TU Delft

Delft University of Technology

Efficient large-scale 3D topology optimization with matrix-free MATLAB code

Wang, Junpeng; Aage, Niels; Wu, Jun; Sigmund, Ole; Westermann, Rüdiger

DOI
10.1007/s00158-025-04127-3

Publication date
2025

Document Version
Final published version

Published in
Structural and Multidisciplinary Optimization

Citation (APA)
Wang, J., Aage, N., Wu, J., Sigmund, O., & Westermann, R. (2025). Efficient large-scale 3D topology optimization with matrix-free MATLAB code. *Structural and Multidisciplinary Optimization*, *68*(9), Article 174. https://doi.org/10.1007/s00158-025-04127-3

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Green Open Access added to TU Delft Institutional Repository
as part of the Taverne amendment.

More information about this copyright law amendment
can be found at https://www.openaccess.nl.

Otherwise as indicated in the copyright section:
the publisher is the copyright holder of this work and the
author uses the Dutch legislation to make this work public.

Structural and Multidisciplinary Optimization (2025) 68:174
https://doi.org/10.1007/s00158-025-04127-3

EDUCATIONAL PAPER

Efficient large-scale 3D topology optimization with matrix-free MATLAB code

Junpeng Wang[1] · **Niels Aage**[2] · **Jun Wu**[3] · **Ole Sigmund**[2] · **Rüdiger Westermann**[1]

Received: 16 May 2025 / Revised: 16 July 2025 / Accepted: 19 August 2025 / Published online: 6 September 2025
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2025

Abstract
This paper presents an efficient MATLAB framework for large-scale density-based topology optimization and porous infill optimization in 3D. Besides showing comparable computational efficiency with existing MATLAB implementations at equivalent simulation scales, this framework supports significantly larger models with up to 128 million hexahedral simulation elements on a standard PC equipped with 64 GB RAM. Furthermore, it can handle arbitrary non-cuboid design domains and does not require powers-of-two differences in the elements' spatial resolutions. To achieve this, the technical contribution concentrates on solving the linear system of static finite element method (FEM). A tailored element-based matrix-free computing stencil is demonstrated to circumvent the vast memory consumption in large-scale FEM. Its computational efficiency is assured by fully leveraging the efficient matrix–vector operations and indexing functionalities in MATLAB. We further improve the computational efficiency and memory consumption of the MATLAB-implemented geometric multigrid method with a non-dyadic Galerkin coarsening and a diagonal relaxation scheme. All code is made publicly available at https://github.com/PSLer/TOP3D_XL.

Keywords Topology optimization · Porous infill · Multigrid method · Matrix-free

1 Replication of results

All important details have been presented in the paper and included in the associated code repository. Are results can be reproduced from it.

2 Introduction

Topology optimization (TO) is a general tool for obtaining optimized structures that fulfill a set of predefined goals and constraints. The simplest and most widely studied topology optimization problem is that of compliance minimization for linear elasticity. Here, the goal is to maximize the stiffness of a structure against external forces while subject to a constraint on the amount of available material. In its basic form, density-based TO operates on a first-order finite element discretization of the design domain to compute the compliance, requiring the solution of a sparse linear system of equations, $\boldsymbol{KU} = \boldsymbol{F}$, where the global stiffness matrix $\boldsymbol{K}$ is assembled from the element stiffness matrices under the assumption of an isotropic material law. The element sensitivities guide the material distribution iteratively by determining, in each iteration, which elements should gain or lose material according to its derivatives of the compliance and material consumption.

In previous works, MATLAB and high-performance implementations of TO have been presented, either designed for educational purposes, as reviewed in (Wang et al. 2021), or for the highest performance and model resolution on parallel computing architectures. For CPU-based parallel implementations, we refer to the works of (Borrvall and Petersson 2001; Evgrafov et al. 2008; Aage et al. 2015; Liu et al. 2018; Lin et al. 2022; Wu et al. 2024) and for GPU-accelerated approaches, see (Wadbro and Berggren 2009; Schmidt and Schulz 2011;

Responsible Editor: Josephine Voigt Carstensen.

✉ Junpeng Wang
junpeng.wang@tum.de

[1] Computer Graphics and Visualzation, Technical University of Munich, Munich, Germany

[2] Department of Civil and Mechanical Engineering, Technical University of Denmark, Kongens Lyngby, Denmark

[3] Department of Sustainable Design Engineering, Delft University of Technology, Delft, The Netherlands

Challis et al. 2014; Martínez-Frutos et al. 2015; Wu et al. 2016; Herrero-Pérez and Castejón 2021; Träff et al. 2023).

Educational codes are usually significantly behind performance-optimized solutions regarding model resolution, in particular due to the use of explicit representations of the large stiffness matrix $\boldsymbol{K}$. Performance-optimized codes achieve high model resolution via efficient CPU and GPU implementations of (*node-based*) matrix-free representations. In combination with multigrid solvers for computing the structural response of the layout (i.e., displacements), efficient systems for topology optimization have been developed. This, however, often requires fine-tuning and computer system-specific adaptations, making such codes more difficult to understand for non-expert users. To further improve the performance, additional model constraints are often baked into the codes, such as powers-of-two differences in the numbers of finite elements along different coordinate axes, or fully occupied cubical simulation domains (Mukherjee et al. 2021).

The motivation behind our work is to demonstrate that MATLAB-implemented educational codes can significantly narrow the gap with performance-optimized implementations — both in terms of model resolution and computational speed — while avoiding restrictive model constraints. Figure 1 illustrates this gap and shows the performance achievable in MATLAB by carefully revising computational blocks to leverage MATLAB's core features and common optimization techniques.

To this end, we introduce an *element-based* matrix-free format that is particularly well-suited for efficient matrix–vector operations, a key strength of MATLAB. This approach eliminates the need for assembling the global stiffness matrix ($\boldsymbol{K}$). Additionally, we highlight adaptations to the geometric multigrid solver used to solve the large linear system, to further reduce memory usage and enhance computational efficiency (Peetz and Elbanna 2021; Herrero-Pérez and Picó-Vicente 2023). We demonstrate the use of these optimizations for improving the performance of both TO and its local volume constraint variant—porous infill optimization (PIO) by Wu et al. (2018); Dou (2020). The PIO gives rise to a bone-mimicking infill structure and has been extended to work with multiple materials (Li et al. 2020), to design fiber-reinforced structures (Li et al. 2021), and to generate the porous structures with gradation (Schmidt et al. 2019). These intricate infill structures hinder the convergence of geometric multigrid solvers and serve to validate our framework.

The remainder of this paper is structured as follows. In Sect. 2, we briefly revisit the concepts of TO and PIO for completeness. Section 3 provides a detailed description of the adaptations to the geometric multigrid method. In Sect. 4, we introduce the element-based matrix-free format. The implementation details and validation results are presented in Sect. 5 and 6, respectively. Finally, we conclude the paper in Sect. 7.

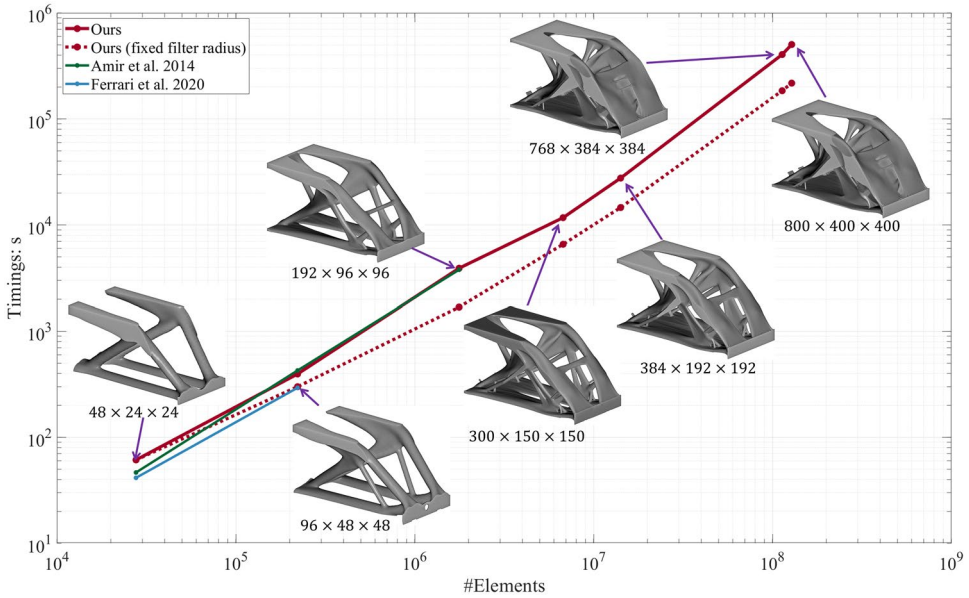


Fig. 1 Performance statistics of recent MATLAB-implemented educational TO codes (Ferrari and Sigmund 2020) (blue curve) and (Amir et al. 2014) (green curve) on a 6-core CPU desktop system with 64 GB RAM, tested up to the highest possible model resolution on this system. Both codes considerably limit the possible model resolution and require a 3D simulation grid with the powers-of-two ratio in spatial resolutions. With our proposed MATLAB code (red curves), significantly increased model resolutions can be simulated while maintaining the multigrid's linear time complexity in the number of simulation elements. The red dotted curve corresponds to the statistics of our method using a physically fixed filtering radius that is resolution-independent for different resolutions

3 Topology optimization

The optimization problem describing density-based TO for isotropic and linear elasticity is defined as

$$\min_{\rho} \quad c = \frac{1}{2}\boldsymbol{U}^T\boldsymbol{K}(\boldsymbol{\rho})\boldsymbol{U}, \tag{1}$$

$$\text{s.t.} \quad \boldsymbol{KU} = \boldsymbol{F}, \tag{2}$$

$$g(\boldsymbol{\rho}) \leq 0, \tag{3}$$

$$\rho_e \in [0.0,\ 1.0],\ \forall e. \tag{4}$$

Here, the objective function (c) is the structural compliance. $\boldsymbol{F}$ and $\boldsymbol{U}$ refer to the nodal force vector and the resulting displacement vector, respectively. Given that the density-based TO pipeline is considered here, $\boldsymbol{\rho}$ is the density vector that determines the material properties of the involved finite elements. It is processed by firstly a filtering/smoothing operation, optionally, followed by a projection, and afterwards, interpolation using *Solid Isotropic Material with Penalization* (SIMP), see Eq. 5.

$$\begin{aligned} \tilde{\boldsymbol{\rho}} &= \mathcal{F}(\boldsymbol{\rho}, r) \\ \bar{\boldsymbol{\rho}} &= \mathcal{P}(\tilde{\boldsymbol{\rho}}, \beta, \eta) \\ E_e(\bar{\rho}_e) &= E_{\min} + \bar{\rho}_e^{\gamma}(E_0 - E_{\min}) \end{aligned} \tag{5}$$

Here, $\mathcal{F}$ is a filter of radius r, which is generally constructed via either convolution or solving a PDE, $\mathcal{P}$ is a Heaviside projection with β as sharpness and η as the threshold. In the code, we use the PDE filter as described in Lazarov and Sigmund (2011) and Sect. 4). In SIMP, the penalization factor (γ) is typically set to 3. E_0 is the Young's modulus of the used isotropic material, and E_{min} is a minimum Young's modulus ($E_{min} = 1.0e^{-6}E_0$ in this paper), introduced to avoid the singularity of the global stiffness matrix, but still chosen sufficiently small such that void regions have no influence on the optimized design (Amir et al. 2014). $E_e(\bar{\rho}_e)$ is the interpolated Young's modulus of the element with physical density $\bar{\rho}_e$.

