Structural and Multidisciplinary Optimization (2026) 69:126
https://doi.org/10.1007/s00158-026-04299-6

EDUCATIONAL PAPER

Leveraging automatic differentiation in modern machine learning frameworks for (neural) topology optimization

Suryanarayanan Manoj Sanu[1] · **Miguel A. Bessa**[2] · **Alejandro M. Aragón**[1]

Received: 10 October 2025 / Revised: 19 February 2026 / Accepted: 24 February 2026

Abstract

Automatic differentiation (AD) was introduced into topology optimization (TO) more than two decades ago to compute accurate gradients through complex computational workflows. Nevertheless, its adoption within the TO community has remained limited, largely due to the strong reliance on adjoint-based sensitivity analysis—which typically offers superior memory efficiency and runtime performance—and the practical difficulties of integrating large-scale simulations into specialized AD frameworks. The recent rise of machine learning (ML) has opened new opportunities for TO through the advanced AD capabilities of modern ML frameworks such as `JAX` and `PyTorch`. A growing body of work at the intersection of ML and TO now focuses on tightly coupling ML components with classical TO workflows. Neural TO is a prominent example, in which an untrained neural network parameterizes the material density field and optimization proceeds over the network parameters. To enable such ML–TO hybrid workflows, a deeper understanding of how AD systems operate in these frameworks is essential. This article explains the practical principles of AD in modern ML frameworks and their relation to classical adjoint-based sensitivity analysis. We present implementation strategies for wrapping essential operations—such as finite element solvers—into AD-compatible components without reimplementing them from scratch. These ideas are illustrated through two compact code examples: a classical TO pipeline with selectively AD-wrapped components and a neural TO workflow.

Keywords Topology optimization · Automatic differentiation · Neural networks · Implicit function theorem

In recent years, advances in machine learning (ML)—particularly in neural networks—have opened up new directions in topology optimization (TO). Much of the early work in this space has focused on treating TO primarily as a data generation tool for training networks (using supervised learning) to predict near-optimal structures from problem characteristics (Kallioras et al. 2020; Xue et al. 2021; Sosnovik and Oseledets 2019; Banga et al. 2018 among others). While these approaches have demonstrated some potential, they depend on expensive training datasets and often exhibit poor generalization to unseen inputs, limiting their practical applicability and robustness (Woldseth et al. 2022).

Responsible Editor: Aaditya Chandrasekhar.

✉ Alejandro M. Aragón
a.m.aragon@tudelft.nl

[1] Faculty of Mechanical Engineering, Delft University of Technology, Mekelweg 2, 2628 CD Delft, The Netherlands

[2] School of Engineering, Brown University, 184 Hope Street, Providence, RI 02912, USA

To overcome these limitations, recent efforts have increasingly focused on tighter integration of ML and TO. One line of work replaces specific TO components with pretrained ML surrogates, such as ML-based material models (Vijayakumaran et al. 2025; White et al. 2019), or approximations of finite element analysis (FEA) and associated sensitivity analysis (Qian and Ye 2021; Keshavarzzadeh et al. 2021; Lee et al. 2020; Chi et al. 2021). A second, conceptually distinct line of work is neural topology optimization (neural TO), which eliminates the need for training data altogether (Hoyer et al. 2019; Zhang et al. 2021; Chandrasekhar and Suresh 2020, 2022; Deng and To 2020; Doosti et al. 2021; Halle et al. 2021). In this unsupervised learning paradigm, a neural network reparameterizes the design space and directly outputs the density field, transforming the optimization problem into one over the network parameters. Although still under active development and sensitive to hyperparameters, these methods may offer advantages for highly non-convex objective landscapes, where traditional approaches are prone to becoming trapped in poor local optima (Both et al. 2023; Herrmann et al. 2024; Sanu et al. 2025). Neural TO shares

Published online: 04 May 2026

key characteristics with physics-based learning (Karniadakis et al. 2021), and recent work has further blurred the boundary between ML and physics by combining multiple neural networks, for example using one network for reparameterization and another to replace FEA entirely to realize fully mesh-free TO workflows (Zehnder et al. 2021a; Joglekar et al. 2023; Sun et al. 2025; Jeong et al. 2023). These approaches represent a significant shift away from viewing ML as a post-processing tool and toward embedding learning directly within the optimization loop.

The accelerated adoption of ML in TO and beyond has been driven not only by advances in hardware, such as GPUs, but also by the emergence of high-level software frameworks with native support for automatic (algorithmic) differentiation (AD). AD is a key enabler of this paradigm shift: by requiring only the forward computation, it removes the need for manually deriving and maintaining gradients. Importantly, AD can propagate derivatives through complex program structures, including iterative solvers and conditional statements (Griewank and Faure 2002). Modern ML libraries such as `PyTorch` (Paszke et al. 2019) and `JAX` (Bradbury et al. 2018) provide scalable and efficient AD, supporting the training of models with billions of parameters. Components implemented within these frameworks are inherently differentiable and can be composed into larger differentiable pipelines, with sensitivities propagated automatically via AD. Researchers can thus reuse existing models and algorithmic building blocks across applications, greatly accelerating prototyping, experimentation, and method development.

In contrast, conventional TO workflows rely on manually derived and implemented sensitivities. This process is labor-intensive, error-prone, and brittle with respect to changes in the problem formulation: even minor modifications—such as introducing a new filter or changing the objective—often require re-derivation and careful verification of sensitivity expressions. Given that gradient-based optimization remains the workhorse of large-scale TO (Sigmund 2011), modern AD frameworks offer significant potential for advancing TO practice. AD is not new to the TO community. Early work explored its use for linear systems (Van Keulen et al. 2005) and adjoint-based fluid problems (Dilgen et al. 2018; Rokicki 2016). However, these approaches relied on specialized low-level software, limiting their accessibility and adoption. In contrast, modern ML frameworks interface directly with high-level languages such as `Python`, enabling users to access AD with minimal overhead. Even so, AD has typically been used in TO only in a limited manner—for example, to compute selected partial derivatives within classical adjoint analyses—rather than as an end-to-end differentiation tool. More recently, full-scale TO implementations interfacing with `Python` and leveraging `JAX` (Chandrasekhar et al. 2021; Xue et al. 2023; Wu 2023) or `FEniCSx` (Jia et al. 2024) as backends have begun to emerge. Unlike many ML applications, however, TO workflows invariably involve physics-based simulation codes that are often implemented in mature, well-tested libraries and are impractical to rewrite entirely in the language of ML frameworks. As a result, adopting AD in TO frequently requires selective differentiation and the integration of black-box operations into differentiable pipelines—tasks that demand a deeper understanding of AD internals than most practitioners currently possess. Moreover, existing studies largely focus on showcasing specific implementations, offering limited guidance for researchers who wish to adapt these ideas to their own workflows.

This article does not propose a new optimization framework. Instead, it is an educational contribution that introduces the inner workings of AD in the context of modern ML libraries, with examples tailored specifically to TO. We explain key abstractions underlying AD—how computational graphs are constructed and traversed, the relationship between Jacobian–vector products (JVPs) and vector–Jacobian products (VJPs) and their registration as graph nodes, and how techniques such as unrolling and the implicit function theorem (IFT) enable differentiation through iterative algorithms (Blondel and Roulet 2024), including solvers used in FEA. While these concepts are foundational in modern ML, they are rarely articulated explicitly in the TO literature. Whereas prior work has demonstrated the flexibility of `JAX`'s AD by rewriting entire TO pipelines within the framework (Chandrasekhar and Suresh 2022; Xue et al. 2023), our focus is different: we show how existing TO components—including black-box solvers—can be embedded into broader AD-enabled workflows without full reimplementation. Following the tradition of educational implementations, we illustrate these ideas using a minimal (neural) TO example in both `JAX` and `PyTorch`, highlighting their portability across ML frameworks. Our goal is to lower the barrier to adopting AD in TO by equipping practitioners with the conceptual and practical tools needed to construct differentiable workflows, even when no ML components are present.

1 Fundamentals of automatic differentiation

At a high level, AD computes *exact* derivatives of computer programs by decomposing them into a sequence of elementary operations (Blondel and Roulet 2024; Griewank 2000). Provided that the derivatives of these primitive operations are known, the chain rule can be systematically applied by the AD system to propagate derivative information through the computation. What is considered "elementary" or "primitive" is determined by the system developers and, in many frameworks, can also be extended by the user. Even rela-

tively large routines—such as singular value decompositions or linear system solves—may be treated as primitive operations, provided that their corresponding derivative rules are defined. In practice, these derivative rules are stored in an internal registry and are automatically invoked whenever the corresponding operation is encountered during differentiation.

Other common approaches to computing derivatives include symbolic differentiation (SD) and finite differencing (FD; Martins and Ning 2022). FD is the simplest to implement, as it requires only function evaluations and can therefore be applied to any black-box code, independent of its internal structure. In essence, the input is perturbed by a small value, and the ratio of the change in the function output to this perturbation provides an estimate of the gradient. However, FD suffers from two main sources of error: truncation error (since the perturbation is finite, the derivative is only approximate) and round-off error (arising from subtracting nearly equal quantities, a well-known pitfall in numerical computation).

SD, as employed by systems such as Mathematica (Wolfram Research, Inc. 2025), instead manipulates mathematical expressions using specialized data structures. Its main advantage is that SD yields exact derivative expressions (as if derived analytically), which can sometimes provide insight into the underlying problem. However, symbolic methods often suffer from *expression swell*: a phenomenon in which derivative expressions grow rapidly in size. These expressions quickly become cumbersome and rarely provide useful insights. For example, repeated application of the product rule can cause the size of expressions to grow exponentially with the length of the computation sequence (Baydin et al. 2017), making symbolic differentiation impractical for larger workflows.

AD differs fundamentally from both approaches. Unlike SD, which constructs explicit algebraic expressions for derivatives, AD evaluates derivatives numerically at runtime. Concretely, for a function $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$, AD returns the value of the derivative at a specific input, i.e., $\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}}\Big|_{\boldsymbol{x}=\boldsymbol{x}_0}$, rather than an expression for $\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}}$ valid for all $\boldsymbol{x}$. Unlike FD, which estimates derivatives using perturbations of fixed size (a choice the user must make a priori), AD computes derivatives exactly up to machine precision, thereby avoiding truncation error. AD achieves this by applying differentiation rules for each elementary operation (e.g., product rule or chain rule) and propagating these through the computational graph. In this sense, AD borrows the symbolic rules of calculus, but instead of producing closed-form expressions, it executes them numerically on the actual data flowing through the program. This allows AD to evaluate derivatives efficiently even in complex programs with loops, branches, and recursion, making it particularly well suited for scientific computing.

1.1 AD at the graph level

AD implementations fall broadly into two categories: source code transformation and operator overloading. Most contemporary ML frameworks adopt the latter approach, and thus our discussion focuses exclusively on this method. We do not dive into these implementation choices here and instead refer the reader to Nørgaard et al. (2017) or Margossian (2019) for a comprehensive discussion.

A key consequence of using operator overloading is the introduction of specialized numerical data containers—`JAX`'s *Array* and `PyTorch`'s *Tensor*—together with functions overloaded to work with them (e.g., `jax.numpy.sin` or `torch.sin`). These data structures are designed to carry, at the same time, the numerical values used in computation together with the derivative signals required for AD. Objects of this type are "traced", i.e., when a sequence of operations is performed on them, each operation is intercepted and recorded in order. This process builds a computational graph, a directed acyclic graph (DAG) in which nodes represent individual elementary operations (e.g., addition, multiplication), and edges encode the flow of data and dependencies between these operations (see Gandarillas et al. (2024) for a comprehensive discussion on computational graphs). In effect, the computational graph provides a complete, structured representation of how the output of a program is computed from its inputs. Unlike earlier ML frameworks such as `TensorFlow` 1.0 (Abadi et al. 2015), modern frameworks like `JAX`, `PyTorch`, and `TensorFlow` 2.0 construct graphs dynamically, building them on the fly as computations are executed. This allows regular Python control flow (e.g., loops and conditionals) to be used directly, leading to more flexible and faster development.

1.1.1 Computational graphs

The computational graph forms the backbone of derivative computation in AD, and an example relevant to TO[1] is shown in Fig. 1. The top figure shows the simplified program execution (forward pass) for a density-based TO problem with SIMP interpolation. The first operation is the application of the density filter (`DensityFilter`), on the input decision variables $\boldsymbol{\rho}$. The output is again a traced variable, the filtered densities $\tilde{\boldsymbol{\rho}}$. This is represented as the first node (orange rectangles), where arrows indicate the direction of program execution.

The graph in Fig. 1 should be viewed as an illustrative example. In practice, AD systems typically construct graphs with much simpler operations as nodes. Higher-level routines—such as the density filter—are themselves compo-

[1] Additional workflows integrating neural networks and illustrating more complex AD usage are shown in "Appendix A".

```
rho_tilde = DensityFilter(rho, rmin)
global_stiffness_K = GetK(elem_stiff, rho_tilde, simp_p, E0, Emin)
displacement_u = Solve(global_stiffness_K, force_f)
objective_phi0 = Objective(displacement_u, force_f)
volconstr_g1 = VolumeConstraint(xTilde)
statecnstr_g0 = StateConstraint(displacement_u)
```

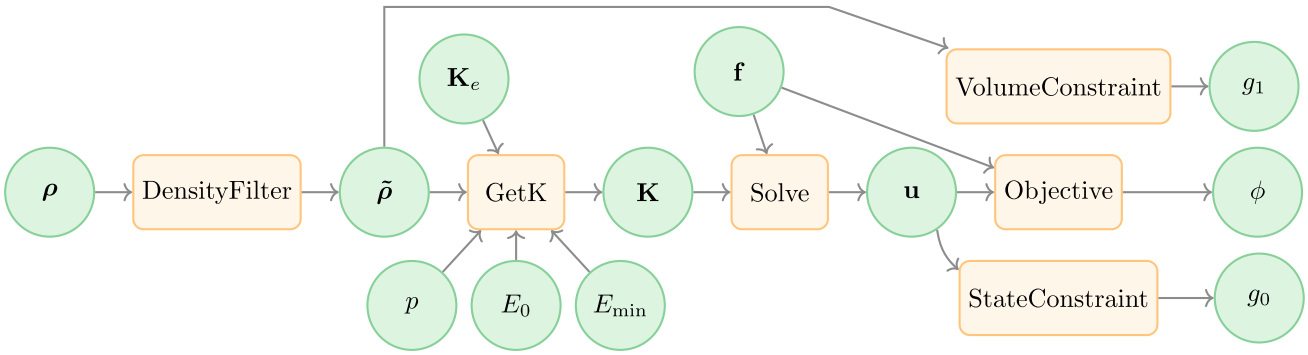

Fig. 1 A typical sequence of computations in a topology optimization pipeline (top): density filtering, material interpolation, stiffness matrix assembly, equilibrium solution, and evaluation of objective and constraints. The corresponding computational graph (bottom) represents each operation as an orange rectangle (node), with its input and output data shown as green circles. Arrows indicate the direction of execution. For clarity, higher-level operations (e.g., filtering, linear solve) are shown instead of the low-level primitives (e.g., matrix multiplication, addition) that would normally appear in an automatic differentiation graph. (Color figure online)

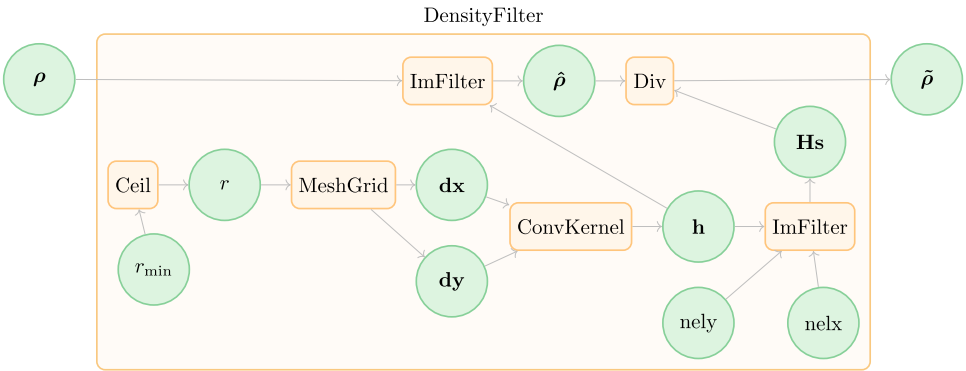

Fig. 2 Expansion of the density filter using one representative implementation (Ferrari and Sigmund 2020) to illustrate the arbitrariness of graph nodes. While the `DensityFilter` appears as a single node in Fig. 1, it can equivalently be represented as a sequence of simpler operations with the same inputs and outputs. Here, $\boldsymbol{\rho}$ denotes the input density field and $\tilde{\boldsymbol{\rho}}$ the filtered density. This emphasizes that nodes in an AD computational graph are abstractions, and only their input–output behavior matters

sitions of such primitives. For instance, as shown in Fig. 2, the density filter implementation in Ferrari and Sigmund (2020) can be expanded into a sequence of lower-level operations, each of which could be a node in the computational graph. Thus, the notion of what constitutes a "node" is flexible. From the AD system's perspective, a node is simply an opaque black-box function: its internal workings are not inspected (or traced). Instead, AD only requires that (1) the function can be evaluated during forward execution given inputs, and (2) its differentiation rule is registered. Once such a rule is in the system's registry, the node can participate in the computational graph just like any other primitive—whether it represents a basic operation like addition or even an entire nonlinear solve.

