Structural and Multidisciplinary Optimization (2024) 67:140
https://doi.org/10.1007/s00158-024-03818-7

EDUCATIONAL ARTICLE

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

Yingqi Jia[1] · **Chao Wang**[1] · **Xiaojia Shelly Zhang**[1,2,3]

Received: 22 November 2023 / Revised: 3 March 2024 / Accepted: 15 May 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

[1] Department of Civil and Environmental Engineering, University of Illinois Urbana-Champaign, Urbana, IL 61801, USA

[2] Department of Mechanical Science and Engineering, University of Illinois Urbana-Champaign, Urbana, IL 61801, USA

[3] 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-α 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

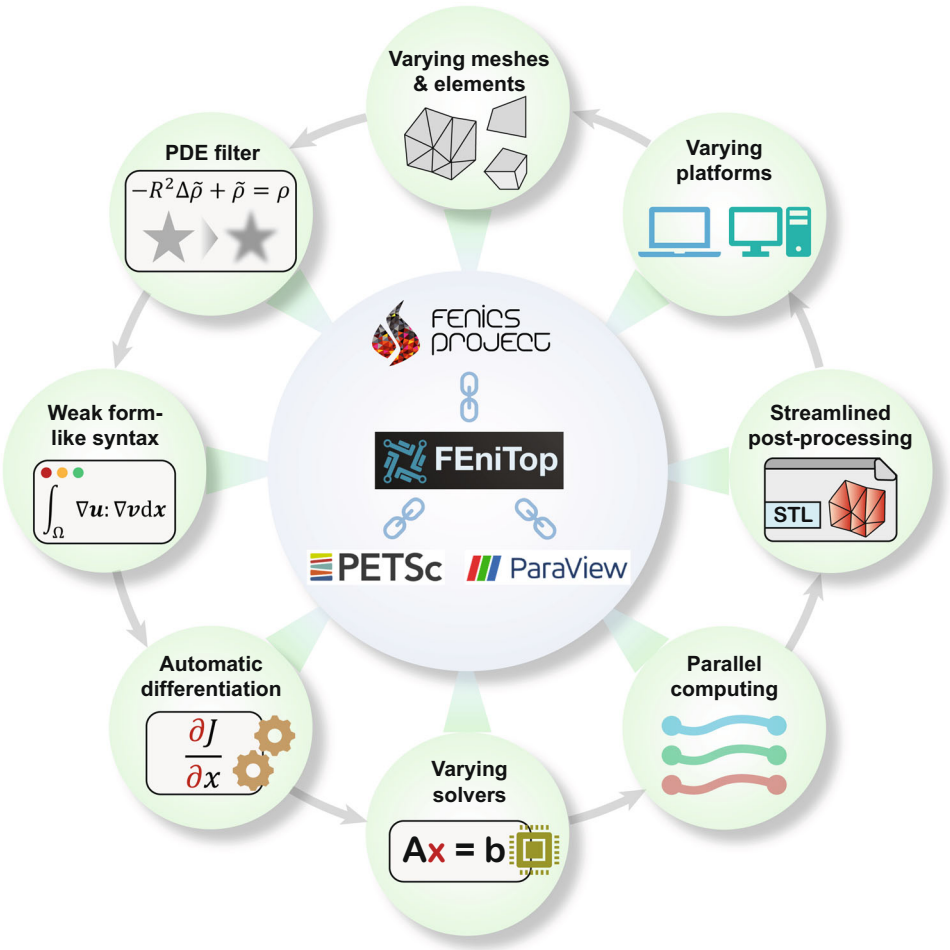


Fig. 1 Illustration of the innovative `FEniTop` software for facilitating streamlined topology optimization

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

$$\begin{cases} \underset{\rho\in[0,1]}{\text{minimize}}: & C(\overline{\rho},\mathbf{u}(\overline{\rho})) = \displaystyle\int_\Omega \boldsymbol{\sigma}(\overline{\rho},\mathbf{u}(\overline{\rho})) : \boldsymbol{\epsilon}(\mathbf{u}(\overline{\rho}))\mathrm{d}\mathbf{X}, \\ \text{subjected to}: & V(\overline{\rho}) = \dfrac{1}{|\Omega|}\displaystyle\int_\Omega \overline{\rho}\mathrm{d}\mathbf{X} - \overline{V} \le 0, \\ \text{with}: & \displaystyle\int_\Omega \boldsymbol{\sigma}(\overline{\rho},\mathbf{u}(\overline{\rho})) : \boldsymbol{\epsilon}(\mathbf{v})\mathrm{d}\mathbf{X} = \int_\Omega \overline{\mathbf{b}}\cdot\mathbf{v}\mathrm{d}\mathbf{X} \\ & \quad + \int_{\partial\Omega_{\mathcal{N}}} \overline{\mathbf{t}}\cdot\mathbf{v}\mathrm{d}\mathbf{X}. \end{cases} \tag{1}$$

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 $\overline{\rho} \in [0, 1]$ signifies the physical density field, where $\overline{\rho} = 1$ denotes solid material and $\overline{\rho} = 0$ indicates void. The variable $\mathbf{u}$ is the displacement field, and the symbol Ω denotes the design domain. The variables $\boldsymbol{\sigma}$ and $\boldsymbol{\epsilon} = (\nabla\mathbf{u} + \nabla\mathbf{u}^\top)/2$ stand for the infinitesimal stress and strain tensors, respectively. The symbol V represents the material volume constraint, with $\overline{V} \in (0, 1]$ specifying the allowable material volume fraction. The vector $\mathbf{v}$ represents the test displacement field, and the variables $\overline{\mathbf{b}}$ and $\overline{\mathbf{t}}$ denote the applied body force defined in the domain (Ω) and the traction defined on the domain's Neumann boundary ($\partial\Omega_{\mathcal{N}}$), respectively.

As for the compliant mechanism problem, it is expressed as

$$\begin{cases} \underset{\rho\in[0,1]}{\text{minimize}}: & U(\mathbf{u}(\overline{\rho})) = \mathbf{U}(\mathbf{u}(\overline{\rho}))\cdot\mathbf{L}, \\ \text{subjected to}: & C(\overline{\rho},\mathbf{u}(\overline{\rho})) - \overline{C} \le 0, \\ & V(\overline{\rho}) = \dfrac{1}{|\Omega|}\displaystyle\int_\Omega \overline{\rho}\mathrm{d}\mathbf{X} - \overline{V} \le 0, \\ \text{with}: & \displaystyle\int_\Omega \boldsymbol{\sigma}(\overline{\rho},\mathbf{u}(\overline{\rho})) : \boldsymbol{\epsilon}(\mathbf{v})\mathrm{d}\mathbf{X} = \int_\Omega \overline{\mathbf{b}}\cdot\mathbf{v}\mathrm{d}\mathbf{X} \\ & \quad + \int_{\partial\Omega_{\mathcal{N}}} \overline{\mathbf{t}}\cdot\mathbf{v}\mathrm{d}\mathbf{X}. \end{cases} \tag{2}$$

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 $\overline{C}$ 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), $\overline{\rho}$, can be obtained from the Heaviside projection (Wang et al. 2011), which is expressed as

$$\overline{\rho} = \frac{\tanh(\beta\theta) + \tanh(\beta(\widetilde{\rho} - \theta))}{\tanh(\beta\theta) + \tanh(\beta(1 - \theta))} \tag{3}$$

where the coefficients β and $\theta = 0.5$ are the sharpness parameter and the projection threshold value, respectively. The variable $\widetilde{\rho} \in [0, 1]$ 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

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

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)

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

where λ and μ are Lamé constants and can be calculated as

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

The variables E and ν 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

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

Here, E^0 and ν^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., $\overline{\rho} \in (0, 1)$).

