Expressing general constitutive models in FEniCSx using external operators and algorithmic automatic differentiation

Andrey Latyshev, Jérémy Bleyer, Corrado Maurini, Jack S Hale

OVERLAY

Journal of Theoretical, Computational and Applied Mechanics

JTCAM

DIAMOND OPEN ACCESS

Identifiers
DOI 10.46298/jtcam.14449
HAL hal-04735022v3

History
Received Oct 14, 2024
Accepted Mar 27, 2025
Published Sep 22, 2025

Associate Editor
Alexander Popp

Reviewers
Thomas Helfer
Chris Richardson
Anonymous

Open Review
HAL hal-05082559

Supplementary Material
DOI 10.5281/zenodo.15577842.
See also addendum for additional information on datasets and software solutions.

Expressing general constitutive models in FEniCSx using external operators and algorithmic automatic differentiation

Andrey Latyshev[1,2], Jérémy Bleyer[3], Corrado Maurini[2], and Jack S. Hale[1]

[1] Institute of Computational Engineering, Department of Engineering, Faculty of Science, Technology and Medicine, Université du Luxembourg, Luxembourg
[2] Institut Jean Le Rond d'Alembert, Sorbonne Université, UMR CNRS 7190, France
[3] Laboratoire Navier, École des Ponts ParisTech, Université Gustave Eiffel, UMR CNRS 8205, France

Many problems in solid mechanics involve general and non-trivial constitutive models that are difficult to express in variational form. Consequently, it can be challenging to define these problems in automated finite element solvers, such as the FEniCS Project, that use domain-specific languages specifically designed for writing variational forms. In this article, we describe a methodology and software framework for FEniCSx / DOLFINx that enables the expression of constitutive models in nearly any general programming language. We demonstrate our approach on two solid mechanics problems; the first is a simple von Mises elastoplastic model with isotropic hardening implemented with Numba, and the second a Mohr-Coulomb elastoplastic model with apex smoothing implemented with JAX. In the latter case we show that by leveraging JAX's algorithmic automatic differentiation transformations we can avoid error-prone manual differentiation of the terms necessary to resolve the constitutive model. We show extensive numerical results, including Taylor remainder testing, that verify the correctness of our implementation. The software framework and fully documented examples are available as supplementary material under the LGPLv3 or later license.

Keywords: constitutive models, automated finite element solvers, algorithmic automatic differentiation, external operators, FEniCSx, JAX, Numba

1 Introduction

The finite element method (FEM) has proven itself as a robust numerical method for solving partial differential equations (PDEs) arising from problems in solid mechanics (Bucalem and Bathe 2011). In the past fifteen years, automated finite element solvers, such as FEniCS(x) (M. Alnæs et al. 2015; Baratta et al. 2023), FreeFEM++ (Hecht 2012) and Firedrake (Ham et al. 2023), have introduced high-level domain-specific languages (DSLs) specifically designed for writing finite element variational forms of PDEs, e.g. the Unified Form Language (UFL) (M. S. Alnæs et al. 2014). Through a sequence of analysis and code transformation steps (Kirby and Logg 2006; Logg et al. 2012; Homolya et al. 2018), automated finite element solvers can automatically translate the DSL specification of a particular problem into a high-performance finite element solver.

In the field of solid mechanics, a wide variety of constitutive models are used to predict the behaviour of materials in response to mechanical loads. As a non-exhaustive list of examples, we mention elasto-plastic (Simo and Hughes 1998), hyper-elastic (Ogden 1997), poro-elastic (Coussy 2004; Wang and Hong 2012) and visco-elastic (Ferry 1980) material behaviours, for which constitutive models can be derived from a variety of phenomenological (Tschoegl 2012), statistical (Buche and Silberstein 2020), thermodynamic (Rajagopal and Srinivasa 2000), and more recently, data-centric justifications (Fuhg et al. 2025).

