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

Liang He [1], Long Chen[2], Xuehai Huang[3,*], Huayi Wei[1]

[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*

[*]Corresponding author. *Email addresses:* cbtxs@smail.xtu.edu.cn (C. Chen), chenlong@math.uci.edu (L. Chen), huang.xuehai@sufe.edu.cn (X. Huang), weihuayi@xtu.edu.cn (H. Wei)

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.

The main innovations of SOPTX include: (1) A cross-platform, multi-backend support system compatible with various computational backends such as NumPy, PyTorch, and JAX, enabling efficient algorithm execution on CPUs and flexible acceleration using GPUs, as well as efficient sensitivity computation for objective and constraint functions via automatic differentiation (AD); (2) A fast matrix assembly technique, overcoming the performance bottleneck of traditional numerical integration methods and significantly enhancing computational efficiency; (3) A modular framework designed to support TO problems for arbitrary dimensions and meshes, complemented by a rich library of composable components, including diverse filters and optimization methods, enabling users to flexibly configure and extend optimization workflows according to specific needs.

Taking the density-based method as an example, this paper elaborates the architecture, computational workflow, and usage of SOPTX through the classical compliance minimization problem with volume constraints. Numerical examples demonstrate that SOPTX significantly outperforms existing open-source packages in terms of computational efficiency and memory usage, especially exhibiting superior performance in large-scale problems. The modular design of the software not only enhances the flexibility and extensibility of the code but also provides opportunities for exploring novel TO problems, offering robust support for education, research, and engineering applications in TO.

AMS subject classifications: 65N30, 35Q60

Key words: Topology Optimization, Multi-Backend Computing, Automatic Differentiation, Modular Framework

---

1 Introduction

Topology optimization (TO) is an essential class of structural optimization techniques aimed at improving structural performance by optimizing material distribution within a design domain. In fields such as aerospace, automotive, and civil engineering, TO addresses critical design challenges through efficient material utilization and mechanical performance optimization. Among various approaches, density-based methods are particularly popular due to their intuitiveness and practicality. In these methods, the distribution of material and void within the design domain is optimized, and the relative density of each finite element is treated as a design variable. The most widely adopted density-based method is the Solid Isotropic Material with Penalization (SIMP) approach. This approach promotes binary $(0-1)$ solutions by penalizing intermediate densities,

and due to its simplicity and seamless integration with finite element analysis (FEA), it has been extensively employed since its introduction by Bendsøe et al. [5].

However, TO problems inherently involve large-scale computations. This is due to the tight coupling between structural analysis and element-level optimization of design variables: at each iteration, it is necessary not only to solve boundary value problems to obtain structural responses but also to calculate derivatives of the objective and constraint functions with respect to design variables, supporting gradient-based optimization algorithms. For large-scale problems, this implies solving large linear systems and performing sensitivity analysis at every iteration, placing significant demands on computing resources and performance. Therefore, improving computational efficiency and scalability while ensuring accuracy has become a major challenge in TO research and applications.

To lower the entry barrier and promote widespread adoption of TO, researchers have published numerous educational studies and literature. A pioneering example is the 99-line MATLAB code by Sigmund [24], which demonstrated the fundamentals of a two-dimensional SIMP algorithm in a concise and self-contained manner, profoundly impacting both TO education and research. Subsequently, improved versions of this educational code have emerged, including the more efficient 88-line version proposed by Andreassen et al. [2] and the extended 3D implementation by Liu and Tovar [19]. These educational codes convey practical knowledge of TO in the simplest possible form and provide self-contained examples of basic numerical algorithms.

Meanwhile, efforts have also been made to leverage the advantages of open-source software development for addressing TO problems. For example, Chung et al. [11] proposed a modular TO approach based on OpenMDAO [14], an open-source multidisciplinary design optimization framework. They decomposed the TO problem into multiple components, where users provide forward computations and analytic partial derivatives, while OpenMDAO automatically assembles and computes total derivatives, enhancing code flexibility and extensibility. Subsequently, Gupta et al. [17] developed a parallel-enabled TO implementation based on the open-source finite element software FEniCS [1], demonstrating its potential for handling large-scale problems. More recently, Ferro and Pavanello [12] advanced this direction by introducing a concise and efficient TO implementation in just 51 lines of code. Their work utilized FEniCS for modeling and FEA, Dolfin Adjoint for automatic sensitivity analysis, and Interior Point OPTimizer for optimization, greatly simplifying the implementation process. These projects provided standardized interfaces enabling convenient integration with existing FEA tools, thereby lowering implementation complexity. However, despite improvements in modularity achieved by these open-source software packages, challenges in terms of functionality extension and flexibility remain when dealing with complex engineering applications.

To accelerate the development of TO, automating sensitivity analysis has become a critical step. This involves automatically computing derivatives of objectives, constraints, material models, projections, filters, and other components with respect to the design variables. Currently, the common practice involves manually calculating sensitivities, which, despite not being theoretically complex, can be tedious and error-prone, often becoming a bottleneck in the development of new TO modules and exploratory research. Automatic differentiation (AD) provides an efficient and accurate approach for evaluating derivatives of numerical functions [15]. By decomposing

complex functions into a series of elementary operations (such as addition and multiplication), AD accurately computes derivatives of arbitrary differentiable functions. In TO, the Jacobian matrices represent sensitivities of objective functions and constraints with respect to design variables, and software can automate this process, relieving developers from manually deriving and implementing sensitivity calculations. With its capability of easily obtaining accurate derivative information, AD offers significant advantages in design optimization, particularly for highly nonlinear problems.

In recent years, the use of AD in TO has gradually increased. For instance, Nørgaard et al. [21] employed the AD tools CoDiPack and Tapenade to achieve automatic sensitivity analysis in unsteady flow TO, significantly enhancing computational efficiency. Building upon this, Chandrasekhar [10] utilized the high-performance Python library JAX [8] to apply AD techniques in density-based TO, achieving efficient solutions to classical TO problems such as 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 topology optimization 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 [29]. FEALPy is an open-source intelligent CAE computing engine designed to provide an efficient, flexible, and extensible platform for numerical simulation. The choice of FEALPy as the underlying platform is motivated by several key advantages:

- First, FEALPy supports multiple computational backends, including NumPy, PyTorch, and JAX, enabling efficient execution across a wide range of hardware architectures such as CPUs and GPUs.

- Second, FEALPy provides powerful support for multiple numerical methods. It not only implements finite element methods (FEM) of arbitrary order and dimension, but also in-

corporates alternative schemes such as the virtual element method (VEM) and the finite difference method (FDM), offering great flexibility for tackling a wide range of numerical problems.

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

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

Therefore, FEALPy provides a solid foundation for building a modular and extensible TO framework.