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

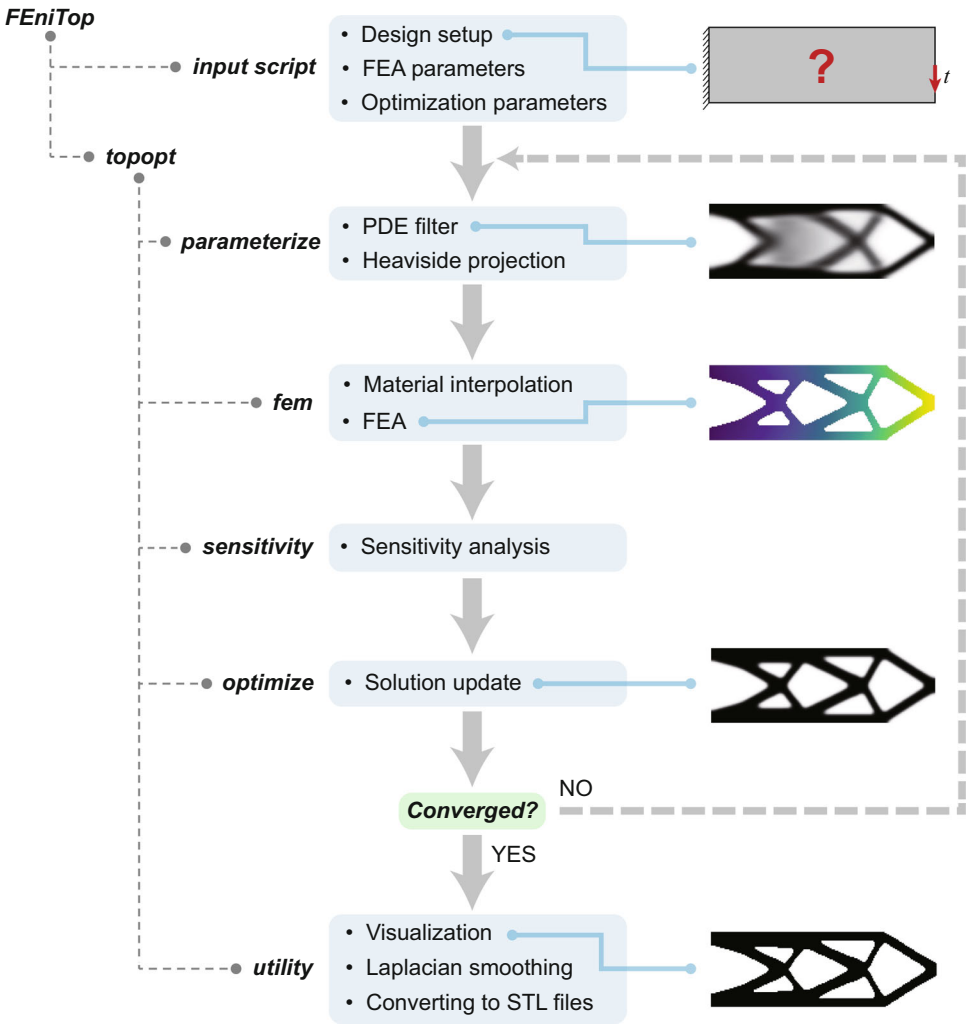


Fig. 2 Illustration of the `FEniTop` implementation for topology optimization

```
mesh = create_rectangle(MPI.COMM_WORLD, [[0, 0], [60, 20]],
                        [200, 60], CellType.quadrilateral)
if MPI.COMM_WORLD.rank == 0:
    mesh_serial = create_rectangle(MPI.COMM_SELF, [[0, 0], [60, 20]],
    [200, 60], CellType.quadrilateral)
else:
    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 (ν^0)
`"disp_bc"`	`lambda` function	Locator of displacement boundaries ($\partial\Omega_{\mathcal{D}} = \partial\Omega \setminus \partial\Omega_{\mathcal{N}}$)
`"traction_bcs"`	`List` of `Lists`	Traction values ($\bar{\mathbf{t}}$) and locators of traction boundaries ($\partial\Omega_{\mathcal{N}}$)
`"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

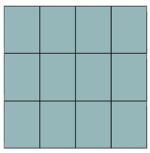
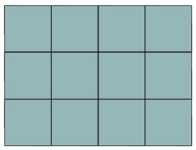
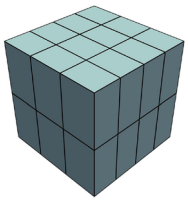
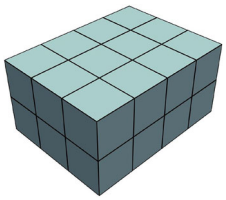
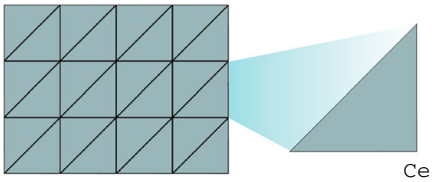
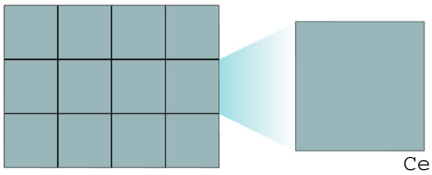
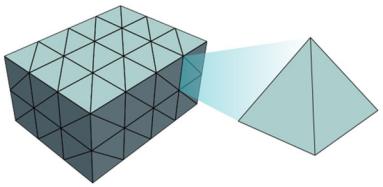
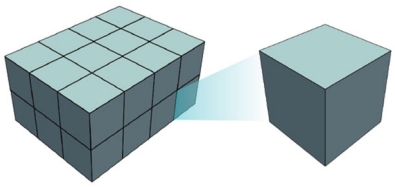
Design domains	Commands	Descriptions
	`create_unit_square( MPI.COMM_WORLD, nx, ny, cell_type)`	• Create a 2D unit square • `nx`, `ny`: the element numbers in x- and y-directions, respectively • `cell_type`: the element geometry provided in Table 3
	`create_rectangle( MPI.COMM_WORLD, [[0, 0], [lx, ly]], [nx, ny], cell_type)`	• Create a 2D rectangle • `lx`, `ly`: the edge lengths in x- and y-directions, respectively • `nx`, `ny`: the element numbers in x- and y-directions, respectively • `cell_type`: the element geometry provided in Table 3
	`create_unit_cube( MPI.COMM_WORLD, nx, ny, nz, cell_type)`	• Create a 3D unit cube • `nx`, `ny`, `nz`: the element numbers in x-, y-, and z-directions, respectively • `cell_type`: the element geometry provided in Table 3
	`create_box( MPI.COMM_WORLD, [[0, 0, 0], [lx, ly, lz]], [nx, ny, nz], cell_type)`	• Create a 3D box • `lx`, `ly`, `lz`: the edge lengths in x-, y-, and z-directions, respectively • `nx`, `ny`, `nz`: the element numbers in x-, y-, and z-directions, respectively • `cell_type`: the element geometry provided in Table 3

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

Element geometries	Commands	Descriptions
	`CellType.triangle`	Triangular element
	`CellType.quadrilateral`	Quadrilateral element
	`CellType.tetrahedron`	Tetrahedral element
	`CellType.hexahedron`	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 ν^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 ($\partial\Omega_{\mathcal{D}} = \partial\Omega \backslash \partial\Omega_{\mathcal{N}}$) by defining a `lambda` function in `fem["disp_bc"]` as

```
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

```
[[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 ($\partial\Omega_{\mathcal{N}}$), as detailed in Sect. 4.4. In the current example, we define a single traction boundary condition as

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

```
{"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 `topopt` module with the command as

```
if __name__ == "__main__":
    topopt(fem, opt)
```

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

3.2 `topopt` module for topology optimization

The `topopt` 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 `topopt` 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.

```
# Initialization
comm = MPI.COMM_WORLD
linear_problem, u_field,
lambda_field, rho_field,
rho_phys_field = form_fem(fem, opt)
density_filter = DensityFilter(comm,
 rho_field, rho_phys_field, opt["
filter_radius"], fem["petsc_options
"])
heaviside = Heaviside(rho_phys_field
)
sens_problem = Sensitivity(comm, opt
, linear_problem, u_field,
lambda_field, rho_phys_field)
S_comm = Communicator(rho_phys_field
.function_space, fem["mesh_serial"
])
if comm.rank == 0:
    plotter = Plotter(fem["
mesh_serial"])
num_consts = 1 if opt["
opt_compliance"] else 2
num_elems = rho_field.vector.array.
size
if not opt["use_oc"]:
    rho_old1, rho_old2, = np.zeros(
num_elems), np.zeros(num_elems)
    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 (ε^{opt})
`"vol_frac"`	0.5	Upper bound of the material volume fraction ($\overline{V}$)
`"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 (ε) 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 (β^{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.

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

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`), we apply the Heavisdie 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.

```
# Start topology optimization
opt_iter, beta, change = 0, 1, 2*opt["opt_tol"]
while opt_iter < opt["max_iter"] and change > opt["opt_tol"]:
    opt_start_time = time.perf_counter()
    opt_iter += 1

    # Density filter and Heaviside projection
    density_filter.forward()
    if opt_iter % opt["beta_interval"] == 0 and beta < opt["beta_max"]:
        beta *= 2
        change = opt["opt_tol"] * 2
    heaviside.forward(beta)
```

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

```
# Solve FEM
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.

```
        # Compute function values and sensitivities
        [C_value, V_value, U_value], sensitivities = sens_problem.evaluate()
        heaviside.backward(sensitivities)
        [dCdrho, dVdrho, dUdrho] = density_filter.backward(sensitivities)
        if opt["opt_compliance"]:
            g_vec = np.array([V_value-opt["vol_frac"]])
            dJdrho, dgdrho = dCdrho, np.vstack([dVdrho])
        else:
            g_vec = np.array([V_value-opt["vol_frac"], C_value-opt["compliance_bound"]])
            dJdrho, dgdrho = dUdrho, np.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.

```
        # Update the design variables
        rho_values = rho_field.vector.array.copy()
        if opt["opt_compliance"] and opt["use_oc"]:
            rho_new, change = optimality_criteria(
                rho_values, rho_min, rho_max, g_vec, dJdrho, dgdrho[0], opt["move"])
        else:
            rho_new, change, low, upp = mma_optimizer(
                num_consts, num_elems, opt_iter, rho_values, rho_min, rho_max,
                rho_old1, rho_old2, dJdrho, g_vec, dgdrho, low, upp, opt["move"])
            rho_old2 = rho_old1.copy()
            rho_old1 = rho_values.copy()
        rho_field.vector.array = rho_new.copy()

        # Output the histories
        opt_time = time.perf_counter() - opt_start_time
        if comm.rank == 0:
            print(f"opt_iter: {opt_iter}, opt_time: {opt_time:.3g} (s), "
                  f"beta: {beta}, C: {C_value:.3f}, V: {V_value:.3f}, "
                  f"U: {U_value:.3f}, 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

```
values = S_comm.gather(rho_phys_field)
if comm.rank == 0:
    plotter.plot(values)
save_xdmf(fem["mesh"], 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.

```
def form_fem(fem, opt):
    """Form an FEA problem."""
    # Function spaces and functions
    mesh = fem["mesh"]
    V = VectorFunctionSpace(mesh, ("CG", 1))
    S0 = FunctionSpace(mesh, ("DG", 0))
    S = FunctionSpace(mesh, ("CG", 1))
    u, v = ufl.TrialFunction(V), ufl.TestFunction(V)
    u_field = Function(V)  # Displacement field
    lambda_field = Function(V)  # Adjoint variable field
    rho_field = Function(S0)  # Density field
    rho_phys_field = Function(S)  # 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

```
    V = VectorFunctionSpace(mesh, ("CG", 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.[1] Users can readily access these distinct element types by executing a command such as

```
    V = VectorFunctionSpace(mesh, ("CR", 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

```
    # Material interpolation
    E0, nu = fem["young's modulus"], fem["poisson's ratio"]
    p, eps = opt["penalty"], opt["epsilon"]
    E = (eps + (1-eps)*rho_phys_field**p) * E0
    _lambda, mu = E*nu/(1+nu)/(1-2*nu), E/(2*(1+nu))  # Lame 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

```
    # Kinematics
    def epsilon(u):
        return ufl.sym(ufl.grad(u))

    def sigma(u):  # 3D or plane strain
        return 2*mu*epsilon(u) + _lambda*ufl.tr(epsilon(u))*ufl.Identity(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 $\overline{\mathbf{u}} = \mathbf{0}$ as previously mentioned), the controlled DOFs, and the function space associated with the displacement field.

```
    # Boundary conditions
    dim = mesh.topology.dim
    fdim = dim - 1
    disp_facets = locate_entities_boundary(mesh, fdim, fem["disp_bc"])
    bc = dirichletbc(Constant(mesh, np.full(dim, 0.0)), locate_dofs_topological(V, fdim, disp_facets), V)
```

[1] https://docs.fenicsproject.org/basix/main.

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

```
tractions, facets, markers = [], []
for marker, (traction, traction_bc) in enumerate(fem["traction_bcs"]):
    tractions.append(Constant(mesh, np.array(traction, dtype=float)))
    current_facets = locate_entities_boundary(mesh, fdim, traction_bc)
    facets.extend(current_facets)
    markers.extend([marker,]*len(current_facets))
facets = np.array(facets, dtype=np.int32)
markers = np.array(markers, dtype=np.int32)
_, unique_indices = np.unique(facets, return_index=True)
facets, markers = facets[unique_indices], markers[unique_indices]
sorted_indices = np.argsort(facets)
facet_tags = meshtags(mesh, fdim, facets[sorted_indices], markers[sorted_indices])

metadata = {"quadrature_degree": fem["quadrature_degree"]}
dx = ufl.Measure("dx", metadata=metadata)
ds = ufl.Measure("ds", domain=mesh, metadata=metadata, subdomain_data=facet_tags)
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

```
# Establish the equilibrium and adjoint equations
lhs = ufl.inner(sigma(u), epsilon(v))*dx
rhs = ufl.dot(b, v)*dx
for marker, t in enumerate(tractions):
    rhs += ufl.dot(t, v)*ds(marker)
if opt["opt_compliance"]:
    spring_vec = opt["l_vec"] = None
else:
    spring_vec, opt["l_vec"] = create_mechanism_vectors(
        V, opt["in_spring"], opt["out_spring"])
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.

```
# Define optimization-related variables
opt["f_int"] = ufl.inner(sigma(u_field), epsilon(v))*dx
opt["compliance"] = ufl.inner(sigma(u_field), epsilon(u_field))*dx
opt["volume"] = rho_phys_field*dx
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 $E^0 = 100$ N/mm^2 and $\nu^0 = 0.25$, respectively. We vanish the body force as $\overline{\mathbf{b}} = \mathbf{0}$ 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[2] 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 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

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

These configurations are specified within the input variable fem. Concerning topology optimization parameters, we set the upper bound for the material volume fraction as $\overline{V} = 0.5$ and establish a density filter radius of $R = 1.2$ 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

```
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

[2] Available at https://github.com/giampaolo/psutil.

