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

Liang He [1], Huayi Wei[2,*], Tian Tian[3], Wang Pengxiang[4]

[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. [5], 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. [11] leveraged OpenMDAO [14], decomposing TO into modular components with automated derivative assembly, enhancing flexibility and extensibility. Gupta et al. [18] developed a parallel-enabled implementation using the FEniCS finite element framework [1], addressing large-scale optimization. Recently, Ferro and Pavanello [12] provided an efficient 51-line TO implementation integrating FEniCS, Dolfin Adjoint, and Interior Point Optimizer, simplifying the development process. These projects provided standardized interfaces for convenient integration with existing 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 [15]. By decomposing complex functions into a series of elementary operations (such as addition and multiplication), AD accurately computes derivatives of arbitrary differentiable functions. In TO, the Jacobian matrices represent sensitivities of objective functions and constraints with respect to design variables, and software can automate this process, relieving developers from manually deriving and implementing sensitivity calculations. With its capability of easily obtaining accurate derivative information, AD offers significant advantages in design optimization, particularly for highly nonlinear optimization problems.

In recent years, the adoption of AD in TO has gradually increased. For instance, Nørgaard et al. [23] employed the AD tools CoDiPack and Tapenade to achieve automatic sensitivity analysis in unsteady flow TO, significantly enhancing computational efficiency. Building upon this, Chandrasekhar [10] utilized the high-performance Python library JAX [8] to apply AD techniques in density-based TO, achieving efficient solutions to classical TO problems, exemplified by compliance minimization..

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 [34]. 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 [22] 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.

Notation. Italic symbols such as ρ, u, f and Greek symbols such as σ, ε denote continuous fields, whereas bold symbols such as $\boldsymbol{\rho}, \mathbf{U}, \mathbf{F}, \mathbf{K}$ denote discrete vectors or matrices arising from the finite-element discretisation.

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 [3]. 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 [35], 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 [29].

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 $u \in H^1(\Omega)$, 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:

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

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

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

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

with

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

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

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

2.3 Optimization Algorithms

The Optimality Criteria (OC) method is a classical TO algorithm widely used for compliance minimization problems 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 [4] proposed a heuristic update scheme for the design variables, given by the following formula:

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

where m is the move limit (maximum change in each iteration) and $\eta \in (0, 1]$ is a damping exponent controlling the step size.

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

Algorithm 2.1 OC pseudo-code

Input: Initial design variables $\boldsymbol{\rho}^{(0)}$ (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 $\boldsymbol{\rho}$
Set $\boldsymbol{\rho}^{(k)} \leftarrow \boldsymbol{\rho}^{(0)}$, $k \leftarrow 0$
while $k < \text{MaxIter}$ *and* $\|\boldsymbol{\rho}^{(k+1)} - \boldsymbol{\rho}^{(k)}\|_\infty > \epsilon$ **do**
    Evaluate the objective function $c(\boldsymbol{\rho}^{(k)})$ and the volume constraint $g(\boldsymbol{\rho}^{(k)})$
    Compute the sensitivities $\nabla c(\boldsymbol{\rho}^{(k)})$ and $\nabla g(\boldsymbol{\rho}^{(k)})$
    (Optional) Apply a filter to the sensitivities
    Update the Lagrange multiplier λ using the bisection method
    Update the design variables $\boldsymbol{\rho}^{\text{new}}$ using the optimality criterion
    (Optional) Apply a filter to $\boldsymbol{\rho}^{\text{new}}$
    Set $\boldsymbol{\rho}^{(k+1)} \leftarrow \boldsymbol{\rho}^{\text{new}}$, $k \leftarrow k+1$
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 [32]. It is particularly well-suited for problems involving multiple constraints or complex objective functions. The core idea of MMA is to dynamically adjust the approximation range of the design variables using "moving asymptotes," thereby transforming the original nonlinear optimization problem into a sequence of convex subproblems. This strategy ensures numerical stability throughout the iterative process.

In the compliance minimization problem, MMA approximates the original problem by the

following convex subproblem:

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

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:

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

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 $\boldsymbol{\rho}^{(0)}$, problem-specific parameters, MMA parameters
Output: Optimized design variables $\boldsymbol{\rho}$
Set $\boldsymbol{\rho}^{(k)} \leftarrow \boldsymbol{\rho}^{(0)}$, $k \leftarrow 0$
while $k <$ MaxIter *and* $\|\boldsymbol{\rho}^{(k+1)} - \boldsymbol{\rho}^{(k)}\|_\infty > \epsilon$ **do**
  Compute sensitivities $\nabla f_0(\boldsymbol{\rho}^{(k)})$ and $\nabla f_i(\boldsymbol{\rho}^{(k)})$, $i = 1,\ldots,m$
  (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 $\tilde{f}_i^{(k)}(\boldsymbol{\rho})$ to form the MMA subproblem
  Solve the MMA subproblem using a primal-dual Newton method to obtain $\boldsymbol{\rho}^{\text{new}}$
  (Optional) Apply filter to $\boldsymbol{\rho}^{\text{new}}$
  Set $\boldsymbol{\rho}^{(k+1)} \leftarrow \boldsymbol{\rho}^{\text{new}}$, $k \leftarrow k+1$
end

2.4 Filtering Methods

In practical computations of TO, pathological issues such as mesh dependency, checkerboard patterns, and local minima are frequently encountered [6]. 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 [30]. A comprehensive overview of various filtering methods was provided by Sigmund [29].