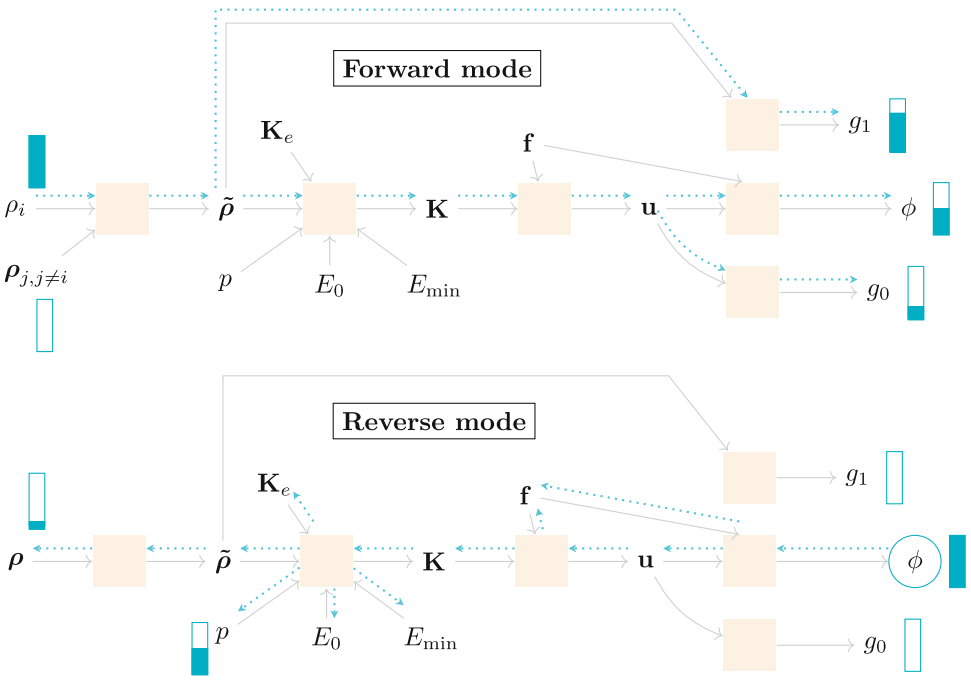

Fig. 3 TO computational graph from Fig. 1 illustrating forward and reverse mode AD. In forward mode, the derivative signal (blue dotted lines) propagates alongside the data flow (gray continuous lines) during execution; in reverse mode, it travels backward from outputs to inputs. In both cases, one endpoint—an input for forward mode or an output for reverse mode—must be seeded with a unit signal (indicated by a filled bar). Note that the filled bar is attached only to a scalar quantity (either a single design variable component ρ_i or the objective ϕ). As the signal flows through the graph, contributions accumulate, and the value collected at the opposite endpoint represents the derivative with respect to the seeded variable. (Color figure online)

1.1.2 Forward and reverse modes

The computational graph can be traversed in two directions to compute derivatives: forward mode, which follows the original direction of execution, and reverse mode, which proceeds from outputs back to inputs (Fig. 3). In forward mode, one begins by selecting an input variable whose derivative information is of interest. Here, ρ_i is chosen as the input, depicted by a filled bar attached to it. Seeding refers to the initialization of derivative information at a chosen node. Note that we cannot seed the whole vector in forward mode, since seeding has to be done separately for each scalar.[2] We could have equally chosen the SIMP penalty p or the material properties E_0 or $E_{\min}$, as they are also scalar inputs to the graph. The derivative signal (as blue dotted lines) then propagates through the graph, flowing from this data structure to every output along the computational pathway. Ultimately, the signal accumulates at the outputs g_0, ϕ, and g_1, thereby providing the sensitivities of all outputs with respect to a single input in a single forward pass. Reverse mode, by contrast, inverts this procedure. Rather than seeding at an input, the process begins with the choice of an output variable, in this case ϕ. The derivative signal is then traced backwards against the execution flow of the program, redistributing through each node until it reaches all the inputs.

In these settings, it is more natural to speak of *accumulating* derivatives, since the contribution to the derivative signal at a given node may originate from multiple downstream operations. This is clearly visible at $\mathbf{f}$, where the total derivative signal combines contributions from two different branches of the graph. To conceptualize how this propagation of derivative information occurs, it is useful to imagine the computational graph as a system of pipes with input funnels and output faucets. Each edge in the graph acts as a hollow pipe, and each node serves as a junction where multiple pipes meet. One may picture a physical fluid moving through this network. At each junction, incoming flows are redistributed and potentially scaled along the outgoing pipes. However, unlike physical fluids, there is no conservation principle here: the outgoing flows may be amplified or diminished. In forward mode, a unit quantity of fluid is introduced at an input funnel, and as it flows downstream it accumulates at all the output faucets. The amounts collected at each faucet represent the sensitivities of the corresponding output with respect to the chosen input. This analogy clarifies why forward mode is best suited to problems with relatively few inputs and many outputs: each forward pass delivers the full set of derivatives for one input.

In contrast, reverse mode inverts the flow direction. After the function has executed once and the computational graph is available, a unit quantity of fluid is injected at a chosen output faucet and then *pulled* upstream, as if pumps at the input funnels were drawing the fluid backward through the pipes. The redistributed flows eventually reach all input funnels, yielding the sensitivities of the output with respect to all inputs. This property makes reverse mode particularly effective for optimization problems with a scalar objective and a large number of design variables.

[2] In forward mode, it is also possible to seed an entire input vector to compute directional derivatives; this will be discussed later.

Thus, there are two essential differences between the two modes. The first concerns efficiency: forward mode is preferable when the number of inputs is smaller than the number of outputs, while reverse mode is preferable in the opposite case. The second difference lies in the timing of derivative evaluation. In forward mode, the primary computation and derivative accumulation proceed simultaneously, as if the piping network is being extended while the fluid flows. In reverse mode, by contrast, the forward computation must be completed in its entirety—producing the outputs and constructing the graph—before the backward sweep can take place.

This decoupling introduces a distinctive memory-computation trade-off in reverse mode: intermediate values required for the backward pass must either be stored during the forward execution or recomputed later. Consider, for example, the `DensityFilter` operation, which can be written as the linear mapping $\boldsymbol{f}(\boldsymbol{\rho}) = \mathbf{H}\boldsymbol{\rho} = \tilde{\boldsymbol{\rho}}$. In forward mode, the matrix $\mathbf{H}$ is still in memory when the derivative $\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{\rho}} = \mathbf{H}$ is needed, making the calculation straightforward. In reverse mode, however, derivative evaluation occurs only after the *entire* forward execution has finished, and $\mathbf{H}$ may no longer be available unless it was explicitly stored. Modern AD frameworks automate much of this bookkeeping by deciding which intermediate values to retain or recompute, usually without user intervention. If, however, an operation such as `DensityFilter` is abstracted as a single black-box node rather than decomposed into primitive operations, then the user must explicitly specify what quantities should be stored in the derivative rule defined for that node.

1.2 AD at the node level

We now zoom into the behavior of AD at the level of individual nodes in the computational graph. Each node represents an elementary operation with well-defined inputs and outputs. Mathematically, each operation can be abstracted as a function, where the inputs (as well as outputs) have been concatenated together to form a vector. $\boldsymbol{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$. AD computes how to propagate derivative information across this function—from inputs ($\boldsymbol{x}$) to outputs ($\boldsymbol{y}$) in forward mode, and vice versa in reverse mode—using only its local derivative: the Jacobian matrix, defined as:

$$\boldsymbol{J}_f(\boldsymbol{x}) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla f_1^\top \\ \nabla f_2^\top \\ \vdots \\ \nabla f_m^\top \end{bmatrix} \in \mathbb{R}^{m \times n}, \tag{1}$$

where n and m are the input and output dimensionalities respectively and $f_i = y_i$ is the ith output. Each column of the Jacobian represents how a specific input influences all outputs while each row describes how a specific output depends on all inputs, and thus corresponds to the gradient of that output (∇f_i). AD exploits the fact that the computational graph is the result of a composition of functions, which when differentiated using the chain rule, results in a product of Jacobian matrices. For instance, if $\boldsymbol{h} = \boldsymbol{f} \circ \boldsymbol{g}$, then the Jacobian of $\boldsymbol{h}$ evaluated at a given point $\boldsymbol{x}$ is given by $\boldsymbol{J}_{\boldsymbol{h}}(\boldsymbol{x}) = \boldsymbol{J}_f(\boldsymbol{g}(\boldsymbol{x})) \cdot \boldsymbol{J}_{\boldsymbol{g}}(\boldsymbol{x})$.

Algorithm 1 Registering the JVP rule of a node with an AD framework.

```
def f(x_1, x_2):
    """A black-box function mapping (
       x_1, x_2) -> (y_1, y_2, y_3).
       """
    # Perform computations
    ...
    return y_1, y_2, y_3

def f_jvp_rule(input_primals,
    input_tangents):
    """
    Forward mode AD rule for f.
    Computes both the output primals
       and tangents.
    """
    # Unpack primals and tangents
    x_1, x_2 = input_primals
    x_1_dot, x_2_dot = input_tangents

    # Compute output primals
    y_1, y_2, y_3 = f(x_1, x_2)
    output_primals = (y_1, y_2, y_3)

    # Compute output tangents
    # Jf here is the Jacobian evaluated
       at (x_1, x_2)
    y_1_dot, y_2_dot, y_3_dot = Jf @ [
       x_1_dot, x_2_dot]
    output_tangents = (y_1_dot, y_2_dot
       , y_3_dot)

    return output_primals,
       output_tangents

# Register JVP rule with ML framework
register_jvp(func=f, jvp_rule=
    f_jvp_rule)
```

In practice, AD systems never construct the full Jacobian matrix explicitly, as this is often memory intensive. For example, consider activation functions in NNs, which are typically applied element-wise to a vector. Although each scalar activation function ϕ maps $\mathbb{R} \to \mathbb{R}$, applying it to a vector $\boldsymbol{x} \in \mathbb{R}^n$ yields a vector-valued function $\boldsymbol{f}(\boldsymbol{x}) = [\phi(x_1), \phi(x_2), \ldots, \phi(x_n)]^\top$, i.e., $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$. In this case, the Jacobian is a diagonal matrix of size $n \times n$, where each diagonal entry is simply $\frac{\mathrm{d}y_i}{\mathrm{d}x_i}$, where $y_i = \phi(x_i)$. Mate-

rializing such large sparse matrices is inefficient. Instead, AD frameworks rely on two fundamental matrix-free linear operators:

- the Jacobian–vector product (JVP) for forward mode, and
- the vector–Jacobian product (VJP) for reverse mode,

each of which maps a vector to another vector without materializing the full Jacobian. As a result, the derivative rule for each individual node is specified using these operators.

1.2.1 Specifying JVP/VJP rules

The forward mode only requires specifying one additional function, as shown in Algorithm 1. When the function `f` is executed normally, the inputs `(x_1, x_2)` are used to compute the outputs `(y_1, y_2, y_3)`. In AD terminology, these primary values are called *primals*.

When derivatives are requested, the AD framework checks whether a corresponding JVP rule (`f_jvp_rule`) is attached to the operation. If no rule exists, the framework traces into the function directly. If a rule is available, however, that rule is executed. Unlike the primal computation, the JVP rule receives both the primal inputs and their associated tangent values `input_tangents`, which carry derivative information accumulated from prior computations. Conceptually, the signal entering a node is the tuple $(\boldsymbol{x}, \dot{\boldsymbol{x}})$, where $\boldsymbol{x}$ are the primals and $\dot{\boldsymbol{x}} \in \mathbb{R}^n$ are the tangents. The node then performs two coupled computations. First, it performs primal evaluation, i.e., the usual function execution $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$. Second, it propagates the derivative information by computing the output tangents, $\dot{\boldsymbol{y}} = \boldsymbol{J}_f \cdot \dot{\boldsymbol{x}}$, where $\boldsymbol{J}_f$ is the Jacobian of the function with respect to its inputs.[3] This second computation is precisely what defines the JVP rule: it specifies how perturbations to the inputs should be pushed through the node. Among modern machine learning frameworks, `JAX` provides full support for forward mode AD, whereas `PyTorch` primarily focuses on reverse mode and, at the time of writing, offers only limited forward mode functionality.

[3] If the function has multiple inputs $\boldsymbol{f}(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots)$ rather than a single concatenated vector, the forward mode update is

$$\dot{\boldsymbol{y}} = \sum_i \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_i} \dot{\boldsymbol{x}}_i,$$

where each $\boldsymbol{x}_i$ may itself be a vector and $\dot{\boldsymbol{x}}_i$ denotes the corresponding tangents.

Algorithm 2 Registering the VJP rule of a node for reverse mode AD.

```
def f(x_1, x_2):
    """ A black-box function."""
    # Perform computation
    ...
    return y_1, y_2, y_3

def f_forward(input_primals):
    """Forward pass during reverse mode
       ."""
    x_1, x_2 = input_primals
    output_primals = y_1, y_2, y_3 = f(
       x_1, x_2)
    to_store = (x_1, x_2, x_1*x_2**2,
       ...)  # Values needed for vJp
    return down_primals, to_store

def f_vjp_rule(stored_stuff,
   output_cotangents):
    """Derivative rule for function."""
    x_1, x_2, x_1*x_2**2, ... =
       stored_stuff
    y_1_bar, y_2_bar, y_3_bar =
       down_cotangents
    # We need to propagate y_bars to
       get x_1_bar and x_2_bar
    x_bar = Jf**T * output_cotangents #
        The vector-jacobian product
    input_cotangents = x_bar = (x_1_bar
       , x_2_bar)
    return input_cotangents

# Register the vjp rule in the
   framework
register_vjp(func=f,
             func_forward=f_forward,
             func_backward=f_vjp_rule)
```

In contrast to forward mode, reverse mode requires defining two additional functions (Algorithm 2). The first (`f_forward`) is nearly identical to `f` but allows saving intermediate variables (`to_store`) in memory. This separation is necessary since reverse mode decouples the primal computation from the derivative computation. These stored values are retrieved when executing the second function (`f_vjp_rule`), which propagates derivatives—referred to as *cotangents* or *adjoints*—backward from outputs to inputs. The downstream cotangents $\bar{\boldsymbol{y}} \in \mathbb{R}^m$ are propagated to each of the inputs using $\bar{\boldsymbol{x}} \in \mathbb{R}^n = \boldsymbol{J}_f^\top \cdot \bar{\boldsymbol{y}}$, where this transpose operation defines the VJP rule.