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

Building upon this modular foundation, SOPTX seamlessly integrates AD into the TO workflow. It supports multiple computational backends and can automatically switch between them based on hardware availability and performance requirements. In addition, SOPTX alleviates the computational bottlenecks associated with traditional numerical integration through a fast matrix assembly technique. Specifically, it separates element-dependent and element-independent components, avoiding redundant computation of invariant data during iterative procedures. Moreover, symbolic integration via SymPy [20] replaces conventional numerical quadrature, further reducing computational cost while improving accuracy. While maintaining extensibility and reusability, SOPTX adopts a non-intrusive design that avoids disruptive modifications to existing analysis or optimization modules. Users can easily replace or augment filters, constraints, and objectives without altering the core framework. Overall, SOPTX is designed to serve as a low-barrier, high-performance platform for TO research and engineering applications. Through the organic combination of modularity and AD, developers are empowered 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 more complex and multiphysics-coupled scenarios in TO and MDO.

The remainder of this paper is organized as follows. Section 2 presents the problem formulation and mathematical modeling, including the density-based method, the compliance minimization problem, optimization algorithms, and the theoretical foundation and application of filters. Section 3 elaborates on the design philosophy of the SOPTX framework built on the FEALPy platform, with a particular focus on its modular architecture and unique multi-backend switch-

ing mechanism for achieving high-performance and flexible topology optimization. Section 4 provides a step-by-step demonstration of SOPTX installation and usage through the classical 2D cantilever beam problem. In Section 5, the effectiveness and flexibility of SOPTX are comprehensively demonstrated through a series of examples, including the benchmark MBB beam problem, comparative experiments with different filters and optimization algorithms, a three-dimensional cantilever beam case, and performance analyses of fast matrix assembly, AD, and multi-backend switching. Finally, Section 6 concludes the paper and discusses directions for future research.

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

In this chapter, we first introduce the density-based method widely employed in topology optimization (TO), elucidating the fundamental idea of optimizing material distribution through the definition of a material density function and interpolation models. Next, taking the compliance minimization problem as an example, we explicitly present both continuous and discrete mathematical formulations of this method. Subsequently, we provide detailed descriptions of two classical optimization algorithms commonly used in TO: the Optimality Criteria (OC) method and the Method of Moving Asymptotes (MMA), including their algorithmic implementations and characteristics. Finally, we discuss essential filtering techniques in TO, including sensitivity, density, and the Heaviside projection filters, analyzing how these methods effectively mitigate numerical instabilities and enhance the physical feasibility of optimization results.

2.1 Density-based Method

The core objective of TO is to determine the optimal material distribution within a given design domain to achieve predefined performance targets. Density-based methods are among the most widely used strategies in the field of TO. These methods parameterize the structure by defining a material density function $\rho(x)$, where:

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

In a discretized design domain, the mechanical properties (e.g., stiffness) of material elements transition smoothly between solid and void states via interpolation models [3]. Among these, the power-law interpolation model has been widely adopted due to its implicit penalization of intermediate densities. This method is known as the Solid Isotropic Material with Penalization (SIMP) approach [30].

The SIMP method establishes a power-law relationship between the material density ρ and Young's modulus E:

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

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

As the element density ρ approaches zero, the standard SIMP model causes the element stiffness matrix to approach zero, potentially leading to singularities in the global stiffness matrix. To address this issue, the modified SIMP model introduces a minimum Young's modulus $E_{\min}$:

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

where $E_{\min}=10^{-9}E_0$. This modification ensures that the stiffness matrix remains positive definite at $\rho=0$, preventing numerical failure [25].

Although the SIMP method suppresses intermediate densities through the penalization factor, optimization results may still contain "gray areas," which lack 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 compliance minimization problem with volume constraints aims to minimize the structural compliance (i.e., the total strain energy) under external loads and boundary conditions by optimizing the material distribution within a given design domain Ω, while ensuring that the total material volume does not exceed a specified threshold. Here, compliance is defined as the total strain energy of the structure, which can be physically interpreted as the work done by external loads on the displacement field.

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

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

where:

- **Design variable**: $\rho(x)$ represents the material density distribution, and the density function belongs to the function space $L^2(\Omega)$, ensuring square integrability:

$$L^2(\Omega)=\{\rho:\Omega\to\mathbb{R}:\|\rho\|_{L^2(\Omega)}<\infty\},\ \|\rho\|_{L^2(\Omega)}=\left(\int_\Omega|\rho|^2\mathrm{d}x\right)^{1/2}$$

- **Displacement field**: $u(x)$ belongs to the function space $H^1(\Omega)$ to accommodate the weak form of equilibrium equations:

$$H^1(\Omega) = \{u \in L^2(\Omega) : D^1 u \in L^2(\Omega)\}$$

- **Strain and stress**: The strain tensor is defined as

$$\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$$

and the stress tensor as

$$\sigma(u) = D(\rho) : \varepsilon(u)$$

where $D(\rho)$ denotes the material stiffness tensor, belonging to the function space $L^\infty(\Omega)$ to ensure boundedness:

$$L^\infty(\Omega) = \{D : \Omega \to \mathbb{R} : \|D\|_{L^\infty(\Omega)} < \infty\},\ \|D\|_{L^\infty(\Omega)} = \sup_{x \in \Omega} |D(x)|$$

- **Volume constraint**: The constraint $g(\rho) \leq 0$ restricts the total material volume, with V^* representing the prescribed volume threshold.

- **Boundary conditions**: Γ_D denotes the Dirichlet boundary where displacements are fixed, Γ_N denotes the Neumann boundary where surface traction t are applied, f represents the body force, and n is the outward normal direction on the boundary.

To solve the compliance minimization problem numerically, the design domain Ω is discretized into N_e elements using the finite element method, and each element is assigned a constant density ρ_e. This piecewise-constant approximation transforms the continuous formulation into a discrete optimization problem.

The discrete form of the compliance minimization problem can be formulated as follows:

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

where:

- **Design variables**: $\boldsymbol{\rho} = [\rho_1, \rho_2, \ldots, \rho_{N_e}]^T$ represents the density of each element.

- **Stiffness matrix**: The global stiffness matrix is defined as $\boldsymbol{K}(\boldsymbol{\rho}) = \sum_{e=1}^{N_e} \boldsymbol{K}_e(\rho_e)$, where the element stiffness matrix $\boldsymbol{K}_e(\rho_e) = E(\rho_e)\boldsymbol{K}_e^0$, with $\boldsymbol{K}_e^0$ being the element stiffness matrix corresponding to unit Young's modulus.

- **Load**: The load vector $\boldsymbol{F}$ represents the discrete form of the body force f and surface traction t.

- **Displacement**: The displacement vector $\boldsymbol{U}(\boldsymbol{\rho})$ is obtained by solving the linear system $\boldsymbol{K}(\boldsymbol{\rho})\boldsymbol{U} = \boldsymbol{F}$, providing an approximation to the continuous equilibrium equations.

- **Volume constraint**: The constraint $g(\boldsymbol{\rho}) \leq 0$ ensures that the total material volume (weighted sum of element volumes v_e and densities ρ_e) does not exceed V^*.

2.3 Optimization Algorithms

The Optimality Criteria (OC) method is a classical TO algorithm widely used for compliance minimization problems under volume constraints. This method iteratively updates the design variables to improve the structural topology by satisfying the optimality conditions of the optimization problem. The core idea of the OC method is to adjust the material density ρ_e based on the update factor

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

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

Bendsøe [4] proposed a heuristic update scheme for the design variables, given by the following formula:

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

where:

- m is the move limit, which constrains the maximum allowable change in the design variables,

- η is the damping coefficient, which controls the step size and stability of the algorithm.