The sensitivity filter was proposed by Sigmund [27] 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 $H_{ij} = \max\{0,\ r_{\min} - \mathrm{dist}(i,j)\}$ is a linearly decaying weight, $N_i = \{j : \mathrm{dist}(i,j) \leq r_{\min}\}$ the neighbour set, $\mathrm{dist}(i,j)$ the centroidal distance, $r_{\min}$ the filter radius, and $\gamma = 10^{-3}$ a small stabilising constant.

The density filter was originally proposed by Bruns and Tortorelli [9], and later mathematically justified by Bourdin [7] 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 [17]. 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 $r_{\min} \to 0$, the effect of the filter diminishes, making the optimized structure prone to local oscillations or checkerboard patterns;
- When $r_{\min} \to \infty$, 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, emphasizing its multi-backend switching capability for improved computational performance and flexibility on central processing units (CPUs) and graphics processing units (GPUs). The chapter is divided into three parts: an overview of the FEALPy architecture as the tensor computation foundation, a description of SOPTX's modular structure (material, solver, filter, and optimization modules), and an analysis of the multi-backend mechanism supporting NumPy, PyTorch, and JAX for diverse computational needs.

3.1 FEALPy Architecture Design

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

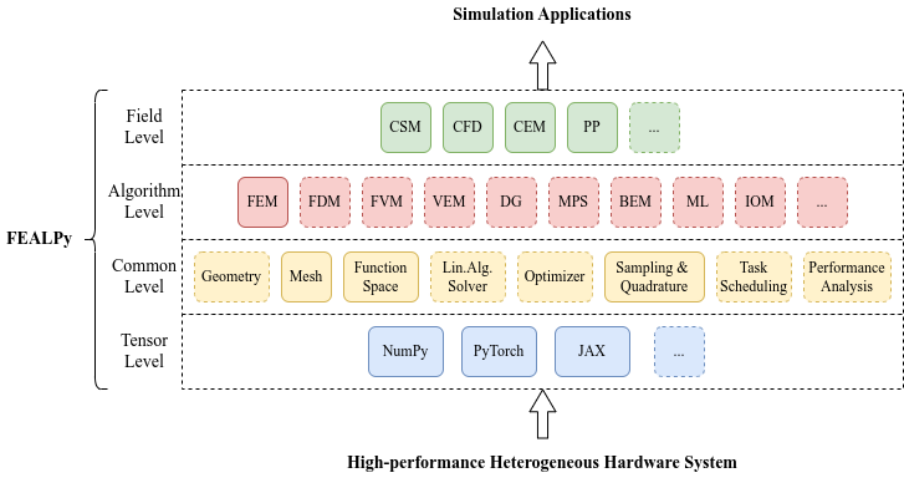


Figure 1: The layered architecture of FEALPy, comprising tensor, common, algorithm, and field levels, progressing from low-level functionalities to high-level applications. Modules in dashed boxes are under development.

Specifically, the functions of each layer are as follows:

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

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

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

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

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

3.2 SOPTX Architecture Design

The SOPTX framework is positioned within the field level of FEALPy's layered architecture, targeting structural topology optimization (STO) applications. SOPTX fully inherits and extends FEALPy's *Tensor Backend Manager*, 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.

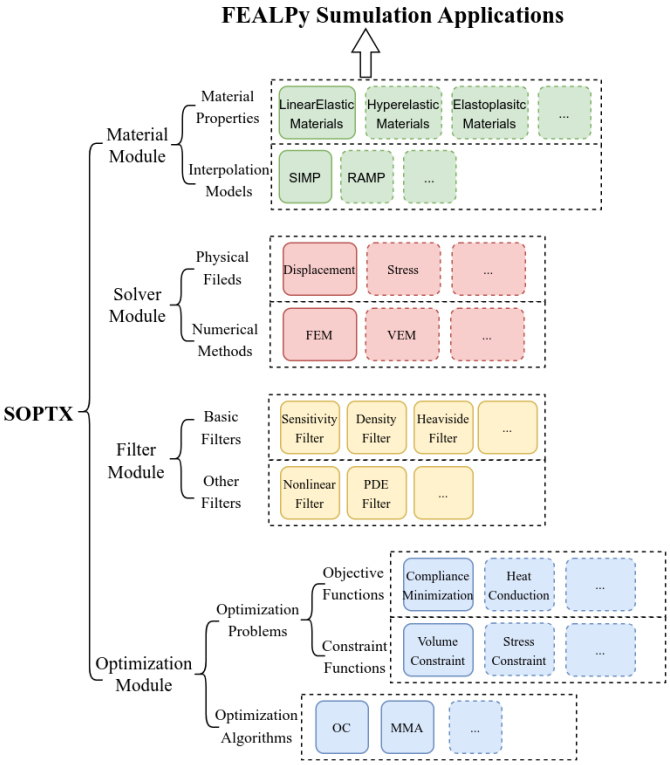


Figure 2: The modular architecture of SOPTX, consisting of material, solver, filter, and optimization modules. The material module serves as the foundation, while the solver and filter modules manage intermediate computations, collectively supporting the optimization module. Components in dashed boxes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

3.2.1 Material Module

The material module in SOPTX currently focuses on linear elastic materials, extending FEALPy's implementation with TO-specific interfaces for computing Lamé constants, elasticity matrices, and strain matrices. It implements the Solid Isotropic Material with Penalization (SIMP) model, typically using a penalization factor $p = 3$, to transform continuous density fields into discrete material distributions. The module's abstract interface design ensures extensibility, allowing future support for models like the Rational Approximation of Material Properties (RAMP) or complex behaviors such as anisotropy, hyperelasticity, and elastoplasticity through subclassing.