1.3 Treating iterative algorithms as nodes for AD

In many scientific and engineering workflows, iterative algorithms are unavoidable, as they provide practical means of numerically solving nonlinear or large-scale systems. These methods typically generate a sequence of updates of

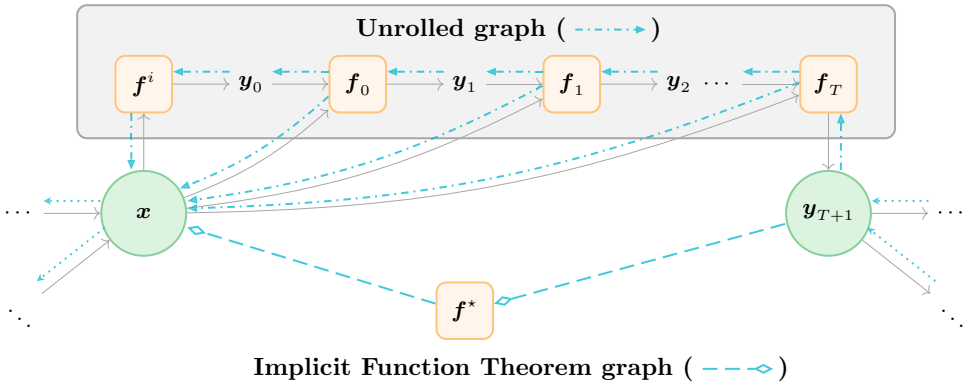

Fig. 4 Computational graph of an iterative algorithm (e.g., root-finding or optimization) used to solve for $\boldsymbol{y}$ given parameters $\boldsymbol{x}$. When implemented in an AD framework, the loop is typically *unrolled* (top, with the set of involved operations as rectangles), and the derivative signal (blue dotted/dashed lines) is propagated through each iteration. However, by applying the Implicit Function Theorem, this entire iterative process can be treated as a single node $\boldsymbol{f}^\star$, eliminating the need to differentiate through the entire computation. (Color figure online)

the form $\boldsymbol{y}_t = \boldsymbol{f}_{t-1}(\boldsymbol{y}_{t-1})$, so that after T steps (or upon meeting a convergence criterion), the approximate solution $\boldsymbol{y}_T \approx \boldsymbol{y}^\star$ is obtained. For example, the Newton–Raphson method finds the root $\boldsymbol{y}^\star$ of an equation $\boldsymbol{F}(\boldsymbol{y}) = \boldsymbol{0}$ by iterating $\boldsymbol{y}_t = \boldsymbol{y}_{t-1} - \frac{\boldsymbol{F}(\boldsymbol{y}_{t-1})}{\boldsymbol{F}'(\boldsymbol{y}_{t-1})}$, where $\boldsymbol{F}'$ denotes the derivative of $\boldsymbol{F}$. Well-tested software libraries (such as PETSc, Balay et al. 1997) provide efficient and robust implementations of such iterative solvers.

In gradient-based design workflows such as TO, however, a new challenge arises: one must differentiate through these iterative routines. The update function then depends not only on the current iterate but also on a set of fixed problem parameters, i.e., $\boldsymbol{y}_t = \boldsymbol{f}_{t-1}(\boldsymbol{y}_{t-1}, \boldsymbol{x})$, where $\boldsymbol{x}$ denotes system parameters. The central goal is to compute sensitivities of the converged solution with respect to these parameters $\left(\frac{\mathrm{d}\boldsymbol{y}^\star}{\mathrm{d}\boldsymbol{x}}\right)$, which quantify how the solution shifts when the system itself is perturbed. By leveraging JVPs or VJPs of the operations inside the iterative solver, the AD frameworks can compute this derivative. Figure 4 illustrates reverse mode AD applied to an iterative scheme, where each node (rectangle) corresponds to an update of the current estimate. During the backward pass, a portion of the derivative signal propagates to the previous iterate, while another portion accumulates at $\boldsymbol{x}$. This procedure, known as *unrolling*, transforms the iterative loop into a deep feed-forward computational graph.[4] Despite its conceptual clarity, unrolling can be memory inefficient and numerically unstable (Scieur et al. 2022), especially over large number of steps.

To address these concerns, it is often preferable to represent the entire iterative algorithm as a single node within the AD graph. This approach avoids the inefficiencies of unrolling while leveraging existing, well-tested solvers without requiring their reimplementation inside the AD framework. A key enabler of this strategy is the distinction between the *procedure* used to obtain a solution and the *solution* itself. Many iterative methods can be formulated as fixed-point problems, where the solution is defined by a stationarity condition of the form $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{x})$, which holds only at convergence. In fact, the update rules employed by common solvers are often derived directly from such stationarity conditions. For example, in unconstrained optimization of a scalar function $f(y)$, the solution is characterized by the condition $\nabla f = 0$ (Nocedal and Wright 2006). While gradient-based optimizers may employ sophisticated update schemes to search for such a point, the defining property of the solution lies solely in satisfying this condition, not in the specific trajectory taken by the algorithm.

In the previous sections, we discussed how *explicit* functions of the form $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$, with well-defined input–output behavior, can be inserted into AD graphs as nodes. In contrast, stationarity conditions are examples of *implicit* functions, where the output $\boldsymbol{y}$ is defined not by a direct mapping from $\boldsymbol{x}$, but as the solution of a relation involving both variables. In general, such functions are characterized by a relation of the form: $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{0}$, where $\boldsymbol{x} \in \mathbb{R}^n$ is the input, $\boldsymbol{y} \in \mathbb{R}^m$ the output, and the function $\boldsymbol{F} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ encodes the system to be solved.

A familiar example is the unit circle in 2D, described by the equation $F(x, y) = x^2 + y^2 - 1 = 0$, adopted from Blondel and Roulet (2024). This equation cannot be written as an explicit function $y = f(x)$ over the full domain, since for every $x \in (-1, 1)$ there are two corresponding values of y, i.e., $y = \pm\sqrt{1 - x^2}$. Hence, this violates the mathematical definition of a function—in the strict sense—as a single-valued mapping. However, if we restrict the domain locally—e.g., to a quarter circle $x \geq 0$, $y \geq 0$, then a single-

[4] Loops are, by definition, not acyclic and must therefore be "straightened out" to form a valid computational graph.

valued explicit function exists. This illustrates how implicit relationships can be locally re-expressed as functions—a key idea underlying the *Implicit Function Theorem* (IFT; Krantz and Parks 2013), which guarantees the existence of such locally defined functions under mild conditions and provides a means to differentiate them.

Suppose we are given a root $(\boldsymbol{x}_0, \boldsymbol{y}_0)$ of the implicit function $\boldsymbol{F}$, i.e, $\boldsymbol{F}(\boldsymbol{x}_0, \boldsymbol{y}_0) = \boldsymbol{0}$. Then IFT states that there exists an explicit function $\boldsymbol{f}^\star$ with $\boldsymbol{y}_0 = \boldsymbol{f}^\star(\boldsymbol{x_0})$, defined in the neighborhood of the root. Thus, $\boldsymbol{F}$ can be made explicit locally as:

$$\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{F}\left(\boldsymbol{x}, \boldsymbol{f}^\star(\boldsymbol{x})\right) = \boldsymbol{0}. \tag{2}$$

We emphasize that the function is only guaranteed to exist if $\boldsymbol{F}$ is continuously differentiable in the neighborhood of $(\boldsymbol{x}_0, \boldsymbol{y}_0)$, and its partial derivative with respect to the second argument (i.e., $\boldsymbol{y}$), denoted by $\partial_2 \boldsymbol{F} \in \mathbb{R}^{m\times m}$, is invertible. Under these mild conditions, we can differentiate both sides of the identity with respect to the input variable $\boldsymbol{x}$, and apply the chain rule to get the input–output Jacobian as:

$$\begin{aligned} \frac{\mathrm{d}\boldsymbol{F}}{\mathrm{d}\boldsymbol{x}} &= \partial_1 \boldsymbol{F} + \partial_2 \boldsymbol{F} \cdot \boldsymbol{J}_{\boldsymbol{f}^\star} = \boldsymbol{0}, \\ &\Rightarrow \boldsymbol{J}_{\boldsymbol{f}^\star} = -\left(\partial_2 \boldsymbol{F}\right)^{-1} \partial_1 \boldsymbol{F}. \end{aligned}$$

For tangents $\dot{\boldsymbol{x}} \in \mathbb{R}^n$ in forward mode and cotangents $\bar{\boldsymbol{y}} \in \mathbb{R}^m$ in reverse mode, we can express the vector product rules as[5]:

$$\mathrm{JVP}_{\boldsymbol{F}} = -\left(\partial_2 \boldsymbol{F}\right)^{-1} \partial_1 \boldsymbol{F}\, \dot{\boldsymbol{x}}, \tag{3}$$

$$\mathrm{VJP}_{\boldsymbol{F}} = -\left(\partial_1 \boldsymbol{F}\right)^{\top} \boldsymbol{\lambda}, \quad \text{where } \boldsymbol{\lambda} = \left(\partial_2 \boldsymbol{F}\right)^{-\top} \bar{\boldsymbol{y}}. \tag{4}$$

Since this idea is general and can be applied in a largely mechanical fashion, Blondel et al. (2022) developed a `JAX`-based package that automates this process, requiring only a specification of the function $\boldsymbol{F}$ that defines the stationarity condition.

1.4 Constructing AD nodes using the IFT

To connect the theoretical insights from IFT to practical algorithms, we demonstrate how to wrap two iterative procedures commonly encountered in TO: (i) the bisection algorithm, a simple scalar root-finding method, and (ii) a generic solver for solving systems of linear equations. In both cases, we provide concrete implementations of custom vector–Jacobian product (VJP) rules following Eq. (4). We focus exclusively on VJPs, since reverse mode differentiation is computationally more efficient for TO problems and, when combined with the IFT, avoids the memory scaling issues typically associated with unrolling-based approaches. This makes the IFT–VJP combination particularly well suited for large-scale TO applications.

All code examples are written in `JAX`. The functional programming paradigm enforced by `JAX` closely mirrors the underlying mathematical structure, which makes the relationship between the theory and the implementation particularly transparent. Equivalent implementations in `PyTorch` are provided in the accompanying codebase. All examples for constructing VJP rules from IFT follow the same conceptual template:

(1) Implement a call to a black-box algorithm, typically provided by an external library. Such algorithms are treated as black boxes because AD cannot (and should not) trace their internal operations. Consequently, the function is treated as a primitive node in the computational graph, and the user is required to explicitly supply both the forward and backward passes via the custom VJP interface.
(2) Define the associated stationarity function $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ denotes the parameters of interest and $\boldsymbol{y}$ the solution variables. The function $\boldsymbol{F}$ only evaluates the stationarity condition and does not perform any iterative solve.
(3) In the forward pass, the black-box algorithm computes $\boldsymbol{y} = \boldsymbol{y}^\star$ such that $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y}^\star) = \boldsymbol{0}$, while storing all quantities required for the backward pass.
(4) In the backward pass, solve the adjoint system $\left(\partial_2 \boldsymbol{F}\right)^{\top} \boldsymbol{\lambda} = \bar{\boldsymbol{y}}$, where $\bar{\boldsymbol{y}}$ denotes the cotangents at the output.
(5) Compute the VJP with respect to the parameters as $-\left(\partial_1 \boldsymbol{F}\right)^{\top} \boldsymbol{\lambda}$. The partial derivatives of $\boldsymbol{F}$ can be obtained using AD, provided that $\boldsymbol{F}$ is implemented using AD-compatible primitives.

1.4.1 Bisection algorithm

We begin with the simplest case of wrapping the bisection algorithm. This case makes the computations simpler since the stationarity function is scalar valued. Given an interval containing the root (i.e., the function changes sign across the interval), bisection iteratively selects the midpoint, evaluates the function, and shortens the bounding interval until the root is located with sufficient precision (Code 1). This procedure is useful in TO—for instance, in the volume-preserving filter (Xu et al. 2009) or when constraining neural network outputs (Hoyer et al. 2019). In both cases, a projection function (sigmoidal or hyperbolic tangent) is applied, and the task reduces to finding the scalar parameter $\eta \in \mathbb{R}$ such that the

[5] Note that the VJP is formed as $\boldsymbol{J}_{\boldsymbol{f}^\star}^{\top} \bar{\boldsymbol{y}}$ and here we use the property of matrices that $(\mathbf{AB})^{\top} = \mathbf{B}^{\top}\mathbf{A}^{\top}$

projected variables achieve a prescribed volume fraction V_0. Formally, this requires solving $P(\eta; \boldsymbol{x}) = V_0$, where P computes the volume of the structure after projection, where $\boldsymbol{x}$ denotes the underlying parameters (e.g., filtered densities or unconstrained network outputs).[6] This problem fits directly into the framework of Eq. (2), with $\boldsymbol{y} = \eta$ and the stationarity condition $F(\boldsymbol{x}, \boldsymbol{\eta}) = P(\eta; \boldsymbol{x}) - V_0$. Importantly, the specific choice of root-finding algorithm is immaterial to the application of the IFT; what matters is that one can find an algorithm to calculate $\eta^\star$ (for given parameters) that makes $F = 0$.

Code 1 Black-box bisection algorithm implementation and its associated *scalar* stationarity condition.

```
def bisection_algorithm(F_fn, params,
    lb=-10.0, ub=10.0, max_iter=100,
    tol=1e-10):
    """Black-box bisection algorithm
       that finds the scalar root of
       F_fn, given parameters."""
    for _ in range(max_iter):
        mid = 0.5 * (lb + ub)
        val = F_fn(params, mid)
        # Interval refinement (sign
           change assumed)
        if val > 0.0:
            ub = mid
        else:
            lb = mid
        if abs(val) < tol:
            break
    return mid

def F_stationarity_bisection(params,
    eta_star):
    """Stationarity condition for root-
       finding.
    F(params, eta^star) should be 0"""
    ...
    return F_val
```

We define a black-box implementation of the bisection method (`solve_bisection`) and annotate it with `jax.custom_vjp` to explicitly specify its behavior under AD (Code 2). The argument `nondiff_argnums` instructs `JAX` to ignore the first argument, corresponding to the stationarity function, during AD. Although `problem_data` could also be marked as non-differentiable, this is discouraged, as it contains traceable objects.

[6] We use the notation $P(\eta; \boldsymbol{x})$ to emphasize that P is parameterized by $\boldsymbol{x}$, while the algorithm controls only the scalar η.

Code 2 Custom VJP for the bisection algorithm using the Implicit Function Theorem.

```
def solve_bisection(F_fn, params, problem_data)
    :
    (lb, ub, max_iter, tol) = problem_data
    eta_star = bisection_algorithm(F_fn, params
        , lb, ub, max_iter, tol)
    return eta_star


# Mark the bisection solver as a custom_vjp
    primitive.
# The stationarity function F_fn is a non-
    differentiable Python argument.
bisection_solve_ad = jax.custom_vjp(
    solve_bisection, nondiff_argnums=(0,))

def bisection_fwd(F_fn, params, problem_data):
    """Forward pass for the custom VJP.

    Executes the black-box bisection algorithm
        and stores the primal solution eta*
        along with all data required for
        implicit differentiation."""
    eta_star = solve_bisection(F_fn, params,
        problem_data)
    residuals = (eta_star, params)
    return eta_star, residuals

def bisection_bwd(F_fn, residuals,
    output_cotangent):
    """Backward pass (VJP) computed using the
        Implicit Function Theorem."""
    eta_star, params = residuals

    # Compute partial derivatives of the
        stationarity condition using JAX's
        autodiff
    dF_dparams, dF_deta = jax.grad(F_fn, (0, 1)
        )(params, eta_star)

    # Adjoint solve: In the scalar case, this
        reduces to a simple division.
    lambda_val = output_cotangent / dF_deta

    # VJP calculation
    vjp_params = -lambda_val * dF_dparams

    return vjp_params, (None, None, None, None)

bisection_solve_ad.defvjp(bisection_fwd,
    bisection_bwd)
```

The forward pass is implemented in `bisection_fwd`. It performs the same computation as the original function, but additionally stores the fixed input `params` and the solution `eta_star`. These quantities are required to define the VJP rule. The backward pass is implemented in `bisection_bwd` using IFT. Specifically, we compute the partial derivatives of the stationarity function `F_fn` with respect to both arguments, $\frac{\partial F}{\partial \eta} \in \mathbb{R}$ and $\frac{\partial F}{\partial \boldsymbol{x}} \in \mathbb{R}^n$, evaluated at the solution $\eta^\star$, with n being the dimensionality of the parameters. These derivatives are obtained using `jax.grad`, which computes both quantities in a single reverse mode pass, assuming that the stationarity condition is written in `JAX`. Since the out-

put of the bisection solver is scalar, the incoming cotangent (`output_cotangent`) is also scalar, and the adjoint variable is obtained by a simple division. Finally, we return the VJP with respect to `params`, while returning `None` for all remaining inputs, as only the parameters are treated as differentiable. The remaining arguments—`lb`, `ub`, `max_iter`, and `tol`—are explicitly treated as non-differentiable under an IFT-based rule.

