JAX-FEM: A differentiable GPU-accelerated 3D finite element solver for automatic inverse design and mechanistic data science

Tianju Xue[1], Shuheng Liao[1], Zhengtao Gan[2], Chanwook Park[1], Xiaoyu Xie[1], Wing Kam Liu[1], and Jian Cao *[1]

[1]Department of Mechanical Engineering, Northwestern University, Evanston, IL 60208,USA
[2]Department of Aerospace and Mechanical Engineering, The University of Texas at El Paso, El Paso, TX 79968, USA

arXiv:2212.00964v1 [cs.MS] 2 Dec 2022

Abstract

This paper introduces JAX-FEM, an open-source differentiable finite element method (FEM) library. Constructed on top of Google JAX, a rising machine learning library focusing on high-performance numerical computing, JAX-FEM is implemented with pure Python while scalable to efficiently solve problems with moderate to large sizes. For example, in a 3D tensile loading problem with 7.7 million degrees of freedom, JAX-FEM with GPU achieves around 10× acceleration compared to a commercial FEM code depending on platform. Beyond efficiently solving forward problems, JAX-FEM employs the automatic differentiation technique so that inverse problems are solved in a fully automatic manner without the need to manually derive sensitivities. Examples of 3D topology optimization of nonlinear materials are shown to achieve optimal compliance. Finally, JAX-FEM is an integrated platform for machine learning-aided computational mechanics. We show an example of data-driven multi-scale computations of a composite material where JAX-FEM provides an all-in-one solution from microscopic data generation and model training to macroscopic FE computations. The source code of the library and these examples are shared with the community to facilitate computational mechanics research.

1 Introduction

Research in computational science and engineering involving partial differential equations (PDEs) has long focused on developing efficient numerical algorithms. Yet the overall efficiency of PDE-based computational analysis depends not only on the smart use of computer hardware, but also on the efficient use of human resources [1].

The finite element method (FEM) [2] is one of the most powerful approaches for numerical solutions to PDEs that appear in structural analysis, heat transfer, fluid flow, electromagnetic potential, etc. This paper proposes a library called `JAX-FEM` that aims at automating the finite element analysis workflows and enhancing human productivity. `JAX-FEM` is built on `JAX` [3], a library for high-performance numerical computing and machine learning research. Besides its success in machine learning applications, `JAX` has proven to be a powerful building block for high-performance scientific simulations, including computational fluid dynamics [4, 5], structural dynamics of metamaterials [6], molecular dynamics [7], phase-field simulation of microstructure evolution [8], etc. We emphasize the following three features that differentiate `JAX-FEM` from other FEM libraries:

1. Efficient solution to forward PDE with GPU acceleration;
2. Differentiable simulation for automatic inverse design;
3. Seamless integration with machine learning.

In the following paragraphs, we introduce the background and motivation for the three features in detail.

*jcao@northwestern.edu (corresponding author)

Feature 1 Most existing FEM libraries (e.g., the `Dealii` [9] library) are implemented with compiled languages like `C/C++` or `Fortran`. `JAX-FEM` is implemented with pure `Python`, a higher-level language known for its dynamic nature and flexibility. Due to the `XLA` (Accelerated Linear Algebra) backend of `JAX`, `JAX-FEM` has a highly competitive performance, especially when GPU is available. Therefore, our `Python` frontend provides both users and developers with fast production experience without condemning them to only small-sized problems. Another benefit of `JAX-FEM` is automatic linearization for nonlinear problems. Many FEM libraries require users to derive the linearized incremental form in Newton's method as input to the program, while `JAX-FEM` works directly with the weak form and performs the linearization using automatic differentiation (AD) [10] for the user. While the AD technique is well known to be the workhorse of deep learning [11], its potential in scientific applications has just gained more attention in recent years [12, 13, 14].

Feature 2 Inverse design problems are of great interest in various engineering applications. Solving inverse problems are more challenging and computationally demanding due to the need to iteratively solve the forward problems. Mathematically, inverse problems can often be formulated as PDE-constrained optimization problems [15]. Successfully computing the "sensitivity" (gradient of the objective function to design parameters) is key to gradient-based optimization algorithms. The derivation of the sensitivity, however, can be quite non-trivial particularly when the forward problem involves complicated nonlinear constitutive relations [16]. Based on automatic differentiation, `JAX-FEM` computes the sensitivity in a fully automatic manner, freeing the users from deriving the sensitivities by hand. The dimension of design parameters is usually much larger than the design objective, hence the the adjoint method [17, 18] is used for efficiency.

Feature 3 The ever-increasing interest in data-driven computational mechanics in recent years has posed a strong need for an integrated platform with unified solutions. For example, machine learning-based constitutive models have been a rapidly growing research area, including data-driven elasticity [19], plasticity [20], viscoelasticity [21], etc. Yet, the current workflow requires using multiple tools and transferring data back and forth, which is cumbersome. For instance, simulation data is first generated with certain FEM software, models are then trained using a machine learning library, and finally the trained model is implemented back into the FEM software, often in a hard-coded way. Built on `JAX` and having access to all its machine learning functionalities, `JAX-FEM` provides an ideal platform to solve the sub-problems all in the same ecosystem with high efficiency.

The paper is organized as follows. Section 2 solves several representative forward problems and the computational performance of `JAX-FEM` is compared to an open-source FEM software `FEniCSx` [22] and a commercial software `Abaqus`. Section 3 introduces the formulation of solving inverse problems. Two applications are presented: full-field reconstruction from sparse observations and structural topology optimization. Section 4 discusses the role of `JAX-FEM` as an integrated platform for machine learning-enabled computational mechanics. One numerical example of data-driven multi-scale computations of composite material is presented. The three sections correspond to the three features discussed previously. We then conclude in Section 5 with possible future improvement.

Standard notation is used throughout the paper. Normal fonts are used for scalars, and boldface fonts for vectors (lower-case) and second-order tensors (upper-case). All tensor and vector components are written with respect to a fixed Cartesian coordinate system with orthonormal basis $\{\boldsymbol{e}_i\}$. We denote by $\boldsymbol{I}$ the second-order identity tensor. The prefixes *tr* and *det* indicate the trace and the determinant. The superscript $\top$ means the transpose of a second-order tensor. Let $(\boldsymbol{a}, \boldsymbol{b})$ be vectors, $(\boldsymbol{A}, \boldsymbol{B})$ be second-order tensors and ∇ the gradient operator; we define the following:

$\boldsymbol{a} \cdot \boldsymbol{b} = a_i b_i$, $(\boldsymbol{A} \cdot \boldsymbol{a})_i = A_{il} a_l$, $(\boldsymbol{A} \cdot \boldsymbol{B})_{il} = A_{ip} B_{pl}$, $\boldsymbol{A} : \boldsymbol{B} = A_{il} B_{il}$, $(\nabla \boldsymbol{a})_{il} = \partial_l a_i$, $\nabla \cdot \boldsymbol{a} = \partial_i a_i$ and $(\nabla \cdot \boldsymbol{A})_i = \partial_l A_{il}$. We denote by $H^k(\Omega, \mathbb{R}^{\text{dim}})$ the Sobolev space $W^{k,2}(\Omega, \mathbb{R}^{\text{dim}})$ and the square-integrable function space $L^2 = H^0$. Norm is denoted by $\|\square\|$ for a function while $|\square|$ for a finite-dimensional vector (Euclidean norm). Note that boldface font is also used for large matrix/vector assembled by FEM, e.g., a vector of nodal degrees of freedom (DOF) $\boldsymbol{U}$ or a matrix $\boldsymbol{K}$. We use $\boldsymbol{K}^*$ to denote the adjoint of $\boldsymbol{K}$.

Our code is continuously being developed and available at https://github.com/tianjuxue/jax-am/tree/main/jax_am/fem.

2 Solving forward problems

In this section, we first define the class of problems to solve. Then we discuss several key features of `JAX-FEM` that are distinguished from the classic implementation of FEM, including array programming style and the use of automatic differentiation technique. By solving typical solid mechanics problems of linear elasticity, hyperelasticity, and plasticity with both `JAX-FEM` and `FEniCSx` and comparing their results, we ensure the correctness of `JAX-FEM`. Finally, we conduct a performance test to show the scalability of `JAX-FEM` for efficiently solving large-size problems of DOF around 7.7 million.

2.1 Problem statement: nonlinear FEM

For illustration purposes, let us consider second-order elliptic partial differential equations with the following form: Find $\boldsymbol{u} : \Omega \to \mathbb{R}^{\text{vec}}$ such that

$$\begin{aligned} -\nabla \cdot \big(\boldsymbol{f}(\nabla \boldsymbol{u})\big) &= \boldsymbol{b} \quad \text{in } \Omega, \\ \boldsymbol{u} &= \boldsymbol{u}_D \quad \text{on } \Gamma_D, \\ \boldsymbol{f}(\nabla \boldsymbol{u}) \cdot \boldsymbol{n} &= \boldsymbol{t} \quad \text{on } \Gamma_N. \end{aligned} \tag{1}$$

where $\Omega \subset \mathbb{R}^{\text{dim}}$ is the problem domain, $\boldsymbol{b}$ is the source term, $\boldsymbol{u}_D$ is the Dirichlet boundary condition defined on $\Gamma_D \subset \partial\Omega$, $\boldsymbol{n}$ is the outward normal, $\boldsymbol{t}$ prescribes Neumann boundary condition on $\Gamma_N \subset \partial\Omega$ ($\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \emptyset$), and $\boldsymbol{f} : \mathbb{R}^{\text{vec}\times\text{dim}} \to \mathbb{R}^{\text{vec}\times\text{dim}}$ is a general tensor-valued function that governs the physics of the problem. Here, "vec" is the number of vector variable components, and "dim" is the spatial dimension. In this work, we only consider three-dimensional problems with dim $= 3$.