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

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

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

3.2.2 Solver Module

The solver module forms the computational core of the SOPTX framework, tasked with solving physical field problems (e.g., displacement fields) based on given material properties, boundary conditions, and external loads. Its design philosophy centers on creating an efficient, flexible, and backend-independent solver engine to address the demands of repeatedly solving large-scale linear systems in TO. To achieve this, the module enhances computational efficiency through optimized matrix assembly, intelligent caching, and multi-backend acceleration, with support for backends such as NumPy, PyTorch, and JAX.

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

1. **Dimensional and element adaptability**: It supports various spatial dimensions, element types, and boundary conditions, with the current version representing the density field as piecewise constant per element and the displacement field using linear finite elements to ensure stability and efficiency
2. **Efficient numerical strategies**: The module implements a fast matrix assembly technique that separates element-independent and element-dependent components to eliminate redundant computations. Additionally, it incorporates symbolic integration to boost efficiency by precomputing exact expressions.

3. **Multiple solution strategies**: It provides both direct solvers (e.g., MUMPS) and iterative solvers (e.g., Conjugate Gradient, CG), with automatic optimization tailored to the backend, such as utilizing GPU acceleration on PyTorch and JAX.

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

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

1. **Interaction with the material module**: It receives material properties, such as elasticity matrices, via standardized interfaces, ensuring the solver remains independent of specific material interpolation schemes.

2. **Output to the optimization module**: It supplies the displacement field for objective function evaluation and the stiffness matrix for sensitivity analysis.

3. **Result reuse**: It employs an intelligent caching mechanism to avoid recomputing invariant components, substantially reducing computational overhead.

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

3.2.3 Filter Module

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

The module implements three key filtering techniques:

1. **Sensitivity filtering**: Applies weighted averaging to objective function sensitivities, effectively suppressing checkerboard patterns.

2. **Density filtering**: Maps raw design variables to physical densities, overcoming locality limitations of sensitivity filtering.

3. **Heaviside projection filtering**: Enhances density filtering with a smoothed Heaviside function, driving intermediate densities toward 0 or 1 for distinct black-and-white designs.

The module employs a KD-tree-based neighborhood search to construct filtering matrices, enabling efficient regularization on unstructured meshes and complex geometries. Optimization

strategies, such as precomputed neighborhoods and sparse matrix representations, ensure scalability for large-scale TO problems.

The filter module is highly extensible. Users can subclass the base filter class to implement custom algorithms (e.g., nonlinear filters or PDE-based filters), inheriting multi-backend compatibility seamlessly. Additionally, the framework supports chained filter combinations, allowing tailored regularization strategies for diverse optimization needs.

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

1. **Solver Module**: Receives raw design variables and outputs filtered physical densities.
2. **Optimization Module**: Filters unprocessed sensitivities to maintain numerical stability during optimization
3. **Material Module**: Provides filtered densities for material interpolation, forming a streamlined data flow:

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

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

3.2.4 Optimization Module

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

The module currently focuses on compliance minimization under volume constraints, a foundational problem in TO, and supports two mainstream algorithms: Optimality Criteria (OC), a lightweight method for volume-constrained problems, and Method of Moving Asymptotes (MMA), based on Svanberg's 2007 formulation [33], adapted for multi-backend compatibility and optimized for efficiency.

The design of the optimization module follows two core principles

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

The optimization module is designed with high extensibility, allowing users to

1. **Custom problems**: Users can extend the module by subclassing the base problem class to introduce new objectives (e.g., heat conduction, compliant mechanisms) or constraints (e.g., stress, frequency).

2. **Algorithm integration**: Additional algorithms, such as Sequential Quadratic Programming (SQP) or Sequential Linear Programming (SLP), can be incorporated modularly.

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

1. **Solver Module**: Utilizes physical outputs (e.g., displacement fields, stiffness matrices) to compute objectives and sensitivities.

2. **Filter Module**: Sends raw sensitivities for processing, receives filtered sensitivities for updates, and applies filtering to design variables to derive physical densities.

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

3.3 Multi-Backend Switching

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

Specifically, each backend offers distinct advantages for different scenarios

- **NumPy backend**: Ideal for small-scale tasks and rapid prototyping, providing stable performance and efficient memory management on standard CPU platforms.

- **PyTorch and JAX backends**: Both support GPU acceleration and AD, making them suitable for large-scale or high-dimensional problems. AD streamlines sensitivity analysis, while JAX offers just-in-time (JIT) compilation and automatic vectorization, which can further enhance performance on GPU platforms.

4 Getting Started with SOPTX

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

4.1 Software Installation

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

Core installation steps:

1. Clone the FEALPy repository from GitHub:

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

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

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

3. Similarly, install SOPTX:

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

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

4.2 Example: 2D Cantilever Beam

We use a popular compliance minimization benchmark to demonstrate the usage of SOPTX: minimizing the structural compliance of a cantilever beam under tip loading (see Figure 3) [6]. The left end of the beam is fixed, and a downward concentrated load $T=-1$ is applied to the bottom of the right end. A 160×100 uniform quadrilateral mesh is used. The target volume fraction is set to 0.4, with material properties $E=1$ and $\nu=0.3$. The penalization factor is $p=3$, and the sensitivity filter radius is $r=6.0$, which matches the mesh element size to ensure structural smoothness and eliminate checkerboard patterns.

