# 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.** In recent years, topology optimization (TO) has gained widespread attention in both industry and academia as an ideal structural design method. However, its application has high barriers to entry due to the deep expertise and extensive development work typically required. Traditional numerical methods for TO are tightly coupled with computational mechanics methods such as finite element analysis (FEA), making the algorithms intrusive and requiring comprehensive understanding of the entire system. This paper presents SOPTX, a TO package based on FEALPy, which implements a modular architecture that decouples analysis from optimization, supports multiple computational backends (NumPy, PyTorch, JAX), and achieves a non-intrusive design paradigm. Core innovations include: (1) cross-platform backend integration with automatic differentiation (AD), enabling seamless computation across diverse hardware and simplifying sensitivity analysis for complex TO problems; (2) fast matrix assembly techniques that overcome the performance bottlenecks of traditional numerical integration methods, significantly accelerating finite element computations and enhancing overall efficiency; (3) a modular framework supporting TO problems for arbitrary dimensions and meshes, allowing flexible configuration and extensibility of optimization workflows through a rich library of composable components. Using the density-based method for the classic compliance minimization problem with volume constraints as an example, numerical experiments demonstrate SOPTX's high efficiency in computational speed and memory usage, while showcasing its strong potential for research and engineering applications.

**AMS subject classifications**: 65N30, 35Q60

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

*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)

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

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

Meanwhile, efforts in open-source software development have also advanced TO practices. Chung et al. [10] leveraged OpenMDAO [13], decomposing TO into modular components with automated derivative assembly, enhancing flexibility and extensibility. Gupta et al. [17] developed a parallel-enabled implementation using the FEniCS finite element framework [1], addressing large-scale optimization. Recently, Ferro and Pavanello [11] provided an efficient 51-line TO implementation integrating FEniCS, Dolfin Adjoint, and Interior Point Optimizer, simplifying the development process. These projects provided standardized interfaces for convenient integration with existing FEA tools, reducing implementation complexity. However, despite improvements, existing open-source TO packages still face limitations in terms of functionality extensibility and flexibility, particularly for complex engineering applications.

To accelerate the development of TO, automating sensitivity analysis has become a critical step. This involves automatically computing derivatives of objectives, constraints, material models, projections, filters, and other components with respect to the design variables. Currently, the common practice involves manually calculating sensitivities, which, despite not being theoretically complex, can be tedious and error-prone, often becoming a bottleneck in the development of new TO modules and exploratory research. Automatic differentiation (AD) provides an efficient and accurate approach for evaluating derivatives of numerical functions [14]. By decomposing complex functions into a series of elementary operations (such as addition and multiplication), AD accurately computes derivatives of arbitrary differentiable functions. In TO, the Jacobian matrices represent sensitivities of objective functions and constraints with respect to design variables, and software can automate this process, relieving developers from manually deriving and implementing sensitivity calculations. With its capability of easily obtaining accurate derivative information, AD offers significant advantages in design optimization, particularly for highly nonlinear optimization problems.

In recent years, the adoption of AD in TO has gradually increased. For instance, Nørgaard et al. [21] employed the AD tools CoDiPack and Tapenade to achieve automatic sensitivity anal-

ysis in unsteady flow TO, significantly enhancing computational efficiency. Building upon this, Chandrasekhar [9] utilized the high-performance Python library JAX [7] to apply AD techniques in density-based TO, achieving efficient solutions to classical TO problems, exemplified by compliance minimization..

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

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

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

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

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

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

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

Building on the strengths of FEALPy, the SOPTX framework achieves a highly modular design. SOPTX adopts the component-based philosophy commonly used in MDO. Complex systems

are decomposed into independent and reusable modules to enhance flexibility and maintainability. Specifically, SOPTX introduces a clear separation between analysis and optimization, with numerical solvers, AD tools, and optimization algorithms implemented as independent, interchangeable modules. This architecture allows users to flexibly select or replace components based on specific needs without substantial changes to the overall code structure. Through this modular design, SOPTX inherits the efficiency and flexibility of FEALPy while significantly enhancing extensibility and reusability, providing strong support for complex engineering applications.

Leveraging this modular foundation, SOPTX seamlessly integrates AD into the topology optimization workflow. It supports multiple computational backends and can automatically switch between them based on hardware availability and performance requirements. Moreover, SOPTX mitigates computational bottlenecks associated with traditional numerical integration through a fast matrix assembly technique, which separates element-dependent and element-independent components, avoiding redundant computations during iterative procedures. Symbolic integration via SymPy [20] further reduces computational cost while improving accuracy. Maintaining extensibility and reusability, SOPTX adopts a non-intrusive design, enabling users to easily replace or augment filters, constraints, and objectives without modifying the core framework. Overall, SOPTX is designed as a low-barrier, high-performance platform for TO research and engineering applications, empowering developers to focus on algorithmic innovation and problem modeling rather than implementation details. These features make SOPTX not only well-suited for education and research, but also capable of supporting complex and multiphysics-coupled scenarios in TO and MDO.