The weak form of Eq. (1) reads the following: Find $\boldsymbol{u} \in \mathcal{U}$ such that $\forall \boldsymbol{v} \in \mathcal{V}$

$$F(\boldsymbol{u}; \boldsymbol{v}) = \int_\Omega \boldsymbol{f}(\nabla \boldsymbol{u}) : \nabla \boldsymbol{v} \, \mathrm{d}\Omega - \int_{\Gamma_N} \boldsymbol{t} \cdot \boldsymbol{v} \, \mathrm{d}\Gamma - \int_\Omega \boldsymbol{b} \cdot \boldsymbol{v} \, \mathrm{d}\Omega = 0, \tag{2}$$

where the trial and test function spaces are

$$\begin{aligned} \mathcal{U} &= \big\{\boldsymbol{u} \in H^1(\Omega, \mathbb{R}^{\text{dim}}) \; \big| \; \boldsymbol{u} = \boldsymbol{u}_D \text{ on } \Gamma_D\big\}, \\ \mathcal{V} &= \big\{\boldsymbol{v} \in H^1(\Omega, \mathbb{R}^{\text{dim}}) \; \big| \; \boldsymbol{v} = \boldsymbol{0} \text{ on } \Gamma_D\big\}. \end{aligned} \tag{3}$$

Newton's method for solving the nonlinear problem (2) yields the following linearized incremental problem: Find the incremental solution $\delta\boldsymbol{u} \in \mathcal{V}$ such that

$$F'(\boldsymbol{u}, \delta\boldsymbol{u}; \boldsymbol{v}) = -F(\boldsymbol{u}; \boldsymbol{v}), \tag{4}$$

where the Gateaux derivative is defined as

$$F'(\boldsymbol{u}, \delta\boldsymbol{u}; \boldsymbol{v}) = \lim_{\epsilon \to 0} \frac{F(\boldsymbol{u} + \epsilon\delta\boldsymbol{u}; \boldsymbol{v}) - F(\boldsymbol{u}; \boldsymbol{v})}{\epsilon} = \int_\Omega \nabla \boldsymbol{v} : \mathbb{C} : \nabla \delta\boldsymbol{u} \, \mathrm{d}\Omega, \tag{5}$$

where $\mathbb{C}$ is the fourth-order tangent tensor with $C_{ijkl} = f_{ij,kl}$. The Galerkin finite element method discretizes Eq. (4) so that the following finite-dimensional linear system is solved:

$$F'(\boldsymbol{u}^h, \delta\boldsymbol{u}^h; \boldsymbol{v}^h) = -F(\boldsymbol{u}^h; \boldsymbol{v}^h), \tag{6}$$

where $u^h \in \mathcal{U}^h \subset \mathcal{U}$ is the current solution, $\delta u^h \in \mathcal{V}^h \subset \mathcal{V}$ is the incremental solution we need to solve for, and $v^h \in \mathcal{V}^h$ is the test function. Here, $\mathcal{U}^h$ and $\mathcal{V}^h$ are the finite element function spaces.

2.2 Array programming and automatic differentiation

Central to FEM implementation is to assemble the linear system corresponding to Eq. (6). The procedure is shown in Alg. 1, where N_e is the total number of elements and N_d is the number of DOF associated with each element.

Algorithm 1: Conceptual process of matrix assembly

```
Initialize global stiffness matrix K // Sparse matrix
for e ← 1 to N_e do
    Initialize element stiffness matrix K_e // Dense matrix of size N_d × N_d
    for i ← 1 to N_d do
        for j ← 1 to N_d do
            Update K_e[i, j]
    Add K_e to K
Return K
```

When implementing Alg. 1, there are two features that fundamentally distinguish `JAX-FEM` from traditional approaches: array programming and automatic differentiation.

Array programming While for-loops exist in the conceptual illustration of Alg. 1, we never explicitly write any of these for-loops as common practice in `Fortran` or `C/C++` implementation. Instead, array programming style [23] (in the same spirit of `NumPy` [24]) is used for fully utilizing the power of GPU acceleration. The implementation makes use of `jax.vmap`, a core function of `JAX` for vectorized operations.

Automatic differentiation Classic FEM implementation requires computing explicitly the entries of the element stiffness matrix $\boldsymbol{K}_e$ such that

$$\boldsymbol{K}_e[i, j] = \int_{\Omega_e} \nabla\boldsymbol{\phi}_i : \mathbb{C} : \nabla\boldsymbol{\phi}_j \, \mathrm{d}\Omega, \tag{7}$$

where $\boldsymbol{\phi}_i$ is the ith FEM basis function restricted to the element domain Ω_e. `JAX-FEM` does not require explicitly deriving the linearized form as in Eq. (5). Instead, the program works directly with the weak form defined in Eq. (2). We define the element residual vector function $\boldsymbol{r}_e$ as

$$\begin{aligned} \boldsymbol{r}_e : \mathbb{R}^{N_d} &\to \mathbb{R}^{N_d}, \\ \boldsymbol{U}_e &\mapsto \boldsymbol{R}_e, \\ \boldsymbol{R}_e[i] &= \int_{\Omega_e} \boldsymbol{f}(\nabla\boldsymbol{u}^h) : \nabla\boldsymbol{\phi}_i \, \mathrm{d}\Omega, \end{aligned} \tag{8}$$

where $\boldsymbol{U}_e$ is the vector of nodal DOF defined in the element, $\boldsymbol{R}_e$ is the residual vector, and $\boldsymbol{u}^h(\boldsymbol{x}) = \sum_k^{N_d} \boldsymbol{U}_e[k]\boldsymbol{\phi}_k(\boldsymbol{x})$ is the FEM solution field. To obtain $\boldsymbol{K}_e[i, j]$, we simply compute the Jacobian matrix of $\boldsymbol{r}_e$ at $\boldsymbol{U}_e$ such that

$$\boldsymbol{K}_e[i, j] = \frac{\partial \boldsymbol{r}_e}{\partial \boldsymbol{U}_e}[i, j]. \tag{9}$$

We use automatic differentiation provided by `JAX` to compute this Jacobian matrix. In many applications, e.g., plasticity, the fourth-order tangent tensor $\mathbb{C}$ is nontrivial to derive, and `JAX-FEM` frees developers from this tedious procedure.

2.3 Linear elasticity, hyperelasticity, and plasticity

To verify the correctness of `JAX-FEM`, we consider three typical solid mechanics problems, i.e., linear elasticity, hyperelasticity, and plasticity. Specifically, we impose uniaxial tensile loadings on a cylinder (see Fig. 1 (a)) where the bottom boundary is fixed and the top boundary is subject to fixed displacement conditions. The height of the cylinder is 10 mm and the radius is 5 mm. For simplicity, we assume zero body force and free traction force for all three problems. We solve the problems with both `JAX-FEM` and `FEniCSx` and compare the results.

Linear elasticity We replace the tensor function $\boldsymbol{f}$ in Eq. (1) with the Cauchy stress $\boldsymbol{\sigma}$ so that the governing equation is

$$\begin{aligned} -\nabla \cdot \boldsymbol{\sigma} &= \boldsymbol{0} \quad \text{in } \Omega, \\ \boldsymbol{u} &= \boldsymbol{u}_D \quad \text{on } \Gamma_D, \\ \boldsymbol{\sigma} \cdot \boldsymbol{n} &= \boldsymbol{0} \quad \text{on } \Gamma_N, \end{aligned} \tag{10}$$

where we have

$$\begin{aligned} \boldsymbol{\sigma} &= \lambda \operatorname{tr}(\boldsymbol{\varepsilon}) \boldsymbol{I} + 2\mu \boldsymbol{\varepsilon}, \\ \boldsymbol{\varepsilon} &= \frac{1}{2}\left[\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^\top\right], \\ \boldsymbol{f}(\nabla \boldsymbol{u}) &= \boldsymbol{\sigma}, \end{aligned} \tag{11}$$

where $\boldsymbol{I}$ is the identity tensor and λ and μ are the Lamé parameters. We assume quasi-static incremental loadings from 0 to 0.1 mm with 10 steps, and show the plot of force versus displacement in Fig. 1 (b), where the results agree well between `JAX-FEM` and `FEniCSx`.

Hyperelasticity For a typical neo-Hookean solid, the governing equation is

$$\begin{aligned} -\nabla \cdot \boldsymbol{P} &= \boldsymbol{0} \quad \text{in } \Omega, \\ \boldsymbol{u} &= \boldsymbol{u}_D \quad \text{on } \Gamma_D, \\ \boldsymbol{P} \cdot \boldsymbol{n} &= \boldsymbol{0} \quad \text{on } \Gamma_N, \end{aligned} \tag{12}$$

where $\boldsymbol{P}$ is the first Piola-Kirchhoff stress. Eq. (12) is simply a specific form of Eq. (1) in the sense that $\boldsymbol{f}$ can be defined through

$$\begin{aligned} \boldsymbol{P} &= \frac{\partial W}{\partial \boldsymbol{F}}, \\ \boldsymbol{F} &= \nabla \boldsymbol{u} + \boldsymbol{I}, \\ W(\boldsymbol{F}) &= \frac{G}{2}(J^{-2/3} I_1 - 3) + \frac{\kappa}{2}(J-1)^2, \\ \boldsymbol{f}(\nabla \boldsymbol{u}) &= \boldsymbol{P}, \end{aligned} \tag{13}$$

where $\boldsymbol{F}$ is the deformation gradient, W is the strain energy density function, $J = \det(\boldsymbol{F})$, $I_1 = \operatorname{tr}(\boldsymbol{C})$; $G = \frac{E}{2(1+\nu)}$ and $\kappa = \frac{E}{3(1-2\nu)}$ denote the initial shear and bulk moduli, respectively, with E being the Young's modulus and and ν the Poisson's ratio of the material. The above W is commonly used to model isotropic elastomers that are almost incompressible [25]. We assume quasi-static incremental loadings from 0 to 2 mm with 10 steps. We show the plot of force versus displacement in Fig. 1 (c), where the results agree well between `JAX-FEM` and `FEniCSx`.