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

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

Code 1: Module imports and PDE model

```
from fealpy.backend import backend_manager as bm
from fealpy.mesh import UniformMesh2d
from fealpy.functionspace import LagrangeFESpace,
    TensorFunctionSpace

from soptx.material import DensityBasedMaterialConfig,
    DensityBasedMaterialInstance
from soptx.solver import ElasticFEMSolver, AssemblyMethod
from soptx.filter_ import SensitivityBasicFilter
from soptx.opt import ComplianceObjective, ComplianceConfig,
    VolumeConstraint, VolumeConfig
from soptx.opt import OCOptimizer
from soptx.pde import Cantilever2dData1

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

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

Code 2: Mesh and function space definitions

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

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

Code 3: Material module

```
material_config = DensityBasedMaterialConfig(
                  elastic_modulus=1.0, minimal_modulus=1e-9,
                  poisson_ratio=0.3,
                  plane_assumption="plane_stress",
                  interpolation_model="SIMP",
                  penalty_factor=3.0)
materials = DensityBasedMaterialInstance(config=
    material_config)
```

Subsequently, the solver module is initialized with the material module, PDE model, standard matrix assembly, and a direct solver (MUMPS), as shown in Code 4. A sensitivity filter with radius $r = 6.0$ is also set up to regularize the optimization.

Code 4: Solver and filter module

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

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

Code 5: Optimization module