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

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

In this chapter, we first introduce the density-based method widely employed in topology optimization (TO), elucidating the fundamental idea of optimizing material distribution through the definition of a material density function and interpolation models. Next, taking the compliance minimization problem as an example, we explicitly present both continuous and discrete mathematical formulations of this method. Subsequently, we provide detailed descriptions of two classical optimization algorithms commonly used in TO: the Optimality Criteria (OC) method and the Method of Moving Asymptotes (MMA), including their algorithmic implementations and char-

acteristics. 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 $\rho, u, f$ and Greek symbols such as $\sigma, \varepsilon$ 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 [2]. Among these, the power-law interpolation model has been widely adopted due to its implicit penalization of intermediate densities. This method, known as the Solid Isotropic Material with Penalization (SIMP) approach [32], establishes a power-law relationship between the material density $\rho$ 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 $\rho$ approaches zero, the standard SIMP model causes the element stiffness matrix to approach zero, potentially leading to singularities in the global stiffness matrix. To address this issue, the modified SIMP model introduces a minimum Young's modulus $E_{\min}$:

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

where $E_{\min} = 10^{-9} E_0$. This modification ensures that the stiffness matrix remains positive definite at $\rho = 0$, preventing numerical failure [26].

Although the SIMP method suppresses intermediate densities through the penalization factor, optimization results may still contain "gray areas", i.e., regions where the density is neither 0 nor 1, lacking clear physical correspondence to either material or void regions, potentially causing manufacturing difficulties. In practical applications, the SIMP approach is often combined with

Heaviside projection filters to achieve a distinct $0-1$ topology. Additionally, sensitivity or density filtering techniques are typically employed alongside the SIMP model to mitigate numerical instabilities, such as checkerboard patterns.

## 2.2 Compliance Minimization Problem

The goal of compliance minimization with a volume constraint is to find a material density field $\rho(x) \in [0,1]$ that minimizes the total strain energy of an elastic body occupying a bounded domain $\Omega \subset \mathbb{R}^d$ $(d = 2,3)$ under prescribed loads and boundary conditions, while the available material does not exceed a given volume $V^*$. Throughout this paper we assume the displacement $u \in H^1(\Omega)$, 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 $\Omega$ 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 $\rho_e$ 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 $\lambda$ is the Lagrange multiplier associated with the volume constraint.

Bendsøe [3] proposed a heuristic update scheme for the design variables, given by the following formula:

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

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 $\eta$, maximum iterations MaxIter, convergence tolerance $\epsilon$
**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**
    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 $\lambda$ 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 [29]. It is particularly well-suited for problems involving multiple constraints or complex objective functions. The core idea of MMA is to dynamically adjust the approximation range of the design variables using "moving asymptotes," thereby transforming the original nonlinear optimization problem into a sequence of convex subproblems. This strategy ensures numerical stability throughout the iterative process.

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

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

where $m$ is the number of constraints, $n$ is the number of design variables, and $\rho_j$ 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 $\rho_j$ 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 $i \ge 1$ 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** $k < \text{MaxIter}$ *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 [5]. These numerical difficulties can lead to suboptimal or unstable designs. Filtering techniques serve as an important form of regularization by smoothing the spatial distribution of either design variables or sensitivities, thereby suppressing numerical oscillations and improving both the reliability and physical realizability of the optimized structures [27]. A comprehensive overview of various filtering methods was provided by Sigmund [26].

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

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

where $H_{ij} = \max\{0,\ r_{\min} - \text{dist}(i,j)\}$ is a linearly decaying weight, $N_i = \{j : \text{dist}(i,j) \leq r_{\min}\}$ the neighbour set, $\text{dist}(i,j)$ 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 [8], and later mathematically justified by Bourdin [6] as a valid regularization technique. The basic form of the filtered density function is defined as:

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

where $\tilde{\rho}_i$ is the filtered physical density, $\rho_j$ 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 $\rho_i$, 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 $\psi$ with respect to the design variable $\rho_j$ 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:

- $\psi$ 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 $\rho_j$, i.e., all elements $i$ for which $\rho_j \in N_i$.

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

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

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

where the parameter $\beta$ 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 $\beta$ 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 [12], enabling seamless adaptation across various software and hardware platforms.

Specifically, the functions of each layer are as follows:

1. **Tensor level**: Provides core tensor operations and manages backends (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.

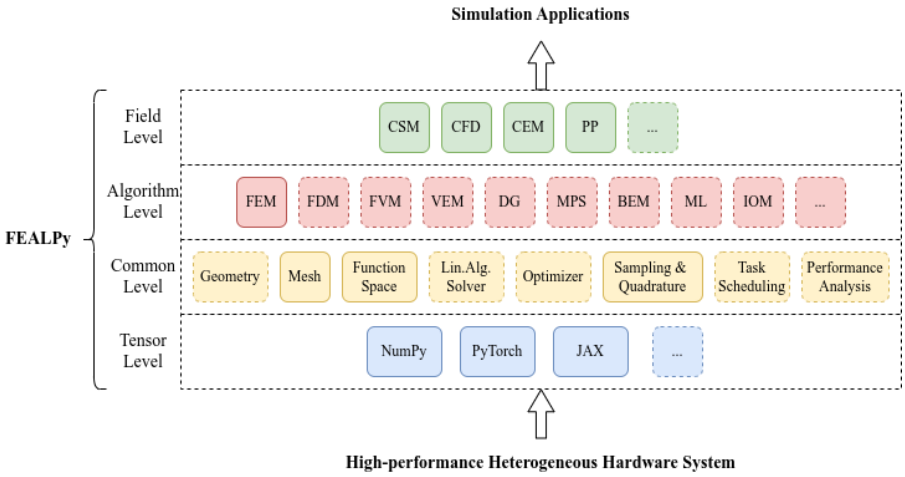



Figure 1: The layered architecture of FEALPy, comprising tensor, common, algorithm, and field levels, progressing from low-level functionalities to high-level applications. Modules in dashed boxes are under development.