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

Algorithm 2.1 OC pseudo-code

Input: Initial design variables $\boldsymbol{\rho}^{(0)}$ (with $0 \leq \rho_e^{(0)} \leq 1$), volume constraint V^*, move limit m, damping factor η, maximum iterations MaxIter, convergence tolerance ϵ
Output: Optimized design variables $\boldsymbol{\rho}$
Set $\boldsymbol{\rho}^{(k)} \leftarrow \boldsymbol{\rho}^{(0)}$, $k \leftarrow 0$
while $k < \text{MaxIter}$ *and* $\|\boldsymbol{\rho}^{(k+1)} - \boldsymbol{\rho}^{(k)}\|_\infty > \epsilon$ **do**
  Evaluate the objective function $c(\boldsymbol{\rho}^{(k)})$ and the volume constraint $g(\boldsymbol{\rho}^{(k)})$
  Compute the sensitivities $\nabla c(\boldsymbol{\rho}^{(k)})$ and $\nabla g(\boldsymbol{\rho}^{(k)})$
  (Optional) Apply a filter to the sensitivities
  Update the Lagrange multiplier λ using the bisection method
  Update the design variables $\boldsymbol{\rho}^{\text{new}}$ using the optimality criterion
  (Optional) Apply a filter to $\boldsymbol{\rho}^{\text{new}}$
  Set $\boldsymbol{\rho}^{(k+1)} \leftarrow \boldsymbol{\rho}^{\text{new}}$, $k \leftarrow k+1$
end

In TO, in addition to the OC method, the Method of Moving Asymptotes (MMA) is a widely used gradient-based optimization algorithm proposed by Svanberg [27]. It is particularly well-suited for problems involving multiple constraints or complex objective functions. The core idea of MMA is to dynamically adjust the approximation range of the design variables using "moving asymptotes," thereby transforming the original nonlinear optimization problem into a sequence of convex subproblems. This strategy ensures numerical stability throughout the iterative process.

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

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

where m is the number of constraints, n is the number of design variables, and ρ_j denotes the j-th design variable. The lower and upper bounds of the design variables in the k-th iteration are denoted by $\alpha_j^{(k)}$ and $\beta_j^{(k)}$, respectively. The variables y_i and z are auxiliary variables introduced to ensure convex approximations of the objective and constraint functions. The parameters a_0, a_i, c_i, d_i are given constants.

The convex approximation functions are defined as:

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

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

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

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

Algorithm 2.2 MMA pseudo-code

Input: Initial design variables $\boldsymbol{\rho}^{(0)}$, 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 [6]. These numerical difficulties can lead to suboptimal or unstable designs. Filtering techniques serve as an important form of regularization by smoothing the spatial distribution of either design variables or sensitivities, thereby suppressing numerical oscillations and improving both the reliability and physical realizability of the optimized structures [26]. A comprehensive overview of various filtering methods was provided by Sigmund [25].

The sensitivity filter was proposed by Sigmund [23] as a means to smooth the update of design variables by performing a weighted average on the sensitivities. The original mesh-dependent sensitivity filtering formula is given by:

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

where:

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

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

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

The density filter was originally proposed by Bruns and Tortorelli [9], and later mathematically justified by Bourdin [7] as a valid regularization technique. The basic form of the filtered density function is defined as:

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

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

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

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

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

where:

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

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

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

algorithm, a smooth approximation is used in practice to replace the traditional Heaviside step function:

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

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

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

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

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

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

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

- When $r_{\min} \to 0$, 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, with an emphasis on its support for multi-backend switching to enhance computational performance and flexibility. The chapter is organized into three parts. We first examine the architecture of FEALPy and its role in providing a foundational tensor computation environment. We then describe the modular structure of SOPTX, including its material, solver, filter, and optimization modules. Finally, we discuss the design and advantages of the multi-backend mechanism, which enables compatibility with NumPy, PyTorch, and JAX to meet a range of computational needs.

3.1 FEALPy Architecture Design

As shown in Figure 1, FEALPy adopts a layered architecture consisting of four levels, arranged from bottom to top: the tensor level, the common level, the algorithm level, and the field level. Building on this architecture, FEALPy introduces the concept of a *Tensor Backend Manager*, which provides unified management and scheduling of different tensor computation backends. This mechanism offers a consistent tensor operation interface for the upper layers. The interface

currently follows the Python Array API Standard v2023.12 [13], and is compatible with multiple backends such as NumPy, PyTorch, and JAX, enabling the same codebase to adapt seamlessly to different software and hardware platforms.

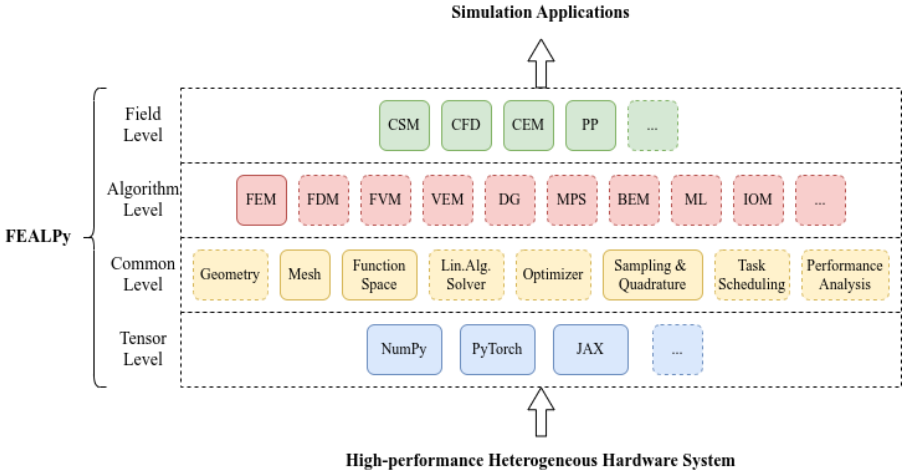


Figure 1: The layered architecture of FEALPy, comprising tensor, common, algorithm, and field levels, progressing from low-level functionalities to high-level applications. Modules in dashed boxes are under development.

Specifically, the functions of each layer are as follows:

- **Tensor level**: Provides basic tensor operations and manages backend systems such as NumPy, PyTorch, and JAX through the *Tensor Backend Manager*. This mechanism forms the foundation for SOPTX's multi-backend switching. In particular, the automatic differentiation (AD) features of PyTorch and JAX enable automatic computation of sensitivities for objective and constraint functions in topology optimization (TO), greatly simplifying the derivative evaluation process.

- **Common level**: Includes components such as mesh generation and finite element spaces, offering the capability to rapidly construct meshes and function spaces required for finite element analysis in TO. This serves as the foundational support for finite element analysis in SOPTX.

- **Algorithm level**: Encompasses solvers and optimization algorithms, providing efficient computational support for optimization methods in SOPTX and ensuring both performance and stability during the optimization process.