```
objective = ComplianceObjective(solver=solver)
constraint = VolumeConstraint(solver=solver, volume_fraction
    =0.4)
```

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

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

Code 6: Main program and post-processing

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

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

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

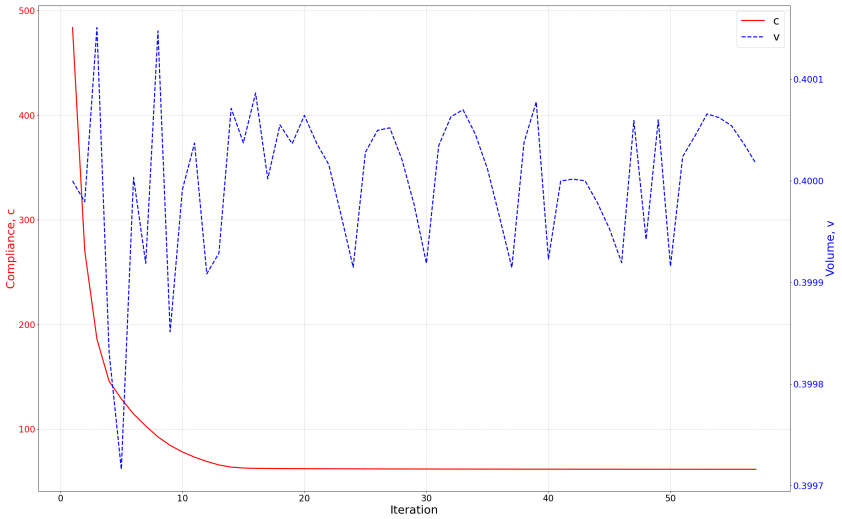
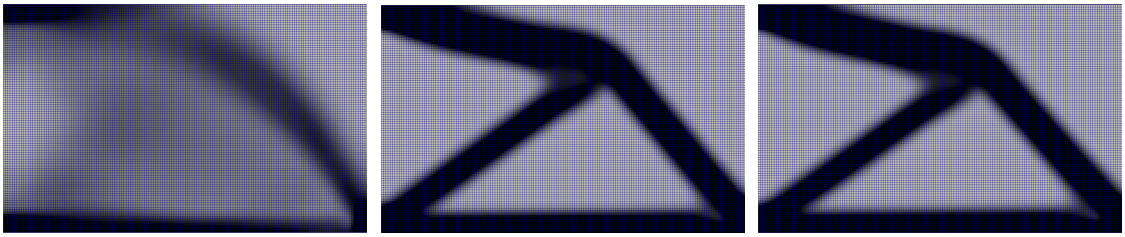


Figure 4: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam initialized with a uniform density of 0.4.

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



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

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

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

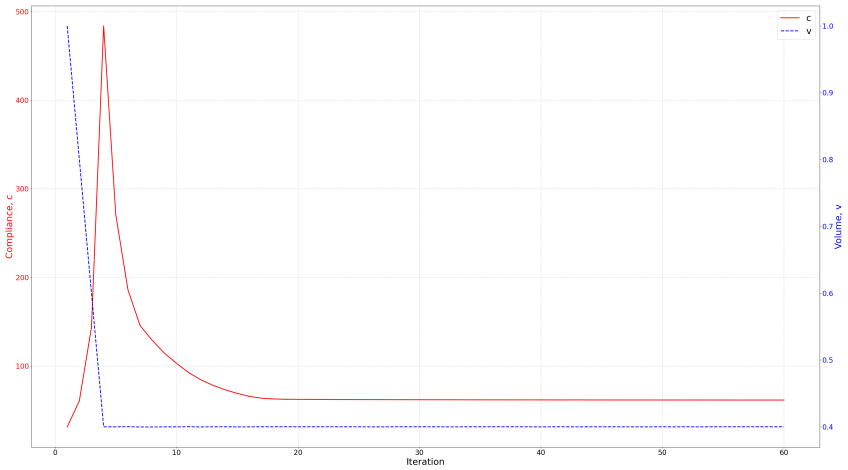
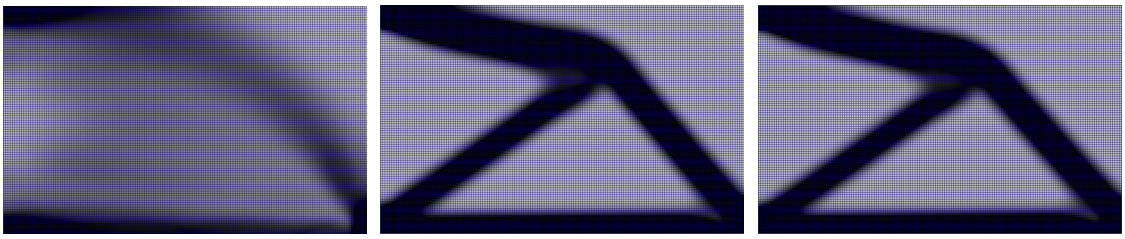


Figure 6: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam with an initial density of 1.

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



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

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

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

5 Numerical Examples

This chapter presents numerical examples to validate the SOPTX framework's capabilities in topology optimization (TO), spanning 2D MBB beam problems to 3D cantilever structures. These examples demonstrate SOPTX's versatility across diverse partial differential equation (PDE) models, meshes, filters, and algorithms, while showcasing advanced features like fast matrix assembly, automatic differentiation (AD), and multi-backend switching.

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

5.1 MBB Beam

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

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

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

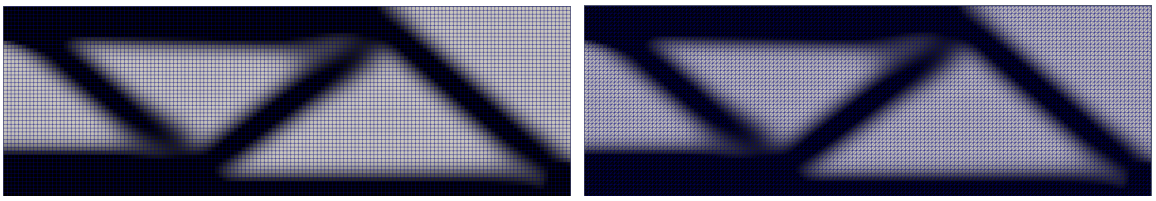
The complete model definition is provided in Appendix Appendix B.

To demonstrate SOPTX's adaptability across mesh types, we perform TO on a 150×50 uniform quadrilateral mesh and a triangular mesh, both initialized with a uniform material density

equal to the target volume fraction 0.5. The triangular mesh is generated using:

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

The optimized topologies, shown in Figure 9, are consistent across both mesh types, with compliance and volume fraction differences below 1%. This consistency underscores SOPTX's mesh-independence and algorithmic robustness, facilitated by its modular design, which allows easy switching between PDE models and mesh configurations.



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

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

5.2 Different Filtering Methods

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

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 $r = 6.0$.

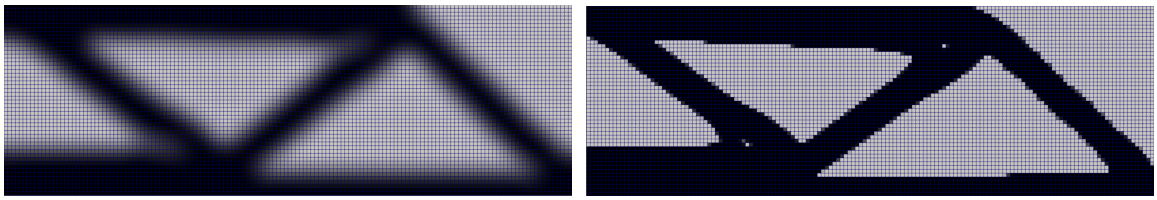
The density filter performs weighted averaging of the design variables to achieve a smooth spatial distribution, effectively eliminating small-scale features. This method is well-suited for design tasks that require structural continuity. In SOPTX, the density filter can be applied simply by instantiating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

```
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:

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

Figure 10 shows the optimized topologies obtained using both filters:



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

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

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

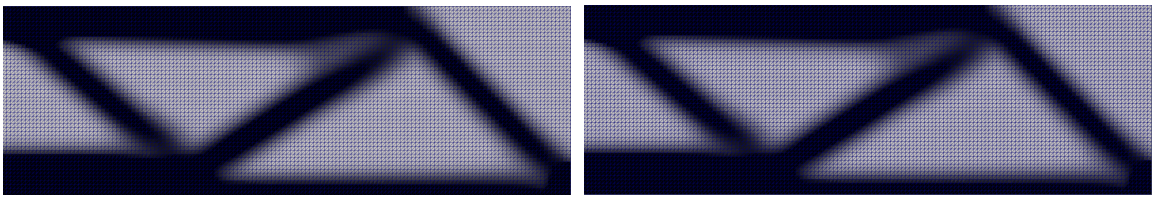
In TO, the choice of optimization algorithm directly affects both computational efficiency and the quality of the resulting design. SOPTX adopts a modular design that enables seamless switching between different optimizers, allowing users to easily tailor optimization strategies to specific design requirements. 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:

```
from soptx.opt import MMAOptimizer

optimizer = MMAOptimizer(objective=objective,
                         constraint=constraint,
                         filter_=sens_filter,
                         options={'max_iterations': 200,
                                  '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.



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

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

The MMA algorithm in SOPTX is based on the classical MATLAB implementation provided by Krister Svanberg [33], 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 `xmin` and `xmax`, as well as internal control parameters such as `a0`, `a`, `c`, and `d`:

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

```
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.