The constraint function (g) is specified according to the concrete design requirements. In conventional TO, the global volume fraction is employed to restrict the material consumption so that it does not exceed the available material budget, i.e.,

$$g(\boldsymbol{\rho}) = \frac{\sum \bar{\rho}_e}{N_e V_0} - 1 \leq 0 \tag{6}$$

Here, V_0 is the volume fraction of the permitted material consumption and N_e the number of design variables, i.e., the number of involved finite elements. Under such a constraint, TO usually gives rise to a mono-scale structural design.

In contrast to the global volume constraint, PIO imposes local volume constraints and produces a multi-scale structural design, i.e., both the topology and the infills are optimized concurrently (Wu et al. 2018). The mechanism is to let the material deposition around each finite element (e) in a certain region not exceed a given threshold (V_{e0})

$$\begin{aligned} g(\boldsymbol{\rho}) &= \frac{\left(\frac{1}{N_e}\sum_{i=1}^{N_e}\hat{\rho}_i^p\right)^{\frac{1}{p}}}{V_{e0}} - 1 \leq 0, \\ \hat{\rho}_e &= \frac{\sum_{j\in\mathbb{N}_e}\bar{\rho}_j}{|\mathbb{N}_e|},\ \mathbb{N}_e = \{j \mid \| x_j - x_e \|_2 \leq R_e\},\ \forall e \end{aligned} \tag{7}$$

Under this constraint, the resulting structural design becomes a porous infill structure with locally defined maximum volume fraction V_{e0}. The raw format of Eq. 7 corresponds to N_e constraint functions, which is cumbersome to solve. These constraints are aggregated by using the p-norm function into a single differentiable expression, as shown in Eq. 7, with $p = 16$ in our examples. Computing the local volume can be reformulated as a PDE filter, as shown in Träff et al. 2021.

The proposed framework supports both global and local volume constraints. This capability addresses functional requirements and also serves to further validate the robustness and applicability of the presented linear system solver, considering the significant differences in the heterogeneity of the resulting density distributions between these two approaches. This is important since the heterogeneity of media plays a critical role in the convergence behavior of the geometric multigrid method. Fish and Belsky (1995); Erlangga et al. (2006); Liu et al. (2020).

Solving: Solving the optimization problem (Eqs. 1, 2, 3 and 4) involves iteratively updating the design variables to minimize the objective function while fulfilling the constraint conditions. We adopt the Optimality Criteria (OC) to solve TO and the method of moving asymptotes (MMA) for PIO (Svanberg 1987; Träff et al. 2023; Wu et al. 2018). We chose the PDE filter (Lazarov and Sigmund 2011) for computational compatibility reasons because it shares the same mechanism as applying a local volume constraint (Träff et al. 2021).

In standard density-based TO or PIO, the structural design is explicitly represented by finite elements, leading to high computational overhead, especially the need for detailed structural designs. This stems from solving large-scale FEM linear systems at each step, posing challenges in convergence and memory consumption. A widely adopted solution is the geometric multigrid-preconditioned conjugate gradient (MGCG) solver, known for its efficiency (Amir et al. 2014), and combining a matrix-free computational

stencil for large-scale problems (Aage et al. 2015; Wu et al. 2016). However, its efficient MATLAB implementation remains underexplored. The following sections discuss a tailored geometric multigrid method and element-based matrix-free computing stencils.

4 Geometric multigrid method

Due to its improved convergence, MGCG is widely used to solve the FEM equation in TO and PIO, with additional performance optimizations such as using the solution vector from the previous step ($\boldsymbol{U}$) as the starting guess for the current optimization step. Algorithm 1 provides the code framework of MGCG.

Algorithm 1 MGCG

Input: $\boldsymbol{F}$, $\boldsymbol{U}$, M, ϵ_0
Output: $\boldsymbol{U}$
$\boldsymbol{r}_1 = \boldsymbol{F} - \boldsymbol{K}\boldsymbol{U}$
$\boldsymbol{z}_1 = \boldsymbol{p}_1 = \mathcal{G}(\boldsymbol{r}_1)$
$\boldsymbol{x}_1 = \boldsymbol{z}_1^{\mathrm{T}}\boldsymbol{r}_1$, $f = \|\mathbf{F}\|_2$
for $j = 1 : M$ **do**
  $\boldsymbol{v} = \boldsymbol{K}\boldsymbol{p}_j$ *%Matrix-free stiffness matrix-vector multiplication*
  $\lambda_j = \boldsymbol{x}_1/(\boldsymbol{p}_j^{\mathrm{T}}\boldsymbol{v})$
  $\boldsymbol{U} = \boldsymbol{U} + \lambda_j\boldsymbol{p}_j$
  $\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \lambda_j\boldsymbol{v}$
  if $\|\mathbf{r_{j+1}}\|_2 < f\epsilon_0$ **then** break **end if**
  $\boldsymbol{z}_{j+1} = \mathcal{G}(\boldsymbol{r}_{j+1})$ *%MG Preconditioning*
  $x_2 = \boldsymbol{z}_{j+1}^{\mathrm{T}}\boldsymbol{r}_{j+1}$
  $\boldsymbol{p}_{j+1} = \boldsymbol{z}_{j+1} + x_2/x_1\boldsymbol{p}_j$
  $x1 = x2$
end for

In Algorithm 1, $\mathcal{G}$ refers to the multigrid preconditioner, i.e., the V-cycle to compute the correction term (z) from the residual ($\boldsymbol{r}$). The parameters M and ϵ_0 respectively control the maximum number of iterations and the tolerance in the residual-based convergence criteria.

4.1 V-cycle

The geometric multigrid method uses a grid hierarchy in combination with a relaxation scheme to recursively restrict a fine-grid residual to the next coarser grid, where a correction term is computed and then interpolated to the fine grid. This process forms a so-called *V-cycle*, which takes the residual ($\boldsymbol{r}$) as input and returns the correction term ($\boldsymbol{z}$). Algorithm 2 outlines the major computational steps of a V-cycle.

Algorithm 2 Standard V-cycle

Input: $\boldsymbol{r}^{[l]}$, $l = 1$, $L \geqq 2$
Output: $\boldsymbol{z}^{[1]}$
%Restriction
for $l = 1 : 1 : L - 1$ **do**
  $\boldsymbol{z}^{[l]} = w \cdot \boldsymbol{r}^{[l]}/\boldsymbol{D}^{[l]}$ % Apply smoother
  $\boldsymbol{r}^{[l]} = \boldsymbol{r}^{[l]} - \boldsymbol{K}^{[l]}\boldsymbol{z}^{[l]}$% Compute residual
  $\boldsymbol{r}^{[l+1]} = (\boldsymbol{P}^{[l+1]})^{\mathrm{T}}\boldsymbol{r}^{[l]}$ % Restrict residual
end for
Solve: $\boldsymbol{K}^{[L]}\boldsymbol{z}^{[L]} = \boldsymbol{r}^{[L]}$ % Directly solve on coarsest level
%Interpolation
for $l = L - 1 : -1 : 1$ **do**
  $\boldsymbol{z}^{[l]} = \boldsymbol{z}^{[l]} + \boldsymbol{P}^{[l+1]}\boldsymbol{z}^{[l+1]}$ % Interpolate error & correct
  $\boldsymbol{z}^{[l]} = \boldsymbol{z}^{[l]} + w(\boldsymbol{r}^{[l]} - \boldsymbol{K}^{[l]}\boldsymbol{z}^{[l]})/\boldsymbol{D}^{[l]}$ % Apply smoother
end for

In Algorithm 2, $\boldsymbol{K}^{[l]}$, $\boldsymbol{r}^{[l]}$, and $\boldsymbol{z}^{[l]}$ are the system matrix, residual, and correction term on level l, respectively. Level numbers 1 and L correspond to the finest and coarsest level. $\left(\boldsymbol{P}^{[l+1]}\right)^{\mathrm{T}}$ is the prolongation operator to restrict the residual at level l to level $l+1$. Its transpose is used to interpolate the correction term at level l from the values at level $l+1$. A Jacobi smoother reduces high-frequency errors on the current fine grid, with the damping factor w set to 0.65. $\boldsymbol{D}^{[l]}$ is the vector of diagonal entries of $\boldsymbol{K}^{[l]}$.

