SOPTX: A Modular and Extensible Framework for Topology Optimization with Multi-Backend Support

Liang He [1], Huayi Wei[2,*], Tian Tian[3]

[1] *School of Mathematics and Computational Science, Xiangtan University, Xiangtan 411105, China*
[2] *School of Mathematics and Computational Science, Xiangtan University; National Center of Applied Mathematics in Hunan, Hunan Key Laboratory for Computation and Simulation in Science and Engineering, Xiangtan 411105, China*
[3] *School of Mathematics and Computational Science, Xiangtan University, Xiangtan 411105, China*

Abstract. Topology optimization (TO) is a powerful structural design method, but its application remains challenging due to the deep expertise and extensive development effort required. Traditional TO methods, tightly coupled with computational mechanics like finite element method (FEM), result in intrusive algorithms demanding a comprehensive system understanding. This paper presents SOPTX, a TO package based on FEALPy, which implements a modular architecture that decouples analysis from optimization, supports multiple computational backends (NumPy, PyTorch, JAX), and achieves a non-intrusive design. Core innovations include: (1) a cross-platform design supporting multiple backends for efficient CPU/GPU execution, while leveraging automatic differentiation (AD) for sensitivity computation; (2) fast matrix assembly techniques that overcome performance bottlenecks of traditional methods, significantly accelerating finite element computations; (3) a modular framework supporting TO problems for arbitrary dimensions and meshes, allowing flexible configuration of optimization workflows. Using the density-based method for a classic compliance minimization problem, numerical experiments demonstrate SOPTX's significant improvements in computational speed and memory usage, showcasing its strong potential for research and engineering applications.

AMS subject classifications: 74P15, 68N30, 65N30

Key words: Topology Optimization, Multiple Computational Backends, Automatic Differentiation, Modular Framework

1 Introduction

Topology optimization (TO) is a class of structural optimization techniques aimed at improving structural performance by optimizing material distribution within a given design domain. Widely

*Corresponding author. *Email addresses:* lianghe@smail.xtu.edu.cn (L. He), weihuayi@xtu.edu.cn (H. Wei), tiantian@smail.xtu.edu.cn (T. Tian)

applied in aerospace, automotive, and civil engineering, TO addresses critical design challenges by efficiently utilizing materials. Among various approaches, density-based methods are particularly popular due to their intuitive concept and practicality. The Solid Isotropic Material with Penalization (SIMP) method, introduced by Bendsøe et al. [5], is the most widely adopted density-based approach, promoting binary (0 - 1) solutions by penalizing intermediate densities and integrating effectively with finite element method (FEM).

However, TO inherently involves large-scale computations due to the tight coupling between structural analysis and optimization. Each iteration requires solving boundary value problems and performing sensitivity analysis for gradient-based optimization algorithms. For large-scale problems, repeatedly solving large linear systems and evaluating sensitivities imposes significant demands on computational resources. Therefore, improving computational efficiency and scalability without sacrificing accuracy remains a major challenge in TO research and applications.

To lower the entry barrier and promote widespread adoption of TO, researchers have published numerous educational studies and literature. A pioneering example is the 99-line MATLAB code by Sigmund [30], which demonstrated the fundamentals of a two-dimensional SIMP algorithm in a concise and self-contained manner, profoundly impacting both TO education and research. Subsequently, improved versions of this educational code have emerged, including the more efficient 88-line version proposed by Andreassen et al. [2] and the extended 3D implementation by Liu and Tovar [21]. These codes convey practical TO knowledge simply and provide self-contained examples of basic numerical algorithms.

Meanwhile, efforts in open-source software development have also advanced TO practices. Chung et al. [11] leveraged OpenMDAO [14], decomposing TO into modular components with automated derivative assembly, enhancing flexibility and extensibility. Gupta et al. [18] developed a parallel-enabled implementation using the FEniCS finite element framework [1], addressing large-scale optimization. Recently, Ferro and Pavanello [12] provided an efficient 51-line TO implementation integrating FEniCS, Dolfin Adjoint, and Interior Point Optimizer, simplifying the development process. These projects provided standardized interfaces for convenient integration with existing finite element analysis (FEA) tools, reducing implementation complexity. However, despite improvements, existing open-source TO packages still face limitations in terms of functionality extensibility and flexibility, particularly for complex engineering applications.

To accelerate the development of TO, automating sensitivity analysis has become a critical step. This involves automatically computing derivatives of objectives, constraints, material models, projections, filters, and other components with respect to the design variables. Currently, the common practice involves manually calculating sensitivities, which can be tedious and error-prone despite their theoretical simplicity, often becoming a bottleneck in the development of new TO modules and exploratory research. Automatic differentiation (AD) provides an efficient and accurate approach for evaluating derivatives of numerical functions [15]. By decomposing complex functions into a series of elementary operations (such as addition and multiplication), AD accurately computes derivatives of arbitrary differentiable functions. In TO, the Jacobian matrices represent sensitivities of objective functions and constraints with respect to design variables, and software can automate this process, relieving developers from manually deriving and implementing sensitivity calculations. With its capability of easily obtaining accurate derivative information,

AD offers significant advantages in design optimization, particularly for highly nonlinear optimization problems.

In recent years, the adoption of AD in TO has gradually increased. For instance, Nørgaard et al. [24] employed the AD tools CoDiPack and Tapenade to achieve automatic sensitivity analysis in unsteady flow TO, significantly enhancing computational efficiency. Building upon this, Chandrasekhar [10] utilized the high-performance Python library JAX [8] to apply AD techniques in density-based TO, achieving efficient solutions to classical TO problems, exemplified by compliance minimization.

In addition to sensitivity analysis, computational efficiency remains a critical challenge for large-scale topology optimization, particularly for high-resolution 3D problems. Harnessing the massive parallelism of GPUs has become a mainstream solution. For instance, Träff et al. [36] focused on code simplicity and efficiency, releasing a lightweight GPU-accelerated code based on C++ and OpenMP, demonstrating that high-performance computing for 3D problems can be achieved through simple parallel programming models. More recently, to circumvent the limitations of proprietary software and leverage the ecosystem advantages of Python, Hou et al. [19] developed a parallel computing framework based on the Python library CuPy. They proposed a vectorized programming approach based on a novel decomposition of the finite element matrix, transforming Sparse Matrix-Vector multiplication (SpMV) into efficient vector operations. This approach significantly reduced GPU memory usage and enhanced computational speed, enabling the efficient solution of topology optimization problems with tens of millions of degrees of freedom. These advancements underscore the immense potential of hardware acceleration in handling large-scale design problems.

Although existing TO programs, ranging from educational codes to open-source implementations, have significantly advanced the field, they often employ a procedural programming paradigm that tightly couples analysis modules, such as FEA, with optimization modules. This tight coupling restricts software extensibility, necessitating invasive changes for new features, and diminishes reusability, complicating integration into broader frameworks like multidisciplinary design optimization (MDO) in aerospace. Consequently, developing a decoupled architecture that enhances both extensibility and reusability without sacrificing accuracy remains a critical challenge in TO.

To address the aforementioned challenges, this paper proposes SOPTX, a modular and extensible TO framework built upon the FEALPy platform [37]. FEALPy is an open-source intelligent CAX computing engine providing an efficient, flexible, and extensible platform for numerical simulation. Its selection as the underlying platform is supported by key advantages:

1. FEALPy supports multiple computational backends, including NumPy, PyTorch, and JAX, enabling efficient execution across a wide range of hardware architectures such as central processing units (CPUs) and graphics processing units (GPUs).

2. FEALPy supports a broad range of numerical schemes, including FEM of arbitrary order and dimension, as well as alternative methods such as the virtual element method (VEM) and the finite difference method (FDM).

3. FEALPy adopts highly efficient vectorized operations that fully exploit the computational power of modern processors.

4. FEALPy features a clean and extensible API design, which aligns well with our goal of developing a modular and reusable TO framework.

Consequently, FEALPy provides a robust and versatile foundation for developing modular and extensible TO frameworks.

Leveraging the strengths of FEALPy, the SOPTX framework achieves a highly modular design. SOPTX adopts a component-based philosophy commonly used in MDO, decomposing complex systems into independent and reusable modules to enhance flexibility and maintainability. Specifically, SOPTX introduces a clear separation between analysis and optimization: numerical solvers, AD tools, and optimization algorithms are designed as independent and interchangeable modules. This design allows users to flexibly select or replace individual modules based on specific needs without requiring substantial changes to the overall code structure. Through this modular architecture, SOPTX not only inherits the efficiency and flexibility of FEALPy but also significantly enhances the framework's extensibility and reusability, providing robust support for complex engineering applications.

Building on this modular foundation, SOPTX seamlessly integrates AD into the TO workflow. It supports multiple computational backends and can automatically switch between them based on hardware availability and performance requirements. Additionally, SOPTX mitigates the computational bottlenecks associated with traditional numerical integration through a fast matrix assembly technique. This technique separates element-dependent and element-independent components, avoiding redundant computations of invariant data during iterative procedures. Furthermore, symbolic integration via SymPy [22] replaces conventional numerical quadrature, reducing computational cost while improving accuracy. SOPTX employs a non-intrusive design, allowing users to easily replace or extend filters, constraints, and objectives without modifying the core framework. Overall, SOPTX is designed as a low-barrier, flexible and efficient platform that, through the combination of modularity and AD, empowers users to focus on algorithmic innovation and problem modeling rather than implementation details. This makes it well-suited for education, research, and complex multiphysics-coupled scenarios in TO and MDO.

The remainder of this paper is organized as follows. Section 2 introduces the mathematical formulation of TO, covering density-based methods, compliance minimization, optimization algorithms, and filtering techniques. Section 3 presents the design philosophy of SOPTX built on FEALPy, with emphasis on its modular architecture and backend switching for cross-platform, flexible implementation. Section 4 demonstrates SOPTX installation and usage via a classical 2D cantilever beam problem. Section 5 showcases its effectiveness through a series of 2D and 3D examples, including comparative studies with different filters and optimization algorithms, and performance analyses of fast matrix assembly, AD, and backend switching. Section 6 concludes the paper and outlines future research directions.

2 Density-based Topology Optimization: Formulation, Algorithms, and Regularization

In this section, we introduce the density-based method in topology optimization (TO), explaining how it optimizes material distribution using a density function and interpolation models. Using the compliance minimization problem as an example, we present its continuous and discrete formulations. We then describe two key optimization algorithms in TO: the Optimality Criteria (OC) method and the Method of Moving Asymptotes (MMA), including their implementations and features. Finally, we discuss key filtering techniques in TO, such as sensitivity, density, and Heaviside projection filters, and their role in improving numerical stability and physical feasibility.

2.1 Density-based Method

The core objective of TO is to determine the optimal material distribution within a given design domain to achieve predefined performance targets. Density-based methods are among the most widely used strategies in the field of TO. These methods parameterize the structure by defining a material density function $\rho(x)$, where $\rho(x)=1$ represents solid material, $\rho(x)=0$ represents voids, and intermediate values $0<\rho(x)<1$ are used to handle the non-convexity of discrete problems.

In a discretized design domain, the mechanical properties (e.g., stiffness) of material elements transition smoothly between solid and void states via interpolation models [3]. A widely used model is the SIMP approach [38], establishes a power-law relationship between the material density ρ and Young's modulus E:

$$E(\rho)=\rho^p E_0, \quad \rho\in[0,1],$$

where E_0 denotes the Young's modulus of solid material, and $p>1$ is the penalization factor. By increasing the relative stiffness cost of intermediate densities, this power-law relationship encourages the optimization results toward a clear 0-1 distribution.

To avoid singularities in the stiffness matrix, a modified SIMP model introduces a small minimum Young's modulus $E_{\min}$:

$$E(\rho)=E_{\min}+\rho^p(E_0-E_{\min}), \quad \rho\in[0,1],$$

where $E_{\min}=10^{-9}E_0$. This modification ensures that the stiffness matrix remains positive definite at $\rho=0$, preventing numerical failure [31].

However, optimization results may still exhibit intermediate densities, or "gray areas", which can complicate manufacturing. To address this, techniques like Heaviside projection and sensitivity/density filtering are commonly used to achieve clearer topologies and mitigate numerical issues such as checkerboard patterns.

2.2 Compliance Minimization Problem

The goal of compliance minimization with a volume constraint is to find a material density field $\rho(x)\in[0,1]$ that minimizes the total strain energy of an elastic body occupying a bounded domain

$\Omega \subset \mathbb{R}^d$ $(d=2,3)$ under prescribed loads and boundary conditions, while the available material does not exceed a given volume $V^* \in (0,|\Omega|)$, where $|\Omega|$ is the total volume of the design domain. Throughout this paper we assume the displacement $u \in H^1(\Omega;\mathbb{R}^d)$, the density $\rho \in L^\infty(\Omega)$ and the material stiffness tensor $D(\rho) \in L^\infty(\Omega;\mathbb{R}^{d\times d\times d\times d})$, where $\mathbb{R}^d$ denotes the d-dimensional Euclidean space and $\mathbb{R}^{d\times d\times d\times d}$ the space of fourth-order tensors over $\mathbb{R}^d$.

The compliance minimization problem in its continuous form can be formulated as follows:

$$\begin{aligned} \min_{\rho}&: c(\rho) = \int_\Omega \sigma(u):\varepsilon(u)\,\mathrm{d}x \\ \text{subject to}&: g(\rho) = v(\rho) - V^* \le 0, \quad 0 \le \rho(x) \le 1, \\ &\quad -\nabla\cdot\sigma(u) = f \quad \text{in } \Omega, \\ &\quad u = 0 \text{ on } \Gamma_D, \quad \sigma(u)\cdot n = t \text{ on } \Gamma_N, \end{aligned} \tag{2.1}$$

where $c(\rho)$ is defined as $\int_\Omega \sigma(u):\varepsilon(u)\,\mathrm{d}x$, this definition equals twice the strain energy, a convention widely adopted in TO literature, $g(\rho) = v(\rho) - V^* \le 0$ denotes the volume constraint, $v(\rho) = \int_\Omega \rho\,\mathrm{d}x$ is the material volume, $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$ is the strain tensor, and $\sigma(u) = D(\rho):\varepsilon(u)$ is the stress tensor. Here f and t denote body forces and surface tractions, respectively. The boundary is decomposed into Dirichlet and Neumann parts such that $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \emptyset$.