- **Field level**: Targets specific physical problems (e.g., linear elasticity), enabling SOPTX to flexibly handle various types of TO problems such as compliance minimization under volume constraints, and offers customized support for application-specific scenarios.

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

3.2 SOPTX Architecture Design

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

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

3.2.1 Material Module

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

The material module encapsulates the interpolation, update, and evaluation of elastic constants within a unified interface, enabling convenient access to material information for downstream modules. Its main functional interfaces include:

1. Computing elasticity tensors from a given density field, providing material stiffness data required by the solver module.

2. Supporting efficient batch updates of material properties, suitable for repeated evaluations in iterative optimization processes.

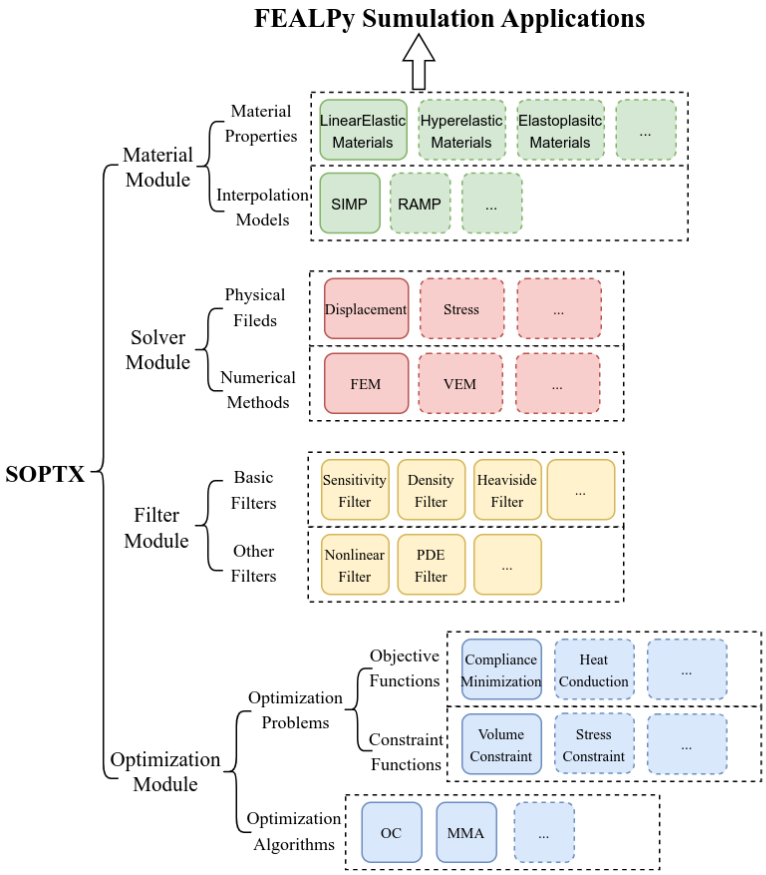


Figure 2: SOPTX is organized into four main modules: material, solver, filter, and optimization—forming a modular and extensible architecture. The material module provides the foundation, while the solver and filter modules handle intermediate computations and jointly support the optimization module.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

3. Supplying essential material parameters for the optimization module, such as the elastic constants of the base material.

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

3.2.2 Solver Module

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

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

1. **Dimensional and element adaptability**: It supports various spatial dimensions, element types, and boundary condition formulations, accommodating diverse engineering analysis requirements. In the current version, given that the density field is represented as piecewise constant per element, the displacement field is discretized using linear finite elements to ensure numerical stability and computational efficiency.

2. **Efficient numerical strategies**: In addition to standard finite element integration and matrix assembly, a fast matrix assembly technique is implemented. This approach separates the element-independent and element-dependent parts of the stiffness matrix, avoiding redundant computations common in traditional methods and significantly improving efficiency for large-scale problems. Furthermore, symbolic integration is supported, enabling precomputed exact expressions without the need for numerical quadrature, thereby further reducing computational cost.

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

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

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

1. **Interaction with the material module**: The solver module receives material properties such as elasticity matrices from the material module, which are used to assemble the global stiffness matrix. This interaction is handled via unified interfaces, so that the solver does not depend on the specific material interpolation scheme.

2. **Output to the optimization module**: The solver module provides two key types of outputs to the optimization module: the displacement field, which is used to evaluate the objective function, and the stiffness matrix, which is used in sensitivity analysis. These outputs follow consistent interface definitions across different backends, while their internal representations are backend-specific and optimized to ensure efficient downstream computations.

3. **Result reuse**: Given the iterative nature of topology optimization, the solver module incorporates an intelligent caching mechanism to avoid redundant computations of invariant components (e.g., shape function derivative matrices). This significantly reduces computational overhead, particularly in large-scale problems.

Through its modular design and clearly defined interfaces, the solver module performs physical field computations efficiently, providing essential computational support for the entire topology optimization process while maintaining consistency and high performance across multi-backend environments.

3.2.3 Filter Module

The filter module is a key component in SOPTX responsible for processing design variables and applying regularization. It plays a dual role in the topology optimization process: eliminating

numerical instabilities such as checkerboarding and mesh dependency, and enhancing the manufacturability of the resulting structures. Following the overall loosely coupled design philosophy of the framework, the filter operations are implemented as an independent functional unit, while maintaining efficient data exchange with other modules. Similar to the material and solver modules, the filter module provides a consistent interface across different computational backends (NumPy / PyTorch / JAX), while its internal implementations are optimized according to backend-specific features to ensure maximal computational efficiency.

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

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

2. **Density filtering**: Directly filters the design variables by mapping raw design values to physical densities, addressing the locality issues that may arise with sensitivity filtering in some problems.

3. **Heaviside projection filtering**: Builds upon density filtering by applying a smoothed Heaviside function to force intermediate densities toward 0 or 1, thereby promoting clear black-and-white structural designs.

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

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

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

1. **Interaction with the solver module**: It receives the raw design variables and outputs the filtered physical density field.

2. **Interaction with the optimizer module**: It takes in the unfiltered sensitivities computed by the optimization algorithm and returns the filtered sensitivities, ensuring numerical stability during the optimization process.

3. **Coordination with the material module**: The filtered physical density is directly used for material interpolation, forming a clear data flow:

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

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

3.2.4 Optimization Module

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

In the current version, the optimization module focuses on the classical compliance minimization problem under volume constraints and supports two mainstream topology optimization algorithms: Optimality Criteria (OC) and the Method of Moving Asymptotes (MMA). The MMA implementation is based on the standard version proposed by Svanberg in 2007 [28], and has been extended in SOPTX to support multiple backend implementations. While preserving mathematical equivalence, backend-specific optimizations have been introduced to improve computational performance, particularly enabling GPUs acceleration in PyTorch and JAX environments.

The design of the optimization module follows two core principles:

1. **Separation of problem definition and algorithm**: The definition of the optimization problem—namely, the objective functions, the constraint functions, and their sensitivities—is strictly decoupled from the optimization algorithms. This allows users to flexibly combine different problem formulations with various optimization solvers.