## 3.2 SOPTX Architecture Design

The SOPTX framework is positioned within the field level of FEALPy's layered architecture, targeting structural topology optimization (STO) applications. SOPTX fully inherits and extends FEALPy's *Tensor Backend Manager*, diverse numerical algorithm components, and general-purpose mesh and geometry handling capabilities. This design not only ensures flexibility and extensibility, but also significantly improves computational efficiency for TO problems.

As shown in Figure 2, SOPTX adopts a modular architecture consisting of four primary components: the material, solver, filter, and optimizer modules. These components communicate and share data through well-defined interfaces, forming a loosely coupled and easily extensible multi-backend framework for TO.

### 3.2.1 Material Module

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:

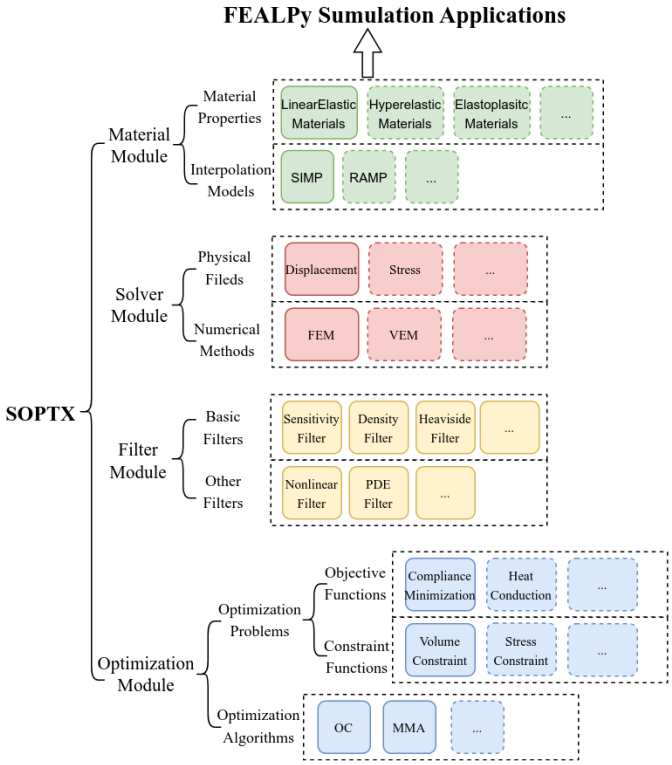



Figure 2: The modular architecture of SOPTX, consisting of material, solver, filter, and optimization modules. The material module serves as the foundation, while the solver and filter modules manage intermediate computations, collectively supporting the optimization module. Components in dashed boxes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

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 [30], 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) [5]. 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 $160\times100$ 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 Solid Isotropic Material with Penalization (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.

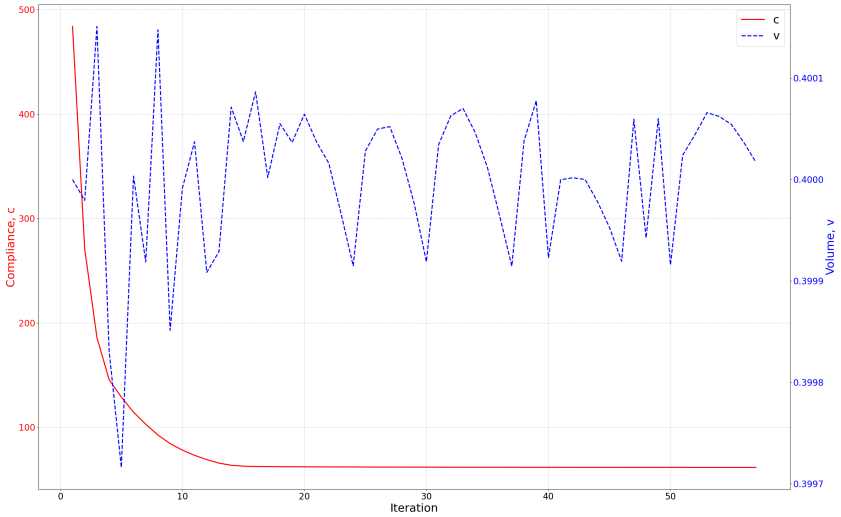
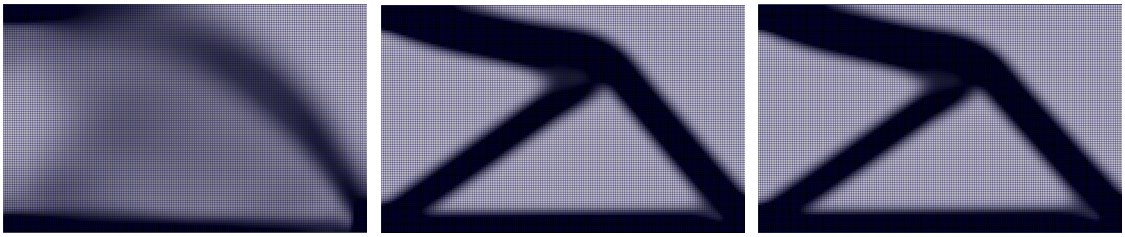



Figure 4: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam initialized with a uniform density of 0.4.