Discretising Ω into N_e finite elements and letting $\boldsymbol{\rho} = [\rho_1, \rho_2, \ldots, \rho_{N_e}]^T$ be the element-wise constant densities, the problem becomes

$$\begin{aligned} \min_{\boldsymbol{\rho}}&: c(\boldsymbol{\rho}) = \mathbf{F}^T\mathbf{U}(\boldsymbol{\rho}) \\ \text{subject to}&: g(\boldsymbol{\rho}) = v(\boldsymbol{\rho}) - V^* \le 0, \quad 0 \le \rho_e \le 1, \\ &\quad \mathbf{K}(\boldsymbol{\rho})\mathbf{U} = \mathbf{F}, \end{aligned} \tag{2.2}$$

with

$$v(\boldsymbol{\rho}) = \sum_{e=1}^{N_e} v_e \rho_e, \quad \mathbf{K}(\boldsymbol{\rho}) = \sum_{e=1}^{N_e} E(\rho_e)\mathbf{K}_e^0.$$

Here v_e is the volume of element e, $E(\rho_e)$ follows the SIMP interpolation introduced in Section 2.1, and $\mathbf{K}_e^0$ is the elemental matrix for unit Young's modulus.

Solving $\mathbf{K}(\boldsymbol{\rho})\mathbf{U}(\boldsymbol{\rho}) = \mathbf{F}$ yields the discrete displacement vector $\mathbf{U}$, after which the objective $c(\boldsymbol{\rho})$ and its sensitivities are evaluated for the optimization algorithm.

2.3 Optimization Algorithms

The Optimality Criteria (OC) method is a classical TO algorithm widely used for compliance minimization problems with volume constraints. It iteratively adjusts the material density ρ_e to satisfy the optimality conditions, based on the update factor:

$$B_e = -\frac{\partial c(\rho)}{\partial \rho_e}\left(\lambda \frac{\partial g(\rho)}{\partial \rho_e}\right)^{-1},$$

where λ is the Lagrange multiplier associated with the volume constraint.

Bendsøe [4] proposed a heuristic update scheme for the design variables:

$$\rho_e^{\text{new}} = \begin{cases} \max(0,\, \rho_e - m) & \text{if} \quad \rho_e B_e^{\eta} \le \max(0,\, \rho_e - m) \\ \min(1,\, \rho_e + m) & \text{if} \quad \rho_e B_e^{\eta} \ge \min(1,\, \rho_e + m) \\ \rho_e B_e^{\eta} & \text{if} \quad \text{otherwise} \end{cases}$$

where m is the move limit and $\eta \in (0,1]$ is a damping exponent. The algorithm is summarized in Algorithm 2.1.

Algorithm 2.1 OC pseudo-code

Input: Initial design variables $\boldsymbol{\rho}^{(0)}$, volume constraint V^*, move limit m, damping factor η, maximum iterations MaxIter, tolerance ϵ
Output: Optimized design variables $\boldsymbol{\rho}$
Set $\boldsymbol{\rho}^{(k)} \leftarrow \boldsymbol{\rho}^{(0)}$, $k \leftarrow 0$;
while $k <$ MaxIter *and* $\|\boldsymbol{\rho}^{(k+1)} - \boldsymbol{\rho}^{(k)}\|_\infty > \epsilon$ **do**
    Compute objective, constraint, and sensitivities;
    (Optional) Apply filters;
    Update λ using bisection;
    Update $\boldsymbol{\rho}^{\text{new}}$ using optimality criterion;
    Set $\boldsymbol{\rho}^{(k+1)} \leftarrow \boldsymbol{\rho}^{\text{new}}$, $k \leftarrow k+1$;
end

In TO, in addition to the OC method, the MMA is a widely used gradient-based optimization algorithm proposed by Svanberg [34]. It is particularly well-suited for problems involving multiple constraints or complex objective functions. The core idea of MMA is to dynamically adjust the approximation range of the design variables using "moving asymptotes", thereby transforming the original nonlinear optimization problem into a sequence of convex subproblems. This strategy ensures numerical stability throughout the iterative process.

In the compliance minimization problem, MMA approximates the original problem by the following convex subproblem:

$$\begin{aligned} \min:\ & \tilde{f}_0^{(k)}(\boldsymbol{\rho}) + a_0 z + \sum_{i=1}^{m_c} \left(c_i y_i + \frac{1}{2} d_i y_i^2\right) \\ \text{subject to:}\ & \tilde{f}_i^{(k)}(\boldsymbol{\rho}) - a_i z - y_i \le 0, && i = 1, \cdots, m_c \\ & \alpha_j^{(k)} \le \rho_j \le \beta_j^{(k)}, && j = 1, \cdots, n \\ & y_i \ge 0,\ z \ge 0, \end{aligned}$$

where m_c is the number of constraints, n is the number of design variables, y_i are auxiliary variables for the constraints, and z is an auxiliary variable for the objective. The approximation $\tilde{f}_0^{(k)}(\boldsymbol{\rho})$

is the convex approximation of the objective function, while $\tilde{f}_i^{(k)}(\boldsymbol{\rho})$ approximates the i-th constraint, both dynamically adjusted based on current gradient information. The bounds $\alpha_j^{(k)}$ and $\beta_j^{(k)}$ are the dynamic lower and upper limits for the design variable ρ_j in the k-th iteration. The parameters a_0, a_i, c_i, d_i are given constants.

The procedure of the MMA method is summarized in Algorithm 2.2.

Algorithm 2.2 MMA pseudo-code

Input: Initial design variables $\boldsymbol{\rho}^{(0)}$, problem parameters, MMA parameters
Output: Optimized design variables $\boldsymbol{\rho}$
Set $\boldsymbol{\rho}^{(k)} \leftarrow \boldsymbol{\rho}^{(0)}$, $k \leftarrow 0$;
while $k < \text{MaxIter}$ *and* $\|\boldsymbol{\rho}^{(k+1)} - \boldsymbol{\rho}^{(k)}\|_\infty > \epsilon$ **do**
    Compute sensitivities;
    (Optional) Apply filters;
    Update asymptotes and variable bounds;
    Construct and solve the MMA subproblem to obtain $\boldsymbol{\rho}^{\text{new}}$;
    Set $\boldsymbol{\rho}^{(k+1)} \leftarrow \boldsymbol{\rho}^{\text{new}}$, $k \leftarrow k+1$
end

2.4 Filtering Methods

In TO, numerical issues such as mesh dependency, checkerboard patterns, and local minima often arise [6]. Filtering techniques, as a form of regularization, smooth the design variables or sensitivities to mitigate these problems and enhance the physical realizability of optimized structures [31, 32].

The sensitivity filter, proposed by Sigmund [29], smooths the sensitivities via a weighted average:

$$\widetilde{\frac{\partial f}{\partial \rho_i}} = \frac{1}{\max\{\gamma, \rho_i\} \sum_{j \in N_i} H_{ij}} \sum_{j \in N_i} H_{ij} \rho_j \frac{\partial f}{\partial \rho_j},$$

where $H_{ij} = \max\{0, r_{\min} - \text{dist}(i,j)\}$ is a linearly decaying weight, N_i is the set of elements within filter radius $e_{\min}$, and $\gamma = 10^{-3}$ is a stabilizing constant.

The density filter, proposed by Bruns and Tortorelli [9] and mathematically justified by Bourdin [7], smooths the material distribution:

$$\tilde{\rho}_i = \frac{\sum_{j \in N_i} H_{ij} v_j \rho_j}{\sum_{j \in N_i} H_{ij} v_j},$$

where $\tilde{\rho}_i$ is the filtered physical density, ρ_j is the original design variable associated with element j, and v_j is the volume of element j.

The density filter computes a weighted average of densities using weights H_{ij}, smoothing the material distribution. The filtered density $\tilde{\rho}_i$, rather than the design variable ρ_i, is used to evaluate structural performance and constraints, serving as the final design in practice.

Under density filtering, the sensitivity of a function ψ with respect ot ρ_j is:

$$\frac{\partial \psi}{\partial \rho_j} = \sum_{i \in N_j} \frac{\partial \psi}{\partial \tilde{\rho}_i} \frac{\partial \tilde{\rho}_i}{\partial \rho_j} = \sum_{i \in N_j} \frac{H_{ij} v_j}{\sum_{k \in N_i} H_{ik} v_k} \frac{\partial \psi}{\partial \tilde{\rho}_i},$$

where ψ is an objective or constraint function, and N_j s the set of elements influenced by ρ_j.

Heaviside projection filtering [17] projects the filtered density to a physical density $\bar{\rho}_i$, achieving a clear black-and-white topology and enforcing a minimum length scale. The projection is defined as:

$$\bar{\rho}_i = 1 - e^{-\beta \tilde{\rho}_i} + \tilde{\rho}_i e^{-\beta},$$

where β controls smoothness. A continuation scheme gradually increases β to enhance the projection effect and ensure numerical stability.

After applying the Heaviside projection filter, the sensitivities of a function with respect to $\tilde{\rho}_i$ must also be computed using the chain rule:

$$\frac{\partial \psi}{\partial \tilde{\rho}_i} = \frac{\partial \psi}{\partial \bar{\rho}_i} (\beta e^{-\beta \tilde{\rho}_i} + e^{-\beta}).$$

The filter radius $r_{\min}$ significantly affects the result: small $r_{\min}$ leads to oscillations or checkerboard patterns, while large $r_{\min}$ causes excessive smoothing and loss of design details.

3 SOPTX Framework Design and Multi-Backend Switching

This section introduces the design principles and key components of the SOPTX framework, emphasizing its multi-backend switching capability for improved computational performance and flexibility on central processing units (CPUs) and graphics processing units (GPUs). The section is divided into three parts: an overview of the FEALPy architecture as the tensor computation foundation, a description of SOPTX's modular structure (material, solver, filter, and optimization modules), and an analysis of the multi-backend mechanism supporting NumPy, PyTorch, and JAX for diverse computational needs.

3.1 FEALPy Architecture Design

As shown in Figure 1, FEALPy adopts a layered architecture with four levels, arranged from bottom to top: tensor, common, algorithm, and field. Building on this, FEALPy introduces the *Tensor Backend Manager*, which unifies the management of computational backends like NumPy, PyTorch, and JAX. This manager provides a consistent tensor operation interface, following the Python Array API Standard v2023.12 [13], enabling seamless adaptation across various software and hardware platforms.

Specifically, the functions of each layer are as follows:

1. **Tensor level**: Provides core tensor operations and manages backends via the *Tensor Backend Manager*, supporting SOPTX's multi-backend switching. It also enables AD in PyTorch and JAX, simplifying sensitivity computations in TO.

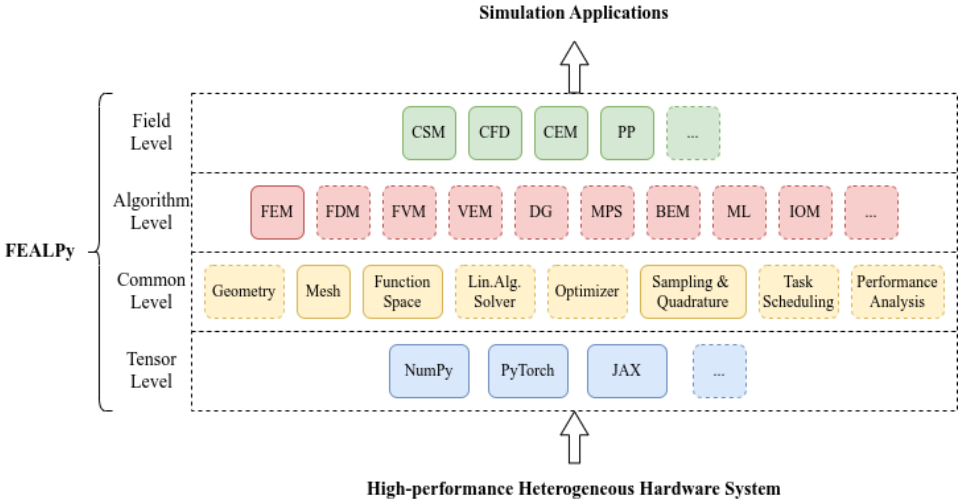


Figure 1: The layered architecture of FEALPy, comprising tensor, common, algorithm, and field levels, progressing from low-level functionalities to high-level applications. Modules in dashed boxes are under development.

2. **Common level**: Includes mesh generation and finite element spaces, supporting rapid construction of meshes and function spaces for finite element analysis in SOPTX.

3. **Algorithm level**: Encompasses solvers and optimization algorithms, offering efficient and stable computational support for SOPTX's optimization methods.

4. **Field level**: Targets specific physical problems (e.g., linear elasticity), enabling SOPTX to address diverse TO problems, such as compliance minimization under volume constraints, with tailored application support.

This layered design ensures FEALPy's functionality, extensibility, and performance, with the *Tensor Backend Manager* abstracting backend differences to enhance user focus on algorithms and applications.

3.2 SOPTX Architecture Design

The SOPTX framework is positioned within the field level of FEALPy's layered architecture, targeting structural topology optimization (STO) applications. SOPTX fully inherits and extends FEALPy's *Tensor Backend Manager*. It also leverages a variety of numerical algorithm components and generic mesh and geometry handling capabilities. This design ensures flexibility and extensibility while enhancing computational efficiency for TO problems.