2. **Dual-mode sensitivity computation**: The framework supports both manually derived and automatically differentiated sensitivities. On PyTorch and JAX backends, users can directly leverage AD to avoid complex manual derivations, while still preserving the physical consistency and interpretability offered by hand-derived formulations.

The optimization module is designed with high extensibility:

1. **Problem type extension**: Users can easily define new optimization objectives—such as heat conduction, Compliant mechanism synthesis, or multiphysics problems—and corresponding constraint conditions, such as stress or frequency constraints, by subclassing the base optimization problem class.

2. **Algorithm extensibility**: The framework allows integration of new optimization algorithms, including Sequential Quadratic Programming (SQP), Sequential Linear Programming (SLP), and other gradient-based methods.

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

1. **Interaction with the solver module**: It receives physical field outputs such as the displacement field and stiffness matrix, which are used to evaluate the objective function (e.g., compliance) and its corresponding sensitivities.

2. **Interaction with the filter module**: It passes raw sensitivities to the filter module for processing and receives the filtered sensitivities to update the design variables. Additionally, the newly updated design variables are filtered again to obtain the physical density field.

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

3.3 Multi-Backend Switching

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

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

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

- **PyTorch and JAX backends**: Both support GPUs acceleration and AD, making them ideal for large-scale or high-dimensional problems. The AD capability greatly simplifies the

computation of sensitivities for objective and constraint functions, improving development efficiency and significantly shortening the research cycle. JAX goes one step further by offering more flexible compilation strategies and automatic vectorization, achieving high computational efficiency, especially on GPUs platforms.

4 Getting Started with SOPTX

This chapter aims to provide readers with installation instructions and usage guidance for SOPTX, helping them quickly become familiar with the basic operations of the framework and apply it to topology optimization (TO) tasks. As a TO toolkit built on top of FEALPy, SOPTX offers a highly efficient and flexible computational environment through its multi-backend switching mechanism and modular design. The content of this chapter is divided into two parts: first, we present a detailed explanation of the installation steps for SOPTX and its dependency FEALPy, ensuring that readers can correctly configure the development environment; then, we demonstrate the usage workflow of SOPTX through a classical 2D cantilever beam compliance minimization example.

4.1 Software Installation

SOPTX is a TO toolkit built on top of FEALPy, an intelligent CAE simulation engine that provides numerical computing capabilities. Installing FEALPy is a prerequisite, and it is recommended to install it from source. Before installation, ensure that Git and Python are properly installed. It is also recommended to use a virtual environment when installing and running FEALPy.

Core installation steps:

1. Clone the FEALPy repository from GitHub:

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

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

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

Code 1: Module imports and PDE model

```
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)
```

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

Code 2: Mesh and function space definitions

```
    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')
```

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

Code 3: Material module

```
    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)
```

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

Code 4: Solver and filter module

```
    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)
```

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

Code 5: Optimization module

```
    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})
```

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

Code 6: Main program and post-processing

```
    if __name__ == "__main__":
        @cartesian
        def density_func(x):
            val = bm.ones(x.shape[0])
```

```
            # val = config.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)
```

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

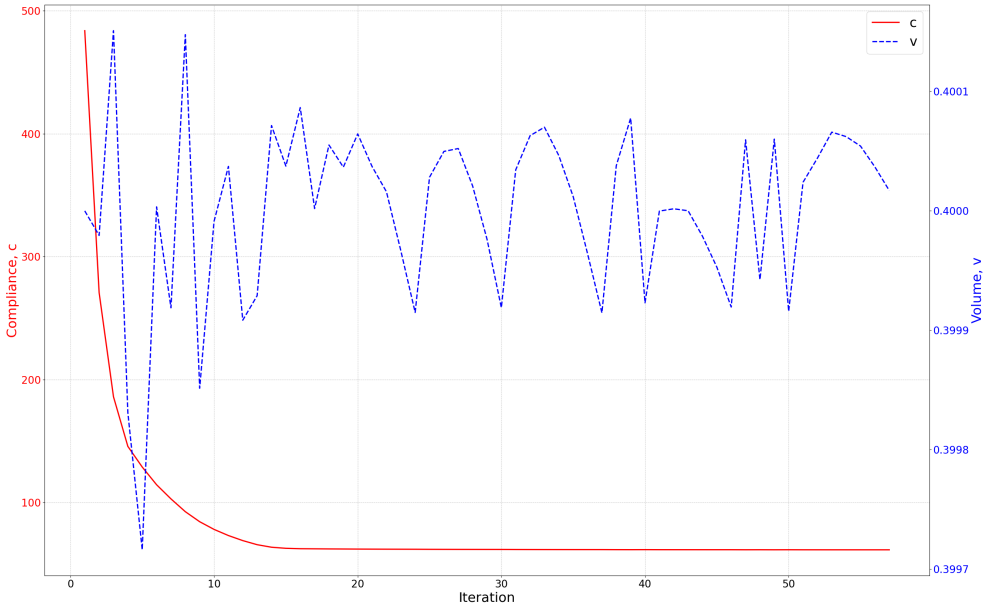
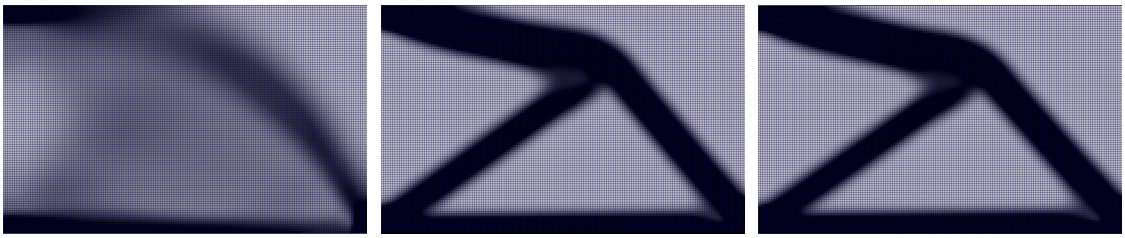


Figure 4: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam initialized with a uniform density of 0.4.

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

The initial material density is uniformly set to 1 over the design domain. After running the



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

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.