For nonlinear FEA, multidimensional root-finding algorithms such as Newton–Raphson are commonly employed. In this setting, the solution $\boldsymbol{\eta}^\star = \boldsymbol{u}^\star$ corresponds to the converged state of the nonlinear system (i.e., the equilibrium displacement field in structural mechanics), while the forward pass consists of iteratively linearizing the residual and solving a sequence of linear systems. From the perspective of AD, the treatment is conceptually identical to the bisection example: the entire nonlinear solver is viewed as a black-box algorithm that returns $\boldsymbol{u}^\star$ satisfying the stationarity condition $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{u}^\star) = \boldsymbol{0}$. Applying the implicit function theorem then requires the action of the inverse Jacobian $(\partial \boldsymbol{F}/\partial \boldsymbol{u})^{-1} \in \mathbb{R}^{m\times m}$ on a cotangent vector, which corresponds to solving a linear system involving the tangent stiffness matrix evaluated at convergence (unlike the simple division done for bisection). While this extension is mathematically straightforward, it introduces an additional practical challenge: implementing a differentiable interface to a linear solver. We therefore treat the linear solver itself as the primary case study in the following section.

1.4.2 Linear solver

A simplified computational graph corresponding to the FEA part of a TO pipeline is shown in Fig. 5, with some data dependencies not important for differentiation removed for clarity. The filtered densities are used in an interpolation scheme (here SIMP) which outputs the effective material properties. These are used to assemble the stiffness matrix, which is then reduced by removing the fixed degrees of freedom. Then, a linear solver is used to compute the solution of the linear system $\mathbf{Ku} = \mathbf{f}$, where $\mathbf{K}$ is a large, sparse stiffness matrix, with the fixed degrees of freedom removed, ensuring that the system is square, real, and of full rank. Finally, an objective is calculated using the equilibrium displacement field. While direct solvers may be suitable for small-scale problems, iterative solvers such as CG or GMRES are the standard choice in large-scale TO (Aage et al. 2014). However, as in the bisection example, the specific implementation of the solver—and even whether it is direct or iterative—is immaterial when using the implicit function theorem. Once the stationarity condition is defined, the resulting VJP rule applies to any linear solver. For wrapping the FE solver, we showcase three different methods (Cases **a**, **b**, **c** in Fig. 5), each with its own trade-offs and advantages.

In Code 3, the function `external_linear_solver` serves as a placeholder for any routine that solves the linear system, given the matrix in triplet form (data and corresponding indices) and a right-hand side vector.[7] Unlike `PyTorch`, `JAX` does not allow traced *Array*s to be directly converted to `NumPy` arrays. To interface with external libraries, we therefore use `jax.pure_callback`, which safely transfers data out of the traced computation. Once converted to `NumPy` arrays, the data can be passed to essentially any `Python`-interfacing linear algebra backend.

Code 3 Definition of the black-box linear solver for solving the reduced system of equations.

```
def external_linear_solver(K_data,
    i_inds, j_inds, f):
    """Black-box linear solver for K u
       = f.

    This function may call PETSc,
       CVXOPT, SciPy, or any other
       external library. It is treated
        as a non-differentiable
       primitive."""
    ...
    return u_solution

def wrap_external_solve(K_data, i_inds,
    j_inds, f):
    """JAX wrapper around the external
       solver. This allows calling non
       -JAX code inside a JAX program.
       """
    return jax.pure_callback(
        external_linear_solver, # The
           function to call
        jax.ShapeDtypeStruct(f.shape, f
           .dtype), # Expected output
           shape and datatype
        K_data, i_inds, j_inds, f) #
           Arguments to pass to this
           function
```

[7] In the codebase, we use a solver from `SciPy` purely for demonstration purposes, as it can be integrated with minimal boilerplate code. The approach is not restricted to this choice.

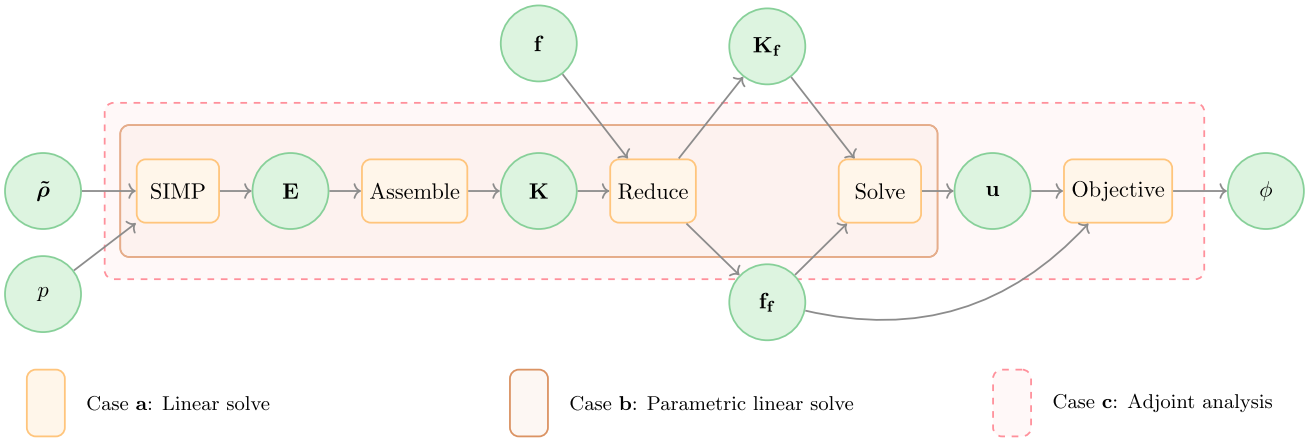

Fig. 5 Simplified computational graph of an FE simulation in topology optimization illustrating different AD wrapping strategies. Case **a** wraps only the linear solver as an AD node. Case **b** additionally includes the FEA preprocessing steps inside the node, so that the inputs are densities and other differentiable parameters. Case **c** wraps the entire simulation and objective into a single node, corresponding to classical adjoint analysis

Code 4 Custom VJP for a black-box linear solver.

```
solve = jax.custom_vjp(
    wrap_external_solve)

def solve_fwd(K_data, i_inds, j_inds, f
    ):
    """Forward pass for the custom VJP.
        """
    u_f_star = solve(K_data, i_inds,
        j_inds, f)
    residuals = (u_f_star, K_data,
        i_inds, j_inds)
    return u_f_star, residuals

def solve_bwd(residuals,
    output_cotangent):
    """Backward pass (VJP) computed
        using the Implicit Function
        Theorem."""
    u_f_star, K_data, i_inds, j_inds =
        residuals

    # Solve adjoint problem K^T *
        lambda = g
    lambda_ = solve(K_data, j_inds,
        i_inds, output_cotangent)  #
        note the transpose by swapping
        i and j
    # Compute gradient w.r.t. K_data
    dK_data = -lambda_[i_inds] * x[
        j_inds]
    df = lambda_
    return (dK_data, None, None, df)

solve.defvjp(solve_fwd, solve_bwd)
```

For case a, we can express the stationary condition as $\boldsymbol{F}(\boldsymbol{x} = \{\mathbf{K}, \mathbf{f}\}, \mathbf{u}^\star) = \mathbf{K}\mathbf{u}^\star - \mathbf{f} = \mathbf{0}$, where the inputs (obtained after elimination of free degrees of freedom) are treated as parameters. Written in this form, the VJP rule follows directly from Eq. (4). The remaining question is how to compute the required partial derivatives, namely $\partial_2 \boldsymbol{F}$ and $\partial_1 \boldsymbol{F}$. Unlike the bisection example, we deliberately avoid computing $\partial_2 \boldsymbol{F}$ using `jax.jacobian`, the vector-valued counterpart of `jax.grad`. While correct in principle, this would explicitly materialize the stiffness matrix as a dense Jacobian, which is prohibitively expensive for large-scale sparse systems and can easily lead to out-of-memory errors, since these operations just perform repeated VJP evaluations. This reflects a broader limitation of current ML frameworks, which provide limited support for automatic sparse differentiation (Hill et al. 2025).

Instead, we exploit problem structure. The Jacobian $\partial_2 \boldsymbol{F}$ is precisely the stiffness matrix $\mathbf{K}$, which is already available in sparse form. We therefore assemble $\mathbf{K}$ explicitly and solve the adjoint system $\mathbf{K}^\top \boldsymbol{\lambda} = \bar{\mathbf{u}}$ using the same external solver (see Code 11). Next, we compute the action of $\partial_1 \boldsymbol{F}^\top$ on the adjoint vector $\boldsymbol{\lambda}$. This separates naturally into contributions from $\mathbf{K}$ and $\mathbf{f}$. As before, we do not rely on AD to compute these partial derivatives, but derive them analytically. The resulting VJPs are

$$\bar{\mathbf{K}} = -\boldsymbol{\lambda} \otimes \mathbf{u}^\star, \tag{5}$$

$$\bar{\mathbf{f}} = \boldsymbol{\lambda}, \tag{6}$$

where barred quantities denote propagated cotangents. The outer product structure of $\bar{\mathbf{K}}$ follows directly from the bilinear form $\mathbf{K}\mathbf{u}$. The corresponding implementation is shown in Code 4.

An important consequence of invoking `solve` inside the backward rule is that, provided the backward pass itself is

fully traceable by JAX, higher-order derivatives are obtained automatically. During higher-order differentiation, AD is recursively applied to the backward pass, requiring no additional derivations. This wrapping strategy offers maximal flexibility, except when the solver is explicitly unrolled and differentiated within the AD framework, thereby exposing its internal iterations to the computational graph. The linear solver is fully agnostic to the surrounding computational graph and can therefore be reused across different TO pipelines (e.g., self-weight problems, stress constraints, or thermal analyses). This is not the only possible wrapping strategy; an alternative approach, in which additional preprocessing steps are included inside the wrapped node (corresponding to case **b**), is discussed in "Appendix B".

1.4.3 Traditional adjoint analysis from the lens of IFT

In TO, adjoint state sensitivity analysis is used commonly to compute gradients efficiently. The idea is to combine the objective function $\phi(\tilde{\boldsymbol{\rho}})$ (where $\tilde{\boldsymbol{\rho}}$ is the physical density) with the equilibrium constraint $\mathbf{K}\mathbf{u} = \mathbf{f}$ using Lagrange multipliers. These multipliers are chosen to eliminate the computationally expensive derivative terms. The Lagrangian is defined as:

$$L = \phi\left(\mathbf{u}\left(\tilde{\boldsymbol{\rho}}\right)\right) - \boldsymbol{\lambda}^{\top}\left(\mathbf{K}\mathbf{u}\left(\tilde{\boldsymbol{\rho}}\right) - \mathbf{f}\right),$$

where $\boldsymbol{\lambda}$ is the adjoint (Lagrange multiplier) vector. Differentiating L with respect to $\tilde{\boldsymbol{x}}$, we get:

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}\tilde{\boldsymbol{\rho}}} &= \frac{\partial\phi}{\partial\mathbf{u}}\frac{\partial\mathbf{u}}{\partial\tilde{\boldsymbol{\rho}}} - \boldsymbol{\lambda}^{\top}\left(\frac{\partial\mathbf{K}}{\partial\tilde{\boldsymbol{\rho}}}\mathbf{u} + \mathbf{K}\frac{\partial\mathbf{u}}{\partial\tilde{\boldsymbol{\rho}}}\right) \\ &= -\boldsymbol{\lambda}^{\top}\frac{\partial\mathbf{K}}{\partial\tilde{\boldsymbol{\rho}}}\mathbf{u} + \left(\frac{\partial\phi}{\partial\mathbf{u}} - \boldsymbol{\lambda}^{\top}\mathbf{K}\right)\frac{\partial\mathbf{u}}{\partial\tilde{\boldsymbol{\rho}}},\end{aligned} \tag{7}$$

assuming that the force vector $\mathbf{f}$ does not depend on $\tilde{\boldsymbol{\rho}}$. The adjoint vector $\boldsymbol{\lambda}$ is then chosen to cancel the second term, which contains the expensive Jacobian $\frac{\partial\mathbf{u}}{\partial\tilde{\boldsymbol{\rho}}}$, by solving:

$$\mathbf{K}^{\top}\boldsymbol{\lambda} = \left(\frac{\partial\phi}{\partial\mathbf{u}}\right)^{\top},$$

where we have transposed the terms. This leaves only the first term in the sensitivity expression, which is typically inexpensive to compute.

The resemblance to the linear solve in the VJP rule from Eq. (4), which arises when $\bar{\boldsymbol{y}} = \left(\frac{\partial\phi}{\partial\mathbf{u}}\right)^{\top}$ is not coincidental. If we formulate the stationarity criteria as $\boldsymbol{F}(\boldsymbol{x} = \tilde{\boldsymbol{\rho}}, \mathbf{u}) = \mathbf{K}(\tilde{\boldsymbol{\rho}})\mathbf{u} - \mathbf{f}$, then the required partial derivative is given by $\partial_{\tilde{\boldsymbol{\rho}}}\boldsymbol{F} = \frac{\partial\mathbf{K}}{\partial\tilde{\boldsymbol{\rho}}}u$. Thus, the IFT-based VJP rule gives the exact same formula for the gradients. The only difference is that adjoint analysis corresponds to differentiating the combination of an implicit function—namely, solving the nonlinear equation $\mathbf{K}\mathbf{u} - \mathbf{f} = \mathbf{0}$—and a subsequent explicit objective ϕ. Thus, adjoint analysis can be viewed as a special case of IFT application where the solver (and the associated preprocessing steps) and objective nodes are fused into a single node (Blondel and Roulet 2024), shown as case **c** in Fig. 5.

