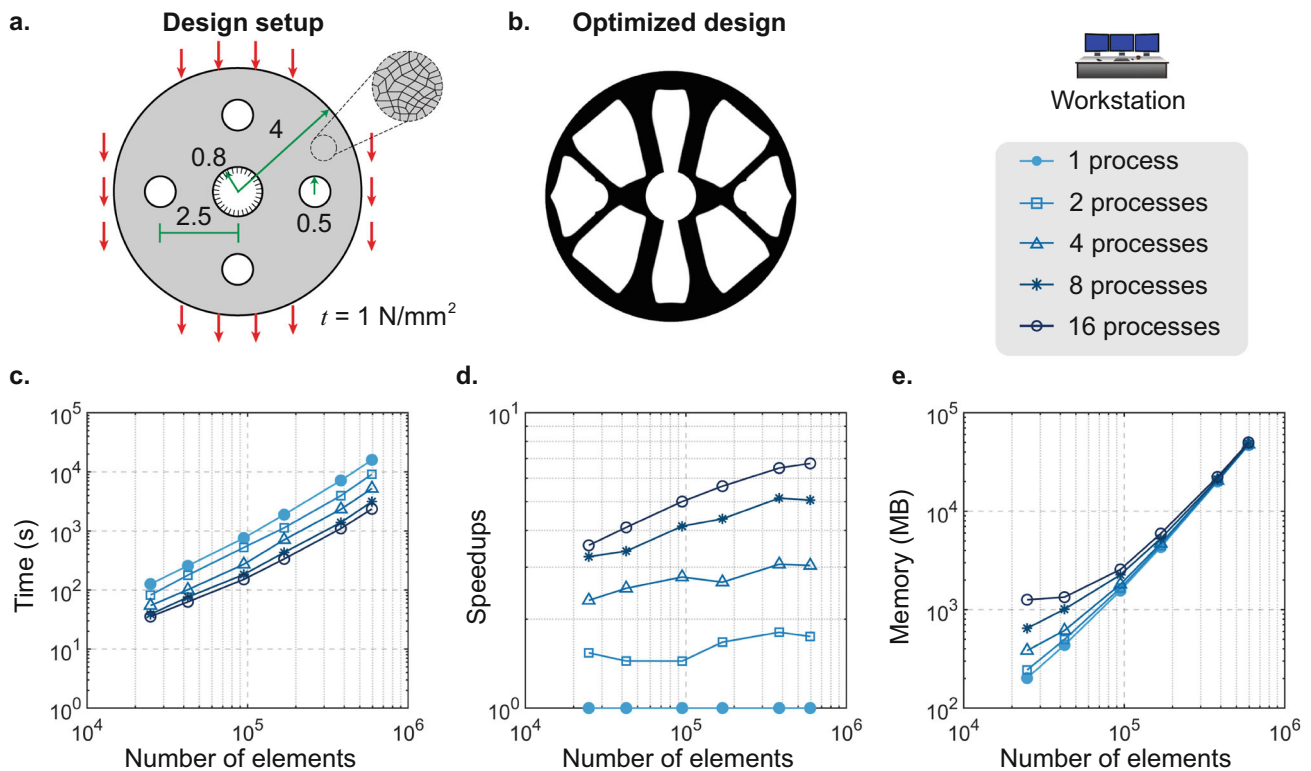
$R = 0.32$  mm. We set the interval to double the Heaviside sharpness parameter as  $\beta^{\text{int}} = 30$  and the maximum sharpness parameter as  $\beta^{\text{max}} = 128$ . Additionally, we impose a move limit of  $m = 0.02$  in the OC optimizer. Based on these setups, an optimized design, corresponding to the case with 168,936 finite elements executed on the workstation with 16 processes, is illustrated in Fig. 5b, where we observe the axial symmetry of the design due to the symmetric design domain and anti-symmetric boundary conditions.

To evaluate FEniTop's performance, we analyze unstructured meshes with finite element counts ranging from 24, 848 to 594, 888. Tests are conducted with process counts of 1, 2, 4, 8, and 16 to assess parallel efficiency. Results in Fig. 5c–d show increasing computational time with larger problem sizes but clear speedup in parallel computing. In addition, as shown in Fig. 5e, the memory usage remains consistent across different process counts for larger problem sizes, indicating efficient memory management. Overall, FEniTop supports topology optimization on unstructured meshes, achieving

good speedup (up to 6.74 times) without excessive memory usage.

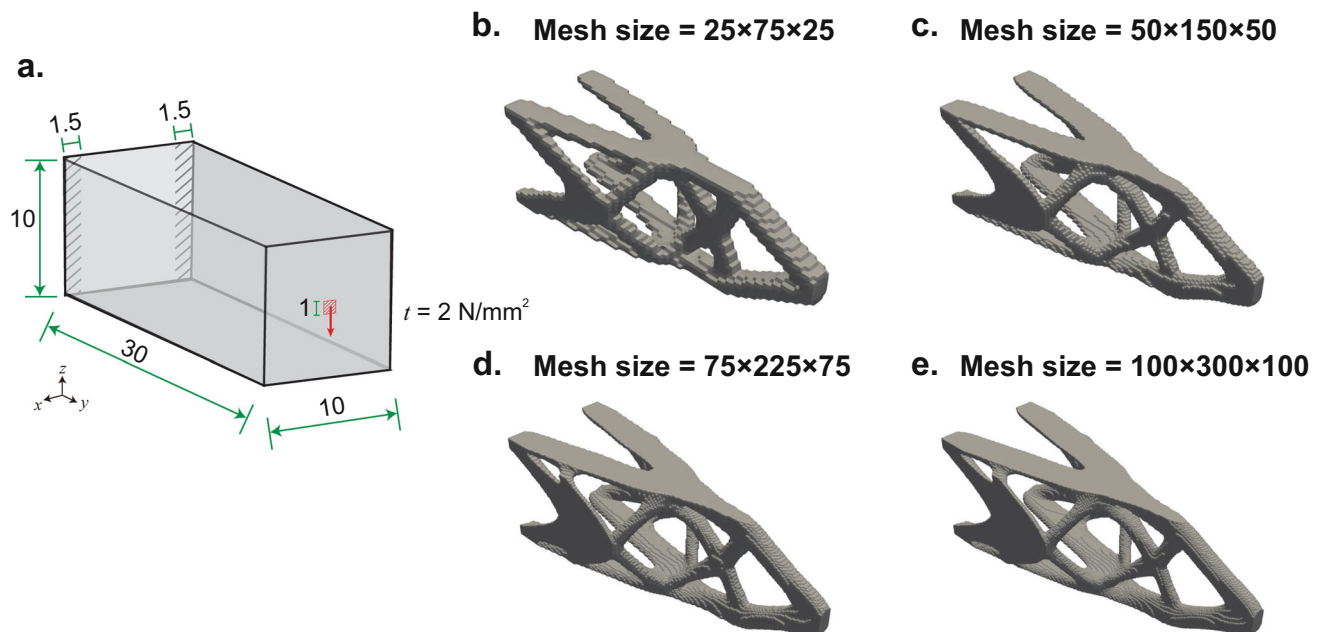
### 4.3 Extension to the 3D case and various solvers

In this subsection, we shift our focus to 3D topology optimization employing the FEniTop software and introduce the utilization of various linear solvers and preconditioners accessible through the PETSc backend of the underlying FEniCSx library. Figure 6a presents the design domain and boundary conditions of a 3D cantilever beam. We constrain two strip-shaped regions on one face and apply a downward traction load, denoted as  $\bar{\mathbf{t}} = (0, 0, -2)$  in units of  $\text{N/mm}^2$ , to the central region of the opposite face. To facilitate FEA and topology optimization, we discretize the design domain using first-order hexahedral Lagrange elements. To evaluate the performance of various solver configurations, we examine four representative mesh sizes:  $25 \times 75 \times 25$ ,  $50 \times 150 \times 50$ ,  $75 \times 225 \times 75$ , and  $100 \times 300 \times 100$  finite elements, cor-



**Fig. 5** Topology optimization and performance investigation for the 2D disk example. **a** Presents the design setup including the design domain and boundary conditions. The inset illustrates the unstructured mesh used for FEA and topology optimization. **b** Presents one optimized

design consisting of 168,936 finite elements and optimized with 16 processes on the workstation. **c–e** Present the performance investigations including the computational time, speedups, and memory usage



**Fig. 6** Topology optimization for the 3D beam example. **a** Presents the design setup including the design domain and boundary conditions. **b–e** Present the optimized designs for various mesh sizes,  $25 \times 75 \times 25$ ,

$50 \times 150 \times 50$ ,  $75 \times 225 \times 75$ , and  $100 \times 300 \times 100$ , respectively. These designs are optimized with 48 processes on the cluster and correspond to the GMRES–AMG solver configuration

**Table 7** Representative linear solvers available in FEniTop

| Keys      | Descriptions                                         |
|-----------|------------------------------------------------------|
| "preonly" | Direct solver                                        |
| "cg"      | Preconditioned CG method (Hestenes and Stiefel 1952) |
| "gmres"   | GMRES method (Saad and Schultz 1986)                 |
| "minres"  | Minimum residual method                              |

**Table 8** Representative preconditioners available in FEniTop

| Keys       | Descriptions                     |
|------------|----------------------------------|
| "lu"       | Lower–upper (LU) factorization   |
| "cholesky" | Cholesky factorization           |
| "gamg"     | AMG preconditioner (Stüben 2001) |

responding to 154,128, 1,178,253, 3,916,128, and 9,211,503 DOFs, respectively. The parameters for topology optimization are as follows: we set the upper limit for the material volume fraction to  $\bar{V} = 0.08$  and define a density filter radius of  $R = 0.6$  mm. Additionally, we impose a move limit of  $m = 0.02$  in the OC optimizer.

As for the solver configurations, Tables 7 and 8 provide an overview of representative linear solvers<sup>7</sup> and preconditioners<sup>8</sup> accessible in FEniTop through the PETSc backend. In this work, we primarily investigate three key linear solvers: a direct solver, a CG (Hestenes and Stiefel 1952) iterative solver, and a generalized minimal residual (GMRES) (Saad and Schultz 1986) iterative solver. Note that a direct solver can produce “exact” solutions with accuracy up to the computer’s precision. A CG solver is particularly suitable for solving symmetric positive-definite systems of linear equations, as encountered in this study’s linear elastostatics problems. Meanwhile, the GMRES solver is known for its robustness, applying even to indefinite nonsymmetric systems of linear equations.

To activate these linear solvers and preconditioners, we simply provide their keys to the “petsc\_options” argument in the input fem variable. For example, to configure the direct solver with LU factorization, we use the following commands as

```
1 "petsc_options": {
2   "ksp_type": "preonly",
3   "pc_type": "lu",
4   "pc_factor_mat_solver_type": "
5   mumps",
6 }
```

<sup>7</sup> See a complete list of supported linear solvers in <https://petsc.org/release/manualpages/KSP/KSPType>.

<sup>8</sup> See a complete list of supported preconditioners in <https://petsc.org/release/manualpages/PC/PCType>.

Additionally, to configure CG and GMRES solvers with the AMG preconditioner, we simply utilize the commands as

```
1 "petsc_options": {
2   "ksp_type": "cg",
3   "pc_type": "gamg",
4 }
```

and

```
1 "petsc_options": {
2   "ksp_type": "gmres",
3   "pc_type": "gamg",
4 }
```

Within these commands, we recall that “ksp\_type” specifies the linear solver type, while “pc\_type” specifies the preconditioner type. Furthermore, the PETSc backend offers various options to customize solver configurations, including those listed in Table 9. Each linear solver and preconditioner also includes numerous parameters to fine-tune performance. In this section, all parameters, except for the linear solver and preconditioner types, are maintained at their default values for simplicity. We leave these parameters for experienced readers to explore.

After configuring the solver information, we initiate topology optimization using 48 processes on the cluster in Table 6 for all solver configurations with the following command as

```
1 mpirun -n 48 python3 scripts/beam_3d
2 .py
```

Fig. 6b–e depict the optimized designs corresponding to the GMRES–AMG solver configuration for the four mesh sizes. Notably, larger mesh sizes result in optimized designs with smoother boundaries, thanks to more refined finite element discretization. It is important to mention that all solver configurations with the default parameters in Table 9 yield similar optimized designs, although the optimized designs for the LU solver and CG–AMG solver are not presented here for brevity.