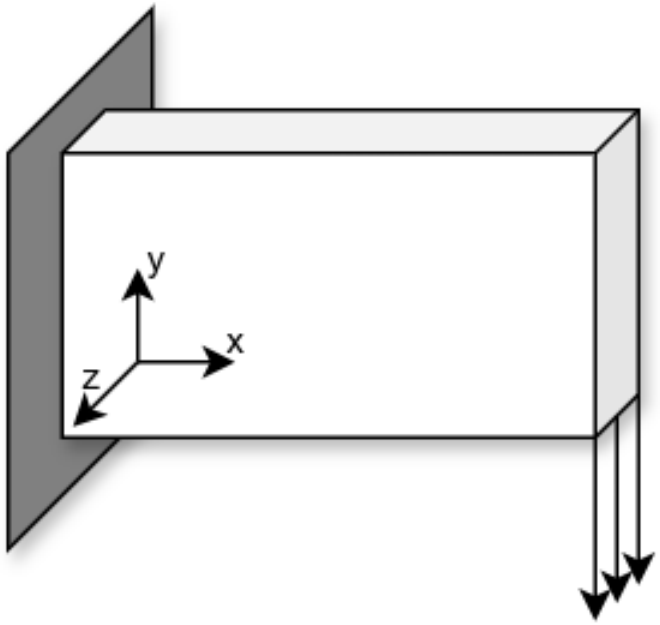


Figure 12: Geometric configuration of the 3D cantilever beam. The left end is fully fixed, and a downward concentrated load is applied at the bottom of the right end.

Compared to the 2D cantilever beam example in Section 4.2, SOPTX can be seamlessly ex-

tended 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

```
from soptx.pde import Cantilever3dData1
pde = Cantilever3dData1(xmin=0, xmax=60, ymin=0, ymax=20,
zmin=0, zmax=4, T = -1)

from fealpy.mesh import UniformMesh3d
mesh = UniformMesh3d(extent=[0, 60, 0, 20, 0, 4], h=[1,
1, 1], origin=[0, 0, 0])
```

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.

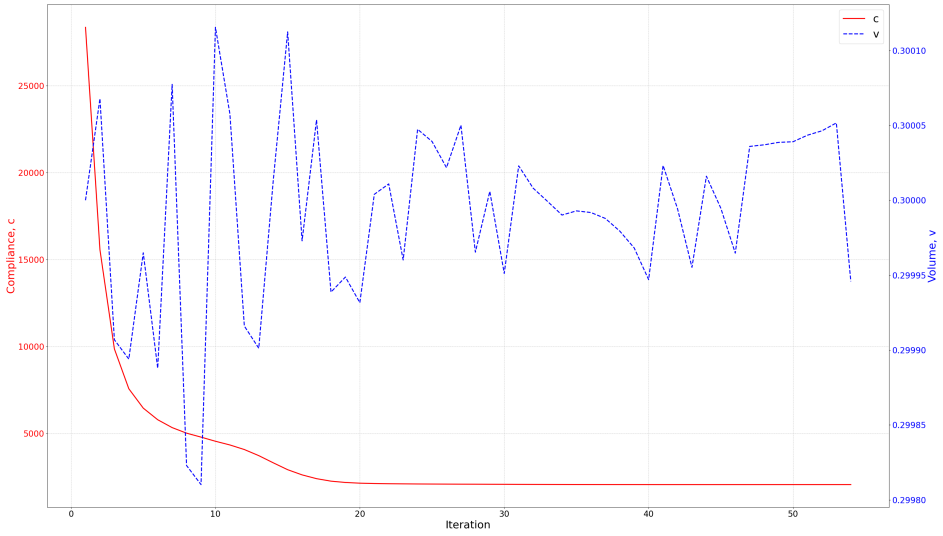
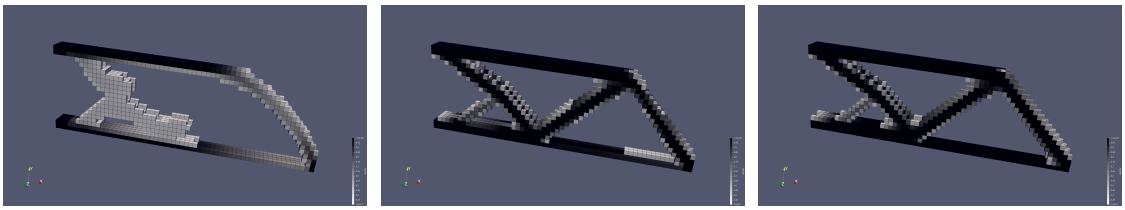


Figure 13: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam initialized with a uniform density of 0.3.

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.



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

Figure 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.

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 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.