Hierarchical representation From the global stiffness matrix at the finest resolution level ($\boldsymbol{K}^{[l=1]}$), the system matrices at the coarser levels are computed as

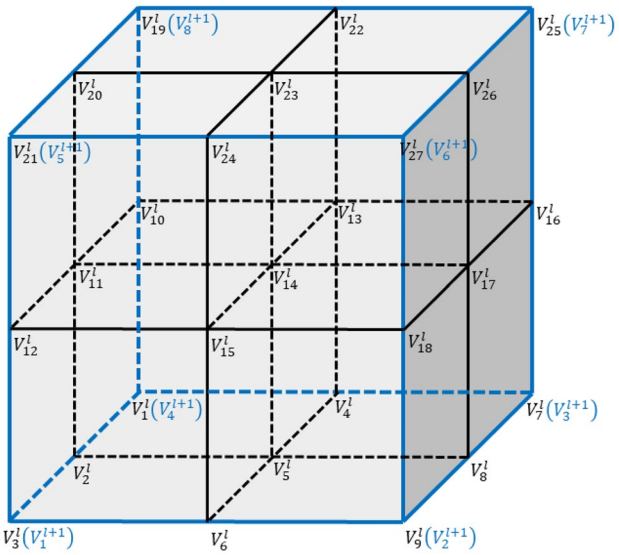


Fig. 2 Schematic diagram of a hexahedral element ($h_e^{[l+1]}$) in blue at level $l+1$ and its nested eight elements ($[h_e]^{[l]}$) in black at level l

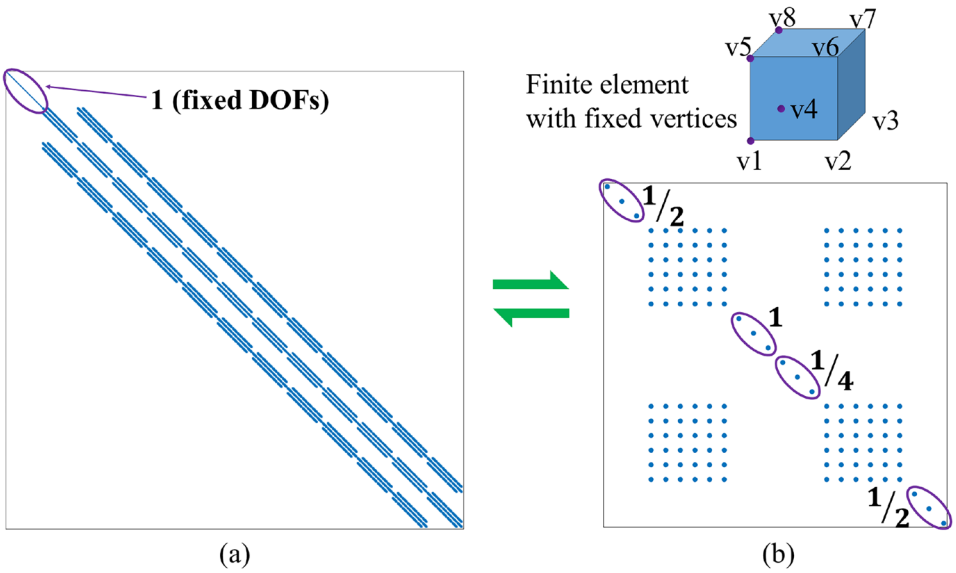


Fig. 3 Applying boundary conditions on the global stiffness matrix (**a**) and the element stiffness matrix involving fixed DOFs (**b**). In (**b**), the numerics $\frac{1}{x}$ indicates that the vertex, including this DOF, is shared by x finite elements

$$\boldsymbol{K}^{[l+1]} = (\boldsymbol{P}^{[l+1]})^{\mathrm{T}} \boldsymbol{K}^{[l]} \boldsymbol{P}^{[l+1]}, \quad l = 2 : L. \tag{8}$$

For a nested Cartesian grid hierarchy, where each element on level $l+1$ nests eight elements from level l (see Fig. 2), the prolongation operator is constructed using trilinear FEM shape functions. That is, the values at the fine-grid vertices are trilinearly interpolated from the values at the vertices of the next coarser grid.

For a dyadic hierarchy, i.e., the element size is doubled from level l to level $l+1$, the same prolongation operator is applied across all levels. We refer to this operator as the element prolongation operator $\boldsymbol{P}_e$.

Boundary conditions To compute the correction terms at the coarser resolutions, the boundary conditions specified at the vertices of the finest grid need to be transmitted to the system matrices at coarser resolutions. We adopt the scheme by Amir et al. (2014) to tackle this problem. Specifically, the entries corresponding to the fixed degrees of freedom (DOFs) in the global stiffness matrix ($\boldsymbol{K}$) are identified and split into two groups: entries on the diagonal of $\boldsymbol{K}$ are set to 1, and all other entries are set to 0 (see Fig. 3a), regardless of whether the elasticity tensors of the elements with the fixed DOFs are updated or not during the optimization. In this way, the dimensions of $\boldsymbol{K}$ are not changed after applying the boundary conditions, which ensures data alignment when transferring among different levels.

Once the boundary conditions are applied and transmitted appropriately, the correction terms at levels 2 to $L-1$ can be computed via the relaxation scheme. At level L, a linear system with significantly reduced dimension is directly solved via Cholesky decomposition.

4.2 MG preconditioner

Preconditioning aims to improve the convergence rate and stability of iterative solvers when solving large systems of linear equations. An effective preconditioner should substantially reduce the total number of iterations while maintaining low overall computational overhead. In our work, we consider two adaptations to the standard V-cycle to enhance its preconditioning effectiveness.

Non-dyadic V-cycle Let us first emphasize that our approach builds upon a matrix-free representation of the global stiffness matrices to reduce memory usage. Instead of explicit assembly, the matrices are stored as individual element stiffness matrices (see Sect. 4). This is especially advantageous for $\boldsymbol{K}^{[1]}$, where all element stiffness matrices are identical except for a scaling factor derived from the Young's modulus in the SIMP method (Eq. 5). Consequently, we store only a single unit Young's modulus element stiffness matrix and a per-element scaling vector.

However, this strategy does not extend to system matrices at levels 2 to L, where element stiffness matrices are unique and must be stored explicitly. This leads to either high memory consumption or computationally expensive on-the-fly assembly, creating a bottleneck for geometric multigrid methods on high-resolution grids. For instance, storing the element stiffness matrices for $\boldsymbol{K}^{[2]}$ on a 512^3 simulation grid requires about 72 GB of memory.

To circumvent this issue, we adopt the non-dyadic V-cycle from Wu et al. (2016), which skips level 2 in the grid hierarchy and V-cycle construction and directly computes the stencils at level 3 from level 1. This approach leverages the flexibility of the MG coarsening scheme, which is not limited to dyadic grid decimation but allows alternative decimation factors. Practically, this strategy integrates

seamlessly into the nested grid hierarchy of a Cartesian grid, requiring only a new element prolongation operator ($\hat{\boldsymbol{P}}_e$) between levels 1 and 3, analogous to $\boldsymbol{P}_e$ but involving more fine-resolution elements per coarse element.

With the non-dyadic V-cycle, trilinear interpolation is performed over a larger element, leading to reduced accuracy of the residual and interpolation vector at the finest level. This may slow V-cycle convergence due to an increased iteration count. However, it reduces per-iteration computational cost and memory usage by eliminating level 2 operations while also lowering V-cycle initialization overhead. In Sect. 6, we demonstrate that these improvements ultimately lead to faster overall solve times, even under strict error tolerances.

Diagonal relaxation To further reduce the total solve time, we use the non-dyadic V-cycle and omit all intermediate matrix–vector multiplications. Specifically, we replace Jacobi relaxation with a simple diagonal relaxation by omitting line 6 in Algorithm 2 and simplifying line 13 to

$$z^{[l]} = z^{[l]} + w\boldsymbol{r}^{[l]}/\boldsymbol{D}^{[l]} \tag{9}$$

While this approach is expected to increase the iteration count due to less effective damping of high- and low-frequency errors during restriction and interpolation, it significantly reduces the per-iteration computational cost. Unlike the non-dyadic V-cycle, which requires per iteration three computations of $\boldsymbol{K}^{[1]}\boldsymbol{u}$ and two computations of $\boldsymbol{K}^{[l]}\boldsymbol{u}$ for $l = 3 : L-1$, refer to line 7 in Algorithm 1, line 6 and 13 in Algorithm 2, the proposed method requires only a single computation of $\boldsymbol{K}^{[1]}\boldsymbol{u}$ (line 7 in Algorithm 1). Here $\boldsymbol{u}$ is generic, representing the displacement vector, as well as the residual and correction vector. If the iteration count increases by no more than a factor of two, a net reduction in total processing time is still achieved. Moreover, memory usage is further reduced, since $\boldsymbol{K}^{[l]}$, for $l = 2 : L-1$, no longer needs to be stored.

The convergence behavior of the V-cycle and its variants is demonstrated in Fig. 8 and 9 in Sect. 6. We refer to these V-cycle implementations as the *Standard V-cycle*, *Non-dyadic V-cycle*, and *Adapted Non-dyadic V-cycle*.

5 Element-based matrix-free format

High-resolution TO and PIO demand significant memory, primarily due to the assembly and storage of the global stiffness matrix ($\boldsymbol{K}$). As discussed in Sect. 3, $\boldsymbol{K}$ is used to perform matrix–vector multiplication ($\boldsymbol{Ku}$) and compute coarser-level stiffness matrices for the V-cycle. A matrix-free representation can be employed to conduct these operations without explicitly assembling or storing $\boldsymbol{K}$. The matrix-free representation benefits from the SIMP material model and the Cartesian simulation grid, where all elements share the same shape and size. Consequently, the same element strain matrix can be used and the element matrix remains constant up to a scaling factor, i.e.,

$$\boldsymbol{K}_e(\bar{\rho}_e) = E_e(\bar{\rho}_e)\int_{\Omega_e} \boldsymbol{B}_e^{\mathrm{T}}\boldsymbol{S}_0\boldsymbol{B}_e \mathrm{d}x = E_e(\bar{\rho}_e)\boldsymbol{K}_{e0}, \tag{10}$$

where $\boldsymbol{K}_{e0}$ is the identical element stiffness matrix corresponding to the unit Young's modulus, $\boldsymbol{S}_0$ is the corresponding identical elasticity tensor, and $\boldsymbol{B}_e$ is the element strain matrix. With $\boldsymbol{K}_{e0}$ and a vector ($[\boldsymbol{E}_e]$) storing the per-element Young's modulus, all element stiffness matrices at level 1 can be represented.

