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. [**?**].

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 [**?**], 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. [**?**] and the extended 3D implementation by Liu and Tovar [**?**]. 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. [**?**] proposed a modular TO approach based on OpenMDAO [**?**], 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. [**?**] developed a parallel-enabled TO implementation based on the open-source finite element software FEniCS [**?**], demonstrating its potential for handling large-scale problems. More recently, Ferro and Pavanello [**?**] 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 [**?**]. 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. [**?**] employed the AD tools CoDiPack and Tapenade to achieve automatic sensitivity analysis in unsteady flow TO, significantly enhancing computational efficiency. Building upon this, Chandrasekhar [**?**] utilized the high-performance Python library JAX [**?**] 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 [**?**]. 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 [**?**] 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 [**?**]. 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 [**?**].

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 [**?**].

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 [**?**] 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 [**?**]. 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 $k <$ 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 [**?**]. 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 [**?**]. A comprehensive overview of various filtering methods was provided by Sigmund [**?**].

The sensitivity filter was proposed by Sigmund [**?**] 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:

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

- $N_i = \{j : \mathrm{dist}(i,j) \leq r_{\min}\}$ 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 [**?**], and later mathematically justified by Bourdin [**?**] 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 [**?**]. 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 [**?**], 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.

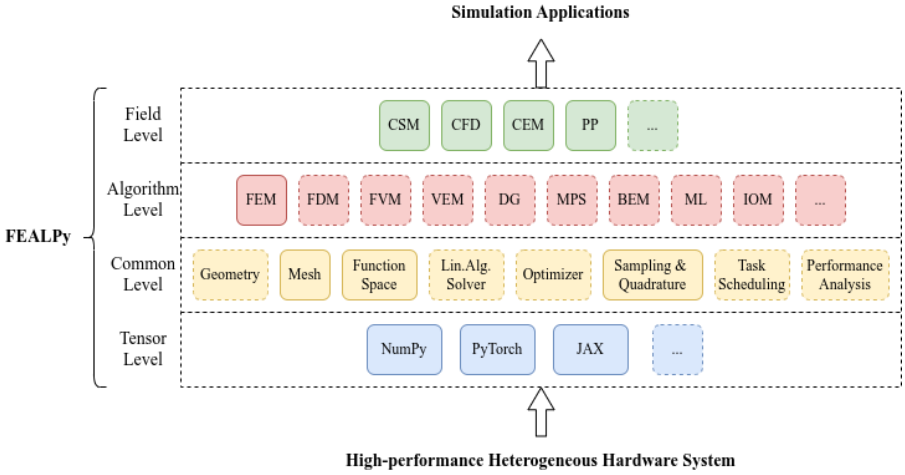


Figure 1: The layered architecture of FEALPy, comprising tensor, common, algorithm, and field levels, progressing from low-level functionalities to high-level applications. Modules in dashed boxes are under development.

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.

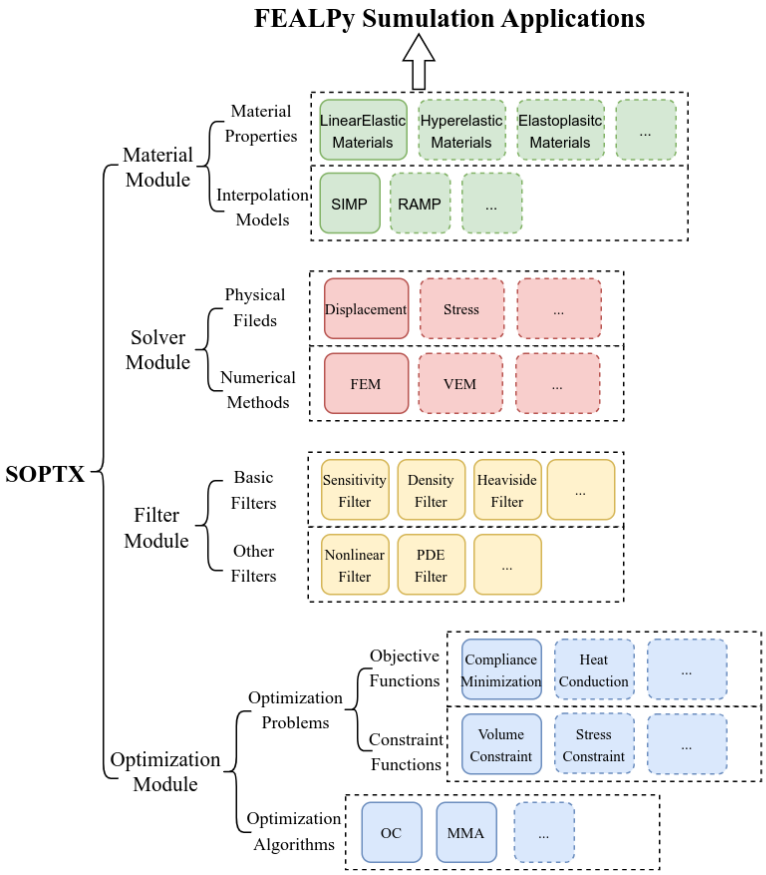


Figure 2: SOPTX is organized into four main modules: material, solver, filter, and optimization—forming a modular and extensible architecture. The material module provides the foundation, while the solver and filter modules handle intermediate computations and jointly support the optimization module. 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 [**?**], 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 [**?**]: 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 2D mesh is used. The target volume fraction is set to 0.4, with material properties $E = 1$ and $\nu = 0.3$. The penalization factor is $p = 3$, and the sensitivity filter radius is $r = 6.0$, which matches the mesh element size to ensure structural smoothness and eliminate checkerboard patterns.