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 7 shows the topology layouts at iterations 3, 31, and 60.

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

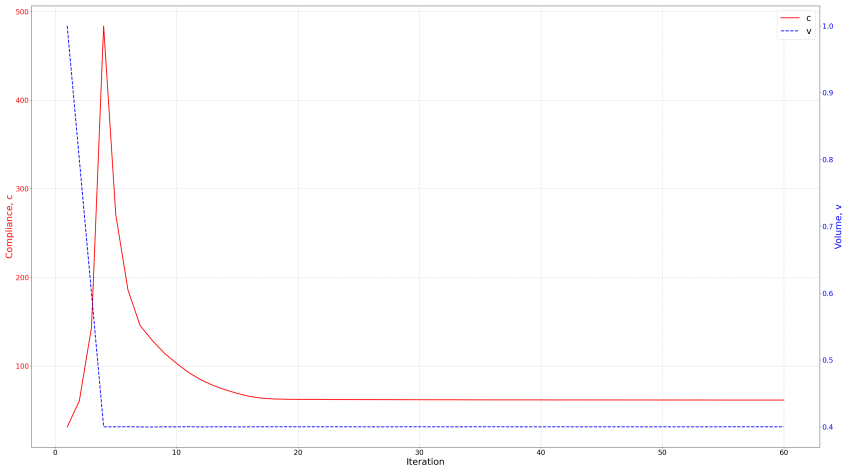
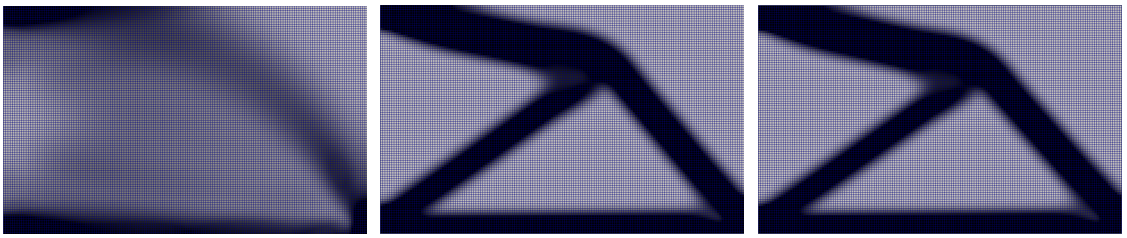



Figure 6: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam with an initial density of 1.



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

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

# 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:

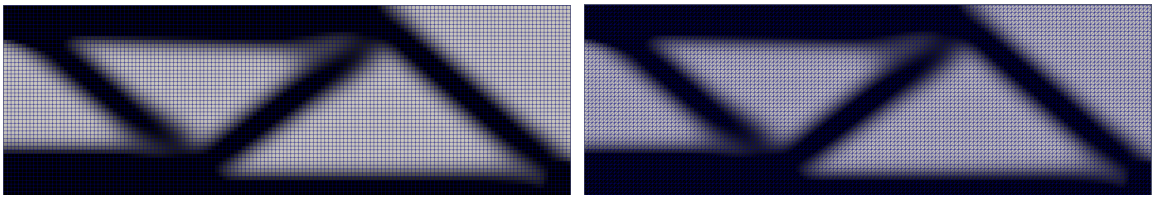
```
from soptx.pde import MBBBeam2dData1
pde = MBBBeam2dData1(xmin=0, xmax=150, ymin=0, ymax=50, T=-1)
```

The complete model definition is provided in Appendix Appendix B.

To demonstrate SOPTX's adaptability across mesh types, we perform TO on a $150 \times 50$ 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.



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

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

## 5.2 Different Filtering Methods

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

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

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

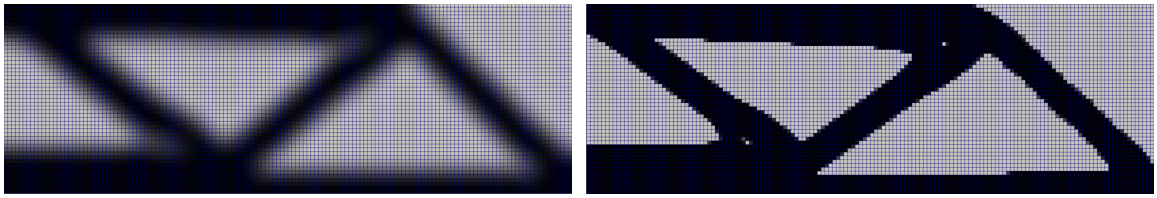
```
dens_filter = DensityBasicFilter(mesh=mesh, rmin=6.0)
```

In contrast, the Heaviside projection filter builds on the density filter by adding a projection step. It gradually increases the projection parameter $\beta$ to drive variables toward 0 or 1, yielding a

clear black-and-white topology suited for strict manufacturability. To avoid overly thick structures early on, the filter radius is reduced to $r = 4.5$. It is enabled in SOPTX with:

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

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



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

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

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

### 5.3 Different Optimization Algorithms

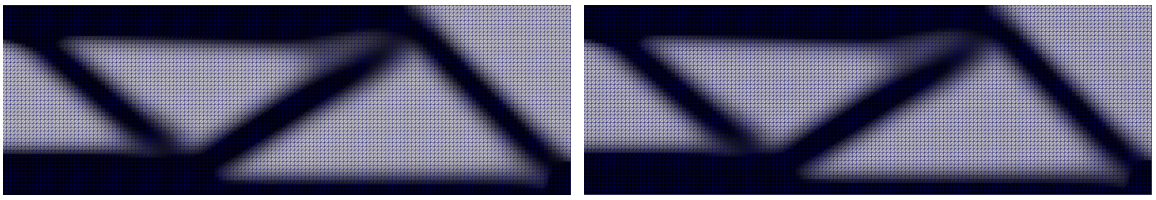
In TO, the choice of optimization algorithm impacts both computational efficiency and design quality. SOPTX's modular design allows seamless switching between optimizers, such as from the Optimality Criteria (OC) method to the more versatile Method of Moving Asymptotes (MMA). This section demonstrates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

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

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

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

SOPTX's MMA, refactored from Krister Svanberg's implementation [30], allows users to adjust internal parameters, including `m`, `n`, `xmin`, `xmax`, and control parameters like `a0`, `a`, `c`, `d`. This provides greater flexibility than traditional *black-box* versions, enhancing performance and adaptability across platforms. An example configuration is shown below:



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

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

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

Through this 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

SOPTX's modular design enables a smooth transition from 2D to 3D TO. This section showcases its capability with a 3D cantilever beam compliance minimization problem (see Figure 12). The beam is fixed on the left and bears a downward load $T = -1$ at the bottom right. A $60 \times 20 \times 4$ hexahedral mesh is applied, with a target volume fraction of 0.3, material properties $E = 1$ and $\nu = 0.3$, and a Solid Isotropic Material with Penalization (SIMP) penalty factor $p = 3$. A sensitivity filter (radius $r = 1.5$) ensures smooth results.