Table 5 Sample examples provided by `FEniTop`

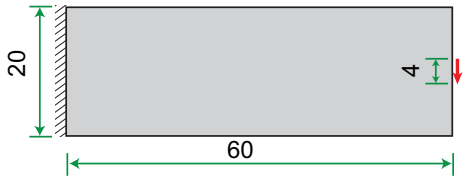
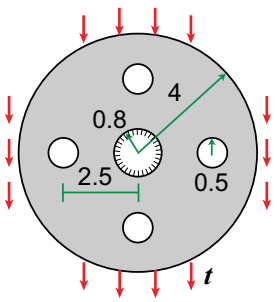
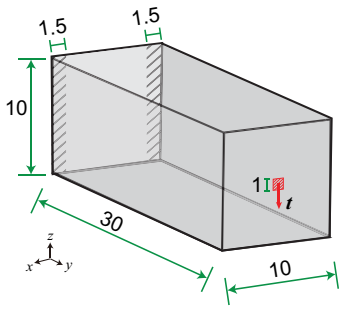
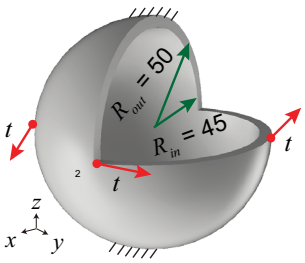
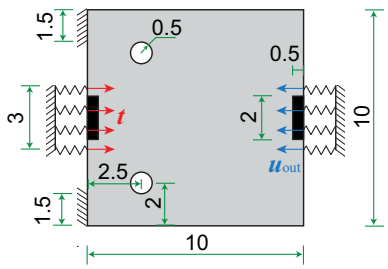
Design domains	Commands	Descriptions
	`mpirun -n 8 python3 scripts/beam_2d.py`	• $E = 100$ MPa, $\nu = 0.25$ • $n^{\max} = 400$, $\varepsilon^{\text{opt}} = 10^{-5}$ • $\overline{V} = 0.5$, $R = 1.2$ mm • $\beta^{\text{int}} = 50$, $\beta^{\max} = 128$ • $p = 3$, $\varepsilon = 10^{-6}$, $m = 0.05$
	`mpirun -n 8 python3 scripts/disk_2d.py`	• $E = 100$ MPa, $\nu = 0.25$ • $n^{\max} = 400$, $\varepsilon^{\text{opt}} = 10^{-5}$ • $\overline{V} = 0.5$, $R = 0.32$ mm • $\beta^{\text{int}} = 30$, $\beta^{\max} = 128$ • $p = 3$, $\varepsilon = 10^{-6}$, $m = 0.02$
	`mpirun -n 8 python3 scripts/beam_3d.py`	• $E = 100$ MPa, $\nu = 0.25$ • $n^{\max} = 400$, $\varepsilon^{\text{opt}} = 10^{-5}$ • $\overline{V} = 0.08$, $R = 0.6$ mm • $\beta^{\text{int}} = 50$, $\beta^{\max} = 128$ • $p = 3$, $\varepsilon = 10^{-6}$, $m = 0.02$
	`mpirun -n 8 python3 scripts/shell_3d.py`	• $E = 100$ MPa, $\nu = 0.25$ • $n^{\max} = 400$, $\varepsilon^{\text{opt}} = 10^{-5}$ • $\overline{V} = 0.12$, $R = 3.5$ mm • $\beta^{\text{int}} = 50$, $\beta^{\max} = 128$ • $p = 3$, $\varepsilon = 10^{-6}$, $m = 0.02$
	`mpirun -n 8 python3 scripts/mechanism_2d.py`	• $E = 100$ MPa, $\nu = 0.25$ • $n^{\max} = 500$, $\varepsilon^{\text{opt}} = 10^{-5}$ • $\overline{V} = 0.25$, $R = 0.5$ mm • $\beta^{\text{int}} = 50$, $\beta^{\max} = 128$ • $p = 3$, $\varepsilon = 10^{-6}$, $m = 0.05$

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

Configurations	Laptop	Desktop	Workstation	Cluster[a]
CPU brand	Intel(R) Core(TM) i7-10875 H	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

[a]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, $N_p \in \{1, 2, 4, 6, 8, 10, 12, 14\}$,[3] for three representative mesh sizes: 200×60 for the coarse mesh, 600×200 for the medium mesh, and 1500×500 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 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

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

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

[3] 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).

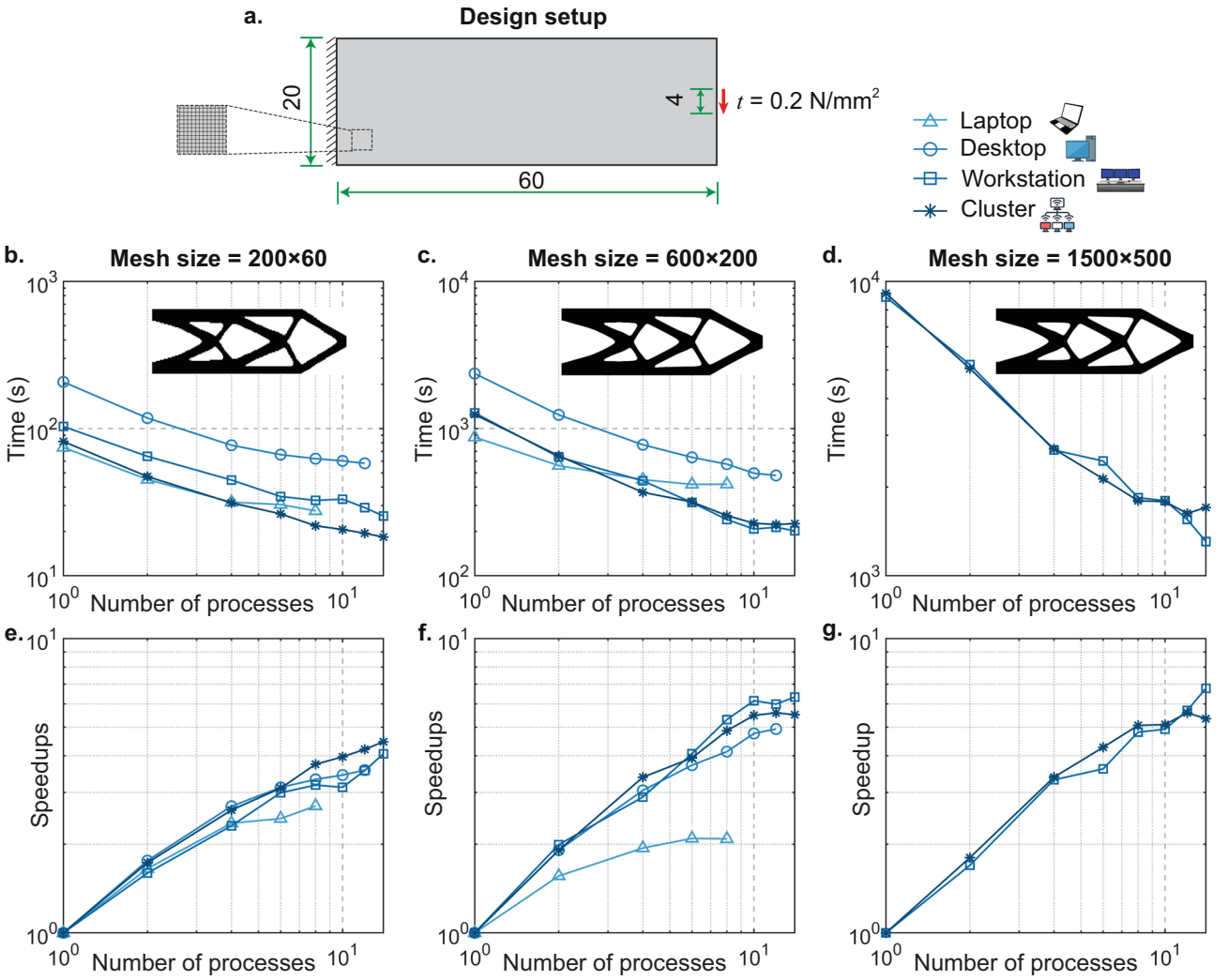


Fig. 3 Parallel computing efficiency across various mesh sizes and process counts on different computing platforms. **a** Presents the design setup including the design domain and boundary conditions. The variable t is the amplitude of the applied traction (i.e., $t = |\bar{\mathbf{t}}|$). The inset demonstrates the structured mesh used in FEA and topology optimization. **b**–**d** Present the total computational time for three distinct mesh sizes, 200 × 60, 600 × 200, and 1500 × 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

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

These commands employ the `dolfinx.io.XDMFFile` module, which supports parallel computing, to load a predefined unstructured mesh. Such unstructured meshes can be readily generated using open-source libraries like `gmsh`[4] (its Python interface is available in `FEniTop`) and commercial software like Abaqus.[5] Next, we can make use of the open-source package `meshio`[6] 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 $\overline{V} = 0.5$ and use a density filter radius of

[4] Available at https://gmsh.info.

[5] Available at https://www.3ds.com/products-services/simulia/products/abaqus.

[6] Available at https://github.com/nschloe/meshio.

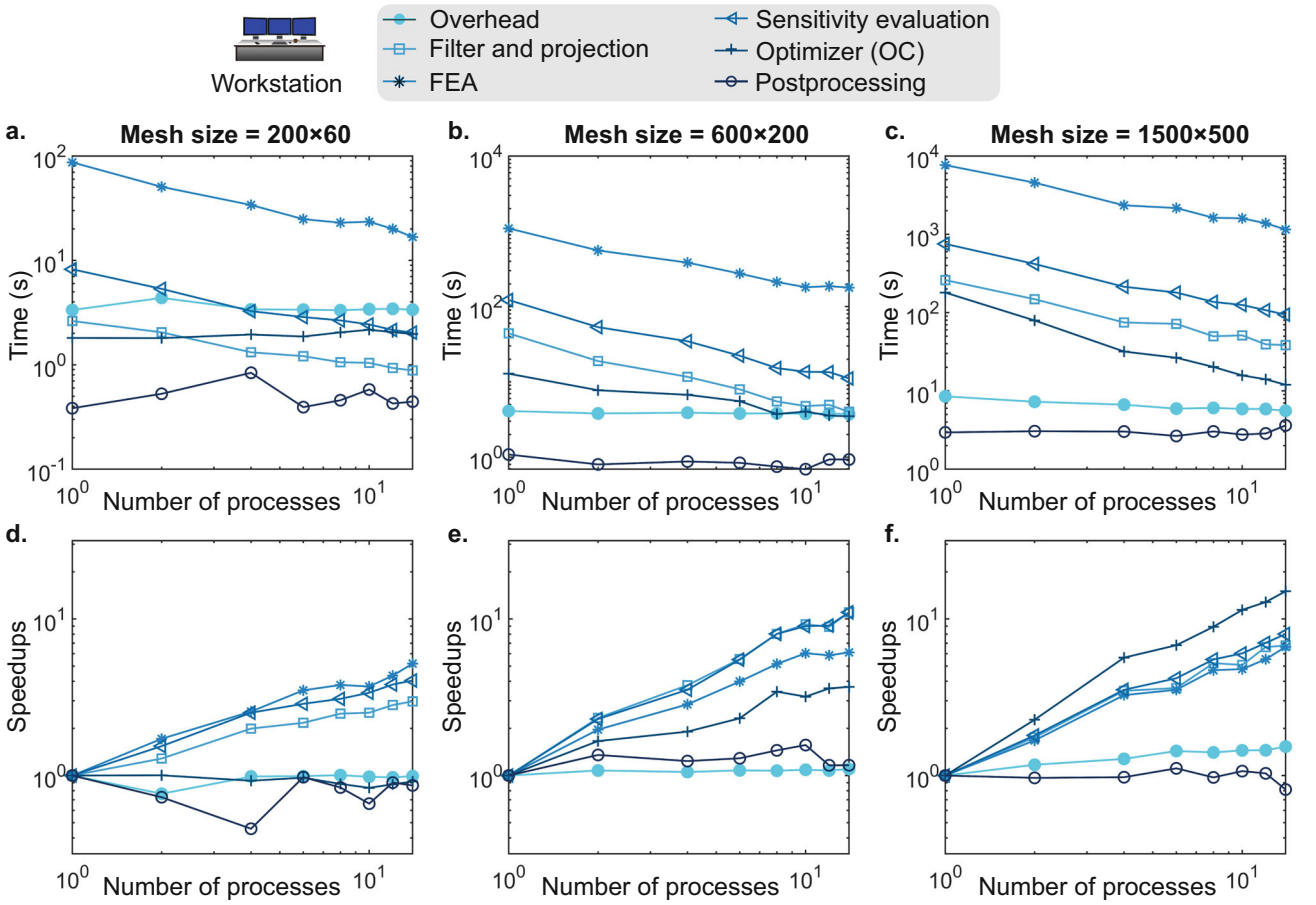


Fig. 4 Breakdown time and speedup of the workstation for various mesh sizes and process counts. **a**–**c** Present the breakdown time of the workstation for three distinct mesh sizes, 200 × 60, 600 × 200, and 1500 × 500, respectively. Each panel displays the correlation between the breakdown time and 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 $\beta^{\max} = 128$. 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 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: $25\times75\times25$, $50\times150\times50$, $75 \times 225 \times 75$, and $100 \times 300 \times 100$ finite elements, cor-