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

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

Code 1: Module imports and PDE model

```
1    from fealpy.backend import backend_manager as bm
2    from fealpy.mesh import UniformMesh2d
3    from fealpy.functionspace import LagrangeFESpace,
        TensorFunctionSpace
4
```

```
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.

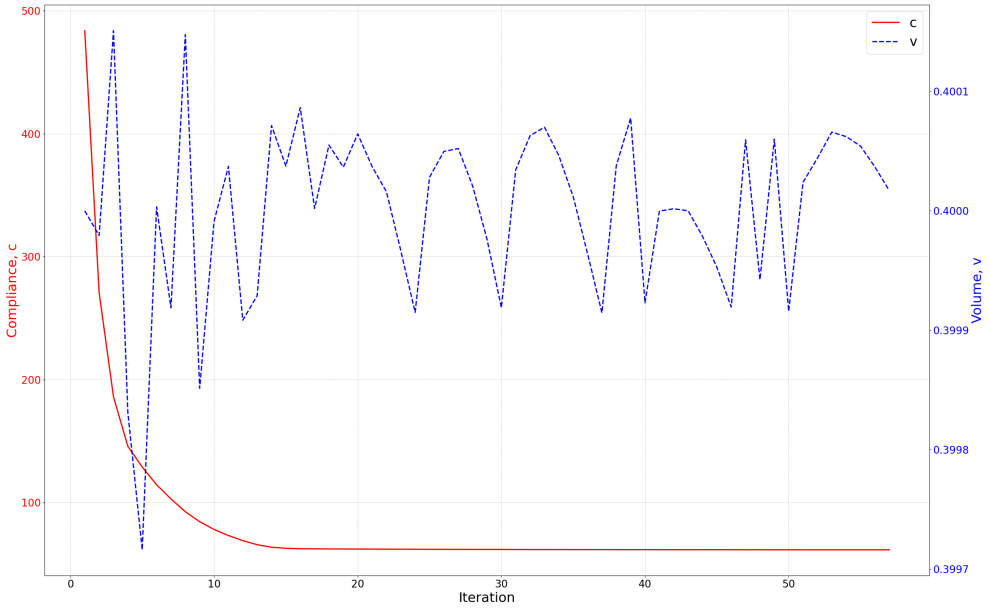
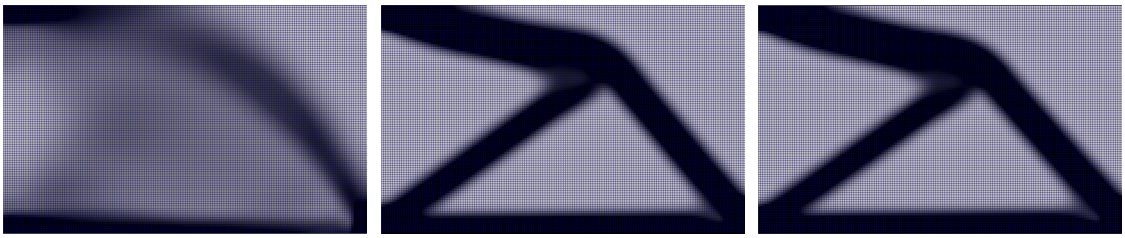


Figure 4: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam initialized with a uniform density of 0.4.

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.

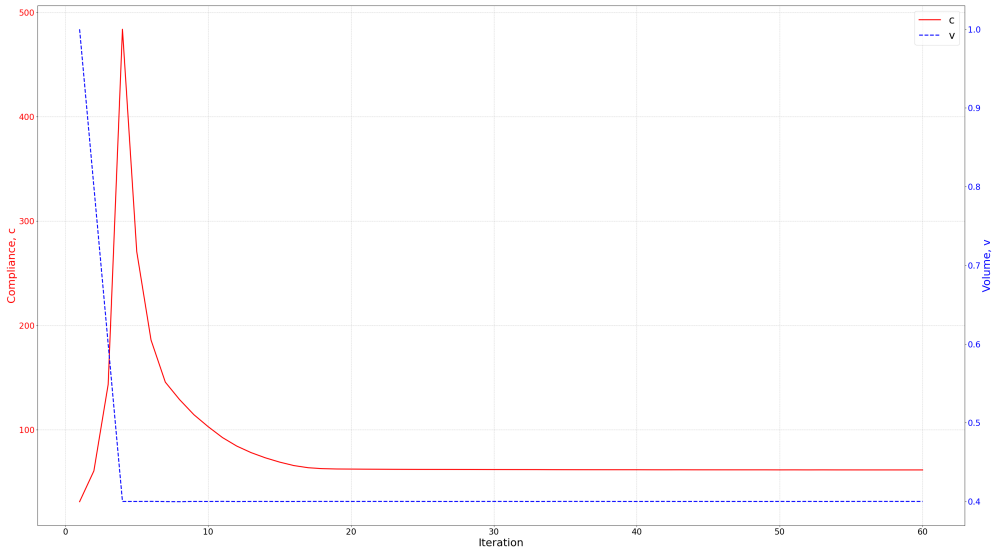
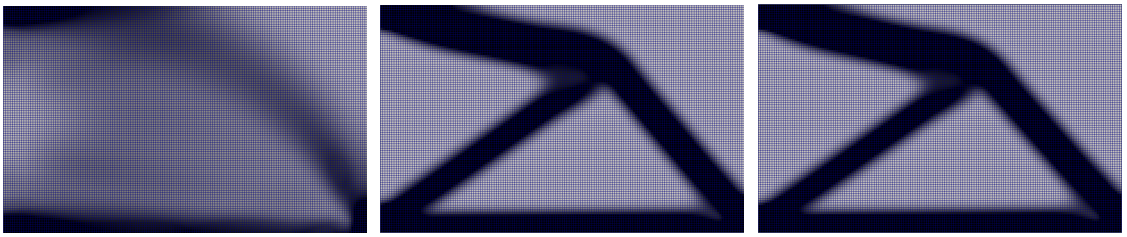


Figure 6: Convergence histories of the compliance $c(\rho)$ and volume fraction $v(\rho)$ for the 2D cantilever beam with an initial density of 1.

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

5 Geometric Decompositions of Lagrange Elements

In this section, we present a geometric decomposition for Lagrange finite elements on n-dimensional simplices. We introduce the concept of Lagrange interpolation basis functions, where function values at interpolation points serve as degrees of freedom.

The multi-backend switching mechanism in SOPTX is built upon FEALPy's *Tensor Backend Manager*, which unifies tensor operation interfaces and ensures a clear separation between high-level algorithms and low-level implementations. Users can switch between different computational backends seamlessly through a simple configuration, without modifying any algorithmic code. This design significantly reduces the learning curve for new users and lowers the complexity of maintenance and future extensions. Moreover, the multi-backend support enables SOPTX to be easily deployed across various hardware environments. Whether on CPUs or GPUs, from standalone systems to cluster environments, the framework maintains a consistent interface, greatly improving user productivity and software portability.

5.1 Geometric decomposition

For the polynomial space $\mathbb{P}_k(T)$ with $k\geq 1$ on an n-dimensional simplex T, we have the following geometric decomposition of Lagrange element [**?**, (2.6)] and a proof can be found in [**?**]. The integral at a vertex is understood as the function value at that vertex and $\mathbb{P}_k(\boldsymbol{x})=\mathbb{R}$.

Theorem 5.1 (Geometric decomposition of Lagrange element)**.** *For the polynomial space $\mathbb{P}_k(T)$ with $k\geq 1$ on an n-dimensional simplex T, we have the following decomposition*

$$\mathbb{P}_k(T)=\bigoplus_{\ell=0}^{n}\bigoplus_{f\in\Delta_\ell(T)} b_f\mathbb{P}_{k-(\ell+1)}(f). \tag{5.1}$$

The function $u\in\mathbb{P}_k(T)$ is uniquely determined by DoFs

$$\int_f up\,\mathrm{d}s, \qquad p\in\mathbb{P}_{k-(\ell+1)}(f), f\in\Delta_\ell(T), \ell=0,1,\ldots,n. \tag{5.2}$$

Introduce the bubble polynomial space of degree k on a sub-simplex f as

$$\mathbb{B}_k(f) := b_f \mathbb{P}_{k-(\ell+1)}(f), \quad f \in \Delta_\ell(T), 1 \le \ell \le n.$$

It is called a bubble space as

$$\mathrm{tr}^{\mathrm{grad}} u := u|_{\partial f} = 0, \quad u \in \mathbb{B}_k(f).$$

Then we can write (5.1) as

$$\mathbb{P}_k(T) = \mathbb{P}_1(T) \oplus \bigoplus_{\ell=1}^{n} \bigoplus_{f \in \Delta_\ell(T)} \mathbb{B}_k(f). \tag{5.3}$$

That is a polynomial of degree k can be decomposed into a linear polynomial plus bubbles on edges, faces, and all sub-simplexes.

Based on a conforming triangulation $\mathcal{T}_h$, the k-th order Lagrange finite element space $V_k^{\mathrm{L}}(\mathcal{T}_h)$ is defined as

$$V_k^{\mathrm{L}}(\mathcal{T}_h) = \{v \in C(\Omega) : v|_T \in \mathbb{P}_k(T), T \in \mathcal{T}_h, \text{ and DoFs (5.2) are single valued}\},$$

and will have a geometric decomposition

$$V_k^{\mathrm{L}}(\mathcal{T}_h) = V_1^{\mathrm{L}}(\mathcal{T}_h) \oplus \bigoplus_{\ell=1}^{n} \bigoplus_{f \in \Delta_\ell(\mathcal{T}_h)} \mathbb{B}_k(f). \tag{5.4}$$

Here we extend the polynomial on f to each element T containing f by the Bernstein form and extension of multi-index; see $E(\alpha)$ defined in (**??**). Consequently the dimension of V_k^{L} is

$$V_k^{\mathrm{L}}(\mathcal{T}_h) = \sum_{\ell=0}^{n} |\Delta_\ell(\mathcal{T}_h)| \binom{k-1}{\ell},$$

where $|\Delta_\ell(\mathcal{T}_h)|$ is the cardinality of number of $\Delta_\ell(\mathcal{T}_h)$, i.e., the number of ℓ-dimensional simplices in $\mathcal{T}_h$. We understand $\binom{k-1}{\ell} = 0$ if $\ell > k-1$. That is the degree of the polynomial dictates the dimension of the sub-simplex in the geometric decompositions (5.1) and (5.4).

The geometric decomposition (5.1) can be naturally extended to vector Lagrange elements. For $k \ge 1$, define

$$\mathbb{B}_k^n(f) := b_f \mathbb{P}_{k-(\ell+1)}(f) \otimes \mathbb{R}^n.$$

Clearly we have

$$\mathbb{P}_k^n(T) = \mathbb{P}_1^n(T) \oplus \bigoplus_{\ell=1}^{n} \bigoplus_{f \in \Delta_\ell(T)} \mathbb{B}_k^n(f). \tag{5.5}$$

For an $f \in \Delta_\ell(T)$, we choose a $t-n$ coordinate $\{\boldsymbol{t}_i^f, \boldsymbol{n}_j^f, i = 1, \ldots, \ell, j = 1, \ldots, n-\ell\}$ so that

- $\mathscr{T}^f := \mathrm{span}\{\boldsymbol{t}_1^f, \ldots, \boldsymbol{t}_\ell^f\}$, is the tangential plane of f;
- $\mathscr{N}^f := \mathrm{span}\{\boldsymbol{n}_1^f, \ldots, \boldsymbol{n}_{n-\ell}^f\}$ is the normal plane of f.

When $\ell=0$, i.e., for vertices, no tangential component, and for $\ell=n$, no normal component. We abbreviate $\boldsymbol{n}_1^F$ as $\boldsymbol{n}_F$ for $F\in\Delta_{n-1}(T)$, and $\boldsymbol{t}_1^e$ as $\boldsymbol{t}_e$ for $e\in\Delta_1(T)$. We have the trivial decompositions

$$\mathbb{R}^n=\mathscr{T}^f\oplus\mathscr{N}^f,\quad \mathbb{B}_k^n(f)=\Big[\mathbb{B}_k(f)\otimes\mathscr{T}^f\Big]\bigoplus\Big[\mathbb{B}_k(f)\otimes\mathscr{N}^f\Big]. \tag{5.6}$$

Restricted to an ℓ-dimensional sub-simplex $f\in\Delta_\ell(T)$, define

$$\mathbb{B}_k^\ell(f):=\mathbb{B}_k(f)\otimes\mathscr{T}^f,$$

which is a space of ℓ-dimensional vectors on the tangential space $\mathscr{T}^f$ with vanishing trace $\mathrm{tr}^{\mathrm{grad}}$ on ∂f.

When move to a triangulation $\mathcal{T}_h$, we shall call a basis of $\mathscr{T}^f$ or $\mathscr{N}^f$ is global if it depends only on f not the element T containing f. Otherwise it is called local and may vary in different elements.

5.2 Lagrange interpolation basis functions

Previously DoFs (5.2) are given by moments on sub-simplexes. Now we present a set of DoFs as function values on the interpolation points and give its dual basis for the k-th order Lagrange element on an n-simplex.

Lemma 5.1 (Lagrange interpolation basis functions [**?**])**.** *A basis function of the k-th order Lagrange finite element space on T is:*

$$\phi_{\boldsymbol{\alpha}}(\boldsymbol{x})=\frac{1}{\boldsymbol{\alpha}!}\prod_{i=0}^{n}\prod_{j=0}^{\alpha_i-1}(k\lambda_i(\boldsymbol{x})-j),\quad \boldsymbol{\alpha}\in\mathbb{T}_k^n,$$

with the DoFs defined as the function value at the interpolation points:

$$N_{\boldsymbol{\alpha}}(u)=u(\boldsymbol{x}_\alpha),\quad \boldsymbol{x}_\alpha\in\mathcal{X}_T.$$

Proof. It is straightforward to verify the duality of the basis and DoFs

$$N_{\boldsymbol{\beta}}(\phi_{\boldsymbol{\alpha}})=\phi_{\boldsymbol{\alpha}}(\boldsymbol{x}_\beta)=\delta_{\boldsymbol{\alpha},\boldsymbol{\beta}}=\begin{cases}1 & \text{if } \boldsymbol{\alpha}=\boldsymbol{\beta}\\ 0 & \text{otherwise}\end{cases}.$$

As

$$|\mathbb{T}_k^n|=\binom{n+k}{k}=\dim\mathbb{P}_k(T),$$

$\{\phi_{\boldsymbol{\alpha}},\boldsymbol{\alpha}\in\mathbb{T}_k^n\}$ is a basis of $\mathbb{P}_k(T)$ and $\{N_{\boldsymbol{\alpha}},\boldsymbol{\alpha}\in\mathbb{T}_k^n\}$ is a basis of the dual space $\mathbb{P}_k^*(T)$. □

Given a triangulation $\mathcal{T}_h$ and degree k, recall that

$$\mathcal{X}_{\mathcal{T}_h}=\bigcup_{T\in\mathcal{T}_h}\mathcal{X}_T=\bigoplus_{\ell=0}^n\bigoplus_{f\in\Delta_\ell(\mathcal{T}_h)}\mathcal{X}_{\mathring{f}}.$$

Denote by

$$\mathbb{T}_k^n(\mathcal{T}_h) := \bigoplus_{\boldsymbol{x}_i \in \Delta_0(\mathcal{T}_h)} \mathbb{T}_k^0(\boldsymbol{x}_i) \oplus \bigoplus_{\ell=1}^n \bigoplus_{f \in \Delta_\ell(\mathcal{T}_h)} \mathring{\mathbb{T}}_k^\ell(\mathring{f}).$$

For a lattice $\alpha \in \mathring{\mathbb{T}}_k^\ell(\mathring{f})$, we use extension operator E defined in (**??**) to extend α to each simplex T containing f. We also extend the polynomial on f to T by the Bernstein form.

Theorem 5.2 (DoFs of Lagrange finite element on $\mathcal{T}_h$)**.** *A basis for the k-th Lagrange finite element space $V_k^{\mathrm{L}}(\mathcal{T}_h)$ is*

$$\{\phi_{\boldsymbol{\alpha}}, \boldsymbol{\alpha} \in \mathbb{T}_k^n(\mathcal{T}_h)\}$$

with DoFs

$$N_{\boldsymbol{\alpha}}(u) = u(\boldsymbol{x}_\alpha), \quad \boldsymbol{x}_\alpha \in \mathcal{X}_{\mathcal{T}_h}.$$

Proof. For $F \in \Delta_{n-1}(\mathcal{T}_h)$, thanks to Lemma 5.1, $\phi_{\boldsymbol{\alpha}}|_F$ is uniquely determined by DoFs $\{u(\boldsymbol{x}_\alpha), \boldsymbol{x}_\alpha \in \mathcal{X}_F\}$ on face F, hence $\phi_{\boldsymbol{\alpha}} \in V_k^{\mathrm{L}}(\mathcal{T}_h)$. Clearly the cardinality of $\{\phi_{\boldsymbol{\alpha}}, \boldsymbol{\alpha} \in \mathbb{T}_k^n(\mathcal{T}_h)\}$ is same as the dimension of space $V_k^{\mathrm{L}}(\mathcal{T}_h)$. Then we only need to show these functions are linearly independent, which follows from the fact $N_{\boldsymbol{\beta}}(\phi_{\boldsymbol{\alpha}}) = \phi_{\boldsymbol{\alpha}}(\boldsymbol{x}_\beta) = \delta_{\boldsymbol{\alpha},\boldsymbol{\beta}}$ for $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{T}_k^n(\mathcal{T}_h)$. □

We now generalize the basis for a scalar Lagrange element to a vector Lagrange element. For an interpolation point $\boldsymbol{x} \in \mathcal{X}_T$, let $\{\boldsymbol{e}_i^{\boldsymbol{x}}, i = 0, \ldots, n-1\}$ be a basis of $\mathbb{R}^n$, and its dual basis is denoted by $\{\hat{\boldsymbol{e}}_i^{\boldsymbol{x}}, i = 0, \ldots, n-1\}$, i.e.,