Matrix-free formats are node-based or element-based, depending on whether $\boldsymbol{Ku}$ is computed node-wise or element-wise. A node-wise computation scheme loops over all nodes, reads the stiffness matrices of all elements the current node belongs to via indexed memory access operations, and multiplies each matrix with the 3×1 vector corresponding to this node. Our experiments have shown that the indexed memory access operations slow down the performance significantly in MATLAB. Consequently, we switch to an element-based representation, which enables MATLAB to compute all matrix–vector products between the generic element matrix and the nodal displacement vectors efficiently. Therefore, the nodal displacement vectors are organized into columns of an element-wise displacement matrix, so that MATLAB's built-in capability for efficient matrix–vector operations can be exploited.

Conducting Ku The computationally most expensive step in linear elasticity FEM analysis is conducting $\boldsymbol{Ku}$, where $\boldsymbol{u}_{[N_d\times 1]}$ stores the displacement corrections at each grid node. $\boldsymbol{Ku}$ can be split into a series of matrix–vector products $\boldsymbol{K}_e\boldsymbol{u}_e$, where $\boldsymbol{u}_{e[24\times 1]}$ stores the nodal values of element e. The information is obtained by indexing with the corresponding grid information ($\boldsymbol{T}_{[N_e\times 8]}$) providing the element-vertex connection list (refer to Step 1 in Fig. 4). Let $\boldsymbol{Y} = \boldsymbol{Ku}$, $\boldsymbol{Y}_e = \boldsymbol{K}_e\boldsymbol{u}_e$, and $[\boldsymbol{Y}_e]_{[24\times N_e]}$ a matrix storing the $\boldsymbol{Y}_e$ of all elements. Then, $\boldsymbol{Y}$ can be obtained by accumulating the entries of $[\boldsymbol{Y}_e]$ in the corresponding positions according to $\boldsymbol{T}$ (Step 4 in Fig. 4). The MATLAB code of this procedure is provided in Appendix A.

In practice, the computation of $\boldsymbol{K}_e\boldsymbol{u}_e$ is split into two steps to fully exploit the structure of Eq. 10. First, we let $\boldsymbol{K}_{e0}$ multiply $[\boldsymbol{u}_e]$ (Step 2 in Fig. 4). Here, the unique element stiffness matrices (Fig. 3b) incorporating fixed DOFs must be considered to enforce boundary conditions. This is achieved by replacing the corresponding columns of $[\boldsymbol{Y}_{e0}]$ with the precomputed products of these matrices and the corresponding columns in $[\boldsymbol{u}_e]$. Next, $[\boldsymbol{Y}_{e0}]$ is scaled column-wise by $[\boldsymbol{E}_e]$ (Step 3 in Fig. 4), the vector storing the per-element Young's modulus. Notably, the presumably more efficient

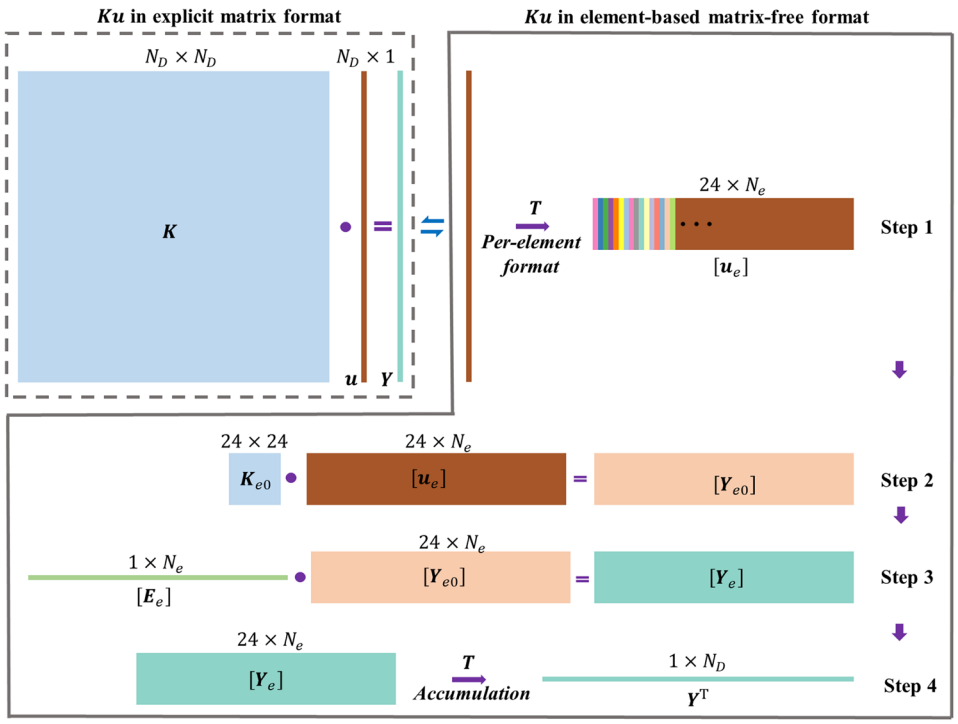


Fig. 4 Conducting ***Ku*** in the element-based matrix-free format. The explicit format is provided in the top left for reference. N_e and N_D are the number of elements and DOFs, respectively. $\boldsymbol{T}$ is the matrix storing the element-vertex connection list

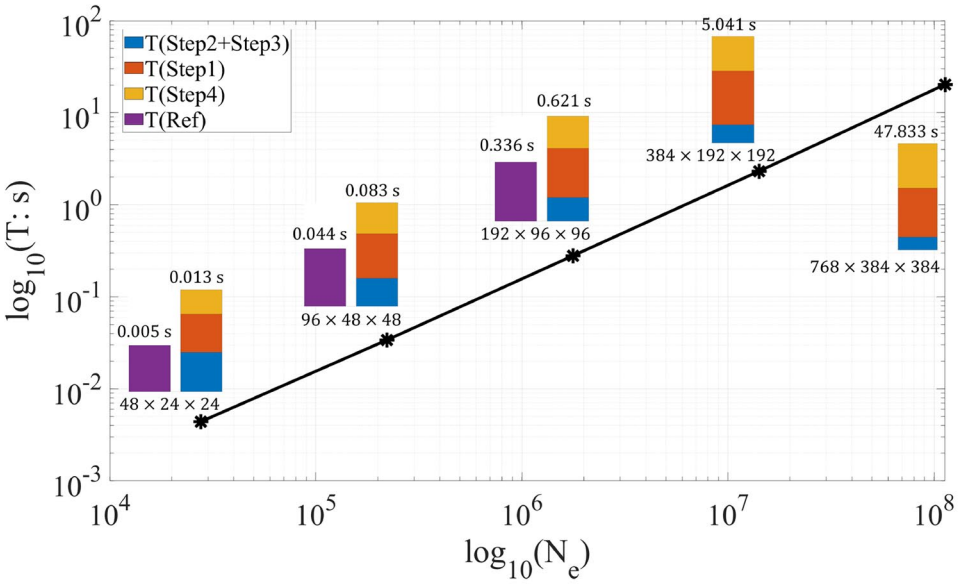


Fig. 5 Performance statistics of conducting ***Ku*** using MATLAB's explicit matrix format (purple bars) and the element-based matrix-free format. Steps 1 to 4 refer to Fig. 4

approach of setting fixed-DOF values to 0 after projecting the product vector to global indices was found to adversely affect convergence behavior.

Figure 5 compares the efficiency of the proposed element-based matrix-free format with MATLAB's built-in sparse matrix–vector multiplication. The test cases, detailed in Sect. 6, involve a cuboid design domain discretized into $48 \times 24 \times 24, 96 \times 48 \times 48, 192 \times 96 \times 96, 384 \times 192 \times 192$, and $768 \times 384 \times 384$ elements simulation elements. For each case, we compute ***Ku*** using both the proposed matrix-free format and MATLAB's explicit sparse format, comparing processing times. Due to memory constraints, MATLAB's explicit approach runs out of memory beyond the first three resolutions. The element-based matrix-free format is approximately twice as slow as MATLAB's built-in function, but it scales linearly with problem size. As resolution increases, Steps 1 and 4 in Fig. 4 consume a growing portion of the total runtime. This is because these steps involve indexing

operations, which MATLAB processes in a single thread. In contrast, Steps 2 and 3, involving matrix–vector multiplications, benefit from MATLAB's automatic parallelization.

Similarly, the PDE filter used in TO and PIO can also benefit from the proposed element-based matrix-free format. The filter is implemented by solving a Helmholtz-type differential equation, which is formulated using standard FEM as a linear system (Lazarov and Sigmund 2011). The system matrix is constructed by assembling the per-vertex kernels of the grid, which remain constant when a uniform filtering radius is employed. We solve the resulting linear system iteratively using the matrix–vector multiplication scheme illustrated in Fig. 4. The filtering operation involves only one DOF per node, compared to the three DOFs needed for the mechanical problem. Furthermore, due to the well-conditioned system matrix, the computational cost of the filtering operation becomes negligible in practice.

Coarse grid computation With the diagonal relaxation, for $l \in [2 : L-1]$ it becomes unnecessary to store $\boldsymbol{K}^{[l]}$ once its diagonal entries ($\boldsymbol{D}^{[l]}$) are extracted and $\boldsymbol{K}^{[l+1]}$ is computed. The computation of $\boldsymbol{K}^{[2]}$ depends on $\boldsymbol{K}^{[1]}$, but with the matrix-free format, $\boldsymbol{K}^{[1]}$ is no longer explicitly stored. Therefore, we propose to compute the element stiffness matrices $[\boldsymbol{K}_e^{[2]}]$ from $[\boldsymbol{K}_e^{[1]}]$ first, and then recursively obtain $[\boldsymbol{K}_e^{[l]}]$ for $l > 2$. Specifically, at each resolution, $[\boldsymbol{K}^{[l]}]$ can be represented by the corresponding $[\boldsymbol{K}_e^{[l]}]$, except for $[\boldsymbol{K}^{[L]}]$, which needs to be explicitly assembled from $[\boldsymbol{K}_e^{[L]}]$ for the direct Cholesky solver. In Appendix B, we provide the MATLAB code of extracting $\boldsymbol{D}^{[l]}$ from $\boldsymbol{K}^{[l]}$, $l = 1 : L-1$, stored in a matrix-free format.

