

SOPTX: A High-Performance Multi-Backend Framework for Topology Optimization

Liang He¹, Huayi Wei^{2,*}, Tian Tian³, Wang Pengxiang⁴

¹ 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

² Department of Mathematics, University of California at Irvine, Irvine, CA 92697, USA

³ School of Mathematics, Shanghai University of Finance and Economics, Shanghai 200433, China

Abstract. Topology optimization (TO) has gained widespread attention but remains challenging due to tight coupling between structural analysis and optimization algorithms. This paper presents SOPTX, a high-performance, modular TO framework built on FEALPy, featuring a non-intrusive design and multi-backend support (NumPy, PyTorch, JAX) for efficient computation on central processing units (CPUs) and graphics processing units (GPUs), along with automatic differentiation (AD) for sensitivity analysis. Core innovations include: (1) cross-platform backend integration with AD, (2) fast matrix assembly techniques significantly accelerating finite element computations, and (3) a modular architecture enabling flexible configuration and extensibility of optimization workflows. Using compliance minimization problems as examples, numerical experiments demonstrate SOPTX's superior efficiency, especially in large-scale scenarios, and its strong potential for research, education, and engineering applications.

AMS subject classifications: 65N30, 35Q60

Key words: Topology Optimization, Multi-Backend Computing, 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 applied in aerospace, automotive, and civil engineering, TO addresses critical design challenges by efficiently utilizing materials. Among various approaches, density-based methods are particularly

*Corresponding author. Email addresses: 202331510117@smail.xtu.edu.cn (L. Liang), weihuayi@xtu.edu.cn (H. Wei), chenlong@math.uci.edu (T. Tian), huang.xuehai@sufe.edu.cn (P. Wang)

popular due to their intuitive concept and practicality. The Solid Isotropic Material with Penalization (SIMP) method, introduced by Bendsøe et al. [4], is the most widely adopted density-based approach, promoting binary (0–1) solutions by penalizing intermediate densities and integrating effectively with finite element analysis (FEA).

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.

Meanwhile, efforts in open-source software development have also advanced TO practices. Chung et al. [10] leveraged OpenMDAO [13], decomposing TO into modular components with automated derivative assembly, enhancing flexibility and extensibility. Gupta et al. [17] developed a parallel-enabled implementation using the FEniCS finite element framework [1], addressing large-scale optimization. Recently, Ferro and Pavanello [11] 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 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, despite not being theoretically complex, can be tedious and error-prone, 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 [14]. 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. [21] employed the AD tools CoDiPack and Tapenade to achieve automatic sensitivity analysis in unsteady flow TO, significantly enhancing computational efficiency. Building upon this, Chandrasekhar [9] utilized the high-performance Python library JAX [7] to apply AD techniques in density-based TO, achieving efficient solutions to classical TO problems, exemplified by compliance minimization..

Although the programs described above—including early educational codes, open-source software implementations, and initial frameworks incorporating AD—have significantly promoted

the adoption and development of TO methods, they typically employ a procedural programming paradigm. This approach divides the numerical computation process into multiple interdependent subroutines, leading to a tightly coupled relationship between analysis modules (e.g., FEA) and optimization modules. Such tightly coupled architectures limit code extensibility and reusability: on one hand, the strong interdependencies between subroutines mean that even adding a new objective function or constraint often requires invasive modifications across multiple modules, increasing development time and potentially introducing new errors; on the other hand, this coupled architectural pattern makes it challenging to integrate TO programs as standalone modules within multidisciplinary design optimization (MDO) frameworks or system-level engineering processes. For instance, in aerospace applications, TO modules need seamless integration with other disciplines (such as fluid mechanics or thermodynamics), yet tightly coupled architectures typically result in complicated and inefficient integration procedures, hindering the application of TO in more complex scenarios. Consequently, designing an architecture that decouples analysis from optimization, thereby enhancing extensibility and reusability without sacrificing algorithmic accuracy, has become an important challenge that urgently needs addressing in the TO community.

To address the aforementioned challenges, this paper proposes SOPTX, a high-performance topology optimization framework built upon the FEALPy platform [31]. FEALPy is an open-source intelligent CAE computing engine providing an efficient, flexible, and extensible platform for numerical simulation. The choice of FEALPy as the underlying platform is motivated by several 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 finite element methods (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.

Building on the strengths of FEALPy, the SOPTX framework achieves a highly modular design. SOPTX adopts the component-based philosophy commonly used in MDO. Complex systems are decomposed into independent and reusable modules to enhance flexibility and maintainability. Specifically, SOPTX introduces a clear separation between analysis and optimization, with numerical solvers, AD tools, and optimization algorithms implemented as independent, interchangeable modules. This architecture allows users to flexibly select or replace components based on specific needs without substantial changes to the overall code structure. Through this modular design,

SOPTX inherits the efficiency and flexibility of FEALPy while significantly enhancing extensibility and reusability, providing strong support for complex engineering applications.

Leveraging this modular foundation, SOPTX seamlessly integrates AD into the topology optimization workflow. It supports multiple computational backends and can automatically switch between them based on hardware availability and performance requirements. Moreover, SOPTX mitigates computational bottlenecks associated with traditional numerical integration through a fast matrix assembly technique, which separates element-dependent and element-independent components, avoiding redundant computations during iterative procedures. Symbolic integration via SymPy [20] further reduces computational cost while improving accuracy. Maintaining extensibility and reusability, SOPTX adopts a non-intrusive design, enabling users to easily replace or augment filters, constraints, and objectives without modifying the core framework. Overall, SOPTX is designed as a low-barrier, high-performance platform for TO research and engineering applications, empowering developers to focus on algorithmic innovation and problem modeling rather than implementation details. These features make SOPTX not only well-suited for education and research, but also capable of supporting complex and 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 high-performance, 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 chapter, we first introduce the density-based method widely employed in topology optimization (TO), elucidating the fundamental idea of optimizing material distribution through the definition of a material density function and interpolation models. Next, taking the compliance minimization problem as an example, we explicitly present both continuous and discrete mathematical formulations of this method. Subsequently, we provide detailed descriptions of two classical optimization algorithms commonly used in TO: the Optimality Criteria (OC) method and the Method of Moving Asymptotes (MMA), including their algorithmic implementations and characteristics. Finally, we discuss critical filtering techniques in TO, including sensitivity, density, and the Heaviside projection filters, analyzing how these methods effectively mitigate numerical instabilities and enhance the physical feasibility of optimization results.

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:

- $\rho(x) = 1$ indicates regions occupied by solid material;
- $\rho(x) = 0$ represents void regions;
- $0 < \rho(x) < 1$ corresponds to intermediate densities, introduced to avoid the non-convexity inherent in discrete optimization problems by employing continuous design variables.

In a discretized design domain, the mechanical properties (e.g., stiffness) of material elements transition smoothly between solid and void states via interpolation models [2]. Among these, the power-law interpolation model has been widely adopted due to its implicit penalization of intermediate densities. This method, known as the Solid Isotropic Material with Penalization (SIMP) approach [32], 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.

As the element density ρ approaches zero, the standard SIMP model causes the element stiffness matrix to approach zero, potentially leading to singularities in the global stiffness matrix. To address this issue, the modified SIMP model introduces a 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 [26].

Although the SIMP method suppresses intermediate densities through the penalization factor, optimization results may still contain "gray areas", i.e., regions where the density is neither 0 nor 1, lacking clear physical correspondence to either material or void regions, potentially causing manufacturing difficulties. In practical applications, the SIMP approach is often combined with Heaviside projection filters to achieve a distinct 0–1 topology. Additionally, sensitivity or density filtering techniques are typically employed alongside the SIMP model to mitigate numerical instabilities, 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^* . Throughout this paper we assume the displacement $u \in H^1(\Omega; \mathbb{R}^d)$, the density $\rho \in L^2(\Omega)$ and the material stiffness tensor $D(\rho) \in L^\infty(\Omega)$.

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

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

Here $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$ and $\sigma(u) = D(\rho) : \varepsilon(u)$, f and t denote body forces and surface tractions, respectively, and $\Gamma_D \cup \Gamma_N = \partial D$.

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

$$\begin{aligned} \min_{\rho} : c(\rho) &= \mathbf{F}^T \mathbf{U} \\ \text{subject to: } &\sum_{e=1}^{N_e} v_e \rho_e - V^* \leq 0, \quad 0 \leq \rho_e \leq 1 \\ &\mathbf{K}(\rho) \mathbf{U}(\rho) = \mathbf{F} \end{aligned} \quad (2.2)$$

The global stiffness matrix is assembled as where $E(\rho_e)$ follows the SIMP law described in Section 2.1 and K_0^0 is the elemental matrix for unit Young's modulus.

Solving $\mathbf{K}(\rho) \mathbf{U}(\rho) = \mathbf{F}$ yields the discrete displacement vector \mathbf{U} , after which the objective $c(\rho)$ 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 under volume constraints. This method iteratively updates the design variables to improve the structural topology by satisfying the optimality conditions of the optimization problem. The core idea of the OC method is to adjust the material density ρ_e 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 [3] proposed a heuristic update scheme for the design variables, given by the following formula:

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

where:

- m is the move limit, which constrains the maximum allowable change in the design variables,
- η is the damping coefficient, which controls the step size and stability of the algorithm.

The procedure of the OC method is summarized in Algorithm 2.1.

Algorithm 2.1 OC pseudo-code

Input: Initial design variables $\rho^{(0)}$ (with $0 \leq \rho_e^{(0)} \leq 1$), volume constraint V^* , move limit m , damping factor η , maximum iterations MaxIter, convergence tolerance ϵ

Output: Optimized design variables ρ

Set $\rho^{(k)} \leftarrow \rho^{(0)}$, $k \leftarrow 0$

while $k < \text{MaxIter}$ and $\|\rho^{(k+1)} - \rho^{(k)}\|_\infty > \epsilon$ **do**

- Evaluate the objective function $c(\rho^{(k)})$ and the volume constraint $g(\rho^{(k)})$
- Compute the sensitivities $\nabla c(\rho^{(k)})$ and $\nabla g(\rho^{(k)})$
- (Optional) Apply a filter to the sensitivities
- Update the Lagrange multiplier λ using the bisection method
- Update the design variables ρ^{new} using the optimality criterion
- (Optional) Apply a filter to ρ^{new}
- Set $\rho^{(k+1)} \leftarrow \rho^{\text{new}}$, $k \leftarrow k + 1$

end

In TO, in addition to the OC method, the Method of Moving Asymptotes (MMA) is a widely used gradient-based optimization algorithm proposed by Svanberg [29]. 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:

$$\begin{aligned} \text{minimize} \quad & \tilde{f}_0^{(k)}(\rho) + a_0 z + \sum_{i=1}^m (c_i y_i + \frac{1}{2} d_i y_i^2) \\ \text{subject to} \quad & \tilde{f}_i^{(k)}(\rho) - a_i z - y_i \leq 0, \quad i = 1, \dots, m \\ & \alpha_j^{(k)} \leq \rho_j \leq \beta_j^{(k)}, \quad j = 1, \dots, n \\ & y_i \geq 0, \quad i = 1, \dots, m \\ & z \geq 0, \end{aligned}$$

where m is the number of constraints, n is the number of design variables, and ρ_j denotes the j -th design variable. The lower and upper bounds of the design variables in the k -th iteration are

denoted by $\alpha_j^{(k)}$ and $\beta_j^{(k)}$, respectively. The variables y_i and z are auxiliary variables introduced to ensure convex approximations of the objective and constraint functions. The parameters a_0, a_i, c_i, d_i are given constants.

The convex approximation functions are defined as:

$$\tilde{f}_i^{(k)}(\rho) = \sum_{j=1}^n \left(\frac{p_{ij}^{(k)}}{u_j^k - \rho_j} + \frac{q_{ij}^{(k)}}{\rho_j - l_j^{(k)}} \right) + r_i^{(k)}, \quad i = 0, 1, \dots, m,$$

where $l_j^{(k)}$ and $u_j^{(k)}$ are the lower and upper asymptotes (moving bounds) for ρ_j in the k -th iteration, and the coefficients $p_{ij}^{(k)}$, $q_{ij}^{(k)}$, and $r_i^{(k)}$ are computed based on the gradient information at the current iteration. Here, $i=0$ corresponds to the approximation of the objective function, and $i \geq 1$ corresponds to the approximations of the constraint functions.