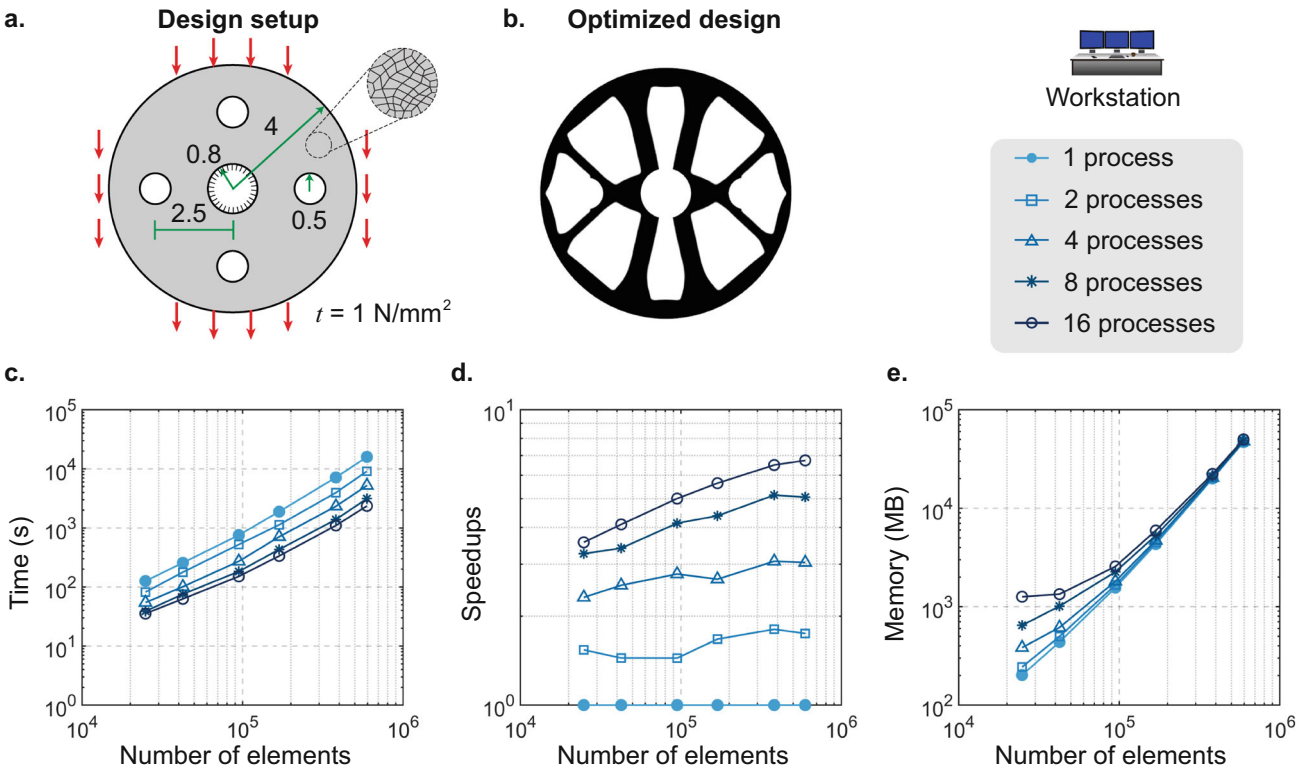


Fig. 5 Topology optimization and performance investigation for the 2D disk example. **a** Presents the design setup including the design domain and boundary conditions. The inset illustrates the unstructured mesh used for FEA and topology optimization. **b** Presents one optimized design consisting of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

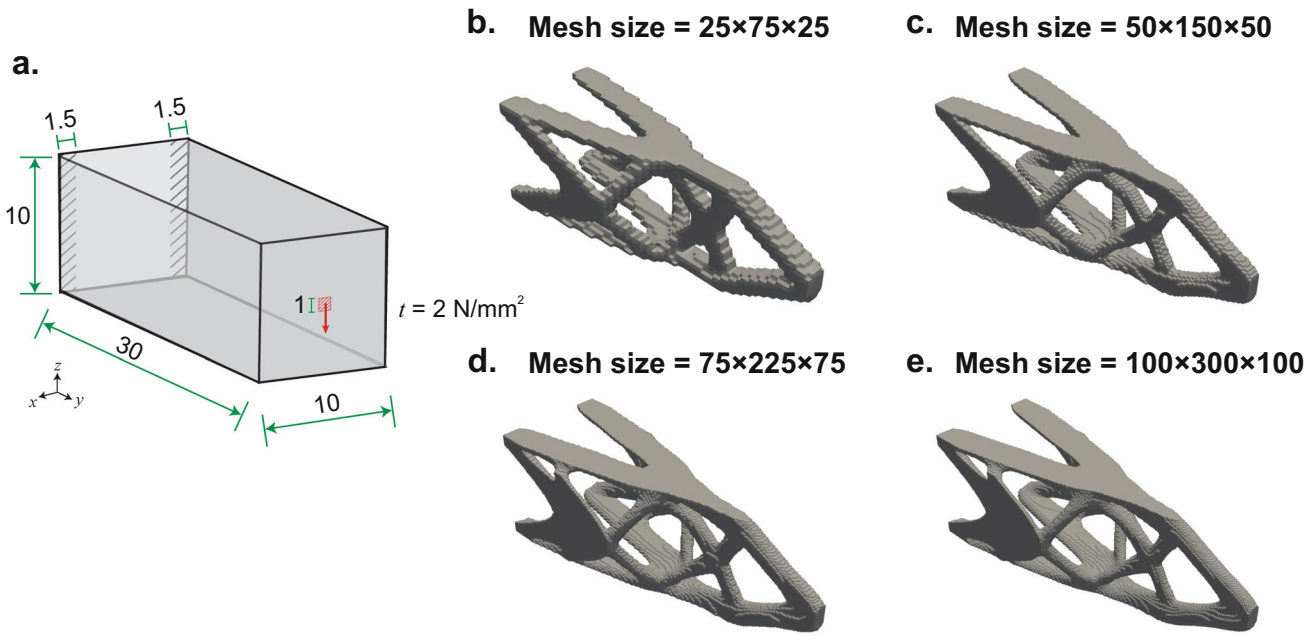


Fig. 6 Topology optimization for the 3D beam example. **a** Presents the design setup including the design domain and boundary conditions. **b**–**e** Present the optimized designs for various mesh sizes, $25 \times 75 \times 25$, $50 \times 150 \times 50$, $75 \times 225 \times 75$, and $100 \times 300 \times 100$, 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 $\overline{V} = 0.08$ 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[7] and preconditioners[8] 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

```
"petsc_options": {
    "ksp_type": "preonly",
    "pc_type": "lu",
    "pc_factor_mat_solver_type": "mumps",
}
```

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

[8] 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

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

and

```
"petsc_options": {
    "ksp_type": "gmres",
    "pc_type": "gamg",
}
```

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

```
mpirun -n 48 python3 scripts/beam_3d.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 Number of DOFs	$25 \times 75 \times 25$ 154,128	$50 \times 150 \times 50$ 1,178,253	$75 \times 225 \times 75$ 3,916,128	$100 \times 300 \times 100$ 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

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

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

```
def on_surface(x):
    return (np.isclose(x[0]**2+x[1]**2+x[2]**2, 50**2)
            & np.greater(x[2], -2) & np.less(x[2], 2))

traction_bcs = [
    [(0, 0.1, 0), lambda x: on_surface(x) & np.greater(x[0], 49.8)],
    [(-0.1, 0, 0), lambda x: on_surface(x) & np.greater(x[1], 49.8)],
    [(0, -0.1, 0), lambda x: on_surface(x) & np.less(x[0], -49.8)],
    [(0.1, 0, 0), lambda x: on_surface(x) & np.less(x[1], -49.8)],
]
```

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

```
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 Number of DOFs	25 × 75 × 25 154,128	50 × 150 × 50 1,178,253	75 × 225 × 75 3,916,128	100 × 300 × 100 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

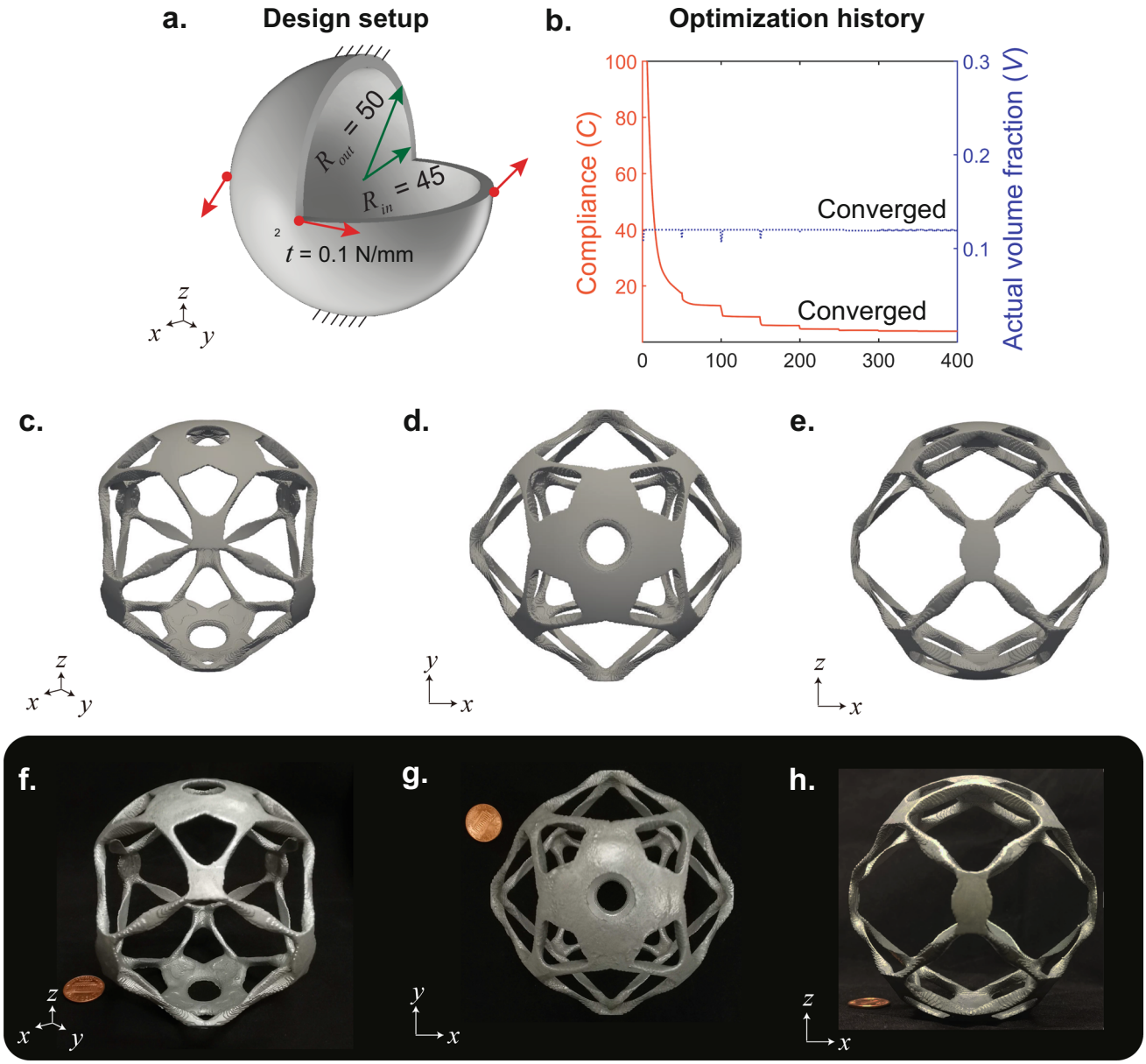


Fig. 7 Digital rendering and 3D printed physical models of the optimized design. **a** Presents the design setup including the design domain and boundary conditions. **b** Shows the optimization histories of structural compliance (C) and the actual material volume fraction (V). **c–e** Present the digital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