Restriction and interpolation. To further reduce memory consumption, the *restriction* and *interpolation* operations in the V-cycle can also be implemented in the element-based matrix-free format, eliminating the need to explicitly assemble the global prolongation operators ($\boldsymbol{P}^{[l]}, : l = 2 : L$). Similarly to the way of conducting $\boldsymbol{Ku}$, these operations can be performed on a per-element basis as well. Then, the per-element residual ($\boldsymbol{r}_e^{[l]}$) and correction term ($\boldsymbol{z}_e^{[l]}$) vectors are projected onto their global indices to obtain $\boldsymbol{r}^{[l]}$ and $\boldsymbol{z}^{[l]}$, respectively. Here, the difference to the matrix-free computation of $\boldsymbol{Ku}$ is that the entity at a vertex of the mesh is concurrently determined by the elements sharing this vertex. In this case, an averaging process is used.

6 Implementation details

6.1 Non-cuboid domains

In practice, the design domain on which TO and PIO operate is not necessarily a fully-filled cuboid, meaning that a nested grid hierarchy cannot be computed directly. This is problematic for the element-based multigrid solver, where a boundary element at a certain level may not include 8 elements at the next finer level. In this case the per-element operations need to be adapted, which can disrupt data alignment and reduce processing efficiency. In the following, we propose a workaround specifically tailored to a hexahedral simulation domain.

Consider an arbitrary closed boundary surface defining the simulation domain (Fig. 6a) and its axis-aligned bounding volume. Computing the simulation grid starts by discretizing the bounding volume with a Cartesian grid. The cells inside and outside the boundary surface are marked as solid (1) and void (0), respectively. The hexahedral elements corresponding to solid voxels are extracted and used to form the FEM simulation model (see Fig. 6b).

Our first strategy ensures that for a given level L a nested grid hierarchy can be constructed, which is the case if at all levels l the resolutions along each of the three grid axis divided by 2^l are integers. This can easily be achieved by adding extra layers of void hexahedral cells until all resolutions up to level L satisfy the criterion.

Our second strategy addresses the case where a boundary element at level $l+1$ does not enclose eight solid elements

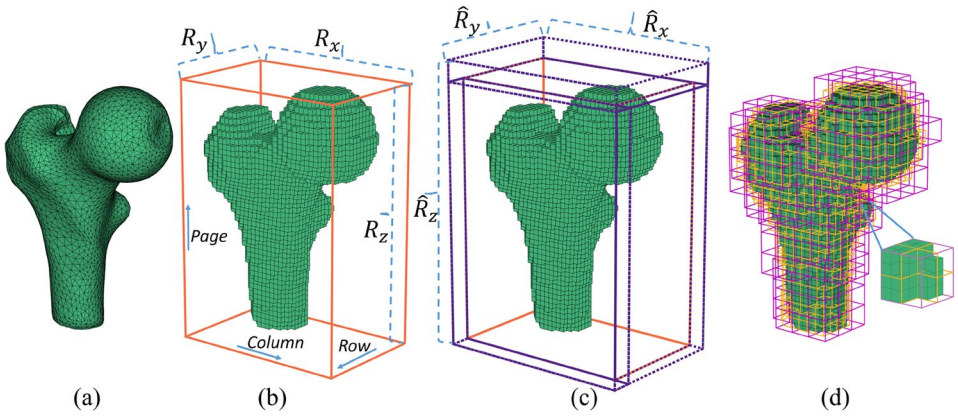


Fig. 6 Domain discretization and hierarchy construction. **a** The domain boundary given as closed triangle mesh. **b** Discretization of the bounding volume (orange) with a hexahedral grid. **c** The enlarged bounding volume (purple) so that an integer number of cells is obtained along either axis. **d** The mesh hierarchy (green), the mesh edges at the coarser levels are in orange and magenta, respectively

at level l. To resolve this, we adapt the mesh hierarchy by including void elements (within the boundary elements of the coarser level) in the simulation model. These additional elements are assigned zero stiffness, and their corresponding vertices are excluded from the simulation to preserve the original geometric model's properties.

The benefit of this strategy is twofold: First, it does not introduce additional DOFs since the solid voxels remain unchanged. Second, it ensures a consistent computational layout per element, simplifying implementation and maintaining uniform memory access patterns.

6.2 Memory access optimization

By analyzing the MGCG solver, we identify Step 1 as the most memory-intensive step (Fig. 4). In this step, the vertex-wise vector $\boldsymbol{u}$ is converted to an element-wise matrix ($[\boldsymbol{u}_e]$), requiring $24 \times 8 \times N_e$ bytes of memory. Here, 24 represents the DOFs for each hexahedral element, and 8 indicates the use of double-precision numbers. For instance, $[\boldsymbol{u}_e]$ consumes 24 GB of memory for a 512^3 element simulation grid.

Rather than processing a single large data block internally, we partition the elements into smaller chunks based on their global indices, ensuring each chunk does not exceed a predefined target size. This guarantees that all individual blocks fit within the available RAM, preventing frequent swap operations. Steps 1 to 4 are then executed separately for each chunk, using only the relevant data subset. Additionally, the memory allocated for $[\boldsymbol{u}_e]$ can be reused when processing subsequent chunks.

Splitting the computation into smaller tasks per chunk may slightly reduce efficiency, as it may not fully utilize the capacity of high-end hardware. However, it decreases the solver's dependence on problem size, particularly on mid-range hardware, thereby improving scalability in the model resolution. In practice, we find that setting the chunk size to 10^7 (approximately 1.8 GB of memory) gives a good balance between efficiency and model scalability on a machine with 64 GB of RAM.

7 Results

In the following, we demonstrate the capabilities of MATLAB TO and PIO framework from various perspectives. All numerical experiments are performed on a Windows desktop PC equipped with an Intel Xeon W-2235 CPU (6 cores, 3.8 GHz) and 64 GB of RAM. For the conjugate gradient solver, we set a relatively high convergence tolerance $\epsilon_0 = 10^{-3}$, which has been verified to give a stable solution when only the minimum compliance problem is considered. In addition, the maximum iteration (M) is set to be 600 for safety, though

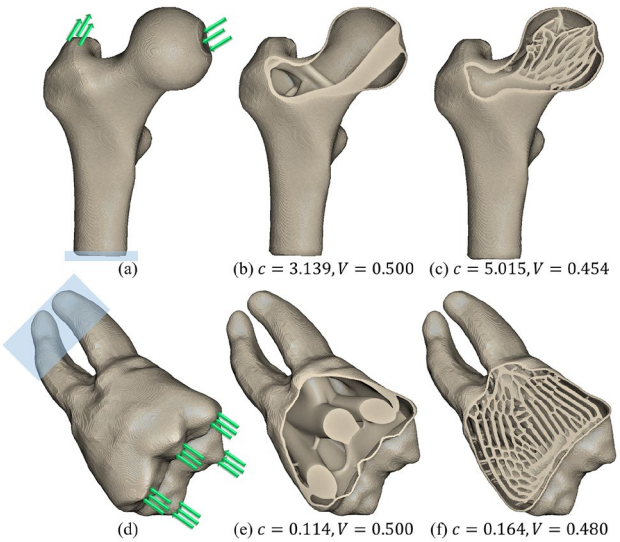


Fig. 7 Problem descriptions (**a**, **d**) and the corresponding TO (**b**, **e**) and PIO (**c**, **f**) results. Green arrows indicate the loading conditions, regions covered by the translucent patches in cyan are fixed. For both 'Femur' and 'Molar', the largest resolution is set to 400, corresponding to about 4.7 and 8.2 million finite elements, respectively

it's never reached apart from the extreme testing case when setting $\epsilon_0 = 10^{-12}$ for convergence behavior verification.

The move limits of OC and MMA, respectively, for TO and PIO, are selected as 0.2 and 0.1. All examples run in MATLAB R2023b. The Young's modulus and Poisson's ratio are set to 1.0 and 0.3, respectively. For all PIO examples, the outermost two layers of elements are defined as passive elements. The framework outputs two result files with standard formats for downstream operations like visual inspection and manufacturing. One is the structural design given as an isosurface in the STL format, and the other is the density volume in the NIFTI format. For the latter, one can inspect it via the WebGL-based volume renderer provided by Wang et al. (2025).

The experimental setup is organized as follows: First, the proposed V-cycle variant is verified using the models *'Femur'* and *'Molar'* (Fig. 7). Next, a benchmark example, *'Cantilever'* (Fig. 10), is used to compare computational efficiency and model scalability of the framework against existing MATLAB implementations. Finally, the *'GE Bracket'* (Fig. 12) is considered to highlight the framework's support for optimization problems under multiple loading conditions.

Comparison of V-cycle variants We conduct TO (Fig. 7b, e) and PIO (Fig. 7c, f) on *'Femur'* and *'Molar'* using the V-cycle implementations *Standard V-cycle*, *Non-dyadic V-cycle*, and *Adapted Non-dyadic V-cycle*. For TO, only the smoothing filter is used, whereas both the smoothing filter and projection are used for PIO, c.f. Equation 5.