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

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

As shown in Table 1, the original 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. 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 [19], 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 [24] requires only a single line of code:

```
    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:

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

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

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.

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. he compliance converges to approximately 2063.5625 after 54 iterations, and the volume fraction remains stably around 0.3. These results confirm that AD accurately computes the sensitivities and effectively guides the optimization process.

To evaluate the computational efficiency of AD, we compare the optimization times of AD and manual differentiation under the same 3D cantilever beam problem setting. The results are summarized in Table 2.

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

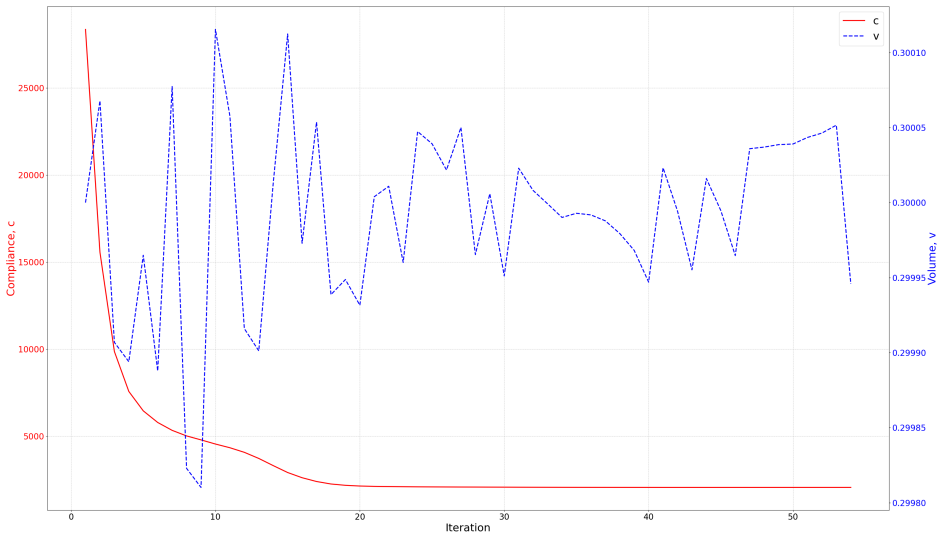
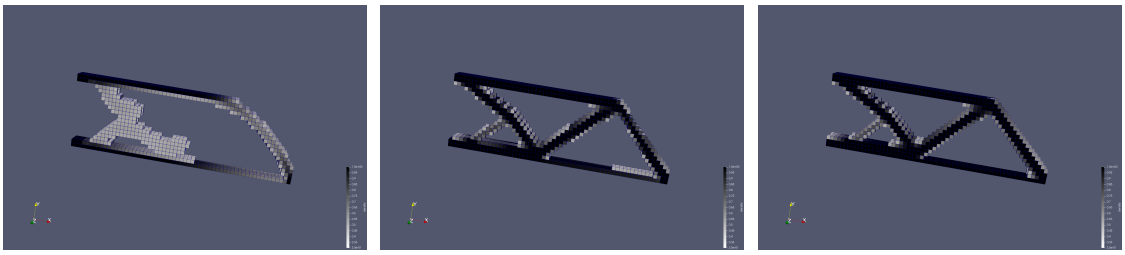


Figure 15: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam optimization using automatic differentiation.



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

Figure 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.

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

computational backends. In addition to PyTorch, SOPTX also supports the JAX backend [8], which offers equally powerful AD capabilities. The sensitivity computation code remains identical when using JAX, users simply need to switch to the JAX backend by executing:

```
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.

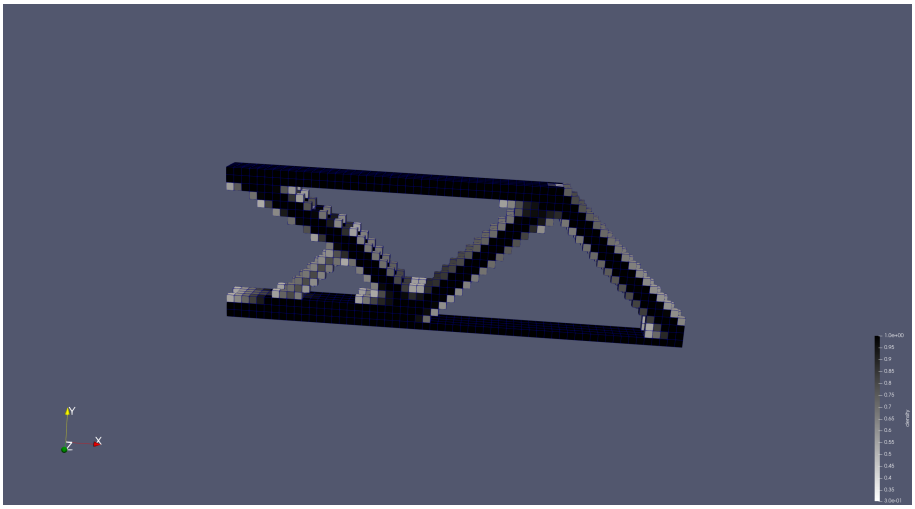


Figure 17: Final optimized topology of the 3D cantilever beam using AD with the JAX backend.

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:

- **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 [31] 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 [16], connectivity [20], overhang limitations [25], and material usage restrictions [26] 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.