Compared to the OC method, which is primarily suitable for problems with a single constraint, MMA is capable of efficiently handling multiple constraints and thus offers broader applicability.

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

Algorithm 2.2 MMA pseudo-code

Input: Initial design variables $\rho^{(0)}$, problem-specific parameters, MMA parameters

Output: Optimized design variables ρ

Set $\rho^{(k)} \leftarrow \rho^{(0)}$, $k \leftarrow 0$

while $k < \text{MaxIter}$ and $\|\rho^{(k+1)} - \rho^{(k)}\|_\infty > \epsilon$ **do**

Compute sensitivities $\nabla f_0(\rho^{(k)})$ and $\nabla f_i(\rho^{(k)})$, $i = 1, \dots, m$

(Optional) Apply filter to the sensitivities

Update asymptotes $l_j^{(k+1)}$ and $u_j^{(k+1)}$

Define variable bounds $\alpha_j^{(k)}$ and $\beta_j^{(k)}$

Construct convex approximations $\tilde{f}_i^{(k)}(\rho)$ to form the MMA subproblem

Solve the MMA subproblem using a primal-dual Newton method to obtain ρ^{new}

(Optional) Apply filter to ρ^{new}

Set $\rho^{(k+1)} \leftarrow \rho^{\text{new}}$, $k \leftarrow k + 1$

end

2.4 Filtering Methods

In practical computations of TO, pathological issues such as mesh dependency, checkerboard patterns, and local minima are frequently encountered [5]. These numerical difficulties can lead to suboptimal or unstable designs. Filtering techniques serve as an important form of regularization by smoothing the spatial distribution of either design variables or sensitivities, thereby suppressing numerical oscillations and improving both the reliability and physical realizability of the optimized structures [27]. A comprehensive overview of various filtering methods was provided by Sigmund [26].

The sensitivity filter was proposed by Sigmund [25] as a means to smooth the update of design variables by performing a weighted average on the sensitivities. The original mesh-dependent sensitivity filtering formula is given by:

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

where:

- H_{ij} is the weighting factor, defined using a linearly decaying function:

$$H_{ij} = \max\{0, r_{\min} - \text{dist}(i, j)\},$$

- $N_i = \{j : \text{dist}(i, j) \leq r_{\min}\}$ denotes the neighborhood of element i ;
- $\text{dist}(i, j)$ is the Euclidean distance between the centroids of elements i and j ;
- r_{\min} is the filter radius, which controls the range of influence of the filter;
- $\gamma = 10^{-3}$ is a numerical stability parameter used to avoid division by zero.

The density filter was originally proposed by Bruns and Tortorelli [8], and later mathematically justified by Bourdin [6] as a valid regularization technique. The basic form of the filtered density function is defined as:

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

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 smooths the spatial distribution of material by computing a weighted average of the densities within the neighborhood of each element using the weights H_{ij} . As a result, the original design variable ρ_i , which is directly updated during the optimization process, loses a clear physical interpretation. Instead, the filtered physical density $\tilde{\rho}_i$ is used to evaluate the structural performance and constraints. Therefore, the filtered density should be regarded as the final design in practical applications.

The sensitivity of the objective or constraint function ψ with respect to the design variable ρ_j is computed using the chain rule:

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

where:

- ψ denotes an objective or constraint function (e.g., compliance c or volume constraint g);
- N_j is the set of elements affected by the design variable ρ_j , i.e., all elements i for which $\rho_j \in N_i$.

In addition to sensitivity and density filters, the Heaviside projection filter is another important filtering technique, widely used to achieve clear black-and-white structural topologies [16]. The main objectives of the Heaviside projection filter are: (1) to impose a minimum length scale in the optimized design; and (2) to obtain a crisp, binary (0–1) solution.

The Heaviside projection filter introduces a Heaviside step function on top of the density filter to project the intermediate density $\tilde{\rho}_i$ to a physical density $\bar{\rho}_i$. Ideally, if $\tilde{\rho}_i > 0$, then the physical density $\bar{\rho}_i = 1$; and if $\tilde{\rho}_i < 0$, then $\bar{\rho}_i = 0$. To ensure differentiability and stability of the optimization algorithm, a smooth approximation is used in practice to replace the traditional Heaviside step function:

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

where the parameter β controls the smoothness of the approximation function. To avoid convergence to poor local minima and to ensure both the convergence and numerical stability of the algorithm, a continuation scheme is typically employed in numerical implementations, where β is gradually increased during the optimization. This progressively enhances the black-and-white projection effect of the filter, leading to a clearer structural topology.

After applying the Heaviside projection filter, the sensitivities of the objective and constraint functions with respect to the intermediate density $\tilde{\rho}_i$ must also be computed using the chain rule:

$$\frac{\partial \psi}{\partial \tilde{\rho}_i} = \frac{\partial \psi}{\partial \bar{\rho}_i} \frac{\partial \bar{\rho}_i}{\partial \tilde{\rho}_i},$$

where the derivative of the physical density $\bar{\rho}_i$ with respect to the intermediate density $\tilde{\rho}_i$ is given by:

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

The choice of the filter radius r_{\min} has a significant impact on the optimization result:

- When $r_{\min} \rightarrow 0$, the effect of the filter diminishes, making the optimized structure prone to local oscillations or checkerboard patterns;
- When $r_{\min} \rightarrow \infty$, the filter becomes overly aggressive, leading to excessive smoothing and loss of design details.

3 SOPTX Framework Design and Multi-Backend Switching

This chapter introduces the design principles and key components of the SOPTX framework, with an emphasis on its support for multi-backend switching to enhance computational performance and flexibility. The chapter is organized into three parts. We first examine the architecture of FEALPy and its role in providing a foundational tensor computation environment. We then describe the modular structure of SOPTX, including its material, solver, filter, and optimization modules. Finally, we discuss the design and advantages of the multi-backend mechanism, which enables compatibility with NumPy, PyTorch, and JAX to meet a range of computational needs.

3.1 FEALPy Architecture Design

As shown in Figure 1, FEALPy adopts a layered architecture consisting of four levels, arranged from bottom to top: the tensor level, the common level, the algorithm level, and the field level. Building on this architecture, FEALPy introduces the concept of a *Tensor Backend Manager*, which provides unified management and scheduling of different tensor computation backends. This mechanism offers a consistent tensor operation interface for the upper layers. The interface currently follows the Python Array API Standard v2023.12 [12], and is compatible with multiple backends such as NumPy, PyTorch, and JAX, enabling the same codebase to adapt seamlessly to different software and hardware platforms.

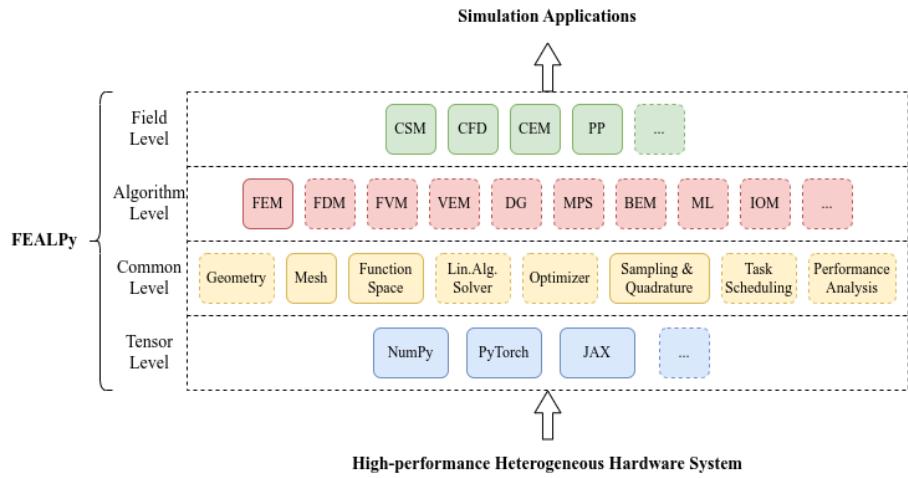


Figure 1: The layered architecture of FEALPy, comprising tensor, common, algorithm, and field levels, progressing from low-level functionalities to high-level applications. Modules in dashed boxes are under development.

Specifically, the functions of each layer are as follows:

- **Tensor level:** Provides basic tensor operations and manages backend systems such as NumPy, PyTorch, and JAX through the *Tensor Backend Manager*. This mechanism forms the foundation for SOPTX’s multi-backend switching. In particular, the automatic differentiation (AD) features of PyTorch and JAX enable automatic computation of sensitivities for objective and constraint functions in topology optimization (TO), greatly simplifying the derivative evaluation process.
- **Common level:** Includes components such as mesh generation and finite element spaces, offering the capability to rapidly construct meshes and function spaces required for finite element analysis in TO. This serves as the foundational support for finite element analysis in SOPTX.

- **Algorithm level:** Encompasses solvers and optimization algorithms, providing efficient computational support for optimization methods in SOPTX and ensuring both performance and stability during the optimization process.
- **Field level:** Targets specific physical problems (e.g., linear elasticity), enabling SOPTX to flexibly handle various types of TO problems such as compliance minimization under volume constraints, and offers customized support for application-specific scenarios.

Through this layered design, FEALPy balances functionality, extensibility, and performance. In particular, the introduction of the *Tensor Backend Manager* allows users to focus on upper-level algorithms and application logic without worrying about differences in underlying hardware or computational libraries. Moreover, the modules marked as under development in Figure ?? demonstrate FEALPy’s potential for continuous improvement, and the completion of these modules in the future is expected to further enhance SOPTX’s capabilities in solving complex TO problems.

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*, diverse numerical algorithm components, and general-purpose mesh and geometry handling capabilities. This design not only ensures flexibility and extensibility, but also significantly improves 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 well-defined interfaces, forming a loosely coupled and easily extensible multi-backend framework for TO.

3.2.1 Material Module

In its current version, SOPTX primarily focuses on linear elastic materials. The linear elastic material class is developed based on FEALPy’s corresponding implementation and extends its core functionalities—including the computation of Lamé constants, elasticity matrices, and strain matrices—by introducing additional interfaces tailored for TO. At present, the material module implements the Solid Isotropic Material with Penalization (SIMP) interpolation model, typically using a penalization factor $p=3$, which effectively transforms continuously varying density fields into discrete material distributions. The module is designed with highly abstract interfaces, ensuring extensibility. This design allows for easy future extensions to support other interpolation models such as the Rational Approximation of Material Properties (RAMP), as well as more complex material behaviors, including anisotropic, hyperelastic, and elastoplastic materials. These extensions can be realized by subclassing the base material class and implementing the required interfaces, without modifying other parts of the framework.

The material module encapsulates the interpolation, update, and evaluation of elastic constants within a unified interface, enabling convenient access to material information for downstream

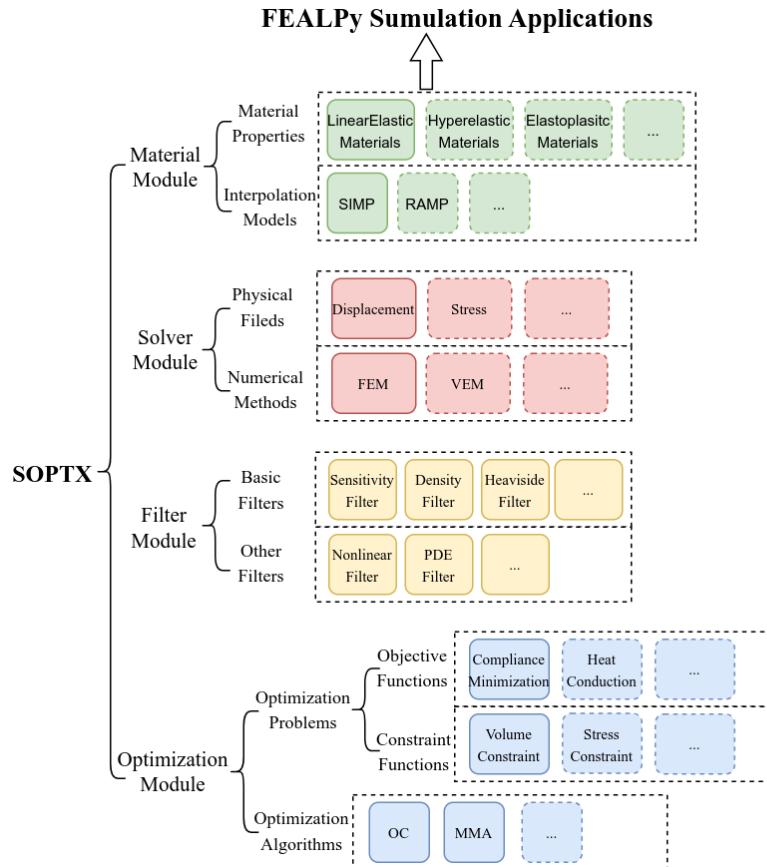


Figure 2: SOPTX is organized into four main modules: material, solver, filter, and optimization—forming a modular and extensible architecture. The material module provides the foundation, while the solver and filter modules handle intermediate computations and jointly support the optimization module. Dashed boxes indicate components under development.

modules. Its main functional interfaces include:

1. Computing elasticity tensors from a given density field, providing material stiffness data required by the solver module.
2. Supporting efficient batch updates of material properties, suitable for repeated evaluations in iterative optimization processes.
3. Supplying essential material parameters for the optimization module, such as the elastic constants of the base material.