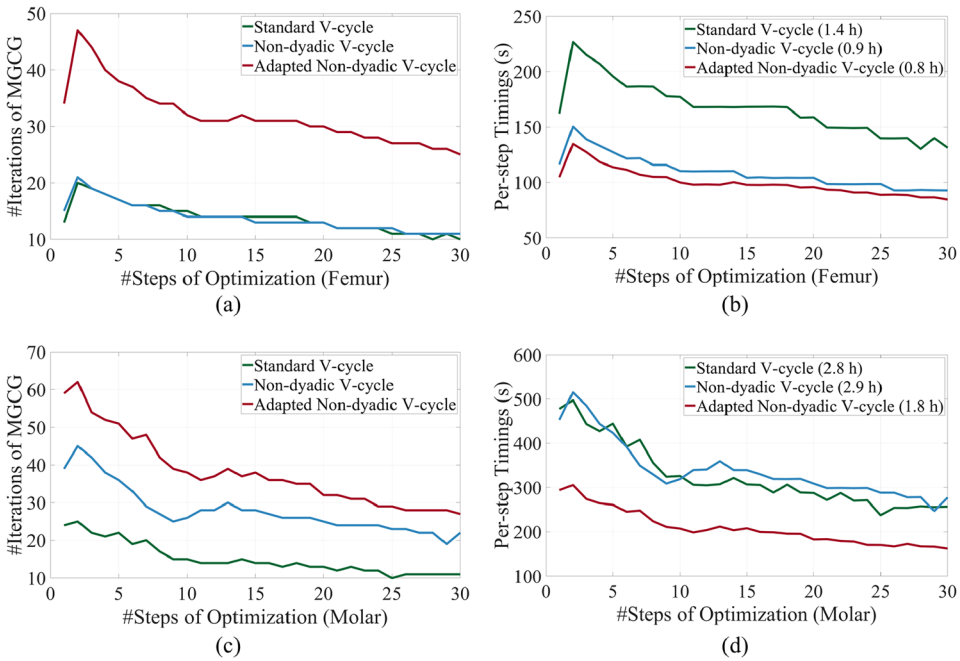


Fig. 8 TO statistics with different V-cycle implementations. **a** The number of iterations required by different V-cycle implementations to solve the linear system at each optimization step ('Femur'). **b** The per-step processing time of different V-cycle implementations and the processing time of the entire optimization process are provided in the legend ('Femur'). Similarly, **c** and **d** show the corresponding statistics for the 'Molar' example

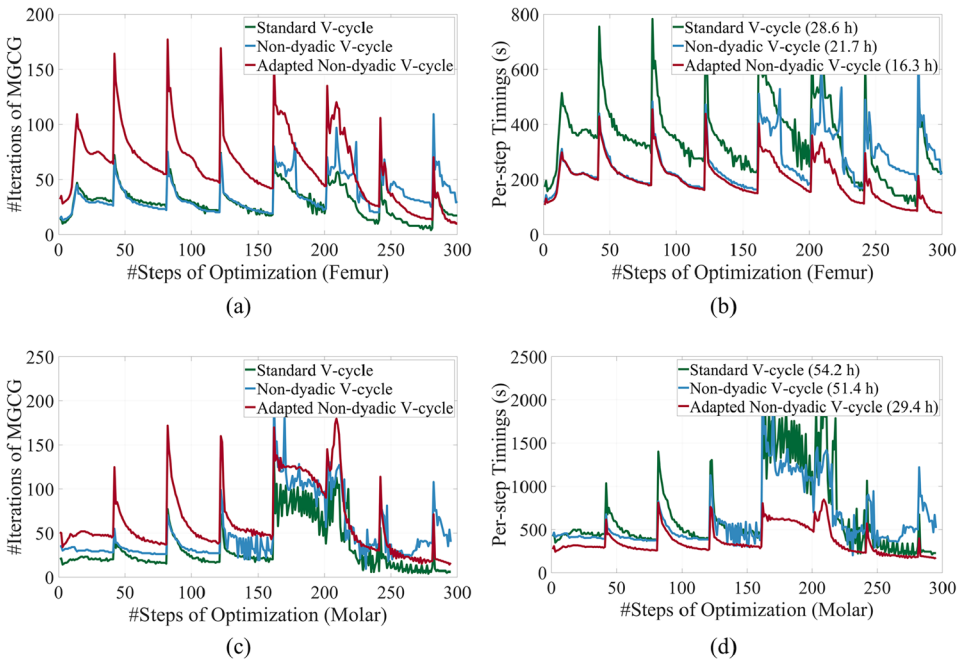


Fig. 9 Statistics of conducting PIO with different V-cycle implementations. The figure layout remains consistent with Fig. 8

Figure 8 shows the TO statistics, showing the MGCG iterations required to solve the FEM linear systems at each optimization step in 8a (*'Femur'*) and c (*'Molar'*), respectively, and the corresponding processing times of each optimization step in 8b (*'Femur'*) and d (*'Molar'*). The results demonstrate that for both cases the Standard V-cycle requires the fewest iterations but the longest processing time due to highest computational cost per MGCG iteration. The Adapted Non-dyadic V-cycle achieves the shortest processing times (both per step and overall) despite needing more iterations, benefiting from significantly reduced computational load per iteration. The Non-dyadic V-cycle shows intermediate convergence behavior, though not perfectly consistent between examples.

Following the layout of Figs. 8, 9 presents the PIO statistics. These results corroborate the earlier conclusion that the

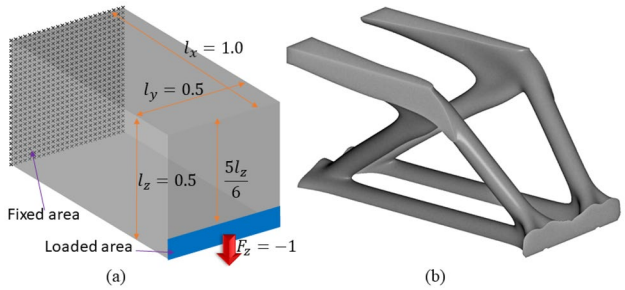


Fig. 10 **a** Problem description for *'Cantilever'*. **b** Identical TO results at different resolutions and physically fixed filtering radius

Adapted Non-dyadic V-cycle achieves the shortest processing time among the three implementations.

A comparison between Figs. 8 and 9 reveals additional insights: Despite identical linear system dimensions, all three V-cycle implementations require more MGCG iterations and consequently longer processing times per optimization step—for PIO than for TO. This stems from PIO's more heterogeneous density distribution (i.e., higher spatial frequencies), which geometrically complicates the linear system and degrades the geometric multigrid solver's convergence.

Most notably, between optimization steps 150–250, the required iterations per step increase significantly. This phenomenon typically results from the interplay between growing density-field heterogeneity and persistently large update magnitudes.

The *Adapted Non-dyadic V-cycle* demonstrates the highest computational efficiency for both TO and PIO, while simultaneously achieving minimal memory consumption, as element stiffness matrices from level 2 to $L-1$ no longer require storage. This verifies its superior capacity. Finally, we observe that the peaks in the PIO results originate from the continuation of the beta parameter in the smooth Heaviside projection filter.

Computational efficiency We evaluate the computational efficiency of our framework by comparing it to existing alternatives. To ensure maximal consistency, we restrict our comparison to MATLAB implementations from Amir et al. (2014) and Ferrari and Sigmund (2020), focusing specifically on their linear system solvers.

Amir et al. propose an MGCG solver using an explicit matrix format, while Ferrari and Sigmund employ a direct solver coupled with an external C++ code for efficient global stiffness matrix assembly. For fair large-resolution performance comparisons, we replace Ferrari's direct solver with Amir's MGCG solver, as recommended in Ferrari and Sigmund (2020). Thus, both implementations solve large-scale FEM systems via MGCG (without matrix-free approaches), differing only in matrix assembly: Ferrari and Sigmund (2020) uses a C++ program through MATLAB's MEX interface, achieving higher efficiency than MATLAB's native *sparse()* function.

We omit comparisons to the node-based matrix-free implementation in Träff et al. (2023), which is significantly slower due to loop-intensive indexing, as discussed in Sect. 4, and Liu and Tovar (2014)'s direct solver, which fails at higher resolutions due to memory constraints.

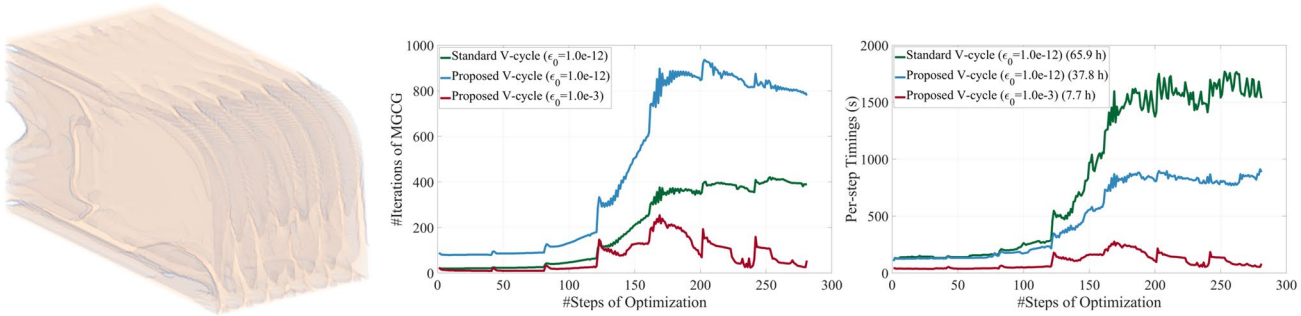
We use *'Cantilever'* as the benchmark for this experiment. The design domain is a cuboid with dimensions $1.0 \times 0.5 \times 0.5$ (Fig. 10a). The domain is discretized into $48 \times 24 \times 24$ ($R48$), $96 \times 48 \times 48$ ($R96$), $192 \times 96 \times 96$ ($R192$), $300 \times 150 \times 150$ ($R300$), $384 \times 192 \times 192$ ($R384$), $768 \times 384 \times 384$ ($R768$), and $800 \times 400 \times 400$ ($R800$) hexahedral elements, respectively. TO is performed using our MATLAB code and, for performance comparison, the codes from Amir et al. (2014); Amir (2015) and Ferrari and Sigmund (2020).

Except the resolution, we use the same settings in all experiments, including a material budget of $V_0 = 0.12$, 50 optimization steps, and a PDE filter radius (r) of $\sqrt{3}$ times the element size. Additionally, we investigate mesh-independence by imposing a physically fixed filter radius across different resolutions, see Fig. 10b for the corresponding TO result. Specifically, we set $r = \sqrt{3}$ for $R48$, $r = 2\sqrt{3}$ for $R96$, $r = 4\sqrt{3}$ for $R192$, $r = 6\sqrt{3}$ for $R300$, $r = 8\sqrt{3}$ for $R384$, $r = 16\sqrt{3}$ for $R768$, and $r = 17\sqrt{3}$ for $R800$, respectively.

Figure 1 presents the processing time statistics for each method across different resolutions. When using the code from Ferrari and Sigmund (2020), a memory issue occurs when assembling the global stiffness matrix at $R192$, whereas with the code from Amir et al. (2014); Amir (2015), this issue arises at $R300$, benefiting from MATLAB's automatic RAM-to-hard-drive swapping. Regardless of the memory limitations, the existing implementations cannot directly work with resolutions like $R300$ since they require powers-of-two differences in the simulation grid's spatial resolutions. Note that this is not a methodological limitation but rather a trade-off between computational efficiency and model variability. In contrast, our MATLAB code supports resolutions up to the maximum tested ($R800$) and comfortably accommodates $R300$.