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

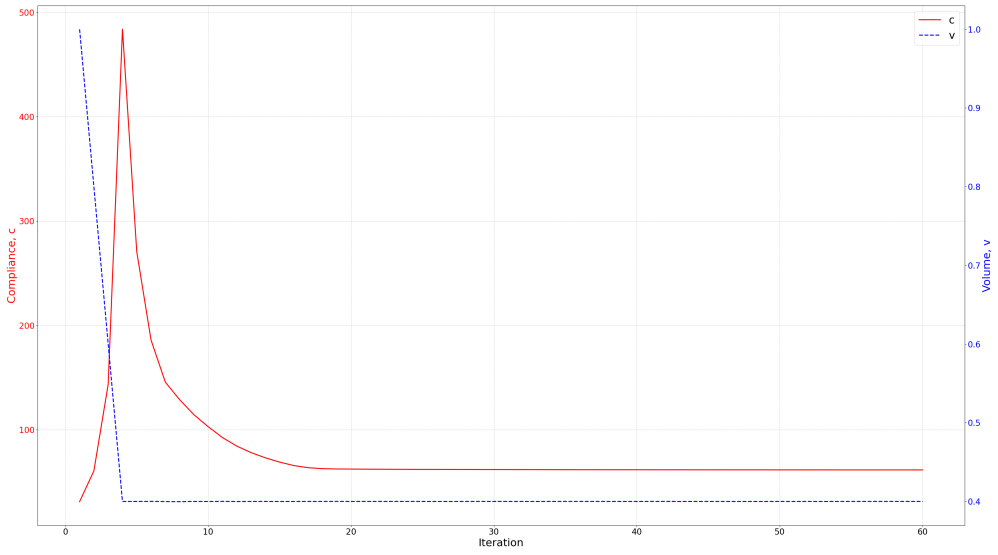
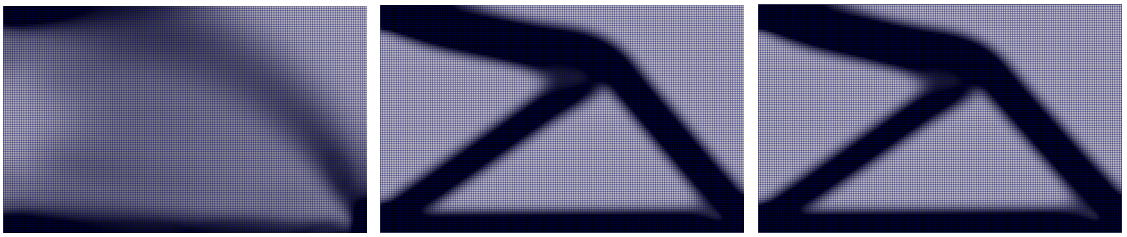


Figure 6: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam with an initial density of 1.

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



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

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

5 Numerical Examples

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

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

5.1 MBB Beam

The MBB beam is a classical benchmark problem in TO, widely used to verify the effectiveness and robustness of optimization algorithms. In this section, we minimize the structural compliance of the MBB beam (see Figure 8) to demonstrate the flexibility and consistency of the SOPTX framework in handling different PDE models and mesh settings. The beam is subject to hinged boundary conditions along its left edge and the bottom right corner, where only horizontal displacements are constrained (i.e., zero horizontal movement is enforced while vertical motion is free). A downward concentrated load $T = -1$ is applied to the upper left corner. The target vol-

ume fraction is set to 0.5, and the material parameters are $E = 1$ and $\nu = 0.3$. The penalization factor is $p = 3$, and the filter radius is $r = 6.0$, which matches the mesh element size to ensure smooth topology and suppress checkerboard patterns.

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

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

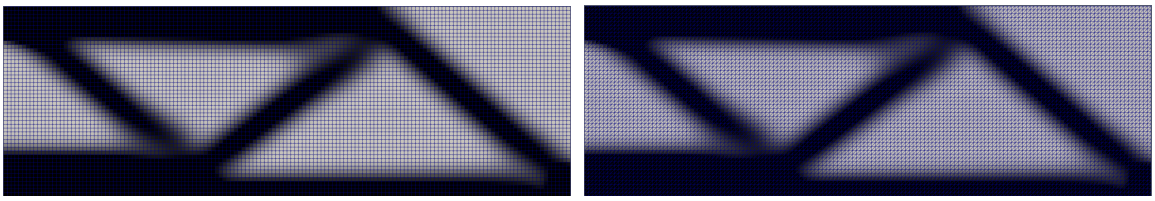
```
from soptx.pde import MBBBeam2dData1
pde = MBBBeam2dData1(xmin=0, xmax=150, ymin=0, ymax=50, T =
    -1)
```

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

```
from fealpy.mesh import TriangleMesh
```

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

The resulting topologies are shown in Figure 9:



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

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

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

5.2 Different Filtering Methods

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

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

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

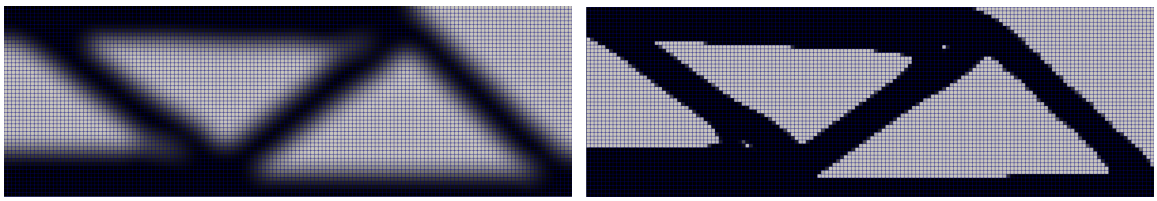
```
    dens_filter = DensityBasicFilter(mesh=mesh, rmin=6.0)
```

The Heaviside projection filter builds upon the density filter by introducing a projection operation. It employs a continuation strategy to gradually increase the projection parameter β, driving design variables toward 0 or 1 and producing clear black-and-white topologies. This method is

particularly suitable for structural design problems with strict manufacturability requirements. To avoid overly wide bar-like structures in the early optimization stages, we slightly reduce the filter radius to $r = 4.5$. The filter can be enabled in SOPTX using the following code:

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

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



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

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

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

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

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

5.3 Different Optimization Algorithms

In TO, the choice of optimization algorithm directly affects both computational efficiency and the quality of the resulting design. SOPTX adopts a modular design that enables seamless switching between different optimizers, allowing users to easily tailor optimization strategies to specific design requirements. In this section, we take the MBB beam problem as an example to demonstrate

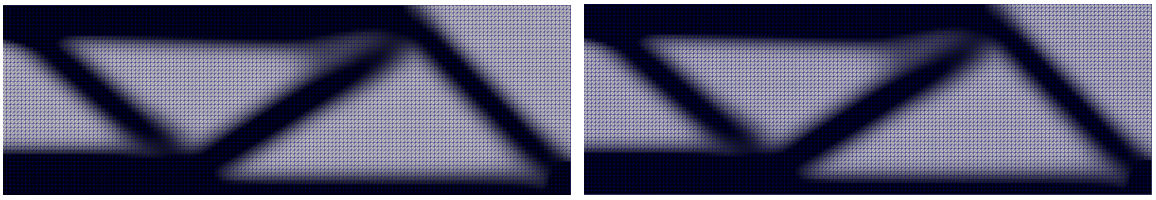
the process of switching from the Optimality Criteria (OC) optimizer to the Method of Moving Asymptotes (MMA). We also present the refactored MMA implementation within SOPTX and highlight its distinctive advantages.