Plasticity For perfect J2-plasticity model [26], we assume that the total strain $\boldsymbol{\varepsilon}^{k-1}$ and stress $\boldsymbol{\sigma}^{k-1}$ from the previous loading step are known, and the problem states that find the displacement field $\boldsymbol{u}^k$ at the current loading step such that

$$
\begin{aligned}
-\nabla \cdot \left(\boldsymbol{f}(\nabla \boldsymbol{u}^k, \boldsymbol{\varepsilon}^{k-1}, \boldsymbol{\sigma}^{k-1})\right) &= \boldsymbol{0} \quad \text{in } \Omega, \\
\boldsymbol{u}^k &= \boldsymbol{u}_D \quad \text{on } \Gamma_D, \\
\boldsymbol{f} \cdot \boldsymbol{n} &= \boldsymbol{0} \quad \text{on } \Gamma_N.
\end{aligned} \tag{14}
$$

Eq. (14) is a specific form of Eq. (1), where the function $\boldsymbol{f}$ is defined with the following plasticity-related equations

$$
\begin{aligned}
\boldsymbol{\sigma}_{\text{trial}} &= \boldsymbol{\sigma}^{k-1} + \Delta\boldsymbol{\sigma}, \\
\Delta\boldsymbol{\sigma} &= \lambda \, \text{tr}(\Delta\boldsymbol{\varepsilon})\boldsymbol{I} + 2\mu \, \Delta\boldsymbol{\varepsilon}, \\
\Delta\boldsymbol{\varepsilon} &= \boldsymbol{\varepsilon}^k - \boldsymbol{\varepsilon}^{k-1} = \frac{1}{2}\left[\nabla \boldsymbol{u}^k + (\nabla \boldsymbol{u}^k)^\top\right] - \boldsymbol{\varepsilon}^{k-1}, \\
\boldsymbol{s} &= \boldsymbol{\sigma}_{\text{trial}} - \frac{1}{3}\text{tr}(\boldsymbol{\sigma}_{\text{trial}})\boldsymbol{I}, \\
s &= \sqrt{\frac{3}{2}\boldsymbol{s} : \boldsymbol{s}}, \\
f_{\text{yield}} &= s - \sigma_{\text{yield}}, \\
\boldsymbol{\sigma}^k &= \boldsymbol{\sigma}_{\text{trial}} - \frac{\boldsymbol{s}}{s}\langle f_{\text{yield}} \rangle_+, \\
\boldsymbol{f}(\nabla \boldsymbol{u}^k, \boldsymbol{\varepsilon}^{k-1}, \boldsymbol{\sigma}^{k-1}) &= \boldsymbol{\sigma}^k,
\end{aligned} \tag{15}
$$

where $\boldsymbol{\sigma}_{\text{trial}}$ is the elastic trial stress, $\boldsymbol{s}$ is the devitoric part of $\boldsymbol{\sigma}_{\text{trial}}$, f_{yield} is the yield function, σ_{yeild} is the yield strength, $\langle x \rangle_+ := \frac{1}{2}(x + |x|)$ is the ramp function, and $\boldsymbol{\sigma}^k$ is the stress at the currently loading step. Deriving the four-order elastoplastic tangent moduli tensor $\mathbb{C}$ is usually required by traditional FEM implementation, but is not needed by `JAX-FEM` due to automatic differentiation. We assume quasi-static loadings from 0 to 0.1 mm and then unload from 0.1 mm to 0. We show the plot of the z-z component of volume-averaged stress versus displacement of the top surface in Fig. 1 (d), where the path-dependent results match exactly between `JAX-FEM` and `FEniCSx`.

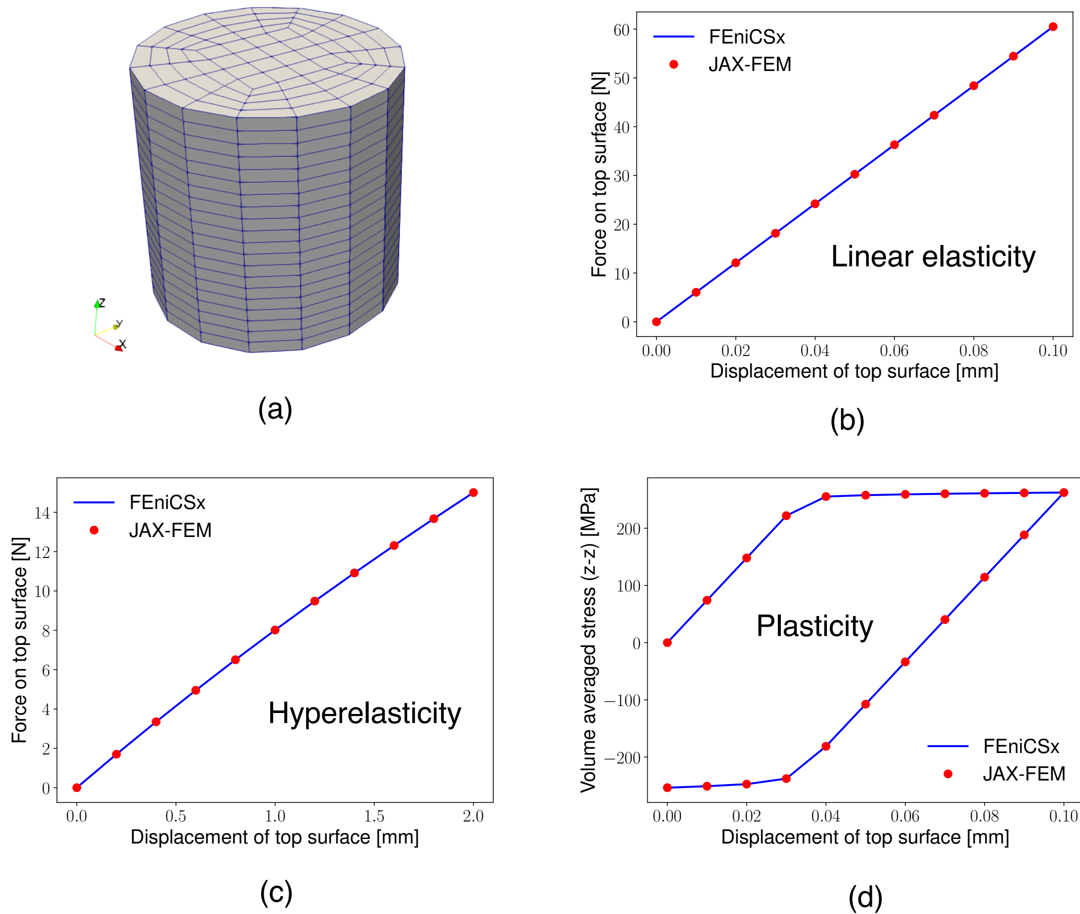

Figure 1: Testing JAX−FEM with FEniCSx being the ground truth, where (a) is the problem mesh, (b) considers a linear elastic material, (c) considers a hyperelastic material, and (d) considers an elasto-plastic material.

Besides the comparison results above, we have maintained a test suite in our open source repository for more tests like body force, Neumann boundary conditions, etc., so that each new version of JAX−FEM must pass these tests.

2.4 A sample user program

The interface of JAX−FEM for users is succinct. Here, we revisit the tensile problem for the cylinder with linear elastic material. As shown in the code snippet below, users first create the cylinder mesh. Dirichlet boundary conditions are imposed component-wisely, i.e., we need to specify conditions for bottom-x, bottom-y, bottom-z, top-x, top-y, and top-z separately. Then, mesh and dirichlet_bc_info are passed to the FEM model LinearElasticity and the solver solves the problem

```
import jax.numpy as np
from jax_am.fem.models import LinearElasticity
from jax_am.fem.solver import solver
from jax_am.fem.generate_mesh import cylinder_mesh

mesh = cylinder_mesh()

bottom = lambda point: np.isclose(point[2], 0.)
top = lambda point: np.isclose(point[2], 10.)
zero_disp = lambda point: 0.
```

```
top_z_disp = lambda point: 0.1

dirichlet_bc_info = [[bottom, bottom, bottom, top, top, top],
                     [0, 1, 2, 0, 1, 2],
                     [zero_disp, zero_disp, zero_disp,
                      zero_disp, zero_disp, top_z_disp]]

problem = LinearElasticity(mesh, dirichlet_bc_info)
solution = solver(problem)
```

2.5 Performance and scalability

JAX-FEM is based on JAX and uses just-in-time compilation for high performance. Therefore, the fact that JAX-FEM is written in pure Python does not limit us to only small problems. For benchmarking the performance, we solve a standard uniaxial tensile loading problem on an ASTM D638 Type 1 specimen (see Fig. 2) assuming linear elastic material.

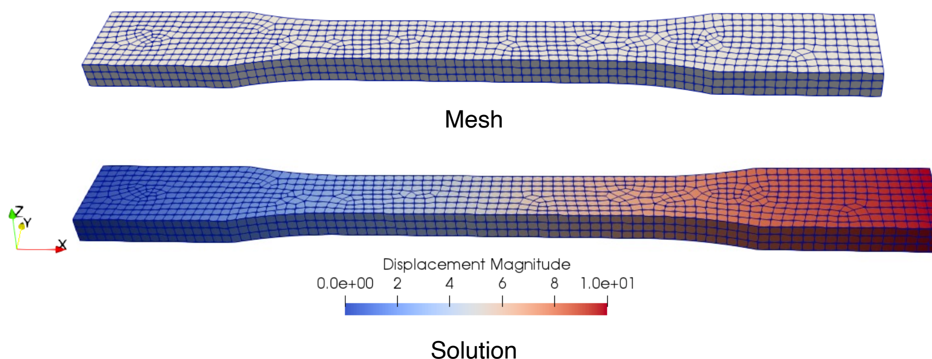

Figure 2: The dog bone shaped ASTM D638 Type 1 specimen, which is a standard test specimen for tensile properties of polymers [27]. The tensile experiment fixes the left side and pulls the right side with a prescribed displacement condition.

The same problem with different levels of mesh resolution is solved using JAX-FEM with CPU-only mode and with GPU, FEniCSx with MPI for parallel programming, and Abaqus running on CPU with/without MPI. The wall time measurements with respect to the number of DOF are shown in Fig. 3. JAX-FEM shows a predominant advantage when GPU is used. The largest problem has 7,703,841 DOF and takes 8409 s and 4769 s for Abaqus with CPU and MPI (24 cores), respectively, and 523 s for JAX-FEM on GPU. JAX-FEM on GPU achieves 16.1× and 9.1× acceleration compared to Abaqus on CPU and with MPI, respectively. The problem in Fig. 2 has 10,224 DOF and corresponds to the first column of data points in Fig. 3.

Note that for Abaqus the MPI acceleration is not significant as the number of DOF becomes larger. For example, the Abaqus MPI speedup compared with Abaqus CPU is 9.1× for 2,344,230 DOF but 1.8× for 7,703,841 DOF. This decreased performance in Abaqus as DOF increases is attributed to the severe message passing delay for large DOF problems. In Abaqus, if the size of transient variables (mainly the global stiffness matrix) exceeds CPU memory limit, they are stored on local storage where most of the delay takes place. It is also worth mentioning that with 24 cores, we need 19 paid tokens. Compared to this, JAX-FEM is an open-source software and faster than Abaqus with an extended license for tokens.

We report the platforms for those numerical experiments. JAX-FEM runs on 2.3 GHz Intel(R) Xeon(R) W-2195 CPU (18 cores) with NVIDIA Quadro RTX 8000 GPU (48 GB Graphics memory) on Ubuntu 20.04.5 LTS. FEniCSx runs on 2.4 GHz Intel i9 CPU (8 cores) on macOS Big Sur 11.6.5.

Abaqus CPU runs on Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz under Windows operating system. Abaqus CPU with MPI runs on Intel(R) Xeon (R) CPU E5-2680 v3 @ 2.50 GHz (total 24 cores) with 19 tokens under CentOS 6.9 operating system.