These interfaces maintain a consistent calling convention across different computational backends, while their internal implementations are optimized according to backend-specific characteristics to ensure maximal computational efficiency. With this design, the material module serves as a key component in SOPTX, bridging physical modeling and numerical optimization. It maintains functional independence while providing essential material information throughout the topology optimization workflow. The modular structure not only simplifies the solution of current linear elasticity problems, but also offers a flexible foundation for future extensions to more complex material models.

3.2.2 Solver Module

The solver module serves as the computational core of the SOPTX framework. It is responsible for formulating and solving physical field problems (e.g., displacement fields) given material properties, boundary conditions, and external loads. The key design principle of this module is to construct an efficient, flexible, and backend-independent solver engine that meets the demands of repeatedly solving large-scale linear systems in TO. During iterative optimization, solving the physical field often becomes the main performance bottleneck as the design variables are updated. To address this, the solver module emphasizes computational efficiency through optimized matrix assembly, intelligent caching strategies, and multi-backend acceleration. It achieves high performance across various computational backends (NumPy / PyTorch / JAX). Meanwhile, the module maintains loose coupling with other components by accessing material data and providing results through clearly defined interfaces, ensuring smooth backend switching and modular extensibility.

In the current version, SOPTX primarily focuses on solving linear elasticity equations using the finite element method (FEM). The FEM solver class is developed based on FEALPy's finite element module, inheriting core functionalities such as linear elasticity integrators and finite element computation components. The solver module implements several key features:

1. **Dimensional and element adaptability:** It supports various spatial dimensions, element types, and boundary condition formulations, accommodating diverse engineering analysis requirements. In the current version, given that the density field is represented as piecewise constant per element, the displacement field is discretized using linear finite elements to ensure numerical stability and computational efficiency.
2. **Efficient numerical strategies:** In addition to standard finite element integration and matrix assembly, a fast matrix assembly technique is implemented. This approach separates the

element-independent and element-dependent parts of the stiffness matrix, avoiding redundant computations common in traditional methods and significantly improving efficiency for large-scale problems. Furthermore, symbolic integration is supported, enabling pre-computed exact expressions without the need for numerical quadrature, thereby further reducing computational cost.

3. **Multiple solution strategies:** The module supports both direct solvers (e.g., MUMPS) and iterative solvers (e.g., Conjugate Gradient, CG), allowing users to select the most suitable algorithm based on problem size and characteristics. Solver strategies are automatically optimized for the underlying backend—for example, leveraging GPUs acceleration for large sparse matrix computations on PyTorch and JAX backends.

The solver module is designed with strong extensibility. In terms of physical fields, beyond the current displacement field solver, the framework is planned to support efficient computation of additional fields such as stress. Regarding numerical methods, SOPTX intends to extend beyond the classical FEM to incorporate advanced techniques such as the Virtual Element Method (VEM). Moreover, the framework is extensible to accommodate nonlinear mechanics problems, multiphysics coupling, adaptive mesh refinement strategies, and parallel computing. In particular, as the framework evolves toward multi-resolution topology optimization, it will support intra-element heterogeneous density distributions and corresponding high-order finite element displacement fields, thereby improving both boundary resolution and optimization quality. All these extensions can be realized by extending the base solver class or introducing specialized solvers, without altering the overall architecture of the framework.

The solver module plays a pivotal role in SOPTX by bridging physical analysis and optimization computation. Its interactions and responsibilities are summarized as follows:

1. **Interaction with the material module:** The solver module receives material properties such as elasticity matrices from the material module, which are used to assemble the global stiffness matrix. This interaction is handled via unified interfaces, so that the solver does not depend on the specific material interpolation scheme.
2. **Output to the optimization module:** The solver module provides two key types of outputs to the optimization module: the displacement field, which is used to evaluate the objective function, and the stiffness matrix, which is used in sensitivity analysis. These outputs follow consistent interface definitions across different backends, while their internal representations are backend-specific and optimized to ensure efficient downstream computations.
3. **Result reuse:** Given the iterative nature of topology optimization, the solver module incorporates an intelligent caching mechanism to avoid redundant computations of invariant components (e.g., shape function derivative matrices). This significantly reduces computational overhead, particularly in large-scale problems.

Through its modular design and clearly defined interfaces, the solver module performs physical field computations efficiently, providing essential computational support for the entire topology

optimization process while maintaining consistency and high performance across multi-backend environments.

3.2.3 Filter Module

The filter module is a key component in SOPTX responsible for processing design variables and applying regularization. It plays a dual role in the topology optimization process: eliminating numerical instabilities such as checkerboarding and mesh dependency, and enhancing the manufacturability of the resulting structures. Following the overall loosely coupled design philosophy of the framework, the filter operations are implemented as an independent functional unit, while maintaining efficient data exchange with other modules. Similar to the material and solver modules, the filter module provides a consistent interface across different computational backends (NumPy / PyTorch / JAX), while its internal implementations are optimized according to backend-specific features to ensure maximal computational efficiency.

In the current version of the SOPTX framework, the filter module implements three main types of filtering techniques:

1. **Sensitivity filtering:** Applies weighted averaging to the sensitivities of the objective function, effectively suppressing checkerboard patterns.
2. **Density filtering:** Directly filters the design variables by mapping raw design values to physical densities, addressing the locality issues that may arise with sensitivity filtering in some problems.
3. **Heaviside projection filtering:** Builds upon density filtering by applying a smoothed Heaviside function to force intermediate densities toward 0 or 1, thereby promoting clear black-and-white structural designs.

The filter module constructs its filtering matrices using a KD-tree-based neighborhood search algorithm. Traditional linearly decaying filters are typically restricted to structured grids, but by leveraging the KD-tree data structure, SOPTX enables efficient spatial queries for neighboring points. This allows filtering to be seamlessly applied to arbitrary unstructured meshes and complex geometries, greatly expanding the applicability of the framework. To further enhance computational performance, the module adopts several optimization strategies, including precomputation of filter neighborhoods, sparse matrix representations, and backend-specific adaptations. These strategies ensure that filtering operations remain efficient even in large-scale topology optimization problems.

The filter module is designed with strong extensibility. Users can easily incorporate new filtering algorithms by subclassing the base filter class and implementing the required interfaces. Examples include nonlinear filters, PDE-based filters with exponential decay kernels, or filters based on alternative kernel functions. These newly added filters automatically inherit multi-backend compatibility without requiring additional adaptation. Moreover, the SOPTX framework supports chained composition of filters. Users can flexibly configure and combine multiple basic filters to construct complex filtering strategies that meet the specific requirements of different optimization scenarios.

The interactions between the filter module and other components are as follows:

1. **Interaction with the solver module:** It receives the raw design variables and outputs the filtered physical density field.
2. **Interaction with the optimizer module:** It takes in the unfiltered sensitivities computed by the optimization algorithm and returns the filtered sensitivities, ensuring numerical stability during the optimization process.
3. **Coordination with the material module:** The filtered physical density is directly used for material interpolation, forming a clear data flow:

design variables → filtering → physical density → material properties → physical response.

Through this modular design, the filter module effectively addresses numerical instability issues in TO and offers a flexible mechanism for extending various regularization strategies. Within the SOPTX framework, the filter module is not only an essential part of the optimization workflow but also a key enabler for achieving high-quality and manufacturable structural designs.

3.2.4 Optimization Module

The optimization module serves as the computational core of the SOPTX framework, responsible for integrating physical field solutions with optimization objectives to formulate complete mathematical optimization problems and invoke corresponding algorithms for their solution. Closely interacting with the material, solver, and filter modules, it completes the full computational pipeline for TO. Like other modules, the optimization module follows the multi-backend design philosophy of the framework, providing a consistent interface across NumPy, PyTorch, and JAX backends. Its internal implementations are backend-optimized to ensure high performance on each platform.

In the current version, the optimization module focuses on the classical compliance minimization problem under volume constraints and supports two mainstream topology optimization algorithms: Optimality Criteria (OC) and the Method of Moving Asymptotes (MMA). The MMA implementation is based on the standard version proposed by Svanberg in 2007 [30], and has been extended in SOPTX to support multiple backend implementations. While preserving mathematical equivalence, backend-specific optimizations have been introduced to improve computational performance, particularly enabling GPUs acceleration in PyTorch and JAX environments.

The design of the optimization module follows two core principles:

1. **Separation of problem definition and algorithm:** The definition of the optimization problem—namely, the objective functions, the constraint functions, and their sensitivities—is strictly decoupled from the optimization algorithms. This allows users to flexibly combine different problem formulations with various optimization solvers.
2. **Dual-mode sensitivity computation:** The framework supports both manually derived and automatically differentiated sensitivities. On PyTorch and JAX backends, users can directly leverage AD to avoid complex manual derivations, while still preserving the physical consistency and interpretability offered by hand-derived formulations.

The optimization module is designed with high extensibility:

1. **Problem type extension:** Users can easily define new optimization objectives—such as heat conduction, Compliant mechanism synthesis, or multiphysics problems—and corresponding constraint conditions, such as stress or frequency constraints, by subclassing the base optimization problem class.
2. **Algorithm extensibility:** The framework allows integration of new optimization algorithms, including Sequential Quadratic Programming (SQP), Sequential Linear Programming (SLP), and other gradient-based methods.

The optimization module serves as a central hub within the SOPTX framework, coordinating closely with other modules through the following interactions:

1. **Interaction with the solver module:** It receives physical field outputs such as the displacement field and stiffness matrix, which are used to evaluate the objective function (e.g., compliance) and its corresponding sensitivities.
2. **Interaction with the filter module:** It passes raw sensitivities to the filter module for processing and receives the filtered sensitivities to update the design variables. Additionally, the newly updated design variables are filtered again to obtain the physical density field.

Through this modular and flexible design, the optimization module not only efficiently solves the classical TO problems supported in the current version, but also lays a solid foundation for future extensions to more complex optimization scenarios. As the decision-making center of the SOPTX framework, it integrates the computational capabilities of all other modules into a complete structural design solution.

3.3 Multi-Backend Switching

In the field of STO, improving computational performance has always been a central concern in both research and engineering applications. To effectively address the diverse computational demands of problems at different scales, the SOPTX framework builds upon FEALPy to implement a flexible multi-backend support architecture. This allows users to seamlessly switch between multiple tensor computation backends such as NumPy, PyTorch, and JAX. This design not only enhances the applicability and efficiency of the software but also improves its portability and flexibility across different hardware and software platforms.

Specifically, different computational backends offer distinct advantages and are suitable for different application scenarios:

- **NumPy backend:** Suitable for small-scale computational tasks and rapid prototyping. It offers stable performance and broad community support. Due to its lightweight nature and efficient memory management, it is particularly well-suited for fast development and algorithm validation on standard CPUs platforms.

- **PyTorch and JAX backends:** Both support GPUs acceleration and AD, making them ideal for large-scale or high-dimensional problems. The AD capability greatly simplifies the computation of sensitivities for objective and constraint functions, improving development efficiency and significantly shortening the research cycle. JAX goes one step further by offering more flexible compilation strategies and automatic vectorization, achieving high computational efficiency, especially on GPUs platforms.