Code 5 Custom VJP rule obtained using adjoint analysis for compliance calculation.

```
@jax.custom_vjp
def compute_compliance(rho, problem_data)
    :
    """ From physical densities to
        compliance."""
    ...
    # Need to output ce_unscaled for
        derivative
    return compliance, ce_unscaled

# Forward pass
def compute_compliance_fwd(rho,
    problem_data):
    c, ce = compute_compliance(rho,
        problem_data)
    residuals = {
        "elem_comp": ce,          #
            Element-wise compliance
        "rho": rho,               #
            Physical densities
        "penal": problem_data['penal'],
        "E0": problem_data["E0"],
        "Emin": problem_data["E_min"]
    }
    return (c, ce), residuals  # Outputs
        and residuals for backward

# Backward pass (from adjoint analysis)
def compute_compliance_vjp(residuals,
    down_cotangents):
    c_dot, ce_dot = down_cotangents  #
        VJP seed  from upstream
    del ce_dot  # Assuming ce has no
        downstream relevance
    # Unpack residuals
    ce = residuals["elem_comp"]
    rho = residuals["rho"]
    penal = residuals["penal"]
    E0 = residuals["E0"]
    Emin = residuals["Emin"]
    # dc/drho
    dc_drho = -penal * rho**(penal - 1) *
        (E0 - Emin) * ce
    vjp_rho = dc_rho * c_dot  # Apply
        chain rule
    return vjp_rho.reshape(rho.shape,
        order='F'), None  # Second arg is
        dL/dproblem_data = None

# Register VJP rules
compute_compliance_ad.defvjp(
    compute_compliance_ad_fwd,
    compute_compliance_ad_bwd)
```

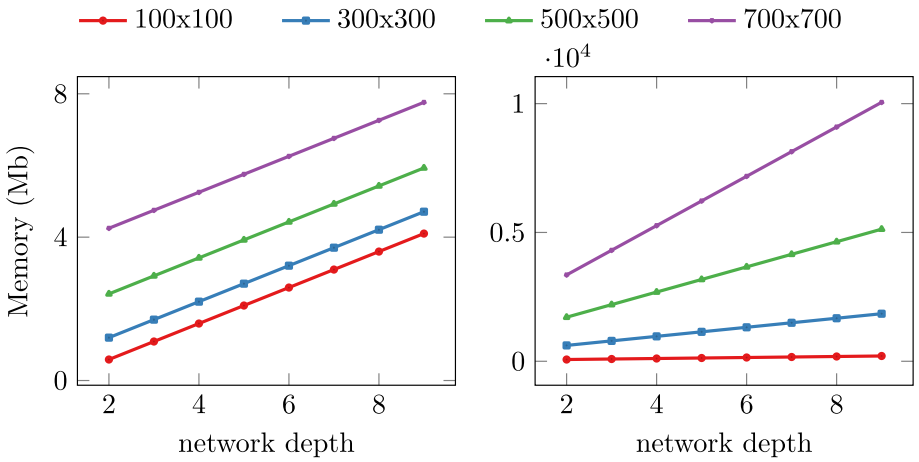

Fig. 6 Memory scaling of reverse mode AD with neural network depth. Different curves correspond to different grid resolutions. The network is evaluated independently at each grid point, and gradients are computed for a dummy loss defined as the spatial average of the network output. *Left* memory footprint of the gradient computation accounting only for inputs, parameters, and outputs. *Right* memory associated with intermediate values saved for backpropagation (AD residuals), which dominates the overall memory footprint

Since the adjoint analysis requires the specification of an objective, we choose compliance as an example here. Mathematically, compliance is a scalar-valued function $c : \mathbb{R}^N \rightarrow \mathbb{R}$, where N is the number of design variables (and the number of finite elements as well). Its gradient with respect to the physical density variables is given by the well-known sensitivity expression from the adjoint method,

$$\nabla c^\top = -p\, \tilde{\boldsymbol{\rho}}^{\,p-1} (E_0 - E_{\text{min}}) \boldsymbol{c}_\text{e}, \tag{8}$$

where p is the SIMP penalty exponent, $\boldsymbol{c}_e \in \mathbb{R}^N$ is the vector of element-wise strain energy densities (unscaled), and E_0 and E_{min} are the Young's moduli of the solid and void phases, respectively (Andreassen et al. 2011). Since the output is a scalar, the Jacobian reduces to the gradient written as a row vector. In this case, the VJP rule (Code 5) is just the gradient in Eq. 8 multiplied by the scalar signal coming from downstream (`c_dot`). The backward rule only defines gradients with respect to `rho`; all other inputs, such as the FEA problem data, are treated as non-differentiable.

1.5 Computational cost

Understanding the computational cost of AD requires considering both the graph traversal and the node-level calculations. AD frameworks propagate derivatives by computing the product with the Jacobian; obtaining the full Jacobian requires multiple passes through the graph using *seeding*. The seed specifies the initial values of the derivative signal: standard unit basis vectors in $\mathbb{R}^n$ for forward mode and in $\mathbb{R}^m$ for reverse mode.[8]

As a concrete example, in reverse mode AD, if the computational graph has two outputs of interest ($[\, y_1 \;\; y_2 \,]^\top$), one would first propagate the seed vector $[\, 1 \;\; 0 \,]^\top$ using VJPs through the entire graph, which yields the first row of the Jacobian (∇y_1). Repeating this process with $[\, 0 \;\; 1 \,]^\top$ gives the second row (the filled bars in Fig. 3 denote ones, empty bars denote zeros). Thus, computing the full Jacobian in reverse mode requires one forward pass to build the graph followed by m backward passes, whereas forward mode requires n forward passes. More generally, reverse mode constructs the Jacobian row by row, while forward mode constructs it column by column. Consequently, for optimization problems with a single objective and relatively few constraints, where the Jacobian has fewer rows than columns (wide), reverse mode is more efficient. Forward mode AD, on the other hand, is useful for computing directional derivatives and randomized gradient estimates (Blondel and Roulet 2024), for accessing second-order information such as Hessian-vector products, and for specific nodes where input and output dimensionalities are comparable. An example is `JAX-FEM` (Xue et al. 2023), where the stiffness matrix is obtained by applying forward mode AD to the PDE residual, since the residual and trial function have the same dimensionality. Nevertheless, the complete TO workflow necessitates the use of reverse mode AD.

The cost of a single pass through the graph is bounded linearly by the cost of the function evaluation (Griewank 2000), typically around twice the cost of the original function (Baydin et al. 2017; Blondel and Roulet 2024), since both the primal value and derivative contributions must be computed. In TO, most of the cost is due to linear solves. This is easily observed in Code 4: one function evaluation involves a linear solve, and the corresponding VJP rule requires another. Using analytical derivatives would also require these two solves. Thus, in TO pipelines, using AD for sensitivity analysis is nearly as cost-efficient as analytical expressions. Empirical studies (Chandrasekhar et al. 2021) confirm this for

[8] Forward mode allows flexible choice of the seed vector, enabling efficient computation of directional derivatives in arbitrary directions. For reverse mode, carefully combining seed vectors based on the sparsity of the Jacobian can improve efficiency (Hill et al. 2025); however, sparsity-aware AD is not yet fully supported in standard ML frameworks.

compliance minimization and compliant mechanism design problems.

Exploiting problem structure via the implicit function theorem (IFT) can further improve efficiency. For self-adjoint problems, adjoint-based analysis renders the backward linear solve unnecessary. Wrapping both solver and objective nodes as a single custom AD node (Code 5) is then particularly efficient. Even when computing partial derivatives manually (Code 4), we avoided AD because the expressions were simple to derive. For nonlinear problems, IFT-based rules are advantageous since the backward pass requires only one linear solve, whereas unrolling multiple solver iterations in the forward pass would be expensive, assuming that the objective depends only on the final state.

Unrolling solver steps in reverse allows users to control the computational cost by selecting the number of iterations, but memory usage can become significant as the computational graph grows deeper. Figure 6 illustrates this effect using a neural network as an illustrative example, where increased depth corresponds to adding more hidden layers. The network has a fully connected architecture with 2 input neurons (design domain coordinates in 2D), 1 output neuron (intermediate density), and 256 hidden neurons per layer, similar to architectures commonly used in neural TO (Chandrasekhar and Suresh 2020; Zehnder et al. 2021b; Doosti et al. 2021). The network is evaluated independently at each grid point.[9] No physical simulation is performed; the objective is a dummy function that simply averages the network output over all grid points.

The left panel shows the memory footprint accounting only for inputs, parameters, and outputs. Memory increases either due to a larger number of parameters (increased depth) or due to higher input and output dimensionality (larger grids at fixed depth). The right panel isolates the memory associated with intermediate values saved for backpropagation (AD residuals), which dominates the total memory usage by orders of magnitude. This residual memory grows linearly with both network depth and input resolution, highlighting a key consideration for neural topology optimization: deep networks combined with large spatial discretizations can lead to substantial memory overheads that are largely independent of parameter count. On GPUs, this memory footprint often becomes the primary limiting factor. To mitigate this issue, modern AD systems employ *checkpointing* strategies. Rather than storing all intermediate values during the forward pass, only a selected subset is retained. Various checkpointing strategies exist, and identifying optimal schemes remains an active area of research. In practice, however, this process is handled automatically by modern frameworks and typically requires no user intervention. Intermediate values that were not stored are then recomputed as needed during the backward pass by re-executing portions of the original computation. This strategy helps reduce memory usage at the expense of extra computation and is widely used to make reverse mode AD more scalable.

[9] Note that this results in a very wide computational graph, since a separate forward pass is required for each grid point. Each such forward pass introduces additional intermediate values that must be stored for reverse mode AD.

Code 6 Calculating compliance and volume constraint.

```
def simp_and_reduced_solve(
    physical_densities, problem_data):
    # Compute compliance
    E = problem_data['E_min'] +
        physical_densities**penal * \
        (problem_data['E0'] -
            problem_data['E_min'])
    iK, jK, sK =
        assemble_stiffness_matrix_parts(
        E, problem_data)
    free_dofs = problem_data['free']
    f = problem_data['F']
    f_f = f[free_dofs]
    # reduce K to K_f
    iK_f, jK_f, sK_f = reduce_K(iK, jK,
        sK, free_dofs, len(f))
    # Solve system
    if ML_framework_to_use == "torch":
        # Torch needs tensor inputs to
            be passed since we provide
            gradients w.r.t the force
            vector as well in teh custom
             VJP rule
        u_f = solve(sK_f, iK_f, jK_f,
            torch.tensor(f_f, device=
            sK_f.device, dtype=sK_f.
            dtype))
    else:
        u_f = solve(sK_f, iK_f, jK_f,
            f_f)
    return u_f

def obj_and_constraint_fn(rho):
    rho = rho.ravel(order='F')
    physical_densities =
        apply_density_filter(rho,
        problem_data)
    # Compute compliance - SIMP,
        assemble K, Remove free DOFs,
        solve system
    u_f = simp_and_reduced_solve(
        physical_densities, problem_data
        )
    f_f = problem_data['F'][problem_data
        ['free']]
    compliance = u_f.T @ f_f
    constraint = jnp.mean(
        physical_densities) - volfrac
    aux_info= (compliance, constraint,
        physical_densities)
    return (compliance, constraint),
        aux_info
```

2 Using the code

As a practical demonstration of the principles discussed earlier, we present a `Python`-based implementation of TO inspired by the well-known 88-line MATLAB code of Andreassen et al. (2011). The goal of this code is not to provide a full-fledged framework, but to help readers make the concepts introduced earlier concrete, thereby becoming familiar with working with AD at a deeper level. In this sense, the code is complementary to that of Chandrasekhar et al. (2021), who provide more extensive examples but rely on fully reimplementing the TO workflow within an AD framework. Unlike the original MATLAB version, our implementation exploits the AD capabilities of modern ML frameworks—specifically `JAX` and `PyTorch`—for gradient computation. Both backends are provided to demonstrate that the core principles underlying custom AD rules remain consistent across frameworks, despite differences in programming style, with `PyTorch` being more object-oriented and `JAX` favoring a functional paradigm.

The code separates black-box components, implemented in `NumPy` and `SciPy`, from natively differentiable components. While a direct solver from `SciPy` is used by default, the design allows users to substitute any external solver (direct or iterative) without modifying the surrounding optimization code. End-to-end differentiability is achieved by defining custom VJP rules, using the implicit function theorem or analytical derivatives where appropriate. This hybrid approach enables gradient propagation through complex pipelines even when parts of the computation lie outside the AD system. The full implementation is provided as Google Colab notebooks, allowing readers to run and experiment with the code online.

2.1 For running TO

For standard TO, the notebook `TO.ipynb` demonstrates how to run the optimization while supporting gradient computation through either `JAX` or `PyTorch`. The user specifies the backend by setting `ML_framework_to_use = "jax"` or `"torch"`. After selecting material properties, mesh discretization, and other optimization parameters, the function `setup_fea_problem(...)` creates the problem data, including the connectivity matrix, element stiffness matrix, filter kernel, and boundary conditions (for the Cantilever beam).

The function in Code 6 calculates the objective (compliance) and constraint (volume fraction) as functions of the design variable `rho`. The function returns both the values and auxiliary information useful for logging. The resulting computational graph, with `rho` as input and compliance and constraint as outputs, can then be differentiated automatically. Reverse mode differentiation is performed using `jacobian, aux = jax. jacrev (obj_and_constraint_fn, has_aux=True)(rho)`. This call executes two backward passes to compute the Jacobian of the computational graph. The resulting Jacobian provides the gradients of both the objective and the constraint, which are then used in the optimality criteria update scheme (Andreassen et al. 2011).

Among the finite element operations—such as stiffness matrix assembly [`assemble_stiffness_matrix_ parts(...)`], linear system solution [`solve(...)`], and compliance evaluation—only the linear solver is implemented outside the AD framework. The density filter (`apply_density_filter`) is reimplemented natively in the chosen AD backend. For improved efficiency, an adjoint-based wrapping of the compliance computation (Code 5) can be employed. This approach sacrifices some code generality but eliminates the need for an additional linear solve during the backward pass. Note that with the current approach of wrapping only the FE solver, the code can be easily extended to handle compliant mechanism design and stress-constrained optimization problems as well.

2.2 For neural TO

To demonstrate a ML-integrated workflow, we extend the code to include neural TO, formulated as

$$\begin{aligned} &\boldsymbol{\rho}^{\star} = \arg\min_{\boldsymbol{\rho}\in\mathbb{R}^N} \; c = \frac{1}{2}\mathbf{u}^{\intercal}\mathbf{f}, \\ &\text{subject to } \mathbf{K}(\boldsymbol{\rho})\,\mathbf{u} = \mathbf{f}, \\ &\boldsymbol{\rho} = \sigma_{\eta}\big(\Phi(\boldsymbol{\theta};\boldsymbol{x})\big) \in [0,1]^N, \\ &\sum_{i=1}^{N} v_i\rho_i = V_0, \end{aligned} \tag{9}$$

where $\boldsymbol{\theta}$ are the parameters of an untrained neural network Φ with input $\boldsymbol{x}$. The network outputs logits, which are projected into [0, 1] via a parametric sigmoidal filter σ_η (Hoyer et al. 2019). The filter enforces the global volume constraint $\sum_i v_i\rho_i = V_0$ by adaptively shifting the sigmoid through a bisection algorithm. An alternative way to handle constraints in ML-based workflows is to include constraint violations as additional loss terms, for example via quadratic penalty methods that convert constrained problems into unconstrained ones. However, such approaches introduce extra hyperparameters whose tuning is nontrivial in practice. Insufficient penalization leads to constraint violations, whereas overly large penalties can result in ill-conditioned optimization problems and convergence to suboptimal solutions. Enforcing constraints directly at the neural network outputs is therefore often preferable, as it guarantees exact enforcement of constraints—an outcome that is difficult to achieve with penalty-based methods. Importantly, because sensitivities are computed via AD, users can easily explore

and compare these alternatives without additional effort in sensitivity analysis. The rest of the formulation follows standard compliance minimization: N is the number of finite elements, $\mathbf{u}$ is the displacement vector obtained from $\mathbf{Ku} = \mathbf{f}$, $\mathbf{K}$ and $\mathbf{f}$ are the global stiffness matrix and force vector, and c is the compliance objective. Material stiffness is interpolated using SIMP (Bendsøe and Sigmund 2004), with v_i denoting the element volumes.

The introduction of a neural network parameterization can enable the optimization process to reach improved optima, although this effect is more pronounced for strongly non-convex problems (Sanu et al. 2025; Herrmann et al. 2024). For compliance minimization—which is formally non-convex, yet usually exhibits a benign optimization landscape (Abdelhamid and Czekanski 2021)—the advantages over traditional density-based parameterizations may therefore be limited. Moreover, both convergence behavior and the quality of the obtained optima depend on the chosen neural architecture, as the network reshapes the objective and constraint landscapes. This landscape transformation is precisely what allows neural TO to explore alternative solutions that may be inaccessible to classical parameterizations. Importantly, because AD is used throughout, different neural network parameterizations can be exchanged seamlessly—without modifying the sensitivity analysis—making it straightforward for readers to experiment with alternative architectures.

2.2.1 Constructing NNs for reparameterization

Reparameterization methods redefine the design space by introducing new decision variables $\boldsymbol{\theta} \in \mathbb{R}^M$. This transformation offers flexibility by decoupling the design representation from the finite element discretization, allowing for over-parameterization ($M > N$) or under-parameterization ($M < N$). NNs serve as powerful function approximators in this setting. By optimizing their parameters, organized across multiple layers, NNs can learn complex mappings from inputs to material densities. Each layer performs a linear transformation followed by a nonlinear activation, and the layers are composed sequentially. The output of one layer becomes the input to the next, forming a recursive structure.

In our implementation, we support three types of neural network architectures (Sanu et al. 2025):

- A simplified U-Net-style convolutional neural network (CNN; Hoyer et al. 2019; Zhang et al. 2021).
- A standard feed-forward network, also known as a multilayer perceptron (MLP; Chandrasekhar and Suresh 2020; Deng and To 2020),
- A sinusoidal representation network (SIREN), which uses sinusoidal activation functions (Sitzmann et al. 2020).