Concerning computational performance, Tables 10 and 11 provide details on total computational time and total memory usage for different solver configurations across various mesh sizes, reflecting different DOF counts. Notably, the CG–AMG and GMRES–AMG solvers outperform the LU solver in terms of both computational time and memory usage, particularly for larger mesh sizes. This observation aligns with existing literature, such as Stüben (2001). For instance, in

**Table 9** Representative solver configuration options available in FEniTop through the PETSc backend

| Keys                        | Default values | Descriptions                            |
|-----------------------------|----------------|-----------------------------------------|
| "pc_factor_mat_solver_type" | "petsc"        | Solver package for matrix factorization |
| "ksp_max_it"                | 10000          | Maximum number of iterations to use     |
| "ksp_rtol"                  | 1e-08          | Relative convergence tolerance          |
| "ksp_atol"                  | 1e-50          | Absolute convergence tolerance          |
| "ksp_divtol"                | 10000          | Divergence tolerance                    |

**Table 10** Total computational time (in hours) of different solvers for various mesh sizes (various DOFs)

| Mesh sizes     | 25 × 75 × 25 | 50 × 150 × 50 | 75 × 225 × 75 | 100 × 300 × 100 |
|----------------|--------------|---------------|---------------|-----------------|
| Number of DOFs | 154,128      | 1,178,253     | 3,916,128     | 9,211,503       |
| LU             | 0.436        | 12.139        | 93.889        | —               |
| CG-AMG         | 0.201        | 0.981         | 3.306         | 6.806           |
| GMRES-AMG      | 0.136        | 0.944         | 3.500         | 5.333           |

the case of the  $75 \times 225 \times 75$  finite element mesh, compared to the LU solver, the CG-AMG and GMRES-AMG solver configurations are 28.4 and 26.8 times faster, respectively. Furthermore, the CG-AMG and GMRES-AMG solver configurations reduce memory usage by 11.1 and 10.7 times, respectively. The performance data for the LU solver with the  $100 \times 300 \times 100$  mesh size is not included due to its high memory demands.

#### 4.4 Extension to various loading directions and post-processing

In this example, we expand the scope of topology optimization to encompass 3D unstructured meshes with intricate boundary conditions. We also demonstrate the use of ParaView software for post-processing, which enables the visualization of optimized designs, smoothing of jagged boundaries, and exporting the optimized designs into STL files for 3D printing. The design domain, as illustrated in Fig. 7a, takes the form of a spherical shell, where the top and bottom are fixed, and rotational traction loading ( $\bar{\mathbf{t}}$ ) is applied at four corners of the midplane.

Following the procedure in Sect. 4.2, the implementation process for such a complex design setup unfolds as

```

1 with dolfinx.io.XDMFFile(MPI.COMM_WORLD, "meshes/shell_3d.xdmf",
  "r") as xdmf:
2     mesh = xdmf.read_mesh(name="Grid
  ")

```

The unstructured mesh considered here comprises 1,533,280 hexahedral finite elements with an average element size of  $h = 5 \mu\text{m}$ . To define the traction boundary conditions, which vary at different positions, we employ the following commands as

```

1 def on_surface(x):

```

```

2     return (np.isclose(x[0]**2+x
3         [1]**2+x[2]**2, 50**2)
4         & np.greater(x[2], -2) &
5         np.less(x[2], 2))
6
7     traction_bcs = [
8         [(0, 0.1, 0), lambda x:
9             on_surface(x) & np.greater(x[0],
10                49.8)],
11         [(-0.1, 0, 0), lambda x:
12             on_surface(x) & np.greater(x[1],
13                49.8)],
14         [(0, -0.1, 0), lambda x:
15             on_surface(x) & np.less(x[0],
16                -49.8)],
17         [(0.1, 0, 0), lambda x:
18             on_surface(x) & np.less(x[1],
19                -49.8)],
20     ]

```

Within these commands, we define a lambda function to identify the mesh nodes situated on the outer surface of the spherical shell. We then specify four distinct traction values at four different locations in the variable `traction_bcs`. Subsequently, we pass the `traction_bcs` variable to the input `fem` variable to complete the setup of traction boundary conditions. Topology optimization is initiated using 48 processes on the cluster described in Table 6 with the following command as

```

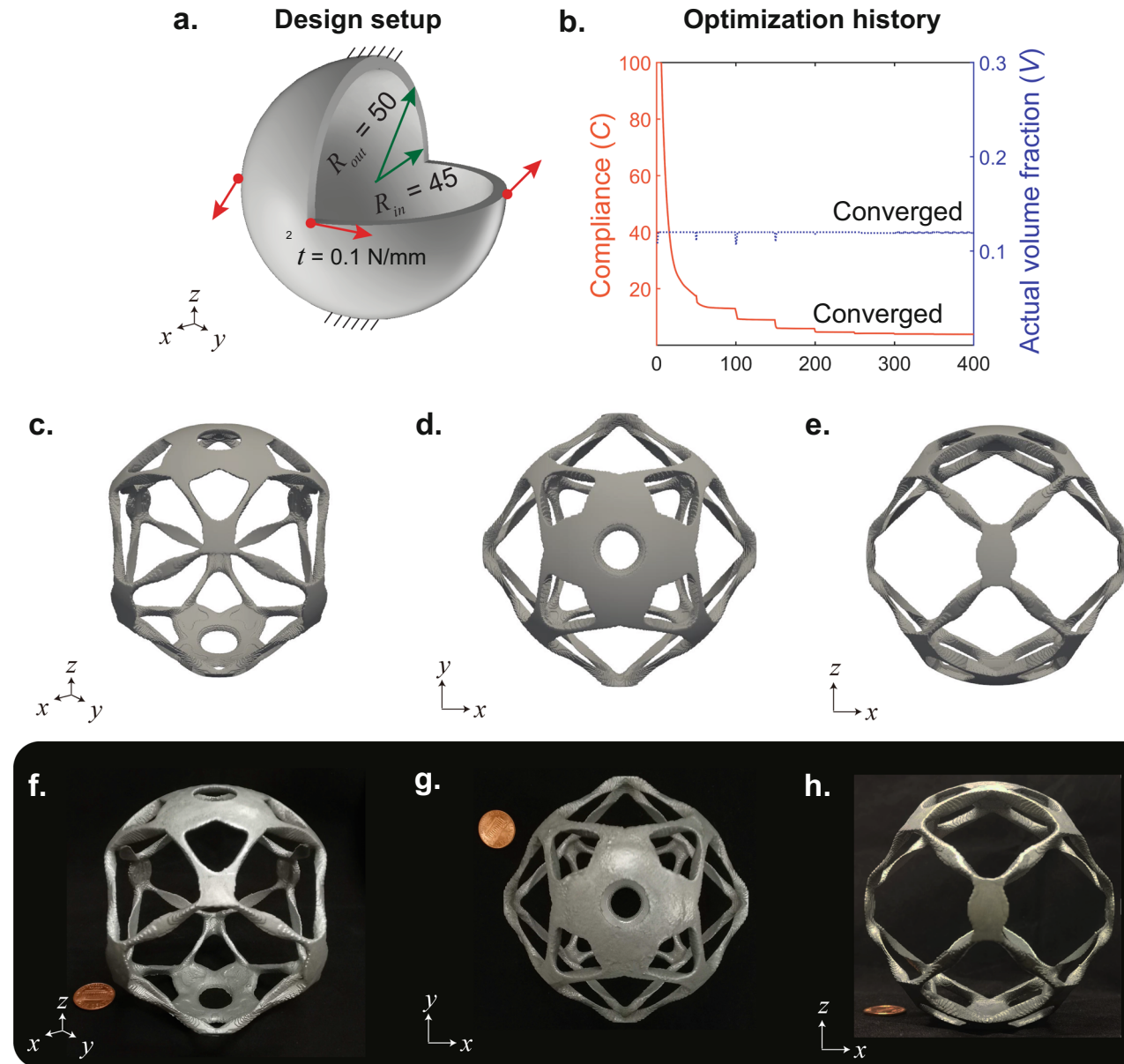
1 mpirun -n 48 python3 scripts/
  shell_3d.py

```

After completing topology optimization, we obtain the converged histories of structural compliance ( $C$ ) and the actual material volume fraction ( $V$ ), as depicted in Fig. 7b. Furthermore, we present various views of the optimized design in Fig. 7c–e, showcasing a non-intuitive material distribution. Notably, the FEniTop software can automatically export the optimized designs into an XDMF file, even if the program is executed in parallel. This XDMF file can be

**Table 11** Total memory usage (in GB) of different solvers for various mesh sizes (various DOFs)

| Mesh sizes     | 25 × 75 × 25 | 50 × 150 × 50 | 75 × 225 × 75 | 100 × 300 × 100 |
|----------------|--------------|---------------|---------------|-----------------|
| Number of DOFs | 154,128      | 1,178,253     | 3,916,128     | 9,211,503       |
| LU             | 10.254       | 89.551        | 410.156       | –               |
| CG-AMG         | 2.754        | 11.719        | 37.012        | 82.227          |
| GMRES-AMG      | 2.832        | 12.109        | 38.282        | 84.863          |

**Fig. 7** Digital rendering and 3D printed physical models of the optimized design. **a** Presents the design setup including the design domain and boundary conditions. **b** Shows the optimization histories of structural compliance ( $C$ ) and the actual material volume fraction ( $V$ ). **c–e**

Present the digital rendering of the optimized design from the 3D, top, and side views, respectively. **f–h** Depict the physical models from the corresponding views. These physical models are printed with the STL file generated by ParaView

directly imported into the open-source ParaView software for visualization. By leveraging ParaView software, we can further filter the low-density regions, smooth the jagged boundaries, and export the smoothed designs to STL files for 3D printing.

Based on the generated STL file, Fig. 7f–h display 3D printed physical models juxtaposed with their corresponding digital renderings in Fig. 7c–e. These physical models were produced using an Elegoo Saturn 2 printer with Elegoo water-washable photopolymer resins. Following the printing process, a layer of metallic paint (Krylon Fusion All-In-One by Krylon Products Group) was applied to enhance the visual impact. By comparing the physical models in Fig. 7f–h with their digital counterparts in Fig. 7c–e, it is evident that all structural features have been accurately preserved. This achievement remarks the robust post-processing capabilities of FEniTop through the ParaView extension, allowing for a seamless transition from digital designs to physical models.

#### 4.5 Extension to compliant mechanism design

In this final example, we extend FEniTop to handle a more complex problem, compliant mechanism in (2). This design involves specifying passive solid and void regions, adding additional springs to the stiffness matrix, sensitivity analysis of non-self-adjoint functions, and using the MMA optimizer to handle multiple constraints. Specifically, as depicted in Fig. 8a, the design domain is fixed at the top-left and bottom-left corners. We apply traction ( $\bar{\mathbf{t}} = (1, 0)$  in units of  $\text{N}/\text{mm}^2$ ) on the left edge and aim to maximize the output displacement ( $\mathbf{u}_{\text{out}}$ ) on the right edge. Additionally, we apply distributed artificial springs following the convention outlined in, where the input spring can also be treated with a displacement constraint. Furthermore, we set two passive solid regions (black rectangles) and two passive void regions (circular holes) within the design domain.

Compared to the previous examples, we need to specify passive zones in the current setup, which can be done by passing lambda functions into the "solid\_zone" and "void\_zone" arguments of the opt variable in the input script as follows.

```
1 {"solid_zone": lambda x: ((np.less(x
2 [0], 0.5) | np.greater(x[0], 9.5))
3 & np.less(x[1], 1) & np.greater(x[1], -1)),
4 "void_zone": lambda x: (np.less((x
5 [0]-2.5)**2+(x[1]-3)**2, 0.5**2)
6 | np.less((x[0]-2.5)**2+(x[1]+3)**2, 0.5**2))}
```

The above commands are mathematically equivalent to

$$\begin{cases} \Omega^{\text{solid}} = \{(x, y) | x \in [0, 0.5] \cup (9.5, 10] \text{ and } y \in (-1, 1)\} \\ \Omega^{\text{void}} = \{(x, y) | (x - 2.5)^2 + (y - 3)^2 < 0.5^2 \text{ or } (x - 2.5)^2 + (y + 3)^2 < 0.5^2\} \end{cases}$$

with the origin defined as the center of the design domain's left edge. The symbols  $\Omega^{\text{solid}} \subset \Omega$  and  $\Omega^{\text{void}} \subset \Omega$  are the passive solid and void regions, respectively. Furthermore, we need to define the locators of the input (in\_locator) and output (out\_locator) springs with commands as

```
1 def in_locator(x):
2     return np.isclose(x[0], 0) & np.
3         less(x[1], 1.5) & np.greater(x[1],
4         -1.5)
5
6 def out_locator(x):
7     return np.isclose(x[0], 10) & np.
8         less(x[1], 1.5) & np.greater(x[1],
9         -1.5)
```

After that, we specify the information of the artificial springs with the following commands as

```
1 {"in_spring": [in_locator, "x",
2 0.2],
3 "out_spring": [out_locator, "x",
4 0.2]}
```

Here, the first argument ("in\_locator" or "out\_locator") in the list is the locator of the spring. The second argument ("x", "y", or "z") specifies the spring direction, and the last argument (0.2 here) determines spring stiffness. In addition, we set

```
1 {"use_oc": False,
2 "opt_compliance": False,
3 "compliance_bound": 0.5}
```

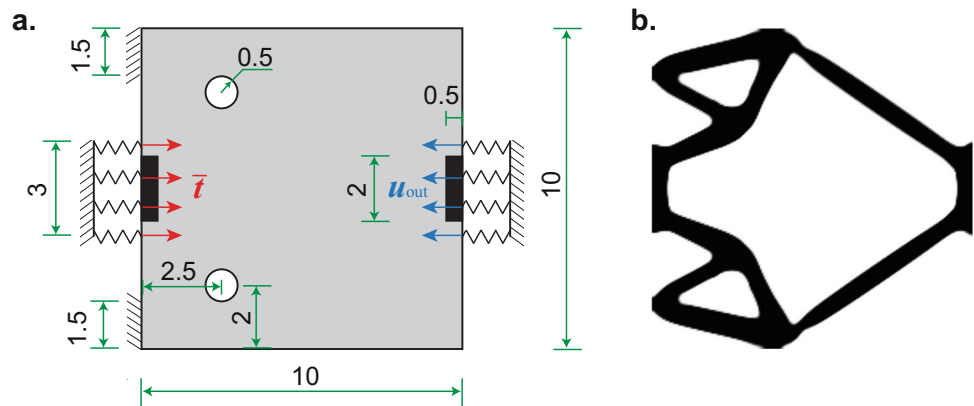
in the opt variable to inform FEniTop that the MMA (instead of the OC) optimizer will be used, and the compliant mechanism (instead of compliance minimization) problem will be considered with a compliance upper bound,  $\bar{C} = 0.5 \text{ N} \cdot \text{mm}$ .