4 Getting Started with SOPTX

This chapter aims to provide readers with installation instructions and usage guidance for SOPTX, helping them quickly become familiar with the basic operations of the framework and apply it to topology optimization (TO) tasks. As a TO toolkit built on top of FEALPy, SOPTX offers a highly efficient and flexible computational environment through its multi-backend switching mechanism and modular design. The content of this chapter is divided into two parts: first, we present a detailed explanation of the installation steps for SOPTX and its dependency FEALPy, ensuring that readers can correctly configure the development environment; then, we demonstrate the usage workflow of SOPTX 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 CAE simulation engine that provides numerical computing capabilities. Installing FEALPy is a prerequisite, and it is recommended to install it from source. Before installation, ensure that Git and Python are properly installed. It is also recommended to use a virtual environment when installing and running FEALPy.

Core installation steps:

1. Clone the FEALPy repository from GitHub:

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

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

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

3. Similarly, install SOPTX:

```
1 git clone https://github.com/weihuayi/soptx.git
2 cd soptx
3 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 [5]: minimizing the structural compliance of a cantilever beam under tip loading (see Figure 3). 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.



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

As shown in Code 1, we first import relevant modules from FEALPy—including the backend, mesh, and function space—and from SOPTX, including the material, solver, filter, and optimization modules. Next, we define the partial differential equation (PDE) model of the cantilever beam, which includes the geometric configuration, applied load, and boundary conditions. The standard interface of a PDE model in SOPTX includes initialization, region definition, loading, and boundary conditions. The complete definition of the cantilever beam model `Cantilever2dData1` is provided in Appendix A.

Code 1: Module imports and PDE model

```

1  from fealpy.backend import backend_manager as bm
2  from fealpy.mesh import UniformMesh2d

```

```

3   from fealpy.functionspace import LagrangeFESpace,
4       TensorFunctionSpace
5
6   from soptx.material import (DensityBasedMaterialConfig,
7       DensityBasedMaterialInstance)
8   from soptx.solver import (ElasticFEMSolver, AssemblyMethod)
9   from soptx.filter_ import SensitivityBasicFilter
10  from soptx.opt import (ComplianceObjective, ComplianceConfig
11      , VolumeConstraint, VolumeConfig)
12  from soptx.opt import OC Optimizer
13
14  from soptx.pde import Cantilever2dData1
15
16  pde = Cantilever2dData1(xmin=0, xmax=160, ymin=0, ymax=100,
17      T = -1)

```

As shown in Code 2, the mesh and finite element function spaces are defined. The displacement field is represented using the first-order continuous Lagrange space, while the density field is defined in the zeroth-order discontinuous Lagrange space.

Code 2: Mesh and function space definitions

```

1  mesh = UniformMesh2d(extent=[0, 160, 0, 100], h=[1, 1],
2      origin=[0, 0])
3
4  space_C = LagrangeFESpace(mesh=mesh, p=1, ctype='C')
5  tensor_space_C = TensorFunctionSpace(scalar_space=space_C,
6      shape=(-1, 2))
5  space_D = LagrangeFESpace(mesh=mesh, p=0, ctype='D')

```

As shown in Code 3, the material module is instantiated using the material properties, the SIMP interpolation model, and the TO constants.

Code 3: Material module

```

1  material_config = DensityBasedMaterialConfig(
2      elastic_modulus=1.0,
3      minimal_modulus=1e-9,
4      poisson_ratio=0.3,
5      plane_assumption="plane_stress",
6      interpolation_model="SIMP",
7      penalty_factor=3.0)

```

```
8     materials = DensityBasedMaterialInstance(config=
  material_config)
```

As shown in Code 4, the solver module is instantiated using the material module, the PDE model, the matrix assembly method, and direct linear system solver (MUMPS). Subsequently, the sensitivity filter is initialized using the specified filter radius.

Code 4: Solver and filter module

```
1 solver = ElasticFEMSolver(
  2             materials=materials,
  3             tensor_space=tensor_space_C,
  4             pde=pde,
  5             assembly_method=AssemblyMethod.STANDARD,
  6             solver_type='direct',
  7             solver_params={'solver_type': 'mumps'})
  8 sens_filter = SensitivityBasicFilter(mesh=mesh, rmin=6.0)
```

As shown in Code 5, the objective function class is instantiated, followed by the volume constraint class based on the specified volume fraction. The optimizer is then initialized using the Optimality Criteria (OC) algorithm with its basic parameters. The maximum number of iterations is set to 200, and the convergence tolerance is set to 0.01 to control the termination criteria of the optimization process.

Code 5: Optimization module

```
1 objective = ComplianceObjective(solver=solver)
  2 constraint = VolumeConstraint(solver=solver, volume_fraction
  3 =0.4)
  4
  5 optimizer = OCOptimizer(objective=objective,
  6 constraint=constraint,
  7 filter=sens_filter,
  options={'max_iterations': 200, 'tolerance': 0.01})
```

As shown in Code 6, the initial density field is defined using an interpolation method. The optimization process is then executed, followed by saving the results and plotting the convergence history.

Code 6: Main program and post-processing

```
1 if __name__ == "__main__":
  2     @cartesian
```

```

3     def density_func(x):
4         val = bm.ones(x.shape[0])
5         # val = config.volume_fraction * bm.ones(x.shape[0],
6             **kwargs)
6         return val
7     rho = space_D.interpolate(u=density_func)
8     rho_opt, history = optimizer.optimize(rho=rho[:])
9
10    from soptx.opt import save_optimization_history,
11        plot_optimization_history
11    save_optimization_history(mesh, history)
12    plot_optimization_history(history)

```

The initial material density is uniformly distributed across the design domain, with each element initialized to the target volume fraction of 0.4. After running the code, Figure 4 shows the convergence histories of the compliance $c(\rho)$ and the volume fraction $v(\rho)$. The compliance $c(\rho)$ drops rapidly from its initial value of approximately 500 to around 100 within the first 10 iterations, and then converges smoothly, reaching convergence at iteration 57. The volume fraction $v(\rho)$ remains stably around 0.4, with only minor fluctuations.

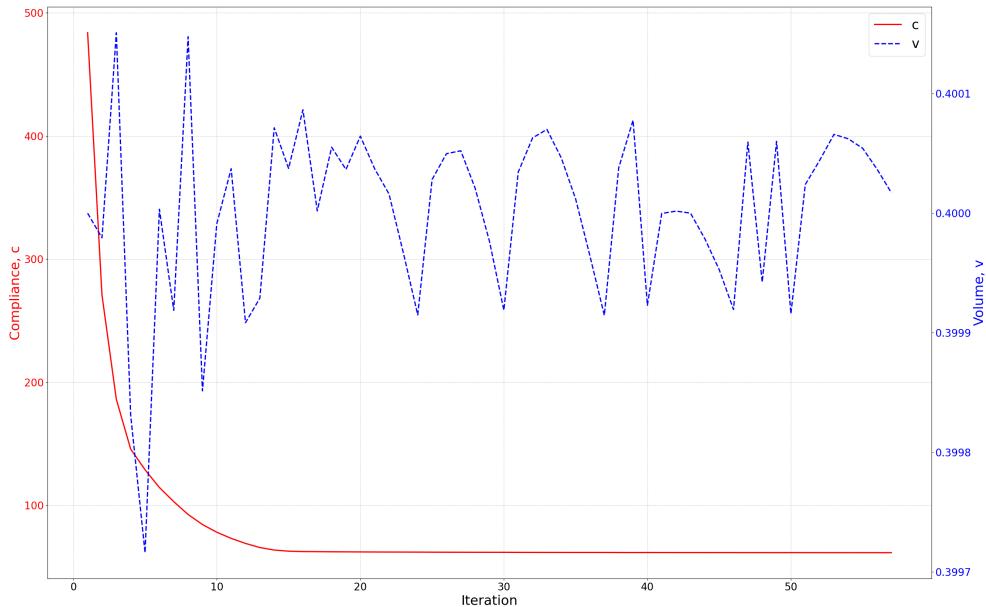


Figure 4: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam initialized with a uniform density of 0.4.

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

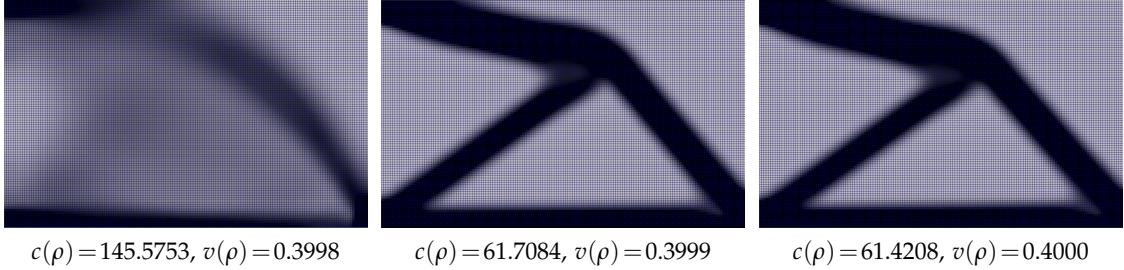


Figure 5: Topology layouts at iterations 3, 30, and 57 during the optimization process. Each subfigure also reports the corresponding compliance and volume fraction values.

The initial material density is uniformly set to 1 over the design domain. After running the optimization, Figure 6 shows the convergence histories of the compliance $c(\rho)$ and the volume fraction $v(\rho)$. The volume fraction smoothly decreases from 1 to the target value of 0.4 and stabilizes after approximately 5 iterations. The compliance $c(\rho)$ first increases from around 30 to 500 and then gradually decreases, converging at iteration 60. Compared with the case initialized at $\rho=0.4$, the optimization process starting from a fully solid design ($\rho=1$) requires slightly more iterations to converge, as it first reduces the volume fraction to satisfy the constraint before refining the topology.

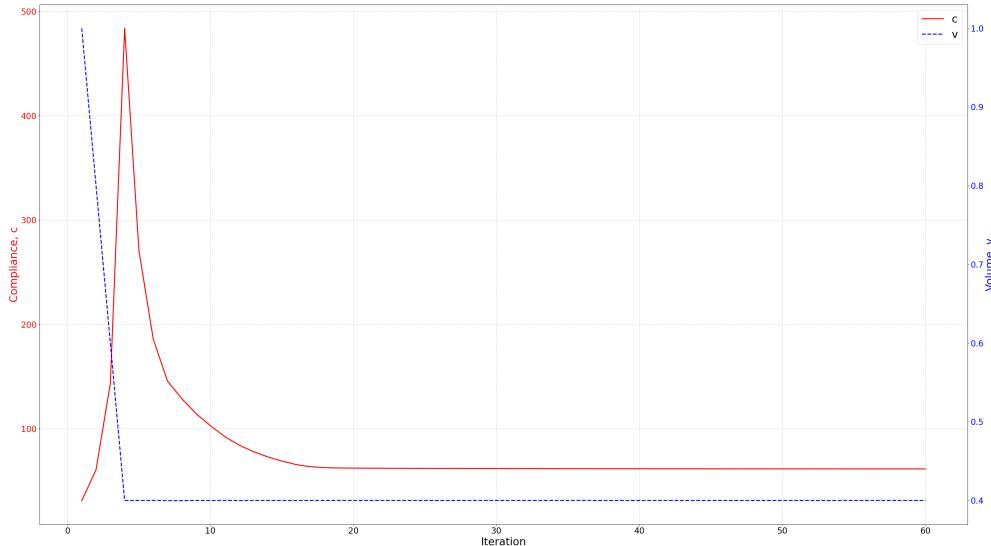


Figure 6: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam with an initial density of 1.

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

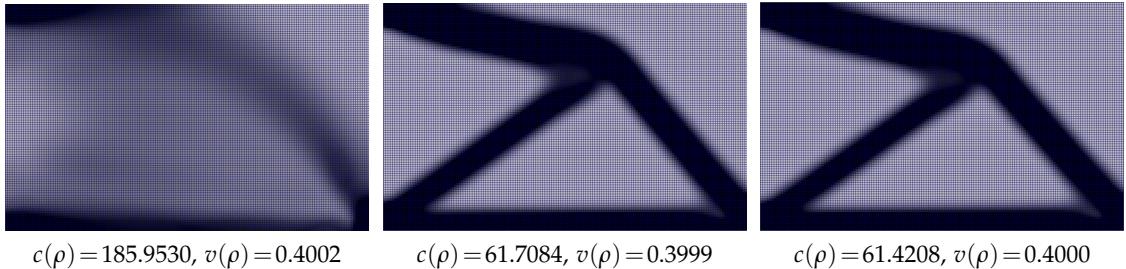


Figure 7: Topology layouts at iterations 6, 33, and 60 during the optimization process. Each subfigure includes the compliance and volume fraction.

5 Numerical Examples

This chapter presents a series of numerical examples that comprehensively demonstrate the capabilities and performance of the SOPTX framework in topology optimization (TO). The test cases span from classical 2D Messerschmitt-Bolkow-Blohm (MBB) beam problems to 3D cantilever structures, highlighting the flexibility and robustness of SOPTX in handling different partial differential equation (PDE) models, mesh types, filtering strategies, and optimization algorithms. In addition, this chapter explores the framework's advanced features, including fast matrix assembly, automatic differentiation (AD), and multi-backend switching, showcasing their significant benefits in improving computational efficiency and simplifying the development workflow.

The chapter is organized into seven sections: first, we present the MBB beam problem to demonstrate the robustness of SOPTX across different mesh resolutions; then, we investigate the impact of different filtering techniques on the optimization results; next, we introduce the algorithm switching interface and a backend-reconstructed version of the MMA optimizer; subsequently, we extend SOPTX to 3D TO; after that, we highlight how fast matrix assembly improves overall runtime efficiency; we then illustrate how AD simplifies the implementation of sensitivity analysis; finally, we discuss the benefits of the multi-backend switching mechanism and demonstrate the GPU acceleration potential. Through these examples, the functionalities of SOPTX are systematically demonstrated, providing readers with guidance for applying the framework in both scientific research and engineering scenarios.

5.1 MBB Beam

The MBB beam is a classical benchmark problem in TO, widely used to verify the effectiveness and robustness of optimization algorithms. In this section, we minimize the structural compliance of the MBB beam (see Figure 8) to demonstrate the flexibility and consistency of the SOPTX framework in handling different PDE models and mesh settings. The beam is subject to hinged

boundary conditions along its left edge and the bottom right corner, where only horizontal displacements are constrained (i.e., zero horizontal movement is enforced while vertical motion is free). A downward concentrated load $T = -1$ is applied to the upper left corner. The target volume fraction is set to 0.5, and the material parameters are $E = 1$ and $\nu = 0.3$. The penalization factor is $p = 3$, and the filter radius is $r = 6.0$, which matches the mesh element size to ensure smooth topology and suppress checkerboard patterns.



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.

Compared to the cantilever beam example in Section 4.2, the SOPTX framework allows users to switch to the MBB beam problem with minimal modifications. Thanks to its modular design, it is sufficient to change the PDE model. The full implementation of the MBB beam model `MBBBeam2dData1` is provided in [Appendix B](#).