Users can select the desired architecture at the beginning of the script using the `nn_type` variable (and choosing from `"cnn"`, `"mlp"` or `"siren"`). Unlike the MLP and SIREN architectures, which predict densities pointwise as functions of design domain coordinates, the CNN directly outputs the full density field over the analysis grid in a single forward pass. As a result, the CNN's parameter count is closely tied to the resolution of the final design grid.

We use the `Keras` package (Chollet 2015) to construct these networks. `Keras` provides a high-level API for network design and includes various unconstrained optimizers. Notably, it supports three major backend frameworks: `TensorFlow`, `PyTorch`, and `JAX`. This flexibility enables seamless integration of `Keras`-based models into different AD pipelines. While `Keras` supports multiple ways of defining networks, we adopt the *functional* API, where layers are treated as callable objects that transform inputs. A typical workflow begins by declaring an input using the `Input` layer,[10] which explicitly defines the input shape. This input is then passed through a sequence of layers to perform intermediate computations. The output of the final layer becomes the output of the model. The complete network is created by wrapping the input and output in a `keras.Model` object.

In our codebase, the function `def create_network_and_input(...)` constructs the selected neural network architecture, initializes its parameters, and outputs both the initialized model as well as the corresponding input tensor. The inputs for SIREN and MLP are the coordinates of the centroids of each finite element, scaled to $[-1, 1]$ while for the CNN, the input is a random vector. Once the model has been created, the layers included in the model can be shown by invoking `model.summary()`.

Code 7 Creating MLP using `Keras`'s functional style.

```
inputs = keras.layers.Input(shape=(2,), name='
    coordinates')
x = inputs
for i in range(n_h_layers):
    x = keras.layers.Dense(units)(x)
    x = keras.layers.LeakyReLU()(x)
    x = keras.layers.BatchNormalization()(x)
outputs = keras.layers.Dense(1)(x)
model = keras.Model(inputs=inputs, outputs=
    outputs, name=f'{nn_type.upper()}')
```

Multi-layer Perceptron (MLP)

Here, we provide an example illustrating the construction of the simplest neural architecture, namely an MLP. In an MLP, each neuron in a given layer is fully connected to all neurons in the subsequent layer. We adopt the architecture proposed in

[10] It is important to note that ML libraries reserve the first (or last) dimension for the batch size.

Chandrasekhar and Suresh (2020). The construction of alternative neural architectures follows the same procedure and is detailed in "Appendix C". The input to the network is the spatial coordinate (x, y) from the design domain, giving an input shape of `(2,)`. Each hidden layer consists of a dense layer [`layers.Dense(...)`], followed by a leaky ReLU activation [`layers.LeakyReLU()`], and batch normalization [`layers.BatchNormalization()`] to stabilize training. We show the code that is used for network creation in Code 7, where the MLP has `n_h_layers` hidden layers with each layer having `units` neurons.

2.3 Performing neural TO

The function `def run_with_jax_backend(...)` integrates all components and carries out the actual optimization using the `JAX` backend. Unlike `PyTorch`, which adopts a more object-oriented and Pythonic style, `JAX` follows a functional programming paradigm. This shift requires several important adaptations. In `JAX`, all functions must be pure—they cannot have side effects or modify global state. While this constraint can feel restrictive at first, it aligns closely with the mathematical notion of a function and makes automatic differentiation more intuitive and predictable. However, it also requires the user to manage state explicitly. Consider the following example, where a function performs a computation and also updates a global counter:

```
count = 0
def compute_and_
    count(x):
    y = x + 1
    count = count + 1
    return y
```

$\Longrightarrow$

```
def compute_and_count_
    pure(x, count):
    y = x + 1
    count = count + 1
    return y, count
```

Code 8 Neural TO forward pass: from neural network parameters to objective calculation.

```
def loss_fn(train_vars, non_train_vars):
    # NN call
    output, non_train_vars = nn_model.
        stateless_call(
            train_vars, non_train_vars,
                nn_input)
    output = output.astype(jnp.float64)
    rho = volume_enforcing_filter(output,
        volfrac)
    rho = rho.ravel(order='F')
    physical_densities = apply_density_filter(
        rho, problem_data)
    # Compute compliance - SIMP, assemble K,
        Remove free DOFs, solve system
    u_f = simp_and_reduced_solve(
        physical_densities, problem_data)
    f_f = problem_data['F'][problem_data['free'
        ]]
    compliance = u_f.T @ f_f
    return compliance, (non_train_vars,
        physical_densities)
```

Code 9 Parametric sigmoidal filter that constrains the network outputs to valid densities while enforcing the volume constraint.

```
def volume_enforcing_filter(x, volfrac):
    """JAX version with differentiable sigmoid
        """
    def root_fn(eta, x_inp):
        return jax.nn.sigmoid(eta + x_inp).mean
            () - volfrac

    eta_star = bisection_differentiable(root_fn
        , x)
    return jax.nn.sigmoid(eta_star + x)
```

The first version is not pure, as it modifies a global variable. In contrast, the second version explicitly receives the state (count) as input and returns the updated value. This is the idiomatic approach in `JAX`: any state that changes within a function—such as model parameters, optimizer states, or training metrics—must be passed in and out of functions explicitly.

After defining the problem data for FEA and initializing the NN model, we construct a `loss_fn` (Code 8). The neural network is decomposed into two disjoint components: `train_vars`, the trainable parameters to be optimized, and `non_train_vars`, which include everything else included in the `Keras` model. This decomposition is a direct result of `JAX`'s functional design. Within the loss function, the neural network is evaluated using `model.stateless_call`, a `Keras` utility that enables applying the model given explicit state and inputs, preserving functional purity.

Code 10 Neural TO training.

```
# Training state
trainable_vars = [v.value for v in nn_model.
    trainable_variables]
non_trainable_vars = nn_model.
    non_trainable_variables
optimizer.build(nn_model.trainable_variables)
opt_vars = optimizer.variables

# Training loop
losses = []
designs = []
for epoch in range(max_iterations):
    (loss, (non_trainable_vars, design)), grads
        = jax.value_and_grad(
        loss_fn, has_aux=True)(trainable_vars,
            non_trainable_vars)
    trainable_vars, opt_vars = optimizer.
        stateless_apply(
        opt_vars, grads, trainable_vars)
    losses.append(loss)
    designs.append(design)
    if epoch % 5 == 0:
        print(f"JAX - Epoch {epoch}, Loss: {
            loss:.6f}")
```

The network output is then passed to the function `def volume_enforcing_filter(...)` (Code 9), which adjusts the design field to satisfy a global volume constraint using the bisection algorithm. Since we implemented a custom VJP rule for this algorithm (Code 2), all downstream operations remain differentiable, and accurate gradients are computed using AD. The volume-constrained field is subsequently filtered using `apply_density_filter`, a differentiable density filter implemented using convolutions, similar to the one described in Ferrari and Sigmund (2020). The final filtered density field is used to compute the compliance (`compute_compliance_differentiable`), which is returned by the loss function along with the updated non-trainable variables for further state tracking across iterations and physical densities for enabling visualization.

Once the loss function is defined, gradient-based optimization can proceed using `jax.grad` or `jax.value_and_grad` to compute sensitivities with respect to the trainable neural network parameters. An optimizer must then be selected to update the parameters. We use the Adam optimizer (Kingma and Ba 2015), a first-order method widely used in machine learning, although users can choose other optimizers provided by `Keras`. The learning rate, a critical hyperparameter, should be tuned based on the network architecture. Low learning rates (1e−4 to 1e−3) result in slow but stable training, which is necessary for SIREN. In contrast, both MLPs and CNNs can often tolerate higher learning rates, up to 1e−2, with CNNs generally exhibiting greater stability. To further improve training stability, we clip the gradient norm to a fixed threshold. A higher clipping threshold allows for faster convergence but can lead to instability, as noted in Chandrasekhar and Suresh (2020). The interplay between the learning rate and the clipping value is therefore crucial for achieving both fast and stable training. To initialize the optimizer, we provide the set of trainable parameters, accessible via `model.trainable_variables` in `Keras`. The non-trainable parameters—such as internal statistics from layers like batch normalization—are grouped in `non_train_vars`. While they are not updated by the optimizer, they are essential for the correct functioning of the model. `Keras` provides a `stateless_apply` method that uses these variables to apply the optimizer update in a functionally pure way (Code 10).

The resulting designs obtained from the code are shown in Fig. 7. As observed in (Sanu et al. 2025), neural networks generally converge more slowly than SIMP with optimality criteria, and each architecture imparts distinct characteristics to the final design. For example, MLP-based parameterizations typically yield simpler layouts with fewer fine features. Moreover, variations in network initialization can lead to different optimization trajectories and, consequently, different final topologies. By modifying the neural architecture and optimizer choices, users can explore a wide range of optimized designs.

3 Summary and conclusions

In this article we explored the automatic differentiation capabilities of modern ML frameworks such as `JAX` and `PyTorch` in the context of topology optimization. Our objective was to bridge the gap in terminology between the two fields and reduce the entry barrier for TO researchers seeking to leverage AD for faster experimentation and development. We demonstrated how TO components can be wrapped using custom rules based on Jacobian–vector products or vector–Jacobian products, enabling seamless integration without requiring complete rewrites. We further compared implicit function theorem-based differentiation with adjoint analysis, highlighting how the former allows black-box solvers to be incorporated into differentiable workflows. Finally, we illustrated these ideas through a compact neural TO example, where neural networks serve as parameterizations within the standard TO pipeline.

While the benefits of AD are substantial, the choice between manual derivations, pure AD, and hybrid approaches should be guided by problem structure, performance constraints, and flexibility needs. Manual derivations exploit mathematical structure efficiently and avoid AD dependencies, but they become error-prone and laborious as workflows grow more complex. Pure AD maximizes flexibility and accelerates prototyping, but naive use can suffer from memory scaling issues, and current ML frameworks provide only limited support for sparse calculations, which are necessary for large-scale problems. Hybrid approaches, where AD is applied selectively, represent a pragmatic middle ground—as demonstrated here—by keeping solvers in high-performance backends while delegating composition and derivative bookkeeping to AD. This pattern, seen in both ML (e.g., deep implicit layers) and modern TO codes such as `FEniTop` (Jia et al. 2024), preserves performance without sacrificing too much flexibility.

Several caveats follow from this hybrid perspective. Treating solvers as opaque nodes can hide intermediate quantities (e.g., displacement fields) that may later be required for constraints such as stresses, and can silently break gradient propagation. Exploiting problem structure remains important: for self-adjoint objectives such as compliance, adjoint analysis can eliminate backward linear solves, though in general it may be cleaner to wrap equilibrium solvers with IFT-based rules. Finally, it is worth noting that ML-oriented AD frameworks sometimes return gradients even for mathematically non-differentiable operations (e.g., clipping), prioritizing the delivery of a usable gradient signal over strict correctness.

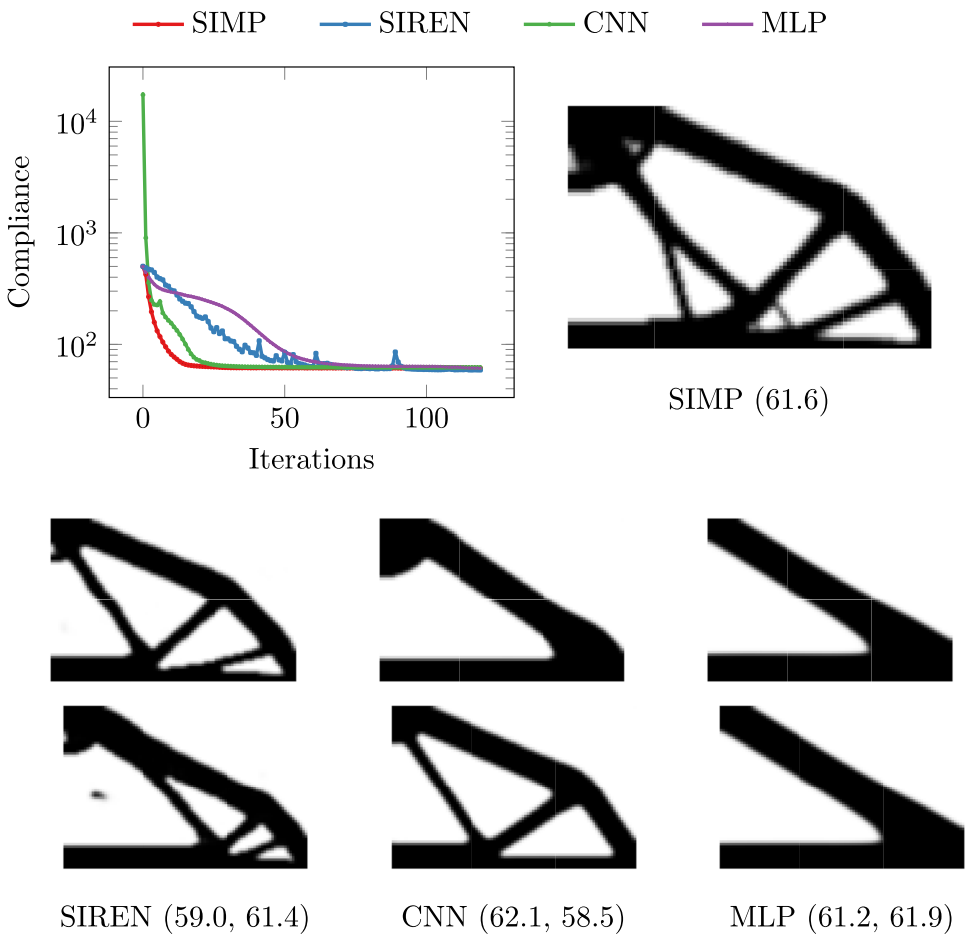

Fig. 7 Results for compliance minimization: SIMP denotes the conventional topology optimization without neural network reparameterization, solved using the optimality criteria method. The three neural network reparameterizations are trained using Adam with a learning rate of 1e−3, each from two different random initializations (corresponding to the two designs). Compliance values (without thresholding to black and white) are shown beside the method name

Appendix A Alternate AD workflows in TO

In the main text, we present a simplified computational graph corresponding to operations commonly performed in density-based topology optimization (TO). Here, we illustrate two alternate workflows that are interesting from an automatic differentiation (AD) perspective when machine learning (ML) components are integrated into a TO pipeline. The nodes shown are illustrative; the actual computational graph would consist of many more primitive operations.

The first workflow (Workflow A in Fig. 8) corresponds to the approach adopted in Sun et al. (2025). In this setting, an ML model $\mathcal{M}_{\boldsymbol{\theta}}$ is used not only to reparameterize the density field ($\tilde{\mathbf{x}}$) but also to simultaneously predict the displacement field ($\mathbf{u}$). These outputs are used to construct a compound loss consisting of a compliance term c and a potential energy (simulation) term. The potential energy is decomposed into p_1 (internal forces), p_2 (external work), and an additional term p_3, introduced to ensure positivity of the total potential energy, computed as $p_1 + p_2$.[11] Dashed connections in Fig. 8 indicate *detached gradients*: while the corresponding quantities participate in the forward computation, the associated connections are removed during the backward pass, preventing gradient propagation through those paths.

This design choice reflects the fact that the model outputs densities and displacements simultaneously, and therefore does not encode the physical dependency $\mathbf{u}(\tilde{\mathbf{x}})$ that arises from a nested equilibrium solve in conventional TO. Consequently, when evaluating the compliance loss, the contribution of the displacement field (entering through the strain $\boldsymbol{\epsilon}$) is intentionally ignored in the backward pass by treating $\mathbf{u}$ as fixed, effectively assuming equilibrium. The SIMP interpolation (SIMP-1) is modified accordingly so that the resulting gradients are consistent with those obtained from traditional adjoint-based sensitivity analysis, which explicitly relies on this physical dependency. In calculating p_1, the densities are treated as constants during differentiation. Further, the gradients for p_3 are also detached to prevent it from affecting the equilibrium displacement.

Workflow B is part of the TO pipeline, adapted from Vijayakumaran et al. (2025), where the design variables extend beyond densities ($\mathbf{x}$) to include element-wise microstructural parameters ($\boldsymbol{\alpha}$), resulting in a multiscale simulation

[11] The figure presents a schematic view that captures the essential AD structure rather than all implementation-level details.