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

```
from soptx.pde import Cantilever3dData1
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 complete definition of the `Cantilever3dData1` model is available in Appendix Appendix C.

The initial material density is uniformly set to the target volume fraction of 0.3. Using the

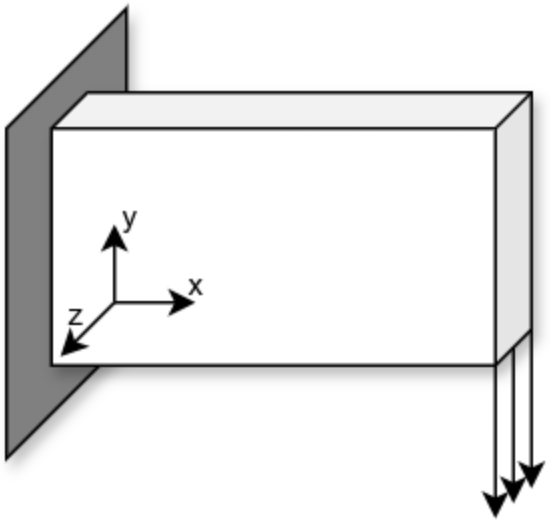



Figure 12: Geometric configuration of the 3D cantilever beam. The left end is fully fixed, and a downward concentrated load is applied at the bottom of the right end.

OC optimizer and a sensitivity filter with radius $r = 1.5$, the compliance $c(\rho)$ shows a rapid initial decrease, followed by a gradual convergence with slight fluctuations, stabilizing at around 2,000 by iteration 54 (see Figure 13). The volume fraction remains nearly constant around 0.3 throughout the process, indicating effective constraint control.

To better visualize the optimization results, Figure 14 shows the topology layouts at iterations 7, 21, and 54. Only elements with density values $\rho \geq 0.3$ are rendered to highlight the structural features.

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

## 5.5 Application of Fast Matrix Assembly

In 3D TO problems, computational efficiency is critical. For the 3D cantilever beam example (Section 5.4), the mesh of $60 \times 20 \times 4$ elements leads to 19,215 displacement degrees of freedom and 4,800 density variables, requiring 54 iterations. Traditional matrix assembly in finite element methods involves redundant computations, creating a bottleneck.

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

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

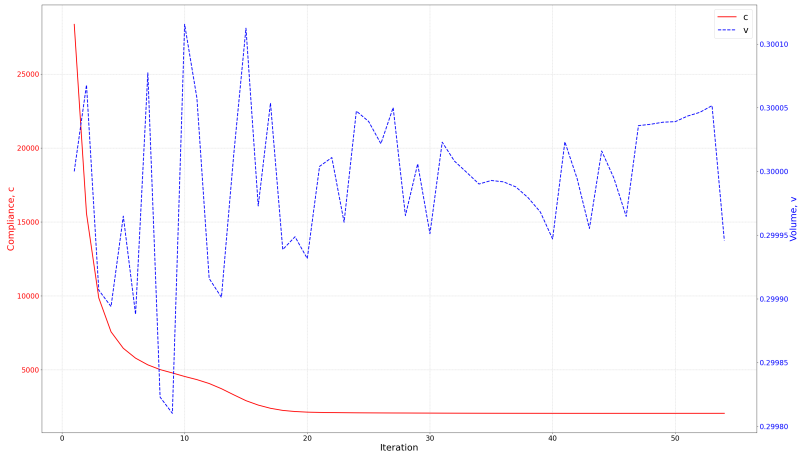
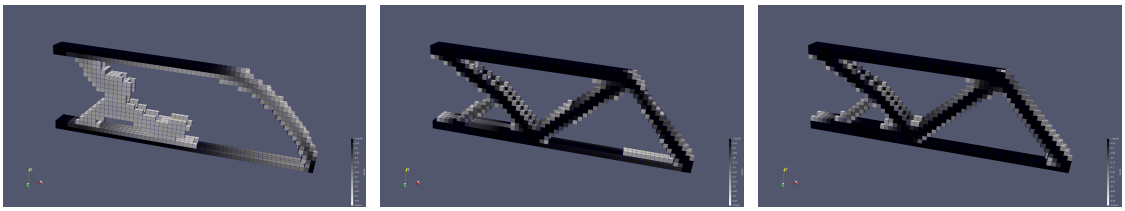



Figure 13: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam initialized with a uniform density of 0.3.



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

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.

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

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

Moreover, SOPTX employs intelligent caching to store invariant data across iterations, significantly reducing computational time in later iterations.

Table 1 compares the performance of different matrix assembly techniques for the 3D cantilever beam optimization (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 assembly method has an average assembly time of 0.838 s per iteration (68% of total iteration time), creating a bottleneck. Fast assembly reduces this to 0.276 s (33% of the original), while symbolic integration precomputation, despite a longer first iteration (3.853 s), achieves the fastest average assembly time (0.272 s) with enhanced accuracy.

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

## 5.6 Application of Automatic Differentiation

Sensitivity analysis is vital in TO to update design variables. Traditional methods rely on manual sensitivity derivation, which is time-consuming and error-prone, especially for complex models. The SOPTX framework introduces AD to streamline this process, enabling users to prioritize problem modeling over mathematical derivations. Here, we demonstrate AD's application using the 3D cantilever beam example from Section 5.4.

The SOPTX framework supports multiple computational backends(e.g., NumPy, PyTorch, JAX). To enable AD, users must switch from the default NumPy backend to an AD-enabled backend like PyTorch [22] with a single line of code:

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

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

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

This setup allows selective use of AD or manual differentiation. Further details on AD implementation are in Section Appendix 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 structural compliance sensitivity is computed using AD. Figure 15 shows the convergence histories of the compliance and volume fraction.

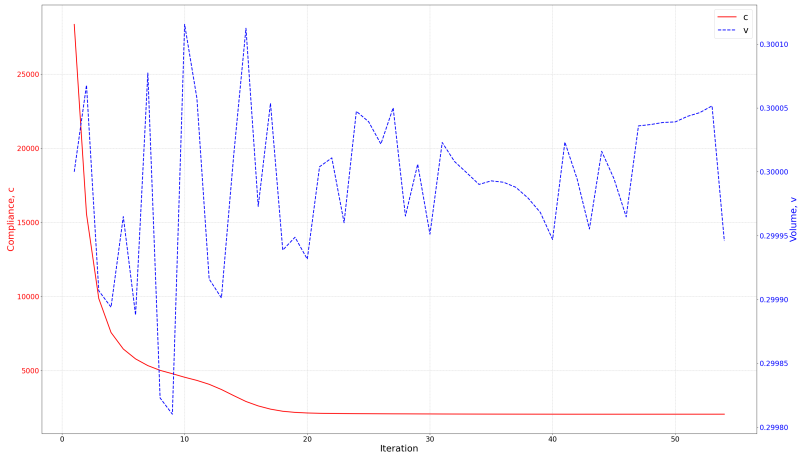
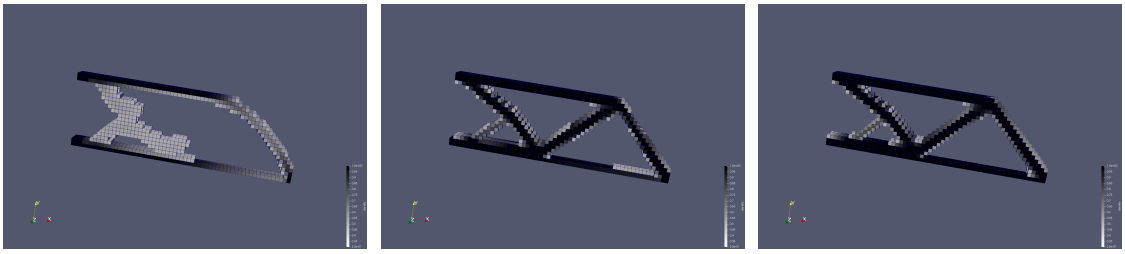



Figure 15: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam optimization using automatic differentiation.

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

As observed from Figure 15 and Figure 16, the optimization results obtained using AD are fully consistent with those achieved through manual differentiation in Section 5.4. The compliance converges to approximately 2063.5625 after 54 iterations, and the volume fraction remains stably around 0.3. These results confirm that AD accurately computes the sensitivities and effectively guides the optimization process.

To evaluate the computational efficiency of AD, we compare the optimization times of AD and manual differentiation under the same 3D cantilever beam problem setting. As shown in Table 2, the total optimization time and average iteration time for both methods are nearly identical,



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

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.

demonstrating that SOPTX's AD implementation is highly efficient, introducing no significant computational overhead while maintaining the same high precision as manual differentiation.

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 |

In addition to PyTorch, SOPTX also supports the JAX backend [7], which offers equally powerful AD capabilities. The sensitivity computation code remains identical when using JAX, users simply need to switch to the JAX backend by executing:

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

Using AD with the JAX backend, we perform topology optimization for the 3D cantilever beam, achieving results fully consistent with those from the PyTorch backend. This consistency underscores the stability and flexibility of the SOPTX framework across different computational backends. The final optimized topology is shown in Figure 17.