As shown in Figure 2, SOPTX adopts a modular architecture consisting of four primary components: the material, solver, filter, and optimizer modules. These components communicate

and share data through clearly defined interfaces, forming a loosely coupled and easily extensible multi-backend framework for TO.

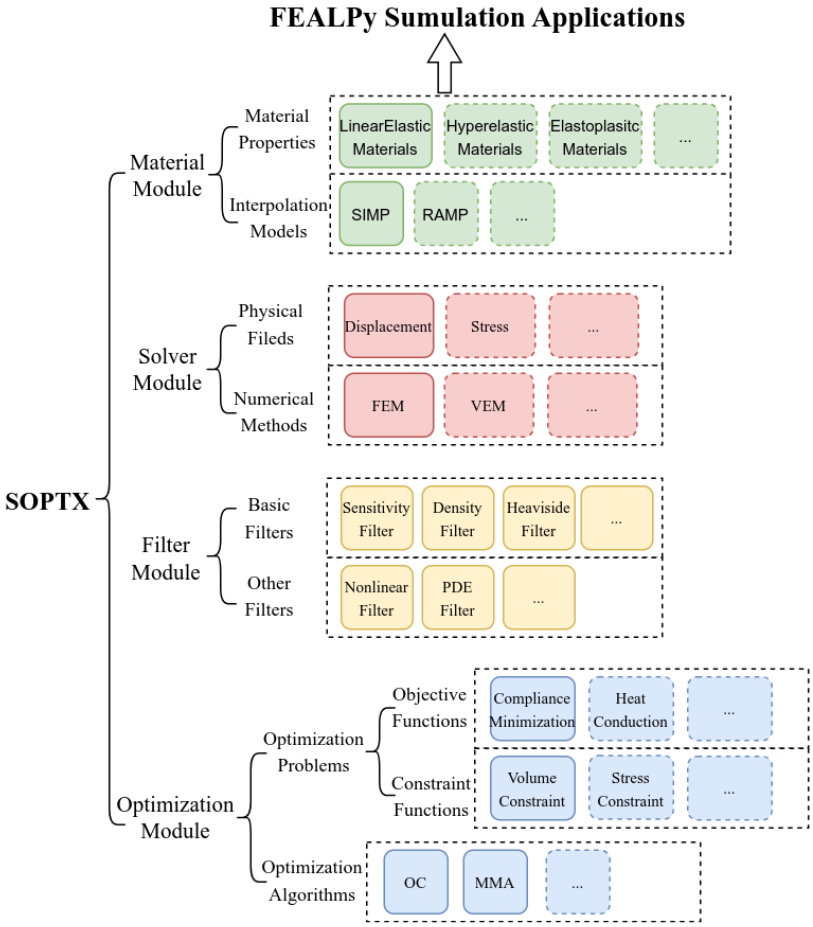


Figure 2: The modular architecture of SOPTX, consisting of material, solver, filter, and optimization modules. The material module serves as the foundation, while the solver and filter modules manage intermediate computations, collectively supporting the optimization module. Components in dashed boxes are under development.

3.2.1 Material Module

The material module in SOPTX currently focuses on linear elastic materials, extending FEALPy's implementation with TO-specific interfaces for computing Lamé constants, elasticity matrices, and strain matrices. It implements the SIMP model, commonly employing a penalization factor of $p = 3$, to penalize continuous density fields towards discrete material distributions. The module's abstract interface design ensures extensibility. This allows future support for other models, such as the Rational Approximation of Material Properties (RAMP), or complex behaviors like anisotropy,

hyperelasticity, and elastoplasticity through subclassing.

The module encapsulates the interpolation, update, and evaluation of elastic constants within a unified interface, providing essential material data for downstream modules. Its key functional interfaces include:

1. Computing elasticity tensors from density fields for the solver module.
2. Efficient batch updates of material properties for iterative optimization.
3. Providing base material parameters (e.g., elastic constants) for optimization.

These interfaces ensure consistent backend compatibility and optimized performance. The material module bridges physical modeling and numerical optimization, maintaining independence while supplying essential data throughout the TO workflow. Its modular design simplifies current linear elasticity solutions and provides a flexible foundation for future material model extensions.

3.2.2 Solver Module

The solver module forms the computational core of the SOPTX framework, tasked with solving physical field problems (e.g., displacement fields) based on given material properties, boundary conditions, and external loads. The module is designed to be efficient, flexible, and backend-independent. This design addresses the demands of repeatedly solving large-scale linear systems in TO. To achieve this, the module enhances computational efficiency through optimized matrix assembly, intelligent caching, and multi-backend acceleration, supporting backends including NumPy, PyTorch, and JAX.

In its current version, SOPTX primarily adopts the FEM to solve linear elasticity equations, leveraging FEALPy's finite element module. The key features of the solver module include:

- Dimensional and element adaptability: It supports various spatial dimensions, element types, and boundary conditions, where the density field is represented as piecewise constant per element and the displacement field uses linear finite elements, ensuring both stability and computational efficiency.
- Efficient numerical strategies: The module implements a fast matrix assembly technique that separates element-independent and element-dependent components to eliminate redundant computations. Additionally, it incorporates symbolic integration to boost efficiency by precomputing exact expressions.
- Multiple solution strategies: It provides both direct solvers (e.g., MUMPS) and iterative solvers (e.g., Conjugate Gradient (CG), with automatic optimization tailored to the backend, such as utilizing GPU acceleration on PyTorch and JAX.

The solver module is designed with strong extensibility, enabling the future incorporation of additional physical fields (e.g., stress fields) and numerical methods (e.g., Virtual Element Method), as well as adaptation to nonlinear mechanics and multiphysics coupling scenarios."

Within the SOPTX framework, the solver module acts as a bridge between physical analysis and optimization computation through the following interactions:

- Interaction with the material module: It receives material properties, such as elasticity matrices, via standardized interfaces, ensuring the solver remains independent of specific material interpolation schemes.
- Output to the optimization module: It provides the displacement field for evaluating the objective function and the stiffness matrix for sensitivity analysis.
- Result reuse: It employs an intelligent caching mechanism to avoid recomputing invariant components. This approach significantly reduces computational overhead.

Through its modular design and well-defined interfaces, the solver module efficiently conducts physical field computations, providing critical support for the TO process while ensuring both consistency and high performance across multiple backends.

3.2.3 Filter Module

The filter module in SOPTX is a cornerstone of TO, tasked with processing design variables and applying regularization techniques. Its primary functions are to mitigate numerical instabilities, such as checkerboarding and mesh dependency, and to improve the manufacturability of optimized structures. Adhering to the framework's loosely coupled design, the filter module operates as an independent unit with efficient data exchange across modules. It offers a unified interface that supports multiple computational backends. Backend-specific optimizations ensure high computational efficiency.

The module implements three key filtering techniques:

- Sensitivity filtering: Applies weighted averaging to objective function sensitivities, which effectively suppresses checkerboard patterns.
- Density filtering: Maps raw design variables to physical densities, addressing the locality limitations of sensitivity filtering.
- Heaviside projection filtering: Enhances density filtering with a smoothed Heaviside function to achieve distinct black-and-white designs by driving intermediate densities toward 0 or 1.

The module employs a KD-tree-based neighborhood search to construct filtering matrices, which enables efficient regularization on unstructured meshes and complex geometries. Optimization strategies, such as precomputed neighborhoods and sparse matrix representations, ensure scalability for large-scale TO problems.

The filter module is highly extensible. Users can subclass the base filter class to implement custom algorithms, such as nonlinear filters or PDE-based filters. This approach seamlessly inherits multi-backend compatibility. Additionally, the framework supports chained filter combinations, allowing for customized regularization strategies to meet various optimization needs.

The filter module interacts with other components in the following ways:

- Processing design variables: It processes raw design variables into filtered physical densities.

- Interaction with optimization module: It filters unprocessed sensitivities ensuring numerical stability during optimization.
- Interaction with material module: It provides filtered densities for material interpolation, defining a streamlined data flow:

$$\text{design variables} \rightarrow \text{filtering} \rightarrow \text{physical density} \rightarrow \text{material properties} \rightarrow \text{physical response}$$

Through its modular and extensible design, the filter module not only addresses numerical challenges in TO but also lays a robust foundation for advanced regularization, enabling high-quality, manufacturable structural designs within the SOPTX framework.

3.2.4 Optimization Module

The optimization module serves as the computational core of the SOPTX framework, tasked with defining and solving optimization problems by integrating physical field solutions with user-defined objectives. It collaborates seamlessly with the material, solver, and filter modules to drive the TO pipeline. Adhering to the framework's multi-backend design, it offers a unified interface across NumPy, PyTorch, and JAX, utilizing backend-specific optimizations, including GPU acceleration in PyTorch and JAX, to enhance performance.

The module currently focuses on compliance minimization under volume constraints, a foundational problem in TO. It supports two mainstream algorithms: Optimality Criteria (OC) and the MMA. OC is a lightweight method for volume-constrained problems, while MMA, based on Svanberg's 2007 formulation [**?**], has been adapted for multi-backend compatibility and optimized for efficiency.

The design of the optimization module follows two core principles

1. Decoupling of problem and algorithm: The optimization problem (e.g., objectives, constraints, sensitivities) is separated from the solving algorithm, which enables flexible pairing of formulations and solvers.
2. Dual-mode sensitivity: Supports both manually derived sensitivities for transparency and AD in PyTorch or JAX for ease and scalability.

This design ensures high extensibility, enabling users to define custom problems by subclassing the base problem class to introduce new objectives (e.g., heat conduction, compliant mechanisms) or constraints (e.g., stress, frequency). Additionally, users can integrate other optimization algorithms, such as Sequential Quadratic Programming (SQP) or Sequential Linear Programming (SLP), in a modular fashion.

The optimization module serves as a central hub within the SOPTX framework, interacting with other modules as follows:

- Interaction with solver module: It utilizes physical outputs, such as displacement fields and stiffness matrices, to compute objectives and sensitivities.

- Interaction with Filter Module: It exchanges sensitivities with the filter module for processing and updates, and receives filtered physical densities derived from design variables.

With its modular and extensible architecture, the optimization module effectively addresses classical TO problems while providing a versatile foundation for advanced multiphysics optimization. It acts as the decision-making hub of SOPTX, balancing computational performance, flexibility, and user adaptability.

3.3 Multi-Backend Switching

In STO, computational performance is a central concern in both research and engineering. To meet diverse computational demands, the SOPTX framework builds upon FEALPy to implement a flexible multi-backend architecture. This allows seamless switching between tensor computation backends such as NumPy, PyTorch, and JAX, enhancing the software's applicability, efficiency, portability, and flexibility across various hardware and software platforms.

Specifically, each backend offers distinct advantages for different scenarios

- **NumPy**: Suited for small-scale tasks and rapid prototyping, offering stable performance and efficient memory management on CPU platforms.
- **PyTorch and JAX**: Both offer GPU acceleration and AD, making them ideal for large-scale or high-dimensional problems. AD facilitates sensitivity analysis, while JAX provides just-in-time (JIT) compilation and automatic vectorization, further enhancing performance on GPU platforms.

4 Getting Started with SOPTX

This section provides installation instructions and usage guidance for SOPTX, enabling readers to quickly grasp the framework's operations and apply it to topology optimization (TO) tasks. Built on FEALPy, SOPTX leverages a multi-backend switching mechanism and modular design to deliver an efficient and flexible computational environment. The section is structured into two sections: first, a detailed guide on installing SOPTX and its dependency FEALPy, ensuring proper configuration of the development environment; second, a demonstration of SOPTX's workflow through a classical 2D cantilever beam compliance minimization example.

4.1 Software Installation

SOPTX is a TO toolkit built on top of FEALPy, an intelligent CAX simulation engine that provides numerical computing capabilities. Installing FEALPy is required before SOPTX. For flexibility and the latest features, install both from source. Ensure Git and Python are installed. Use a virtual environment to avoid dependency conflicts.

Core installation steps:

1. Clone the FEALPy repository from GitHub:

```
git clone https://github.com/weihuayi/fealpy.git
```

2. Change into the FEALPy directory and install it in editable mode:

```
cd fealpy
pip install -e .
```

3. Similarly, install SOPTX:

```
git clone https://github.com:weihuayi/soptx.git
cd soptx
pip install -e .
```

For the complete installation guide, please refer to the official documentation at: https://github.com/weihuayi/fealpy.

4.2 Example: 2D Cantilever Beam

We use a popular compliance minimization benchmark to demonstrate the usage of SOPTX: minimizing the structural compliance of a cantilever beam under tip loading (see Figure 3) [6]. The left end of the beam is fixed, and a downward concentrated load $T = -1$ is applied to the bottom of the right end. A 160×100 uniform quadrilateral mesh is used. The target volume fraction is set to 0.4, with material properties $E = 1$ and $\nu = 0.3$. The penalization factor is $p = 3$, and the sensitivity filter radius is $r = 6.0$, which matches the mesh element size to ensure structural smoothness and eliminate checkerboard patterns. To facilitate reproducibility, the complete source code for this demonstration is available on GitHub at https://github.com/brighthe/soptx/blob/main/soptx/demo/cicp_cantil_2d.py.

To begin, we import essential modules from FEALPy (backend, mesh, function spaces) and SOPTX (material, solver, filter, optimization), as shown in Code 1. The partial differential equation (PDE) model for the cantilever beam, defined in `Cantilever2dData1` (see Appendix A), specifies the geometry, load, and boundary conditions via standard SOPTX interfaces.