$$(\hat{\boldsymbol{e}}_i^{\boldsymbol{x}}, \boldsymbol{e}_j^{\boldsymbol{x}}) = \delta_{i,j}.$$

When $\{\boldsymbol{e}_i^{\boldsymbol{x}}, i = 0, \ldots, n-1\}$ is orthonormal, its dual basis is itself.

Corollary 5.3. A polynomial function $\boldsymbol{u} \in \mathbb{P}_k^n(T)$ can be uniquely determined by the DoFs:

$$N_{\boldsymbol{\alpha}}^i(\boldsymbol{u}) := \boldsymbol{u}(\boldsymbol{x}_{\boldsymbol{\alpha}}) \cdot \boldsymbol{e}_i^{\boldsymbol{x}_{\boldsymbol{\alpha}}}, \quad \boldsymbol{x}_{\boldsymbol{\alpha}} \in \mathcal{X}_T, i = 0, \ldots, n-1.$$

The basis function on T dual to this set of DoFs can be explicitly written as:

$$\boldsymbol{\phi}_{\boldsymbol{\alpha}}^i(\boldsymbol{x}) = \phi_{\boldsymbol{\alpha}}(\boldsymbol{x}) \hat{\boldsymbol{e}}_i^{\boldsymbol{x}_{\boldsymbol{\alpha}}}, \quad \boldsymbol{\alpha} \in \mathbb{T}_k^n, i = 0, \ldots, n-1.$$

Proof. It is straightforward to verify the duality

$$N_{\boldsymbol{\beta}}^j(\boldsymbol{\phi}_{\boldsymbol{\alpha}}^i) = \boldsymbol{\phi}_{\boldsymbol{\alpha}}^i(\boldsymbol{x}_\beta) \cdot \boldsymbol{e}_j^{\boldsymbol{x}_\beta} = \phi_{\boldsymbol{\alpha}}(\boldsymbol{x}_\beta) \hat{\boldsymbol{e}}_i^{\boldsymbol{x}_{\boldsymbol{\alpha}}} \cdot \boldsymbol{e}_j^{\boldsymbol{x}_\beta} = \delta_{i,j} \delta_{\boldsymbol{\alpha},\boldsymbol{\beta}},$$

for $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{T}_k^n, i, j = 0, \ldots, n-1$. □

If the basis $\{\boldsymbol{e}_i^f, i = 0, \ldots, n-1\}$ is global in the sense that it is independent of element T containing $\boldsymbol{x}$, we get the continuous vector Lagrange elements. Choose different basis will give different continuity.

6 Geometric Decompositions of Face Elements

Define $H(\mathrm{div},\Omega):=\{\boldsymbol{v}\in L^2(\Omega;\mathbb{R}^n):\mathrm{div}\,\boldsymbol{v}\in L^2(\Omega)\}$. For a subdomain $K\subseteq\Omega$, the trace operator for the div operator is

$$\mathrm{tr}_K^{\mathrm{div}}\boldsymbol{v}=\boldsymbol{n}\cdot\boldsymbol{v}|_{\partial K}\quad\text{ for }\ \boldsymbol{v}\in H(\mathrm{div},\Omega),$$

where $\boldsymbol{n}$ denotes the outwards unit normal vector of ∂K. Given a triangulation $\mathcal{T}_h$ and a piecewise smooth function $\boldsymbol{u}$, it is well known that $\boldsymbol{u}\in H(\mathrm{div},\Omega)$ if and only if $\boldsymbol{n}_F\cdot\boldsymbol{u}$ is continuous across all faces $F\in\Delta_{n-1}(\mathcal{T}_h)$, which can be ensured by having DoFs on faces. An $H(\mathrm{div})$-conforming finite element is thus also called a face element.

6.1 Geometric decomposition

Define the polynomial div bubble space

$$\mathbb{B}_k(\mathrm{div},T)=\ker(\mathrm{tr}_T^{\mathrm{div}})\cap\mathbb{P}_k^n(T).$$

Recall that $\mathbb{B}_k^\ell(f)=\mathbb{B}_k(f)\otimes\mathscr{T}^f$ consists of bubble polynomials on the tangential plane of f. For $\boldsymbol{u}\in\mathbb{B}_k^\ell(f)$, as $\boldsymbol{u}$ is on the tangent plane, $\boldsymbol{u}\cdot\boldsymbol{n}_F=0$ for $f\subseteq F$. When $f\not\subseteq F$, $b_f|_F=0$. So $\mathbb{B}_k^\ell(f)\subseteq\mathbb{B}_k(\mathrm{div},T)$ for $k\geq 2$ and $\dim f\geq 1$. In [**?**], we have proved that the div-bubble polynomial space has the following decomposition.

Lemma 6.1. *For $k\geq 2$,*

$$\mathbb{B}_k(\mathrm{div},T)=\bigoplus_{\ell=1}^n\bigoplus_{f\in\Delta_\ell(T)}\mathbb{B}_k^\ell(f).$$

Notice that as no tangential plane on vertices, there is no div-bubble associated to vertices and consequently the degree of a div-bubble polynomial is greater than or equal to 2. Next we present two geometric decompositions of a div-element.

Theorem 6.1. *For $k\geq 1$, we have*

$$\mathbb{P}_k^n(T)=\mathbb{P}_1^n(T)\oplus\Big(\bigoplus_{\ell=1}^{n-1}\bigoplus_{f\in\Delta_\ell(T)}(\mathbb{B}_k(f)\otimes\mathscr{N}^f)\Big)\oplus\mathbb{B}_k(\mathrm{div},T),\tag{6.1}$$

$$\mathbb{P}_k^n(T)=\Big(\bigoplus_{F\in\Delta_{n-1}(T)}(\mathbb{P}_k(F)\boldsymbol{n}_F)\Big)\oplus\mathbb{B}_k(\mathrm{div},T).\tag{6.2}$$

Proof. The first decomposition (6.1) is a rearrangement of (5.5) by merging the tangential component $\mathbb{B}_k^\ell(f)$ into the bubble space $\mathbb{B}_k(\mathrm{div},T)$.

Next we prove the decomposition (6.2). For an ℓ-dimensional, $0\leq\ell\leq n-1$, sub-simplex $f\in\Delta_\ell(T)$, we choose the $n-\ell$ face normal vectors $\{\boldsymbol{n}_F:F\in\Delta_{n-1}(T),f\subseteq F\}$ as the basis of $\mathscr{N}^f$. Therefore we have

$$\mathbb{B}_k(f)\otimes\mathscr{N}^f=\bigoplus_{F\in\Delta_{n-1}(T),f\subseteq F}\mathbb{B}_k(f)\boldsymbol{n}_F.$$

Then (6.1) becomes

$$\mathbb{P}_k^n(T)=\mathbb{P}_1^n(T)\oplus\Big(\bigoplus_{\ell=1}^{n-1}\bigoplus_{f\in\Delta_\ell(T)}\bigoplus_{F\in\Delta_{n-1}(T),f\subseteq F}\mathbb{B}_k(f)\boldsymbol{n}_F\Big)\oplus\mathbb{B}_k(\mathrm{div},T).$$

At a vertex v, we choose $\{\boldsymbol{n}_F : F \in \Delta_{n-1}(T), \mathrm{v} \subseteq F\}$ as the basis of $\mathbb{R}^n$ and write

$$\begin{aligned}\mathbb{P}_1^n(T) &= \bigoplus_{\boldsymbol{x}\in\Delta_0(T)} \bigoplus_{F\in\Delta_{n-1}(T),\boldsymbol{x}\in\Delta_0(F)} \operatorname{span}\{\lambda_{\boldsymbol{x}}\boldsymbol{n}_F\} \\ &= \bigoplus_{F\in\Delta_{n-1}(T)} \bigoplus_{\boldsymbol{x}\in\Delta_0(F)} \operatorname{span}\{\lambda_{\boldsymbol{x}}\boldsymbol{n}_F\} \\ &= \bigoplus_{F\in\Delta_{n-1}(T)} \mathbb{P}_1(F)\boldsymbol{n}_F.\end{aligned}$$

Then by swapping the ordering of f and F in the direct sum, i.e.,

$$\bigoplus_{\ell=1}^{n-1} \bigoplus_{f\in\Delta_\ell(T)} \bigoplus_{F\in\Delta_{n-1}(T),f\subseteq F} \to \bigoplus_{F\in\Delta_{n-1}(T)} \bigoplus_{\ell=1}^{n-1} \bigoplus_{f\in\Delta_\ell(F)},$$

and using the decomposition (5.1) of Lagrange element, we obtain the decomposition (6.2). □

In decomposition (6.1), we single out $\mathbb{P}_1^n(T)$ to emphasize an $H(\operatorname{div})$-conforming element can be obtained by adding div-bubble and normal component on sub-simplexes starting from edges. In (6.2), we group all normal components facewisely which leads to the classical BDM element.

As $H(\operatorname{div},\Omega)$-conforming elements require the normal continuity across each F, the normal vector $\boldsymbol{n}_F$ is chosen globally. Namely $\boldsymbol{n}_F$ depends on F only. It may coincide with the outwards or inwards normal vector for an element T containing F. On the contrary, all tangential basis for $\mathscr{T}^f$ are local and thus the tangential component are multi-valued and merged to the element-wise div bubble function $\mathbb{B}_k(\operatorname{div},T)$.

6.2 A nodal basis for the BDM element

For the efficient implementation, we employ the DoFs of the function values at interpolation nodes and combine with $t-n$ decompositions to present a nodal basis for the BDM element.

Given an $f \in \Delta_\ell(T)$, we choose $\{\boldsymbol{n}_F, F \in \Delta_{n-1}(T), f \subseteq F\}$ as the basis for its normal plane $\mathcal{N}^f$ and an arbitrary basis $\{\boldsymbol{t}_i^f, i=1,\ldots,\ell\}$ for the tangential plane. We shall choose $\{\hat{\boldsymbol{n}}_{F_i} \in \mathcal{N}^f, i \in f^*\}$ a basis of $\mathcal{N}^f$ dual to $\{\boldsymbol{n}_{F_i} \in \mathcal{N}^f, i \in f^*\}$, i.e.,

$$(\boldsymbol{n}_F, \hat{\boldsymbol{n}}_{F'}) = \delta_{F,F'}, \quad F, F' \in \Delta_{n-1}(T), f \in F \cap F'.$$

Similarly choose a basis $\{\hat{\boldsymbol{t}}_i^f\}$ dual to $\{\boldsymbol{t}_j^f\}$.

We can always choose an orthonormal basis for the tangential plane $\mathscr{T}^f$ but for the normal plane $\mathscr{N}^f$ with basis $\{\boldsymbol{n}_{F_i} \in \mathcal{N}^f, i \in f^*\}$, we use Lemma 6.2 to find its dual basis.

For $f \in \Delta_\ell(T)$ and $e \in \partial f$, let $\boldsymbol{n}_f^e$ be an unit normal vector of e but tangential to f. When $\ell = 1, f$ is an edge and e is a vertex. Then $\boldsymbol{n}_f^e$ is the edge vector of f. Using these notations we can give an explicit expression of the dual basis $\{\hat{\boldsymbol{n}}_{F_i} \in \mathcal{N}^f, i \in f^*\}$.

Lemma 6.2. *For $f \in \Delta_\ell(T)$,*

$$\{\hat{\boldsymbol{n}}_{F_i} = \frac{1}{\boldsymbol{n}_{f+i}^f \cdot \boldsymbol{n}_{F_i}} \boldsymbol{n}_{f+i}^f \quad i \in f^*\}, \tag{6.3}$$

where $f+i$ denotes the $(\ell+1)$-dimensional face in $\Delta_{\ell+1}(T)$ with vertices $f(0),\ldots,f(\ell)$ and i for $i\in f^*$, is a basis of $\mathcal{N}^f$ dual to $\{\boldsymbol{n}_{F_i}\in\mathcal{N}^f, i\in f^*\}$.

Proof. Clearly $\boldsymbol{n}^f_{f+i}\in\mathcal{N}^f$ for $i\in f^*$. It suffices to prove

$$\boldsymbol{n}^f_{f+i}\cdot\boldsymbol{n}_{F_j}=0 \quad \text{for } i,j\in f^*, i\neq j,$$

which follows from $\boldsymbol{n}^f_{f+i}\in\mathscr{T}^{f+i}$ and $f+i\subseteq F_j$. □

By Corollary 5.3, we obtain a nodal basis for BDM face elements.

Theorem 6.2. *For each $f\in\Delta_\ell(T)$, we choose $\{\boldsymbol{e}^f_i, i=0,\ldots,n-1\}=\{\boldsymbol{t}^f_i, i=1,\ldots,\ell,\ \boldsymbol{n}_{F_i}, i\in f^*\}$ and its dual basis $\{\hat{\boldsymbol{e}}^f_i, i=0,\ldots,n-1\}=\{\hat{\boldsymbol{t}}^f_i, i=1,\ldots,\ell,\ \hat{\boldsymbol{n}}_{F_i}, i\in f^*\}$. A basis function of the k-th order BDM element space on T is:*

$$\{\phi_\alpha(\boldsymbol{x})\hat{\boldsymbol{e}}^f_i, i=0,1,\ldots,n-1,\ \alpha\in\mathbb{T}^\ell_k(\mathring{f})\}_{f\in\Delta_\ell(T),\ell=0,1,\ldots,n},$$

with the DoFs at the interpolation points defined as:

$$\{u(\boldsymbol{x}_\alpha)\boldsymbol{e}^f_i, i=0,1,\ldots,n-1,\ \boldsymbol{x}_\alpha\in\mathcal{X}_{\mathring{f}}\}_{f\in\Delta_\ell(T),\ell=0,1,\ldots,n}. \tag{6.4}$$

By choosing a global normal basis in the sense that $\boldsymbol{n}_F$ depending only on F not the element containing F, we impose the continuity on the normal direction. We choose a local $\boldsymbol{t}^f$, i.e., for different element T containing f, $\phi_\alpha(\boldsymbol{x})\boldsymbol{t}^f(T)$ is different, then no continuity is imposed for the tangential direction.