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

The above commands are mathematically equivalent to

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

with the origin defined as the center of the design domain's left edge. The symbols $\Omega^{\texttt{solid}} \subset \Omega$ and $\Omega^{\texttt{void}} \subset \Omega$ 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

```
def in_locator(x):
    return np.isclose(x[0], 0) & np.less(x[1], 1.5) & np.greater(x[1], -1.5)

def out_locator(x):
    return np.isclose(x[0], 10) & np.less(x[1], 1.5) & np.greater(x[1], -1.5)
```

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

```
{"in_spring": [in_locator, "x", 0.2],
 "out_spring": [out_locator, "x", 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

```
{"use_oc": False,
 "opt_compliance": False,
 "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, $\overline{C} = 0.5$ N $\cdot$ mm.

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

```
mpirun -n 8 python3 scripts/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 mm. Additionally, both the actual compliance and material volume fraction reach their respective bounds, $\overline{C} = 0.5$ N $\cdot$ mm and $\overline{V} = 0.25$, indicating the versatility and efficacy of `FEniTop` in addressing complex design problems.

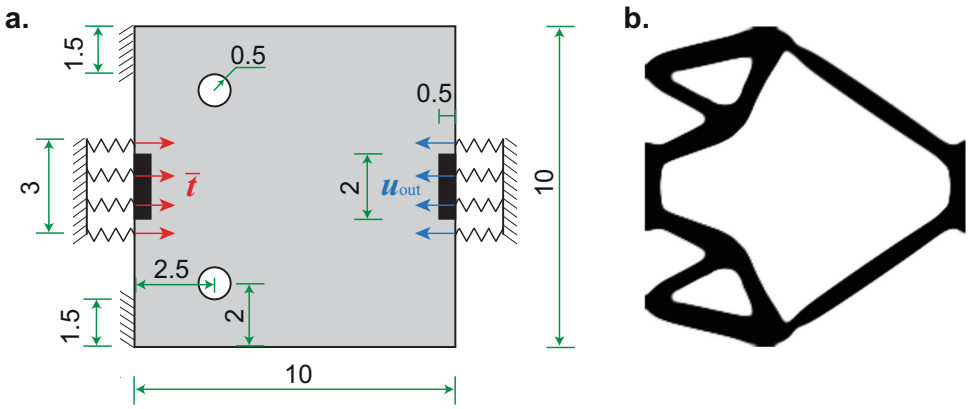


Fig. 8 Topology optimization for compliant mechanism design. **a** Presents the design setup including the design domain and boundary conditions. The design domain consists of two passive solid (black rectangles) and two passive void regions (circular holes), and it is also connected to distributed artificial springs along the input traction, $\bar{\mathbf{t}}$, 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 ($\overline{\boldsymbol{\rho}}$), we can employ the adjoint method (Bendsoe and Sigmund 2003) as follows. We begin by writing out the function's Lagrangian form ($\widehat{\mathcal{F}}$) as

$$\widehat{\mathcal{F}}(\overline{\boldsymbol{\rho}}, \mathbf{U}(\overline{\boldsymbol{\rho}})) = \mathcal{F}(\overline{\boldsymbol{\rho}}, \mathbf{U}(\overline{\boldsymbol{\rho}})) + \boldsymbol{\lambda} \cdot (\mathbf{F}^{\texttt{int}}(\overline{\boldsymbol{\rho}}, \mathbf{U}(\overline{\boldsymbol{\rho}})) - \mathbf{F}^{\texttt{ext}})$$

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

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

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

Rearranging terms renders

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

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

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

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

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

Appendix A.2 Numerical solution with the automatic differentiation

Based on (A.2), the evaluation of the sensitivity of a given function ($\mathrm{d}\mathcal{F}/\mathrm{d}\overline{\boldsymbol{\rho}}$) requires at most four terms (see also the illustration in Fig. 9): vectors $\partial \mathcal{F}/\partial \overline{\boldsymbol{\rho}}$ and $\partial \mathcal{F}/\partial \mathbf{U}$ as well as matrices $\partial \mathbf{F}^{\mathrm{int}}/\partial \overline{\boldsymbol{\rho}}$ and $\partial \mathbf{F}^{\mathrm{int}}/\partial \mathbf{U}$. 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

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

Subsequently, we define the above four pertinent terms at the weak form level, namely $\partial \mathcal{F}/\partial \overline{\rho}$, $\partial \mathcal{F}/\partial \mathbf{u}$, $\partial f^{\mathrm{int}}/\partial \overline{\rho}$, and $\partial f^{\mathrm{int}}/\partial \mathbf{u}$. 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

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

and the two matrices as

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

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

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

Having obtained the vectors $\partial \mathcal{F}/\partial \overline{\boldsymbol{\rho}}$ and $\partial \mathcal{F}/\partial \mathbf{U}$ as well as matrices $\partial \mathbf{F}^{\mathrm{int}}/\partial \overline{\boldsymbol{\rho}}$ and $\partial \mathbf{F}^{\mathrm{int}}/\partial \mathbf{U}$, we can proceed to construct the adjoint equation in (A.1) with $\partial \mathbf{F}^{\mathrm{int}}/\partial \mathbf{U}$ 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 $\partial \mathcal{F}/\partial \overline{\boldsymbol{\rho}}$, $\partial \mathbf{F}^{\mathrm{int}}/\partial \overline{\boldsymbol{\rho}}$, 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`

```
import numpy as np
from mpi4py import MPI
from dolfinx.mesh import
    create_rectangle, CellType

from fenitop.topopt import topopt


mesh = create_rectangle(MPI.COMM_WORLD,
    [[0, 0], [60, 20]],
                        [200, 60],
    CellType.quadrilateral)
if MPI.COMM_WORLD.rank == 0:
```

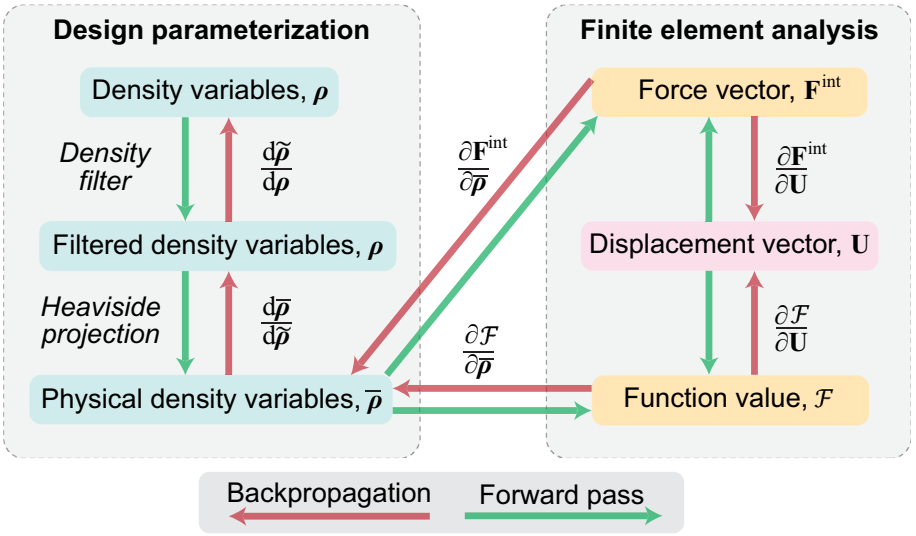


Fig. 9 The illustration of the sensitivity analysis involved in design space parameterization and FEA. Both design space parameterization and FEA consist of the forward pass and backpropagation stages. The forward pass evaluates the global internal force vector, $\mathbf{F}^{\text{int}}$, and the function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

```
    mesh_serial = create_rectangle(MPI.COMM_SELF, [[0, 0], [60, 20]], [200, 60], CellType.quadrilateral)
else:
    mesh_serial = None

fem = {  # FEM parameters
    "mesh": mesh,
    "mesh_serial": mesh_serial,
    "young's modulus": 100,
    "poisson's ratio": 0.25,
    "disp_bc": lambda x: np.isclose(x[0], 0),
    "traction_bcs": [[(0, -0.2), lambda x: (np.isclose(x[0], 60) & np.greater(x[1], 8) & np.less(x[1], 12))]],
    "body_force": (0, 0),
    "quadrature_degree": 2,
    "petsc_options": {
        "ksp_type": "cg",
        "pc_type": "gamg",
    },
}

opt = {  # Topology optimization parameters
    "max_iter": 400,
    "opt_tol": 1e-5,
    "vol_frac": 0.5,
    "solid_zone": lambda x: np.full(x.shape[1], False),
    "void_zone": lambda x: np.full(x.shape[1], False),
    "penalty": 3.0,
    "epsilon": 1e-6,
    "filter_radius": 1.2,
    "beta_interval": 50,
    "beta_max": 128,
    "use_oc": True,
    "move": 0.02,
    "opt_compliance": True,
}

if __name__ == "__main__":
    topopt(fem, opt)
```

Appendix B.2 `topopt` module for topology optimization

```
import time

import numpy as np
from mpi4py import MPI

from fenitop.fem import form_fem
from fenitop.parameterize import DensityFilter, Heaviside
from fenitop.sensitivity import Sensitivity
from fenitop.optimize import optimality_criteria, mma_optimizer
from fenitop.utility import Communicator, Plotter, save_xdmf


def topopt(fem, opt):
    """Main function for topology optimization."""