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

```
from soptx.opt import MMAOptimizer

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

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



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

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

The MMA algorithm in SOPTX is based on the classical MATLAB implementation provided by Krister Svanberg [28], and has been extensively refactored. Unlike traditional MMA implementations commonly found in the literature—where the optimizer is typically treated as a "black box" with limited access to internal mechanisms—the refactored version in SOPTX allows users to conveniently adjust internal algorithmic parameters. These include the number of constraints `m`, the number of design variables `n`, the lower and upper bounds `xmin` and `xmax`, as well as internal control parameters such as `a0`, `a`, `c`, and `d`:

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

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

5.4 3D Extension

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

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

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

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

The initial material density is uniformly set to the target volume fraction of 0.3. Using the OC optimizer and a sensitivity filter with radius $r = 1.5$, the convergence histories of the compliance $c(\rho)$ and the volume fraction $v(\rho)$ are shown in Figure 13. The compliance drops rapidly from an

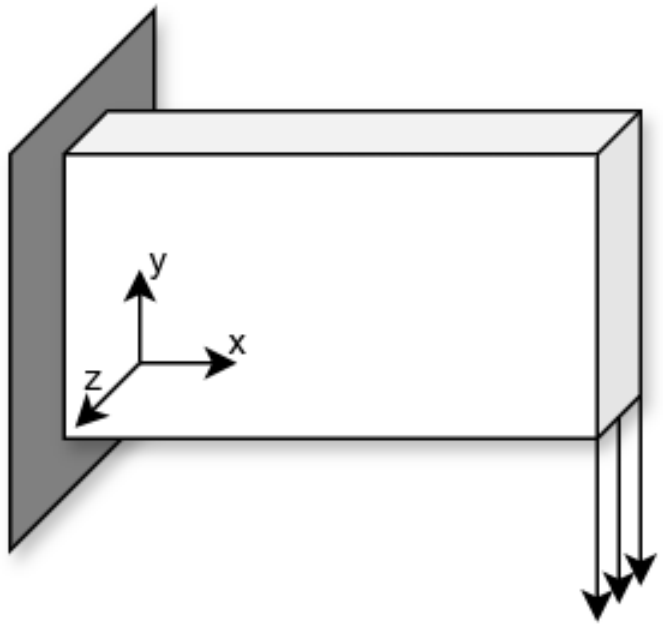


Figure 12: Geometric configuration of the 3D cantilever beam. The left end is fully fixed, and a downward concentrated load is applied at the bottom of the right end.

initial value of approximately 28000 to around 5000 within the first 6 iterations. It then decreases more gradually, fluctuating slightly between iterations 6 and 20 before settling to about 2000. Smooth convergence is eventually achieved at iteration 54. The volume fraction remains nearly constant around 0.3 throughout the process, indicating effective constraint control.

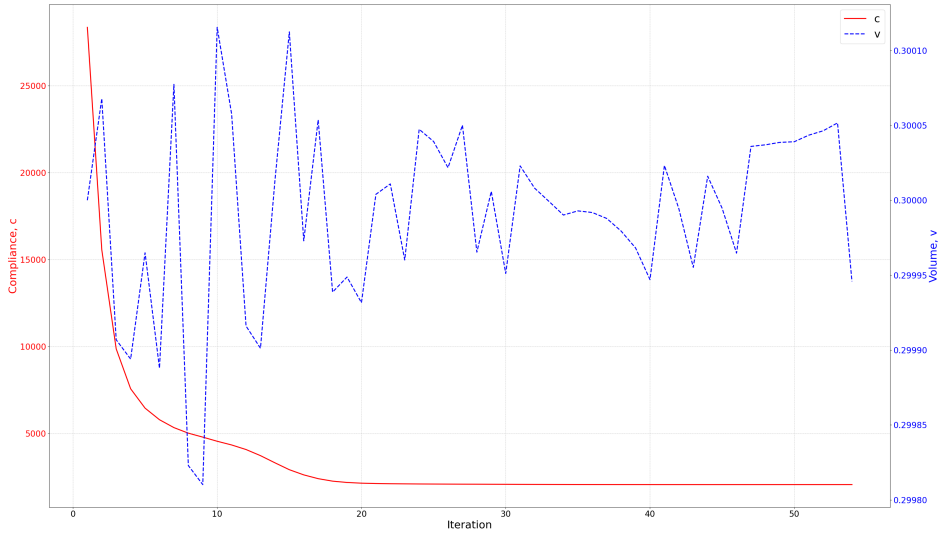
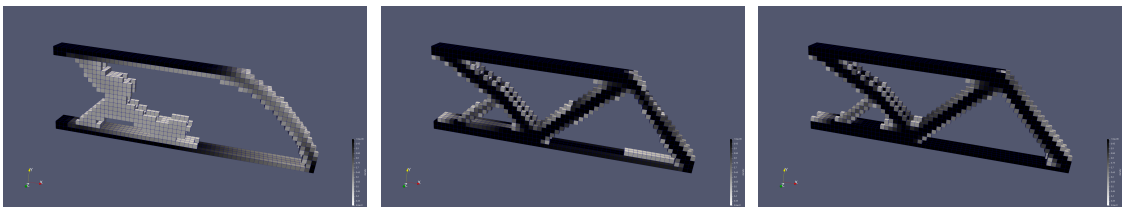


Figure 13: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam initialized with a uniform density of 0.3.

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



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

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

It is worth noting that, compared to 2D problems, 3D TO significantly increases computational complexity. This is particularly evident when using large-scale meshes, where both com-

putational cost and memory requirements grow substantially. The SOPTX framework addresses these challenges through efficient matrix assembly and support for multiple computational backends, demonstrating excellent extensibility and computational performance. The implementation details of these features will be elaborated in the following sections.

5.5 Application of Fast Matrix Assembly

In 3D TO problems, computational efficiency is a central challenge. Taking the 3D cantilever beam example from Section 5.4 as a representative case, the mesh consists of $60 \times 20 \times 4$ elements, resulting in 19,215 displacement degrees of freedom and 4,800 density variables. The optimization process requires 54 iterations in total. In traditional finite element methods, the assembly of the global stiffness matrix often involves a large number of redundant numerical integration operations, which becomes the primary bottleneck in the overall computational workflow.

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

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

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

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

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

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

As shown in Table 1, the original matrix assembly method results in an average assembly time of 0.838 s per iteration after the first one, accounting for approximately 68% of the total iteration

time. This clearly constitutes a performance bottleneck. With the adoption of the fast matrix assembly technique, the average assembly time is reduced to 0.276 s—only about 33% of that of the original method. Although the symbolic fast assembly method incurs a longer initialization time of 3.853 s during the first iteration, it achieves the lowest average assembly time in subsequent iterations (0.272 s), while ensuring analytical accuracy in integration.

In summary, the matrix fast assembly strategy implemented in SOPTX effectively reduces the computational cost of topology optimization and improves both numerical accuracy and stability. This makes it particularly well suited for solving large-scale topology optimization problems with stringent demands on computational efficiency and precision.

5.6 Application of Automatic Differentiation

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

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

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

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

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

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