After the above additional setups, we initiate a compliant mechanism design problem using 8 processes with the following command as

```
1 mpirun -n 8 python3 scripts/
2 mechanism_2d.py
```

The optimized design is depicted in Fig. 8b, where we observe that the passive zones are correctly applied. Furthermore, after 400 iterations, the summation of the output displacement ( $U$ ) is optimized from 0.62 to  $-2.93 \text{ mm}$ . Additionally, both the actual compliance and material volume fraction reach their respective bounds,  $\bar{C} = 0.5 \text{ N} \cdot \text{mm}$  and  $\bar{V} = 0.25$ , indicating the versatility and efficacy of FEniTop in addressing complex design problems.

**Fig. 8** Topology optimization for compliant mechanism design. **a** Presents the design setup including the design domain and boundary conditions. The design domain consists of two passive solid (black rectangles) and two passive void regions (circular holes), and it is also connected to distributed artificial springs along the input traction,  $\mathbf{t}$ , and the output displacement,  $\mathbf{u}_{\text{out}}$ . **b** Presents the optimized design



## 5 Concluding remarks

In this study, we introduce FEniTop, a user-friendly 2D and 3D topology optimization software built upon FEniCSx. FEniTop is designed to facilitate parallel computing across a wide range of platforms, from laptops to clusters. The software comprises a single input script for specifying topology optimization problems and six replaceable modules for executing topology optimization tasks. In the input script, users can efficiently define 2D and 3D topology optimization problems on both structured and unstructured meshes, with the flexibility to incorporate various boundary conditions and solver configurations. Once the problem is defined, the input script seamlessly invokes the FEniTop modules to execute topology optimization tasks in parallel. Additionally, FEniTop automatically exports the optimized designs as XDMF files, which can be directly imported into ParaView software for post-processing. This process facilitates visualization, filtering low-density regions, smoothing the filtered designs, and converting the smoothed designs into STL files for 3D printing.

To demonstrate the capabilities of FEniTop, we present five illustrative examples in this study. Specifically, we begin with a simple demonstration and performance tests by employing a 2D cantilever beam. We then generalize FEniTop to optimize a 2D disk with an unstructured mesh. Next, we extend topology optimization to a 3D cantilever beam where we also introduce and investigate various solver options. Subsequently, we present topology optimization for a 3D spherical shell with complex boundary conditions. Here, we present a tangible demonstration through a 3D-printed physical model, fabricated using the STL file derived from the designs optimized by FEniTop. Finally, we illustrate FEniTop's capability in addressing a compliant mechanism problem. These examples showcase not only the FEniTop software's functionality but also its impressive parallel computing performance, achieved without excessive memory usage. Therefore, FEniTop could benefit both beginners in the field of SMO, who value efficiency and ease of use, as well

as experts in the field who prioritize computing efficiency and the ability to tackle complex optimization problems. The open-source FEniTop is given in Appendix B and available to download at <https://github.com/missionlab/fenitop>.

Looking forward, the foundation provided by the FEniCSx library positions FEniTop for further expansion to address intricate topology optimization problems, because FEniCSx enables users to elegantly express complex problems through concise weak forms. This feature eliminates the need for chain rule computations in both FEA and sensitivity analysis, as aptly demonstrated in Jia et al. (2023, 2024a,b). In addition, the parallel computational efficiency gains with FEniTop will be particularly prominent in nonlinear topology optimization problems, where FEA and adjoint equations are solved multiple times per optimization iteration. Given the growing popularity of open-source software, especially within the Python ecosystem, and the substantial advancements in parallel computing, we hope FEniTop can build a strong foundation for future developments in the SMO field.

## Appendix A Sensitivity analysis through automatic differentiation

### Appendix A.1 Analytical solution with the adjoint method

To evaluate the sensitivity of a general objective or constraint function ( $\mathcal{F}$ ) with respect to the global vector of physical variables ( $\bar{\rho}$ ), we can employ the adjoint method (Bendsoe and Sigmund 2003) as follows. We begin by writing out the function's Lagrangian form ( $\hat{\mathcal{F}}$ ) as

$$\hat{\mathcal{F}}(\bar{\rho}, \mathbf{U}(\bar{\rho})) = \mathcal{F}(\bar{\rho}, \mathbf{U}(\bar{\rho})) + \lambda \cdot (\mathbf{F}^{\text{int}}(\bar{\rho}, \mathbf{U}(\bar{\rho})) - \mathbf{F}^{\text{ext}})$$

where  $\mathbf{U}$  is the global displacement vector, and  $\lambda$  is an adjoint vector with  $\lambda_i = 0$  if DOF  $i$  is constrained and  $\lambda_i$  remaining an unknown otherwise. The symbols  $\mathbf{F}^{\text{int}}$  and  $\mathbf{F}^{\text{ext}}$  are global internal and external force vectors, respectively. Tak-

ing the total derivative of the Lagrangian ( $\widehat{\mathcal{F}}$ ) with respect to the physical design variables ( $\bar{\rho}$ ) yields

$$\frac{d\widehat{\mathcal{F}}}{d\bar{\rho}} = \frac{\partial \mathcal{F}}{\partial \bar{\rho}} + \left( \frac{\partial \mathbf{U}}{\partial \bar{\rho}} \right)^{\top} \cdot \frac{\partial \mathcal{F}}{\partial \mathbf{U}} + \left[ \left( \frac{\partial \mathbf{F}^{\text{int}}}{\partial \bar{\rho}} \right)^{\top} + \left( \frac{\partial \mathbf{U}}{\partial \bar{\rho}} \right)^{\top} \cdot \left( \frac{\partial \mathbf{F}^{\text{int}}}{\partial \mathbf{U}} \right)^{\top} \right] \cdot \boldsymbol{\lambda}.$$

Rearranging terms renders

$$\frac{d\widehat{\mathcal{F}}}{d\bar{\rho}} = \frac{\partial \mathcal{F}}{\partial \bar{\rho}} + \left( \frac{\partial \mathbf{F}^{\text{int}}}{\partial \bar{\rho}} \right)^{\top} \cdot \boldsymbol{\lambda} + \left( \frac{\partial \mathbf{U}}{\partial \bar{\rho}} \right)^{\top} \cdot \left[ \left( \frac{\partial \mathbf{F}^{\text{int}}}{\partial \mathbf{U}} \right)^{\top} \cdot \boldsymbol{\lambda} + \frac{\partial \mathcal{F}}{\partial \mathbf{U}} \right].$$

To vanish the computationally expensive term of  $(\partial \mathbf{U} / \partial \bar{\rho})^{\top}$ , we enforce

$$\left( \frac{\partial \mathbf{F}^{\text{int}}}{\partial \mathbf{U}} \right)^{\top} \cdot \boldsymbol{\lambda} = -\frac{\partial \mathcal{F}}{\partial \mathbf{U}} \quad \text{subjected to} \quad \lambda_i = 0 \text{ if DOF } i \text{ is constrained,} \quad (\text{A.1})$$

which is referred to as the adjoint equation. By solving the adjoint vector ( $\boldsymbol{\lambda}$ ) from the adjoint equation in (A.1), we can evaluate the sensitivity as

$$\frac{d\mathcal{F}}{d\bar{\rho}} = \frac{d\widehat{\mathcal{F}}}{d\bar{\rho}} = \frac{\partial \mathcal{F}}{\partial \bar{\rho}} + \left( \frac{\partial \mathbf{F}^{\text{int}}}{\partial \bar{\rho}} \right)^{\top} \cdot \boldsymbol{\lambda}. \quad (\text{A.2})$$

## Appendix A.2 Numerical solution with the automatic differentiation

Based on (A.2), the evaluation of the sensitivity of a given function ( $d\mathcal{F}/d\bar{\rho}$ ) requires at most four terms (see also the illustration in Fig. 9): vectors  $\partial \mathcal{F} / \partial \bar{\rho}$  and  $\partial \mathcal{F} / \partial \mathbf{U}$  as well as matrices  $\partial \mathbf{F}^{\text{int}} / \partial \bar{\rho}$  and  $\partial \mathbf{F}^{\text{int}} / \partial \mathbf{U}$ . These four terms can be conveniently computed with the automatic differentiation in FEniCSx software. We initiate this process by defining the internal force at the weak form level as

$$f^{\text{int}} = \int_{\Omega} \boldsymbol{\sigma}(\bar{\rho}, \mathbf{u}(\bar{\rho})) : \boldsymbol{\epsilon}(\mathbf{v}) d\mathbf{X}.$$

Subsequently, we define the above four pertinent terms at the weak form level, namely  $\partial \mathcal{F} / \partial \bar{\rho}$ ,  $\partial \mathcal{F} / \partial \mathbf{u}$ ,  $\partial f^{\text{int}} / \partial \bar{\rho}$ , and  $\partial f^{\text{int}} / \partial \mathbf{u}$ . These quantities can be computed directly using the FEniCSx library within FEniTop by leveraging automatic differentiation, circumventing the need for manual derivation involving complex chain rules.

Next, we leverage FEniCSx to effortlessly derive the vector and matrix forms of the above weak form-level quantities.

Specifically, we obtain the two vectors as

$$\frac{\partial \mathcal{F}}{\partial \bar{\rho}} = \mathfrak{V} \left( \frac{\partial \mathcal{F}}{\partial \bar{\rho}} \right) \quad \text{and} \quad \frac{\partial \mathcal{F}}{\partial \mathbf{U}} = \mathfrak{V} \left( \frac{\partial \mathcal{F}}{\partial \mathbf{u}} \right)$$

and the two matrices as

$$\frac{\partial \mathbf{F}^{\text{int}}}{\partial \bar{\rho}} = \mathfrak{M} \left( \frac{\partial f^{\text{int}}}{\partial \bar{\rho}} \right) \quad \text{and} \quad \frac{\partial \mathbf{F}^{\text{int}}}{\partial \mathbf{U}} = \mathfrak{M} \left( \frac{\partial f^{\text{int}}}{\partial \mathbf{u}} \right)$$

where the operators  $\mathfrak{V}(\cdot)$  and  $\mathfrak{M}(\cdot)$  correspond to vector and matrix assembly operations within FEniCSx, respectively, and can be implemented using the following commands as

```
1 dolfinx.fem.petsc.assemble_vector()
2 dolfinx.fem.petsc.assemble_matrix()
```

Having obtained the vectors  $\partial \mathcal{F} / \partial \bar{\rho}$  and  $\partial \mathcal{F} / \partial \mathbf{U}$  as well as matrices  $\partial \mathbf{F}^{\text{int}} / \partial \bar{\rho}$  and  $\partial \mathbf{F}^{\text{int}} / \partial \mathbf{U}$ , we can proceed to construct the adjoint equation in (A.1) with  $\partial \mathbf{F}^{\text{int}} / \partial \mathbf{U}$  and  $\partial \mathcal{F} / \partial \mathbf{U}$ . We then solve for the adjoint vector  $\boldsymbol{\lambda}$  by making use of the PETSc backend that supports parallel computing. Finally, we compute the sensitivity by substituting  $\partial \mathcal{F} / \partial \bar{\rho}$ ,  $\partial \mathbf{F}^{\text{int}} / \partial \bar{\rho}$ , and  $\boldsymbol{\lambda}$  into (A.2).