    # Initialization
    comm = MPI.COMM_WORLD
    linear_problem, u_field, lambda_field, rho_field, rho_phys_field = form_fem(fem, opt)
    density_filter = DensityFilter(comm, rho_field, rho_phys_field, opt["filter_radius"], fem["petsc_options"])
    heaviside = Heaviside(rho_phys_field)
    sens_problem = Sensitivity(comm, opt, linear_problem, u_field, lambda_field, rho_phys_field)
```

```
    S_comm = Communicator(rho_phys_field.function_space, fem["mesh_serial"])
    if comm.rank == 0:
        plotter = Plotter(fem["mesh_serial"])
    num_consts = 1 if opt["opt_compliance"] else 2
    num_elems = rho_field.vector.array.size
    if not opt["use_oc"]:
        rho_old1, rho_old2, = np.zeros(num_elems), np.zeros(num_elems)
        low, upp = None, None

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

    # Start topology optimization
    opt_iter, beta, change = 0, 1, 2*opt["opt_tol"]
    while opt_iter < opt["max_iter"] and change > opt["opt_tol"]:
        opt_start_time = time.perf_counter()
        opt_iter += 1

        # Density filter and Heaviside projection
        density_filter.forward()
        if opt_iter % opt["beta_interval"] == 0 and beta < opt["beta_max"]:
            beta *= 2
            change = opt["opt_tol"] * 2
        heaviside.forward(beta)

        # Solve FEM
        linear_problem.solve_fem()

        # Compute function values and sensitivities
        [C_value, V_value, U_value], sensitivities = sens_problem.evaluate()
        heaviside.backward(sensitivities)
        [dCdrho, dVdrho, dUdrho] = density_filter.backward(sensitivities)
        if opt["opt_compliance"]:
            g_vec = np.array([V_value-opt["vol_frac"]])
            dJdrho, dgdrho = dCdrho, np.vstack([dVdrho])
        else:
            g_vec = np.array([V_value-opt["vol_frac"], C_value-opt["compliance_bound"]])
            dJdrho, dgdrho = dUdrho, np.vstack([dVdrho, dCdrho])

        # Update the design variables
        rho_values = rho_field.vector.array.copy()
        if opt["opt_compliance"] and opt["use_oc"]:
            rho_new, change = optimality_criteria(
                rho_values, rho_min, rho_max, g_vec, dJdrho, dgdrho[0], opt["move"])
        else:
            rho_new, change, low, upp = mma_optimizer(
                num_consts, num_elems, opt_iter, rho_values, rho_min, rho_max,
                rho_old1, rho_old2, dJdrho, g_vec, dgdrho, low, upp, opt["move"])
            rho_old2 = rho_old1.copy()
            rho_old1 = rho_values.copy()
        rho_field.vector.array = rho_new.copy()

        # Output the histories
        opt_time = time.perf_counter() - opt_start_time
        if comm.rank == 0:
            print(f"opt_iter: {opt_iter}, opt_time: {opt_time:.3g} (s), "
                  f"beta: {beta}, C: {C_value:.3f}, V: {V_value:.3f}, "
                  f"U: {U_value:.3f}, change: {change:.3f}", flush=True)

    values = S_comm.gather(rho_phys_field)
    if comm.rank == 0:
        plotter.plot(values)
    save_xdmf(fem["mesh"], rho_phys_field)
```

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

```
import numpy as np
import ufl
from dolfinx import la
from dolfinx.fem import Function, form
from dolfinx.fem.petsc import create_matrix, assemble_matrix
from petsc4py import PETSc


class DensityFilter():
```

```
    def __init__(self, comm, rho, rho_tilde, R=1.0, petsc_options={}):
        """Construct a PDE filter."""
        # Initialization
        S0, S = rho.function_space, rho_tilde.function_space
        u0, u = ufl.TrialFunction(S0), ufl.TrialFunction(S)
        v, self.af = ufl.TestFunction(S), Function(S)

        self.rho, self.rho_tilde = rho, rho_tilde
        self.rho_tilde_wrap = la.create_petsc_vector_wrap(self.rho_tilde.x)
        self.af_wrap = la.create_petsc_vector_wrap(self.af.x)
        self.vec_s0, self.vec_s = rho.vector.copy(), rho_tilde.vector.copy()

        # Construct Kf and T matrices based on the Helmholtz PDE
        dx = ufl.Measure("dx", metadata={"quadrature_degree": 2})
        Kf_expr = (R**2*ufl.dot(ufl.grad(u), ufl.grad(v)) + u*v)*dx
        T_expr = u0*v*dx
        Kf_form, T_form = form(Kf_expr), form(T_expr)
        Kf_mat, self.T_mat = create_matrix(Kf_form), create_matrix(T_form)

        # Construct a filtering solver
        self.solver = PETSc.KSP().create(comm)
        self.solver.setOperators(Kf_mat)
        prefix = f"filter_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()
        Kf_mat.setOptionsPrefix(prefix)
        Kf_mat.setFromOptions()

        # Assemble Kf and T matrices
        assemble_matrix(Kf_mat, Kf_form)
        Kf_mat.assemble()
        assemble_matrix(self.T_mat, T_form)
        self.T_mat.assemble()
        self.T_mat_transpose = self.T_mat.copy()
        self.T_mat_transpose.transpose()

    def forward(self):
        """Compute the filtered variables."""
        self.T_mat.mult(self.rho.vector, self.vec_s)
        self.solver.solve(self.vec_s, self.rho_tilde_wrap)
        self.rho_tilde.x.scatter_forward()
        return self.rho_tilde

    def backward(self, sf_vectors):
        """Recover the sensitivities."""
        values = []
        for sf in sf_vectors:
            if sf is not None:
                self.solver.solve(sf, self.af_wrap)
                self.af.x.scatter_forward()
                self.T_mat_transpose.mult(self.af.vector, self.vec_s0)
                values.append(self.vec_s0.array.copy())
            else:
                values.append(None)
        return values


class Heaviside():
    def __init__(self, rho_phys):
        self.rho_phys = rho_phys

    def forward(self, beta, eta=0.5):
        denominator = np.tanh(beta*eta) + np.tanh(beta*(1-eta))
        self.drho = beta*(1-np.tanh(beta*(self.rho_phys.vector-eta))**2) / denominator
        self.rho_phys.vector.array = (
            np.tanh(beta*eta)+np.tanh(beta*(self.rho_phys.vector-eta))) / denominator
        self.rho_phys.x.scatter_forward()

    def backward(self, vectors):
        for vector in vectors:
            if vector is not None:
                vector.array *= self.drho
```

Appendix B.4 `fem` module for FEA

```
import numpy as np
import ufl
from dolfinx.mesh import locate_entities_boundary, meshtags
from dolfinx.fem import (VectorFunctionSpace, FunctionSpace, Function, Constant,
                         dirichletbc, locate_dofs_topological)

```

```python
from fenitop.utility import create_mechanism_vectors
from fenitop.utility import LinearProblem


def form_fem(fem, opt):
    """Form an FEA problem."""
    # Function spaces and functions
    mesh = fem["mesh"]
    V = VectorFunctionSpace(mesh, ("CG", 1))
    S0 = FunctionSpace(mesh, ("DG", 0))
    S = FunctionSpace(mesh, ("CG", 1))
    u, v = ufl.TrialFunction(V), ufl.TestFunction(V)
    u_field = Function(V)  # Displacement field
    lambda_field = Function(V)  # Adjoint variable field
    rho_field = Function(S0)  # Density field
    rho_phys_field = Function(S)  # Physical density field

    # Material interpolation
    E0, nu = fem["young's modulus"], fem["poisson's ratio"]
    p, eps = opt["penalty"], opt["epsilon"]
    E = (eps + (1-eps)*rho_phys_field**p) * E0
    _lambda, mu = E*nu/(1+nu)/(1-2*nu), E/(2*(1+nu))  # Lame constants

    # Kinematics
    def epsilon(u):
        return ufl.sym(ufl.grad(u))

    def sigma(u):  # 3D or plane strain
        return 2*mu*epsilon(u) + _lambda*ufl.tr(epsilon(u))*ufl.Identity(len(u))

    # Boundary conditions
    dim = mesh.topology.dim
    fdim = dim - 1
    disp_facets = locate_entities_boundary(mesh, fdim, fem["disp_bc"])
    bc = dirichletbc(Constant(mesh, np.full(dim, 0.0)),

    locate_dofs_topological(V, fdim, disp_facets), V)

    tractions, facets, markers = [], [], []
    for marker, (traction, traction_bc) in enumerate(fem["traction_bcs"]):
        tractions.append(Constant(mesh, np.array(traction, dtype=float)))
        current_facets = locate_entities_boundary(mesh, fdim, traction_bc)
        facets.extend(current_facets)
        markers.extend([marker,]*len(current_facets))
    facets = np.array(facets, dtype=np.int32)
    markers = np.array(markers, dtype=np.int32)
    _, unique_indices = np.unique(facets, return_index=True)
    facets, markers = facets[unique_indices], markers[unique_indices]
    sorted_indices = np.argsort(facets)
    facet_tags = meshtags(mesh, fdim, facets[sorted_indices], markers[sorted_indices])

    metadata = {"quadrature_degree": fem["quadrature_degree"]}
    dx = ufl.Measure("dx", metadata=metadata)
    ds = ufl.Measure("ds", domain=mesh, metadata=metadata, subdomain_data=facet_tags)
    b = Constant(mesh, np.array(fem["body_force"], dtype=float))

    # Establish the equilibrium and adjoint equations
    lhs = ufl.inner(sigma(u), epsilon(v))*dx
    rhs = ufl.dot(b, v)*dx
    for marker, t in enumerate(tractions):
        rhs += ufl.dot(t, v)*ds(marker)
    if opt["opt_compliance"]:
        spring_vec = opt["l_vec"] = None
    else:
        spring_vec, opt["l_vec"] = create_mechanism_vectors(
            V, opt["in_spring"], opt["out_spring"])
    linear_problem = LinearProblem(u_field, lambda_field, lhs, rhs, opt["l_vec"],

    spring_vec, [bc], fem["petsc_options"])

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

    return linear_problem, u_field, lambda_field, rho_field, rho_phys_field
```

Appendix B.5 `sensitivity` module for sensitivity analysis

```python
import ufl
from mpi4py import MPI
from dolfinx.fem import form, assemble_scalar
from dolfinx.fem.petsc import create_vector, create_matrix, assemble_vector, assemble_matrix
from petsc4py import PETSc


class Sensitivity():
    def __init__(self, comm, opt, problem, u_field, lambda_field, rho_phys):
        # Compliance
        if opt["opt_compliance"]:
            self.C_form = form(opt["compliance"])
        self.dCdrho_form = form(-ufl.derivative(opt["compliance"], rho_phys))
        self.dCdrho_vec = create_vector(self.dCdrho_form)

        # Volume
        self.comm = comm
        self.total_volume = comm.allreduce(
            assemble_scalar(form(opt["total_volume"])), op=MPI.SUM)
        self.V_form = form(opt["volume"])
        dVdrho_form = form(ufl.derivative(opt["volume"], rho_phys))
        self.dVdrho_vec = create_vector(dVdrho_form)
        assemble_vector(self.dVdrho_vec, dVdrho_form)
        self.dVdrho_vec.ghostUpdate(addv=PETSc.InsertMode.ADD, mode=PETSc.ScatterMode.REVERSE)
        self.dVdrho_vec /= self.total_volume

        # Displacement
        self.opt_compliance = opt["opt_compliance"]
        if not self.opt_compliance:
            self.dfdrho_form = form(ufl.adjoint(ufl.derivative(opt["f_int"], rho_phys)))
            self.dfdrho_mat = create_matrix(self.dfdrho_form)
            self.problem, self.l_vec = problem, opt["l_vec"]
            self.u_field, self.lambda_field = u_field, lambda_field
            self.dUdrho_vec = rho_phys.vector.copy()
            self.prod_vec = u_field.vector.copy()

    def __del__(self):
        if not self.opt_compliance:
            self.prod_vec.destroy()

    def evaluate(self):
        # Compliance
        if self.opt_compliance:
            C_value = self.comm.allreduce(assemble_scalar(self.C_form), op=MPI.SUM)
        else:
            self.problem.lhs_mat.mult(self.u_field.vector, self.prod_vec)
            C_value = self.u_field.vector.dot(self.prod_vec)
        with self.dCdrho_vec.localForm() as loc:
            loc.set(0)
        assemble_vector(self.dCdrho_vec, self.dCdrho_form)
        self.dCdrho_vec.ghostUpdate(addv=PETSc.InsertMode.ADD, mode=PETSc.ScatterMode.REVERSE)
        actual_volume = self.comm.allreduce(assemble_scalar(self.V_form), op=MPI.SUM)
        V_value = actual_volume / self.total_volume
        self.dVdrho_vec_copy = self.dVdrho_vec.copy()