The application of AD in TO offers several significant advantages:

- **Simplified Model Switching**: AD automates gradient computations for material interpolation models like SIMP and Rational Approximation of Material Properties (RAMP) [28], enabling seamless switching without manual derivation and improving development efficiency.

- **Seamless Constraint Switching**: AD simplifies sensitivity analysis for constraints such as volume, length scale [15], connectivity [19], overhangs [23], and material usage [24], enhancing optimization adaptability.

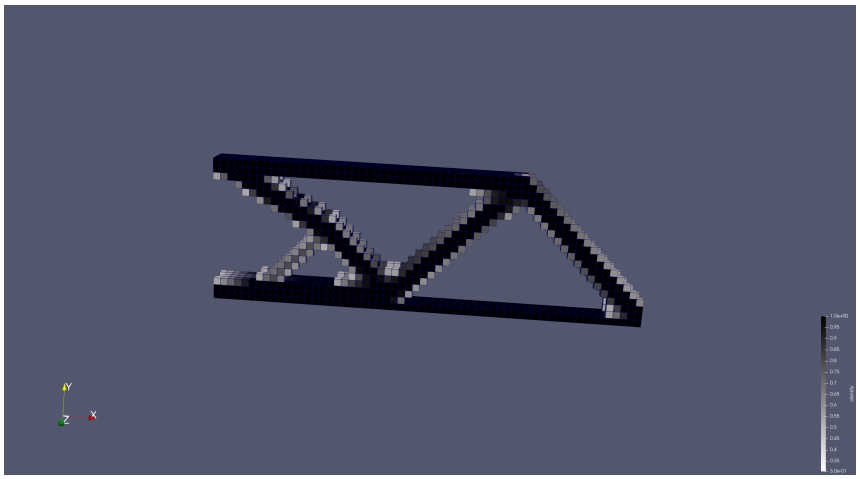


Figure 17: Final optimized topology of the 3D cantilever beam using AD with the JAX backend.

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

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

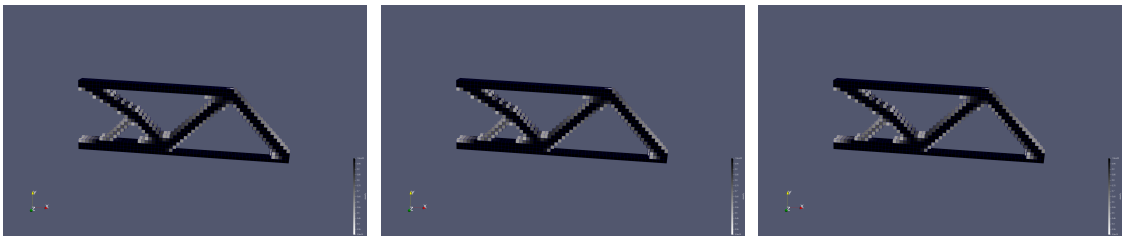
## 5.7 Multi-Backend Switching

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

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

To validate its consistency and reliability across backends, we conducted tests on the 3D cantilever beam optimization problem described in Sections 5.4 and 5.6, using identical parameter settings. The problem was solved independently under each backend, and the results, shown in Figure 18, demonstrate high consistency: compliance converges to 2063.5625, the volume fraction stabilizes at 0.3, and the topology layouts are almost identical. This confirms SOPTX's stability and robustness across different computational backends.

In addition to supporting multi-backend switching, SOPTX enables computations on GPU to enhance efficiency, particularly for large-scale 3D TO problems. By default, computations



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

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.

are performed on the CPU (`device='cpu'`). Users can migrate computations to a GPU (e.g., CUDA devices) in two ways:

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

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

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

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

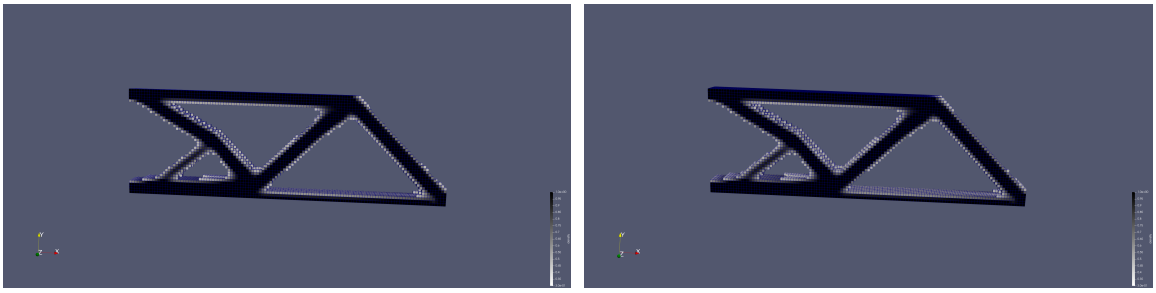
```
mesh = UniformMesh3d(extent=extent, h=h, origin=origin,
    device='cuda')
```

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

GPU acceleration is essential for 3D TO due to the significant increase in computational complexity as mesh size grows. For instance, in the 3D cantilever beam example from Section 5.4, doubling the mesh to $120 \times 40 \times 8$ increases the density design variables to 38,400 and displacement degrees of freedom to 133,947. On the CPU, this leads to prolonged optimization times due to matrix assembly and linear solving costs. In contrast, GPU's parallel computing capabilities greatly enhance efficiency and reduce computation time.

To demonstrate this, we compare CPU and GPU computation times under the PyTorch backend for the same problem. Parameters remain consistent with Section 5.4, except for increasing the filter radius from 1.5 to 3.0 to maintain structural smoothness in the larger domain. Additionally, we use the Conjugate Gradient (CG) method instead of a direct solver (e.g., MUMPS) to handle the larger sparse linear systems, leveraging the GPU's parallel processing for efficient matrix-vector operations.

The tests are conducted on an AMD 9950X CPU and an NVIDIA 5070Ti GPU. As shown in Figure 19, the optimized topologies using CPU and GPU are nearly identical. Minor differences in compliance and volume fraction, both within 1%, arise from variations in floating-point precision and parallel implementation strategies across hardware platforms. These differences are negligible in TO and do not impact the quality or performance of the final design.



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

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

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

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 |

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

# 6 Conclusion

This study presents SOPTX, a high-performance topology optimization (TO) framework built on the open-source FEALPy platform. SOPTX addresses key challenges in TO by offering a

modular, efficient, and accessible solution for both academic and industrial applications. Its fully open-source design eliminates reliance on commercial software, broadening its accessibility for research and education.

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

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

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

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

The experimental analyses in Sections 5.1 through 5.7 demonstrate SOPTX's exceptional performance across key areas: seamless model and filter switching, flexible optimization algorithms, 3D problem extension, enhanced computational efficiency, automatic differentiation, and multi-backend support. These results underscore SOPTX's technical strengths and versatility.

Looking forward, SOPTX is set to expand its capabilities:

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

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

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

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

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

## Acknowledgments

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

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

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

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

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

    @cartesian
```