Define the global finite element space

$$\begin{aligned}V_h^{\mathrm{div}}:=\{&\boldsymbol{u}\in L^2(\Omega;\mathbb{R}^n):\boldsymbol{u}|_T\in\mathbb{P}_k(T;\mathbb{R}^n) \text{ for each } T\in\mathcal{T}_h,\\ &\text{all the DoFs } u(\boldsymbol{x}_\alpha)\boldsymbol{n}_F \text{ in (6.4) across } F\in\Delta_{n-1}(\mathcal{T}_h) \text{ are single-valued}\}.\end{aligned}$$

By Theorem 6.1, $V_h^{\mathrm{div}}\subset H(\mathrm{div},\Omega)$ and is equivalent to the BDM space.

The novelty is that we only need a basis of Lagrange element which is well documented; see Section 5.2. Coupling with different $t-n$ decomposition at different sub-simplex, we obtain the classical face elements. This concept has been explored in the works of [**?**, **?**] and [**?**], focusing on the implementation of the $H(\mathrm{div})$ element in two and three dimensions. Additionally, the adaptation of a 2D $H(\mathrm{curl})$ element through rotation is discussed in [**?**,**?**]. As we will demonstrate in the subsequent section, extending the edge element to higher dimensions presents significant challenges. This extension necessitates a comprehensive characterization of the curl operator and its associated polynomial bubble space.

7 Geometric Decompositions of Edge Elements

In this section we present geometric decompositions of $H(\mathrm{curl})$-conforming finite element space on an n-dimensional simplex. We first generalize the curl differential operator to n dimensions and study its trace which motivates our decomposition. We then give an explicit basis dual to function values at interpolation points.

7.1 Differential operator and its trace

Denote by $\mathbb{S}$ and $\mathbb{K}$ the subspace of symmetric matrices and skew-symmetric matrices of $\mathbb{R}^{n\times n}$, respectively. For a smooth vector function $\boldsymbol{v}$, define

$$\operatorname{curl}\boldsymbol{v} := 2\operatorname{skw}(\operatorname{grad}\boldsymbol{v}) = \operatorname{grad}\boldsymbol{v} - (\operatorname{grad}\boldsymbol{v})^{\intercal},$$

which is a skew-symmetric matrix function. In two dimensions, for $\boldsymbol{v} = (v_1, v_2)^{\intercal}$,

$$\operatorname{curl}\boldsymbol{v} = \begin{pmatrix} 0 & \partial_{x_2} v_1 - \partial_{x_1} v_2 \\ \partial_{x_1} v_2 - \partial_{x_2} v_1 & 0 \end{pmatrix} = \operatorname{mskw}(\operatorname{rot}\boldsymbol{v}),$$

where $\operatorname{mskw} u := \begin{pmatrix} 0 & -u \\ u & 0 \end{pmatrix}$ and $\operatorname{rot}\boldsymbol{v} := \partial_{x_1} v_2 - \partial_{x_2} v_1$. In three dimensions, for $\boldsymbol{v} = (v_1, v_2, v_3)^{\intercal}$,

$$\operatorname{curl}\boldsymbol{v} = \begin{pmatrix} 0 & \partial_{x_2} v_1 - \partial_{x_1} v_2 & \partial_{x_3} v_1 - \partial_{x_1} v_3 \\ \partial_{x_1} v_2 - \partial_{x_2} v_1 & 0 & \partial_{x_3} v_2 - \partial_{x_2} v_3 \\ \partial_{x_1} v_3 - \partial_{x_3} v_1 & \partial_{x_2} v_3 - \partial_{x_3} v_2 & 0 \end{pmatrix} = \operatorname{mskw}(\nabla \times \boldsymbol{v}),$$

where $\operatorname{mskw}\boldsymbol{u} := \begin{pmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{pmatrix}$ with $\boldsymbol{u} = (u_1, u_2, u_3)^{\intercal}$. Hence we can identify $\operatorname{curl}\boldsymbol{v}$ as scalar $\operatorname{rot}\boldsymbol{v}$ in two dimensions, and vector $\nabla \times \boldsymbol{v}$ in three dimensions. In general, $\operatorname{curl}\boldsymbol{u}$ is understood as a skew-symmetric matrix.

Define Sobolev space

$$H(\operatorname{curl}, \Omega) := \{\boldsymbol{v} \in L^2(\Omega; \mathbb{R}^n) : \operatorname{curl}\boldsymbol{v} \in L^2(\Omega; \mathbb{K})\}.$$

Given a face $F \in \Delta_{n-1}(T)$, define the trace operator of curl as

$$\operatorname{tr}_F^{\operatorname{curl}} \boldsymbol{v} = 2\operatorname{skw}(\boldsymbol{v}\boldsymbol{n}_F^{\intercal})|_F = (\boldsymbol{v}\boldsymbol{n}_F^{\intercal} - \boldsymbol{n}_F\boldsymbol{v}^{\intercal})|_F.$$

We define $\operatorname{tr}^{\operatorname{curl}}$ as a piecewise defined operator as

$$(\operatorname{tr}^{\operatorname{curl}} \boldsymbol{v})|_F = \operatorname{tr}_F^{\operatorname{curl}} \boldsymbol{v}, \quad F \in \Delta_{n-1}(T).$$

For a vector $\boldsymbol{v} \in \mathbb{R}^n$ and an $n-1$ dimensional face F, the tangential part of $\boldsymbol{v}$ on F is

$$\Pi_F \boldsymbol{v} := \boldsymbol{v}|_F - (\boldsymbol{v}|_F \cdot \boldsymbol{n}_F)\boldsymbol{n}_F = \sum_{i=1}^{n-1} (\boldsymbol{v}|_F \cdot \boldsymbol{t}_i^F)\boldsymbol{t}_i^F,$$

where $\{\boldsymbol{t}_i^F, i = 1, \ldots, n-1\}$ is an orthonormal basis of F. As we treat $\operatorname{curl}\boldsymbol{v}$ as a matrix, so is the trace $\operatorname{tr}_F^{\operatorname{curl}}\boldsymbol{v}$, while the tangential component of $\boldsymbol{v}$ is a vector. Their relation is given in the following lemma.

Lemma 7.1. *For face $F\in\Delta_{n-1}(T)$, we have*

$$\mathrm{tr}_F^{\mathrm{curl}}\boldsymbol{v}=2\mathrm{skw}\big((\Pi_F\boldsymbol{v})\boldsymbol{n}_F^{\intercal}\big),\quad \Pi_F\boldsymbol{v}=(\mathrm{tr}_F^{\mathrm{curl}}\boldsymbol{v})\boldsymbol{n}_F. \tag{7.1}$$

Proof. By the decomposition $\boldsymbol{v}|_F=\Pi_F\boldsymbol{v}+(\boldsymbol{v}|_F\cdot\boldsymbol{n}_F)\boldsymbol{n}_F$,

$$\mathrm{tr}_F^{\mathrm{curl}}\boldsymbol{v}=2\mathrm{skw}\big((\Pi_F\boldsymbol{v})\boldsymbol{n}_F^{\intercal}+(\boldsymbol{v}|_F\cdot\boldsymbol{n}_F)\boldsymbol{n}_F\boldsymbol{n}_F^{\intercal}\big)=2\mathrm{skw}\big((\Pi_F\boldsymbol{v})\boldsymbol{n}_F^{\intercal}\big),$$

which implies the first identity. Then by $\boldsymbol{n}_F^{\intercal}\boldsymbol{n}_F=1$ and $(\Pi_F\boldsymbol{v})^{\intercal}\boldsymbol{n}_F=0$,

$$(\mathrm{tr}_F^{\mathrm{curl}}\boldsymbol{v})\boldsymbol{n}_F=\big((\Pi_F\boldsymbol{v})\boldsymbol{n}_F^{\intercal}-\boldsymbol{n}_F(\Pi_F\boldsymbol{v})^{\intercal}\big)\boldsymbol{n}_F=\Pi_F\boldsymbol{v},$$

i.e. the second identity holds. □

Thanks to (7.1), the vanishing tangential part $\Pi_F\boldsymbol{v}$ and the vanishing tangential trace $(\mathrm{tr}_F^{\mathrm{curl}}\boldsymbol{v})$ are equivalent.

Lemma 7.2. *Let $\boldsymbol{v}\in L^2(\Omega;\mathbb{R}^n)$ and $\boldsymbol{v}|_T\in H^1(T;\mathbb{S})$ for each $T\in\mathcal{T}_h$. Then $\boldsymbol{v}\in H(\mathrm{curl},\Omega)$ if and only if $\Pi_F\boldsymbol{v}|_{T_1}=\Pi_F\boldsymbol{v}|_{T_2}$ for all interior face $F\in\Delta_{n-1}(\mathcal{T}_h)$, where T_1 and T_2 are two elements sharing F.*

Proof. It is an immediate result of Lemma 5.1 in [**?**] and (7.1). □

7.2 Polynomial bubble space

Define the polynomial bubble space for the curl operator as

$$\mathbb{B}_k(\mathrm{curl},T)=\ker(\mathrm{tr}^{\mathrm{curl}})\cap\mathbb{P}_k^n(T).$$

For Lagrange bubble space $\mathbb{B}_k^n(T)$, all components of the vector function vanish on ∂T and consequently on all sub-simplex with dimension $\le n-1$. For $\boldsymbol{u}\in\mathbb{B}_k(\mathrm{curl},T)$, only the tangential component vanishes, which will imply $\boldsymbol{u}$ vanishes on sub-simplex with dimension $\le n-2$.

Lemma 7.3. *For $\boldsymbol{u}\in\mathbb{B}_k(\mathrm{curl},T)$, it holds $\boldsymbol{u}|_f=\boldsymbol{0}$ for all $f\in\Delta_\ell(T),0\le\ell\le n-2$. Consequently $\mathbb{B}_k(\mathrm{curl},T)\subset\mathbb{B}_k^n(T)\oplus\bigoplus_{F\in\Delta_{n-1}(T)}\mathbb{B}_k^n(F)$.*

Proof. It suffices to consider a sub-simplex $f\in\Delta_{n-2}(T)$. Let $F_1,F_2\in\Delta_{n-1}(T)$ such that $f=F_1\cap F_2$. By $\mathrm{tr}_{F_i}^{\mathrm{curl}}\boldsymbol{u}=\boldsymbol{0}$ for $i=1,2$, we have $\Pi_{F_i}\boldsymbol{u}=0$ and consequently

$$(\boldsymbol{u}\cdot\boldsymbol{t}_i^f)|_f=0,\ (\boldsymbol{u}\cdot\boldsymbol{n}_{F_1}^f)|_f=(\boldsymbol{u}\cdot\boldsymbol{n}_{F_2}^f)|_f=0\quad\text{for } i=1,\dots,n-2,$$

where $\boldsymbol{n}_{F_i}^f$ is a normal vector f sitting on F_i. As $\mathrm{span}\{\boldsymbol{t}_1^f,\dots,\boldsymbol{t}_{n-2}^f,\boldsymbol{n}_{F_1}^f,\boldsymbol{n}_{F_2}^f\}=\mathbb{R}^n$, we acquire $\boldsymbol{u}|_f=\boldsymbol{0}$. By the property of face bubbles, we conclude $\boldsymbol{u}$ is a linear combination of element bubble and $n-1$ face bubbles. □

Obviously $\mathbb{B}_k^n(T)\subset\mathbb{B}_k(\mathrm{curl},T)$. As $\mathrm{tr}^{\mathrm{curl}}$ contains the tangential component only, the normal component $\mathbb{B}_k(F)\boldsymbol{n}_F$ is also a curl bubble. The following result says their sum is precisely all curl bubble polynomials.

Theorem 7.1. *For $k\geq 1$, it holds that*

$$\mathbb{B}_k(\mathrm{curl},T)=\mathbb{B}_k^n(T)\oplus\bigoplus_{F\in\Delta_{n-1}(T)}\mathbb{B}_k(F)\boldsymbol{n}_F, \tag{7.2}$$

and

$$\mathrm{tr}^{\mathrm{curl}}:\mathbb{P}_1^n(T)\oplus\bigoplus_{\ell=1}^{n-2}\bigoplus_{f\in\Delta_\ell(T)}\mathbb{B}_k^n(f)\oplus\bigoplus_{F\in\Delta_{n-1}(T)}\mathbb{B}_k^{n-1}(F)\to\mathrm{tr}^{\mathrm{curl}}\mathbb{P}_k^n(T) \tag{7.3}$$

is a bijection.

Proof. It is obvious that

$$\mathbb{B}_k^n(T)\oplus\bigoplus_{F\in\Delta_{n-1}(T)}\mathbb{B}_k(F)\boldsymbol{n}_F\subseteq\mathbb{B}_k(\mathrm{curl},T).$$

Then apply the trace operator to the decomposition (5.5) to conclude that the map $\mathrm{tr}^{\mathrm{curl}}$ in (7.3) is onto.

Now we prove it is also injective. Take a function $\boldsymbol{u}\in\mathbb{P}_1^n(T)\oplus\bigoplus_{\ell=1}^{n-2}\bigoplus_{f\in\Delta_\ell(T)}\mathbb{B}_k^n(f)\oplus\bigoplus_{F\in\Delta_{n-1}(T)}\mathbb{B}_k^{n-1}(F)$ and $\mathrm{tr}^{\mathrm{curl}}\boldsymbol{u}=\boldsymbol{0}$. By Lemma 7.3, we can assume $\boldsymbol{u}=\sum_{F\in\Delta_{n-1}(T)}\boldsymbol{u}_k^F$ with $\boldsymbol{u}_k^F\in\mathbb{B}_k^{n-1}(F)$. Take $F\in\Delta_{n-1}(T)$. We have $\boldsymbol{u}|_F=\boldsymbol{u}_k^F|_F\in\mathbb{B}_k^{n-1}(F)$. Hence $(\boldsymbol{u}_k^F\cdot\boldsymbol{t})|_F=(\boldsymbol{u}\cdot\boldsymbol{t})|_F=0$ for any $\boldsymbol{t}\in\mathscr{T}^F$, which results in $\boldsymbol{u}_k^F=\boldsymbol{0}$. Therefore $\boldsymbol{u}=\boldsymbol{0}$.