However, a significant number of widely used constitutive models either cannot be easily expressed in variational form, or, even if it is possible, the practical resolution of the constitutive

law on a computer is more easily expressed as an algorithm implemented in a computationally universal programming language. Consequently, domain specific languages such as UFL are often not rich enough to express the algorithms associated with the wide range of constitutive models in use today. Typical examples of such complex constitutive models that cannot be, or are not ideally, expressible in UFL include plasticity models, e.g. (Simo and Hughes 1998), multiscale models, e.g. (Feyel 2003) and modern data-centric models, e.g. (Stainier et al. 2019; Masi et al. 2021; Thakolkaran et al. 2022; Zlatić et al. 2024; Ulloa et al. 2024), with the latter being of particular interest due to their increased popularity in the computational mechanics community in recent years.

Plasticity models are often resolved via algorithms such as the well-known predictor-corrector scheme which involves the application of a Newton-type method at the local level (Simo and Hughes 1998). Multiscale constitutive models are frequently resolved by executing another numerical algorithm, e.g. a molecular dynamics simulation (Saether et al. 2009), or another finite element code (Feyel 2003), to represent the material behaviour at a lower scale. Data-centric methods often include non-trivial model structures, e.g. neural networks (Linka and Kuhl 2023; Zhang et al. 2022), for which high-quality libraries exist for their concise expression, training on data and execution (Abadi et al. 2015; Frostig et al. 2018). Although diverse, none of these methods can be naturally expressed in DSLs such as UFL.

1.1 Methods for incorporating constitutive models

A number of methods have been proposed to incorporate general constitutive models into automated finite element solvers such as DOLFINx. One method is to write programmatic interfaces from the finite element solver to frameworks specifically designed for implementing non-standard constitutive models such as MFront (Helfer et al. 2015) and ZMAT (Abatour et al. 2024; Z-Set Software 2023). Although this approach does not require any changes to the existing functionality of UFL, it requires significant effort to develop interfaces between the finite element solver and the interface provided by each constitutive modelling package. Furthermore, programmatic approaches often feel *ad hoc* as they do not integrate with the abstractions and automatic differentiation tools provided by UFL to write the variational part of the problem.

A more recent approach, and one that forms the basis for this work, is the introduction of the *external operator* extension to UFL (Bouziani and Ham 2021). The external operator is a symbolic UFL object that represents a general mapping between finite element quantities in a form; for a formal definition, see (Bouziani and Ham 2021, Definition 1). The action of the operator itself can then be concretely defined externally, using 'any' programming language. In essence, the external operator concept creates a bridge between the language of forms and the broader possibilities provided by general programming languages. Another key feature of external operators is that they can be symbolically differentiated by UFL, producing new symbolic external operators representing the action of the derivative of the original operator - this opens up numerous possibilities for using programming languages that support *algorithmic automatic differentiation* (AD) to automatically create the necessary derivatives.

1.2 Automatic differentiation in constitutive modelling

In solid mechanics the resolution of a constitutive model often requires the evaluation of derivatives of lengthy expressions, for example, the derivative of the stress field, so-called consistent tangent moduli. Due to the complexity of these expressions, their derivation by hand and subsequent translation into programming code is often a source of errors. Because of this, different approaches can be used to automate and/or approximate the differentiation process, e.g. finite differences (FD), complex-step (CS) (Lyness and Moler 1967; Lyness 1968), symbolic differentiation (SD) and algorithmic automatic differentiation (AD) (Griewank and Walther 2008).

We remark that UFL implements Gâteaux differentiation of forms, including external operators, and the output of this is another UFL form. Hence UFL's differentiation can be considered a type of SD (M. S. Alnæs et al. 2014, p.32).

In this work, we focus on leveraging algorithmic automatic differentiation for the concise expression of constitutive models and their derivatives. AD is a set of techniques for evaluating

derivatives of quantities defined by general computer programs (Griewank and Walther 2008). Compared with other differentiation techniques like FD and SD, AD does not suffer from truncation or round-off errors, does not lead to expression swell and can be applied to general classes of programs involving control flow statements such as loops and if statements. Together, these desirable properties allow the application of AD to a wide range of constitutive models including the ones that cannot be expressed through closed-form symbolic expressions (Brothers et al. 2014; Tanaka et al. 2016; Dummer et al. 2024).