Code 1: Module imports and PDE model

```
from fealpy.backend import backend_manager as bm
from fealpy.mesh import UniformMesh
from fealpy.functionspace import LagrangeFESpace,
    TensorFunctionSpace
from soptx.material import DensityBasedMaterialConfig,
    DensityBasedMaterialInstance
from soptx.solver import ElasticFEMSolver, AssemblyMethod
```

Figure 3: Cantilever beam geometry: fixed on the left, with a downward concentrated load on the right.

```
from soptx.filter_ import SensitivityBasicFilter
from soptx.opt import ComplianceObjective, ComplianceConfig,
    VolumeConstraint, VolumeConfig
from soptx.opt import OCOptimizer
from soptx.pde import Cantilever2dData1

pde = Cantilever2dData1(xmin=0, xmax=160, ymin=0, ymax=100, T
    = -1)
```

Next, we define the mesh and finite element spaces (Code 2). The displacement field uses a continuous and piecewise linear Lagrange finite element space, while the density field is represented in a piecewise constant Lagrange element space, aligning with typical TO discretizations.

Code 2: Mesh and function space definitions

```
mesh = UniformMesh(extent=[0, 160, 0, 100], h=[1, 1], origin
    =[0, 0])
space_C = LagrangeFESpace(mesh=mesh, p=1, ctype='C')
tensor_space_C = TensorFunctionSpace(scalar_space=space_C,
    shape=(-1, 2))
space_D = LagrangeFESpace(mesh=mesh, p=0, ctype='D')
```

The material module is then instantiated with the specified properties and SIMP interpolation model (Code 3). This module handles material property computations based on the density field.

Code 3: Material module

```
material_config = DensityBasedMaterialConfig(
        elastic_modulus=1.0, minimal_modulus=1e-9,
```

```
        poisson_ratio=0.3, plane_assumption="plane_stress",
        interpolation_model="SIMP", penalty_factor=3.0)
materials = DensityBasedMaterialInstance(config=
    material_config)
```

Subsequently, the solver module is initialized with the material module, PDE model, standard matrix assembly, and a direct solver (MUMPS), as shown in Code 4. A sensitivity filter is also set up to regularize the optimization.

Code 4: Solver and filter module

```
solver = ElasticFEMSolver(materials=materials,
            tensor_space=tensor_space_C, pde=pde,
            assembly_method=AssemblyMethod.STANDARD,
            solver_type='direct',
            solver_params={'solver_type': 'mumps'})
sens_filter = SensitivityBasicFilter(mesh=mesh, rmin=6.0)
```

The optimization problem is defined by instantiating the compliance objective and volume constraint classes, followed by configuring the Optimality Criteria (OC) optimizer with a maximum of 200 iterations and a convergence tolerance of 0.01 (Code 5).

Code 5: Optimization module

```
objective = ComplianceObjective(solver=solver)
constraint = VolumeConstraint(solver=solver, volume_fraction
    =0.4)
optimizer = OCOptimizer(objective=objective,
            constraint=constraint, filter=sens_filter,
            options={'max_iterations': 200, 'tolerance':
    0.01})
```

Finally, the initial density field is uniformly set to the target volume fraction, and the optimization is executed. Results are saved, and convergence history is plotted (Code 6).

Code 6: Main program and post-processing

```
if __name__ == "__main__":
    @cartesian
    def density_func(x):
        val = config.init_volume_fraction * bm.ones(x.shape
    [0], **kwargs)
        return val
    rho = space_D.interpolate(u=density_func)
```

```
    rho_opt, history = optimizer.optimize(rho=rho[:])

    from soptx.opt import save_optimization_history,
plot_optimization_history
    save_optimization_history(mesh, history)
    plot_optimization_history(history)
```

Figure 4 shows the convergence histories of the compliance $c(\rho)$ and the volume fraction $v(\rho)$ for an initial density of 0.4. Compliance decreases rapidly from its initial value of approximately 500 to around 100 in the first 10 iterations, converging by iteration 57. The volume fraction remains stably around 0.4, with only minor fluctuations.

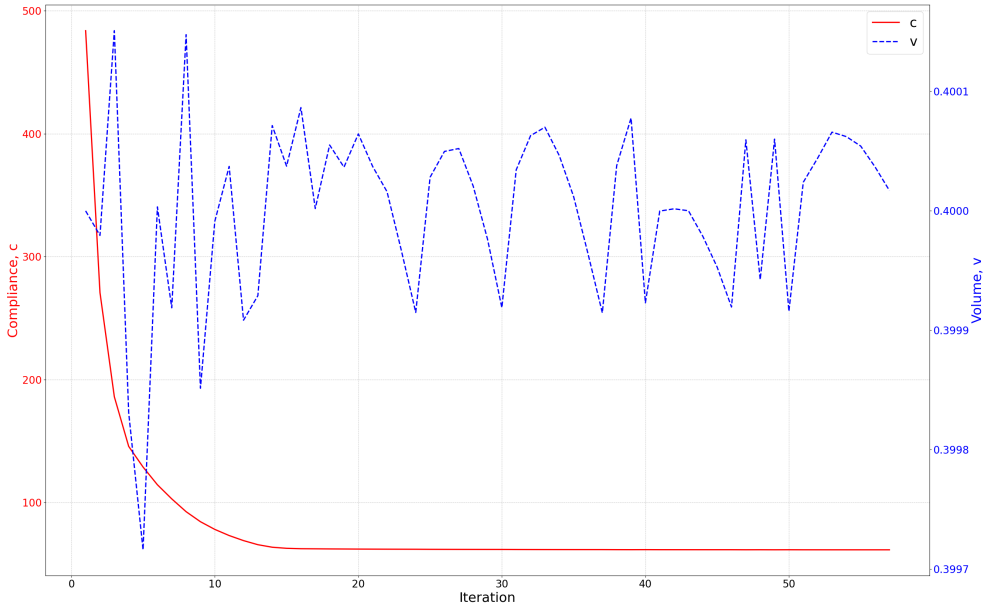


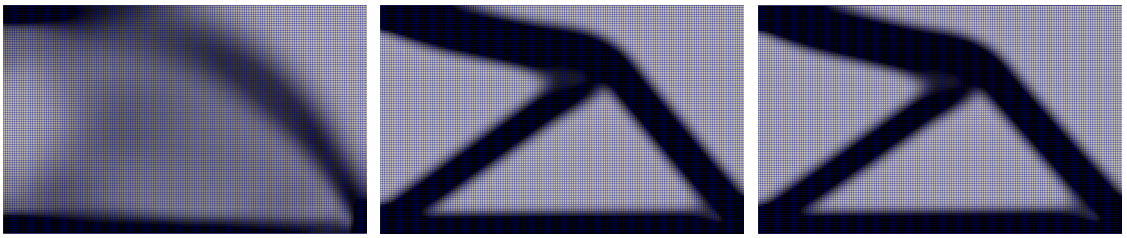
Figure 4: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam initialized with a uniform density of 0.4.

Figure 5 displays the resulting topologies at iterations 3, 30, and 57.

Figure 6 illustrates the convergence behavior for an initial density of 1. The volume fraction decreases from 1 to 0.4 within 5 iterations, while compliance initially increases to around 500 before decreasing to convergence at iteration 60. This process requires slightly more iterations due to the initial adjustment to meet the volume constraint.

Figure 7 shows the topology layouts at iterations 6, 33, and 60.

These results highlight the robustness of SOPTX in handling different initial conditions and its efficiency in achieving convergence with the OC algorithm.



$c(\rho) = 145.5753$, $v(\rho) = 0.3998$ $c(\rho) = 61.7084$, $v(\rho) = 0.3999$ $c(\rho) = 61.4208$, $v(\rho) = 0.4000$

Figure 5: Topology layouts at iterations 3 (left), 30 (middle), and 57 (right) during the optimization process. Each subfigure reports the corresponding compliance and volume fraction values.

5 Numerical Examples

This section presents numerical examples to validate the SOPTX framework's capabilities in topology optimization (TO), spanning 2D MBB beam problems to 3D cantilever structures. These examples demonstrate SOPTX's versatility across diverse partial differential equation (PDE) models, meshes, filters, and algorithms, while showcasing advanced features like fast matrix assembly, AD, and multi-backend switching. All runs are performed on a desktop computer running Ubuntu 24.04, using a CPU AMD Ryzen 9 9950X @ 4.5GHz, 2 × 32 GB of RAM, and an NVIDIA GeForce RTX 5070Ti GPU.

The section is organized as follows: Section 5.1 tests robustness across mesh resolutions; Section 5.2 explores filtering effects; Section 5.3 introduces algorithm switching and an updated MMA optimizer; Section 5.4 extends to 3D TO; Section 5.5 quantifies matrix assembly gains; Section 5.6 highlights AD in sensitivity analysis; and Section 5.7 assesses multi-backend and graphics processing units (GPUs) acceleration benefits. These examples underscore SOPTX's strengths and its potential in research and engineering.

5.1 MBB Beam

As introduced, we first validate SOPTX using the MBB beam, a classical benchmark in TO widely used to assess optimization algorithms. This section minimizes the structural compliance of the MBB beam (see Figure 8), demonstrating SOPTX's adaptability to various PDE models and meshes. The beam is hinged at the left edge and bottom right corner, with horizontal displacements constrained, and a downward load $T = -1$ applied at the upper left corner. The target volume fraction is 0.5, with material properties $E = 1$ and $\nu = 0.3$, penalization factor $p = 3$, and filter radius $r = 6.0$ to ensure smooth topology and suppress checkerboard patterns.

Compared to the cantilever beam in Section 4.2, SOPTX enables a seamless switch to the MBB beam problem. Users can modify the PDE model as follows:

```
from soptx.pde import MBBBeam2dData1
pde = MBBBeam2dData1(xmin=0, xmax=150, ymin=0, ymax=50, T=-1)
```

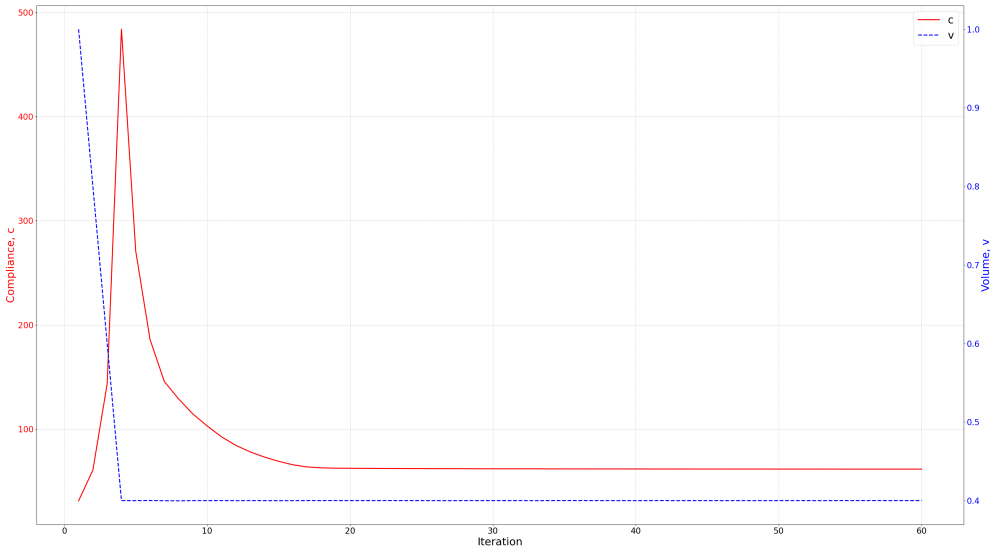


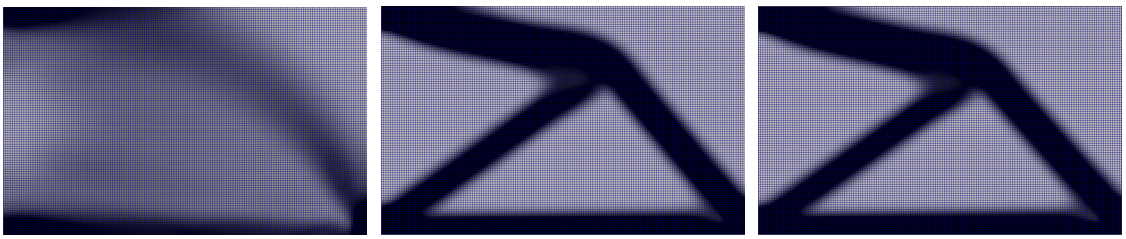
Figure 6: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam with an initial density of 1.

The complete model definition is provided in Appendix Appendix B.

To demonstrate SOPTX's adaptability across mesh types, we perform TO on a 150×50 uniform quadrilateral mesh and a triangular mesh, both initialized with a uniform material density equal to the target volume fraction 0.5. The triangular mesh is generated using:

```
from fealpy.mesh import TriangleMesh
mesh = TriangleMesh.from_box([0, 150, 0, 50], nx=150, ny=50)
```

The optimized topologies, shown in Figure 9, are consistent across both mesh types, with

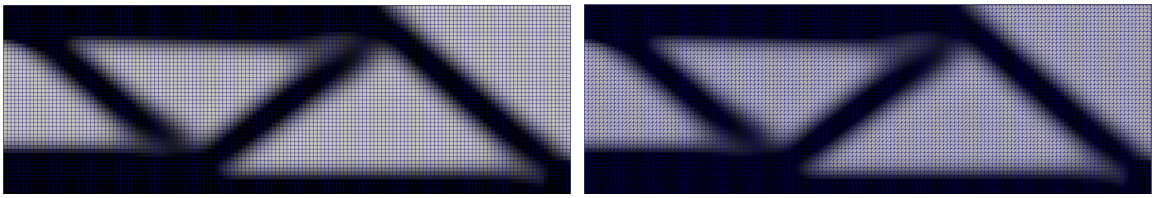


$c(\rho) = 185.9530$, $v(\rho) = 0.4002$ | $c(\rho) = 61.7084$, $v(\rho) = 0.3999$ | $c(\rho) = 61.4208$, $v(\rho) = 0.4000$

Figure 7: Topology layouts at iterations 6 (left), 33 (middle), and 60 (right) during the optimization process. Each subfigure includes the compliance and volume fraction values.