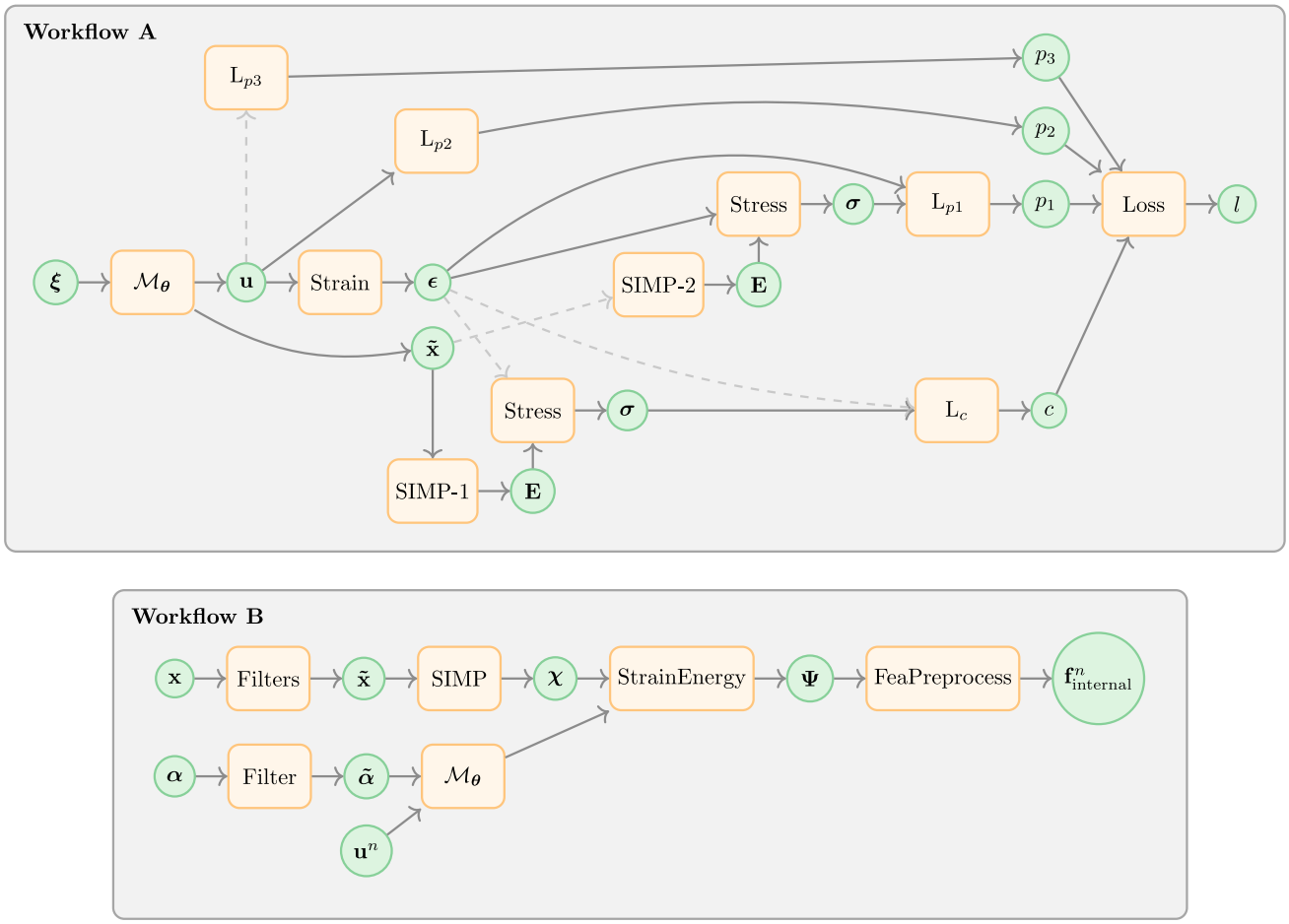

Fig. 8 Illustrative computational graphs for two alternate automatic differentiation (AD) workflows in topology optimization (TO) involving machine learning (ML) components. Circular nodes denote data (e.g., densities $\boldsymbol{x}$, microstructural parameters $\boldsymbol{\alpha}$, displacements $\mathbf{u}$, strains $\boldsymbol{\epsilon}$), while rectangular nodes denote operations (ML models $\mathcal{M}_{\boldsymbol{\theta}}$, constitutive laws, FE operators, and loss terms). Solid arrows indicate standard data dependencies in both evaluation and differentiation, whereas dashed arrows denote detached gradients: values propagate in the forward pass but corresponding paths are removed during the backward pass. Workflow A: an ML model jointly predicts densities and displacements; compliance and potential-energy-based losses (c, p_1, p_2, p_3) are formed with selective gradient detachment to recover adjoint-consistent sensitivities. Workflow B: multiscale TO with element-wise densities $\boldsymbol{x}$ and microstructural parameters $\boldsymbol{\alpha}$; a pretrained ML material model supplies strain energy density within an FE solve, with adjoint-based sensitivities and AD used selectively to differentiate neural network-dependent internal forces

framework. This extension is motivated by the prohibitive computational cost of nonlinear multiscale topology optimization when conventional nested homogenization (e.g., FE^2) schemes are employed, making repeated microscale simulations within the optimization loop impractical. To alleviate this bottleneck, a pretrained neural network material model ($\mathcal{M}_{\boldsymbol{\theta}}$) is used within the finite element (FE) simulation as a surrogate for the homogenized constitutive response, mapping microstructural descriptors to the macroscopic material behavior. The use of a physics-consistent ML model ensures numerical robustness of the nonlinear FE analysis while retaining sufficient expressiveness to capture microstructure-dependent responses.

This model takes the microstructural parameters and the displacement state from the previous load increment as inputs and predicts the strain energy density. The full simulation involves multiple load increments and a Newton–Raphson scheme. Automatic differentiation (AD) is employed to compute stresses and consistent tangent moduli from the ML-based strain energy density, which are essential for the forward problem and the convergence of the nonlinear solver. However, gradients for TO are not obtained by differentiating through these iterative solution procedures. Instead, the authors derive adjoint-based sensitivity expressions, which require evaluating partial derivatives of the internal force vector with respect to the design variables, namely $\partial \mathbf{f}^n_{\text{internal}}/\partial \boldsymbol{\alpha}$ and $\partial \mathbf{f}^n_{\text{internal}}/\partial \mathbf{x}$.

The partial derivatives that pass through the neural network material model are analytically cumbersome. AD is therefore employed selectively to evaluate these constitutive-level derivatives, while the global adjoint sensitivity analysis is carried out using classical methods.

Appendix B Wrapping a parametric linear solver

Code 11 Definition of the black-box linear solver and its associated stationarity condition: $\boldsymbol{F} = \mathbf{K}\mathbf{u} - \mathbf{f}$

```
def F_stationarity(params, u_f_star,
    problem_data):
    """Stationarity condition for a
        parametric linear system.
    F(params, u) = K(params) u - f = 0
        at u = u*"""
    penal, rho = params #
        Differentiable parameters of
        the system

    # Assemble stiffness matrix from
        parameters (e.g. SIMP
        interpolation)
    E = simp(rho, problem_data["E0"],
        problem_data["E_min"], penal)
    iK, jK, sK =
        assemble_stiffness_matrix(E,
        problem_data["KE"],
        problem_data["cMat"])

    # Extract reduced system
        corresponding to free degrees
        of freedom
    free_dofs = problem_data["free"]
    f = problem_data["F"]
    n_dofs = len(f)

    K_f = reduce_K(iK, jK, sK,
        free_dofs, n_dofs)
    f_f = f[free_dofs]

    return K_f @ u_f_star - f_f
```

In this case, we zoom out of the solver node in Fig. 5 and showcase a different way to wrap the linear solver. Here, we consider the stationary function of the form $\boldsymbol{F}(\boldsymbol{\theta} = \{\tilde{\boldsymbol{\rho}}, p\}, \mathbf{u}^\star) = \mathbf{K}(\boldsymbol{\theta})\mathbf{u} = \mathbf{f}(\boldsymbol{\theta})$. While we only consider differentiating with respect to the SIMP penalty (p) and physical densities ($\tilde{\boldsymbol{\rho}}$) here, any additional parameter could be included. The associated stationarity condition is constructed in Code 11. Although more steps are involved, it simply interpolates material properties, assembles the stiffness matrix, eliminates fixed degrees of freedom, and evaluates the stationarity, which vanishes only at the equilibrium solution $\boldsymbol{u}^\star$.

Code 12 registers the VJP rule for the linear system solve. Here, the entire mapping from parameters to the equilibrium displacement components associated with the free degrees of freedom is treated as a single black-box node. As shown in `def solve(...)`, the same operations appearing in the stationarity condition—SIMP interpolation, assembly, and reduction—are performed inline, but are condensed into a single function (`get_reduced_K`) for brevity. In the backward pass, we deliberately avoid computing $\partial_2 \boldsymbol{F}$ using `jax.jacobian`, as in case **a**. However, for the parameter derivatives, we employ a different route. Rather than materializing $\partial_1 \boldsymbol{F}$, we only require its action on the adjoint variable $\boldsymbol{\lambda}$. This corresponds exactly to a vector–Jacobian product and is computed efficiently using `jax.vjp`, which linearizes the stationarity function and returns a callable for evaluating VJPs. Finally, the resulting sensitivities are scaled by a negative sign, consistent with the implicit function theorem.

The main advantage of this wrapping is that we rely on AD to compute the difficult partial derivatives, without doing any mathematical calculations. Further, this solver can also apply for different TO workflows, but is less general than case **a**.

Code 12 Custom VJP for a parametric black-box linear solver using the Implicit Function Theorem.

```
@jax.custom_vjp def solve(params, problem_data):
    """Solves the reduced FEA system using a black-
        box solver."""
    penal, rho = params

    # Assemble reduced stiffness matrix K_f -
        involves SIMP interpolation, assembly, and
         reduction (Same a F_stationarity)
    K_f = get_reduced_K(penal, rho, problem_data)

    # Reduce right-hand side
    free_dofs = problem_data["free"]
    f = problem_data["F"][free_dofs]

    # Extract sparse matrix data for the external
        solver
    sK, iK, jK = K_f.data, K_f.indices[:, 0], K_f.
        indices[:, 1]

    # Black-box solve: K_f u_f = f_f
    u_f_star = wrap_external_solve(sK, iK, jK, f_f)
    return u_f_star

def solve_fwd(params, problem_data):
    """Forward pass for the custom VJP."""
    u_f_star = solve(params, problem_data)
    residuals = (u_f_star, params, problem_data)
    return u_f_star, residuals

def solve_bwd(residuals, output_cotangent):
    """Backward pass (VJP) computed using the
        Implicit Function Theorem."""
    u_f_star, params, problem_data = residuals
    penal, rho = params

    # Compute K_f and form its transpose for the
        adjoint solve
    K_f = get_reduced_K(penal, rho, problem_data).
        transpose()
    sK, iK, jK  = K_f.data, K_f.indices[:, 0], K_f.
        indices[:, 1]

    # Adjoint solve: K_f^T lambda_ =
        output_cotangent
    lambda_f = wrap_external_solve(sK, iK, jK,
        output_cotangent)

    # Compute parial derivative (d_F/d_params)
        using JAX's automatic differentiation
    def F_partial(params):
        return F_stationarity(params, u_f_star,
            problem_data)

    _, vjp_fun = jax.vjp(F_partial, params)
    dparams = vjp_fun(lambda_f)[0]

    # Minus sign from the Implicit Function Theorem
    dparams = tuple(-dp for dp in dparams)
    return dparams, None

solve.defvjp(solve_fwd, solve_bwd)
```

$\omega_0 = 15.0$ (65.7) $\omega_0 = 150.0$ (74.5)

Fig. 9 Results for compliance minimization using a SIREN neural network. No density filter is applied, and all other settings are kept fixed except for the frequency parameter ω_0. As ω_0 increases, high-frequency oscillations emerge in the density field, leading to checkerboard-like patterns

Appendix C Other neural networks

C.1 Sinusoidal representation network (SIREN)

The SIREN architecture (Sitzmann et al. 2020) replaces the standard ReLU nonlinearity with a parametric sinusoidal activation of the form $\sin(\omega_0 z)$, where z denotes the output of a linear (dense) layer. Compared to standard multi-layer perceptrons (MLPs), SIRENs provide explicit control over the spatial frequency content of the representation through the frequency parameter ω_0. Larger values of ω_0 enable higher-frequency representations, corresponding to smaller minimum length scales. In the context of TO, this promotes fine-scale features and can lead to checkerboard-like patterns in the optimized designs when no density filtering is applied, as illustrated in Fig. 9. In addition to the activation function, SIRENs differ from conventional MLPs in two key aspects: (1) they employ a custom parameter initialization, implemented via `sine_init(shape, dtype=None, first=False)`, and (2) they omit batch normalization. Although the initialization remains random, its variance is explicitly scaled as a function of ω_0 to ensure stable signal propagation through the sinusoidal activations.

C.2 Convolutional neural network (CNN)

The third architecture is a CNN, originally designed for image processing tasks. We use a simplified version of the architecture proposed by Hoyer et al. (2019), where we treat the input to the network as non-trainable. The network takes

as input a random latent vector of size `latent_size`. This is passed through a fully connected layer whose output is reshaped into 32 low-resolution feature maps (each 1/8th the size of the final grid). Each subsequent hidden block performs the following sequence:

(1) `Activation('tanh')`: Applies a nonlinear transformation;
(2) `UpSampling2D`: Doubles the spatial resolution using bilinear interpolation;
(3) `LayerNormalization()`: Normalizes the intermediate values;
(4) `Conv2D`: Applies convolutional filters to extract spatial features.

This sequence is repeated five times to gradually refine the resolution. The final layer reshapes the output to match the target design grid dimensions.

Appendix D Extra results

We demonstrate how to modify existing boundary conditions as well as implement new ones in the code. Boundary conditions are implemented as separate functions that return the fixed and free degrees of freedom, as well as the load vector. Note that we follow the same node and DOF numbering convention as in the 88-line code, i.e., column-major ordering starting from the top-left node, except that numbering begins from 0. An example is shown in Code 13. The boundary condition function can be passed to `setup_fea_problem`, allowing it to be used consistently during the optimization process. The implementation relies on helper functions to select specific nodes (e.g., edges or corners) and convert them into the corresponding degrees of freedom. Some resulting designs are shown in Fig. 10, with the corresponding problem settings (resolution and target volume fraction) provided in the figure caption.