Once we have proved the map tr in (7.3) is bijection, we conclude (7.2) from the decomposition (5.5). □

We will use curl_f to denote the curl operator restricted to a sub-simplex f with $\dim f\geq 1$. For $f\in\Delta_\ell(T),\ell=2,\ldots,n-1$, by applying Theorem 7.1 to f, we have

$$\mathbb{B}_k(\mathrm{curl}_f,f)=\mathbb{B}_k^\ell(f)\oplus\bigoplus_{e\in\partial f}\mathbb{B}_k(e)\boldsymbol{n}_f^e. \tag{7.4}$$

Notice that the curl_f-bubble function is defined for $\ell\geq 2$ not including edges. Indeed, for an edge e and a vertex $\boldsymbol{x}$ of e, $\boldsymbol{n}_e^{\boldsymbol{x}}$ is $\boldsymbol{t}_e$ if $\boldsymbol{x}$ is the ending vertex of e and $-\boldsymbol{t}_e$ otherwise. Then for $\ell=1$

$$\mathbb{B}_k(e)\boldsymbol{t}_e\oplus\bigoplus_{\boldsymbol{x}\in\partial e}\mathrm{span}\{\lambda_{\boldsymbol{x}}\boldsymbol{n}_e^{\boldsymbol{x}}\}=\mathbb{P}_k(e)\boldsymbol{t}_e. \tag{7.5}$$

which is the full polynomial not vanishing on ∂e.

7.3 Geometric decompositions

For $e\in\Delta_\ell(T)$, we choose the basis $\{\boldsymbol{n}_f^e:f=e+i,i\in e^*\}$ for $\mathscr{N}^e$ and a basis $\{\boldsymbol{t}_i^e,i=1,\ldots,\ell\}$ for $\mathscr{T}^e$. So we have the following geometric decompositions of $\mathbb{P}_k^n(T)$.

Theorem 7.2. *For $k\geq 1$, we have*

$$\mathbb{P}_k^n(T)=\mathbb{P}_1^n(T)\oplus\bigoplus_{\ell=1}^n\bigoplus_{e\in\Delta_\ell(T)}\mathbb{B}_k(e)\otimes\operatorname{span}\{\boldsymbol{t}_i^e\}_{i=1}^\ell\oplus\mathbb{B}_k(e)\otimes\operatorname{span}\{\boldsymbol{n}_{e+i}^e,i\in e^*\},\quad(7.6)$$

$$\mathbb{P}_k^n(T)=\bigoplus_{e\in\Delta_1(T)}\mathbb{P}_k(e)\boldsymbol{t}_e\oplus\bigoplus_{\ell=2}^n\bigoplus_{f\in\Delta_\ell(T)}\mathbb{B}_k(\operatorname{curl}_f,f).\quad(7.7)$$

Proof. Decomposition (7.6) is the component form of decomposition (5.5). We can write $\mathbb{B}_k(e)\otimes\operatorname{span}\{\boldsymbol{n}_{e+i}^e,i\in e^*\}=\bigoplus_{f\in\Delta_{\ell+1}(T),e\subseteq f}\mathbb{B}_k(e)\boldsymbol{n}_f^e$. Then in the summation (7.6), we shift the normal component one level up and switch the sum of e and f:

$$\begin{aligned}&\bigoplus_{\ell=1}^n\bigoplus_{e\in\Delta_\ell(T)}\big[\mathbb{B}_k^\ell(e)\bigoplus_{f\in\Delta_{\ell+1}(T),e\subseteq f}\mathbb{B}_k(e)\boldsymbol{n}_f^e\big]\\=&\bigoplus_{e\in\Delta_1(T)}\mathbb{B}_k(e)\boldsymbol{t}_e\oplus\bigoplus_{\ell=2}^n\bigoplus_{f\in\Delta_\ell(T)}\big[\mathbb{B}_k^\ell(f)\bigoplus_{e\subseteq\partial f}\mathbb{B}_k(e)\boldsymbol{n}_f^e\big].\end{aligned}$$

Then by the characterization (7.4) of $\mathbb{B}_k(\operatorname{curl}_f,f)$ and (7.5), we get the decomposition (7.7). □

Decomposition (7.7) is the counterpart of (5.3) for Lagrange element.

7.4 Tangential-Normal decomposition of the second family of edge elements

Recall that for $e\in\Delta_{\ell-1}(T)$, the basis $\{\boldsymbol{n}_f^e:f\in\Delta_\ell(T),e\subseteq f\}$ of $\mathscr{N}^e$ is dual to the basis $\{\boldsymbol{n}_F:F\in\Delta_{n-1}(T),e\subseteq F\}$; see Lemma 6.2.

Theorem 7.3. *Take $\mathbb{P}_k^n(T)$ as the shape function space. Then it is determined by the following DoFs*

$$\int_e\boldsymbol{u}\cdot\boldsymbol{t}\,p\,\mathrm{d}s,\quad p\in\mathbb{P}_k(e),e\in\Delta_1(T),\quad(7.8a)$$

$$\int_f\boldsymbol{u}\cdot\boldsymbol{p}\mathrm{d}s,\quad\boldsymbol{p}\in\mathbb{B}_k(\operatorname{curl}_f,f),f\in\Delta_\ell(T),\ell=2,\dots,n\quad(7.8b)$$

Proof. Based on the decomposition (7.6), the shape function $\mathbb{P}_k^n(T)$ is determined by the following DoFs

$$\int_e\boldsymbol{u}\cdot\boldsymbol{t}\,p\,\mathrm{d}s,\quad p\in\mathbb{P}_k(e),e\in\Delta_1(T),\quad(7.9a)$$

$$\int_f(\boldsymbol{u}\cdot\boldsymbol{t}_i^f)\,p\,\mathrm{d}s,\quad p\in\mathbb{P}_{k-(\ell+1)}(f),i=1,\dots,\ell,\quad(7.9b)$$

$$\int_e(\boldsymbol{u}\cdot\boldsymbol{n}_f^e)\,p\,\mathrm{d}s,\quad p\in\mathbb{P}_{k-\ell}(e),e\in\partial f\quad(7.9c)$$

for $f\in\Delta_\ell(T),\ell=2,\dots,n$. By (7.2) and (7.4), DoFs (7.9) are equivalent to DoFs (7.8). □

Remark 7.1. The DoFs of the second kind Nédélec edge element in [**?**, **?**] are

$$\int_e\boldsymbol{u}\cdot\boldsymbol{t}\,p\,\mathrm{d}s,\quad p\in\mathbb{P}_k(e),e\in\Delta_1(T),$$

$$\int_f\boldsymbol{u}\cdot\boldsymbol{p}\mathrm{d}s,\quad\boldsymbol{p}\in\mathbb{P}_{k-\ell}^\ell(f)+(\Pi_f\boldsymbol{x})\mathbb{P}_{k-\ell}(f),f\in\Delta_\ell(T),\ell=2,\dots,n.$$

There is an isomorphism between $\mathbb{P}^{\ell}_{k-\ell}(f)+(\Pi_f\boldsymbol{x})\mathbb{P}_{k-\ell}(f)$ and $\mathbb{B}_k(\mathrm{curl}_f,f)$ for $f\in\Delta_\ell(T)$ with $\ell=2,\ldots,n$, that is $\mathbb{B}_k(\mathrm{curl}_f,f)$ is uniquely determined by DoF

$$\int_f \boldsymbol{u}\cdot\boldsymbol{p}\,\mathrm{d}s,\quad \boldsymbol{p}\in\mathbb{P}^{\ell}_{k-\ell}(f)+(\Pi_f\boldsymbol{x})\mathbb{P}_{k-\ell}(f),$$

whose proof can be found in Lemma 4.7 in [**?**]. □

Given an $e\in\Delta_{\ell-1}(T_h)$, we choose a global $\{\boldsymbol{n}^e_f, e\subseteq f\in\Delta_\ell(T_h)\}$ as the basis for the normal plane $\mathscr{N}^e$ and a global basis $\{\boldsymbol{t}^e_i\}$ of $\mathscr{T}^e$. Define the global finite element space

$$V_h^{\mathrm{curl}}:=\{\boldsymbol{u}\in L^2(\Omega;\mathbb{R}^n):\boldsymbol{u}|_T\in\mathbb{P}_k(T;\mathbb{R}^n)\text{ for each }T\in\mathcal{T}_h,\\ \text{all the DoFs (7.8) are single-valued}\}.$$

Lemma 7.4. *We have* $V_h^{\mathrm{curl}}\subset H(\mathrm{curl},\Omega)$.

Proof. For an $F\in\Delta_{n-1}(T)$, DoFs (7.9) related to $\Pi_F\boldsymbol{u}$ are

$$\int_e(\Pi_F\boldsymbol{u}\cdot\boldsymbol{t}^e_i)\,p\,\mathrm{d}s,\quad i=1,\ldots,\ell-1,p\in\mathbb{P}_{k-\ell}(e),e\in\Delta_{\ell-1}(F),\ell=2,\ldots,n,$$
$$\int_e(\Pi_F\boldsymbol{u}\cdot\boldsymbol{n}^e_f)\,p\,\mathrm{d}s,\quad f\in\Delta_\ell(F),e\subseteq f,p\in\mathbb{P}_{k-\ell}(e),e\in\Delta_{\ell-1}(F),\ell=1,\ldots,n-1,$$

thanks to DoFs (7.9), which uniquely determine $\Pi_F\boldsymbol{u}$. □

7.5 A nodal basis for the second-kind Nédélec edge element

By Corollary 5.3, we obtain a nodal basis for the second-kind Nédélec edge element.

Theorem 7.4. *For each* $e\in\Delta_\ell(T),\ell=0,1,\ldots,n$, *we choose* $\{\boldsymbol{e}^e_i,i=0,\ldots,n-1\}=\{\boldsymbol{t}^e_i,i=1,\ldots,\ell,\boldsymbol{n}^e_f,f=e+i,i\in e^*\}$ *and its dual basis* $\{\hat{\boldsymbol{e}}^e_i,i=0,\ldots,n-1\}=\{\hat{\boldsymbol{t}}^e_i,i=1,\ldots,\ell,\boldsymbol{n}_{F_i}/(\boldsymbol{n}_{F_i}\cdot\boldsymbol{n}^e_{e+i}),i\in e^*\}$. *A basis function of the k-th order second kind Nédélec edge element space on* T *is:*

$$\{\phi_{\boldsymbol{\alpha}}(\boldsymbol{x})\hat{\boldsymbol{e}}^e_i,i=0,1,\ldots,n-1,\,\alpha\in\mathbb{T}^{\ell}_k(\mathring{e})\}_{e\in\Delta_\ell(T),\ell=0,1,\ldots,n},$$

with the DoFs at the interpolation points defined as:

$$\{u(\boldsymbol{x}_\alpha)\boldsymbol{e}^e_i,i=0,1,\ldots,n-1,\,\boldsymbol{x}_\alpha\in\mathcal{X}_{\mathring{e}}\}_{e\in\Delta_\ell(T),\ell=0,1,\ldots,n}.\tag{7.10}$$

Notice that the basis $\{\boldsymbol{t}^e_i\}$ depends only on e and $\boldsymbol{n}^e_f$ depends on e and f but independent T. By asking the corresponding DoFs single valued, we obtain the tangential continuity. We can define the edge finite element space as

$$V_h^{\mathrm{curl}}:=\{\boldsymbol{u}\in L^2(\Omega;\mathbb{R}^n):\boldsymbol{u}|_T\in\mathbb{P}_k(T;\mathbb{R}^n)\text{ for each }T\in\mathcal{T}_h,\text{ the DoFs }u(\boldsymbol{x}_\alpha)\boldsymbol{t}^e_i,u(\boldsymbol{x}_\alpha)\boldsymbol{n}^e_f\\ \text{in (7.10) are single-valued across all }e\text{ and }f\text{ with }\dim e,\dim f\le n-1\}.$$

7.5.1 2D basis on triangular meshes

Let T be a triangle, for lattice point $\boldsymbol{x}$ located in different sub-simplices, we shall choose different frame $\{\boldsymbol{e}_{\boldsymbol{x}}^0, \boldsymbol{e}_{\boldsymbol{x}}^1\}$ at $\boldsymbol{x}$ and its dual frame $\{\hat{\boldsymbol{e}}_{\boldsymbol{x}}^0, \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1\}$ as follows:

1. If $\boldsymbol{x} \in \Delta_0(T)$, assume the two adjacent edges are e_0 and e_1, then

$$\boldsymbol{e}_{\boldsymbol{x}}^0 = \boldsymbol{t}_{e_0}, \quad \boldsymbol{e}_{\boldsymbol{x}}^1 = \boldsymbol{t}_{e_1}, \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^0 = \frac{\boldsymbol{n}_{e_1}}{\boldsymbol{n}_{e_1} \cdot \boldsymbol{t}_{e_0}}, \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1 = \frac{\boldsymbol{n}_{e_0}}{\boldsymbol{n}_{e_0} \cdot \boldsymbol{t}_{e_1}}.$$

2. If $\boldsymbol{x} \in \mathcal{X}_{\mathring{e}}, e \in \Delta_1(T)$, then

$$\boldsymbol{e}_{\boldsymbol{x}}^0 = \boldsymbol{t}_e, \quad \boldsymbol{e}_{\boldsymbol{x}}^1 = \boldsymbol{n}_e, \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^0 = \boldsymbol{t}_e, \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1 = \boldsymbol{n}_e.$$

3. If $\boldsymbol{x} \in \mathcal{X}_{\mathring{T}}$, then

$$\boldsymbol{e}_{\boldsymbol{x}}^0 = (1,0), \quad \boldsymbol{e}_{\boldsymbol{x}}^1 = (0,1), \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^0 = (1,0), \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1 = (0,1).$$

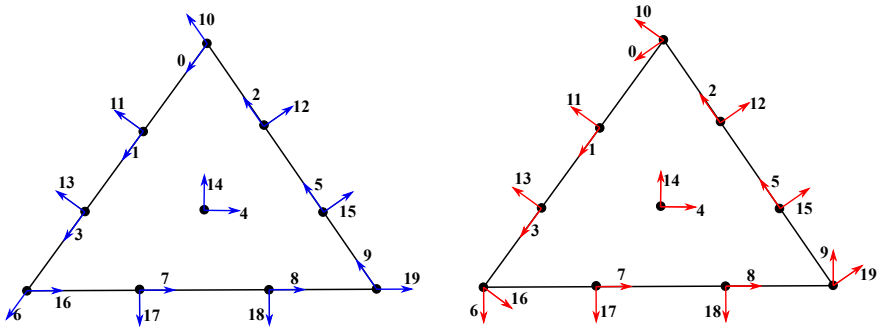


Figure 8: The left figure shows $\{\boldsymbol{e}_0, \boldsymbol{e}_1\}$ at each interpolation point, the right figure shows $\{\hat{\boldsymbol{e}}_0, \hat{\boldsymbol{e}}_1\}$ at each interpolation point.

7.5.2 3D basis on tetrahedron meshes

Let T be a tetrahedron, for any $\boldsymbol{x} \in \mathcal{X}_T$, define a frame $\{\boldsymbol{e}_{\boldsymbol{x}}^0, \boldsymbol{e}_{\boldsymbol{x}}^1, \boldsymbol{e}_{\boldsymbol{x}}^2\}$ at $\boldsymbol{x}$ and its dual frame $\{\hat{\boldsymbol{e}}_{\boldsymbol{x}}^0, \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1, \hat{\boldsymbol{e}}_{\boldsymbol{x}}^2\}$ as follows:

1. If $\boldsymbol{x} \in \Delta_0(T)$ and adjacent edges of $\boldsymbol{x}$ are e_0, e_1, e_2, adjacent faces of $\boldsymbol{x}$ are f_0, f_1, f_2, then

$$\boldsymbol{e}_{\boldsymbol{x}}^0 = \boldsymbol{t}_{e_0}, \quad \boldsymbol{e}_{\boldsymbol{x}}^1 = \boldsymbol{t}_{e_1}, \quad \boldsymbol{e}_{\boldsymbol{x}}^2 = \boldsymbol{t}_{e_2},$$

$$\hat{\boldsymbol{e}}_{\boldsymbol{x}}^0 = \frac{\boldsymbol{n}_{f_0}}{\boldsymbol{n}_{f_0} \cdot \boldsymbol{t}_{e_0}}, \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1 = \frac{\boldsymbol{n}_{f_1}}{\boldsymbol{n}_{f_1} \cdot \boldsymbol{t}_{e_1}}, \quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^2 = \frac{\boldsymbol{n}_{f_2}}{\boldsymbol{n}_{f_2} \cdot \boldsymbol{t}_{e_2}}.$$

2. If $\boldsymbol{x}\in\mathcal{X}_{\mathring{e}}, e\in\Delta_1(T)$ and adjacent faces are f_0, f_1 then

$$\boldsymbol{e}_{\boldsymbol{x}}^0=\boldsymbol{t}_e,\quad \boldsymbol{e}_{\boldsymbol{x}}^1=\boldsymbol{n}_{f_0}\times\boldsymbol{t}_e,\quad \boldsymbol{e}_{\boldsymbol{x}}^2=\boldsymbol{n}_{f_1}\times\boldsymbol{t}_e,$$

$$\hat{\boldsymbol{e}}_{\boldsymbol{x}}^0=\boldsymbol{t}_e,\quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1=\frac{\boldsymbol{n}_{f_1}}{\boldsymbol{n}_{f_1}\cdot(\boldsymbol{n}_{f_0}\times\boldsymbol{t}_e)},\quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^2=\frac{\boldsymbol{n}_{f_0}}{\boldsymbol{n}_{f_0}\cdot(\boldsymbol{n}_{f_1}\times\boldsymbol{t}_e)}.$$

3. If $\boldsymbol{x}\in\mathcal{X}_{\mathring{f}}, f\in\Delta_2(T)$, the first edge of f is e, then

$$\boldsymbol{e}_{\boldsymbol{x}}^0=\boldsymbol{t}_e,\quad \boldsymbol{e}_{\boldsymbol{x}}^1=\boldsymbol{t}_e\times\boldsymbol{n}_f,\quad \boldsymbol{e}_{\boldsymbol{x}}^2=\boldsymbol{n}_f,$$

$$\hat{\boldsymbol{e}}_{\boldsymbol{x}}^0=\boldsymbol{t}_e,\quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1=\boldsymbol{t}_e\times\boldsymbol{n}_f,\quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^2=\boldsymbol{n}_f.$$

4. If $\boldsymbol{x}\in\mathcal{X}_{\mathring{T}}$, then

$$\boldsymbol{e}_{\boldsymbol{x}}^0=(1,0,0),\quad \boldsymbol{e}_{\boldsymbol{x}}^1=(0,1,0),\quad \boldsymbol{e}_{\boldsymbol{x}}^2=(0,0,1),$$
$$\hat{\boldsymbol{e}}_{\boldsymbol{x}}^0=(1,0,0),\quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^1=(0,1,0),\quad \hat{\boldsymbol{e}}_{\boldsymbol{x}}^2=(0,0,1).$$

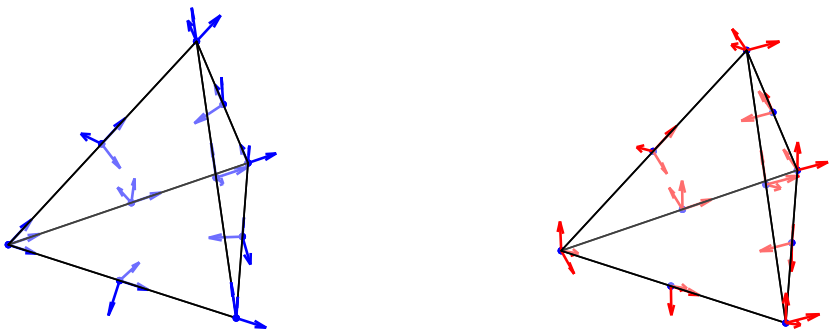


Figure 9: The left figure shows $\{\boldsymbol{e}_0,\boldsymbol{e}_1,\boldsymbol{e}_2\}$ at each interpolation point, the right figure shows $\{\hat{\boldsymbol{e}}_0,\hat{\boldsymbol{e}}_1,\hat{\boldsymbol{e}}_2\}$ at each interpolation point.

8 Indexing Management of Degrees of Freedom

In Section **??**, we have already discussed the dictionary indexing rule for interpolation points in each element. In this section, we will address the global indexing rules for Lagrange interpolation points, ensuring that interpolation points on a sub-simplex, shared by multiple elements, have a globally unique index. Given the one-to-one relationship between interpolation points and DoFs, this is equivalent to providing an indexing rule for global DoFs in the scalar Lagrange finite element space. Based on this, we will further discuss the indexing rules for DoFs in face and edge finite element spaces.

8.1 Lagrange finite element space

We begin by discussing the data structure of the tetrahedral mesh, denoted by $\mathcal{T}_h$. Let the numbers of nodes, edges, faces, and cells in $\mathcal{T}_h$ be represented as `NN`, `NE`, `NF`, and `NC`, respectively. We utilize two arrays to represent $\mathcal{T}_h$:

- `node` (shape: `(NN,3)`): `node[i,j]` represents the j-th component of the Cartesian coordinate of the i-th vertex.
- `cell` (shape: `(NC,4)`): `cell[i,j]` gives the global index of the j-th vertex of the i-th cell.

Given a tetrahedron denoted by `[0,1,2,3]`, we define its local edges and faces as:

- `SEdge = [(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)];`
- `SFace = [(1,2,3),(0,2,3),(0,1,3),(0,1,2)];`
- `OFace = [(1,2,3),(0,3,2),(0,1,3),(0,2,1)];`

Here, we introduced two types of local faces. The prefix `S` implies sorting, and `O` indicates outer normal direction. Both `SFace[i,:]` and `OFace[i,:]` represent the face opposite to the i-th vertex but with varied ordering. The normal direction as determined by the ordering of the three vertices of `OFace` matches the outer normal direction of the tetrahedron. This ensures that the outer normal direction of a boundary `face` points outward from the mesh. Meanwhile, `SFace` aids in determining the global index of the interpolation points on the face. For an in-depth discourse on indexing, ordering, and orientation, we direct readers to `sc3` in *i*FEM [**?**].

Leveraging the `unique` algorithm for arrays, we can derive the following arrays from `cell`, `SEdge`, and `OFace`:

- `edge` (shape: `(NE,2)`): `edge[i,j]` gives the global index of the j-th vertex of the i-th edge.
- `face` (shape: `(NF,3)`): `face[i,j]` provides the global index of the j-th vertex of the i-th face.
- `cell2edge` (shape: `(NC,6)`): `cell2edge[i,j]` indicates the global index of the j-th edge of the i-th cell.
- `cell2face` (shape: `(NC,4)`): `cell2face[i,j]` signifies the global index of the j-th face of the i-th cell.

Having constructed the edge and face arrays and linked cells to them, we next establish indexing rules for interpolation points on $\mathcal{T}_h$. Let k be the degree of the Lagrange finite element space. The number of interpolation points on each cell is

$$\texttt{ldof} = \dim \mathbb{P}_k(T) = \frac{(k+1)(k+2)(k+3)}{6},$$

and the total number of interpolation points on $\mathcal{T}_h$ is

$$\texttt{gdof} = \texttt{NN} + n_e^k \cdot \texttt{NE} + n_f^k \cdot \texttt{NF} + n_c^k \cdot \texttt{NC},$$

where

$$n_e^k = k-1, \quad n_f^k = \frac{(k-2)(k-1)}{2}, \quad n_c^k = \frac{(k-3)(k-2)(k-1)}{6},$$

are numbers of interpolation points inside edge, face, and cell, respectively. We need an index mapping from $[0{:}\texttt{ldof}-1]$ to $[0{:}\texttt{gdof}-1]$. See Fig. 10 for an illustration of the local index and the global index of interpolation points.

The tetrahedron's four vertices are ordered according to the right-hand rule, and the interpolation points adhere to the dictionary ordering map $R_3(\boldsymbol{\alpha})$. As Lagrange element is globally continuous, the indexing of interpolation points on the boundary ∂T should be global. Namely a unique index for points on vertices, edges, faces should be used and a mapping from the local index to the global index is needed.

We first set a global indexing rule for all interpolation points. We sort the index by vertices, edges, faces, and cells. For the interpolation points that coincide with the vertices, their global index are set as $0,1,\ldots,\texttt{NN}-1$. When $k>1$, for the interpolation points that inside edges, their global index are set as

$$\begin{matrix} 0 \\ 1 \\ \vdots \\ \texttt{NE}-1 \end{matrix} \begin{pmatrix} 0\cdot n_e^k & \cdots & 1\cdot n_e^k-1 \\ 1\cdot n_e^k & \cdots & 2\cdot n_e^k-1 \\ \vdots & & \vdots \\ (\texttt{NE}-1)\cdot n_e^k & \cdots & \texttt{NE}\cdot n_e^k-1 \end{pmatrix} + \texttt{NN}.$$

Here recall that $n_e^k = k-1$ is the number of interior interpolation points on an edge. When $k>2$, for the interpolation points that inside each face, their global index are set as

$$\begin{matrix} 0 \\ 1 \\ \vdots \\ \texttt{NF}-1 \end{matrix} \begin{pmatrix} 0\cdot n_f^k & \cdots & 1\cdot n_f^k-1 \\ 1\cdot n_f^k & \cdots & 2\cdot n_f^k-1 \\ \vdots & & \vdots \\ (\texttt{NF}-1)\cdot n_f^k & \cdots & \texttt{NF}\cdot n_f^k-1 \end{pmatrix} + \texttt{NN} + \texttt{NE}\cdot n_e^k,$$

where $n_f^k = (k-2)(k-1)/2$ is the number of interior interpolation points on f. When $k>3$, the global index of the interpolation points that inside each cell can be set in a similar way.

Then we use the two-dimensional array named `cell2ipoint` of shape `(NC,ldof)` for the index map. On the j-th interpolation point of the i-th cell, we aim to determine its unique global index and store it in `cell2ipoint[i,j]`.

For vertices and cell interpolation points, the mapping is straightforward by the global indexing rule. Indeed `cell` is the mapping of the local index of a vertex to its global index. However, complications arise when the interpolation point is located within an edge or face due to non-unique representations of an edge and a face.

We use the more complicated face interpolation points as a typical example to illustrate the situation. Consider, for instance, the scenario where the j-th interpolation point lies within the 0-th

local face F_0 of the i-th cell. Let $\alpha = $ `m = [m0,m1,m2,m3]` be its lattice point. Given that F_0 is opposite to vertex 0, we deduce that $\lambda_0|_{F_0} = 0$, which implies `m0` is 0. The remaining components of `m` are non-zero, ensuring that the point is interior to F_0.

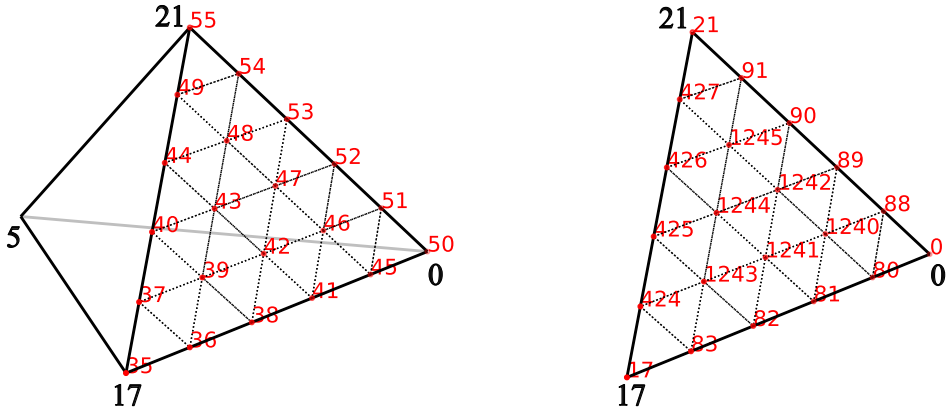


Figure 10: Local indexing (Left) and global indexing(Right) of DoFs for a face of tetrahedron, where the local vertex order is `[17, 0, 21]` and global vertex order is `[0, 17, 21]`. Due to the different ordering of vertices in local and global representation of the face, the ordering of the local indexing and the global indexing is different.

Two representations for the face with global index `cell2face[i,0]` are subsequently acquired:

- `LFace = cell[i,SFace[0,:]]` (local representation)
- `GFace = face[cell2face[i,0],:]` (global representation)

Although `LFace` and `GFace` comprise identical vertex numbers, their ordering differs. For example, `LFace = [5 6 10]` while `GFace =[10 6 5]`.

The array `m = [m1,m2,m3]` has a one-to-one correspondence with the vertices of `LFace`. To match this array with the vertices of `GFace`, a reordering based on argument sorting is performed:

```
i0 = argsort(argsort(GFace));
i1 = argsort(LFace);
i2 = i1[i0];
m = m[i2];
```

In the example of `LFace = [5 6 10]` while `GFace = [10 6 5]`, the input `m = [m1, m2,m3]` will be reordered to `m = [m3,m2,m1]`.

From the reordered `m = [m1,m2,m3]`, the local index ℓ of the j-th interpolation point on

the global face `f = cell2face[i,0]` can be deduced:

$$\begin{aligned}\ell &= \frac{(\mathtt{m2}-1+\mathtt{m3}-1)(\mathtt{m2}-1+\mathtt{m3}-1+1)}{2}+\mathtt{m3}-1\\ &= \frac{(\mathtt{m2}+\mathtt{m3}-2)(\mathtt{m2}+\mathtt{m3}-1)}{2}+\mathtt{m3}-1.\end{aligned}$$

It's worth noting that the index of interpolation points solely within the face needs consideration. Finally, the global index for the j-th interpolation point within the 0-th local face of the i-th cell is:

$$\mathtt{cell2ipoint[i\ j]} = \mathtt{NN}+n_e^k\cdot\mathtt{NE}+n_f^k\cdot\mathtt{f}+\ell.$$

Here, we provide a specific example. Consider a 5th-degree Lagrange finite element space on the `c`-th tetrahedron in a mesh depicted in Fig. 10. The vertices of this tetrahedron are `[5,17,0,21]`, and its 0th face is denoted as `f`, with vertices `[0,17,21]`. Assuming that

$$\mathtt{NN}+n_e^k\cdot\mathtt{NE}+n_f^k\cdot\mathtt{f}=1240.$$

For the 39-th local DoF on tetrahedron, its corresponding multi-index is `[0,3,1,1]` on cell and `[3,1,1]` on face. Since the local face is `[17,0,21]` and the global face is `[0,17,21]`, then `m=[1,3,1]` and ℓ =3 thus