AD techniques have already been applied successfully to constitutive modelling in solid mechanics. For instance, Rothe and Hartmann (2015) used AD to define finite strain hyperelasticity, finite strain elasto-visco-plasticity and fully coupled small strain thermo-visco-plasticity, while Vigliotti and Auricchio (2021) considered a similar hyperelastic constitutive model, but for a problem with specific constraints. Seidl and Granzow (2022) and Q. Chen et al. (2014) focused on AD application to finite strain elasto-plasticity, where the former additionally considered the hyperelasto-plastic model within the large deformation framework (Q. Chen et al. 2014). Dummer et al. (2024) also addressed the hyperelasto-plastic model. Blühdorn et al. (2022) leveraged AD techniques within the constitutive laws of generalized standard materials (GSM) specifically using an elasto-visco-plastic model as a case study. Lindsay et al. (2021) examined the role of AD in several multiphysics simulations, such as laser melt pools, phase-field modelling with neural network-generated free energies, and simulations of metallic nuclear fuel.

Across these works the value of AD tools for implementing constitutive models is evident. A key benefit is that AD eliminates the need for manual derivation of tangent operators and stiffness matrices, which significantly reduces the potential for human error and speeds up development time, without sacrificing accuracy (Rothe and Hartmann 2015; Lindsay et al. 2021). Another side benefit of some commonly used AD tools is that they can emit code that is well-suited for parallel computation on Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) (Blühdorn et al. 2022). In summary, AD has proven to be a useful tool in implementing constitutive models, enhancing both computational efficiency and the accuracy of derivative quantities while minimizing much of the manual effort usually required.

1.3 Existing software

Because of the benefits outlined above, it is natural that there are already contributions that integrate AD techniques into software for solving PDEs. One of the pioneers is the AceFEM library (Korelc and Wriggers 2016) which is based on the commercial software system Mathematica. AceFEM is equipped with the automatic code generation package AceGEN (Korelc 1997; Korelc and Wriggers 2016), where AD is used to reduce the need for manual linearization of nonlinear models. The derivatives obtained through AD are then employed to automatically generate efficient code for the finite element residual vector and stiffness matrix. Similarly, a recent work (Lindsay et al. 2021) has enhanced the MOOSE multiphysics library by using AD to compute the global finite element Jacobian matrix. The software package JAX-FEM (Xue et al. 2023) takes the use of AD further by applying JAX (Frostig et al. 2018) to the entire finite element solution algorithm, allowing for 'end-to-end' differentiation of the finite element solver.

In addition to AD-based software, various other projects have enabled the use of constitutive models by extending the functionality of existing FEM packages. For example, the fenics-solid-mechanics project (Ølgaard and Garth 2017) was specifically designed to solve Drucker-Prager and von Mises plasticity problems from the C++ interface to DOLFIN. The convex-plasticity project (Latyshev and Bleyer 2022) uses the CVXPY package (Diamond and Boyd 2016) to solve plasticity problems in a convex optimization framework within FEniCSx. However, these projects are limited in scope, as they require each constitutive model to be explicitly adapted to the finite element solver.

Another approach to implementing constitutive models in finite element environments is to develop specialized interfaces. This idea has been realized in software such as MGIS (Helfer et al. 2020), dolfinx_materials (Bleyer 2024a) and fenicsx_constitutive (Rosenbusch et al. 2024). MGIS provides an interface that is used by MFront to define constitutive laws, generate code and provide this code to other finite element solvers, e.g. within the legacy version of the FEniCS

library. Similarly, dolfinx_materials allows for the definition of sophisticated material behaviours that cannot be expressed using standard UFL operators. It does so by leveraging AD from the JAX package (Frostig et al. 2018) and utilizing the convex optimization framework from CVXPY (Diamond and Boyd 2016) to define constitutive models. fenicsx_constitutive focuses on linking DOLFINx with constitutive models written in languages such as C, C++, Rust and Fortran that can accept data as contiguous arrays, and also existing constitutive models written to well-established interfaces such as ABAQUS UMAT (Lucarini and Martínez-Pañeda 2024). MGIS, dolfinx_materials and fenicsx_constitutive all require users to adapt their constitutive models to the specific interfaces provided by these libraries. Furthermore, they necessitate the development of additional interfaces to be compatible with other finite element environments.