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

# Appendix B MBBBeam2dData1

```
1  class MBBBeam2dData1:
2  def __init__(self,
3               xmin: float=0, xmax: float=60,
4               ymin: float=0, ymax: float=20,
5               T: float = -1):
6      self.xmin, self.xmax = xmin, xmax
7      self.ymin, self.ymax = ymin, ymax
8      self.T = T
9      self.eps = 1e-12
10
11 def domain(self) -> list:
12     box = [self.xmin, self.xmax, self.ymin, self.ymax]
13     return box
14
15 @cartesian
16 def force(self, points: TensorLike) -> TensorLike:
17     domain = self.domain()
18     x = points[..., 0]
```

```
    y = points[..., 1]
    coord = ((bm.abs(x - domain[0]) < self.eps) &
             (bm.abs(y - domain[3]) < self.eps))
    kwargs = bm.context(points)
    val = bm.zeros(points.shape, **kwargs)
    val[coord, 1] = self.T
    return val

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

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

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

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

# Appendix C Cantilever3dData1

```
class Cantilever3dData1:
    def __init__(self,
```

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

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

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

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

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

```

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

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

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

# Appendix D Automatic Differentiation

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

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

    def compliance_contribution(rho_i: float, ue_i:
    TensorLike, ke0_i: TensorLike) -> float:
        E = self.materials.calculate_elastic_modulus(rho_i)
        c_i = -E * bm.einsum('i, ij, j', ue_i, ke0_i, ue_i)
        return c_i

    vmap_grad = bm.vmap(lambda r, u, k:bm.jacrev(lambda x:
```

```
        compliance_contribution(x, u, k))(r))
    dc = vmap_grad(rho, ue, ke0)
    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] M. P. Bendsøe. Optimal shape design as a material distribution problem. *Structural optimization*, 1:193–202, 1989.

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

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

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

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

[18] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

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

[20] A. Meurer, C. Smith, M. Paprocki, O. Čertík, S. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. Moore, S. Singh, T. Rathnayake, S. Vig, B. Granger, R. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, and A. Scopatz. Sympy: Symbolic computing in python, 06 2016.

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

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

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

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

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

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

[27] O. Sigmund and J. Petersson. Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Structural optimization*, 16:68–75, 1998.

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

[29] K. Svanberg. The method of moving asymptotes—a new method for structural optimization. *International journal for numerical methods in engineering*, 24(2):359–373, 1987.

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

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

[32] M. Zhou and G. I. Rozvany. The coc algorithm, part ii: Topological, geometrical and generalized shape optimization. *Computer methods in applied mechanics and engineering*, 89(1-3):309–336, 1991.