In terms of processing times, the implementation from Ferrari and Sigmund (2020) achieves the highest efficiency at $R48$ and $R96$, owing to its OpenMP-accelerated C++ backend for matrix assembly. At $R48$, $R96$, and $R192$, our framework's computational efficiency is comparable to that of Amir et al. (2014). Moreover, our approach demonstrates near-linear scaling in computational complexity from $R48$ to $R800$.

While the implementations from Amir et al. (2014) and Ferrari and Sigmund (2020) could be optimized further (e.g., by exploiting the symmetry of the global stiffness matrix or



(a) $c^{\epsilon_0=10^{-12}} = 155.556, c^{\epsilon_0=10^{-3}} = 155.553$ (b) (c)

Fig. 11 **a** The PIO result (directly shown by the density values) under $\epsilon_0 = 10^{-12}$, which is visually identical with the one under $\epsilon_0 = 10^{-3}$. **b** and **c** The MGCG iterations and timings of each optimization step of the three involved test scenarios

using chunked assembly operations), such adjustments would not fully resolve the memory constraints of explicit matrix assembly. Thus, we leave these codes unmodified, except for replacing their filtering operation with the PDE filter for consistency. The corresponding compliance and measure of non-discreteness (MDN) values are provided in Table 1, where MDN is computed as $\frac{4}{N_e}\sum_e \bar{\rho}_e(1-\bar{\rho}_e)$.

With *'Cantilever'*, we further design an experiment to verify that even when setting $\epsilon_0 = 10^{-12}$, the proposed V-cycle still converges. Specifically, we conduct PIO with parameters $(R_e = 8,\ r = 2\sqrt{3},\ V_{e0} = 0.6)$ on the resolution $R192$, by separately using the *Standard V-cycle* with $\epsilon_0 = 10^{-12}$, and the *Adapted Non-dyadic V-cycle* with $\epsilon_0 = 10^{-12}$ and $\epsilon_0 = 10^{-3}$. The optimization process stops either after 300 optimization steps or when MDN is below 0.01. Given the low MGCG tolerance, we set $M = 1500$ for safety reasons.

The results in Fig. 11 demonstrate that the proposed *Adapted Non-dyadic V-cycle* still delivers high-precision solutions efficiently, confirming that the convergence behavior observed with *'Femur'* and *'Molar'* remains valid. Additionally, the PIO results for $\epsilon_0 = 10^{-3}$ show no significant difference compared to those for $\epsilon_0 = 10^{-12}$.

Table 1 Statistics on the number of elements, compliance (c), and MDN for *'Cantilever'*. The value of the filter radius (r) refers to the number of elements it considers

Models	#Elements	Ferrari et al. 2020 ($r = \sqrt{3}$)		Amir et al. 2014 ($r = \sqrt{3}$)		Ours ($r = \sqrt{3}$)		Ours		
		c	MDN	c	MDN	c	MDN	c	MDN	r
$R48$	27,648	963.467	0.153	963.464	0.153	965.226	0.156	965.226	0.156	$\sqrt{3}$
$R96$	221,184	695.020	0.089	695.020	0.089	679.250	0.091	886.209	0.143	$2\sqrt{3}$
$R192$	1,769,472	–	–	609.529	0.057	591.716	0.057	866.852	0.140	$4\sqrt{3}$
$R300$	6,750,000	–	–	–	–	566.538	0.043	843.392	0.135	$6\sqrt{3}$
$R384$	14,155,776	–	–	–	–	555.891	0.036	861.937	0.139	$8\sqrt{3}$
$R768$	113,246,208	–	-	–	-	545.192	0.020	860.676	0.139	$16\sqrt{3}$
$R800$	128,000,000	–	–	–	–	545.068	0.020	861.685	0.139	$17\sqrt{3}$

(a) (b) $c = 0.753, V = 0.450$ (c) $c = 0.936, V = 0.471$

Fig. 12 **a** Problem description for *'GE Bracket'*: the largest resolution is set to 512, corresponding to about 11 million finite elements. The 4 loading conditions are indicated by arrows with different colors, the fixed regions are in black. **b** and **c** show the results of TO and PIO, respectively

Multiple loading conditions In a final experiment, we use *'GE Bracket'* to demonstrate applying multiple loading conditions. In this case, the objective function (Eq. 1) needs to be slightly adapted to account for the contributions from different loads. This is achieved by replacing Eq. 1 with a weighted sum over the compliance values corresponding to different loading conditions. For PIO, the effecting radius (R_e) and local volume threshold (V_{e0}) are consistently set to 6 element sizes and 0.6 across the domain. For TO, the global volume threshold (V_0) is chosen to be 0.45. For both TO and PIO, the radius of the smoothing filtering is set to be 2 element sizes, and the outermost 5 layers of elements in the fixed area and the outermost 15 layers of elements in the loading area are set as passive elements. Note that the outermost 2 layers of elements are also set to be passive elements for the PIO experiment.

Figure 12a shows the result using 4 different loading conditions, meaning that 4 linear systems need to be solved at each optimization step. Considering that all loading conditions correspond to the same fixed regions, the V-cycle only needs to be initialized once at each optimization step. The TO result (Fig. 12b) is obtained after 50 optimization steps, while the PIO result (Fig. 12c) is obtained after 300 optimization steps, which take 7.6 and 144 h, respectively.

Regarding multiple loading conditions, we acknowledge that our current approach requires a separate solve pass for each individual load case. Consequently, compared to factorization-based solvers, which can efficiently solve multiple right-hand sides after a single factorization, our method is less efficient in this setting. However, factorization-based solvers have high computational complexity and can suffer from exponential growth of components—issues our approach is specifically designed to overcome. An interesting research question is whether the advantageous properties of both strategies can be combined.

8 Conclusions

This paper presents and analyzes an efficient MATLAB implementation of density-based topology optimization and porous infill optimization in 3D, with a focus on high-resolution simulation domains. To achieve this, we provide insights into efficiently solving large-scale linear elasticity problems using the finite element method (FEM) in MATLAB. We demonstrate that the widely used multigrid preconditioner for a conjugate gradient solver can be effectively combined with a matrix-free computing stencil. By optimizing the matrix-free format to fully leverage MATLAB's built-in computational routines and employing an adapted V-cycle implementation, our approach achieves a significant performance improvement over existing alternatives—enabling resolutions approximately two orders of magnitude higher. Furthermore, the proposed MATLAB framework offers high usability, seamlessly accommodating arbitrary domains, resolutions, and multiple loading conditions.

Although this framework demonstrates compelling performance advantages over existing alternatives, it has inherent limitations for further optimization. The primary bottleneck lies in MATLAB's single-threaded indexing operations that are frequently used in conducting the matrix–vector multiplication in the element-based matrix-free format. This becomes particularly restrictive for large-scale problems. This limitation also explains why the framework's performance still remains below state-of-the-art C/C++ implementations.

Potential solutions include leveraging MATLAB's GPU acceleration or MEX functionality. However, GPU-based approaches are constrained by the limited VRAM of mid-range graphics cards, making them unsuitable for high-resolution simulations. Meanwhile, MEX would require external C/C++ code, which contradicts this work's focus on a pure MATLAB implementation. In future work, we will monitor MATLAB's development of enhanced indexing capabilities to address this challenge.

A Appendix

Demonstration code for conducting ***Ku*** in the proposed element-based matrix-free format.

```
%%Conduct "Y = Ku"
%%Step 1
Y=zeros(numNodes,3);
uMat = zeros(size(eNodMat,1),24);
tmp=u(:,1); uMat(:,1:3:24)=tmp(eNodMat);
tmp=u(:,2); uMat(:,2:3:24)=tmp(eNodMat);
tmp=u(:,3); uMat(:,3:3:24)=tmp(eNodMat);
%%Apply for Boundary Conditions 1
eleWithFixedDOFs=find(mapUniqueKes_>0);
eleWithFixedDOFsLocal=mapUniqueKes_(
    eleWithFixedDOFs);
subDisVecUnique=uMat(eleWithFixedDOFs,:);
for kk=1:numel(eleWithFixedDOFs)
ss=eleWithFixedDOFsLocal(kk);
subDisVecUnique(kk,:)=subDisVecUnique(kk,:)
    *(reshape(uniqueKesFree_(:,ss),24,24)*
    Ee(eleWithFixedDOFs(kk))+reshape(
    uniqueKesFixed_(:,ss),24,24));
end
%%Step 2 & 3
uMat=uMat*Ke.*Ee(:);
%%Apply for Boundary Conditions 2
uMat(eleWithFixedDOFs,:)=subDisVecUnique;
%%Step 4
tmp=uMat(:,1:3:24);
Y(:,1)=Y(:,1)+accumarray(eNodMat(:),tmp(:)
    ,[numNodes,1]);
tmp=uMat(:,2:3:24);
Y(:,2)=Y(:,2)+accumarray(eNodMat(:),tmp(:)
    ,[numNodes,1]);
tmp=uMat(:,3:3:24);
Y(:,3)=Y(:,3)+accumarray(eNodMat(:),tmp(:)
    ,[numNodes,1]);
Y=Y'; Y=Y(:);
```