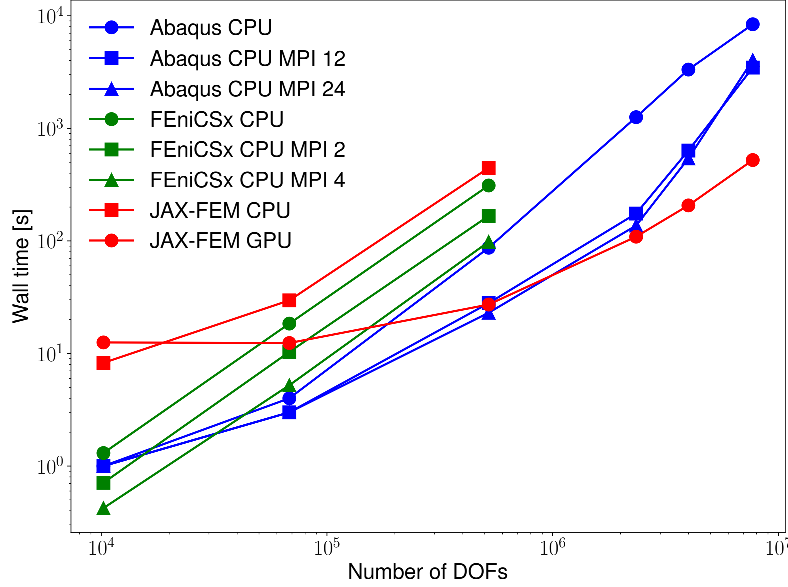

Figure 3: Performance report. Here, "FEniCSx CPU MPI 4" means FEniCSx runs with 4 processes of MPI parallel programming.

3 Solving inverse problems

We formulate inverse problems as PDE-constrained optimization (PDE-CO) problems. There are in general two strategies for solving PDE-CO problems: optimize-then-discretize and discretize-then-optimize [28, 29]. We follow the discretize-then-optimize approach. The discretized PDE-CO problem is formulated as

$$\begin{aligned} \min_{\boldsymbol{U}\in\mathbb{R}^N,\boldsymbol{\theta}\in\mathbb{R}^M} & \mathcal{J}(\boldsymbol{U},\boldsymbol{\theta}) \\ \text{s.t.} \quad & \boldsymbol{C}(\boldsymbol{U},\boldsymbol{\theta}) = \boldsymbol{0}, \end{aligned} \tag{16}$$

where $\boldsymbol{U}$ is the finite element solution vector of DOF, $\boldsymbol{\theta}$ is the parameter vector, and $\mathcal{J}(\cdot,\cdot) : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}$ is the objective function. The constraint function $\boldsymbol{C}(\cdot,\cdot) : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^N$ represents the discretized governing PDE and should be regarded as the direct consequence of discretizing the weak form in Eq. (2) and imposing Dirichlet boundary conditions.

A reduced formulation is used to embed the PDE constraint so that the problem posed in (16) is re-formulated as

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^M} \widehat{\mathcal{J}}(\boldsymbol{\theta}), \tag{17}$$

where $\widehat{\mathcal{J}}(\boldsymbol{\theta}) := \mathcal{J}(\boldsymbol{U}(\boldsymbol{\theta}),\boldsymbol{\theta})$ and $\boldsymbol{U}(\boldsymbol{\theta})$ is the implicit function that arises from solving the PDE. For efficient optimization algorithms, gradient information is necessary. The total derivative of $\widehat{\mathcal{J}}$ with respect to parameters $\boldsymbol{\theta}$ is computed with chain rules

$$\frac{\mathrm{d}\widehat{\mathcal{J}}}{\mathrm{d}\boldsymbol{\theta}} = \frac{\partial\mathcal{J}}{\partial\boldsymbol{U}}\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}\boldsymbol{\theta}} + \frac{\partial\mathcal{J}}{\partial\boldsymbol{\theta}}, \tag{18}$$

where the existence of the derivative $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}\boldsymbol{\theta}}$ is justified by the implicit function theorem [30] under certain mild conditions. We then take the derivative of the constraint function in Eq. (16) with respect to $\boldsymbol{\theta}$ so that the following relations are obtained

$$\frac{\mathrm{d}\boldsymbol{C}}{\mathrm{d}\boldsymbol{\theta}} = \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{U}}\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}\boldsymbol{\theta}} + \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{\theta}} = 0. \tag{19}$$

Hence,

$$\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}\boldsymbol{\theta}} = -\left(\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{U}}\right)^{-1}\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{\theta}}. \tag{20}$$

Substitute Eq. (20) to Eq. (18), we obtain

$$\frac{\mathrm{d}\widehat{\mathcal{J}}}{\mathrm{d}\boldsymbol{\theta}} = -\overbrace{\frac{\partial \mathcal{J}}{\partial \boldsymbol{U}}\left(\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{U}}\right)^{-1}}^{\text{adjoint PDE}}\hspace{-3em}\underbrace{\hphantom{\left(\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{U}}\right)^{-1}}\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{\theta}}}_{\text{tangent linear PDE}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}, \tag{21}$$

where the first term can be evaluated either from left to right (solving the adjoint PDE first) or from right to left (solving the tangent linear PDE first). When the size of the parameter vector $\boldsymbol{\theta}$ is larger than that of the objective (e.g., $M \gg 1$ in our case), it is more efficient to solve the adjoint PDE first, giving the name adjoint method [18]. For interested readers, Xu and Darve [31] recently presented a detailed discussion on the cost comparison of the adjoint method and the tangent linear approach. We continue the discussion by adopting the adjoint method. The adjoint PDE is

$$\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{U}}^{*}\boldsymbol{\lambda} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{U}}^{*}, \tag{22}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^N$ is the adjoint variable. Substitute $\boldsymbol{\lambda}$ to Eq. (21) we have

$$\frac{\mathrm{d}\widehat{\mathcal{J}}}{\mathrm{d}\boldsymbol{\theta}} = -\boldsymbol{\lambda}^{*}\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}. \tag{23}$$

Note that Eq. (22) is a linear PDE to solve, but it relies on the Jacobian matrix $\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{U}}$, which requires the solution vector $\boldsymbol{U}$. Therefore, the computational cost is largely dominated by solving the forward problem, not the adjoint PDE, especially when the forward PDE is nonlinear.

The derivations above are abstract and problem independent. When solving specific problems, one typically needs to further derive the concrete expressions of those derivatives, e.g., $\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{\theta}}$, which is often tedious and error-prone. In `JAX-FEM`, we use automatic differentiation to compute these derivatives, which greatly enhances productivity. The following code snippet demonstrates an example of given $\boldsymbol{C}(\cdot,\boldsymbol{\theta})$ (`partial_constraint_fn`) and computing $\boldsymbol{\lambda}^{*}\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{\theta}}$ (`result`) using `JAX` function `jax.vjp`, which stands for vector-Jacobian product.

```
# params: JAX array of shape (M,)
# adjoint: JAX array of shape (N,)
def vec_jac_prod_fn(v):
    primals, vec_jac_prod = jax.vjp(partial_constraint_fn, params)
    val, = vec_jac_prod(v)
    return val
result = vec_jac_prod_fn(adjoint)
```

Similar approaches are applied to other derivative-related computations like $\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{U}}^{*}\boldsymbol{\lambda}$ in Eq. (22) so that the entire workflow of computing the total derivative $\frac{\mathrm{d}\widehat{\mathcal{J}}}{\mathrm{d}\boldsymbol{\theta}}$ is fully automatic. For general information about automatic differentiation involving implicit functions, we refer to the recent work

by Blondel et al. [32]. As a summary, we show the overall workflow of solving PDE-CO problems in Alg. 2. In general, the "optimizer" in the algorithm can use any off-the-shelf gradient-based optimization algorithms. The simplest one is the gradient descent method, but more sophisticated algorithms are usually required given the complexity of the specific problem.

Algorithm 2: PDE-constrained optimization with the adjoint method

Input: $\boldsymbol{\theta}_{\text{ini}}$, $i_{\max}$ `// Initial parameter and maximum iteration number`
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{\text{ini}}$, $i \leftarrow 0$
while $i < i_{\max}$ **do**
  $\boldsymbol{U} \leftarrow \text{ForwardPDESolver}(\boldsymbol{\theta})$ `// See constraint function in Eq. (16)`
  $\frac{\mathrm{d}\widehat{\mathcal{J}}}{\mathrm{d}\boldsymbol{\theta}} \leftarrow \text{AdjointMethod}(\boldsymbol{U}, \boldsymbol{\theta})$ `// See Eq. (17) to Eq. (23)`
  $\boldsymbol{\theta} \leftarrow \text{Optimizer}(\boldsymbol{\theta}, \frac{\mathrm{d}\widehat{\mathcal{J}}}{\mathrm{d}\boldsymbol{\theta}})$ `// Gradient-based optimizer`
  $i \leftarrow i + 1$
Output: $\boldsymbol{U}$, $\boldsymbol{\theta}$

In the next two subsections, we pose specific PDE-CO problems in the form of Eq. (16) and use `JAX-FEM` for solutions.

3.1 Full field inference from sparse observations

In this numerical example, we consider predicting the full scalar field with sparse observations at certain randomly picked points. The forward governing PDE is a linear Poisson's equation:

$$
\begin{aligned}
-\alpha \Delta u &= b \quad \text{in } \Omega, \\
u &= 0 \quad \text{on } \partial\Omega,
\end{aligned}
\tag{24}
$$

where α is a constant coefficient and b is the source function that we can control. The weak form states that find the solution u so that for any test function v we have

$$
\int_{\Omega} \alpha \nabla u \cdot \nabla v \, \mathrm{d}\Omega = \int_{\Omega} b\, v \, \mathrm{d}\Omega,
\tag{25}
$$

The PDE-CO problem states that

$$
\begin{aligned}
\min_{\boldsymbol{U} \in \mathbb{R}^N, \boldsymbol{\theta} \in \mathbb{R}^M} & \sum_{i \in \mathcal{I}_{\text{obs}}} (\boldsymbol{U}[i] - U_{i,\text{obs}})^2 \\
\text{s.t.} \quad & \boldsymbol{A}\,\boldsymbol{U} = \boldsymbol{F}(\boldsymbol{\theta}),
\end{aligned}
\tag{26}
$$

where $\boldsymbol{U}$ is the DOF vector of u, $\boldsymbol{\theta}$ is the discretized version of b, $\boldsymbol{U}[i]$ is the ith component of $\boldsymbol{U}$, $U_{i,\text{obs}}$ is the ith observed value, $\mathcal{I}_{\text{obs}}$ is the index set of observations, and the constraint equation $\boldsymbol{A}\,\boldsymbol{U} = \boldsymbol{F}(\boldsymbol{\theta})$ is the discretized version of the weak form (25).