```

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

```

To highlight the adaptability and robustness of SOPTX across different 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 the following code:

```

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

```

The resulting topologies are shown in Figure 9:

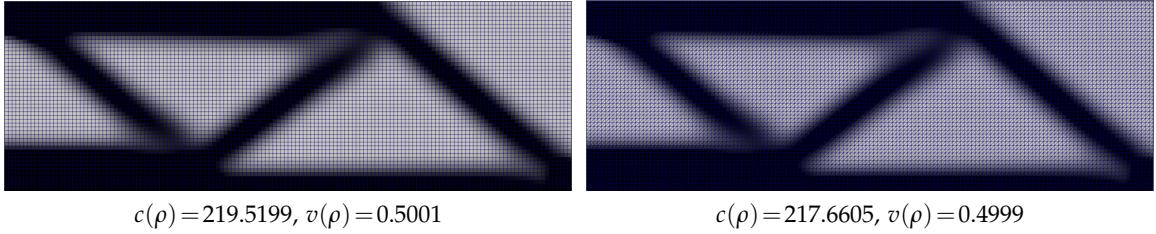


Figure 9: Optimized topologies of the MBB beam using a uniform quadrilateral mesh (left) and a triangular mesh (right).

As shown in Figure 9, the optimized layouts under both mesh types are highly consistent, with relative differences in compliance and volume fraction below 1%. This consistency demonstrates the mesh-independence and algorithmic robustness of SOPTX. Such behavior is enabled by its modular architecture, allowing users to easily switch between different PDE models and mesh configurations without modifying other parts of the framework.

5.2 Different Filtering Methods

In TO, the choice of filtering method plays a crucial role in shaping the final structure. Filters help smooth design variables and control structural details, significantly influencing both the quality and manufacturability of the optimized results. Thanks to its modular architecture, SOPTX supports seamless switching between various filtering strategies—users can simply replace the filter class without modifying other components of the framework.

To illustrate the impact of different filters, this section compares the results obtained using the density filter and the Heaviside projection filter. Both experiments are based on the MBB beam problem introduced in Section 5.1, with all parameters kept identical except for the choice of filter. Unless otherwise specified, the filter radius is set to $r=6.0$.

The density filter performs weighted averaging of the design variables to achieve a smooth spatial distribution, effectively eliminating small-scale features. This method is well-suited for design tasks that require structural continuity. In SOPTX, the density filter can be applied simply by instantiating the *DensityBasicFilter* class:

```

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

```

The Heaviside projection filter builds upon the density filter by introducing a projection operation. It employs a continuation strategy to gradually increase the projection parameter β , driving design variables toward 0 or 1 and producing clear black-and-white topologies. This method is particularly suitable for structural design problems with strict manufacturability requirements. To avoid overly wide bar-like structures in the early optimization stages, we slightly reduce the filter radius to $r=4.5$. The filter can be enabled in SOPTX using the following code:

```

1 heavi_filter = HeavisideProjectionBasicFilter(
2     mesh=mesh, rmin=4.5,
3     beta=1, max_beta=512, continuation_iter=50
4 )

```

Figure 10 shows the final optimized topologies obtained using the two filters:

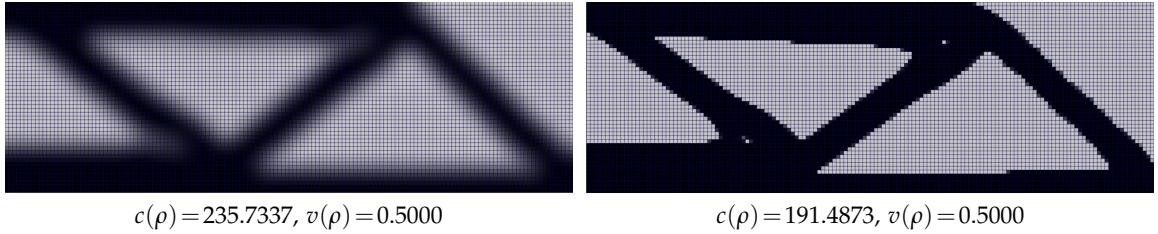


Figure 10: Optimized topologies of the MBB beam using the density filter (left) and the Heaviside projection filter (right).

As illustrated in the figure, the two filtering strategies result in distinctly different topologies and performance metrics. The density filter produces smoother layouts with blurred structural features, making it more suitable for design scenarios that prioritize continuity and gradual material transitions. In contrast, the Heaviside projection filter yields sharply defined boundaries and well-separated regions, leading to a clearly binarized layout with high visual and structural clarity.

Although the Heaviside filter enforces a minimum feature size through the specified filter radius, small holes with sizes below the threshold may still emerge during the optimization process. This is because the filter emphasizes global binarization rather than strict local geometric control, allowing some fine-scale structures to persist.

In summary, the modular design of SOPTX enables users to switch between different filtering strategies simply by modifying a single line of code. This facilitates efficient exploration of various design intents and helps strike a practical balance between structural smoothness, manufacturability, and binary clarity in TO.

5.3 Different Optimization Algorithms

In TO, the choice of optimization algorithm directly affects both computational efficiency and the quality of the resulting design. SOPTX adopts a modular design that enables seamless switching

between different optimizers, allowing users to easily tailor optimization strategies to specific design requirements. In this section, we take the MBB beam problem as an example to demonstrate the process of switching from the Optimality Criteria (OC) optimizer to the Method of Moving Asymptotes (MMA). We also present the refactored MMA implementation within SOPTX and highlight its distinctive advantages.

To address complex constrained optimization problems, SOPTX allows users to easily replace the OC optimizer with the more general MMA optimizer by making a simple modification. The switching process only requires importing the MMAOptimizer class and configuring the associated parameters as follows:

```

1  from soptx.opt import MMAOptimizer
2
3  optimizer = MMAOptimizer(objective=objective,
4                           constraint=constraint,
5                           filter_=sens_filter,
6                           options={'max_iterations': 200,
7                                   'tolerance': 0.01})

```

In the MBB beam problem, using the same parameters and sensitivity filter as with the OC optimizer, the MMA optimizer produces nearly identical optimized topologies across different meshes. As shown in Figure 11, the relative differences in compliance and volume fraction are both less than 1%. This indicates that although MMA is a more general optimization method, its performance on this specific problem is highly consistent with that of the OC optimizer, demonstrating its reliability and applicability.

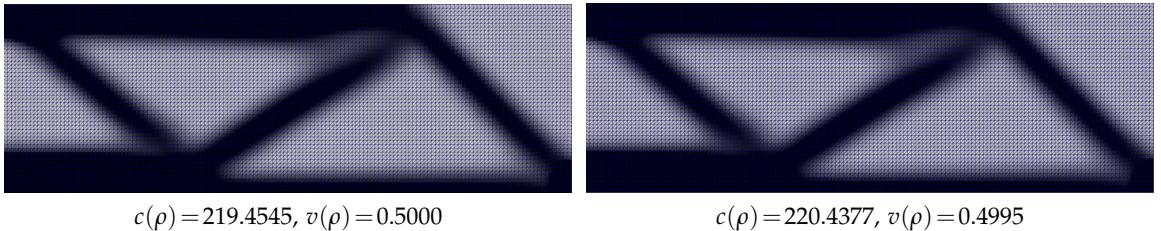


Figure 11: Optimized topologies of the MBB beam using MMA optimizer on a structured quadrilateral mesh (left) and a triangular mesh (right).

The MMA algorithm in SOPTX is based on the classical MATLAB implementation provided by Krister Svanberg [30], and has been extensively refactored. Unlike traditional MMA implementations commonly found in the literature—where the optimizer is typically treated as a “black box” with limited access to internal mechanisms—the refactored version in SOPTX allows users to conveniently adjust internal algorithmic parameters. These include the number of constraints m , the number of design variables n , the lower and upper bounds x_{\min} and x_{\max} , as well as internal

control parameters such as a_0 , a , c , and d :

```

1   optimizer.options.set_advanced_options(
2       m=1, n=NC,
3       xmin=bm.zeros((NC, 1)),
4       xmax=bm.ones((NC, 1)),
5       a0=1,
6       a=bm.zeros((1, 1)),
7       c=1e4 * bm.ones((1, 1)),
8       d=bm.zeros((1, 1)),
9   )

```

Through this refactoring, 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 3D Extension

Thanks to its modular design, SOPTX facilitates a straightforward transition from 2D to 3D models. In this section, we demonstrate the capability of SOPTX in solving 3D TO problems through the compliance minimization of a 3D cantilever beam, as illustrated in Figure 12. The beam is fully fixed on the left end, and a downward concentrated load of magnitude $T = -1$ is applied at the bottom of the right end. A structured uniform grid of hexahedral elements with size $60 \times 20 \times 4$ is employed. The target volume fraction is set to 0.3, and the material parameters are $E = 1$ and $\nu = 0.3$. The SIMP penalty factor is chosen as $p = 3$. A sensitivity filter with radius $r = 1.5$ (matched to the mesh element size) is used to ensure structural smoothness and suppress checkerboard patterns.

Compared to the 2D cantilever beam example in Section 4.2, SOPTX can be seamlessly extended to the 3D cantilever beam problem by simply replacing the PDE model and mesh configuration. The 3D cantilever beam model is implemented via the `Cantilever3dData1` class, as shown below, and the complete code definition is available in [Appendix C](#).