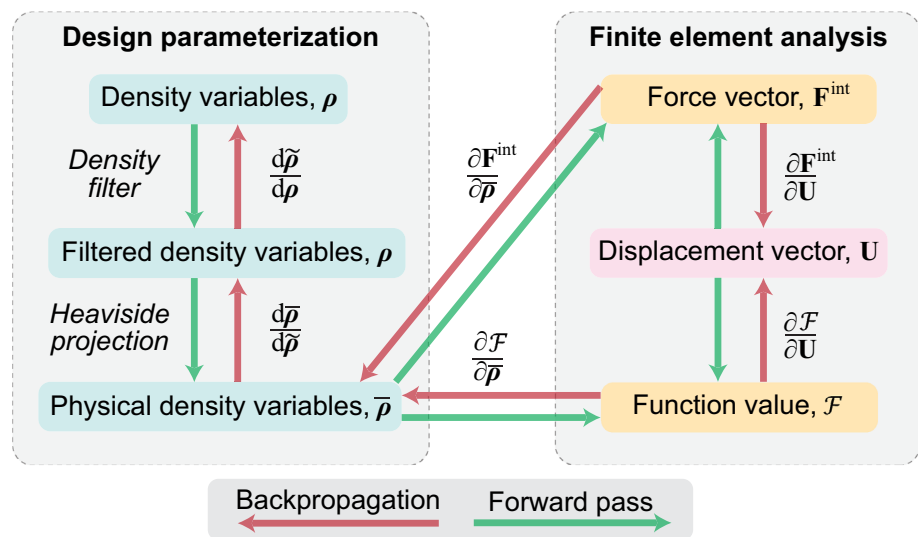
We highlight that this proposed numerical approach is versatile and can be applied to the sensitivity analysis of general objective and constraint functions, which eliminates the need for laborious manual derivation of sensitivity expressions involving complex chain rules. Interested readers can find more applications of this method in Jia et al. (2023), which handles the sensitivity analyses of various nonlinear functions such as damaged strain energy, fracture energy, and post-fracture effective toughness. The sensitivity verification of the proposed method is also presented in Appendices "B.1–B.2" of Jia et al. (2023), where we observe a close agreement between the proposed method and the forward finite difference approach.

## Appendix B The complete implementation of FEniTop

### Appendix B.1 Sample input script of beam\_2d.py

```
1 import numpy as np
2 from mpi4py import MPI
3 from dolfinx.mesh import
4     create_rectangle, CellType
5 from fenitop.topopt import topopt
6
7 mesh = create_rectangle(MPI.COMM_WORLD,
8     [[0, 0], [60, 20]],
9     [200, 60],
10     CellType.quadrilateral)
11 if MPI.COMM_WORLD.rank == 0:
```

**Fig. 9** The illustration of the sensitivity analysis involved in design space parameterization and FEA. Both design space parameterization and FEA consist of the forward pass and backpropagation stages. The forward pass evaluates the global internal force vector,  $\mathbf{F}^{\text{int}}$ , and the function value,  $\mathcal{F}$ . By inverting the forward pass, the backpropagation achieves the sensitivity analysis



```

11 mesh_serial = create_rectangle(MPI.COMM_SELF, [[0, 0], [60, 20]],
12                                     [200,
13                                     60], CellType.quadrilateral)
14 else:
15     mesh_serial = None
16 fem = { # FEM parameters
17     "mesh": mesh,
18     "mesh_serial": mesh_serial,
19     "young's modulus": 100,
20     "poisson's ratio": 0.25,
21     "disp_bc": lambda x: np.isclose(x[0], 0),
22     "traction_bcs": [(0, -0.2),
23                      lambda x: (np.isclose(x[0], 60) & np.greater(x[1], 8) & np.less(x[1], 12))],
24     "body_force": (0, 0),
25     "quadrature_degree": 2,
26     "petsc_options": {
27         "ksp_type": "cg",
28         "pc_type": "gamg",
29     },
30 }
31
32 opt = { # Topology optimization parameters
33     "max_iter": 400,
34     "opt_tol": 1e-5,
35     "vol_frac": 0.5,
36     "solid_zone": lambda x: np.full(x.shape[1], False),
37     "void_zone": lambda x: np.full(x.shape[1], False),
38     "penalty": 3.0,
39     "epsilon": 1e-6,
40     "filter_radius": 1.2,
41     "beta_interval": 50,
42     "beta_max": 128,
43     "use_oc": True,
44     "move": 0.02,
45     "opt_compliance": True,

```

```

46 }
47
48 if __name__ == "__main__":
49     toptopt(fem, opt)

```

## Appendix B.2 toptopt module for topology optimization

```

1 import time
2
3 import numpy as np
4 from mpi4py import MPI
5
6 from fenitop.fem import form_fem
7 from fenitop.parameterize import DensityFilter, Heaviside
8 from fenitop.sensitivity import Sensitivity
9 from fenitop.optimize import optimality_criteria, mma_optimizer
10 from fenitop.utility import Communicator, Plotter, save_xdmf
11
12
13 def toptopt(fem, opt):
14     """Main function for topology optimization."""
15
16     # Initialization
17     comm = MPI.COMM_WORLD
18     linear_problem, u_field, lambda_field, rho_field,
19     rho_phys_field = form_fem(fem, opt)
20     density_filter = DensityFilter(comm, rho_field, rho_phys_field,
21                                   opt["filter_radius"], fem["petsc_options"])
22     heaviside = Heaviside(rho_phys_field)
23     sens_problem = Sensitivity(comm, opt, linear_problem, u_field,
24                               lambda_field, rho_phys_field)

```

```

23 S_comm = Communicator(rho_phys_field
24   .function_space, fem["mesh_serial"
25   ])
26 if comm.rank == 0:
27     plotter = Plotter(fem["
28     mesh_serial"])
29 num_consts = 1 if opt["
30 opt_compliance"] else 2
31 num_elems = rho_field.vector.array.
32 size
33 if not opt["use_oc"]:
34     rho_old1, rho_old2, = np.zeros(
35     num_elems), np.zeros(num_elems)
36     low, upp = None, None
37
38 # Apply passive zones
39 centers = rho_field.function_space.
40 tabulate_dof_coordinates()[0:
41 num_elems].T
42 solid, void = opt["solid_zone"](
43 centers), opt["void_zone"](centers)
44 rho_ini = np.full(num_elems, opt["
45 vol_frac"])
46 rho_ini[solid], rho_ini[void] =
47 0.995, 0.005
48 rho_field.vector.array[:] = rho_ini
49 rho_min, rho_max = np.zeros(
50 num_elems), np.ones(num_elems)
51 rho_min[solid], rho_max[void] =
52 0.99, 0.01
53
54 # Start topology optimization
55 opt_iter, beta, change = 0, 1, 2*opt
56 ["opt_tol"]
57 while opt_iter < opt["max_iter"] and
58     change > opt["opt_tol"]:
59     opt_start_time = time.
60     perf_counter()
61     opt_iter += 1
62
63     # Density filter and Heaviside
64     projection
65     density_filter.forward()
66     if opt_iter % opt["beta_interval"]
67     == 0 and beta < opt["beta_max"]:
68         beta *= 2
69         change = opt["opt_tol"] * 2
70         heaviside.forward(beta)
71
72     # Solve FEM
73     linear_problem.solve_fem()
74
75     # Compute function values and
76     sensitivities
77     [C_value, V_value, U_value],
78     sensitivities = sens_problem.
79     evaluate()
80     heaviside.backward(sensitivities
81 )
82     [dCdrho, dVdrho, dUdrho] =
83     density_filter.backward(
84     sensitivities)
85     if opt["opt_compliance"]:
86         g_vec = np.array([V_value-
87         opt["vol_frac"]])
88
89         dJdrho, dgdrho = dCdrho, np.
90         vstack([dVdrho])
91         else:
92             g_vec = np.array([V_value-
93             opt["vol_frac"], C_value-opt["
94             compliance_bound"]])
95             dJdrho, dgdrho = dUdrho, np.
96             vstack([dVdrho, dCdrho])
97
98             # Update the design variables
99             rho_values = rho_field.vector.
100             array.copy()
101             if opt["opt_compliance"] and opt
102             ["use_oc"]:
103                 rho_new, change =
104                 optimality_criteria(
105                     rho_values, rho_min,
106                     rho_max, g_vec, dJdrho, dgdrho[0],
107                     opt["move"])
108             else:
109                 rho_new, change, low, upp =
110                 mma_optimizer(
111                     num_consts, num_elems,
112                     opt_iter, rho_values, rho_min,
113                     rho_max,
114                     rho_old1, rho_old2,
115                     dJdrho, g_vec, dgdrho, low, upp,
116                     opt["move"])
117                 rho_old2 = rho_old1.copy()
118                 rho_old1 = rho_values.copy()
119                 rho_field.vector.array = rho_new
120                 .copy()
121
122             # Output the histories
123             opt_time = time.perf_counter() -
124             opt_start_time
125             if comm.rank == 0:
126                 print(f"opt_iter: {opt_iter
127                 }, opt_time: {opt_time:.3g} (s), "
128                 f"beta: {beta}, C: {
129                 C_value:.3f}, V: {V_value:.3f}, "
130                 f"U: {U_value:.3f},
131                 change: {change:.3f}", flush=True)
132
133             values = S_comm.gather(
134             rho_phys_field)
135             if comm.rank == 0:
136                 plotter.plot(values)
137                 save_xdmf(fem["mesh"],
138                 rho_phys_field)

```

### Appendix B.3 parameterize module for design space parameterization

```

1 import numpy as np
2 import ufl
3 from dolfinx import la
4 from dolfinx.fem import Function, form
5 from dolfinx.fem.petsc import
6     create_matrix, assemble_matrix
7 from petsc4py import PETSc
8
9 class DensityFilter():

```

```

10 def __init__(self, comm, rho,
    rho_tilde, R=1.0, petsc_options={}):
11     """Construct a PDE filter."""
12     # Initialization
13     S0, S = rho.function_space,
    rho_tilde.function_space
14     u0, u = ufl.TrialFunction(S0),
    ufl.TrialFunction(S)
15     v, self.af = ufl.TestFunction(S)
    , Function(S)
16
17     self.rho, self.rho_tilde = rho,
    rho_tilde
18     self.rho_tilde_wrap = la.
    create_petsc_vector_wrap(self.
    rho_tilde.x)
19     self.af_wrap = la.
    create_petsc_vector_wrap(self.af.x)
20     self.vec_s0, self.vec_s = rho.
    vector.copy(), rho_tilde.vector.
    copy()
21
22     # Construct Kf and T matrices
    based on the Helmholtz PDE
23     dx = ufl.Measure("dx", metadata
    ={"quadrature_degree": 2})
24     Kf_expr = (R**2*ufl.dot(ufl.grad
    (u), ufl.grad(v)) + u*v)*dx
25     T_expr = u0*v*dx
26     Kf_form, T_form = form(Kf_expr),
    form(T_expr)
27     Kf_mat, self.T_mat =
    create_matrix(Kf_form),
    create_matrix(T_form)
28
29     # Construct a filtering solver
30     self.solver = PETSc.KSP().create
    (comm)
31     self.solver.setOperators(Kf_mat)
32     prefix = f"filter_solver_{id(
    self)}"
33     self.solver.setOptionsPrefix(
    prefix)
34
35     # Apply PETSc options
36     opts = PETSc.Options()
37     opts.prefixPush(prefix)
38     for key, value in petsc_options.
    items():
39         opts[key] = value
40         opts.prefixPop()
41     self.solver.setFromOptions()
42     Kf_mat.setOptionsPrefix(prefix)
43     Kf_mat.setFromOptions()
44
45     # Assemble Kf and T matrices
46     assemble_matrix(Kf_mat, Kf_form)
47     Kf_mat.assemble()
48     assemble_matrix(self.T_mat,
    T_form)
49     self.T_mat.assemble()
50     self.T_mat_transpose = self.
    T_mat.copy()
51     self.T_mat_transpose.transpose()
52