The problem domain Ω is a $1 \times 1 \times 0.2$ rectangular box discretized with $50 \times 50 \times 10$ linear hexahedral elements. We randomly pick 250 points for observing the true solution values, as shown in the top panel of Fig. 4. The ground truth solution obtained from solving Eq. (24) is shown in the lower left panel of Fig. 4. The source term function b is set to have a bimodal shape such that $b(\boldsymbol{x}) = 10 \cdot \exp(-10 \cdot |\boldsymbol{x} - (2.5, 2.5, 5.0)|^2) + 10 \cdot \exp(-10 \cdot |\boldsymbol{x} - (7.5, 7.5, 5.0)|^2)$. The predicted solution by solving the PDE-CO problem is shown in the lower right panel Fig. 4.

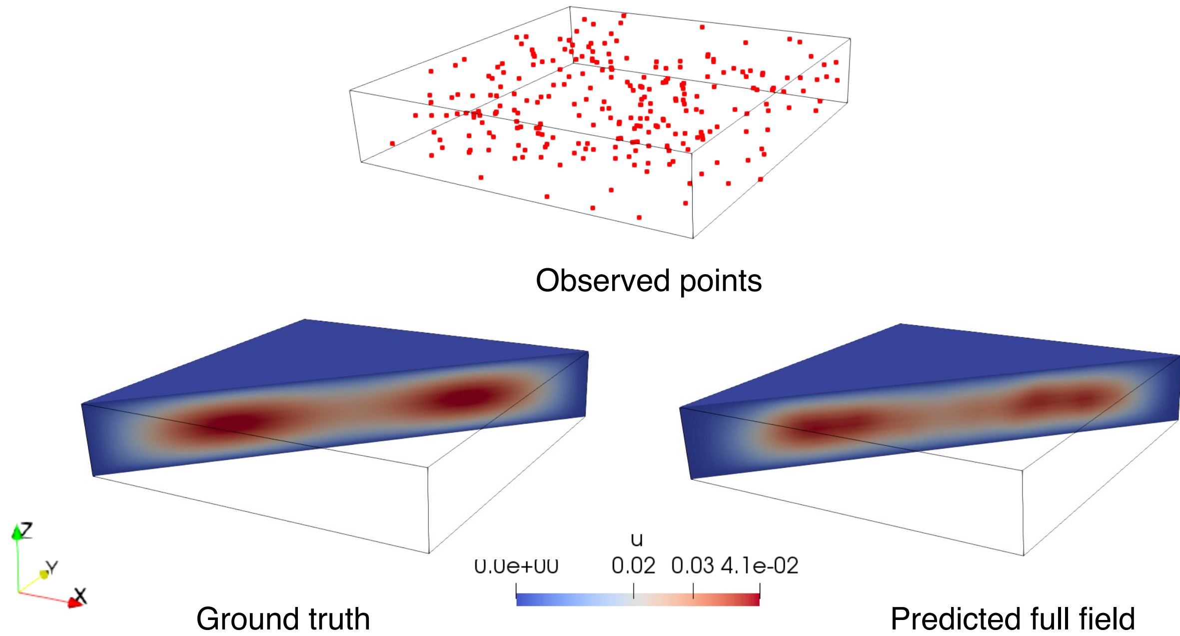

Figure 4: Configurations and results of the full field prediction example. The top panel shows the 250 points where the solution value can be observed. The left panel shows the ground truth solution. The right panel shows the predicted solution field with PDE-CO.

We show the optimization iterations of the PDE-CO problem in Fig. 5 (a). In the plot, the objective value (y-axis) is defined in Eq. (26) and the optimization step (x-axis) is defined as each time the gradient information is queried by the optimizer (see Alg. 2). In this problem, the limited-memory BFGS algorithm [33] provided by the `SciPy` [34] package is used as the optimizer. As shown, the objective value quickly drops to nearly zero within only 20 steps. To quantitatively show the error of the predicted full field solution u_{pred} compared with the ground truth u_{true}, we define the relative L^2 norm error $\frac{\|u_{\text{pred}}-u_{\text{true}}\|_{L^2}}{\|u_{\text{true}}\|_{L^2}}$ and show this inference error in Fig. 5 (b). As seen, the error is about 12.0% when the optimization is over. Note that this is the result of observing only 250 points, which is less than 1% of the total points, and we anticipate the error to decrease if more points are observed. For example, with 2500 observed points, the error is decreased to about 1.4%.

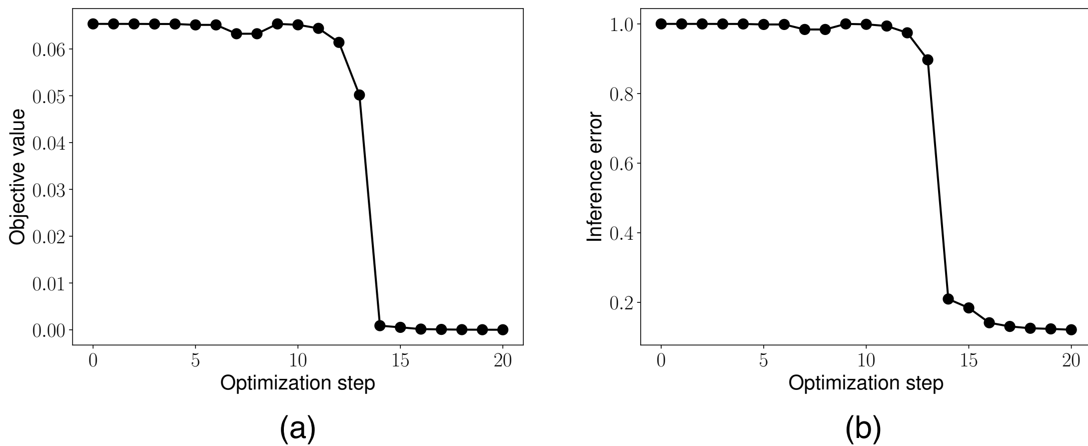

Figure 5: Optimization results of the full field prediction example. Subfigure (a) shows the optimization objective converging to zero, and subfigure (b) shows the relative inference error of the predicted solution to the true solution.

In this PDE-CO problem, the total derivative $\frac{\mathrm{d}\widehat{J}}{\mathrm{d}\boldsymbol{\theta}}$ is computed automatically by `JAX-FEM`. We perform a sanity test here to ensure that this derivative is computed correctly. Following the works of [35, 36], Taylor test can be used to check the accuracy of the computed gradient. Given a perturbation $\delta\boldsymbol{\theta}$, the convergence rate of the residual should be 1 using a zeroth-order expansion:

$$r_{\text{zeroth}} = \left|\widehat{J}(\boldsymbol{\theta} + h\delta\boldsymbol{\theta}) - \widehat{J}(\boldsymbol{\theta})\right| \to 0 \text{ at } \mathcal{O}(h), \tag{27}$$

and the convergence rate should be 2 by a first-order expansion:

$$r_{\text{first}} = \left|\widehat{J}(\boldsymbol{\theta} + h\delta\boldsymbol{\theta}) - \widehat{J}(\boldsymbol{\theta}) - h\frac{\mathrm{d}\widehat{J}}{\mathrm{d}\boldsymbol{\theta}} \cdot \delta\boldsymbol{\theta}\right| \to 0 \text{ at } \mathcal{O}(h^2). \tag{28}$$

The results above are the direct consequence of Taylor's theorem [30]. We set the step size h to be $h = 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$ and calculate r_{zeroth} and r_{first}, and show the results in Fig. 6. As expected, the $r_{\text{zeroth}} \propto h$ and the $r_{\text{first}} \propto h^2$, which demonstrates the correct computation of the gradient by JAM-FEM.

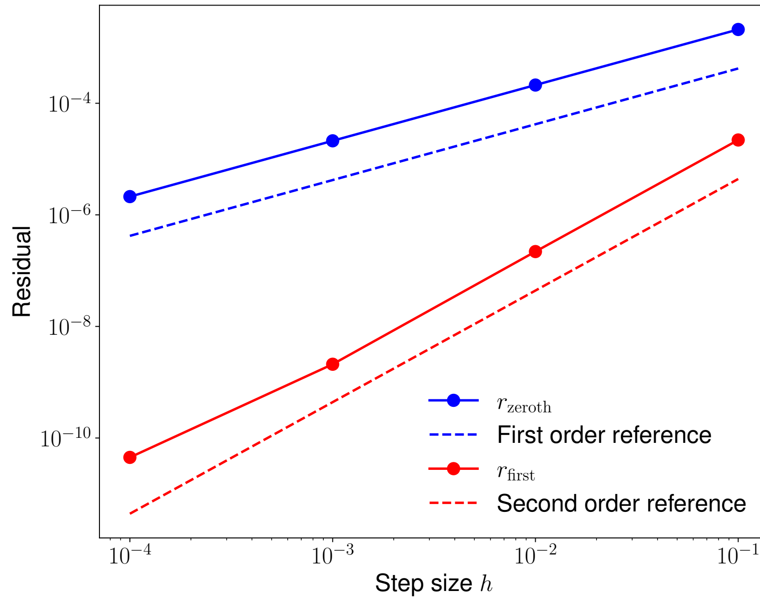

Figure 6: Taylor test results. As expected, the zeroth-order expansion of the residual achieves a first order convergence, and the first-order expansion achieves a second order convergence.