```

1   from soptx.pde import Cantilever3dData1
2   pde = Cantilever3dData1(xmin=0, xmax=60, ymin=0, ymax=20,
3                           zmin=0, zmax=4, T = -1)
4
5   from fealpy.mesh import UniformMesh3d
6   mesh = UniformMesh3d(extent=[0, 60, 0, 20, 0, 4], h=[1, 1,
7                           1], origin=[0, 0, 0])

```

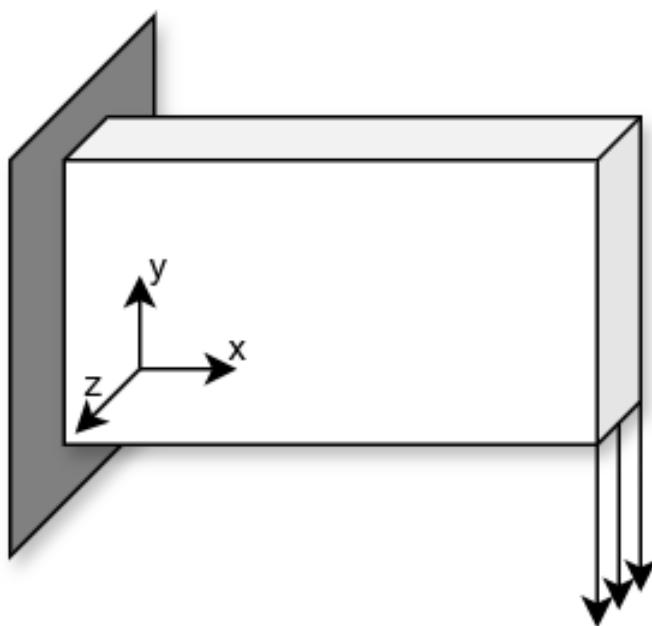


Figure 12: Geometric configuration of the 3D cantilever beam. The left end is fully fixed, and a downward concentrated load is applied at the bottom of the right end.

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 convergence histories of the compliance $c(\rho)$ and the volume fraction $v(\rho)$ are shown in Figure 13. The compliance drops rapidly from an initial value of approximately 28000 to around 5000 within the first 6 iterations. It then decreases more gradually, fluctuating slightly between iterations 6 and 20 before settling to about 2000. Smooth convergence is eventually achieved at iteration 54. The volume fraction remains nearly constant around 0.3 throughout the process, indicating effective constraint control.

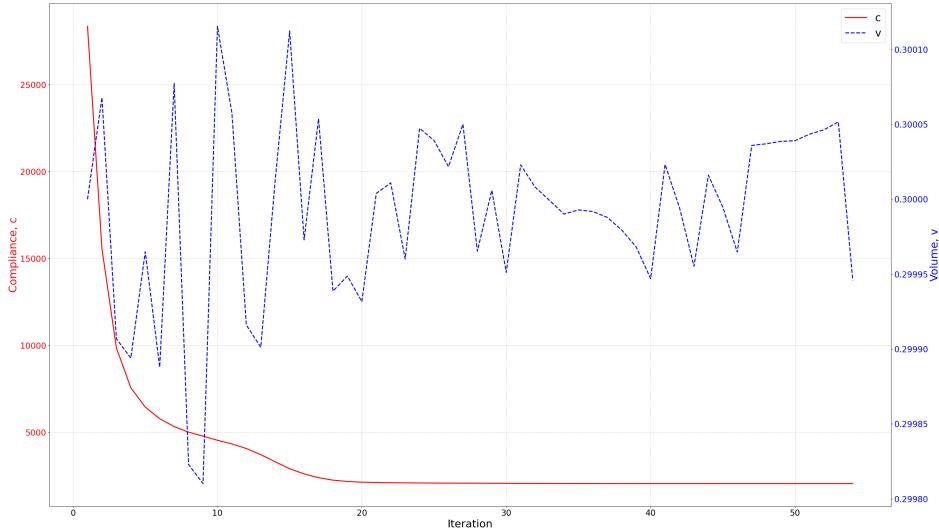


Figure 13: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam initialized with a uniform density of 0.3.

To better visualize the optimization results, Figure 14 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.

It is worth noting that, compared to 2D problems, 3D TO significantly increases computational complexity. This is particularly evident when using large-scale meshes, where both computational cost and memory requirements grow substantially. The SOPTX framework addresses these challenges through efficient matrix assembly and support for multiple computational backends, demonstrating excellent extensibility and computational performance. The implementation details of these features will be elaborated in the following sections.

5.5 Application of Fast Matrix Assembly

In 3D TO problems, computational efficiency is a central challenge. Taking the 3D cantilever beam example from Section 5.4 as a representative case, the mesh consists of $60 \times 20 \times 4$ elements, resulting in 19,215 displacement degrees of freedom and 4,800 density variables. The optimization

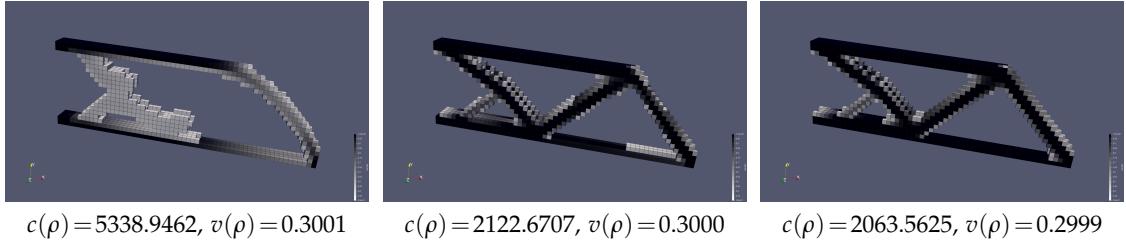


Figure 14: Topology layouts at iterations 7, 21, and 54 during the optimization of the 3D cantilever beam. Elements with $\rho > 0.3$ are visualized. Each subfigure also reports the compliance and volume fraction.

process requires 54 iterations in total. In traditional finite element methods, the assembly of the global stiffness matrix often involves a large number of redundant numerical integration operations, which becomes the primary bottleneck in the overall computational workflow.

To overcome this bottleneck, the SOPTX framework incorporates a fast matrix assembly technique, which separates the element-dependent and element-independent parts of stiffness matrix computation. This decomposition enables efficient evaluation and reuse of invariant quantities, significantly accelerating the assembly process. In addition, to further improve the accuracy of matrix assembly, SOPTX implements a symbolic integration precomputation technique. This approach analytically integrates the element-independent components using symbolic computation in advance, eliminating numerical integration errors and enhancing both numerical stability and overall accuracy.

- **Fast matrix assembly:** Accelerates matrix assembly by separating element-dependent and element-independent terms, coupled with efficient numerical integration.
- **Symbolic integration precomputation:** Further improves accuracy and stability by analytically integrating invariant terms ahead of time.

Moreover, SOPTX employs intelligent caching to store invariant data across iterations, enabling direct reuse and substantially reducing average computational time in later iterations.

Table 1 summarizes the performance of different matrix assembly techniques under the same problem setting of the 3D cantilever beam optimization described in Section 5.4.

As shown in Table 1, the original matrix assembly method results in an average assembly time of 0.838 s per iteration after the first one, accounting for approximately 68% of the total iteration time. This clearly constitutes a performance bottleneck. With the adoption of the fast matrix assembly technique, the average assembly time is reduced to 0.276 s—only about 33% of that of the original method. Although the symbolic fast assembly method incurs a longer initialization time of 3.853 s during the first iteration, it achieves the lowest average assembly time in subsequent iterations (0.272 s), while ensuring analytical accuracy in integration.

In summary, the matrix fast assembly strategy implemented in SOPTX effectively reduces the computational cost of topology optimization and improves both numerical accuracy and stability.

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

This makes it particularly well suited for solving large-scale topology optimization problems with stringent demands on computational efficiency and precision.

5.6 Application of Automatic Differentiation

Sensitivity analysis is a core component of TO, guiding the update of design variables. Traditional methods typically require manual derivation and implementation of sensitivity formulas, which can be time-consuming and error-prone, especially for complex material models or constraint conditions. The SOPTX framework introduces AD to automate sensitivity computations, allowing users to focus on problem modeling and solution strategies without dealing with tedious mathematical derivations. In this section, we use the 3D cantilever beam example from Section 5.4 to demonstrate how SOPTX leverages AD to compute the sensitivities of the compliance objective and the volume constraint. The advantages and application scenarios of AD in TO are also discussed.

The SOPTX framework supports multiple computational backends, including NumPy, PyTorch, and JAX. While FEALPy by default operates with the NumPy backend [18], NumPy does not support AD. Therefore, users need to switch to a backend that supports AD in order to enable this functionality. For example, switching to the PyTorch backend [22] requires only a single line of code:

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

In SOPTX, the mode of sensitivity computation for objective and constraint functions is controlled by the configuration parameter `diff_mode`. By default, `diff_mode` is set to '`manual`', meaning that manually derived sensitivity expressions are used. To enable AD, users simply set `diff_mode` to '`auto`'. The following code snippet shows how to enable AD for the structural compliance objective while retaining manual differentiation for the volume constraint:

```
1 obj_config = ComplianceConfig(diff_mode='auto')
2 objective = ComplianceObjective(solver=solver, config=
3     obj_config)
```