```
        =0.5, config=cons_config)
```

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

To verify the correctness and effectiveness of AD, we conduct tests using the 3D cantilever beam problem described in Section 5.4. All parameters are kept identical: the mesh size is $60\times 20\times 4$, the target volume fraction is set to 0.3, the initial material density is 0.3, material properties are $E=1$ and $\nu=0.3$, the SIMP penalty factor is $p=3$, and the sensitivity filter radius is $r=1.5$. The structural compliance sensitivity is computed using AD. Figure 15 shows the convergence histories of the compliance and volume fraction, while Figure 16 presents the topology layouts at iterations 7, 21, and 54.

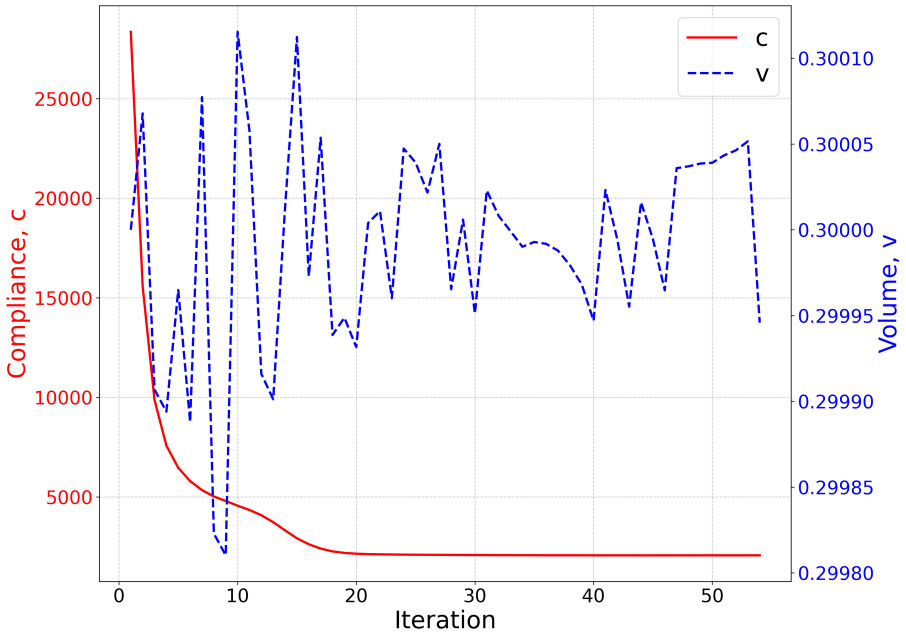
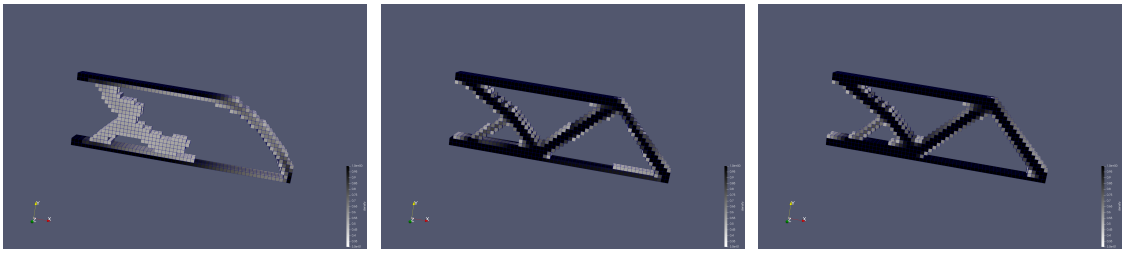


Figure 15: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 3D cantilever beam optimization using automatic differentiation.

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



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

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.

and DMS-2309785. The third author was supported by the National Natural Science Foundation of China (NSFC) (Grant No. 12171300), and the Natural Science Foundation of Shanghai (Grant No. 21ZR1480500).

Appendix A. Cantilever2dData1

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

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

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

```
                (bm.abs(x - domain[1]) < self.eps) &
                (bm.abs(y - domain[2]) < self.eps)
                )
            kwargs = bm.context(points)
            val = bm.zeros(points.shape, **kwargs)
            val[coord, 1] = self.T
            return val

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

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

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

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

Appendix B. MBBBeam2dData1

```
    class MBBBeam2dData1:
    def __init__(self,
                 xmin: float=0, xmax: float=60,
                 ymin: float=0, ymax: float=20,
```

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

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

    @cartesian
    def force(self, points: TensorLike) -> TensorLike:
        domain = self.domain()

        x = points[..., 0]
        y = points[..., 1]

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

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

    @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. 3

[2] E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, and O. Sigmund. Efficient topology optimization in matlab using 88 lines of code. *Structural and Multidisciplinary Optimization*, 43:1–16, 2011. 3

[3] M. P. Bendsøe. Optimal shape design as a material distribution problem. *Structural optimization*, 1:193–202, 1989. 6

[4] M. P. Bendsøe. *Optimization of structural topology, shape, and material*, volume 414. Springer, 1995. 9

[5] M. P. Bendsøe and O. Sigmund. *Topology optimization by distribution of isotropic material*, pages 1–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. 3

[6] M. P. Bendsoe and O. Sigmund. *Topology optimization: theory, methods, and applications*. Springer Science & Business Media, 2013. 11, 23

[7] B. Bourdin. Filters in topology optimization. *International journal for numerical methods in engineering*, 50(9):2143–2158, 2001. 12

[8] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018. 4

[9] T. E. Bruns and D. A. Tortorelli. Topology optimization of non-linear elastic structures and compliant mechanisms. *Computer methods in applied mechanics and engineering*, 190(26-27):3443–3459, 2001. 12

[10] A. Chandrasekhar, S. Sridhara, and K. Suresh. Auto: a framework for automatic differentiation in topology optimization. *Structural and Multidisciplinary Optimization*, 64(6):4355–4365, 2021. 4

[11] H. Chung, J. T. Hwang, J. S. Gray, and H. A. Kim. Topology optimization in openmdao. *Structural and multidisciplinary optimization*, 59:1385–1400, 2019. 3

[12] R. M. Ferro and R. Pavanello. A simple and efficient structural topology optimization implementation using open-source software for all steps of the algorithm: Modeling, sensitivity analysis and optimization. *CMES-Computer Modeling in Engineering & Sciences*, 136(2), 2023. 3

[13] C. for Python Data API Standards. Array api standard, version 2023.12. https://data-apis.org/array-api/2023.12/, 2023. Accessed April 2025. 14

[14] J. S. Gray, J. T. Hwang, J. R. Martins, K. T. Moore, and B. A. Naylor. Openmdao: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, 2019. 3

[15] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008. 3

[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. 12

[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. 3

[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. 37

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

[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. 5

[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. 4

[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. 37

[23] O. Sigmund. On the design of compliant mechanisms using topology optimization. *Journal of Structural Mechanics*, 25(4):493–524, 1997. 11

[24] O. Sigmund. A 99 line topology optimization code written in matlab. *Structural and multidisciplinary optimization*, 21(2):120–127, 2001. 3

[25] O. Sigmund. Morphology-based black and white filters for topology optimization. *Structural and*

Multidisciplinary Optimization, 33(4):401–424, 2007. 7, 11

[26] 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. 11

[27] 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. 10

[28] K. Svanberg. Mma and gcmma, versions september 2007. 2007. 20, 32

[29] H. Wei and Y. Huang. Fealpy: Finite element analysis library in python. https://github.com/weihuayi/fealpy, Xiangtan University, 2017-2024. 4

[30] 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. 6