        # Displacement
        if not self.opt_compliance:
            U_value = self.u_field.vector.dot(self.l_vec)
            self.problem.solve_adjoint()
            self.dfdrho_mat.zeroEntries()
            assemble_matrix(self.dfdrho_mat, self.dfdrho_form)
            self.dfdrho_mat.assemble()
            self.dfdrho_mat.mult(self.lambda_field.vector, self.dUdrho_vec)
        else:
            U_value, self.dUdrho_vec = 0, None

        func_values = [C_value, V_value, U_value]
        sensitivities = [self.dCdrho_vec, self.dVdrho_vec_copy, self.dUdrho_vec]
        return func_values, sensitivities
```

Appendix B.6 `optimize` module for solution update

```python
import numpy as np
from mpi4py import MPI
from scipy import sparse as sparse
from scipy.linalg import solve


```

```python
def optimality_criteria(rho, rho_min, rho_max, V, dCdrho, dVdrho, move=0.05):
    """Solution update scheme with optimality criteria (OC)."""
    lb, ub = 0.0, 1e6
    comm = MPI.COMM_WORLD
    while ub-lb > 1e-4:
        mid = (lb+ub) / 2.0
        rho_new = np.maximum.reduce([np.minimum.reduce(
            [rho*(-dCdrho/(dVdrho+1e-12)/mid)**0.5, rho+move, rho_max]), rho-move, rho_min])
        dV = comm.allreduce(dVdrho@(rho_new-rho), op=MPI.SUM)
        if V + dV > 0:
            lb = mid
        else:
            ub = mid
    change = comm.allreduce(np.max(np.abs(rho_new-rho), initial=0), op=MPI.MAX)
    return rho_new, change


def mma_optimizer(m, n, opt_iter, xval, xmin, xmax, xold1, xold2, df0dx, fval,
                  dfdx, low, upp, a0=1, a=None, c=None, d=None, move=0.05,
                  asyinit=0.5, asydecr=0.7, asyincr=1.2,
                  low_bnd=0.002, up_bnd=1.0, albefa=0.1, feps=1e-6):
    """Solution update scheme with the method of moving asymptotes (MMA).
    The algorithm is available in https://doi.org/10.1002/nme.1620240207.

    Minimize:
        f_0(x) + a_0*z + sum(c_i*y_i + 0.5*d_i*(y_i)^2)
    Subjected to:
        f_i(x) - a_i*z - y_i <= 0,    i = 1, 2, ..., m
        xmin_j <= x_j <= xmax_j,      j = 1, 2, ..., n
        y_i >= 0,                     i = 1, 2, ..., m
        z >= 0

    Args:
        m: The number of general constraints.
        n: The number of the variables, x_j.
        opt_iter: Iteration counter.
        xval: Current values of the variables, x_j.
        xmin, xmax: Lower and upper bounds of the variables, x_j.
        xold1, xold2: The values of x_j at one and two iterations ago.
        df0dx: The derivatives of the objective function, f_0(x),
            with respect to the variables, x_j, calculated at xval.
        fval: The values of the constraint functions, f_i(x), calculated at xval.
        dfdx: An (m, n) array with the derivatives of the constraint functions,
            f_i(x), with respect to the variables, x_j, calculated at xval.
        low, upp: Lower and upper asymptotes from the previous iteration.
        a0, a, c, d: Coefficients in the objective function.
        move: Move limit of the variables, x_j.
        asyinit: Initial rate of the asymptotes.
        asydecr: Decreasing rate of the asymptotes when the variables are oscillating.
        asyincr: Increasing rate of the asymptotes when the variables are monotonically updated.
        low_bnd, up_bnd: Lower and upper bounds for determining the asymptotes.
        albefa: A parameter for determining the bounds of the variables.
        feps: A parameter for approximating the objective function.

    Returns:
        x_new: Optimal values of the variables, x_j, in the current subproblem.
        change: Maximum change of the variables, x_j.
        low, upp: Lower and upper asymptotes calculated and used in the current subproblem.
    """

    # Initialize the coefficients of the objective function
    if a is None:
        a = np.zeros(m)
    if c is None:
        c = np.full(m, 1000)
    if d is None:
        d = np.zeros(m)
    comm = MPI.COMM_WORLD
    n_global = comm.allreduce(n, op=MPI.SUM)
    epsimin = np.sqrt(m+n_global)*1e-9

    # Evaluate the asymptotes (low and upp)
    xrange = xmax - xmin
    if opt_iter < 2.5:
        low = xval - asyinit*xrange
        upp = xval + asyinit*xrange
    else:
```

```
        idx = (xval-xold1) * (xold1-xold2)  # Check the oscillation
        gamma = np.ones(n)
        gamma[idx > 0] = asyincr  # Relax the asymptotes if monotonic
        gamma[idx < 0] = asydecr  # Tighten the asymptotes if oscillating
        low = xval - gamma*(xold1-low)
        upp = xval + gamma*(upp-xold1)
        lowmin = xval - up_bnd*xrange
        lowmax = xval - low_bnd*xrange
        uppmin = xval + low_bnd*xrange
        uppmax = xval + up_bnd*xrange
        low = np.minimum(np.maximum(low, lowmin), lowmax)
        upp = np.maximum(np.minimum(upp, uppmax), uppmin)

    # Evaluate the bounds of the variables (alpha and beta)
    alpha = np.maximum.reduce([xmin, low+albefa*(xval-low), xval-move*xrange])
    beta = np.minimum.reduce([xmax, upp-albefa*(upp-xval), xval+move*xrange])

    # Evaluate the coefficients of the approximating objective fuction (p0 and q0)
    idx = df0dx > 0
    p0, q0 = np.zeros(n), np.zeros(n)
    p0[idx], p0[~idx] = 1.001*df0dx[idx], -0.001*df0dx[~idx]
    q0[idx], q0[~idx] = 0.001*df0dx[idx], -1.001*df0dx[~idx]
    ul1 = upp - low
    p0, q0 = (p0+feps/ul1)*(upp-xval)**2, (q0+feps/ul1)*(xval-low)**2

    # Evaluate the coefficients of the approximating constraint fuctions (P_mat and Q_mat)
    P_mat, Q_mat = np.zeros((m, n)), np.zeros((m, n))
    dfdx = dfdx.reshape(m, n)
    idx = dfdx > 0
    P_mat[idx], Q_mat[~idx] = dfdx[idx], -dfdx[~idx]
    P_mat, Q_mat = P_mat*(upp-xval)**2, Q_mat*(xval-low)**2

    # Evaluate the b vector for constraint functions
    b = P_mat@(1.0/(upp-xval)) + Q_mat@(1.0/(xval-low))
    b = comm.allreduce(b, op=MPI.SUM) - fval

    # Solve the subproblem with a primal-dual interior-point approach
    x_new = solve_subproblem(m, epsimin, low, upp, alpha, beta, p0, q0,
                             P_mat, Q_mat, a0, a, b, c, d)
    change = comm.allreduce(np.max(np.abs(x_new-xval), initial=0), op=MPI.MAX)
    return x_new, change, low, upp


def solve_subproblem(m, epsimin, low, upp, alpha, beta, p0, q0, P_mat, Q_mat,
                     a0, a, b, c, d):
    """Solve the MMA subproblem with a primal-dual interior-point approach.

    Minimize:
        sum[p0j/(uppj-xj) + q0j/(xj-lowj)] + a0*z + sum(ci*yi + 0.5*di*yi^2)
    Subjected to:
        sum[pij/(uppj-xj) + qij/(xj-lowj)] - ai*z - yi <= bi,    i = 1, 2, ..., m
        alphaj <= xj <= betaj,    j = 1, 2, ..., n
        yi >= 0,                  i = 1, 2, ..., m
        z >= 0
    """
    # Set the initial guess for variables and Lagrange multipliers
    comm = MPI.COMM_WORLD
    eps = 1.0
    x = 0.5*(alpha+beta)
    xi = np.maximum(1.0/(x-alpha), 1.0)
    eta = np.maximum(1.0/(beta-x), 1.0)
    _lambda = np.ones(m)
    if comm.rank == 0:
        y, z, s = np.ones(m), 1.0, np.ones(m)
        mu = np.maximum(0.5*c, 1.0)
        zeta = 1.0

    while eps > epsimin:  # Primal-dual interior point method
        p_lambda = p0 + _lambda@P_mat
        q_lambda = q0 + _lambda@Q_mat
        g_vec = P_mat@(1.0/(upp-x)) + Q_mat@(1.0/(x-low))
        g_vec = comm.allreduce(g_vec, op=MPI.SUM)
        dpsi_dx = p_lambda/(upp-x)**2 - q_lambda/(x-low)**2

        # Evaluate the residual norm
        res_x = comm.gather(dpsi_dx-xi+eta, root=0)
        res_xi = comm.gather(xi*(x-alpha)-eps, root=0)
        res_eta = comm.gather(eta*(beta-x)-eps, root=0)
        if comm.rank == 0:
            res_x, res_xi, res_eta = np.hstack(res_x), np.hstack(res_xi), np.hstack(res_eta)
            res_y = c + d*y - mu - _lambda
```

```
        res_z = a0 - zeta - a@_lambda
        res_lambda = g_vec - a*z - y + s - b
        res_mu = mu*y - eps
        res_zeta = zeta*z - eps
        res_s = _lambda*s - eps
        res = np.hstack([res_x, res_y, res_z, res_lambda, res_xi, res_eta,
                         res_mu, res_zeta, res_s])
        res_norm = np.linalg.norm(res, ord=2)
        res_max = np.linalg.norm(res, ord=np.inf)
    else:
        res_norm, res_max = None, None
    res_norm = comm.bcast(res_norm, root=0)
    res_max = comm.bcast(res_max, root=0)

    newton_iter = 0
    while res_max > 0.9*eps and newton_iter < 100:  # Newton's method for the search direction
        newton_iter += 1
        if newton_iter == 100 and comm.rank == 0:
            print("Maximum inner iterations reached in the MMA optimizer", flush=True)

        # Evaluate the left-hand side
        p_lambda = p0 + _lambda@P_mat
        q_lambda = q0 + _lambda@Q_mat
        G_mat = P_mat/(upp-x)**2 - Q_mat/(x-low)**2
        G_mat_global = comm.gather(G_mat, root=0)
        diag_x = 2*(p_lambda/(upp-x)**3+q_lambda/(x-low)**3) + xi/(x-alpha) + eta/(beta-x)
        diag_x_global = comm.gather(diag_x, root=0)
        if comm.rank == 0:
            G_mat_global = np.hstack(G_mat_global)
            diag_x_global = np.hstack(diag_x_global)
            diag_y = d + mu/y
            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
            lhs = np.vstack([np.hstack([A_mat, a.reshape(-1, 1)]),
                             np.hstack([a, -zeta/z])])

        # Evaluate the right-hand side
        dpsi_dx = p_lambda/(upp-x)**2 - q_lambda/(x-low)**2
        delta_x = dpsi_dx - eps/(x-alpha) + eps/(beta-x)
        delta_x_global = comm.gather(delta_x, root=0)
        g_vec = P_mat@(1.0/(upp-x)) + Q_mat@(1.0/(x-low))
        g_vec = comm.allreduce(g_vec, op=MPI.SUM)
        if comm.rank == 0:
            delta_x_global = np.hstack(delta_x_global)
            delta_y = c + d*y - _lambda - eps/y
            delta_z = a0 - a@_lambda - eps/z
            delta_lambda = g_vec - a*z - y - b + eps/_lambda
            b_lambda = delta_lambda + delta_y/diag_y - G_mat_global@(delta_x_global/diag_x_global)
            rhs = np.hstack([b_lambda, delta_z])