Figure 8: Geometry of the MBB beam: hinged at the left edge and bottom right corner, with a downward concentrated load applied at the top left.

compliance and volume fraction differences below 1%. This consistency underscores SOPTX's mesh-independence and algorithmic robustness, facilitated by its modular design, which allows easy switching between PDE models and mesh configurations.



$c(\rho) = 219.5199$, $v(\rho) = 0.5001$ $\quad$ $c(\rho) = 217.6605$, $v(\rho) = 0.4999$

Figure 9: Optimized topologies of the MBB beam using a uniform quadrilateral mesh (left) and a triangular mesh (right).

5.2 Different Filtering Methods

In TO, filtering methods are essential for smoothing design variables, controlling structural details, and ensuring the quality and manufacturability of the optimized results. Thanks to its modular architecture, SOPTX allows users to seamlessly switch between different filtering strategies by simply replacing the filter class, without modifying other components of the framework.

This section compares the results of two common filters (the density filter and the Heaviside projection filter) applied to the MBB beam problem from Section 5.1. All parameters are kept identical except for the filter choice, with the default filter radius set to $r = 6.0$.

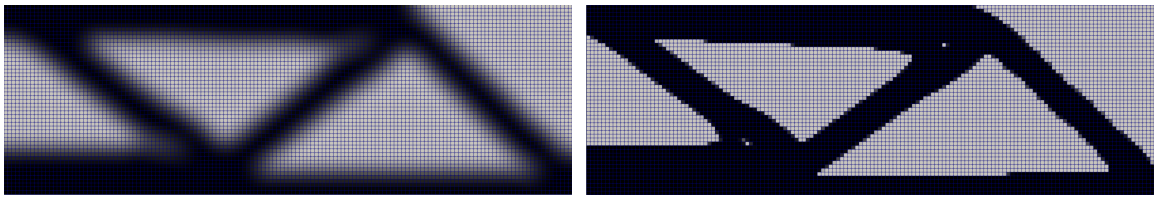
The density filter smooths the design variables through weighted averaging, eliminating small-scale features and producing a gradual material transition. This is ideal for designs requiring structural continuity. In SOPTX, it is applied via:

```
dens_filter = DensityBasicFilter(mesh=mesh, rmin=6.0)
```

In contrast, the Heaviside projection filter builds on the density filter by adding a projection step. It gradually increases the projection parameter β to drive variables toward 0 or 1, yielding a clear black-and-white topology suited for strict manufacturability. To avoid overly thick structures early on, the filter radius is reduced to $r = 4.5$. It is enabled in SOPTX with:

```
heavi_filter = HeavisideProjectionBasicFilter(mesh=mesh, rmin
    =4.5, beta=1, max_beta=512, continuation_iter=50)
```

The optimized topologies are shown in Figure 10. The density filter yields a smoother design with blurred edges, while the Heaviside projection filter produces a crisp, binarized layout with lower compliance (191.4873 vs. 235.7337), indicating a stiffer structure. However, with the Heaviside projection filter, small holes may still appear due to its focus on global binarization rather than strict local control.



$c(\rho) = 235.7337$, $v(\rho) = 0.5000$ $c(\rho) = 191.4873$, $v(\rho) = 0.5000$

Figure 10: Optimized topologies of the MBB beam using the density filter (left) and the Heaviside projection filter (right).

In summary, SOPTX's flexibility allows efficient exploration of filter impacts on design smoothness, manufacturability, and performance.

5.3 Different Optimization Algorithms

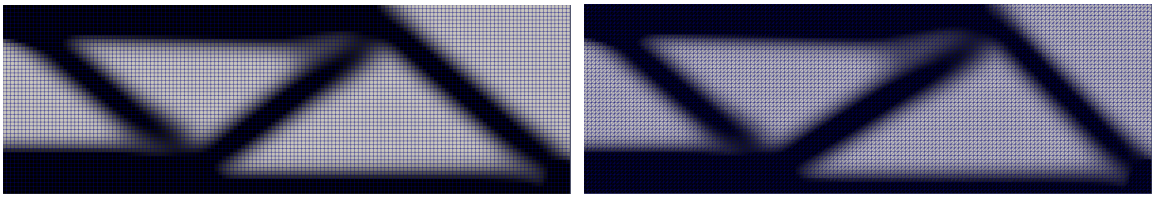
In TO, the choice of optimization algorithm impacts both computational efficiency and design quality. SOPTX's modular design allows seamless switching between optimizers, such as from the Optimality Criteria (OC) method to the more versatile MMA. This section demonstrates this process using the MBB beam problem and highlights the re-implemented MMA in SOPTX.

To switch to MMA, users simply import and configure the `MMAOptimizer` class:

```
from soptx.opt import MMAOptimizer
```

```
optimizer = MMAOptimizer(objective=objective,
        constraint=constraint, filter_=sens_filter,
        options={'max_iterations': 200, 'tolerance': 0.01})
```

For the MBB beam problem, MMA produces topologies nearly identical to OC's, with compliance and volume fraction differences below 1%, as shown in Figure 11. This consistency confirms MMA's reliability for single-constraint problems.



$c(\rho) = 219.4545$, $v(\rho) = 0.5000$ $c(\rho) = 220.4377$, $v(\rho) = 0.4995$

Figure 11: Optimized topologies of the MBB beam using MMA optimizer on a structured quadrilateral mesh (left) and a triangular mesh (right).

SOPTX's MMA, developed based on Krister Svanberg's implementation [35], allows users to adjust internal parameters, including `m`, `n`, `xmin`, `xmax`, and control parameters like `a0`, `a`, `c`, `d`. This provides greater flexibility than traditional "black-box" versions, enhancing performance and adaptability across platforms. An example configuration is shown below:

```
optimizer.options.set_advanced_options(m=1, n=NC,
            xmin=bm.zeros((NC, 1)), xmax=bm.ones((NC, 1)),
            a0=1, a=bm.zeros((1, 1)),
            c=1e4 * bm.ones((1, 1)), d=bm.zeros((1, 1)))
```

Through this re-implementation, SOPTX not only enhances the performance of the MMA algorithm but also overcomes the limitations of conventional implementations. It provides users with greater control and cross-platform flexibility, making it particularly advantageous for both research and engineering applications in TO.

5.4 Additional 2D Benchmark Problems

While the MBB beam example presented earlier serves as an excellent benchmark for validating the performance and stability of optimization algorithms, SOPTX is designed as a general-purpose framework capable of flexibly handling diverse boundary conditions and loading scenarios. To further demonstrate the applicability of SOPTX across a broader range of 2D engineering contexts, this section introduces two additional classic benchmark problems: the bridge structure

and the half-wheel structure. These examples demonstrate the extensibility of the SOPTX framework. Users can seamlessly adapt to new physical problems by simply inheriting and defining new PDE data classes, without modifying the core solver or optimization modules.

2D Simple Bridge Structure

Consider the bridge structure design problem shown in Figure 12. In this example, the design domain is defined as a 60×30 rectangular region, discretized by a uniform quadrilateral mesh. The boundary conditions consist of fully fixed supports at the two bottom corners. A downward concentrated load $T = -1$ is applied at the midpoint of the bottom edge. The target volume fraction is set to 0.3, and the sensitivity filter radius is set to $r = 2.4$ (chosen to match the mesh element size, ensuring structural smoothness and suppressing checkerboard patterns).

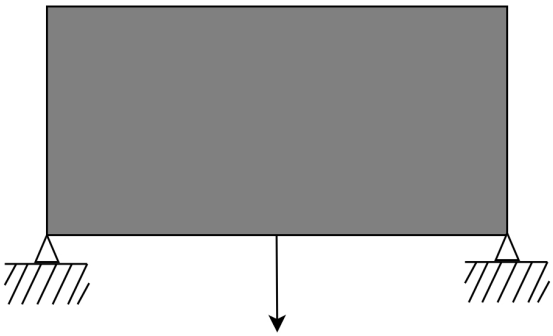


Figure 12: Geometric configuration of the 2D simple bridge structure. The bottom two corners are fully fixed, and a downward concentrated load is applied at the midpoint of the bottom edge.

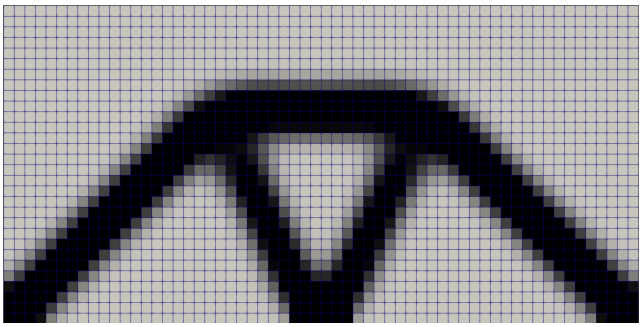
Compared to the cantilever beam example in Section 4.2, adapting SOPTX to this simple bridge problem requires only a simple substitution of the PDE model. The core definition of the `SimpleBridge2dData1` model is as follows(see Appendix Appendix D for the complete implementation):

```
from soptx.pde import SimpleBridge2dData1
pde = SimpleBridge2dData1(xmin=0, xmax=60, ymin=0, ymax=30, T
    =-1)
```

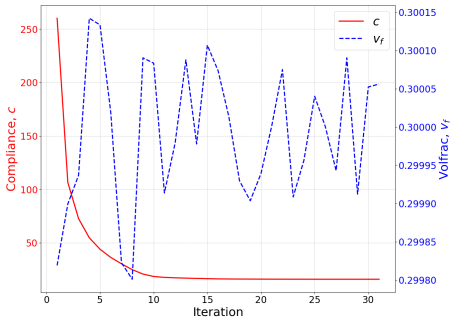
The optimized topology and convergence history are presented in Figure 13. As shown in Figure 13a, the material distribution automatically evolves into a distinct arch-like structure. This configuration efficiently transfers the central downward load to the supports via axial compression. Furthermore, the convergence history in Figure 13b indicates that the structural compliance decreases rapidly within the first 10 iterations and stabilizes around iteration 30, with the volume constraint strictly satisfied throughout the process."

2D Half-Wheel Structure

Following the bridge example, we consider the half-wheel benchmark problem (also known as



(a) Optimized arch-like topology



(b) Convergence of compliance $c(\rho)$ and volume fraction $v(\rho)$

Figure 13: Optimization results for the 2D simple bridge problem. (a) The final material distribution evolves into a clear arch structure. (b) Convergence history showing rapid decrease in compliance and strict satisfaction of the volume constraint.

a variant of the Michell structure) shown in Figure 14. The aspect ratio of the geometry remains consistent with the bridge example (2 : 1). However, to capture a clearer spoke-like structure, the design domain is enlarged to 120×60 and discretized by a corresponding 120×60 quadrilateral mesh. The loading condition remains unchanged, with a downward force applied at the midpoint of the bottom edge. The key difference lies in the boundary conditions: the bottom-left corner is fully fixed, while the bottom-right corner is supported by a roller, constraining only the vertical displacement and allowing horizontal sliding. The target volume fraction is set to 0.3, and the sensitivity filter radius is set to $r = 1.5$.

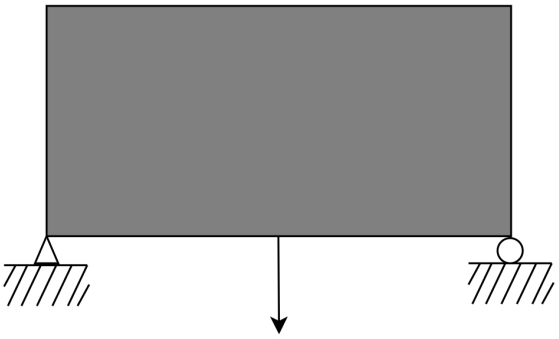


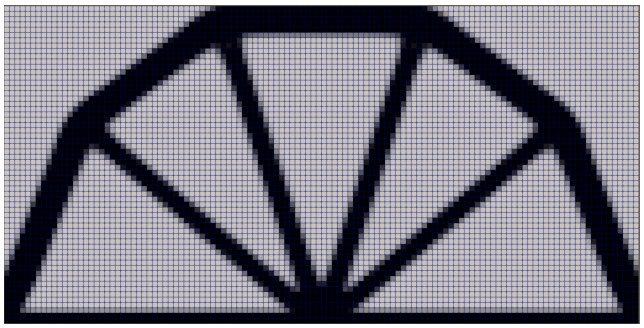
Figure 14: Geometric configuration of the 2D half-wheel structure. The bottom-left corner is fully fixed, while the bottom-right corner is supported by a roller. A downward concentrated load is applied at the midpoint of the bottom edge.

Similarly, adapting SOPTX to the half-wheel problem requires only a simple substitution of the PDE model. The core definition of the `HalfWheel2dData1` model is as follows(see Ap-

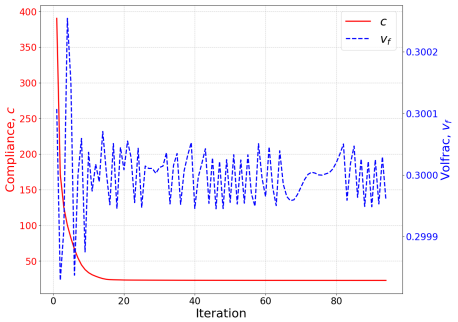
pendix Appendix C for the complete implementation):

```
from soptx.pde import HalfWheel2dData1
pde = HalfWheel2dData1(xmin=0, xmax=120, ymin=0, ymax=60, T
    =-1)
```

The optimization results are presented in Figure 15. As shown in Figure 15a, SOPTX successfully captures a complex topology resembling bicycle spokes, which is a typical characteristic of the theoretical optimal configuration (Michell-like structure) under these boundary conditions.The convergence history is plotted in Figure 15b. The structural compliance decreases rapidly within the first 20 iterations and stabilizes. It is worth noting that although the volume fraction curve exhibits high-frequency oscillations visually, the scale of the right y-axis reveals that the fluctuation amplitude is confined within the order of 10^{-4} (less than 0.03% error relative to the target). This indicates that the volume constraint is strictly satisfied throughout the optimization process.