```

```

53 def forward(self):
54     """Compute the filtered
    variables."""
55     self.T_mat.mult(self.rho.vector,
    self.vec_s)
56     self.solver.solve(self.vec_s,
    self.rho_tilde_wrap)
57     self.rho_tilde.x.scatter_forward
    ()
58     return self.rho_tilde
59
60 def backward(self, sf_vectors):
61     """Recover the sensitivities."""
62     values = []
63     for sf in sf_vectors:
64         if sf is not None:
65             self.solver.solve(sf,
    self.af_wrap)
66             self.af.x.
    scatter_forward()
67             self.T_mat_transpose.
    mult(self.af.vector, self.vec_s0)
68             values.append(self.
    vec_s0.array.copy())
69         else:
70             values.append(None)
71     return values
72
73
74 class Heaviside():
75     def __init__(self, rho_phys):
76         self.rho_phys = rho_phys
77
78     def forward(self, beta, eta=0.5):
79         denominator = np.tanh(beta*eta)
    + np.tanh(beta*(1-eta))
80         self.drho = beta*(1-np.tanh(beta
    *(self.rho_phys.vector-eta))**2) /
    denominator
81         self.rho_phys.vector.array = (
82             np.tanh(beta*eta)+np.tanh(
    beta*(self.rho_phys.vector-eta))) /
    denominator
83         self.rho_phys.x.scatter_forward
    ()
84
85     def backward(self, vectors):
86         for vector in vectors:
87             if vector is not None:
88                 vector.array *= self.
    drho

```

## Appendix B.4 fem module for FEA

```

1 import numpy as np
2 import ufl
3 from dolfinx.mesh import
    locate_entities_boundary, meshtags
4 from dolfinx.fem import (
    VectorFunctionSpace, FunctionSpace,
    Function, Constant,
5                                     dirichletbc,
    locate_dofs_topological)
6

```

```

7 from fenitop.utility import
  create_mechanism_vectors
8 from fenitop.utility import
  LinearProblem
9
10
11 def form_fem(fem, opt):
12     """Form an FEA problem."""
13     # Function spaces and functions
14     mesh = fem["mesh"]
15     V = VectorFunctionSpace(mesh, ("CG",
16     1))
17     S0 = FunctionSpace(mesh, ("DG", 0))
18     S = FunctionSpace(mesh, ("CG", 1))
19     u, v = ufl.TrialFunction(V), ufl.
20     TestFunction(V)
21     u_field = Function(V) #
22     Displacement field
23     lambda_field = Function(V) #
24     Adjoint variable field
25     rho_field = Function(S0) # Density
26     field
27     rho_phys_field = Function(S) #
28     Physical density field
29
30     # Material interpolation
31     E0, nu = fem["young's modulus"], fem
32     ["poisson's ratio"]
33     p, eps = opt["penalty"], opt["
34     epsilon"]
35     E = (eps + (1-eps)*rho_phys_field**p
36     ) * E0
37     _lambda, mu = E*nu/(1+nu)/(1-2*nu),
38     E/(2*(1+nu)) # Lamé constants
39
40     # Kinematics
41     def epsilon(u):
42         return ufl.sym(ufl.grad(u))
43
44     def sigma(u): # 3D or plane strain
45         return 2*mu*epsilon(u) + _lambda
46         *ufl.tr(epsilon(u))*ufl.Identity(
47         len(u))
48
49     # Boundary conditions
50     dim = mesh.topology.dim
51     fdim = dim - 1
52     disp_facets =
53     locate_entities_boundary(mesh, fdim
54     , fem["disp_bc"])
55     bc = dirichletbc(Constant(mesh, np.
56     full(dim, 0.0)),
57
58     locate_dofs_topological(V, fdim,
59     disp_facets), V)
60
61     tractions, facets, markers = [], [],
62     []
63     for marker, (traction, traction_bc)
64     in enumerate(fem["traction_bcs"]):
65         tractions.append(Constant(mesh,
66         np.array(traction, dtype=float)))
67         current_facets =
68         locate_entities_boundary(mesh, fdim
69         , traction_bc)
70         facets.extend(current_facets)
71
72         markers.extend([marker,]*len(
73         current_facets))
74     facets = np.array(facets, dtype=np.
75     int32)
76     markers = np.array(markers, dtype=np.
77     int32)
78     _, unique_indices = np.unique(facets
79     , return_index=True)
80     facets, markers = facets[
81     unique_indices], markers[
82     unique_indices]
83     sorted_indices = np.argsort(facets)
84     facet_tags = meshtags(mesh, fdim,
85     facets[sorted_indices], markers[
86     sorted_indices])
87
88     metadata = {"quadrature_degree": fem
89     ["quadrature_degree"]}
90     dx = ufl.Measure("dx", metadata=
91     metadata)
92     ds = ufl.Measure("ds", domain=mesh,
93     metadata=metadata, subdomain_data=
94     facet_tags)
95     b = Constant(mesh, np.array(fem["
96     body_force"], dtype=float))
97
98     # Establish the equilibrium and
99     adjoint equations
100     lhs = ufl.inner(sigma(u), epsilon(v)
101     )*dx
102     rhs = ufl.dot(b, v)*dx
103     for marker, t in enumerate(tractions
104     ):
105         rhs += ufl.dot(t, v)*ds(marker)
106     if opt["opt_compliance"]:
107         spring_vec = opt["l_vec"] = None
108     else:
109         spring_vec, opt["l_vec"] =
110         create_mechanism_vectors(
111             V, opt["in_spring"], opt["
112             out_spring"])
113     linear_problem = LinearProblem(
114         u_field, lambda_field, lhs, rhs,
115         opt["l_vec"],
116
117         spring_vec, [bc], fem["
118         petsc_options"])
119
120     # Define optimization-related
121     variables
122     opt["f_int"] = ufl.inner(sigma(
123         u_field), epsilon(v))*dx
124     opt["compliance"] = ufl.inner(sigma(
125         u_field), epsilon(u_field))*dx
126     opt["volume"] = rho_phys_field*dx
127     opt["total_volume"] = Constant(mesh,
128     1.0)*dx
129
130     return linear_problem, u_field,
131     lambda_field, rho_field,
132     rho_phys_field

```

## Appendix B.5 sensitivity module for sensitivity analysis

```

1 import ufl
2 from mpi4py import MPI
3 from dolfinx.fem import form,
  assemble_scalar
4 from dolfinx.fem.petsc import
  create_vector, create_matrix,
  assemble_vector, assemble_matrix
5 from petsc4py import PETSc
6
7
8 class Sensitivity():
9     def __init__(self, comm, opt,
10        problem, u_field, lambda_field,
11        rho_phys):
12         # Compliance
13         if opt["opt_compliance"]:
14             self.C_form = form(opt["compliance"])
15             self.dCdrho_form = form(-ufl.
16                derivative(opt["compliance"],
17                rho_phys))
18             self.dCdrho_vec = create_vector(
19                self.dCdrho_form)
20
21         # Volume
22         self.comm = comm
23         self.total_volume = comm.
24         allreduce(
25             assemble_scalar(form(opt["total_volume"])), op=MPI.SUM)
26         self.V_form = form(opt["volume"])
27
28         dVdrho_form = form(ufl.
29            derivative(opt["volume"], rho_phys))
30
31         self.dVdrho_vec = create_vector(
32            dVdrho_form)
33         assemble_vector(self.dVdrho_vec,
34            dVdrho_form)
35         self.dVdrho_vec.ghostUpdate(addv
36            =PETSc.InsertMode.ADD, mode=PETSc.
37            ScatterMode.REVERSE)
38         self.dVdrho_vec /= self.
39         total_volume
40
41         # Displacement
42         self.opt_compliance = opt["opt_compliance"]
43         if not self.opt_compliance:
44             self.dfdrho_form = form(ufl.
45                adjoint(ufl.derivative(opt["f_int"], rho_phys)))
46             self.dfdrho_mat =
47             create_matrix(self.dfdrho_form)
48             self.problem, self.l_vec =
49             problem, opt["l_vec"]
50             self.u_field, self.
51             lambda_field = u_field,
52             lambda_field
53             self.dUdrho_vec = rho_phys.
54             vector.copy()
55             self.prod_vec = u_field.
56             vector.copy()
57
58     def __del__(self):
59         if not self.opt_compliance:

```

```

39         self.prod_vec.destroy()
40
41     def evaluate(self):
42         # Compliance
43         if self.opt_compliance:
44             C_value = self.comm.
45             allreduce(assemble_scalar(self.
46                C_form), op=MPI.SUM)
47         else:
48             self.problem.lhs_mat.mult(
49                self.u_field.vector, self.prod_vec)
50             C_value = self.u_field.
51             vector.dot(self.prod_vec)
52             with self.dCdrho_vec.localForm()
53             as loc:
54                 loc.set(0)
55                 assemble_vector(self.dCdrho_vec,
56                    self.dCdrho_form)
57                 self.dCdrho_vec.ghostUpdate(addv
58                    =PETSc.InsertMode.ADD, mode=PETSc.
59                    ScatterMode.REVERSE)
60                 actual_volume = self.comm.
61                 allreduce(assemble_scalar(self.
62                    V_form), op=MPI.SUM)
63                 V_value = actual_volume / self.
64                 total_volume
65                 self.dVdrho_vec_copy = self.
66                 dVdrho_vec.copy()
67
68         # Displacement
69         if not self.opt_compliance:
70             U_value = self.u_field.
71             vector.dot(self.l_vec)
72             self.problem.solve_adjoint()
73             self.dfdrho_mat.zeroEntries
74             ()
75             assemble_matrix(self.
76                dfdrho_mat, self.dfdrho_form)
77             self.dfdrho_mat.assemble()
78             self.dfdrho_mat.mult(self.
79                lambda_field.vector, self.
80                dUdrho_vec)
81         else:
82             U_value, self.dUdrho_vec =
83             0, None
84
85         func_values = [C_value, V_value,
86            U_value]
87         sensitivities = [self.dCdrho_vec,
88            self.dVdrho_vec_copy, self.
89            dUdrho_vec]
90         return func_values,
91         sensitivities

```

## Appendix B.6 optimize module for solution update

```

1 import numpy as np
2 from mpi4py import MPI
3 from scipy import sparse as sparse
4 from scipy.linalg import solve
5
6

```

```

7 def optimality_criteria(rho, rho_min,
  rho_max, V, dCdrho, dVdrho, move
  =0.05):
8     """Solution update scheme with
  optimality criteria (OC)."""
9     lb, ub = 0.0, 1e6
10    comm = MPI.COMM_WORLD
11    while ub-lb > 1e-4:
12        mid = (lb+ub) / 2.0
13        rho_new = np.maximum.reduce([np.
  minimum.reduce(
14            [rho*(-dCdrho/(dVdrho+1e-12)
  /mid)**0.5, rho+move, rho_max]),
  rho-move, rho_min])
15        dV = comm.allreduce(dVdrho@
  rho_new-rho), op=MPI.SUM)
16        if V + dV > 0:
17            lb = mid
18        else:
19            ub = mid
20    change = comm.allreduce(np.max(np.
  abs(rho_new-rho), initial=0), op=
  MPI.MAX)
21    return rho_new, change
22
23 def mma_optimizer(m, n, opt_iter, xval,
  xmin, xmax, xold1, xold2, df0dx,
  fval,
24
25     dfdx, low, upp, a0=1,
  a=None, c=None, d=None, move=0.05,
26     asyinit=0.5, asydecr
  =0.7, asyincr=1.2,
27     low_bnd=0.002, up_bnd
  =1.0, albefa=0.1, feps=1e-6):
28     """Solution update scheme with the
  method of moving asymptotes (MMA).
  The algorithm is available in https
  ://doi.org/10.1002/nme.1620240207.
29
30     Minimize:
31         f_0(x) + a_0*z + sum(c_i*y_i +
  0.5*d_i*(y_i)^2)
32     Subjected to:
33         f_i(x) - a_i*z - y_i <= 0,      i
  = 1, 2, ..., m
34         xmin_j <= x_j <= xmax_j,      j
  = 1, 2, ..., n
35         y_i >= 0,                      i
  = 1, 2, ..., m
36         z >= 0
37
38     Args:
39         m: The number of general
  constraints.
40         n: The number of the variables,
  x_j.
41         opt_iter: Iteration counter.
42         xval: Current values of the
  variables, x_j.
43         xmin, xmax: Lower and upper
  bounds of the variables, x_j.
44         xold1, xold2: The values of x_j
  at one and two iterations ago.
45         df0dx: The derivatives of the
  objective function, f_0(x),
46
47         with respect to the
  variables, x_j, calculated at xval.
48         fval: The values of the
  constraint functions, f_i(x),
  calculated at xval.
49         dfdx: An (m, n) array with the
  derivatives of the constraint
  functions,
50         f_i(x), with respect to the
  variables, x_j, calculated at xval.
51         low, upp: Lower and upper
  asymptotes from the previous
  iteration.
52         a0, a, c, d: Coefficients in the
  objective function.
53         move: Move limit of the
  variables, x_j.
54         asyinit: Initial rate of the
  asymptotes.
55         asydecr: Decreasing rate of the
  asymptotes when the variables are
  oscillating.
56         asyincr: Increasing rate of the
  asymptotes when the variables are
  monotonically updated.
57         low_bnd, up_bnd: Lower and upper
  bounds for determining the
  asymptotes.
58         albefa: A parameter for
  determining the bounds of the
  variables.
59         feps: A parameter for
  approximating the objective
  function.
60
61     Returns:
62         x_new: Optimal values of the
  variables, x_j, in the current
  subproblem.
63         change: Maximum change of the
  variables, x_j.
64         low, upp: Lower and upper
  asymptotes calculated and used in
  the current subproblem.
65         """
66
67     # Initialize the coefficients of the
  objective function
68     if a is None:
69         a = np.zeros(m)
70     if c is None:
71         c = np.full(m, 1000)
72     if d is None:
73         d = np.zeros(m)
74     comm = MPI.COMM_WORLD
75     n_global = comm.allreduce(n, op=MPI.
  SUM)
76     epsimin = np.sqrt(m+n_global)*1e-9
77
78     # Evaluate the asymptotes (low and
  upp)
79     xrange = xmax - xmin
80     if opt_iter < 2.5:
81         low = xval - asyinit*xrange
82         upp = xval + asyinit*xrange
83     else:

```

```

84     idx = (xval-xold1) * (xold1-
85     xold2) # Check the oscillation
86     gamma = np.ones(n)
87     gamma[idx > 0] = asyincr #
88     Relax the asymptotes if monotonic
89     gamma[idx < 0] = asydecr #
90     Tighten the asymptotes if
91     oscillating
92     low = xval - gamma*(xold1-low)
93     upp = xval + gamma*(upp-xold1)
94     lowmin = xval - up_bnd*xrange
95     lowmax = xval - low_bnd*xrange
96     uppmix = xval + low_bnd*xrange
97     uppmix = xval + up_bnd*xrange
98     low = np.minimum(np.maximum(low,
99     lowmin), lowmax)
100     upp = np.maximum(np.minimum(upp,
101     uppmix), uppmix)
102
103 # Evaluate the bounds of the
104 # variables (alpha and beta)
105 alpha = np.maximum.reduce([xmin, low
106 +albepa*(xval-low), xval-move*
107 xrange])
108 beta = np.minimum.reduce([xmax, upp-
109 albepa*(upp-xval), xval+move*xrange
110 ])
111
112 # Evaluate the coefficients of the
113 # approximating objective fuction (p0
114 # and q0)
115 idx = df0dx > 0
116 p0, q0 = np.zeros(n), np.zeros(n)
117 p0[idx], p0[~idx] = 1.001*df0dx[idx
118 ], -0.001*df0dx[~idx]
119 q0[idx], q0[~idx] = 0.001*df0dx[idx
120 ], -1.001*df0dx[~idx]
121 u11 = upp - low
122 p0, q0 = (p0+fepr/u11)*(upp-xval)
123 **2, (q0+fepr/u11)*(xval-low)**2
124
125 # Evaluate the coefficients of the
126 # approximating constraint fuctions (
127 # P_mat and Q_mat)
128 P_mat, Q_mat = np.zeros((m, n)), np.
129 zeros((m, n))
130 dfdx = dfdx.reshape(m, n)
131 idx = dfdx > 0
132 P_mat[idx], Q_mat[~idx] = dfdx[idx],
133 -dfdx[~idx]
134 P_mat, Q_mat = P_mat*(upp-xval)**2,
135 Q_mat*(xval-low)**2
136
137 # Evaluate the b vector for
138 # constraint functions
139 b = P_mat@(1.0/(upp-xval)) + Q_mat@
140 (1.0/(xval-low))
141 b = comm.allreduce(b, op=MPI.SUM) -
142 fval
143
144 # Solve the subproblem with a primal-
145 # dual interior-point approach
146 x_new = solve_subproblem(m, epsimin,
147 low, upp, alpha, beta, p0, q0,
148 P_mat,
149 Q_mat, a0, a, b, c, d)
150
151 change = comm.allreduce(np.max(np.
152 abs(x_new-xval), initial=0), op=MPI
153 .MAX)
154 return x_new, change, low, upp
155
156 def solve_subproblem(m, epsimin, low,
157 upp, alpha, beta, p0, q0, P_mat,
158 Q_mat,
159 a0, a, b, c, d):
160 """Solve the MMA subproblem with a
161 primal-dual interior-point approach
162 .
163 Minimize:
164     sum[p0j/(uppj-xj) + q0j/(xj-lowj
165 )] + a0*z + sum(ci*yi + 0.5*di*yi
166 ^2)
167 Subjected to:
168     sum[p1j/(uppj-xj) + q1j/(xj-lowj
169 )] - ai*z - yi <= bi, i = 1, 2,
170 ..., m
171     alphaj <= xj <= betaj, j = 1,
172 2, ..., n
173     yi >= 0, i = 1,
174 2, ..., m
175     z >= 0
176 """
177 # Set the initial guess for
178 # variables and Lagrange multipliers
179 comm = MPI.COMM_WORLD
180 eps = 1.0
181 x = 0.5*(alpha+beta)
182 xi = np.maximum(1.0/(x-alpha), 1.0)
183 eta = np.maximum(1.0/(beta-x), 1.0)
184 _lambda = np.ones(m)
185 if comm.rank == 0:
186     y, z, s = np.ones(m), 1.0, np.
187 ones(m)
188 mu = np.maximum(0.5*c, 1.0)
189 zeta = 1.0
190
191 while eps > epsimin: # Primal-dual
192 interior point method
193     p_lambda = p0 + _lambda@P_mat
194     q_lambda = q0 + _lambda@Q_mat
195     g_vec = P_mat@(1.0/(upp-x)) +
196 Q_mat@(1.0/(x-low))
197     g_vec = comm.allreduce(g_vec, op
198 =MPI.SUM)
199     dpsid_x = p_lambda/(upp-x)**2 -
200 q_lambda/(x-low)**2
201
202     # Evaluate the residual norm
203     res_x = comm.gather(dpsid_x-xi+
204 eta, root=0)
205     res_xi = comm.gather(xi*(x-alpha
206 )-eps, root=0)
207     res_eta = comm.gather(eta*(beta-
208 x)-eps, root=0)
209     if comm.rank == 0:
210         res_x, res_xi, res_eta = np.
211 hstack(res_x), np.hstack(res_xi),
212 np.hstack(res_eta)
213         res_y = c + d*y - mu -
214 _lambda

```

```

165         res_z = a0 - zeta -
a@_lambda
166         res_lambda = g_vec - a*z - y
+ s - b
167         res_mu = mu*y - eps
168         res_zeta = zeta*z - eps
169         res_s = _lambda*s - eps
170         res = np.hstack([res_x,
res_y, res_z, res_lambda, res_xi,
res_eta,
171                             res_mu,
res_zeta, res_s])
172         res_norm = np.linalg.norm(
res, ord=2)
173         res_max = np.linalg.norm(res
, ord=np.inf)
174         else:
175             res_norm, res_max = None,
None
176             res_norm = comm.bcast(res_norm,
root=0)
177             res_max = comm.bcast(res_max,
root=0)
178
179             newton_iter = 0
180             while res_max > 0.9*eps and
newton_iter < 100: # Newton's
method for the search direction
181                 newton_iter += 1
182                 if newton_iter == 100 and
comm.rank == 0:
183                     print("Maximum inner
iterations reached in the MMA
optimizer", flush=True)
184
185                     # Evaluate the left-hand
side
186                     p_lambda = p0 +
_lambda@P_mat
187                     q_lambda = q0 +
_lambda@Q_mat
188                     G_mat = P_mat/(upp-x)**2 -
Q_mat/(x-low)**2
189                     G_mat_global = comm.gather(
G_mat, root=0)
190                     diag_x = 2*(p_lambda/(upp-x)
**3+q_lambda/(x-low)**3) + xi/(x-
alpha) + eta/(beta-x)
191                     diag_x_global = comm.gather(
diag_x, root=0)
192                     if comm.rank == 0:
193                         G_mat_global = np.hstack
(G_mat_global)
194                         diag_x_global = np.
hstack(diag_x_global)
195                         diag_y = d + mu/y
196                         A_mat = sparse.spdiags(s
/_lambda+1.0/diag_y, 0, m, m) \
+ G_mat_global*(1.0/
diag_x_global)@G_mat_global.T
197                         lhs = np.vstack([np.
hstack([A_mat, a.reshape(-1, 1)]),
np.
hstack([a, -zeta/z])])
200
201         # Evaluate the right-hand
side
202         dps_i_dx = p_lambda/(upp-x)
**2 - q_lambda/(x-low)**2
203         delta_x = dps_i_dx - eps/(x-
alpha) + eps/(beta-x)
204         delta_x_global = comm.gather
(delta_x, root=0)
205         g_vec = P_mat*(1.0/(upp-x))
+ Q_mat*(1.0/(x-low))
206         g_vec = comm.allreduce(g_vec
, op=MPI.SUM)
207         if comm.rank == 0:
208             delta_x_global = np.
hstack(delta_x_global)
209             delta_y = c + d*y -
_lambda - eps/y
210             delta_z = a0 - a@_lambda
- eps/z
211             delta_lambda = g_vec - a
*z - y - b + eps/_lambda
212             b_lambda = delta_lambda
+ delta_y/diag_y - G_mat_global@
(delta_x_global/diag_x_global)
213             rhs = np.hstack([
b_lambda, delta_z])
214
215         # Solve the search direction
216         if comm.rank == 0:
217             solution = solve(lhs,
rhs)
218             incr_lambda, incr_z =
solution[:m], solution[m]
219             incr_y = -delta_y/diag_y
+ incr_lambda/diag_y
220             incr_mu = -mu + eps/y -
(mu*incr_y)/y
221             incr_zeta = -zeta + eps/
z - zeta*incr_z/z
222             incr_s = -s + eps/
_lambda - (s*incr_lambda)/_lambda
223         else:
224             incr_lambda = None
225             incr_lambda = comm.bcast(
incr_lambda, root=0)
226             incr_x = -delta_x/diag_x - (
incr_lambda@G_mat)/diag_x
227             incr_xi = -xi + eps/(x-alpha)
- (xi*incr_x)/(x-alpha)
228             incr_eta = -eta + eps/(beta-
x) + (eta*incr_x)/(beta-x)
229             incr_xi_global = comm.gather
(incr_xi, root=0)
230             incr_eta_global = comm.
gather(incr_eta, root=0)
231             xi_global = comm.gather(xi,
root=0)
232             eta_global = comm.gather(eta
, root=0)
233             if comm.rank == 0:
234                 xi_global = np.hstack(
xi_global)
235                 eta_global = np.hstack(
eta_global)

```

```

236         values = np.hstack([y, z
237         , _lambda, xi_global, eta_global,
238         mu, zeta, s])
239         incr_xi_global = np.
240         hstack(incr_xi_global)
241         incr_eta_global = np.
242         hstack(incr_eta_global)
243         incr = np.hstack([incr_y
244         , incr_z, incr_lambda,
245         incr_xi_global,
246         incr_eta_global, incr_mu, incr_zeta
247         , incr_s])
248
249         # Evaluate the upper bound
250         of the step size
251         step_inv_alpha = np.max
252         (-1.01*incr_x/(x-alpha), initial=0)
253         # Ensure x-alpha >= 0.01/1.01*(
254         x_old-alpha)
255         step_inv_beta = np.max(1.01*
256         incr_x/(beta-x), initial=0) #
257         Ensure beta-x >= 0.01/1.01*(beta-
258         x_old)
259         step_inv_alpha = comm.
260         allreduce(step_inv_alpha, op=MPI.
261         MAX)
262         step_inv_beta = comm.
263         allreduce(step_inv_beta, op=MPI.MAX
264         )
265         if comm.rank == 0:
266             step_inv = max(-1.01*
267             incr/values) # Ensure y >=
268             0.01/1.01*y_old, etc.
269             step_inv = max(
270             step_inv_alpha, step_inv_beta,
271             step_inv, 1.0) # Ensure step <= 1
272             step = 1.0 / step_inv
273         else:
274             step = None
275             step = comm.bcast(step, root
276             =0)
277
278         x_old, lambda_old = x.copy()
279         , _lambda.copy()
280         xi_old, eta_old = xi.copy(),
281         eta.copy()
282         if comm.rank == 0:
283             y_old, z_old = y.copy(),
284             z
285             mu_old, zeta_old, s_old
286             = mu.copy(), zeta, s.copy()
287
288         search_iter = 0
289         res_norm_new = 2*res_norm
290         while res_norm_new >
291         res_norm and search_iter < 50: #
292         Line search for the step size
293         search_iter += 1
294
295         # Update the solutions
296         x = x_old + step*incr_x
297         _lambda = lambda_old +
298         step*incr_lambda
299         xi = xi_old + step*
300         incr_xi