B Appendix

Demonstration code for extracting the diagonal entries ($\boldsymbol{D}^{[l]}$) from $\boldsymbol{K}^{[l]}$ with $l = 1 : L - 1$.

```
%% Extract diagonal entries (D) of
    stiffness matrix (K) at Level-l
D_l=zeros(numNodes_l,3);
eNodMatTmp=eNodMat_l'; eNodMatTmp=
    eNodMatTmp(:);
if 1==l
diagKe = diag(Ke0);
diagKe = diagKe(:).*E(:)'; %%E: Young's
    Modulus
eleWithFixedDOFs=find(mapUniqueKes_>0);
eleWithFixedDOFsLocal=mapUniqueKes_(
    eleWithFixedDOFs);
for kk=1:numel(eleWithFixedDOFs)
kKeFreeDOFs=reshape(uniqueKesFree_(:,
    eleWithFixedDOFsLocal(kk)),24,24);
kKeFixedDOFs=reshape(uniqueKesFixed_(:,
    eleWithFixedDOFsLocal(kk)),24,24);
diagKe(:,eleWithFixedDOFs(kk))=diag(
    kKeFreeDOFs)*E(eleWithFixedDOFs(kk))+
    diag(kKeFixedDOFs);
end
tmp=diagKe(1:3:end,:); tmp=tmp(:);
D_l(:,1)=D_l(:,1)+accumarray(eNodMatTmp,tmp
    ,[numNodes_l,1]);
tmp=diagKe(2:3:end,:);tmp=tmp(:);
D_l(:,2)=D_l(:,2)+accumarray(eNodMatTmp,tmp
    ,[numNodes_l,1]);
tmp=diagKe(3:3:end,:);tmp=tmp(:);
D_l(:,3)=D_l(:,3)+accumarray(eNodMatTmp,tmp
    ,[numNodes_l,1]);
else
%% 'Ks' stores element stiffness matrices
    on Level-l (L>1) in the layout 24 x 24
    x numElements_l
KsTmp=reshape(Ks,24*24,numElements_l);
diagKeBlock=KsTmp(1:25:(24*24),:);
tmp=diagKeBlock(1:3:end,:); tmp=tmp(:);
D_l(:,1)=D_l(:,1)+accumarray(eNodMatTmp,tmp
    ,[numNodes_l,1]);
tmp = diagKeBlock(2:3:end,:); tmp=tmp(:);
D_l(:,2)=D_l(:,2)+accumarray(eNodMatTmp,tmp
    ,[numNodes_l,1]);
tmp=diagKeBlock(3:3:end,:); tmp=tmp(:);
D_l(:,3)=D_l(:,3)+accumarray(eNodMatTmp,tmp
    ,[numNodes_l,1]);
end
D_l=reshape(D_l',numDOFs_l,1);
```

Acknowledgements All important details have been revealed. The complete code (*TOP3D_XL*) can be found through the link in the abstract.

Author Contributions Junpeng Wang: Conceptualization of this work, Software, Methodology, Writing - Review & Editing. Niels Aage: Conceptualization of this work, Methodology, Writing - Review & Editing. Jun Wu: Conceptualization of this work, Methodology. Ole Sigmund: Conceptualization of this work, Writing - Review & Editing. Rüdiger Westermann: Conceptualization of this work, Methodology, Writing - Review & Editing, Supervision, Funding acquisition.

Funding This work is supported by the German Research Foundation (DFG) under grant number WE 2754/10-1.

N. Aage and O. Sigmund acknowledge the financial support from the Villum Foundation through the Villum Investigator Project Amstrad (VIL54487).

Data Availability All involved data can be found in the code repository that is publicly available.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Aage N, Andreassen E, Lazarov BS (2015) Topology optimization using PETSc: an easy-to-use, fully parallel, open source topology optimization framework. Struct Multidiscip Optim 51:565–572. https://doi.org/10.1007/S00158-014-1157-0

Amir O (2015) Revisiting approximate reanalysis in topology optimization: on the advantages of recycled preconditioning in a minimum weight procedure. Struct Multidiscip Optim 51:41–57. https://doi.org/10.1007/s00158-014-1098-7

Amir O, Aage N, Lazarov BS (2014) On multigrid-cg for efficient topology optimization. Struct Multidiscip Optim 49:815–829. https://doi.org/10.1007/s00158-013-1015-5

Borrvall T, Petersson J (2001) Large-scale topology optimization in 3d using parallel computing. Comput Methods Appl Mech Eng 190(46–47):6201–6229. https://doi.org/10.1016/S0045-7825(01)00216-X

Challis VJ, Roberts AP, Grotowski JF (2014) High resolution topology optimization using graphics processing units (gpus). Struct Multidiscip Optim 49(2):315–325. https://doi.org/10.1007/s00158-013-0980-z

Dou S (2020) A projection approach for topology optimization of porous structures through implicit local volume control. Struct Multidiscip Optim 62(2):835–850. https://doi.org/10.1007/s00158-020-02539-x

Erlangga YA, Oosterlee CW, Vuik C (2006) A novel multigrid based preconditioner for heterogeneous helmholtz problems. SIAM J Sci Comput 27(4):1471–1492. https://doi.org/10.1137/040615195

Evgrafov A, Rupp CJ, Maute K, Dunn ML (2008) Large-scale parallel topology optimization using a dual-primal substructuring solver. Struct Multidiscip Optim 36(4):329–345. https://doi.org/10.1007/s00158-007-0190-7

Ferrari F, Sigmund O (2020) A new generation 99 line matlab code for compliance topology optimization and its extension to 3d. Struct Multidiscip Optim 62:2211–2228. https://doi.org/10.1007/s00158-020-02629-w

Fish J, Belsky V (1995) Multigrid method for periodic heterogeneous media part 1: Convergence studies for one-dimensional case. Comput Methods Appl Mech Eng 126(1–2):1–16. https://doi.org/10.1016/0045-7825(95)00811-E

Herrero-Pérez D, Castejón PJM (2021) Multi-gpu acceleration of large-scale density-based topology optimization. Adv Eng Softw 157:103006. https://doi.org/10.1016/j.advengsoft.2021.103006

Herrero-Pérez D, Picó-Vicente SG (2023) A parallel geometric multigrid method for adaptive topology optimization. Struct Multidiscip Optim 66(10):225. https://doi.org/10.1007/s00158-023-03675-w

Lazarov BS, Sigmund O (2011) Filters in topology optimization based on helmholtz-type differential equations. Int J Numer Meth Eng 86(6):765–781. https://doi.org/10.1002/nme.3072

Li H, Gao L, Li H, Li X, Tong H (2021) Full-scale topology optimization for fiber-reinforced structures with continuous fiber paths. Comput Methods Appl Mech Eng 377:113668. https://doi.org/10.1016/j.cma.2021.113668

Li H, Gao L, Li H, Tong H (2020) Spatial-varying multi-phase infill design using density-based topology optimization. Comput Methods Appl Mech Eng 372:113354. https://doi.org/10.1016/j.cma.2020.113354

Lin H, Liu H, Wei P (2022) A parallel parameterized level set topology optimization framework for large-scale structures with unstructured meshes. Comput Methods Appl Mech Eng 397:115112. https://doi.org/10.1016/j.cma.2022.115112

Liu H, Hu Y, Zhu B, Matusik W, Sifakis E (2018) Narrow-band topology optimization on a sparsely populated grid. ACM Transactions on Graphics (TOG) 37(6):1–14. https://doi.org/10.1145/3272127.3275012

Liu X, Réthoré J, Baietto MC, Sainsot P, Lubrecht AA (2020) An efficient finite element based multigrid method for simulations of the mechanical behavior of heterogeneous materials using ct images. Comput Mech 66:1427–1441. https://doi.org/10.1007/s00466-020-01909-y

Liu K, Tovar A (2014) An efficient 3d topology optimization code written in matlab. Struct Multidiscip Optim 50(6):1175–1196. https://doi.org/10.1007/s00158-014-1107-x

Martínez-Frutos J, Martínez-Castejón PJ, Herrero-Pérez D (2015) Fine-grained gpu implementation of assembly-free iterative solver for finite element problems. Computers & Structures 157:9–18. https://doi.org/10.1016/j.compstruc.2015.05.010

Mukherjee S, Lu D, Raghavan B, Breitkopf P, Dutta S, Xiao M, Zhang W (2021) Accelerating large-scale topology optimization: state-of-the-art and challenges. Arch Comput Methods Eng 28(7):4549–4571. https://doi.org/10.1007/s11831-021-09544-3

Peetz D, Elbanna A (2021) On the use of multigrid preconditioners for topology optimization. Struct Multidiscip Optim 63:835–853. https://doi.org/10.1007/s00158-020-02750-w

Schmidt MP, Pedersen CB, Gout C (2019) On structural topology optimization using graded porosity control. Struct Multidiscip Optim 60:1437–1453. https://doi.org/10.1007/s00158-019-02275-x

Schmidt S, Schulz V (2011) A 2589 line topology optimization code written for the graphics card. Comput Vis Sci 14:249–256. https://doi.org/10.1007/s00791-012-0180-1

Svanberg K (1987) The method of moving asymptotes—a new method for structural optimization. Int J Numer Meth Eng 24(2):359–373. https://doi.org/10.1002/nme.1620240207

Träff EA, Rydahl A, Karlsson S, Sigmund O, Aage N (2023) Simple and efficient gpu accelerated topology optimisation: Codes and applications. Comput Methods Appl Mech Eng 410:116043

Träff EA, Sigmund O, Aage N (2021) Topology optimization of ultra high resolution shell structures. Thin-Walled Struct 160:107349. https://doi.org/10.1016/j.tws.2020.107349

Wadbro E, Berggren M (2009) Megapixel topology optimization on a graphics processing unit. SIAM Rev 51(4):707–721. https://doi.org/10.1137/070699822

Wang C, Zhao Z, Zhou M, Sigmund O, Zhang XS (2021) A comprehensive review of educational articles on structural and multidisciplinary optimization. Struct Multidiscip Optim 64(5):2827–2880. https://doi.org/10.1007/s00158-021-03050-7

Wang J, Bukenberger DR, Niedermayr S, Neuhauser C, Wu J, Westermann R (2025) Sgldbench: A benchmark suite for stress-guided lightweight 3d designs. arXiv preprint arXiv:2501.03068

Wu J, Aage N, Westermann R, Sigmund O (2018) Infill optimization for additive manufacturing–approaching bone-like porous structures. IEEE Trans Visual Comput Graphics 24(2):1127–1140. https://doi.org/10.1109/TVCG.2017.2655523

Wu J, Dick C, Westermann R (2016) A system for high-resolution topology optimization. IEEE Trans Visual Comput Graphics 22(3):1195–1208. https://doi.org/10.1109/TVCG.2015.2502588

Wu J, Zhu J, Gao J, Gao L, Liu H (2024) A cad-oriented parallel-computing design framework for shape and topology optimization of arbitrary structures using parametric level set. Comput Methods Appl Mech Eng 431:117292. https://doi.org/10.1016/j.cma.2024.117292

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.