(a) Optimized arch-like topology



(b) Convergence of compliance $c(\rho)$ and volume fraction $v(\rho)$

Figure 15: Optimization results for the 2D half-wheel structure. (a) The final material distribution evolves into a clear spoke-like structure. (b) Convergence history showing rapid decrease in compliance and strict satisfaction of the volume constraint..

5.5 3D Extension

SOPTX's modular design enables a smooth transition from 2D to 3D TO. This section showcases its capability with a 3D cantilever beam compliance minimization problem (see Figure 16). The beam is fixed on the left and bears a downward load $T = -1$ at the bottom right. A $60 \times 20 \times 4$ hexahedral mesh is applied, with a target volume fraction of 0.3, material properties $E = 1$ and $\nu = 0.3$, and penalization factor $p = 3$. A sensitivity filter (radius $r = 1.5$) ensures smooth results.

Compared to the 2D cantilever beam in Section 4.2, SOPTX extends to 3D by simply replacing the PDE model and mesh configuration. The 3D cantilever beam model is implemented as follows:

```
from soptx.pde import Cantilever3dData1
```

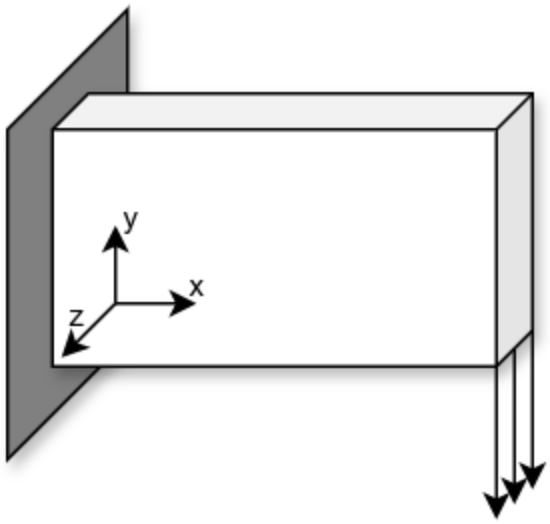


Figure 16: Geometric configuration of the 3D cantilever beam. The left end is fully fixed, and a downward concentrated load is applied at the bottom of the right end.

```
pde = Cantilever3dData1(xmin=0, xmax=60, ymin=0, ymax=20,
    zmin=0, zmax=4, T=-1)
mesh = UniformMesh(extent=[0, 60, 0, 20, 0, 4], h=[1, 1, 1],
    origin=[0, 0, 0])
```

The complete definition of the `Cantilever3dData1` model is available in Appendix Appendix E.

The initial material density is uniformly set to the target volume fraction of 0.3. Using the OC optimizer and a sensitivity filter with radius $r = 1.5$, the compliance $c(\rho)$ shows a rapid initial decrease, followed by a gradual convergence with slight fluctuations, stabilizing at around 2,000 by iteration 54 (see Figure 17). The volume fraction remains nearly constant around 0.3 throughout the process, indicating effective constraint control.

To better visualize the optimization results, Figure 18 shows the topology layouts at iterations 7, 21, and 54. Only elements with density values $\rho > 0.3$ are rendered to highlight the structural features.

3D TO increases computational complexity significantly. SOPTX addresses this with efficient matrix assembly and multi-backend support, enhancing performance for large-scale problems. Details follow in subsequent sections

5.6 Application of Fast Matrix Assembly

In 3D TO problems, computational efficiency is critical. For the 3D cantilever beam example (Section 5.5), the mesh of $60 \times 20 \times 4$ elements leads to 19,215 displacement degrees of freedom

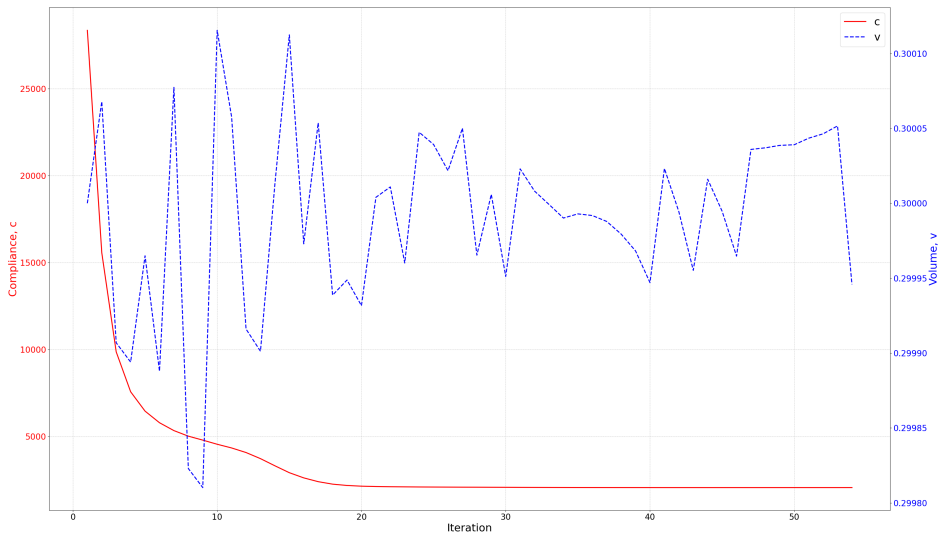
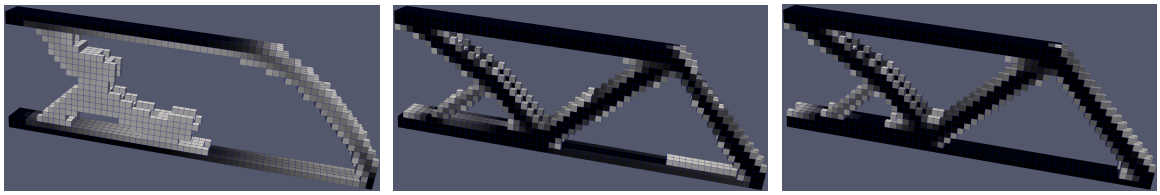


Figure 17: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam initialized with a uniform density of 0.3.

and 4,800 density variables, requiring 54 iterations. Traditional matrix assembly in finite element methods involves redundant computations, creating a bottleneck.

SOPTX addresses this with a fast matrix assembly technique that separates element-dependent and element-independent parts, reusing invariant quantities to accelerate the process. Additionally, symbolic integration precomputation analytically integrates invariant terms in advance, improving accuracy and stability.

- **Fast assembly**: Accelerates matrix assembly by separating element-dependent and element-independent terms, coupled with efficient numerical integration.



$c(\rho) = 5338.9462$, $v(\rho) = 0.3001$ $c(\rho) = 2122.6707$, $v(\rho) = 0.3000$ $c(\rho) = 2063.5625$, $v(\rho) = 0.2999$

Figure 18: Topology layouts at iterations 7 (left), 21 (middle), and 54 (right) during the optimization of the 3D cantilever beam. Elements with ($\rho > 0.3$) are visualized. Each subfigure also reports the compliance and volume fraction.

- **Symbolic Fast Assembly**: Further improves accuracy and stability by analytically integrating invariant terms ahead of time.

Users can easily switch between assembly methods by setting the `assembly_method` parameter in the solver initialization, as shown below:

```
# Fast Assembly
solver = ElasticFEMSolver(..., assembly_method=AssemblyMethod.FAST, ...)
# Symbolic Fast Assembly
solver = ElasticFEMSolver(..., assembly_method=AssemblyMethod.SYMBOLIC, ...)
```

To evaluate the efficiency of different matrix assembly techniques in SOPTX, we use the same 3D cantilever beam optimization problem from Section 5.5 as a representative test case. Table 1 presents a performance comparison of these techniques

Table 1: Performance comparison of matrix assembly techniques in the 3D cantilever beam optimization problem. All values are reported in seconds.

Assembly Technique	Total	1st Iter.	1st Assembly	Avg. Iter.	Avg. Assembly
Original Assembly	68.605	3.342	1.197	1.231	0.838
Fast Assembly	39.826	2.387	0.342	0.706	0.276
Symbolic Fast Assembly	41.194	5.877	3.853	0.666	0.272

As shown in Table 1, the original assembly method exhibits an average assembly time of 0.838 s per iteration, accounting for 68% of the total iteration time and thus creating a significant bottleneck. The fast assembly technique substantially improves this, reducing the average assembly time to 0.276 s, which is only 33% of the original assembly time. The symbolic fast assembly technique, while incurring a longer first iteration time (3.853 s for the first assembly), achieves the fastest average assembly time of 0.272 s with enhanced accuracy. This performance difference arises because symbolic fast assembly requires precomputing symbolic integrals during the first iteration, which increases the initial iteration time but allows the reuse of intermediate results in subsequent iterations, thereby significantly reducing assembly time in later stages.

In summary, SOPTX's fast assembly strategies significantly reduce computational cost and improve accuracy, making it ideal for large-scale TO problems.

5.7 Application of Automatic Differentiation

Sensitivity analysis is vital in TO to update design variables. Traditional methods rely on manual sensitivity derivation, which is time-consuming and error-prone, especially for complex models. The SOPTX framework introduces AD to streamline this process, enabling users to prioritize

problem modeling over mathematical derivations. Here, we demonstrate AD's application using the 3D cantilever beam example from Section 5.5.

The SOPTX framework supports multiple computational backends(e.g., NumPy, PyTorch, JAX). To enable AD, users must switch from the default NumPy backend to an AD-enabled backend like PyTorch [26] with a single line of code:

```
bm.set_backend('pytorch')
```

Sensitivity computation is controlled via the `diff_mode` parameter: `'manual'` for hand-derived formulas and `'auto'` for AD. For example, AD can be enabled for the compliance objective while keeping manual differentiation for the volume constraint:

```
obj_config = ComplianceConfig(diff_mode='auto')
objective = ComplianceObjective(solver=solver, config=
    obj_config)
cons_config = VolumeConfig(diff_mode='manual')
constraint = VolumeConstraint(solver=solver, volume_fraction
    =0.5, config=cons_config)
```

This setup allows selective use of AD or manual differentiation. Further details on AD implementation are in Section Appendix F.

To verify the correctness and effectiveness of AD, we conduct tests using the 3D cantilever beam problem described in Section 5.5. All parameters are kept identical. The structural compliance sensitivity is computed using AD. Figure 19 shows the convergence histories of the compliance and volume fraction.

To further illustrate the evolution of the topology during optimization, the layouts at selected iterations are shown in Figure 20.

As observed from Figure 19 and Figure 20, the optimization results obtained using AD are fully consistent with those achieved through manual differentiation in Section 5.5. The compliance converges to approximately 2063.5625 after 54 iterations, and the volume fraction remains stably around 0.3. These results confirm that AD accurately computes the sensitivities and effectively guides the optimization process.

To evaluate the computational efficiency of AD, we compare the optimization times of AD and manual differentiation under the same 3D cantilever beam problem setting. As shown in Table 2, the total optimization time and average iteration time for both methods are nearly identical. Since manual differentiation involves the direct execution of analytical formulas and typically serves as the performance benchmark/upper bound [10,23,25], matching this performance demonstrates that SOPTX's AD implementation is highly efficient. It introduces negligible computational overhead, thereby allowing users to leverage the modeling flexibility and development speed of AD without incurring the performance penalties often observed in other frameworks.

In addition to PyTorch, SOPTX also supports the JAX backend [8], which offers equally powerful AD capabilities. The sensitivity computation code remains identical when using JAX, users

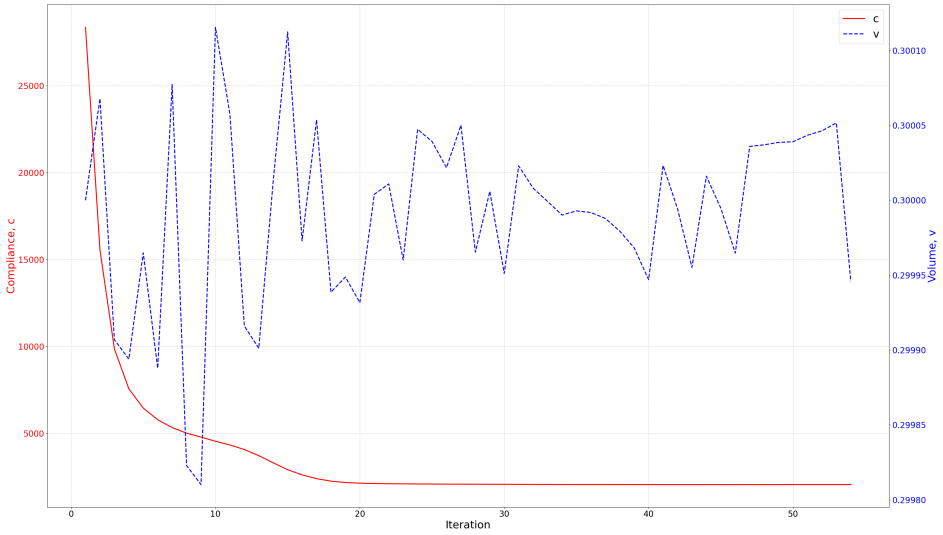


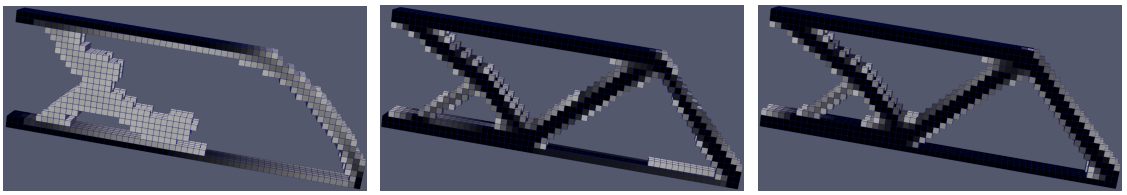
Figure 19: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam optimization using automatic differentiation.

simply need to switch to the JAX backend by executing:

```
bm.set_backend('jax')
```

Using AD with the JAX backend, we perform TO for the 3D cantilever beam, achieving results fully consistent with those from the PyTorch backend. This consistency underscores the stability and flexibility of the SOPTX framework across different computational backends. The final optimized topology is shown in Figure 21.