Code 13 Definition of the MBB boundary condition.

```
def mbb_bc(
    Nx: int, Ny: int, nodeNrs: np.
        ndarray, nDof: int
    ) -> Tuple[np.ndarray, np.ndarray,
        np.ndarray]:
    """
    MBB beam boundary conditions.

    Args:
    Nx: Number of elements in x-
        direction.
    Ny: Number of elements in y-
        direction.
    nodeNrs: Node numbering array (Ny+1
         x Nx+1).
    nDof: Total number of degrees of
        freedom.

    Returns:
    fixed: Array of fixed DOF indices.
    free: Array of free DOF indices.
    F: Load vector with applied forces.
    """
    # Fix left edge in x direction
    left_nodes = select_edge_nodes(
        nodeNrs, "left")
    fixed_x = nodes_to_dofs(left_nodes,
         "x")

    # Fix bottom-right corner in y
        direction
    br_node = select_corner_nodes(
        nodeNrs, "bottom-right")
    fixed_y = nodes_to_dofs(br_node, "y
        ")

    fixed = np.union1d(fixed_x, fixed_y
        )
    free = np.setdiff1d(np.arange(nDof)
        , fixed)

    # Load vector: downward unit load
        at bottom-right
    F = np.zeros(nDof)
    F[1] = -1.0

    return fixed, free, F
```

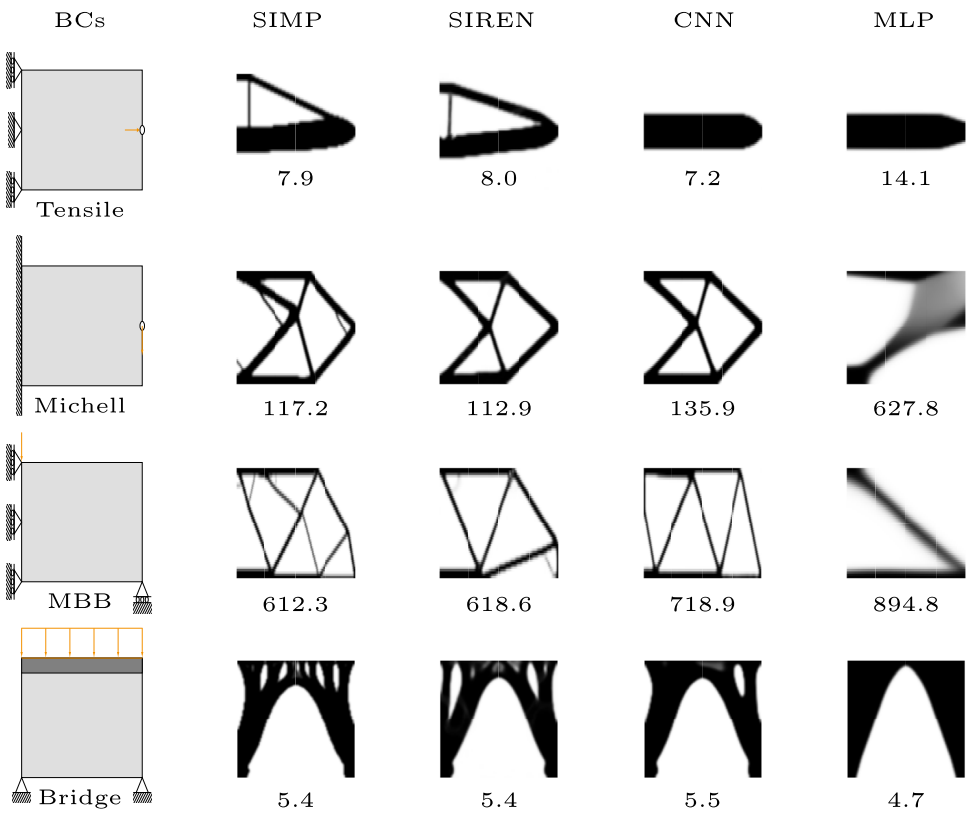

Fig. 10 Results for compliance minimization under different boundary conditions. From top to bottom: Tensile loading (128×64, 0.3), Michell cantilever (128×64, 0.3), MBB beam (144×48, 0.2), and Bridge (96×96, 0.5). In each case, the values in parentheses denote ($N_x \times N_y$, target volume fraction), where N_x and N_y are the numbers of elements in the horizontal and vertical directions, respectively. The reported compliance values (computed without thresholding to black and white) are shown below each design

Author Contributions Alejandro Marcos Aragón and Miguel Anibal Bessa contributed to the study's conception and design. Material preparation, data collection and analysis were performed by Suryanarayanan Manoj Sanu. The first draft of the manuscript was written by Suryanarayanan Manoj Sanu and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Data Availability Code for replication is available open-source at https://github.com/SNMS95/ADTO.git.

Declarations

Conflict of interest The authors declare no conflict of interest relevant to this article.

Replication of results The code accompanying this article is made openly available to facilitate replication and further exploration of the presented concepts. All implementations can be accessed via the associated GitHub repository, which includes annotated Jupyter notebooks and supporting Python modules. These notebooks can be executed directly in Google Colab without installation, allowing readers to reproduce all examples and experiment with different backends (`JAX` or `PyTorch`) to observe how automatic differentiation enables sensitivity analysis.

References

Aage N, Andreassen E, Lazarov BS (2014) Topology optimization using PETSc: an easy-to-use, fully parallel, open source topology optimization framework. Struct Multidisc Optim 51(3):565–572. https://doi.org/10.1007/s00158-014-1157-0

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viegas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu M, Zheng X (2015) TensorFlow: large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/softwareavailablefromtensorflow.org

Abdelhamid M, Czekanski A (2021) Revisiting non-convexity in topology optimization of compliance minimization problems. Eng Comput (Swans Wales). https://doi.org/10.1108/EC-01-2021-0052

Andreassen E, Clausen A, Lazarov SMBS, Sigmund O (2011) Efficient topology optimization in MATLAB using 88 lines of code. Struct Multidisc Optim 43:1–16. https://doi.org/10.1007/s00158-010-0594-7

Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhäuser Press, Boston, pp 163–202

Banga S, Gehani H, Bhilare S, Sagar P, Levent K (2018) 3D topology optimization using convolutional neural networks. arXiv preprint. arXiv:1808.07440

Baydin AG, Pearlmutter BA, Radul AA, Mark SJ (2017) Automatic differentiation in machine learning: a survey. J Mach Learn Res 18(1):5595–5637

Bendsøe MP, Sigmund O (2004) Topology optimization. Springer Berlin. https://doi.org/10.1007/978-3-662-05086-6

Blondel M, Roulet V (2024) The elements of differentiable programming. arXiv preprint. arXiv:2403.14606

Blondel M, Berthet Q, Cuturi M, Roy F, Stephan H, Felipe L-L, Fabian P, Jean-Philippe V (2022) Efficient and modular implicit differentiation. Adv Neural Inf Process Syst 35:5230–5242

Both C, Dehmamy N, Yu R, Albert-László B (2023) Accelerating network layouts using graph neural networks. Nat Commun 14(1):1560. https://doi.org/10.1038/s41467-023-37189-2

Bradbury J, Frostig R, Hawkins P, Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q (2018) JAX: composable transformations of Python + NumPy programs. http://github.com/jax-ml/jax

Chandrasekhar A, Suresh K (2020) TOuNN: topology optimization using neural networks. Struct Multidisc Optim 63(3):1135–1149. https://doi.org/10.1007/s00158-020-02748-4

Chandrasekhar A, Suresh K (2022) Approximate length scale filter in topology optimization using Fourier enhanced neural networks. Comput Aided Des. https://doi.org/10.1016/j.cad.2022.103277

Chandrasekhar A, Sridhara S, Suresh K (2021) AuTO: a framework for automatic differentiation in topology optimization. Struct Multidisc Optim 64(6):4355–4365. https://doi.org/10.1007/s00158-021-03025-8

Chi H, Zhang Y, Tang TLE, Mirabella L, Dalloro L, Song L, Paulino GH (2021) Universal machine learning for topology optimization. Comput Methods Appl Mech Eng. https://doi.org/10.1016/j.cma.2019.112739

Chollet F (2015) Keras. https://keras.io

Deng H, To AC (2020) Topology optimization based on deep representation learning (DRL) for compliance and stress-constrained design. Comput Mech 66(2):449–469. https://doi.org/10.1007/s00466-020-01859-5

Dilgen CB, Dilgen SB, Fuhrman DR, Sigmund O, Lazarov BS (2018) Topology optimization of turbulent flows. Comput Methods Appl Mech Eng 331:363–393. https://doi.org/10.1016/j.cma.2017.11.029

Doosti N, Panetta J, Babaei V (2021) Topology optimization via frequency tuning of neural design representations. In: Proceedings of the 6th annual ACM symposium on computational fabrication, SCF '2, 2021. Association for Computing Machinery, New York. https://doi.org/10.1145/3485114.3485124

Ferrari F, Sigmund O (2020) A new generation 99 line MATLAB code for compliance topology optimization and its extension to 3D. Struct Multidisc Optim 62(4):2211–2228. https://doi.org/10.1007/s00158-020-02629-w

Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT (2024) A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis. Struct Multidisc Optim. https://doi.org/10.1007/s00158-024-03792-0

Griewank A (2000) Evaluating derivatives: principles and techniques of algorithmic differentiation. Society for Industrial and Applied Mathematics, Philadelphia

Griewank A, Faure C (2002) Reduced functions, gradients and Hessians from fixed-point iterations for state equations. Numer Algorithms 30:113–139. https://doi.org/10.1023/A:1016051717120

Halle A, Campanile LF, Hasse A (2021) An artificial intelligence-assisted design method for topology optimization without pre-optimized training data. Appl Sci 11(19):9041. https://doi.org/10.3390/app11199041

Herrmann L, Sigmund O, Li VM, Vogl C, Kollmannsberger S (2024) On neural networks for generating better local optima in topology optimization. Struct Multidisc Optim. https://doi.org/10.1007/s00158-024-03908-6

Hill A, Dalle G, Montoison A (2025) An illustrated guide to automatic sparse differentiation. In: ICLR blogposts 2025, 2025. https://iclr-blogposts.github.io/2025/blog/sparse-autodiff/, https://iclr-blogposts.github.io/2025/blog/sparse-autodiff/

Hoyer S, Sohl-Dickstein J, Greydanus S (2019) Neural reparameterization improves structural optimization. In: NeurIPS 2019 workshop on solving inverse problems with deep networks, 2019

Jeong H, Batuwatta-Gamage C, Bai J, Xie YM, Rathnayaka C, Zhou Y, Gu YT (2023) A complete physics-informed neural network-based framework for structural topology optimization. Comput Methods Appl Mech Eng 417:116401. https://doi.org/10.1016/j.cma.2023.116401

Jia Y, Wang C, Zhang XS (2024) FEniTop: a simple FEniCSx implementation for 2D and 3D topology optimization supporting parallel computing. Struct Multidisc Optim. https://doi.org/10.1007/s00158-024-03818-7

Joglekar A, Chen H, Kara LB (2023) DMF-TONN: direct mesh-free topology optimization using neural networks. Eng Comput 40(4):2227–2240. https://doi.org/10.1007/s00366-023-01904-w

Kallioras NA, Kazakis G, Lagaros ND (2020) Accelerated topology optimization by means of deep learning. Struct Multidisc Optim 62:1185–1212. https://doi.org/10.1007/S00158-020-02545-Z/FIGURES/25

Karniadakis GE, Kevrekidis IG, Lu L, Perdikaris P, Wang S, Yang L (2021) Physics-informed machine learning Nat Rev Phys 3(6):422–440. https://doi.org/10.1038/s42254-021-00314-5

Keshavarzzadeh V, Kirby RM, Narayan A (2021) Robust topology optimization with low rank approximation using artificial neural networks. Comput Mech 68(6):1297–1323. https://doi.org/10.1007/s00466-021-02069-3

Kingma DP, Ba J (2015) Adam: a method for stochastic optimization. In: Bengio Y, LeCun Y (eds) 3rd International conference on learning representations, ICLR 2015, conference track proceedings, San Diego, CA, USA, 7–9 May 2015. http://arxiv.org/abs/1412.6980

Krantz SG, Parks HR (2013) The implicit function theorem: history, theory, and applications. Springer New York. https://doi.org/10.1007/978-1-4614-5981-1

Lee S, Kim H, Lieu QX, Lee J (2020) CNN-based image recognition for topology optimization. Knowl-Based Syst 198:105887. https://doi.org/10.1016/j.knosys.2020.105887

Margossian CC (2019) A review of automatic differentiation and its efficient implementation. WIREs Data Min Knowl Discov. https://doi.org/10.1002/widm.1305

Martins JRRA, Ning A (2022) Engineering design optimization. Cambridge University Press Cambridge. https://doi.org/10.1017/9781108980647

Nocedal J, Wright SJ (2006) Numerical optimization. Springer New York. https://doi.org/10.1007/978-0-387-40065-5

Nørgaard SA, Sagebaum M, Gauger NR, Lazarov BS (2017) Applications of automatic differentiation in topology optimization. Struct Multidisc Optim 56(5):1135–1146. https://doi.org/10.1007/s00158-017-1708-2

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B,

Fang L, Bai J, Chintala S (2019) PyTorch: an imperative style, high-performance deep learning library. In: Advances in neural information processing systems, vol 32. Curran Associates, Inc., pp 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Qian C, Ye W (2021) Accelerating gradient-based topology optimization design with dual-model artificial neural networks. Struct Multidisc Optim 63:1687–1707. https://doi.org/10.1007/S00158-020-02770-6/TABLES/7

Rokicki J (2016) Adjoint lattice Boltzmann for topology optimization on multi-GPU architecture. Comput Math Appl 71(3):833–848

Sanu SM, Aragon AM, Bessa MA (2025) Neural topology optimization: the good, the bad, and the ugly. Struct Multidisc Optim 68(10):1–26

Scieur D, Gidel G, Bertrand Q, Pedregosa F (2022) The curse of unrolling: rate of differentiating through optimization. Adv Neural Inf Process Syst 35:17133–17145

Sigmund O (2011) On the usefulness of non-gradient approaches in topology optimization. Struct Multidisc Optim 43(5):589–596. https://doi.org/10.1007/s00158-011-0638-7

Sitzmann V, Martel JNP, Bergman AW, Lindell DB, Wetzstein G (2020) Implicit neural representations with periodic activation functions. In: Advances in neural information processing systems, December 2020. https://doi.org/10.48550/arxiv.2006.09661

Sosnovik I, Oseledets I (2019) Neural networks for topology optimization. Russ J Numer Anal Math Model 34(4):215–223

Sun X, Yousefpour A, Hosseinmardi S, Bostanabad R (2025) Compliance minimization via physics-informed Gaussian processes. Struct Multidisc Optim. https://doi.org/10.1007/s00158-025-04179-5

van Keulen F, Haftka R, Kim N (2005) Review of options for structural design sensitivity analysis. Part 1: linear systems. Comput Methods Appl Mech Eng 194(30):3213–3243. https://doi.org/10.1016/j.cma.2005.02.002

Vijayakumaran H, Russ JB, Paulino GH, Bessa MA (2025) Consistent machine learning for topology optimization with microstructure-dependent neural network material models. J Mech Phys Solids. https://doi.org/10.1016/j.jmps.2024.106015

White DA, Arrighi WJ, Kudo J, Watts SE (2019) Multiscale topology optimization using neural network surrogate models. Comput Methods Appl Mech Eng 346:1118–1135. https://doi.org/10.1016/j.cma.2018.09.007

Woldseth RV, Aage N, Bærentzen JA, Sigmund O (2022) On the use of artificial neural networks in topology optimisation. Struct Multidisc Optim. https://doi.org/10.1007/s00158-022-03347-1

Wolfram Research, Inc. (2025) Mathematica 8.0. Wolfram Research, Inc. https://www.wolfram.com

Wu G (2023) A framework for structural shape optimization based on automatic differentiation, the adjoint method and accelerated linear algebra. Struct Multidisc Optim. https://doi.org/10.1007/s00158-023-03601-0

Xu S, Cai Y, Cheng G (2009) Volume preserving nonlinear density filter based on Heaviside functions. Struct Multidisc Optim 41(4):495–505. https://doi.org/10.1007/s00158-009-0452-7

Xue L, Liu J, Wen G, Wang H (2021) Efficient, high-resolution topology optimization method based on convolutional neural networks. Front Mech Eng 16(1):80–96. https://doi.org/10.1007/S11465-020-0614-2

Xue T, Liao S, Gan Z, Park C, Xie X, Liu WK, Cao J (2023) JAX-FEM: a differentiable GPU-accelerated 3D finite element solver for automatic inverse design and mechanistic data science. Comput Phys Commun 291:108802. https://doi.org/10.1016/j.cpc.2023.108802

Zehnder J, Li Y, Coros S, Thomaszewski B (2021a) NTopo: mesh-free topology optimization using implicit neural representations. Adv Neural Inf Process Syst 13:10368–10381. https://doi.org/10.48550/arxiv.2102.10782

Zehnder J, Li Y, Coros S, Thomaszewski B (2021b) NTopo: mesh-free topology optimization using implicit neural representations. In: Ranzato M, Beygelzimer A, Dauphin Y et al (eds) Advances in neural information processing systems, 2021, vol 34. Curran Associates, Inc., pp 10368–10381. https://proceedings.neurips.cc/paper_files/paper/2021/file/55d99a37b2e1badba7c8df4ccd506a88-Paper.pdf

Zhang Z, Li Y, Zhou W, Chen X, Yao W, Zhao Y (2021) TONR: an exploration for a novel way combining neural network with topology optimization. Comput Methods Appl Mech Eng 386:114083. https://doi.org/10.1016/j.cma.2021.114083