3.2 Topology optimization

As the second numerical example of PDE-CO, we consider topology optimization [37], an important field that is well-received and developed. We first study compliance minimization of a thin plate made of a hyperelastic material, as shown in the left panel of Fig. 7. Following the classic Solid Isotropic Material with Penalization (SIMP) [37] method, the governing PDE is

$$\begin{aligned} -\nabla \cdot (\theta^p \boldsymbol{P}) &= \boldsymbol{0} \quad \text{in } \Omega, \\ \boldsymbol{u} &= \boldsymbol{0} \quad \text{on } \Gamma_D, \\ \theta^p \boldsymbol{P} \cdot \boldsymbol{n} &= \boldsymbol{t} \quad \text{on } \Gamma_N, \end{aligned} \tag{29}$$

which is similar to Eq. (12), except that $\theta(\boldsymbol{x}) \in [0, 1]$ is the continuous design density field and p is the penalty exponent. The weak form states that find the solution $\boldsymbol{u}$ so that for any test function $\boldsymbol{v}$ we have

$$\int_{\Omega} \theta^p \boldsymbol{P} : \nabla \boldsymbol{v} \, \mathrm{d}\Omega - \int_{\Gamma_N} \boldsymbol{t} \cdot \boldsymbol{v} \, \mathrm{d}\Gamma = 0. \tag{30}$$

The compliance minimization problem states that

$$\min_{\boldsymbol{U}\in\mathbb{R}^N,\boldsymbol{\theta}\in\mathbb{R}^M} \int_{\Gamma_N} \boldsymbol{u}^h \cdot \boldsymbol{t}$$
$$\text{s.t.} \quad \boldsymbol{C}(\boldsymbol{U},\boldsymbol{\theta}) = \boldsymbol{0}, \tag{31}$$

where $\boldsymbol{u}^h(\boldsymbol{x}) = \sum_k \boldsymbol{U}[k]\boldsymbol{\phi}_k(\boldsymbol{x})$ is the finite element solution field constructed with $\boldsymbol{U}$, $\boldsymbol{\theta}$ is the discretized version of θ, and the constraint equation $\boldsymbol{C}(\boldsymbol{U},\boldsymbol{\theta}) = \boldsymbol{0}$ matches the discretized version of Eq. (30). With `JAX-FEM`, we bypass the need to further perform sensitivity analysis, which is usually required in topology optimization. The sensitivity information is computed by the program automatically. The optimized topological structure of this thin plate is shown in the right panel of Fig. 7. The optimizer used in this example is the method of moving asymptotes (MMA) [38]. We have also passed a constraint that requires to only use 50% of the material, i.e., $\int_\Omega \theta \mathrm{d}\Omega / \int_\Omega \mathrm{d}\Omega = 0.5$ to the optimizer. To avoid checkerboard patterns [39], a convolution filter is used to blur the calculated sensitivities. The implementation of MMA in `Python` is borrowed from the work of Chandrasekhar et al. [40].

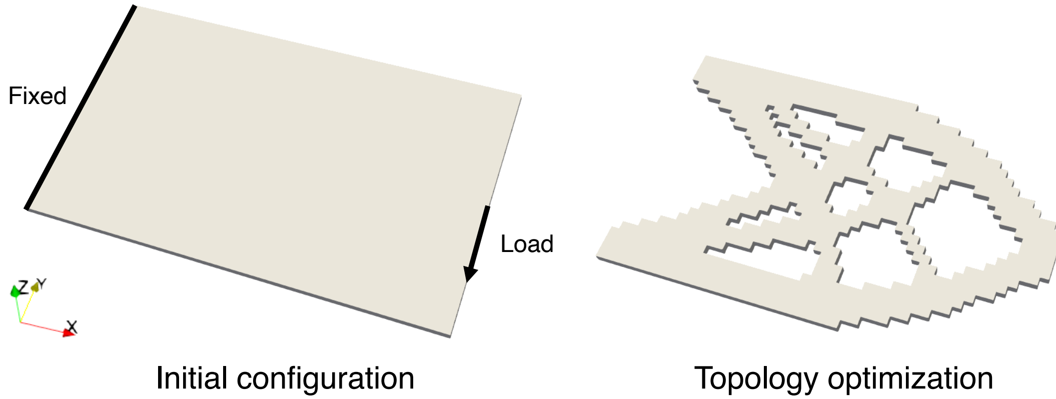

Figure 7: Topology optimization of a thin plate.

The compliance versus optimization step is shown in Fig. 8, where the final structure has a compliance value of 15.63 μJ. As a reminder, the original solid plate with full material has the compliance to be 9.46 μJ.

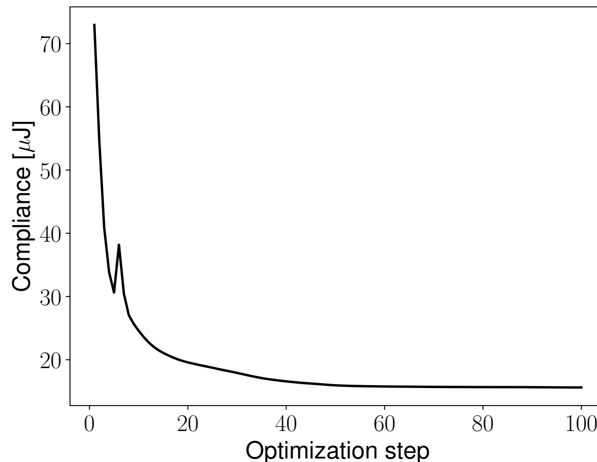

Figure 8: Objective value versus optimization step for the thin plate example.

As a more realistic case with more complex geometry, we use topology optimization for the lightweight design of a bracket with three screw holes. The bracket is assumed to be made of a linear elastic material and the boundary conditions are shown in the upper panel of Fig. 9. We limit the design space to the blue box region (see Fig. 9), and prohibit any material change outside of the blue box. A reasonable human design is shown in the lower left panel that uses 45% material of the blue box region, and the compliance is 3.07 μJ. The deign out of topology optimization that uses the same amount of material is shown in the lower right panel of Fig. 9, whose compliance is 1.10 μJ, achieving a reduction of 64.2% compared with the human design. The optimization iterations are shown in Fig. 10 where the compliance value is plotted against the optimization step.

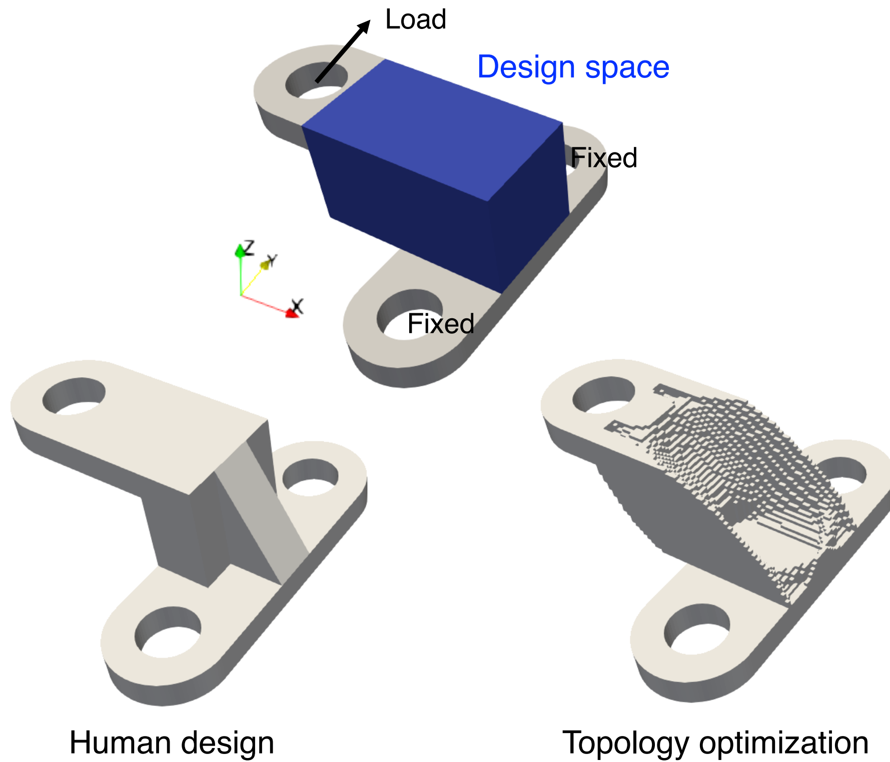

Figure 9: Topology optimization of a three-hole bracket. As shown in the top panel, fixed boundary conditions are applied on the inner walls of the two holes, and uniform loading condition along the positive y-axis is applied on the inner wall of the upper hole. The lower left panel shows a human design while the lower right panel shows the result of topology optimization that uses the same amount of material.

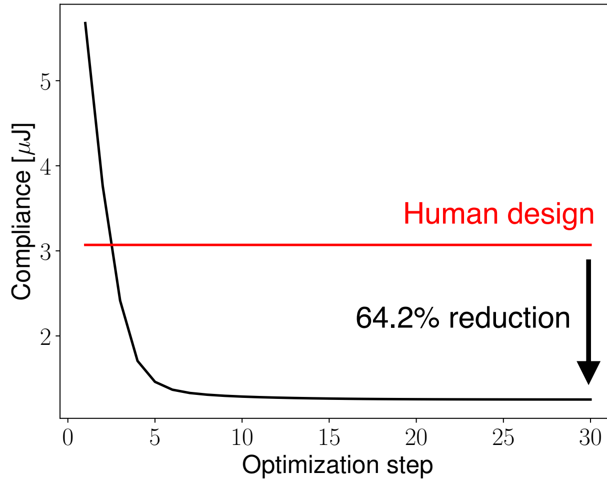

Figure 10: Objective value versus optimization step for the bracket example.

4 Integration with machine learning

In this section, we show one numerical example of solving a data-driven multi-scale computation problem. JAX-FEM provides an all-in-one solution to problems of this kind.

4.1 Data-driven homogenization of composite material

We consider a composite material whose representative volume element (RVE) is a 1 mm cube of the mixture of a soft material (E =100 MPa; ν = 0.4) and a hard material (E =1000 MPa; $\nu = 0.3$), as shown in the left panel of Fig. 12. Both soft and hard materials are assumed to be nearly incompressible neo-Hookean solids (similar to Eq. (13)). The multi-scale computational scheme follows our previous work [41] on data-driven homogenization of mechanical meta-materials. The basic workflow involves three major steps:

1. Performing RVE-level FEM computations and collecting data;
2. Training the neural network that represents the homogenized constitutive relationship;
3. Deploying the trained model to solve a macroscopic problem.

The three steps are described in Fig. 11, where $\overline{\boldsymbol{C}}$ is the macroscopic right Cauchy-Green tensor and $\overline{W}$ is the macroscopic strain energy density function. For usual workflows, one needs to conduct step 1 with FEM software such as Abaqus, conduct step 2 with a machine learning library such as PyTorch [42] and then perhaps most tediously in step 3 get back to the FEM software so that the trained neural network parameters are hard-coded in the user-defined material model program. In our work, all these three steps are performed in JAX-FEM, which is efficient and convenient. In the next three subsections, we introduce the three steps in more detail.