The application of AD in TO offers several significant advantages:



$c(\rho) = 5338.9462$, $v(\rho) = 0.3001$ $c(\rho) = 2122.6707$, $v(\rho) = 0.3000$ $c(\rho) = 2063.5625$, $v(\rho) = 0.2999$

Figure 20: Topology layouts at iterations 7 (left), 21 (middle), and 54 (right) during the 3D cantilever beam optimization using AD. Only elements with $\rho > 0.3$ are visualized. Each subfigure also reports the compliance and volume fraction.

Table 2: Performance comparison of differentiation methods in the 3D cantilever beam optimization problem.

Differentiation Method	**Iterations**	**Total Time (s)**	**1st Iter. Time (s)**	**Avg. Iter. Time (s)**
Manual Differentiation	54	39.562	1.940	0.710
AD	54	39.865	1.832	0.718

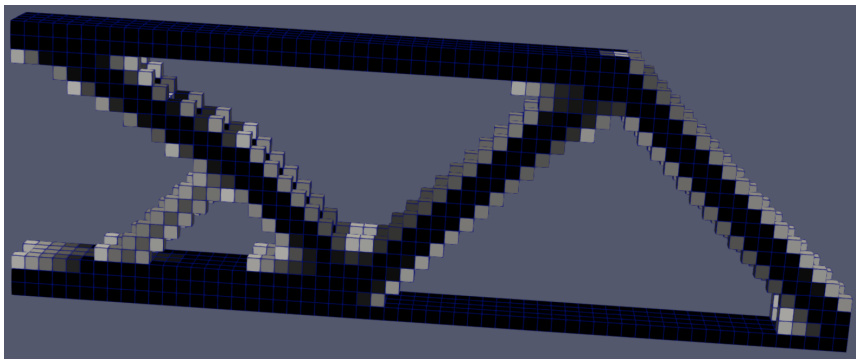


Figure 21: Final optimized topology of the 3D cantilever beam using AD with the JAX backend.

- Simplified model switching: AD automates gradient computations for material interpolation models like SIMP and Rational Approximation of Material Properties (RAMP) [33], enabling seamless switching without manual derivation and improving development efficiency.

- Seamless constraint switching: AD simplifies sensitivity analysis for constraints such as volume, length scale [16], connectivity [20], overhangs [27], and material usage [28], enhancing optimization adaptability.

- Support for complex problems: AD efficiently computes gradients in multi-physics or geometrically constrained TO, automatically handling sensitivity computations for techniques like density filtering and Heaviside projection, allowing users to focus on modeling rather than derivations.

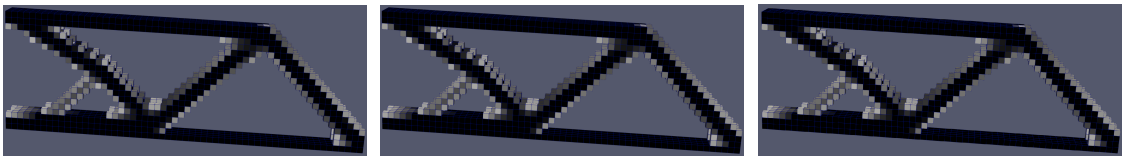
This section highlights AD's potential in SOPTX, delivering precision and efficiency comparable to manual differentiation without added overhead. SOPTX's multi-backend support enhances its flexibility, while AD simplifies sensitivity analysis and enables exploration of new models and constraints, making SOPTX a valuable tool for topology optimization in education, research, and engineering.

5.8 Multi-Backend Switching

SOPTX supports multiple computational backends, including NumPy, PyTorch, and JAX. Users can flexibly switch between them using the `set_backend` function.

```
bm.set_backend('numpy')
bm.set_backend('pytorch')
bm.set_backend('jax')
```

To validate its consistency and reliability across backends, we conducted tests on the 3D cantilever beam optimization problem described in Sections 5.5 and 5.7, using identical parameter settings. The problem was solved independently under each backend, and the results, shown in Figure 22, demonstrate high consistency: compliance converges to 2063.5625, the volume fraction stabilizes at 0.3, and the topology layouts are almost identical. This confirms SOPTX's stability and robustness across different computational backends.



$c(\rho) = 2063.5625$, $v(\rho) = 0.2999$ $c(\rho) = 2063.5625$, $v(\rho) = 0.2999$ $c(\rho) = 2063.5625$, $v(\rho) = 0.2999$

Figure 22: Final optimized topologies of the 3D cantilever beam using three different backends (elements with $\rho > 0.3$ are visualized). Left: NumPy backend; Middle: PyTorch backend; Right: JAX backend.

In addition to supporting multi-backend switching, SOPTX enables computations on GPU to enhance efficiency, particularly for large-scale 3D TO problems. By default, computations are performed on the CPU (`device='cpu'`). Users can migrate computations to a GPU (e.g., CUDA devices) in two ways:

1. Set the default device globally using the following code:

   ```
   bm.set_default_device('cuda')
   ```

   This method is suitable for scenarios where all computations are to be executed on the GPU.

2. Specify the device during mesh creation, as shown below:

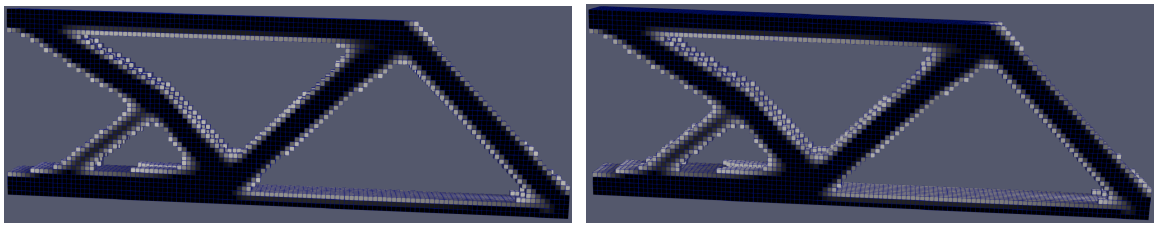
   ```
   mesh = UniformMesh(extent=extent, h=h, origin=origin,
       device='cuda')
   ```

This approach offers greater flexibility, allowing users to mix CPU and GPU computations within the same program.

GPU acceleration is essential for 3D TO due to the significant increase in computational complexity as mesh size grows. For instance, in the 3D cantilever beam example from Section 5.5, doubling the mesh to $120\times40\times8$ increases the density design variables to 38,400 and displacement degrees of freedom to 133,947. On the CPU, this leads to prolonged optimization times due to matrix assembly and linear solving costs. In contrast, GPU's parallel computing capabilities greatly enhance efficiency and reduce computation time.

To demonstrate this, we compare CPU and GPU computation times under the PyTorch backend for the same problem. Parameters remain consistent with Section 5.5, except for increasing the filter radius from 1.5 to 3.0 to maintain structural smoothness in the larger domain. Additionally, we use the CG method instead of a direct solver (e.g., MUMPS) to handle the larger sparse linear systems, leveraging the GPU's parallel processing for efficient matrix-vector operations.

As shown in Figure 23, the optimized topologies using CPU and GPU are nearly identical. Minor differences in compliance and volume fraction, both within 1%, arise from variations in floating-point precision and parallel implementation strategies across hardware platforms. These differences are negligible in TO and do not impact the quality or performance of the final design.



$c(\rho)=3498.8266$, $v(\rho)=0.3001$ $\quad$ $c(\rho)=3494.2391$, $v(\rho)=0.3000$

Figure 23: Optimized topologies of the 3D cantilever beam using CPU (left) and GPU (right), with elements of density $\rho>0.3$ visualized.

Given the consistency in optimization results, we evaluate the computational efficiency of CPU versus GPU, as detailed in Table 3. The GPU acceleration markedly reduces the total optimization time from 3872.041 s (CPU) to 479.412 s (GPU), achieving an approximately speedup of 8.1 times. This superior performance is evident in both initial and subsequent iterations, highlighting the GPU's parallel computing advantages.

Notably, CG method exhibits faster convergence in early iterations due to the uniform material density, which results in a well-conditioned stiffness matrix. As optimization advances, increased heterogeneity in material distribution elevates the matrix condition number, thereby prolonging CG convergence and iteration times.

Table 3: Performance comparison between CPU and GPU for the 3D cantilever beam optimization problem.

Computational Device	Iterations	Total Time (s)	1st Iter. Time (s)	Avg. Iter. Time (s)
CPU (PyTorch)	155	3872.041	15.846	25.040
GPU (PyTorch)	155	479.412	2.111	3.099

6 Conclusion

This study presents SOPTX, a modular and extensible topology optimization (TO) framework built on the open-source FEALPy platform. SOPTX addresses key challenges in TO by offering a modular, efficient, and accessible solution for both academic and industrial applications. Its fully open-source design eliminates reliance on commercial software, broadening its accessibility for research and education.

The core innovations of SOPTX are reflected in the following three aspects:

1. **Modular Architecture**: A loosely coupled design supports 2D and 3D TO problems across structured and unstructured meshes. This flexibility, paired with a rich set of configurable components, empowers users to tailor and extend optimization workflows efficiently.

2. **Multi-Backend Support and Automatic differentiation**: Integration of NumPy, PyTorch, and JAX enables fast central processing units (CPUs) and graphics processing units (GPUs) computation. In benchmark tests, GPU computations reduced time to approximately 12% of CPU time. AD automates gradient calculations, enhancing accuracy.

3. **Efficient Matrix Assembly**: By optimizing assembly processes and exploiting sparsity, SOPTX reduces average assembly time from 0.838 s to 0.273 s (approximately 33% of the original), significantly boosting efficiency.

The experimental analyses in Sections 5.1 through 5.8 demonstrate SOPTX's exceptional performance across key areas: seamless model and filter switching, flexible optimization algorithms, 3D problem extension, enhanced computational efficiency, automatic differentiation, and multi-backend support. These results underscore SOPTX's technical strengths and versatility.

Looking forward, SOPTX is set to expand its capabilities:

- **Level Set Methods**: Integrating FEALPy's level set module will enable boundary-clear optimization, generating smooth, manufacturable designs for precision applications.

- **Adaptive Mesh Refinement**: Incorporating local refinement near boundaries will improve accuracy and detail in complex geometries.

- **Multiphysics Optimization**: Leveraging FEALPy's solvers for heat conduction and Navier Stokes equations, SOPTX can address thermo-fluid-structure coupling, optimizing designs like aerospace thermal protection systems or electronic heat dissipation structures.

- **Manufacturing Constraints**: Future support for stress and additive manufacturing constraints (e.g., overhang control, support minimization) will enhance SOPTX's applicability in aerospace, automotive, and biomedical fields, aligning with lightweight and manufacturability goals.

These extensions will solidify SOPTX's role in advancing TO for both research and industry.

Acknowledgments

National Natural Science Foundation of China (NSFC) (Grant Nos. 12371410, 12261131501). The Construction of Innovative Provinces in Hunan Province (Grant No. 2021GK1010).

Appendix A Cantilever2dData1

```python
class Cantilever2dData1:
    def __init__(self,
                 xmin: float, xmax: float,
                 ymin: float, ymax: float,
                 T: float = -1):
    self.xmin, self.xmax = xmin, xmax
    self.ymin, self.ymax = ymin, ymax
    self.T = T
    self.eps = 1e-12

    def domain(self) -> list:
        box = [self.xmin, self.xmax, self.ymin, self.ymax]
        return box

    @cartesian
    def force(self, points: TensorLike) -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        y = points[..., 1]
        coord = (
            (bm.abs(x - domain[1]) < self.eps) &
            (bm.abs(y - domain[2]) < self.eps)
            )
        kwargs = bm.context(points)
        val = bm.zeros(points.shape, **kwargs)
        val[coord, 1] = self.T
        return val
```

```
28
29      @cartesian
30      def dirichlet(self, points: TensorLike) -> TensorLike:
31          kwargs = bm.context(points)
32          return bm.zeros(points.shape, **kwargs)
33
34      @cartesian
35      def is_dirichlet_boundary_dof_x(self, points: TensorLike)
        -> TensorLike:
36          domain = self.domain()
37          x = points[..., 0]
38          coord = bm.abs(x - domain[0]) < self.eps
39          return coord
40
41      @cartesian
42      def is_dirichlet_boundary_dof_y(self, points: TensorLike)
        -> TensorLike:
43          domain = self.domain()
44          x = points[..., 0]
45          coord = bm.abs(x - domain[0]) < self.eps
46          return coord
47
48      def threshold(self) -> Tuple[Callable, Callable]:
49          return (self.is_dirichlet_boundary_dof_x,
50                  self.is_dirichlet_boundary_dof_y)
```

Appendix B MBBBeam2dData1

```
1  class MBBBeam2dData1:
2      def __init__(self,
3                   xmin: float=0, xmax: float=60,
4                   ymin: float=0, ymax: float=20,
5                   T: float = -1):
6          self.xmin, self.xmax = xmin, xmax
7          self.ymin, self.ymax = ymin, ymax
8          self.T = T
9          self.eps = 1e-12
10
11     def domain(self) -> list:
```

```
        box = [self.xmin, self.xmax, self.ymin, self.ymax]
        return box

    @cartesian
    def force(self, points: TensorLike) -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        y = points[..., 1]
        coord = ((bm.abs(x - domain[0]) < self.eps) &
                 (bm.abs(y - domain[3]) < self.eps))
        kwargs = bm.context(points)
        val = bm.zeros(points.shape, **kwargs)
        val[coord, 1] = self.T
        return val

    @cartesian
    def dirichlet(self, points: TensorLike) -> TensorLike:
        kwargs = bm.context(points)
        return bm.zeros(points.shape, **kwargs)

    @cartesian
    def is_dirichlet_boundary_dof_x(self, points: TensorLike)
    -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        coord = bm.abs(x - domain[0]) < self.eps
        return coord

    @cartesian
    def is_dirichlet_boundary_dof_y(self, points: TensorLike)
    -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        y = points[..., 1]
        coord = ((bm.abs(x - domain[1]) < self.eps) &
                 (bm.abs(y - domain[0]) < self.eps))
        return coord

    def threshold(self) -> Tuple[Callable, Callable]:
        return (self.is_dirichlet_boundary_dof_x,
                self.is_dirichlet_boundary_dof_y)
```