```

4     cons_config = VolumeConfig(diff_mode='manual')
5     constraint = VolumeConstraint(solver=solver, volume_fraction
       =0.5, config=cons_config)

```

Under this configuration, the sensitivity of the compliance objective is computed automatically via AD, whereas the sensitivity of the volume constraint remains manually derived. This flexibility allows users to selectively choose the most appropriate differentiation method according to the problem requirements. The detailed implementation of the compliance sensitivity computation using AD is provided in [Appendix D](#).

To verify the correctness and effectiveness of AD, we conduct tests using the 3D cantilever beam problem described in [Section 5.4](#). All parameters are kept identical: the mesh size is $60 \times 20 \times 4$, the target volume fraction is set to 0.3, the initial material density is 0.3, material properties are $E = 1$ and $\nu = 0.3$, the SIMP penalty factor is $p = 3$, and the sensitivity filter radius is $r = 1.5$. The structural compliance sensitivity is computed using AD. Figure 15 shows the convergence histories of the compliance and volume fraction.

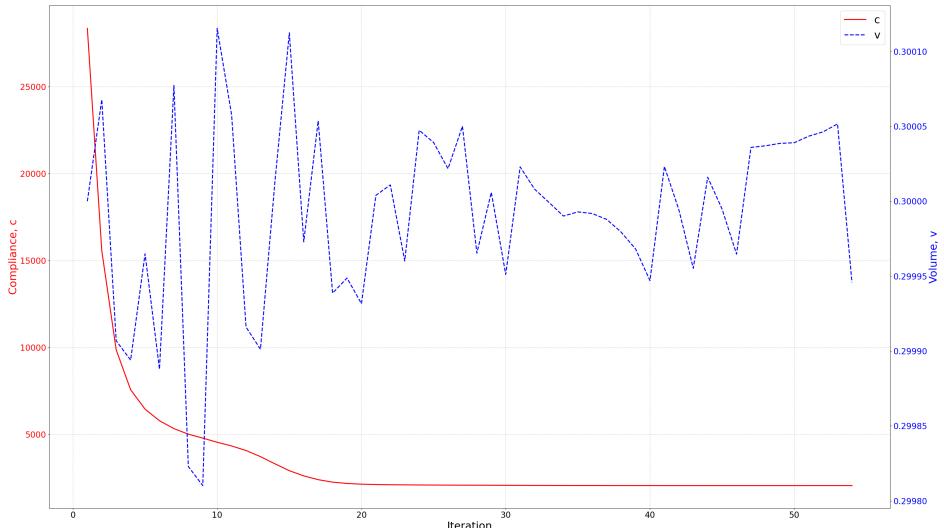


Figure 15: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam optimization using automatic differentiation.

To further illustrate the evolution of the topology during optimization, the layouts at selected iterations are shown in [Figure 16](#).

As observed from [Figure 15](#) and [Figure 16](#), the optimization results obtained using AD are fully consistent with those achieved through manual differentiation in [Section 5.4](#). 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.

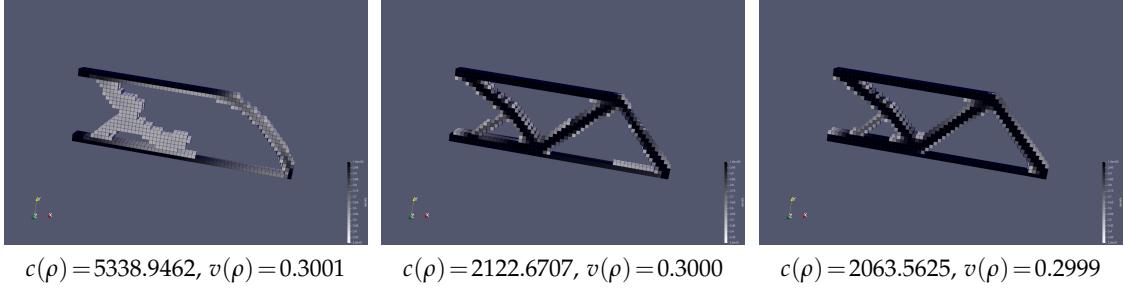


Figure 16: Topology layouts at iterations 7, 21, and 54 during the 3D cantilever beam optimization using AD. Only elements with $\rho > 0.3$ are visualized. Each subfigure also reports the compliance and volume fraction.

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. The results are summarized in Table 2.

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

As shown in Table 2, the total optimization time and the average iteration time between manual differentiation and AD are nearly identical. This demonstrates the high efficiency of the AD implementation in SOPTX, which introduces no significant computational overhead while maintaining the same high precision as manual differentiation.

The multi-backend mechanism of SOPTX allows users to flexibly switch between different computational backends. In addition to PyTorch, SOPTX also supports the JAX backend [7], which offers equally powerful AD capabilities. The sensitivity computation code remains identical when using JAX, users simply need to switch to the JAX backend by executing:

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

Using AD with the JAX backend, we perform topology optimization for the 3D cantilever beam. The final optimized topology is shown in Figure 17.

It is noteworthy that the final optimization result obtained using JAX is fully consistent with that obtained using PyTorch, demonstrating the stability and feasibility of the SOPTX framework when applying AD across different computational backends.

The application of AD in TO offers several significant advantages:

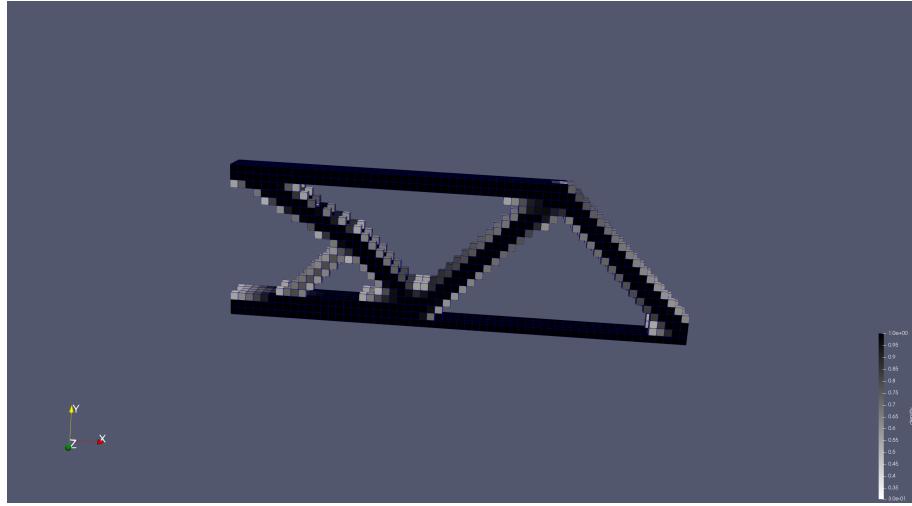


Figure 17: Final optimized topology of the 3D cantilever beam using AD with the JAX backend.

- **Simplified Model Switching:** In TO, the choice of material interpolation model directly influences the optimization results. Besides the commonly used SIMP model, alternative models such as RAMP [28] are also widely applied in specific scenarios. With AD, users can easily switch between different interpolation models. For instance, when switching from SIMP to RAMP, the material stiffness gradient can be automatically computed by AD without manually deriving complex formulas. This greatly improves development efficiency and enhances the flexibility of model experimentation.
- **Seamless Constraint Switching:** TO often involves multiple constraints. In addition to volume constraints, manufacturing constraints such as length scale control [15], connectivity [19], overhang limitations [23], and material usage restrictions [24] are frequently imposed. Traditionally, adding or modifying constraints requires manually re-deriving sensitivity expressions. By introducing AD, this process becomes fully automated. The system can seamlessly adapt to new constraint settings, significantly improving optimization adaptability and development efficiency.
- **Support for Complex Problems:** For TO problems involving multiphysics coupling, nonlinear materials, or complex geometric constraints, AD efficiently handles the computation of complicated gradients, allowing users to focus on modeling rather than mathematical derivations. Moreover, AD can automatically manage combined sensitivity computations when using techniques like density filtering and Heaviside projection. This capability further enhances the practical applicability and flexibility of the SOPTX framework in complex engineering scenarios.

Through the optimization example of the 3D cantilever beam, this section has demonstrated the powerful potential of AD within the SOPTX framework. Numerical results show that AD

achieves comparable precision and computational efficiency to traditional manual differentiation methods without introducing any significant overhead. The multi-backend support of SOPTX, including PyTorch and JAX, further enhances the flexibility and scalability of the framework. The application of AD not only simplifies sensitivity analysis but also facilitates the exploration of novel material models and constraint conditions. These features make SOPTX a valuable tool in the field of topology optimization, providing robust support for education, research, and engineering applications.

5.7 Multi-Backend Switching

The SOPTX framework supports multi-backend switching, currently offering three computational backends: NumPy, PyTorch, and JAX. Backend switching is managed through the *Tensor Backend Manager*, and users can easily switch between different backends by using the `set_backend` function:

```

1 bm.set_backend('numpy')
2 bm.set_backend('pytorch')
3 bm.set_backend('jax')
```

To verify the consistency and reliability of SOPTX across different computational backends, we conduct tests on the 3D cantilever beam optimization problem described in Sections 5.4 and 5.6, using identical parameter settings. The problem is solved separately under the NumPy, PyTorch, and JAX backends. The optimization results show that the convergence histories and final topology layouts are highly consistent across all three backends:

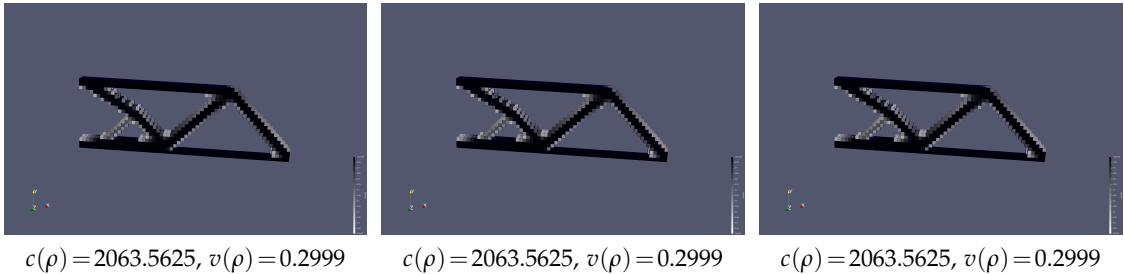


Figure 18: Final optimized topologies of the 3D cantilever beam using three different backends (elements with $\rho > 0.3$ are visualized). Left: NumPy backend; Middle: PyTorch backend; Right: JAX backend.

As shown in Figure 18, after 54 iterations, the compliance in all three backends converges to approximately 2063.5625, and the volume fraction stabilizes around 0.3. The final topology layouts are nearly identical. This consistency highlights the effectiveness of SOPTX’s unified numerical abstraction and rigorous testing strategy, ensuring the reliability and stability of the algorithms across different computational backends.

In addition to multi-backend switching, SOPTX also supports computations on GPUs to further enhance computational efficiency, especially when solving large-scale 3D TO problems. By default, all computations are performed on the CPU (`device='cpu'`). However, users can migrate computations to the GPU (e.g., CUDA devices) through two methods:

- 1. Set the Default Device Globally:** Users can set the default device for all tensors globally by executing:

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

This method places all subsequently created tensors on the GPU by default, and all associated computations, such as matrix assembly and linear solving, are performed on the GPU.

- 2. Specify the Device When Creating the Mesh:** Alternatively, users can specify the device during mesh initialization:

```
1 mesh = UniformMesh3d(extent=extent, h=h, origin=origin,
                      device='cuda')
```

By specifying `device='cuda'` during mesh creation, all computations related to that mesh are performed on the GPU. This approach offers greater flexibility, allowing users to mix CPU and GPU computations within the same program if needed.

GPU acceleration plays a crucial role in 3D TO, as computational complexity increases significantly with the growth of mesh size. Taking the 3D cantilever beam example from Section 5.4 as a reference, when the mesh size is doubled to $120 \times 40 \times 8$, the number of density design variables rises to 38,400, and the number of displacement degrees of freedom reaches 133,947. On the CPU, the optimization process becomes much more time-consuming due to the computational cost associated with matrix assembly and linear solving. In contrast, the parallel computing capability of GPUs can substantially improve efficiency and significantly reduce computation time.

To intuitively demonstrate the effect of GPU acceleration, we compare the computation times between CPU and GPU under the PyTorch backend for the same 3D cantilever beam problem. The parameter settings are kept identical to those in Section 5.4, except that the filter radius is increased from 1.5 to 3.0 to ensure structural smoothness given the enlarged physical domain. Additionally, considering the increased problem size, we switch the linear solver from a direct method (e.g., MUMPS) to an iterative method, namely the Conjugate Gradient (CG) method. The CG method offers advantages for solving large-scale sparse linear systems and can leverage the GPU's parallel computing capabilities to efficiently perform matrix-vector operations.

The tests are conducted on an AMD 9950X CPU and an NVIDIA 5070Ti GPU. The optimization results are shown below:

As shown in Figure 19, the optimized topologies obtained using the CPU and GPU are highly similar. Although slight differences are observed in the final compliance ($c(\rho)$) and volume frac-

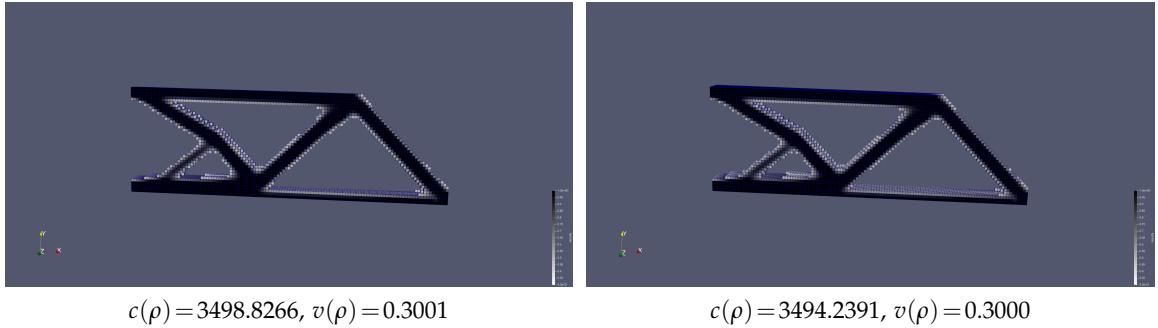


Figure 19: Optimized topologies of the 3D cantilever beam on CPU (left) and GPU (right), with elements of density $\rho > 0.3$ visualized.

tion ($v(\rho)$)—both within 1%—these discrepancies are attributed to minor variations in floating-point arithmetic precision and parallel implementation strategies across different hardware platforms. Nevertheless, in TO, such minor differences are negligible and do not affect the quality or performance of the final design.

Based on the consistency of the optimization results, we further compare the computational efficiency between the CPU and GPU. The detailed performance data are summarized in Table 3.

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

As shown in Table 3, GPU acceleration significantly reduces the total optimization time from 3872.041 s on the CPU to 479.412 s on the GPU, achieving an approximate speedup of 8.1 times. The GPU outperforms the CPU in both the first iteration and subsequent iterations, clearly demonstrating its advantages in parallel computing.

When using the CG method, the first iteration time (CPU: 15.846 s, GPU: 2.111 s) is notably lower than the average subsequent iteration time (CPU: 25.040 s, GPU: 3.099 s). This behavior is due to the fact that, at the initial stage of optimization, the material density distribution is relatively uniform, resulting in a stiffness matrix with a low condition number, allowing the CG method to converge quickly. As the optimization progresses and the material distribution becomes more heterogeneous, the condition number of the matrix deteriorates, requiring more internal CG iterations for convergence, thus increasing the computational time per iteration.

6 Conclusion

This work presents SOPTX, a high-performance topology optimization (TO) framework based on FEALPy. A series of classical benchmark examples have been conducted to validate the effectiveness and practicality of SOPTX in solving TO problems. The development of SOPTX is entirely based on the open-source software FEALPy, without relying on any commercial software. This fully open-source nature not only lowers the technical barrier but also enables users to easily access, learn, and utilize the framework, making it particularly suitable for academic research and educational purposes.

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

1. **Modular Design Framework:** Through a loosely coupled architecture, SOPTX supports topology optimization problems in both two and three dimensions, as well as various mesh types (structured and unstructured). It provides a rich library of combinable components, enabling users to flexibly configure and extend the optimization workflow.
2. **Cross-Platform Multi-Backend Support and Automatic Differentiation:** Numerical results show that GPU computations take only approximately 33% of the CPU time. Leveraging AD, SOPTX enables the automatic computation of gradients for objective and constraint functions, simplifying sensitivity analysis, reducing development difficulty, and improving computational accuracy.
3. **Fast Matrix Assembly Technique:** By optimizing the matrix assembly process, reducing redundant computations, and exploiting matrix sparsity, SOPTX significantly improves finite element matrix assembly efficiency. Benchmark results demonstrate that the average assembly time is reduced from 0.838 s to 0.273 s, corresponding to approximately 33% of the original method.

Through the example analyses presented in Sections 5.1 to 5.7, SOPTX demonstrates outstanding performance in model switching, filter method selection, optimization algorithm flexibility, three-dimensional extension, computational efficiency enhancement, application of automatic differentiation, and multi-backend switching, fully validating its technical advantages.

Although SOPTX currently focuses primarily on density-based TO, it holds promising potential for future development. Leveraging the capabilities of FEALPy in solving heat conduction and Navier–Stokes equations, SOPTX can be extended to optimize thermo-fluid-structure coupling problems, such as thermal protection system designs in aerospace applications or heat dissipation optimization in electronic devices.

By integrating FEALPy’s level set solver module, SOPTX can also support boundary-clear level set methods, enabling the generation of smooth boundaries suitable for precision manufacturing scenarios. In addition, the incorporation of FEALPy’s adaptive mesh refinement techniques—particularly local mesh refinement near boundaries—could further enhance the solution accuracy for complex geometries and improve the geometric details of optimized structures.

Moreover, SOPTX plans to support stress constraints and manufacturing constraints, especially targeting applications in 3D printing and additive manufacturing. Future extensions may

include overhang angle control and support structure optimization, aiming to reduce the use of support materials, improve printing efficiency, and meet the high standards of lightweight design and manufacturability required in fields such as aerospace, automotive engineering, and biomedical engineering.

In summary, SOPTX significantly enhances research and development efficiency in the field of TO through its modular design, automatic differentiation capabilities, and multi-backend support. Its open-source nature further lowers the barrier to adoption. Compared to other implementations, SOPTX exhibits notable advantages in computational efficiency, flexibility, and ease of use, providing a solid foundation for both academic research and industrial applications. Looking ahead, SOPTX is expected to play a greater role in multiphysics optimization, lightweight structural design, and additive manufacturing, accelerating the widespread adoption and application of TO in industrial design and manufacturing.

Acknowledgments

The first and fourth authors were supported by the National Natural Science Foundation of China (NSFC) (Grant No. 12371410, 12261131501), and the construction of innovative provinces in Hunan Province (Grant No. 2021GK1010). The second author was supported by NSF DMS-2012465 and DMS-2309785. The third author was supported by the National Natural Science Foundation of China (NSFC) (Grant No. 12171300), and the Natural Science Foundation of Shanghai (Grant No. 21ZR1480500).

Appendix A. Cantilever2dData1