```

```

301         eta = eta_old + step*
302         incr_eta
303         if comm.rank == 0:
304             y = y_old + step*
305             incr_y
306             z = z_old + step*
307             incr_z
308             mu = mu_old + step*
309             incr_mu
310             zeta = zeta_old +
311             step*incr_zeta
312             s = s_old + step*
313             incr_s
314
315         # Evaluate the residual
316         norm
317         p_lambda = p0 +
318         _lambda@P_mat
319         q_lambda = q0 +
320         _lambda@Q_mat
321         g_vec = P_mat@(1.0/(upp-
322         x)) + Q_mat@(1.0/(x-low))
323         g_vec = comm.allreduce(
324         g_vec, op=MPI.SUM)
325         dpsi_dx = p_lambda/(upp-
326         x)**2 - q_lambda/(x-low)**2
327
328         res_x = comm.gather(
329         dpsi_dx-xi+eta, root=0)
330         res_xi = comm.gather(xi
331         *(x-alpha)-eps, root=0)
332         res_eta = comm.gather(
333         eta*(beta-x)-eps, root=0)
334         if comm.rank == 0:
335             res_x, res_xi,
336             res_eta = np.hstack(res_x), np.
337             hstack(res_xi), np.hstack(res_eta)
338             res_y = c + d*y - mu
339             - _lambda
340             res_z = a0 - zeta -
341             a@_lambda
342             res_lambda = g_vec -
343             a*z - y + s - b
344             res_mu = mu*y - eps
345             res_zeta = zeta*z -
346             eps
347             res_s = _lambda*s -
348             eps
349             res = np.hstack([
350             res_x, res_y, res_z, res_lambda,
351             res_xi,
352             res_eta, res_mu, res_zeta, res_s])
353             res_norm_new = np.
354             linalg.norm(res, ord=2)
355         else:
356             res_norm_new = None
357             res_norm_new = comm.
358             bcast(res_norm_new, root=0)
359             step *= 0.5
360
361         res_norm = res_norm_new
362         res_max = np.linalg.norm(res
363         , ord=np.inf) if comm.rank == 0
364         else None

```

```

306         res_max = comm.bcast(res_max
307         , root=0)
308         eps *= 0.1
309     return x

```

## Appendix B.7 utility module for helper functions

```

1  import numpy as np
2  from scipy.spatial import cKDTree
3  from petsc4py import PETSc
4  import dolfinx.io
5  from dolfinx.fem import form, Function
6  from dolfinx import la
7  from dolfinx.fem.petsc import (
8      create_vector, create_matrix,
9      assemble_vector, assemble_matrix,
10     set_bc)
11 import pyvista
12 def create_mechanism_vectors(func_space
13     , in_spring, out_spring):
14     """Create vectors for compliant
15     mechanism design."""
16     index_map = func_space.dofmap.
17     index_map
18     block_size = func_space.dofmap.
19     index_map_bs
20     spring_vec = la.create_petsc_vector(
21     index_map, block_size)
22     l_vec = spring_vec.copy()
23     local_range = index_map.local_range
24     local_indices = np.arange(
25     local_range[0], local_range[1]).
26     astype(np.int32)
27     local_size = np.ptp(local_range)
28     local_nodes = func_space.
29     tabulate_dof_coordinates()[
30     local_size]
31     for n, (locator, direction, value)
32     in enumerate([in_spring, out_spring
33     ]):
34         ctrl_nodes = local_indices[
35         locator(local_nodes.T)]
36         offset = ["x", "y", "z"].index(
37         direction)
38         ctrl_dofs = ctrl_nodes*
39         block_size + offset
40         spring_vec.setValues(ctrl_dofs,
41         [value,]*ctrl_dofs.size)
42         if n == 1:
43             l_vec.setValues(ctrl_dofs,
44             [1.0,]*ctrl_dofs.size)
45         spring_vec.assemble()
46         l_vec.assemble()
47     return spring_vec, l_vec
48
49 class LinearProblem:

```

```

def __init__(self, u, lam, lhs, rhs,
    l_vec, spring_vec, bcs=[],
    petsc_options={}):
    """Initialize a linear problem.
    """
    # Initialization
    self.u, self.lam = u, lam
    self.u_wrap = la.
    create_petsc_vector_wrap(self.u.x)
    self.lam_wrap = la.
    create_petsc_vector_wrap(self.lam.x
    )
    self.lhs_form, self.rhs_form =
    form(lhs), form(rhs)
    self.lhs_mat = create_matrix(
    self.lhs_form)
    self.rhs_vec = create_vector(
    self.rhs_form)
    self.bcs, self.l_vec, self.
    spring_vec = bcs, l_vec, spring_vec
    # Construct a linear solver
    self.solver = PETSc.KSP().create
    (self.u.function_space.mesh.comm)
    self.solver.setOperators(self.
    lhs_mat)
    prefix = f"linear_solver_{id(
    self)}"
    self.solver.setOptionsPrefix(
    prefix)
    # Apply PETSc options
    opts = PETSc.Options()
    opts.prefixPush(prefix)
    for key, value in petsc_options.
    items():
        opts[key] = value
        opts.prefixPop()
        self.solver.setFromOptions()
        for var in [self.lhs_mat, self.
        rhs_vec, self.l_vec]:
            if var is not None:
                var.setOptionsPrefix(
                prefix)
                var.setFromOptions()
        assemble_vector(self.rhs_vec,
        self.rhs_form)
        self.rhs_vec.ghostUpdate(addv=
        PETSc.InsertMode.ADD, mode=PETSc.
        ScatterMode.REVERSE)
        set_bc(self.rhs_vec, self.bcs)
    def solve_fem(self):
        """Solve K*x=F for FEM."""
        self.lhs_mat.zeroEntries()
        assemble_matrix(self.lhs_mat,
        self.lhs_form, bcs=self.bcs)
        self.lhs_mat.assemble()
        if self.spring_vec is not None:
            self.lhs_mat.setDiagonal(
            self.lhs_mat.getDiagonal()+self.
            spring_vec)
            self.solver.solve(self.rhs_vec,
            self.u_wrap)
            self.u.x.scatter_forward()

```

```

80
81 def solve_adjoint(self):
82     """Solve  $K \cdot \lambda = -L$  for the
83     adjoint equation."""
84     self.solver.solve(-self.l_vec,
85 self.lam_wrap)
86     self.lam.x.scatter_forward()
87
88 def __del__(self):
89     self.solver.destroy()
90     self.lhs_mat.destroy()
91     self.rhs_vec.destroy()
92     self.u_wrap.destroy()
93     self.lam_wrap.destroy()
94     if self.spring_vec is not None:
95         self.spring_vec.destroy()
96         self.l_vec.destroy()
97
98 class Communicator():
99     """Communicate information among
100     different processes."""
101
102 def __init__(self, func_space,
103 mesh_serial, size=1):
104     self.size = size
105     self.comm = func_space.mesh.comm
106     idx_map = func_space.dofmap.
107 index_map
108
109     num_local_nodes = idx_map.
110 size_local
111     num_global_nodes = idx_map.
112 size_global
113     num_nodal_dofs = func_space.
114 dofmap.index_map_bs
115     self.num_global_dofs =
116 num_global_nodes * num_nodal_dofs
117
118     local_nodal_range = np.asarray(
119 idx_map.local_range, dtype=np.int32
120 ) # [start, end]
121     local_dof_range =
122 local_nodal_range * num_nodal_dofs
123     # [start, end]
124     local_nodes = func_space.
125 tabulate_dof_coordinates()[
126 num_local_nodes]
127
128     # Gather to Process 0
129     local_nodal_range_gather = self.
130 comm.gather(local_nodal_range, root
131 =0)
132     self.local_dof_range_gather =
133 self.comm.gather(local_dof_range,
134 root=0)
135     local_nodes_gather = self.comm.
136 gather(local_nodes, root=0)
137
138     element = func_space.ufl_element
139 ()
140     if self.comm.rank == 0:
141         func_space_serial = dolfinx.
142 fem.FunctionSpace(mesh_serial,
143 element)
144
145         nodes_serial =
146 func_space_serial.
147 tabulate_dof_coordinates()
148
149         nodes_collect = np.zeros((
150 num_global_nodes, 3))
151         for r, nodes in zip(
152 local_nodal_range_gather,
153 local_nodes_gather):
154             nodes_collect[r[0]:r[1]]
155 = nodes
156         global_to_local_nodes =
157 compare_matrices(nodes_serial,
158 nodes_collect)
159         local_to_global_nodes =
160 compare_matrices(nodes_collect,
161 nodes_serial)
162
163         def node2dof(nodes,
164 num_nodal_dofs):
165             return (np.tile(nodes, (
166 num_nodal_dofs, 1))*num_nodal_dofs
167 + np.arange(
168 num_nodal_dofs).reshape(-1, 1)).
169 ravel("F")
170
171         global_to_local_dofs =
172 node2dof(global_to_local_nodes,
173 num_nodal_dofs)
174         self.local_to_global_dofs =
175 node2dof(local_to_global_nodes,
176 num_nodal_dofs)
177         self.local_to_global_dofs =
178 (
179 np.tile(self.
180 local_to_global_dofs.reshape(-1, 1)
181 , (1, size))*size + np.arange(size)
182 ).ravel()
183         else:
184             global_to_local_dofs = None
185             global_to_local_dofs = self.comm
186 .bcast(global_to_local_dofs, root
187 =0)
188             self.idx = global_to_local_dofs[
189 local_dof_range[0]:local_dof_range
190 [1]]
191
192 def bcast(self, func, global_values)
193 :
194     """Broadcast data from Process 0
195     to all the other processes."""
196     if func.vector.size !=
197 global_values.size:
198         raise ValueError("Mismatched
199 sizes.")
200     func.vector.array =
201 global_values[self.idx]
202
203 def gather(self, func):
204     """Gather data to Process 0 from
205     all the other processes."""
206     if type(func) is Function:
207         values_gather = self.comm.
208 gather(func.vector.array, root=0)
209     elif type(func) is PETSc.Vec:

```

```

156         values_gather = self.comm.
157         gather(func.array, root=0)
158         elif type(func) is np.ndarray:
159             values_gather = self.comm.
160             gather(func, root=0)
161         else:
162             raise TypeError("Unsupported
163             func.")
164
165         if self.comm.rank == 0:
166             values_collect = np.zeros(
167             self.num_global_dofs*self.size)
168             for r, local_values in zip(
169             self.local_dof_range_gather,
170             values_gather):
171                 values_collect[r[0]*self
172                 .size:r[1]*self.size] =
173                 local_values
174             global_values =
175             values_collect[self.
176             local_to_global_dofs]
177         else:
178             global_values = None
179         return global_values
180
181 def compare_matrices(array1, array2,
182 precision=12, k=1):
183     """Find the "args" such that array1[
184     args] == array2."""
185     kd_tree = cKDTree(array1.round(
186     precision))
187     return kd_tree.query(array2.round(
188     precision), k=k)[1]
189
190 class Plotter():
191     def __init__(self, mesh):
192         """Initialize a plotter."""
193         pyvista.OFF_SCREEN = True
194         pyvista.start_xvfb()
195         self.dim = mesh.topology.dim
196         elements, cell_types, nodes =
197         dolfinx.plot.vtk_mesh(mesh, self.
198         dim)
199         self.grid = pyvista.
200         UnstructuredGrid(elements,
201         cell_types, nodes)
202
203     def plot(self, density, threshold
204     =0.49, smooth_iter=100, path=""):
205         self.grid.point_data["density"]
206         = np.hstack(density)
207         if self.dim == 2:
208             grid = self.grid
209         else:
210             grid = self.grid.threshold(
211             threshold).extract_surface()
212             empty_mesh = (self.dim == 3 and
213             grid.n_faces_strict == 0)
214
215             if not empty_mesh:
216                 if self.dim == 3:
217                     grid = grid.smooth(
218                     n_iter=smooth_iter)
219
220             grid.point_data["density
221             "] = 0.4
222             plotter = pyvista.Plotter()
223             plotter.background_color = "
224             white"
225             lighting = self.dim == 3
226             plotter.add_mesh(grid, clim
227             =[0, 1], cmap="Greys", lighting=
228             lighting,
229             show_scalar_bar=False)
230             if self.dim == 2:
231                 plotter.view_xy()
232                 plotter.screenshot(path+"
233                 optimized_design.jpg", window_size
234                 =(1000, 1000))
235                 plotter.close()
236
237     def save_xdmf(mesh, rho, path=""):
238         xdmf = dolfinx.io.XDMFFile(mesh.comm
239         , path+"optimized_design.xdmf", "w"
240         )
241         xdmf.write_mesh(mesh)
242         rho.name = "density"
243         xdmf.write_function(rho)

```

**Acknowledgements** Authors X.S.Z., Y.J., and C.W. acknowledge the financial support from the U.S. National Science Foundation (NSF) EAGER Award CMMI-2127134, U.S. Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (N660012314013), NSF CAREER Award CMMI-2047692, and NSF Award CMMI-2245251. The information provided in this paper is the sole opinion of the authors and does not necessarily reflect the view of the sponsoring agencies.