```
cell2ipoint[c,39] = 1243.
```

Similar, for the 43-th local DoF on tetrahedron, `m = [1,2,2]` and ℓ=4 thus

```
cell2ipoint[c,43] = 1244.
```

Fig. 10 shows the correspondence between the local and the global indexing of DoFs for the cell.

In conclusion, we've elucidated the construction of global indexing for interpolation points inside cell faces. This method can be generalized for edges and, more broadly, for interior interpolation points of the low-dimensional sub-simplex of an n-dimensional simplex. Please note that for scalar Lagrange finite element spaces, the `cell2ipoint` array is the mapping array from local DoFs to global DoFs.

8.2 Face and edge finite element spaces

First, we want to emphasize that the management of DoFs is to manage the continuity of finite element space.

The BDM and second-kind Nédélec are vector finite element spaces, which define DoFs of vector type by defining a vector frame on each interpolation point. At the same time, they define their vector basis functions by combining the Lagrange basis function and the dual frame of the DoFs.

Alternatively, we can say each DoF in BDM or second-kind Nédélec sapce corresponds to a unique interpolation point p and a unique vector $\boldsymbol{e}$, and each basis funcion also corresponds to a unique Lagrange basis function which is defined on p and a vector $\hat{\boldsymbol{e}}$ which is the dual to $\boldsymbol{e}$.

The management of DoFs is essentially a counting problem. First of all, we need to set global and local indexing rules for all DoFs. We can globally divide the DoFs into shared and unshared among simplexes. The DoFs shared among simplexes can be further divided into on-edge and on-face according to the dimension of the sub-simplex where the DoFs locate. First count the shared DoFs on each edge according to the order of the edges, then count the shared DoFs on each face according to the order of the faces, and finally count the unshared DoFs in the cell. On each edge or face, the DoFs' order can follow the order of the interpolation points. Note that, for BDM and second-kind Nédélec space there are no DoFs shared on nodes. And for 3D BDM space there are no DoFs shared on edges. So the global numbering rule is similar with the Lagrange interpolation points.

According to the global indexing rule, we also can get a array named `dof2vector` with shape `(gdof, GD)`, where `gdof` is the number of global DoFs and `GD` represent geometry dimensions. And `dof2vector[i, :]` store the vector of the i-th DoF.

Next we need to set a local indexing rules and generate a array `cell2dof` with shape `(NC, ldof)`, where `ldof` is the number of local DoFs on each cell. Note that each DoF was determined by an interpolation point and a vector. And for each interpolation point, there is a frame (including `GD` vectors) on it. Given a DoF on i-th cell, denote the local index of its interpolation point as p, and the local index of its vector in the frame denote as q, then one can set a unique local index number j by p and q, for example $j=n\cdot q+p$, where n is the number of interpolation points in i-th cell. Furthermore, we can compute the `cell2dof[i,j]` by the global index `cell2ipoint[i, p]`, the sub-simplex that the interpolation point locate, and the global indexing rule.

Remark 8.1. Note that the local and global number rules mentioned above are not unique. Furthermore, with the array `cell2dof`, the implementation of these higher-order finite element methods mentioned in this paper is not fundamentally different from the conventional finite element in terms of matrix vector assembly and boundary condition handling. □

9 Numerical Examples

In this section, we numerically verify the 3-dimensional BDM elements basis and the second kind of Nédélec element basis using two test problems over the domain $\Omega=(0,1)^3$ partitioned into a structured tetrahedron mesh $\mathcal{T}_h$. All the algorithms and numerical examples are implemented based on the FEALPy package [**?**].

9.1 High Order Elements for Poisson Equation in the Mixed Formulation

Consider the Poisson problem:

$$\begin{cases} \boldsymbol{u}+\nabla p=0 & \text{in } \Omega, \\ \nabla\cdot\boldsymbol{u}=f & \text{in } \Omega, \\ p=g & \text{on } \partial\Omega. \end{cases}$$

The variational problem is : find $\boldsymbol{u}\in H(\mathrm{div},\Omega)$, $p\in L^2(\Omega)$, staify:

$$\int_\Omega \boldsymbol{u}\cdot\boldsymbol{v}\,\mathrm{d}\boldsymbol{x} - \int_\Omega p\nabla\cdot\boldsymbol{v}\,\mathrm{d}\boldsymbol{x} = -\int_{\partial\Omega} g(\boldsymbol{v}\cdot\boldsymbol{n})\,\mathrm{d}\boldsymbol{s}, \quad \boldsymbol{v}\in H(\mathrm{div},\Omega),$$
$$-\int_\Omega (\nabla\cdot\boldsymbol{u})q\,\mathrm{d}\boldsymbol{x} = -\int_\Omega fq\,\mathrm{d}\boldsymbol{x}, \qquad q\in L^2(\Omega).$$

Let $V_{h,k}^{\mathrm{div}}$ be the BDM space with degree k on $\mathcal{T}_h$ and piecewise polynomial space of degree $k-1$ on $\mathcal{T}_h$ by V_{k-1}. The corresponding finite element method is: find $\boldsymbol{u}_h\in V_{h,k}^{\mathrm{div}}, p_h\in V_{k-1}$, such that

$$\int_\Omega \boldsymbol{u}_h\cdot\boldsymbol{v}_h\,\mathrm{d}\boldsymbol{x} - \int_\Omega p\nabla\cdot\boldsymbol{v}_h\,\mathrm{d}\boldsymbol{x} = -\int_{\partial\Omega} g(\boldsymbol{v}_h\cdot\boldsymbol{n})\,\mathrm{d}\boldsymbol{s} \quad \boldsymbol{v}_h\in V_{h,k}^{\mathrm{div}}, \tag{9.1}$$
$$-\int_\Omega (\nabla\cdot\boldsymbol{u}_h)q_h\,\mathrm{d}\boldsymbol{x} = -\int_\Omega fq_h\,\mathrm{d}\boldsymbol{x}, \qquad q_h\in V_{k-1}. \tag{9.2}$$

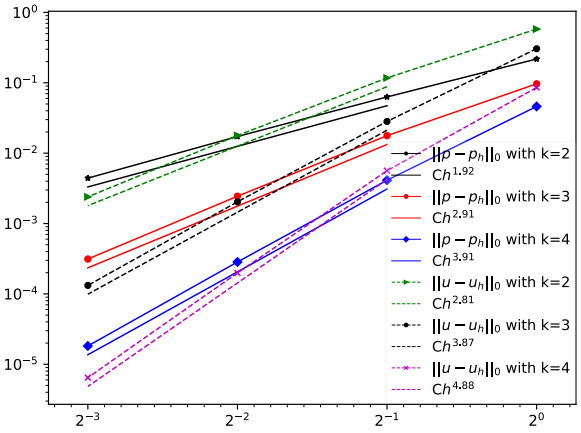


Figure 11: Errors $\|\boldsymbol{u}-\boldsymbol{u}_h\|_0$ and $\|p-p_h\|_0$ of finite element method (9.1) and (9.2) on uniformly refined mesh with $k=2,3,4$.

To test the convergence order of BDM space, we set

$$\boldsymbol{u} = \begin{pmatrix} -\pi\sin(\pi x)\cos(\pi y)\cos(\pi z) \\ -\pi\cos(\pi x)\sin(\pi y)\cos(\pi z) \\ -\pi\cos(\pi x)\cos(\pi y)\sin(\pi z) \end{pmatrix},$$

$$p = \cos(\pi x)\cos(\pi y)\cos(\pi z), \qquad f = 3\pi^2\cos(\pi x)\cos(\pi y)\cos(\pi z).$$

The numerical results are shown in Fig. 11 and it is clear that

$$\|\boldsymbol{u}-\boldsymbol{u}_h\|_0 \le Ch^{k+1}, \quad \|p-p_h\|_0 \le Ch^k.$$

9.2 High Order Elements for Maxwell Equations

Consider the time harmonic problem:

$$\begin{cases} \nabla\times(\mu^{-1}\nabla\times\boldsymbol{E})-\omega^2\tilde{\epsilon}\boldsymbol{E} &= \boldsymbol{J} \quad \text{in} \quad \Omega, \\ \boldsymbol{n}\times\boldsymbol{E} &= \boldsymbol{0} \quad \text{on} \quad \partial\Omega. \end{cases}$$

The variational problem is: find $\boldsymbol{E}\in H_0(\mathrm{curl},\Omega)$, satisfies:

$$\int_\Omega \mu^{-1}(\nabla\times\boldsymbol{E})\cdot(\nabla\times\boldsymbol{v})\,\mathrm{d}\boldsymbol{x}-\int_\Omega \omega^2\tilde{\epsilon}\boldsymbol{E}\cdot\boldsymbol{v}\,\mathrm{d}\boldsymbol{x}=\int_\Omega \boldsymbol{J}\cdot\boldsymbol{v}\,\mathrm{d}\boldsymbol{x} \quad \forall\,\boldsymbol{v}\in H_0(\mathrm{curl},\Omega).$$

Let $\mathring{V}_{h,k}^{\mathrm{curl}}=V_{h,k}^{\mathrm{curl}}\cap H_0(\mathrm{curl},\Omega)$, where $V_{h,k}^{\mathrm{curl}}$ is the edge element space defined in Theorem **??**. The corresponding finite element method is: find $\boldsymbol{E}_h\in\mathring{V}_{h,k}^{\mathrm{curl}}$ s.t.

$$\int_\Omega \mu^{-1}(\nabla\times\boldsymbol{E}_h)\cdot(\nabla\times\boldsymbol{v}_h)\,\mathrm{d}\boldsymbol{x}-\int_\Omega \omega^2\tilde{\epsilon}\boldsymbol{E}_h\cdot\boldsymbol{v}_h\,\mathrm{d}\boldsymbol{x}=\int_\Omega \boldsymbol{J}\cdot\boldsymbol{v}_h\,\mathrm{d}\boldsymbol{x}, \quad \boldsymbol{v}_h\in\mathring{V}_{h,k}^{\mathrm{curl}}. \tag{9.3}$$

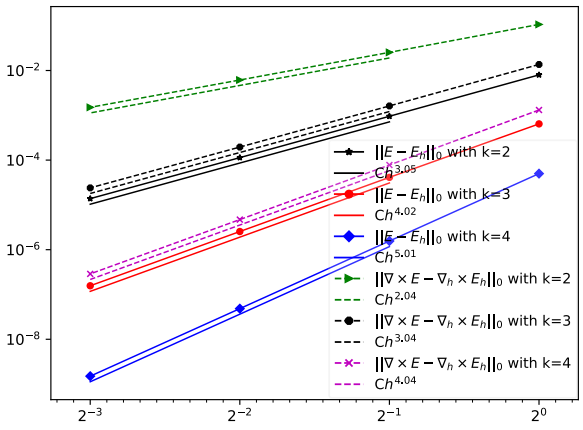


Figure 12: Errors $\|\boldsymbol{E}-\boldsymbol{E}_h\|_0$ and $\|\nabla\times(\boldsymbol{E}-\boldsymbol{E}_h)\|_0$ of finite element method (9.3) on uniformly refined mesh with $k=2,3,4$.

To test the convergence rate of second-kind Nédélec space, we choose

$$\omega=\tilde{\epsilon}=\mu=1, \qquad \boldsymbol{E}=(f,\sin(x)f,\sin(y)f),$$
$$f=(x^2-x)(y^2-y)(z^2-z), \qquad \boldsymbol{J}=\nabla\times\nabla\times\boldsymbol{E}-\boldsymbol{E}.$$

The numerical results are shown in Fig. 12 and it is clear that

$$\|\boldsymbol{E}-\boldsymbol{E}_h\|_0\le Ch^{k+1}, \quad \|\nabla\times(\boldsymbol{E}-\boldsymbol{E}_h)\|_0\le Ch^k.$$

Acknowledgments

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

Appendix A. Cantilever2dData1

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

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

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

        @cartesian
```

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