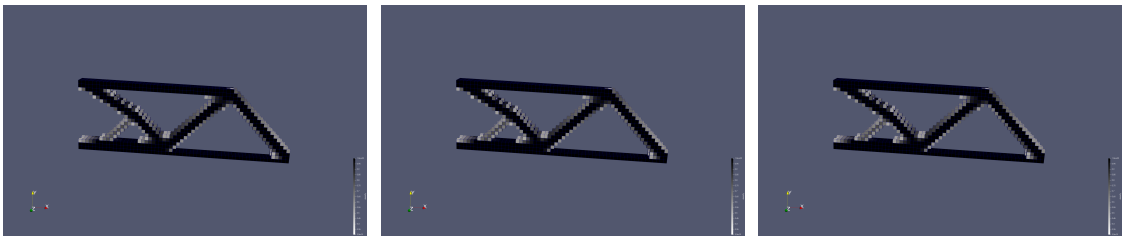
5.7 Multi-Backend Switching

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:

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

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.



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

Figure 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.

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:

```
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:

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

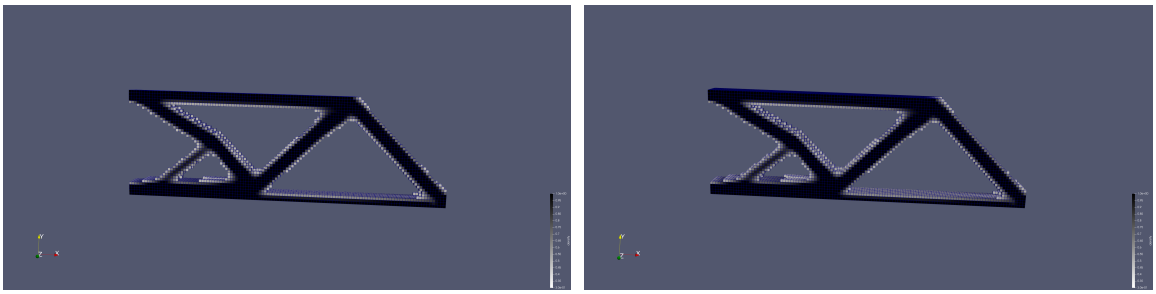
   By specifying `device='cuda'` during mesh creation, all computations related to that mesh are performed on the GPU. This approach offers greater flexibility, allowing users to mix CPU and GPU computations within the same program if needed.

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:



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

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

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 fraction ($v(\rho)$)—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

```
class Cantilever2dData1:
    def __init__(self,
                 xmin: float, xmax: float,
                 ymin: float, ymax: float,
```

```
                 T: float = -1):
    self.xmin, self.xmax = xmin, xmax
    self.ymin, self.ymax = ymin, ymax
    self.T = T
    self.eps = 1e-12

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

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

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

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

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

```
        coord = bm.abs(x - domain[0]) < self.eps
        return coord

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

Appendix B MBBBeam2dData1

```
class MBBBeam2dData1:
def __init__(self,
             xmin: float=0, xmax: float=60,
             ymin: float=0, ymax: float=20,
             T: float = -1):
    self.xmin, self.xmax = xmin, xmax
    self.ymin, self.ymax = ymin, ymax
    self.T = T
    self.eps = 1e-12

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

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

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

```
31
32 @cartesian
33 def is_dirichlet_boundary_dof_x(self, points: TensorLike) ->
       TensorLike:
34     domain = self.domain()
35     x = points[..., 0]
36     coord = bm.abs(x - domain[0]) < self.eps
37     return coord
38
39 @cartesian
40 def is_dirichlet_boundary_dof_y(self, points: TensorLike) ->
       TensorLike:
41     domain = self.domain()
42     x = points[..., 0]
43     y = points[..., 1]
44     coord = ((bm.abs(x - domain[1]) < self.eps) &
45              (bm.abs(y - domain[0]) < self.eps))
46     return coord
47
48 def threshold(self) -> Tuple[Callable, Callable]:
49     return (self.is_dirichlet_boundary_dof_x,
50             self.is_dirichlet_boundary_dof_y)
```

Appendix 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,
```

```
            self.ymin, self.ymax,
            self.zmin, self.zmax]
            return box

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

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

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

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

        @cartesian
```

```
54          def is_dirichlet_boundary_dof_z(self, points:
    TensorLike) -> TensorLike:
55              domain = self.domain()
56              x = points[..., 0]
57              coord = bm.abs(x - domain[0]) < self.eps
58              return coord
59
60          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) ->
    TensorLike:
4          if u is None:
5              u = self._update_u(rho)
6
7          ke0 = self.solver.get_base_local_stiffness_matrix()
8          cell2dof = self.solver.tensor_space.cell_to_dof()
9          ue = u[cell2dof]
10
11         def compliance_contribution(rho_i: float, ue_i:
    TensorLike, ke0_i: TensorLike) -> float:
12             E = self.materials.calculate_elastic_modulus(
    rho_i)
13             c_i = -E * bm.einsum('i, ij, j', ue_i, ke0_i,
    ue_i)
14             return c_i
15
16         vmap_grad = bm.vmap(lambda r, u, k:
17                             bm.jacrev(lambda x:
    compliance_contribution(x, u, k))
18                             (r))
19         dc = vmap_grad(rho, ue, ke0)
20         return dc
```

References

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

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

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

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

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

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

[7] B. Bourdin. Filters in topology optimization. *International journal for numerical methods in engineering*, 50(9):2143–2158, 2001.

[8] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018.

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

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

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

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

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

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

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

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

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

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

[19] 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.

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

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

[22] A. Meurer, C. 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

[33] K. Svanberg. Mma and gcmma, versions september 2007. 2007.

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

[35] 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.