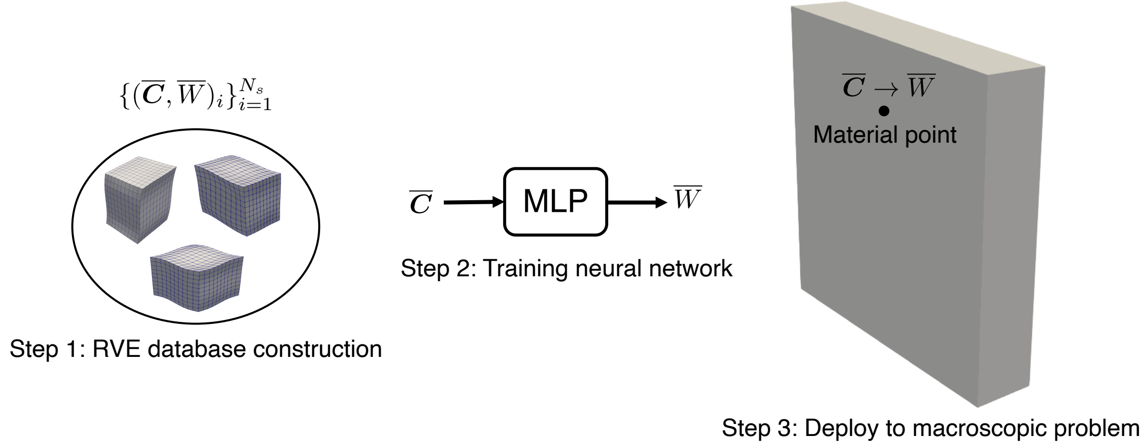

Figure 11: Workflow of the data-driven multi-scale computational scheme. "MLP" stands for multi-layer perceptron.

4.1.1 RVE database construction

For each RVE problem, we prescribe a macroscopic deformation gradient $\overline{\boldsymbol{F}}$ and use FEM to solve for the displacement field $\boldsymbol{u}(\boldsymbol{X})$, where $\boldsymbol{X}$ is the material point. As for the boundary condition, the solution $\boldsymbol{u}(\boldsymbol{X})$ is decomposed into a macroscopic (overall) part $\overline{\boldsymbol{u}} = (\overline{\boldsymbol{F}} - \boldsymbol{I}) \cdot \boldsymbol{X}$ and a microscopic (fluctuating) part $\boldsymbol{u}^{\star}$ such that

$$\boldsymbol{u}(\boldsymbol{X}) = (\overline{\boldsymbol{F}} - \boldsymbol{I}) \cdot \boldsymbol{X} + \boldsymbol{u}^{\star}(\boldsymbol{X}), \tag{32}$$

where the fluctuating part $\boldsymbol{u}^{\star}$ satisfies periodic boundary conditions. Then the macroscopic energy density $\overline{W}$ is calculated as a volume averaged quantity:

$$\overline{W} = \frac{1}{V} \int W, \tag{33}$$

where W is the strain energy density that depends on $\boldsymbol{u}$ and V is the RVE volume. Due to material frame indifference [43], $\overline{W}$ is a function of the macroscopic right Cauchy-Green tensor $\overline{\boldsymbol{C}} = \overline{\boldsymbol{F}}^{\top}\overline{\boldsymbol{F}}$ and that $\overline{W}(\overline{\boldsymbol{C}})$ is the nonlinear constitutive relation we want to approximate with a data-driven approach. The RVE computation is repeated with different $\overline{\boldsymbol{C}}$ so that a database $\{(\overline{\boldsymbol{C}}, \overline{W})\}_{i=1}^{N_s}$ is constructed for supervised learning. Each data point contains the feature vector $\overline{\boldsymbol{C}}$ and the scalar label $\overline{W}$. Sobol sequence [44] method is used to generate around 1000 random data points for training.

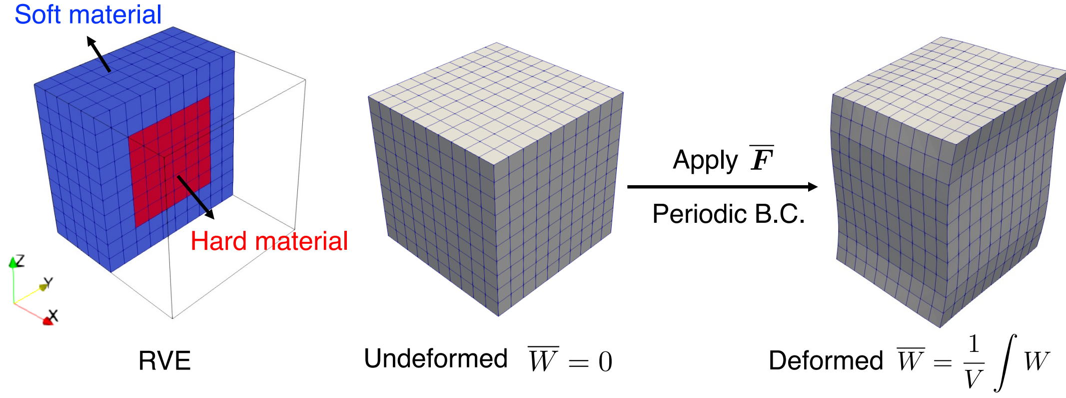

Figure 12: The RVE computation mechanism. The left panel shows half of the RVE (the other half is symmetric). The middle and right panels show the deformation of an RVE subject to a macroscopic deformation gradient $\overline{\boldsymbol{F}}$ condition and periodic boundary conditions.

4.1.2 Training, validation and testing

We train multi-layer perceptron (MLP) [45] for an approximate surrogate model such that $\overline{W}_{\text{MLP}}(\overline{\boldsymbol{C}}) \approx \overline{W}(\overline{\boldsymbol{C}})$, as shown in the middle panel of Fig. 11. The data set is split with an 8:1:1 ratio for training, model validation, and testing. Three MLPs with increasing model capability are considered: "MLP1" has 4 hidden layers and each hidden has 32 neurons; "MLP2" has 8 hidden layers and 64 layer width; "MLP3" has 16 hidden layers and 128 layer width. The activation function is set to be the hyperbolic tangent function. The scaled mean square error (SMSE) is used as the criterion to select the optimal model, which is defined as

$$\text{SMSE} = \frac{1}{N_s}\sum_{i=1}^{N_s}(\bar{y}_{\text{true}} - \bar{y}_{\text{pred}})^2 \quad \text{with} \quad \bar{y} := \frac{y - y_{\min}}{y_{\max} - y_{\min}}, \tag{34}$$

where N_s is the number of samples considered, $\bar{y}_{\text{true}}$ is the scaled true output, $\bar{y}_{\text{pred}}$ is the scaled predicted output, and $(y_{\min}, y_{\max})$ are the lower and upper bounds in the training data. We report training and validation SMSEs for the three MLPs and Fig. 13 (a). As shown, MLP2 has the best validation SMSE and is selected for deployment. The test result of MLP2 is shown in Fig. 13 (b) where the predicted macroscopic strain energy density values match the true values well.

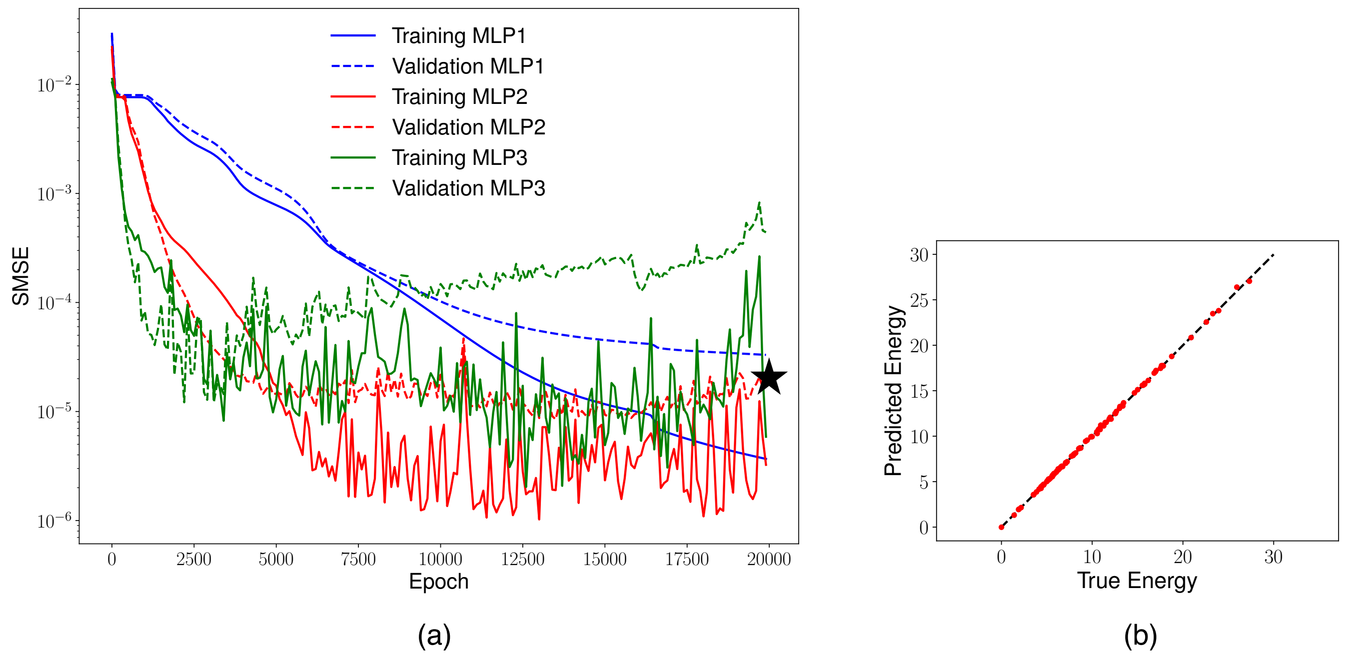

Figure 13: Training, validation, and testing of the models. In (a), training and validation errors are shown for all three MLPs. The black star shows the lowest validation error. In (b), we show the test result for the selected MLP2.

4.1.3 Macroscopic problem

The trained MLP2 represents the homogenized constitutive relationship and is deployed to solve a macroscopic problem. We consider a uni-axial tensile loading on a $10 \times 2 \times 10$ mm^3 (consisting of $10 \times 2 \times 10$ RVEs) sample. The problem setup is shown in Fig. 14, where quasi-static loading up to 1 mm (10% of the y-axis sample size) is applied.