        # Solve the search direction
        if comm.rank == 0:
            solution = solve(lhs, rhs)
            incr_lambda, incr_z = solution[:m], solution[m]
            incr_y = -delta_y/diag_y + incr_lambda/diag_y
            incr_mu = -mu + eps/y - (mu*incr_y)/y
            incr_zeta = -zeta + eps/z - zeta*incr_z/z
            incr_s = -s + eps/_lambda - (s*incr_lambda)/_lambda
        else:
            incr_lambda = None
        incr_lambda = comm.bcast(incr_lambda, root=0)
        incr_x = -delta_x/diag_x - (incr_lambda@G_mat)/diag_x
        incr_xi = -xi + eps/(x-alpha) - (xi*incr_x)/(x-alpha)
        incr_eta = -eta + eps/(beta-x) + (eta*incr_x)/(beta-x)
        incr_xi_global = comm.gather(incr_xi, root=0)
        incr_eta_global = comm.gather(incr_eta, root=0)
        xi_global = comm.gather(xi, root=0)
        eta_global = comm.gather(eta, root=0)
        if comm.rank == 0:
            xi_global = np.hstack(xi_global)
            eta_global = np.hstack(eta_global)
```

```
            values = np.hstack([y, z, _lambda, xi_global, eta_global, mu, zeta, s])
            incr_xi_global = np.hstack(incr_xi_global)
            incr_eta_global = np.hstack(incr_eta_global)
            incr = np.hstack([incr_y, incr_z, incr_lambda, incr_xi_global,
incr_eta_global, incr_mu, incr_zeta, incr_s])

        # Evaluate the upper bound of the step size
        step_inv_alpha = np.max(-1.01*incr_x/(x-alpha), initial=0)  # Ensure x-alpha >= 0.01/1.01*(x_old-alpha)
        step_inv_beta = np.max(1.01*incr_x/(beta-x), initial=0)  # Ensure beta-x >= 0.01/1.01*(beta-x_old)
        step_inv_alpha = comm.allreduce(step_inv_alpha, op=MPI.MAX)
        step_inv_beta = comm.allreduce(step_inv_beta, op=MPI.MAX)
        if comm.rank == 0:
            step_inv = max(-1.01*incr/values)  # Ensure y >= 0.01/1.01*y_old, etc.
            step_inv = max(step_inv_alpha, step_inv_beta, step_inv, 1.0)  # Ensure step <= 1
            step = 1.0 / step_inv
        else:
            step = None
        step = comm.bcast(step, root=0)

        x_old, lambda_old = x.copy(), _lambda.copy()
        xi_old, eta_old = xi.copy(), eta.copy()
        if comm.rank == 0:
            y_old, z_old = y.copy(), z
            mu_old, zeta_old, s_old = mu.copy(), zeta, s.copy()

        search_iter = 0
        res_norm_new = 2*res_norm
        while res_norm_new > res_norm and search_iter < 50:  # Line search for the step size
            search_iter += 1

            # Update the solutions
            x = x_old + step*incr_x
            _lambda = lambda_old + step*incr_lambda
            xi = xi_old + step*incr_xi
            eta = eta_old + step*incr_eta
            if comm.rank == 0:
                y = y_old + step*incr_y
                z = z_old + step*incr_z
                mu = mu_old + step*incr_mu
                zeta = zeta_old + step*incr_zeta
                s = s_old + step*incr_s

            # Evaluate the residual norm
            p_lambda = p0 + _lambda@P_mat
            q_lambda = q0 + _lambda@Q_mat
            g_vec = P_mat@(1.0/(upp-x)) + Q_mat@(1.0/(x-low))
            g_vec = comm.allreduce(g_vec, op=MPI.SUM)
            dpsi_dx = p_lambda/(upp-x)**2 - q_lambda/(x-low)**2

            res_x = comm.gather(dpsi_dx-xi+eta, root=0)
            res_xi = comm.gather(xi*(x-alpha)-eps, root=0)
            res_eta = comm.gather(eta*(beta-x)-eps, root=0)
            if comm.rank == 0:
                res_x, res_xi, res_eta = np.hstack(res_x), np.hstack(res_xi), np.hstack(res_eta)
                res_y = c + d*y - mu - _lambda
                res_z = a0 - zeta - a@_lambda
                res_lambda = g_vec - a*z - y + s - b
                res_mu = mu*y - eps
                res_zeta = zeta*z - eps
                res_s = _lambda*s - eps
                res = np.hstack([res_x, res_y, res_z, res_lambda, res_xi,
res_eta, res_mu, res_zeta, res_s])
                res_norm_new = np.linalg.norm(res, ord=2)
            else:
                res_norm_new = None
            res_norm_new = comm.bcast(res_norm_new, root=0)
            step *= 0.5

        res_norm = res_norm_new
        res_max = np.linalg.norm(res, ord=np.inf) if comm.rank == 0 else None
```

```
            res_max = comm.bcast(res_max, root=0)
            eps *= 0.1

    return x
```

Appendix B.7 `utility` module for helper functions

```
import numpy as np
from scipy.spatial import cKDTree
from petsc4py import PETSc
import dolfinx.io
from dolfinx.fem import form, Function
from dolfinx import la
from dolfinx.fem.petsc import (create_vector, create_matrix,
    assemble_vector, assemble_matrix, set_bc)
import pyvista


def create_mechanism_vectors(func_space, in_spring, out_spring):
    """Create vectors for compliant mechanism design."""
    index_map = func_space.dofmap.index_map
    block_size = func_space.dofmap.index_map_bs
    spring_vec = la.create_petsc_vector(index_map, block_size)
    l_vec = spring_vec.copy()

    local_range = index_map.local_range
    local_indices = np.arange(local_range[0], local_range[1]).astype(np.int32)
    local_size = np.ptp(local_range)
    local_nodes = func_space.tabulate_dof_coordinates()[:local_size]

    for n, (locator, direction, value) in enumerate([in_spring, out_spring]):
        ctrl_nodes = local_indices[locator(local_nodes.T)]
        offset = ["x", "y", "z"].index(direction)
        ctrl_dofs = ctrl_nodes*block_size + offset
        spring_vec.setValues(ctrl_dofs, [value,]*ctrl_dofs.size)
        if n == 1:
            l_vec.setValues(ctrl_dofs, [1.0,]*ctrl_dofs.size)
    spring_vec.assemble()
    l_vec.assemble()
    return spring_vec, l_vec


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()
```

```
    def solve_adjoint(self):
        """Solve K*lambda=-L for the adjoint equation."""
        self.solver.solve(-self.l_vec, self.lam_wrap)
        self.lam.x.scatter_forward()

    def __del__(self):
        self.solver.destroy()
        self.lhs_mat.destroy()
        self.rhs_vec.destroy()
        self.u_wrap.destroy()
        self.lam_wrap.destroy()
        if self.spring_vec is not None:
            self.spring_vec.destroy()
            self.l_vec.destroy()


class Communicator():
    """Communicate information among different processes."""

    def __init__(self, func_space, mesh_serial, size=1):
        self.size = size
        self.comm = func_space.mesh.comm
        idx_map = func_space.dofmap.index_map

        num_local_nodes = idx_map.size_local
        num_global_nodes = idx_map.size_global
        num_nodal_dofs = func_space.dofmap.index_map_bs
        self.num_global_dofs = num_global_nodes * num_nodal_dofs

        local_nodal_range = np.asarray(idx_map.local_range, dtype=np.int32)  # [start, end]
        local_dof_range = local_nodal_range * num_nodal_dofs  # [start, end]
        local_nodes = func_space.tabulate_dof_coordinates()[:num_local_nodes]

        # Gather to Process 0
        local_nodal_range_gather = self.comm.gather(local_nodal_range, root=0)
        self.local_dof_range_gather = self.comm.gather(local_dof_range, root=0)
        local_nodes_gather = self.comm.gather(local_nodes, root=0)

        element = func_space.ufl_element()
        if self.comm.rank == 0:
            func_space_serial = dolfinx.fem.FunctionSpace(mesh_serial, element)

            nodes_serial = func_space_serial.tabulate_dof_coordinates()

            nodes_collect = np.zeros((num_global_nodes, 3))
            for r, nodes in zip(local_nodal_range_gather, local_nodes_gather):
                nodes_collect[r[0]:r[1]] = nodes
            global_to_local_nodes = compare_matrices(nodes_serial, nodes_collect)
            local_to_global_nodes = compare_matrices(nodes_collect, nodes_serial)

            def node2dof(nodes, num_nodal_dofs):
                return (np.tile(nodes, (num_nodal_dofs, 1))*num_nodal_dofs
                        + np.arange(num_nodal_dofs).reshape(-1, 1)).ravel("F")

            global_to_local_dofs = node2dof(global_to_local_nodes, num_nodal_dofs)
            self.local_to_global_dofs = node2dof(local_to_global_nodes, num_nodal_dofs)
            self.local_to_global_dofs = (
                np.tile(self.local_to_global_dofs.reshape(-1, 1), (1, size))*size + np.arange(size)).ravel()
        else:
            global_to_local_dofs = None
        global_to_local_dofs = self.comm.bcast(global_to_local_dofs, root=0)
        self.idx = global_to_local_dofs[local_dof_range[0]:local_dof_range[1]]

    def bcast(self, func, global_values):
        """Broadcast data from Process 0 to all the other processes."""
        if func.vector.size != global_values.size:
            raise ValueError("Mismatched sizes.")
        func.vector.array = global_values[self.idx]

    def gather(self, func):
        """Gather data to Process 0 from all the other processes."""
        if type(func) is Function:
            values_gather = self.comm.gather(func.vector.array, root=0)
        elif type(func) is PETSc.Vec:
```

```
            values_gather = self.comm.gather(func.array, root=0)
        elif type(func) is np.ndarray:
            values_gather = self.comm.gather(func, root=0)
        else:
            raise TypeError("Unsupported func.")

        if self.comm.rank == 0:
            values_collect = np.zeros(self.num_global_dofs*self.size)
            for r, local_values in zip(self.local_dof_range_gather, values_gather):
                values_collect[r[0]*self.size:r[1]*self.size] = local_values
            global_values = values_collect[self.local_to_global_dofs]
        else:
            global_values = None
        return global_values


def compare_matrices(array1, array2, precision=12, k=1):
    """Find the "args" such that array1[args] == array2."""
    kd_tree = cKDTree(array1.round(precision))
    return kd_tree.query(array2.round(precision), k=k)[1]


class Plotter():
    def __init__(self, mesh):
        """Initialize a plotter."""
        pyvista.OFF_SCREEN = True
        pyvista.start_xvfb()
        self.dim = mesh.topology.dim
        elements, cell_types, nodes = dolfinx.plot.vtk_mesh(mesh, self.dim)
        self.grid = pyvista.UnstructuredGrid(elements, cell_types, nodes)

    def plot(self, density, threshold=0.49, smooth_iter=100, path=""):
        self.grid.point_data["density"] = np.hstack(density)
        if self.dim == 2:
            grid = self.grid
        else:
            grid = self.grid.threshold(threshold).extract_surface()
        empty_mesh = (self.dim == 3 and grid.n_faces_strict == 0)

        if not empty_mesh:
            if self.dim == 3:
                grid = grid.smooth(n_iter=smooth_iter)
                grid.point_data["density"] = 0.4
            plotter = pyvista.Plotter()
            plotter.background_color = "white"
            lighting = self.dim == 3
            plotter.add_mesh(grid, clim=[0, 1], cmap="Greys", lighting=lighting,
                             show_scalar_bar=False)
            if self.dim == 2:
                plotter.view_xy()
            plotter.screenshot(path+"optimized_design.jpg", window_size=(1000, 1000))
            plotter.close()


def save_xdmf(mesh, rho, path=""):
    xdmf = dolfinx.io.XDMFFile(mesh.comm, path+"optimized_design.xdmf", "w")
    xdmf.write_mesh(mesh)
    rho.name = "density"
    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.

References

Aage N, Andreassen E, Lazarov BS, Sigmund O (2017) Gigavoxel 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

Bendsoe 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

Dalcin 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

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

Homayouni-Amlashi A, Schlinquer 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

Talischi 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, Ibhadode 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