Appendix C HalfWheel2dData1

```
class HalfWheel2dData1:
    def __init__(self,
                 xmin: float=0, xmax: float=120,
                 ymin: float=0, ymax: float=60,
                 T: float = -1) -> None:
        xmin: float=0, xmax: float=120,
        ymin: float=0, ymax: float=60,
        T: float = -1

    def domain(self) -> list:
        box = [self.xmin, self.xmax, self.ymin, self.ymax]
        return box

    @cartesian
    def force(self, points: TensorLike) -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        y = points[..., 1]
        on_bottom_boundary = bm.abs(y - domain[2]) < self.
    _eps
        mid_x = (domain[0] + domain[1]) / 2.0
        on_middle_boundary = bm.abs(x - mid_x) < self._eps
        coord = on_bottom_boundary & on_middle_boundary
        kwargs = bm.context(points)
        val = bm.zeros(points.shape, **kwargs)
        val[coord, 1] = self.T
        return val

    @cartesian
    def dirichlet(self, points: TensorLike) -> TensorLike:
        kwargs = bm.context(points)
        return bm.zeros(points.shape, **kwargs)

    @cartesian
    def is_dirichlet_boundary_dof_x(self, points: TensorLike)
    -> TensorLike:
    domain = self._domain
    x, y = points[..., 0], points[..., 1]
```

```
37      left_bottom = (bm.abs(x - domain[0]) < self._eps) & (bm.
    abs(y - domain[2]) < self._eps)
38
39      return left_bottom
40
41      @cartesian
42      def is_dirichlet_boundary_dof_y(self, points: 'TensorLike
    ') -> 'TensorLike':
43          domain = self._domain
44          x, y = points[..., 0], points[..., 1]
45          left_bottom = (bm.abs(x - domain[0]) < self._eps) & (
    bm.abs(y - domain[2]) < self._eps)
46          right_bottom = (bm.abs(x - domain[1]) < self._eps) &
    (bm.abs(y - domain[2]) < self._eps)
47          return left_bottom | right_bottom
48
49
50      def threshold(self) -> Tuple[Callable, Callable]:
51          return (self.is_dirichlet_boundary_dof_x,
52                  self.is_dirichlet_boundary_dof_y)
```

Appendix D SimpleBridge2dData1

```
1       class SimpleBridge2dData1:
2       def __init__(self,
3       xmin: float=0, xmax: float=60,
4       ymin: float=0, ymax: float=30,
5       T: float = -1) -> None:
6       xmin: float=0, xmax: float=60,
7       ymin: float=0, ymax: float=20,
8       T: float = -1
9
10      def domain(self) -> list:
11      box = [self.xmin, self.xmax, self.ymin, self.ymax]
12      return box
13
14      @cartesian
15      def force(self, points: TensorLike) -> TensorLike:
16      domain = self.domain()
```

```
    x = points[..., 0]
    y = points[..., 1]
    on_bottom_boundary = bm.abs(y - domain[2]) < self._eps
    mid_x = (domain[0] + domain[1]) / 2.0
    on_middle_boundary = bm.abs(x - mid_x) < self._eps
    coord = on_bottom_boundary & on_middle_boundary
    kwargs = bm.context(points)
    val = bm.zeros(points.shape, **kwargs)
    val[coord, 1] = self.T
    return val

    @cartesian
    def dirichlet(self, points: TensorLike) -> TensorLike:
    kwargs = bm.context(points)
    return bm.zeros(points.shape, **kwargs)

    @cartesian
    def is_dirichlet_boundary_dof_x(self, points: 'TensorLike
') -> 'TensorLike':
    domain = self._domain
    x, y = points[..., 0], points[..., 1]
    left_bottom = (bm.abs(x - domain[0]) < self._eps) & (bm.
abs(y - domain[2]) < self._eps)
    right_bottom = (bm.abs(x - domain[1]) < self._eps) & (bm.
abs(y - domain[2]) < self._eps)
    return left_bottom | right_bottom

    @cartesian
    def is_dirichlet_boundary_dof_y(self, points: 'TensorLike
') -> 'TensorLike':
    domain = self._domain
    x, y = points[..., 0], points[..., 1]
    left_bottom = (bm.abs(x - domain[0]) < self._eps) & (bm.
abs(y - domain[2]) < self._eps)
    right_bottom = (bm.abs(x - domain[1]) < self._eps) & (bm.
abs(y - domain[2]) < self._eps)
    return left_bottom | right_bottom


    def threshold(self) -> Tuple[Callable, Callable]:
    return (self.is_dirichlet_boundary_dof_x,
    self.is_dirichlet_boundary_dof_y)
```

Appendix E Cantilever3dData1

```python
class Cantilever3dData1:
    def __init__(self,
                 xmin: float=0, xmax: float=60,
                 ymin: float=0, ymax: float=20,
                 zmin: float=0, zmax: float=4,
                 T: float = -1):
        self.xmin, self.xmax = xmin, xmax
        self.ymin, self.ymax = ymin, ymax
        self.zmin, self.zmax = zmin, zmax
        self.T = T
        self.eps = 1e-12

    def domain(self) -> list:
        box = [self.xmin, self.xmax,
        self.ymin, self.ymax,
        self.zmin, self.zmax]
        return box

    @cartesian
    def force(self, points: TensorLike) -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        y = points[..., 1]
        z = points[..., 2]
        coord = ((bm.abs(x - domain[1]) < self.eps) &
                 (bm.abs(y - domain[2]) < self.eps))
        kwargs = bm.context(points)
        val = bm.zeros(points.shape, **kwargs)
        val[coord, 1] = self.T
        return val

    @cartesian
    def dirichlet(self, points: TensorLike) -> TensorLike:
        kwargs = bm.context(points)
        return bm.zeros(points.shape, **kwargs)

```

```
    @cartesian
    def is_dirichlet_boundary_dof_x(self, points: TensorLike)
    -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        coord = bm.abs(x - domain[0]) < self.eps
        return coord

    @cartesian
    def is_dirichlet_boundary_dof_y(self, points: TensorLike)
    -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        coord = bm.abs(x - domain[0]) < self.eps
        return coord

    @cartesian
    def is_dirichlet_boundary_dof_z(self, points: TensorLike)
    -> TensorLike:
        domain = self.domain()
        x = points[..., 0]
        coord = bm.abs(x - domain[0]) < self.eps
        return coord

    def threshold(self) -> Tuple[Callable, Callable]:
        return (self.is_dirichlet_boundary_dof_x,
                self.is_dirichlet_boundary_dof_y,
                self.is_dirichlet_boundary_dof_z)
```

Appendix F Automatic Differentiation

```
def _compute_gradient_auto(self, rho: TensorLike, u: Optional
    [TensorLike] = None) -> TensorLike:
    if u is None:
        u = self._update_u(rho)

    ke0 = self.solver.get_base_local_stiffness_matrix()
    cell2dof = self.solver.tensor_space.cell_to_dof()
    ue = u[cell2dof]
```

```
8
9       def compliance_contribution(rho_i: float, ue_i:
        TensorLike, ke0_i: TensorLike) -> float:
10          E = self.materials.calculate_elastic_modulus(rho_i)
11          c_i = -E * bm.einsum('i, ij, j', ue_i, ke0_i, ue_i)
12          return c_i
13
14      vmap_grad = bm.vmap(lambda r, u, k:bm.jacrev(lambda x:
15          compliance_contribution(x, u, k))(r))
16      dc = vmap_grad(rho, ue, ke0)
17      return dc
```

References

[1] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[2] E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, and O. Sigmund. Efficient topology optimization in matlab using 88 lines of code. *Structural and Multidisciplinary Optimization*, 43:1–16, 2011.

[3] M. P. Bendsøe. Optimal shape design as a material distribution problem. *Structural Optimization*, 1:193–202, 1989.

[4] M. P. Bendsøe. *Optimization of structural topology, shape, and material*, volume 414. Springer, 1995.

[5] M. P. Bendsøe and O. Sigmund. *Topology optimization by distribution of isotropic material*, pages 1–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[6] M. P. Bendsoe and O. Sigmund. *Topology optimization: theory, methods, and applications*. Springer Science & Business Media, 2013.

[7] B. Bourdin. Filters in topology optimization. *International Journal for Numerical Methods in Engineering*, 50(9):2143–2158, 2001.

[8] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[9] T. E. Bruns and D. A. Tortorelli. Topology optimization of non-linear elastic structures and compliant mechanisms. *Computer Methods in Applied Mechanics and Engineering*, 190(26-27):3443–3459, 2001.

[10] A. Chandrasekhar, S. Sridhara, and K. Suresh. Auto: a framework for automatic differentiation in topology optimization. *Structural and Multidisciplinary Optimization*, 64(6):4355–4365, 2021.

[11] H. Chung, J. T. Hwang, J. S. Gray, and H. A. Kim. Topology optimization in openmdao. *Structural and Multidisciplinary Optimization*, 59:1385–1400, 2019.

[12] R. M. Ferro and R. Pavanello. A simple and efficient structural topology optimization implementation using open-source software for all steps of the algorithm: Modeling, sensitivity analysis and optimization. *CMES-Computer Modeling in Engineering & Sciences*, 136(2), 2023.

[13] C. for Python Data API Standards. Array api standard, version 2023.12. https://data-apis.org/array-api/2023.12/, 2023. Accessed April 2025.

[14] J. S. Gray, J. T. Hwang, J. R. Martins, K. T. Moore, and B. A. Naylor. Openmdao: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, 2019.

[15] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[16] J. K. Guest. Imposing maximum length scale in topology optimization. *Structural and Multidisciplinary Optimization*, 37:463–473, 2009.

[17] J. K. Guest, J. H. Prévost, and T. Belytschko. Achieving minimum length scale in topology optimization using nodal design variables and projection functions. *International Journal for Numerical Methods in Engineering*, 61(2):238–254, 2004.

[18] A. Gupta, R. Chowdhury, A. Chakrabarti, and T. Rabczuk. A 55-line code for large-scale parallel topology optimization in 2d and 3d. *arXiv preprint arXiv:2012.08208*, 2020.

[19] J. Hou, J. Li, S. Zhu, X. Hu, and Z. Yu. Parallel computing on GPU with CuPy and vectorized SpMV for large-scale topology optimization. *Finite Elements in Analysis and Design*, 250:104388, 2025.

[20] Q. Li, W. Chen, S. Liu, and L. Tong. Structural topology optimization considering connectivity constraint. *Structural and Multidisciplinary Optimization*, 54:971–984, 2016.

[21] K. Liu and A. Tovar. An efficient 3d topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 50(6):1175–1196, 2014.

[22] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

[23] A. Neofytou, T. Rios, M. Bujny, S. Menzel, and H. A. Kim. Level set topology optimization with sparse automatic differentiation. *Structural and Multidisciplinary Optimization*, 67(10):178, 2024.

[24] S. A. Nørgaard, M. Sagebaum, N. R. Gauger, and B. S. Lazarov. Applications of automatic differentiation in topology optimization. *Structural and Multidisciplinary Optimization*, 56:1135–1146, 2017.

[25] S. A. Nørgaard, M. Sagebaum, N. R. Gauger, and B. S. Lazarov. Applications of automatic differentiation in topology optimization. *Structural and Multidisciplinary Optimization*, 56(5):1135–1146, 2017.

[26] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[27] X. Qian. Undercut and overhang angle control in topology optimization: a density gradient based integral approach. *International Journal for Numerical Methods in Engineering*, 111(3):247–272, 2017.

[28] E. D. Sanders, M. A. Aguiló, and G. H. Paulino. Multi-material continuum topology optimization with arbitrary volume and mass constraints. *Computer Methods in Applied Mechanics and Engineering*, 340:798–823, 2018.

[29] O. Sigmund. On the design of compliant mechanisms using topology optimization. *Journal of Structural Mechanics*, 25(4):493–524, 1997.

[30] O. Sigmund. A 99 line topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 21(2):120–127, 2001.

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

[32] O. Sigmund and J. Petersson. Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Structural Optimization*,

16:68–75, 1998.

[33] M. Stolpe and K. Svanberg. An alternative interpolation scheme for minimum compliance topology optimization. *Structural and Multidisciplinary Optimization*, 22(2):116–124, 2001.

[34] K. Svanberg. The method of moving asymptotes—a new method for structural optimization. *International Journal for Numerical Methods in Engineering*, 24(2):359–373, 1987.

[35] K. Svanberg. MMA and GCMMA, versions September 2007. Technical report, Optimization and Systems Theory, Department of Mathematics, KTH Royal Institute of Technology, Stockholm, Sweden, 2007.

[36] E. A. Träff. Simple and efficient GPU accelerated topology optimisation: Codes and applications. *Computer Methods in Applied Mechanics and Engineering*, 410:116043, 2023.

[37] H. Wei and Y. Huang. Fealpy: Finite element analysis library in python. https://github.com/weihuayi/fealpy, Xiangtan University, 2017-2024.

[38] M. Zhou and G. I. Rozvany. The coc algorithm, part ii: Topological, geometrical and generalized shape optimization. *Computer Methods in Applied Mechanics and Engineering*, 89(1-3):309–336, 1991.