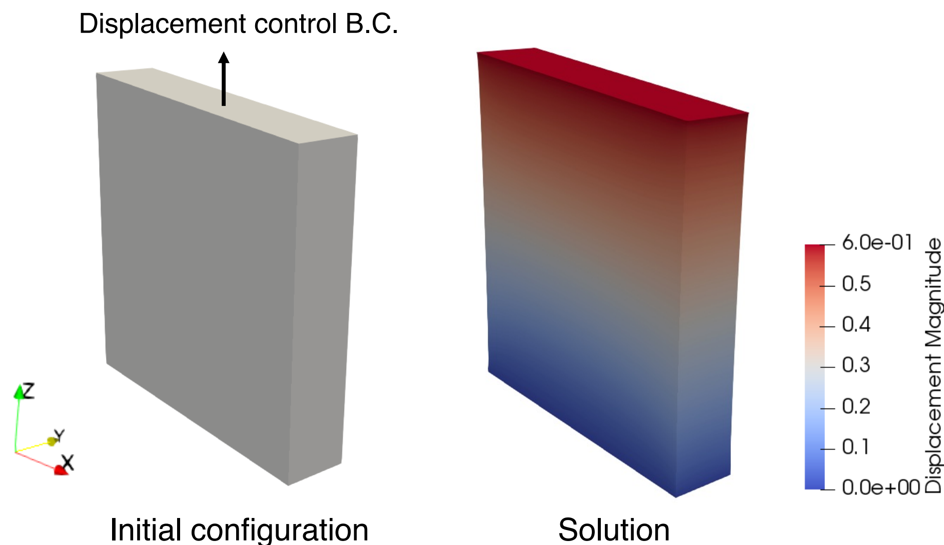

Figure 14: Problem setup. The left panel shows the boundary condition such that the bottom surface is fixed and the top surface is subject to a prescribed y-axis displacement. The right penal shows the deformed configuration.

We use both direct numerical simulation (DNS) and neural network (NN) surrogate models

to solve the problem. The total elastic energy stored in the sample as well as the tensile force applied on the top surface is plotted in Figs. 15 (a) and (b), respectively. Mechanical responses of bulk materials that are made solely by the hard and soft materials are also shown in the figure for reference. We observe good agreements between DNS and NN results, showing the effectiveness of the proposed data-driven multi-scale computational approach. DNS uses 200,000 FEM cells while the NN-based model only uses 25,000 FEM cells, being more computationally efficient.

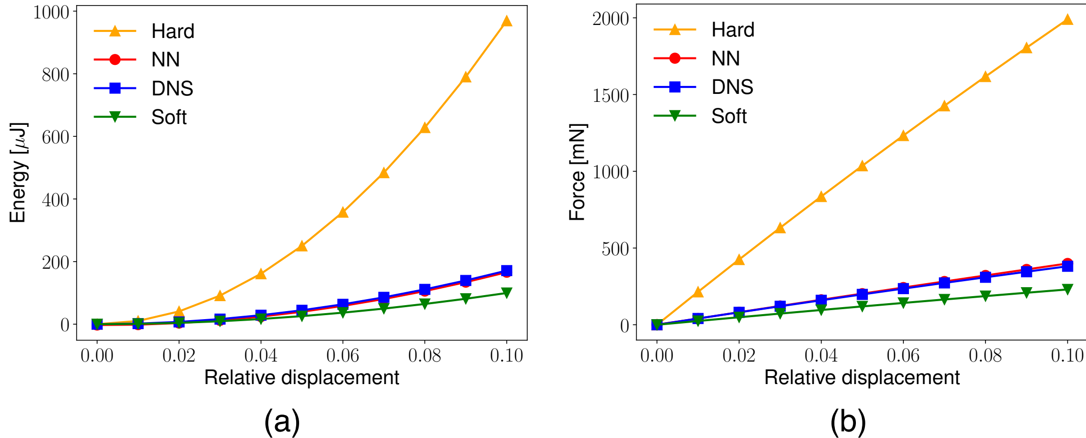

Figure 15: Comparison between DNS and NN results. In (a), the total energy is shown with respect to relative displacement up to 10%. In (b), force on the top surface is plotted.

5 Conclusions and future work

We proposed and shared with the community an open-source FEM library `JAX-FEM`, a fundamental tool that efficiently solves the forward/inverse problems and facilitates research in data-driven computational mechanics. The software can be more powerful and we list the following considerations for possible future improvement:

1. The basic FEM toolkit needs to be more complete, e.g., richer element types, support of triangular/tetrahedron mesh, etc.
2. The current linear solver is the biconjugate gradient stabilized method [46] with the simplest Jacobi preconditioner. A better linear solver with a suitable preconditioner can greatly improve the performance of `JAX-FEM`. Interfacing with powerful external linear solvers like `PETSc` [47] is in progress.
3. Inverse problems considered in this paper are all deterministic. We plan to solve Bayesian inverse problems [48] and consider uncertainty quantification in the future.
4. The largest problem we can solve with a single 48 GB memory NVIDIA GPU is around 10 million DOF, otherwise the memory is insufficient. A multi-GPU version of `JAX-FEM` is highly-desired and is our future research goal.

References

[1] David Kamensky and Yuri Bazilevs. tigar: Automating isogeometric analysis with fenics. *Computer Methods in Applied Mechanics and Engineering*, 344:477–498, 2019.

[2] Thomas JR Hughes. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[4] Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21):e2101784118, 2021.

[5] Deniz A Bezgin, Aaron B Buhendwa, and Nikolaus A Adams. Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Computer Physics Communications*, 282:108527, 2023.

[6] Tianju Xue, Sigrid Adriaenssens, and Sheng Mao. Learning the nonlinear dynamics of soft mechanical metamaterials with graph networks. *arXiv preprint arXiv:2202.13775*, 2022.

[7] Samuel Schoenholz and Ekin Dogus Cubuk. Jax md: a framework for differentiable physics. *Advances in Neural Information Processing Systems*, 33, 2020.

[8] Tianju Xue, Zhengtao Gan, Shuheng Liao, and Jian Cao. Physics-embedded graph network for accelerating phase-field simulation of microstructure evolution in additive manufacturing. *npj Computational Materials*, 8(1):1–13, 2022.

[9] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. deal. ii—a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):24–es, 2007.

[10] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[12] Andrea Vigliotti and Ferdinando Auricchio. Automatic differentiation for solid mechanics. *Archives of Computational Methods in Engineering*, 28(3):875–895, 2021.

[13] Alexander Lindsay, Roy Stogner, Derek Gaston, Daniel Schwen, Christopher Matthews, Wen Jiang, Larry K Aagesen, Robert Carlsen, Fande Kong, Andrew Slaughter, et al. Automatic differentiation in metaphysicl and its applications in moose. *Nuclear Technology*, 207(7):905–922, 2021.

[14] Mojtaba Mozaffar, Shuheng Liao, Jihoon Jeong, Tianju Xue, and Jian Cao. Differentiable simulation for material thermal response design in additive manufacturing processes. *Available at SSRN 4160375*.

[15] Tyrone Rees, H Sue Dollar, and Andrew J Wathen. Optimal solvers for pde-constrained optimization. *SIAM Journal on Scientific Computing*, 32(1):271–298, 2010.

[16] F Van Keulen, RT Haftka, and Na-Hyung Kim. Review of options for structural design sensitivity analysis. part 1: Linear systems. *Computer methods in applied mechanics and engineering*, 194(30-33):3213–3243, 2005.

[17] Ronald M Errico. What is an adjoint model? *Bulletin of the American Meteorological Society*, 78(11):2577–2592, 1997.

[18] Yang Cao, Shengtai Li, Linda Petzold, and Radu Serban. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution. *SIAM Journal on Scientific Computing*, 24(3):1076–1089, 2003.

[19] Yoshihiro Kanno. A kernel method for learning constitutive relation in data-driven computational elasticity. *Japan Journal of Industrial and Applied Mathematics*, 38(1):39–77, 2021.

[20] M Mozaffar, R Bostanabad, W Chen, K Ehmann, Jian Cao, and MA Bessa. Deep learning predicts path-dependent plasticity. *Proceedings of the National Academy of Sciences*, 116(52):26414–26420, 2019.

[21] Kailai Xu, Alexandre M Tartakovsky, Jeff Burghardt, and Eric Darve. Learning viscoelasticity models from indirect data using deep neural networks. *Computer Methods in Applied Mechanics and Engineering*, 387:114124, 2021.

[22] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

[23] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[24] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[25] Raymond W Ogden. *Non-linear elastic deformations*. Courier Corporation, 1997.

[26] Juan C Simo and Thomas JR Hughes. *Computational inelasticity*, volume 7. Springer Science & Business Media, 2006.

[27] Jan W. Gooch. *ASTM D638*, pages 51–51. Springer New York, New York, NY, 2011.

[28] John T Betts and Stephen L Campbell. Discretize then optimize. *Mathematics for industry: challenges and frontiers*, pages 140–157, 2005.

[29] Jun Liu and Zhu Wang. Non-commutative discretize-then-optimize algorithms for elliptic pde-constrained optimal control problems. *Journal of Computational and Applied Mathematics*, 362:596–613, 2019.

[30] Walter Rudin. *Principles of mathematical analysis*. McGraw-Hill Book Company, Inc., New York-Toronto-London, 1953.

[31] Kailai Xu and Eric Darve. Physics constrained learning for data-driven inverse modeling from sparse observations. *Journal of Computational Physics*, 453:110938, 2022.

[32] Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *arXiv preprint arXiv:2105.15183*, 2021.

[33] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, 16(5):1190–1208, 1995.

[34] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[35] Alexander Niewiarowski, Sigrid Adriaenssens, and Ruy Marcelo Pauletti. Adjoint optimization of pressurized membrane structures using automatic differentiation tools. *Computer Methods in Applied Mechanics and Engineering*, 372:113393, 2020.

[36] Tianju Xue and Sheng Mao. Mapped shape optimization method for the rational design of cellular mechanical metamaterials under large deformation. *International Journal for Numerical Methods in Engineering*, 123(10):2357–2380, 2022.

[37] Martin Philip Bendsoe and Ole Sigmund. *Topology optimization: theory, methods, and applications*. Springer Science & Business Media, 2003.

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

[39] Ole Sigmund and Joakim Petersson. Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Structural optimization*, 16(1):68–75, 1998.

[40] Aaditya Chandrasekhar, Saketh Sridhara, and Krishnan Suresh. Auto: a framework for automatic differentiation in topology optimization. *Structural and Multidisciplinary Optimization*, 64(6):4355–4365, 2021.

[41] Tianju Xue, Alex Beatson, Maurizio Chiaramonte, Geoffrey Roeder, Jordan T Ash, Yigit Menguc, Sigrid Adriaenssens, Ryan P Adams, and Sheng Mao. A data-driven computational scheme for the nonlinear mechanical properties of cellular mechanical metamaterials under large deformation. *Soft matter*, 16(32):7524–7534, 2020.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[43] Gerhard A Holzapfel. Nonlinear solid mechanics: a continuum approach for engineering science. *Meccanica*, 37(4):489–490, 2002.

[44] Il'ya Meerovich Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4):784–802, 1967.

[45] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[46] Henk A Van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

[47] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. Petsc users manual. 2019.

[48] Andrew M Stuart. Inverse problems: a bayesian perspective. *Acta numerica*, 19:451–559, 2010.