```

1  class Cantilever2dData1:
2      def __init__(self,
3          xmin: float, xmax: float,
4          ymin: float, ymax: float,
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:
12         box = [self.xmin, self.xmax, self.ymin, self.ymax]
13         return box
14
15     @cartesian

```

```

16     def force(self, points: TensorLike) -> TensorLike:
17         domain = self.domain()
18         x = points[..., 0]
19         y = points[..., 1]
20         coord = (
21             (bm.abs(x - domain[1]) < self.eps) &
22             (bm.abs(y - domain[2]) < self.eps)
23         )
24         kwargs = bm.context(points)
25         val = bm.zeros(points.shape, **kwargs)
26         val[coord, 1] = self.T
27         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
36         ) -> TensorLike:
37         domain = self.domain()
38         x = points[..., 0]
39         coord = bm.abs(x - domain[0]) < self.eps
40         return coord
41
42     @cartesian
43     def is_dirichlet_boundary_dof_y(self, points: TensorLike
44         ) -> TensorLike:
45         domain = self.domain()
46         x = points[..., 0]
47         coord = bm.abs(x - domain[0]) < self.eps
48         return coord
49
50     def threshold(self) -> Tuple[Callable, Callable]:
51         return (self.is_dirichlet_boundary_dof_x,
52                 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:
12          box = [self.xmin, self.xmax, self.ymin, self.ymax]
13          return box
14
15      @cartesian
16      def force(self, points: TensorLike) -> TensorLike:
17          domain = self.domain()
18
19          x = points[..., 0]
20          y = points[..., 1]
21
22          coord = (
23              (bm.abs(x - domain[0]) < self.eps) &
24              (bm.abs(y - domain[3]) < self.eps)
25          )
26          kwargs = bm.context(points)
27          val = bm.zeros(points.shape, **kwargs)
28          val[coord, 1] = self.T
29          return val
30
31      @cartesian
32      def dirichlet(self, points: TensorLike) -> TensorLike:
33          kwargs = bm.context(points)
34          return bm.zeros(points.shape, **kwargs)
35
36      @cartesian
37      def is_dirichlet_boundary_dof_x(self, points: TensorLike) ->
38          TensorLike:
39          domain = self.domain()
40          x = points[..., 0]
41          coord = bm.abs(x - domain[0]) < self.eps

```

```

41     return coord
42
43     @cartesian
44     def is_dirichlet_boundary_dof_y(self, points: TensorLike) ->
45         TensorLike:
46         domain = self.domain()
47         x = points[..., 0]
48         y = points[..., 1]
49         coord = ((bm.abs(x - domain[1]) < self.eps) &
50                   (bm.abs(y - domain[0]) < self.eps))
51     return coord
52
53     def threshold(self) -> Tuple[Callable, Callable]:
54         return (self.is_dirichlet_boundary_dof_x,
55                 self.is_dirichlet_boundary_dof_y)

```

Appendix C. Cantilever3dData1

```

1  class Cantilever3dData1:
2      def __init__(self,
3          xmin: float=0, xmax: float=60,
4          ymin: float=0, ymax: float=20,
5          zmin: float=0, zmax: float=4,
6          T: float = -1):
7          self.xmin, self.xmax = xmin, xmax
8          self.ymin, self.ymax = ymin, ymax
9          self.zmin, self.zmax = zmin, zmax
10         self.T = T
11         self.eps = 1e-12
12
13     def domain(self) -> list:
14         box = [self.xmin, self.xmax,
15                self.ymin, self.ymax,
16                self.zmin, self.zmax]
17     return box
18
19     @cartesian
20     def force(self, points: TensorLike) -> TensorLike:
21         domain = self.domain()

```

```

22     x = points[..., 0]
23     y = points[..., 1]
24     z = points[..., 2]
25     coord = (
26         (bm.abs(x - domain[1]) < self.eps) &
27         (bm.abs(y - domain[2]) < self.eps)
28     )
29     kwargs = bm.context(points)
30     val = bm.zeros(points.shape, **kwargs)
31     val[coord, 1] = self.T
32     return val
33
34     @cartesian
35     def dirichlet(self, points: TensorLike) -> TensorLike:
36         kwargs = bm.context(points)
37         return bm.zeros(points.shape, **kwargs)
38
39     @cartesian
40     def is_dirichlet_boundary_dof_x(self, points: TensorLike
41         ) -> TensorLike:
42         domain = self.domain()
43         x = points[..., 0]
44         coord = bm.abs(x - domain[0]) < self.eps
45         return coord
46
47     @cartesian
48     def is_dirichlet_boundary_dof_y(self, points: TensorLike
49         ) -> TensorLike:
50         domain = self.domain()
51         x = points[..., 0]
52         coord = bm.abs(x - domain[0]) < self.eps
53         return coord
54
55     @cartesian
56     def is_dirichlet_boundary_dof_z(self, points: TensorLike
57         ) -> TensorLike:
58         domain = self.domain()
59         x = points[..., 0]
60         coord = bm.abs(x - domain[0]) < self.eps
61         return coord
62
63     def threshold(self) -> Tuple[Callable, Callable]:

```

```

61     return (self.is_dirichlet_boundary_dof_x,
62             self.is_dirichlet_boundary_dof_y,
63             self.is_dirichlet_boundary_dof_z)

```

Appendix D. Automatic Differentiation

```

1  def _compute_gradient_auto(self,
2      rho: TensorLike,
3      u: Optional[TensorLike] = None) ->
4          TensorLike:
5
6      if u is None:
7          u = self._update_u(rho)
8
9      ke0 = self.solver.get_base_local_stiffness_matrix()
10     cell2dof = self.solver.tensor_space.cell_to_dof()
11     ue = u[cell2dof]
12
13     def compliance_contribution(rho_i: float, ue_i:
14         TensorLike, ke0_i: TensorLike) -> float:
15         E = self.materials.calculate_elastic_modulus(rho_i)
16         c_i = -E * bm.einsum('i, ij, j', ue_i, ke0_i, ue_i)
17         return c_i
18
19     vmap_grad = bm.vmap(lambda r, u, k:
20         bm.jacrev(lambda x:
21             compliance_contribution(x, u, k))
22         (r))
23     dc = vmap_grad(rho, ue, ke0)
24     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](#)
- [2] M. P. Bendsøe. Optimal shape design as a material distribution problem. *Structural optimization*, 1:193–202, 1989. [5](#)
- [3] M. P. Bendsøe. *Optimization of structural topology, shape, and material*, volume 414. Springer, 1995. [6](#)

- [4] M. P. Bendsøe and O. Sigmund. *Topology optimization by distribution of isotropic material*, pages 1–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. [2](#)
- [5] M. P. Bendsoe and O. Sigmund. *Topology optimization: theory, methods, and applications*. Springer Science & Business Media, 2013. [8](#), [20](#)
- [6] B. Bourdin. Filters in topology optimization. *International journal for numerical methods in engineering*, 50(9):2143–2158, 2001. [9](#)
- [7] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018. [2](#), [36](#)
- [8] 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. [9](#)
- [9] 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. [2](#)
- [10] H. Chung, J. T. Hwang, J. S. Gray, and H. A. Kim. Topology optimization in openmdao. *Structural and multidisciplinary optimization*, 59:1385–1400, 2019. [2](#)
- [11] 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. [2](#)
- [12] C. for Python Data API Standards. Array api standard, version 2023.12. <https://data-apis.org/array-api/2023.12/>, 2023. Accessed April 2025. [11](#)
- [13] 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. [2](#)
- [14] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008. [2](#)
- [15] J. K. Guest. Imposing maximum length scale in topology optimization. *Structural and Multidisciplinary Optimization*, 37:463–473, 2009. [37](#)
- [16] 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. [10](#)
- [17] 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. [2](#)
- [18] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020. [34](#)
- [19] Q. Li, W. Chen, S. Liu, and L. Tong. Structural topology optimization considering connectivity constraint. *Structural and Multidisciplinary Optimization*, 54:971–984, 2016. [37](#)
- [20] A. Meurer, C. Smith, M. Paprocki, O. Čertík, S. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. Moore, S. Singh, T. Rathnayake, S. Vig, B. Granger, R. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, and A. Scopatz. Sympy: Symbolic computing in python, 06 2016. [4](#)
- [21] 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. [2](#)
- [22] 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. [34](#)
- [23] 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. 37
- [24] 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. 37
- [25] O. Sigmund. On the design of compliant mechanisms using topology optimization. *Journal of Structural Mechanics*, 25(4):493–524, 1997. 9
- [26] O. Sigmund. Morphology-based black and white filters for topology optimization. *Structural and Multidisciplinary Optimization*, 33(4):401–424, 2007. 5, 8
- [27] 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. 8
- [28] M. Stolpe and K. Svanberg. An alternative interpolation scheme for minimum compliance topology optimization. *Structural and Multidisciplinary Optimization*, 22(2):116–124, 2001. 37
- [29] 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. 7
- [30] K. Svanberg. Mma and gmma, versions september 2007. 2007. 17, 29
- [31] H. Wei and Y. Huang. Fealpy: Finite element analysis library in python. <https://github.com/weihuayi/fealpy>, Xiangtan University, 2017-2024. 3
- [32] 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. 5