## Declarations

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

**Replication of results** The open-source FEniTop is given in Appendix B and available to download at <https://github.com/missionlab/fenitop>. The information required to replicate the results contained herein has been provided in this manuscript. Should additional information be required, please contact the authors directly.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Aage N, Andreassen E, Lazarov BS, Sigmund O (2017) Giga-voxel computational morphogenesis for structural design. *Nature* 550(7674):84–86. <https://doi.org/10.1038/nature23911>
- Aage N, Andreassen E, Lazarov BS (2015) Topology optimization using PETSc: an easy-to-use, fully parallel, open source topology optimization framework. *Struct Multidisc Optim* 51(3):565–572. <https://doi.org/10.1007/s00158-014-1157-0>
- Alnæs MS, Logg A, Ølgaard KB, Rognes ME, Wells GN (2014) Unified form language: a domain-specific language for weak formulations of partial differential equations. *ACM Trans Math Softw* 40(2):9:1–9:37. <https://doi.org/10.1145/2566630>
- Anderson C (2015) Docker [software engineering]. *IEEE Softw* 32(3):102–c3. <https://doi.org/10.1109/MS.2015.62>
- Andreassen E, Clausen A, Schevenels M, Lazarov BS, Sigmund O (2011) Efficient topology optimization in MATLAB using 88 lines of code. *Struct Multidisc Optim* 43(1):1–16. <https://doi.org/10.1007/s00158-010-0594-7>
- Ayachit U (2015) The paraview guide: a parallel visualization application. Kitware Inc, Clifton Park
- Balay S, Abhyankar S, Adams M, Brown J, Brune P, Buschelman K, Dalcin L, Dener A, Eijkhout V, Gropp W, Karpeyev D, Kaushik D, Knepley M, May D, McInnes L, Mills R, Munson T, Rupp K, Sanan P, Smith B, Zampini S, Zhang H (2019) PETSc users manual. Argonne National Laboratory, Lemont
- Beghini LL, Beghini A, Katz N, Baker WF, Paulino GH (2014) Connecting architecture and engineering through structural topology optimization. *Eng Struct* 59:716–726. <https://doi.org/10.1016/j.engstruct.2013.10.032>
- Bendsøe MP, Kikuchi N (1988) Generating optimal topologies in structural design using a homogenization method. *Comput Methods Appl Mech Eng* 71(2):197–224. [https://doi.org/10.1016/0045-7825\(88\)90086-2](https://doi.org/10.1016/0045-7825(88)90086-2)
- Bendsøe MP, Sigmund O (2003) Topology optimization: theory, methods, and applications. Springer, Berlin
- Chandrasekhar A, Suresh K (2021) TOuNN: topology optimization using neural networks. *Struct Multidisc Optim* 63(3):1135–1149. <https://doi.org/10.1007/s00158-020-02748-4>
- Dalcín L, Paz R, Storti M (2005) MPI for Python. *J Parallel Distrib Comput* 65(9):1108–1115. <https://doi.org/10.1016/j.jpdc.2005.03.010>
- Dalcín LD, Paz RR, Kler PA, Cosimo A (2011) Parallel distributed computing using python. *Adv Water Resour* 34(9):1124–1139. *New Computational Methods and Software Tools*. <https://doi.org/10.1016/j.advwatres.2011.04.013>
- Duarte LS, Celes W, Pereira A, Menezes IFM, Paulino GH (2015) PolyTop++: an efficient alternative for serial and parallel topology optimization on CPUs & GPUs. *Struct Multidisc Optim* 52(5):845–859. <https://doi.org/10.1007/s00158-015-1252-x>
- Ferrari F, Sigmund O (2020) A new generation 99 line Matlab code for compliance topology optimization and its extension to 3D. *Struct Multidisc Optim* 62(4):2211–2228. <https://doi.org/10.1007/s00158-020-02629-w>
- Giraldo-Londoño O, Paulino GH (2021a) PolyDyna: a Matlab implementation for topology optimization of structures subjected to dynamic loads. *Struct Multidisc Optim* 64(2):957–990. <https://doi.org/10.1007/s00158-021-02859-6>
- Giraldo-Londoño O, Paulino GH (2021b) PolyStress: a Matlab implementation for local stress-constrained topology optimization using the augmented Lagrangian method. *Struct Multidisc Optim* 63(4):2065–2097. <https://doi.org/10.1007/s00158-020-02760-8>
- Hassani B, Hinton E (1998) A review of homogenization and topology optimization III—topology optimization using optimality criteria. *Comput Struct* 69(6):739–756. [https://doi.org/10.1016/S0045-7949\(98\)00133-3](https://doi.org/10.1016/S0045-7949(98)00133-3)
- Hestenes MR, Stiefel E (1952) Methods of conjugate gradients for solving linear systems. *J Res Natl Bureau Stand* 49(6):409–436
- Hoffman J, Jansson J, Jansson N (2016) FEniCS-HPC: automated predictive high-performance finite element computing with applications in aerodynamics. In: Wyrzykowski R, Deelman E, Dongarra J, Karczewski K, Kitowski J, Wiatr K (eds) *Parallel processing and applied mathematics. Lecture Notes in Computer Science*. Springer, Cham, pp 356–365. [https://doi.org/10.1007/978-3-319-32149-3\\_34](https://doi.org/10.1007/978-3-319-32149-3_34)
- Homayouni-Amlashi A, Schlienger T, Mohand-Ousaid A, Rakotondrabe M (2021) 2D topology optimization MATLAB codes for piezoelectric actuators and energy harvesters. *Struct Multidisc Optim* 63(2):983–1014. <https://doi.org/10.1007/s00158-020-02726-w>
- Hughes TJR (2012) The finite element method: linear static and dynamic finite element analysis. Courier Corporation, Chelmsford
- Jia Y, Lopez-Pamies O, Zhang XS (2023) Controlling the fracture response of structures via topology optimization: from delaying fracture nucleation to maximizing toughness. *J Mech Phys Solids* 173:105227. <https://doi.org/10.1016/j.jmps.2023.105227>
- Jia Y, Liu K, Zhang XS (2024a) Modulate stress distribution with bio-inspired irregular architected materials towards optimal tissue support. *Nat Commun* 15:4072. <https://doi.org/10.1038/s41467-024-47831-2>
- Jia Y, Liu K, Zhang XS (2024b) Topology optimization of irregular multiscale structures with tunable responses using a virtual growth rule. *Comput Methods Appl Mech Eng* 425:116864. <https://doi.org/10.1016/j.cma.2024.116864>
- Kirby RC, Logg A (2006) A compiler for variational forms. *ACM Trans Math Softw* 32(3):417–444. <https://doi.org/10.1145/1163641.1163644>
- Kurtzer GM, Sochat V, Bauer MW (2017) Singularity: scientific containers for mobility of compute. *PLoS ONE* 12(5):e0177459. <https://doi.org/10.1371/journal.pone.0177459>
- Langtangen HP, Logg A (2016) Solving PDEs in Python: the FEniCS tutorial I. Springer, Cham. <https://doi.org/10.1007/978-3-319-52462-7>
- Lazarov BS, Sigmund O (2011) Filters in topology optimization based on Helmholtz-type differential equations. *Int J Numer Methods Eng* 86(6):765–781. <https://doi.org/10.1002/nme.3072>
- Li W, Jia Y, Wang F, Sigmund O, Zhang XS (2023) Programming and physical realization of extreme three-dimensional responses of metastructures under large deformations. *Int J Eng Sci* 191:103881. <https://doi.org/10.1016/j.ijengsci.2023.103881>
- Liu K, Tovar A (2014) An efficient 3D topology optimization code written in Matlab. *Struct Multidisc Optim* 50(6):1175–1196. <https://doi.org/10.1007/s00158-014-1107-x>
- Logg A, Wells GN (2010) DOLFIN: automated finite element computing. *ACM Trans Math Softw* 37(2):20:1–20:28. <https://doi.org/10.1145/1731022.1731030>
- Saad Y, Schultz MH (1986) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J Sci Stat Comput* 7(3):856–869. <https://doi.org/10.1137/0907058>
- Sanders ED, Pereira A, Aguiló MA, Paulino GH (2018) PolyMat: an efficient Matlab code for multi-material topology optimization. *Struct Multidisc Optim* 58(6):2727–2759. <https://doi.org/10.1007/s00158-018-2094-0>
- Scroggs MW, Dokken JS, Richardson CN, Wells GN (2022) Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. *ACM Trans Math Softw* 48(2):18:1–18:23. <https://doi.org/10.1145/3524456>
- Sigmund O (2001) A 99 line topology optimization code written in Matlab. *Struct Multidisc Optim* 21(2):120–127. <https://doi.org/10.1007/s001580050176>

- Sigmund O (2007) Morphology-based black and white filters for topology optimization. *Struct Multidisc Optim* 33(4):401–424. <https://doi.org/10.1007/s00158-006-0087-x>
- Smit T, Aage N, Ferguson SJ, Helgason B (2021) Topology optimization using PETSc: a Python wrapper and extended functionality. *Struct Multidisc Optim* 64(6):4343–4353. <https://doi.org/10.1007/s00158-021-03018-7>
- Stüben K (2001) A review of algebraic multigrid. In: Brezinski C, Wuytack L (eds) *Numerical analysis: historical developments in the 20th century*. Elsevier, Amsterdam, pp 331–359. <https://doi.org/10.1016/B978-0-444-50617-7.50015-X>
- Svanberg K (1987) The method of moving asymptotes—a new method for structural optimization. *Int J Numer Methods Eng* 24(2):359–373. <https://doi.org/10.1002/nme.1620240207>
- Talisci C, Paulino GH, Pereira A, Menezes IFM (2012) PolyTop: a Matlab implementation of a general topology optimization framework using unstructured polygonal finite element meshes. *Struct Multidisc Optim* 45(3):329–357. <https://doi.org/10.1007/s00158-011-0696-x>
- Träff EA, Rydahl A, Karlsson S, Sigmund O, Aage N (2023) Simple and efficient GPU accelerated topology optimisation: codes and applications. *Comput Methods Appl Mech Eng* 410:116043. <https://doi.org/10.1016/j.cma.2023.116043>
- Wallin M, Ivarsson N, Amir O, Tortorelli D (2020) Consistent boundary conditions for PDE filter regularization in topology optimization. *Struct Multidisc Optim* 62(3):1299–1311. <https://doi.org/10.1007/s00158-020-02556-w>
- Wang F, Lazarov BS, Sigmund O (2011) On projection methods, convergence and robust formulations in topology optimization. *Struct Multidisc Optim* 43(6):767–784. <https://doi.org/10.1007/s00158-010-0602-y>
- Wang C, Zhao Z, Zhou M, Sigmund O, Zhang XS (2021) A comprehensive review of educational articles on structural and multidisciplinary optimization. *Struct Multidisc Optim*. <https://doi.org/10.1007/s00158-021-03050-7>
- Wang C, Zhao Z, Zhang XS (2023) Inverse design of magneto-active metasurfaces and robots: theory, computation, and experimental validation. *Comput Methods Appl Mech Eng* 413:116065. <https://doi.org/10.1016/j.cma.2023.116065>
- Xia L, Breitkopf P (2015) Design of materials using topology optimization and energy-based homogenization approach in Matlab. *Struct Multidisc Optim* 52(6):1229–1241. <https://doi.org/10.1007/s00158-015-1294-0>
- Zargham S, Ward TA, Ramli R, Badruddin IA (2016) Topology optimization: a review for structural designs under vibration problems. *Struct Multidisc Optim* 53(6):1157–1177. <https://doi.org/10.1007/s00158-015-1370-5>
- Zhang Z-D, Ibhade O, Bonakdar A, Toyserkani E (2021) TopADD: a 2D/3D integrated topology optimization parallel-computing framework for arbitrary design domains. *Struct Multidisc Optim* 64(3):1701–1723. <https://doi.org/10.1007/s00158-021-02917-z>
- Zhu B, Zhang X, Zhang H, Liang J, Zang H, Li H, Wang R (2020) Design of compliant mechanisms using continuum topology optimization: a review. *Mech Mach Theory* 143:103622. <https://doi.org/10.1016/j.mechmachtheory.2019.103622>

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.