1.4 Contributions

This article describes two main contributions. Firstly, we describe a methodology and software framework that extends the open source library DOLFINx to support the recently introduced symbolic external operator in UFL (Bouziani and Ham 2021). In turn, this provides DOLFINx/FEniCSx users with an interface that allows the definition of general constitutive models via a wide variety of programming languages. As a didactic example, we show the implementation of von Mises with isotropic hardening (Bonnet et al. 2014) using Numba, a high-performance Python compiler (Lam et al. 2015). Building on this software framework, our second main contribution is to explore the use of programming languages with algorithmic automatic differentiation capabilities for expressing constitutive models. To demonstrate this we implement a Mohr-Coulomb elastoplastic model with apex smoothing (Abbo and Sloan 1995) that has previously been implemented using MFront (Helfer et al. 2015). By leveraging JAX (Frostig et al. 2018), a Python library for high-performance array computations with composable transformations for automatic differentiation, we completely avoid both the manual expression and implementation of the necessary derivatives. The resulting implementation of the complete Mohr-Coulomb finite element solver in DOLFINx is remarkably compact and its correctness is verified via a Taylor remainder test and against an existing solution from the literature.

The software framework is openly available as supplementary material (Latyshev and Hale 2024) and includes further fully documented examples. We also note that an earlier version of this work on implementing external operators in DOLFINx by the same authors was published in conference proceedings (Latyshev et al. 2024).

Outline An outline of this article is as follows. In Section 2 we introduce the general plasticity problem which serves as a model problem, and discuss some of the difficulties with implementation in the FEniCSx environment. In Section 3 we introduce the design and API of our framework extending DOLFINx. Then, in Section 4 we show two applications of the framework to von Mises and Mohr-Coulomb plasticity, with the latter leveraging the automatic differentiation features of JAX, before closing with some conclusions and remarks in Section 5.

2 General formulation of a plasticity problem

In this section, we introduce a general formulation of plasticity as a model problem. We assume the reader is relatively familiar with the basic concepts of plasticity theory and the return-mapping algorithm; for a full treatment of plasticity, we refer the reader to the classic references (Simo and Hughes 1998; Bonnet et al. 2014). We then describe the challenges of implementing this class of models in FEniCSx, motivating the extension that we describe in the following section.

Although we cover only one type of constitutive model in this paper - small strain elastoplasticity - it is rich enough to demonstrate the current limitations of automated finite element environments, e.g. FEniCSx, and the advantages of our proposed methodology. The same workflow can be applied to other types of general constitutive models such as those discussed in the introduction and conclusion.

2.1 Notation

Let $\varepsilon(\boldsymbol{u}) = \frac{1}{2}(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^\top)$ be the small strain tensor of the displacement field $\boldsymbol{u}$ in Cartesian coordinate system $\boldsymbol{x} = (x, y, z)$. Throughout the article we utilise Mandel notation (Mandel 1965) for the stress σ and strain ε rank-two symmetric tensors to represent them as vectors $\boldsymbol{\sigma}$ and $\boldsymbol{\varepsilon}$ respectively as follows in a three-dimensional case

$$\boldsymbol{\sigma} = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sqrt{2}\sigma_{xy}, \sqrt{2}\sigma_{xz}, \sqrt{2}\sigma_{yz})^\top, \tag{1}$$

$$\boldsymbol{\varepsilon} = (\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \sqrt{2}\varepsilon_{xy}, \sqrt{2}\varepsilon_{xz}, \sqrt{2}\varepsilon_{yz})^\top. \tag{2}$$

The factor $\sqrt{2}$ is added to the shear components in Equations (1) and (2) to make the following inner products consistent

$$\sigma : \varepsilon = \boldsymbol{\sigma} \cdot \boldsymbol{\varepsilon}, \quad e : e = \boldsymbol{e} \cdot \boldsymbol{e}, \quad s : s = \boldsymbol{s} \cdot \boldsymbol{s}, \tag{3}$$

where $\boldsymbol{e}$ and $\boldsymbol{s}$ are the Mandel vectors of respectively deviatoric parts of the strain and stress tensors $e = \operatorname{dev} \varepsilon$ and $s = \operatorname{dev} \sigma$.

2.2 Model problem

In order to define the model problem, we make several standard assumptions from plasticity theory. First of all, we assume that the total strains are additively decomposed onto elastic $\boldsymbol{\varepsilon}^e$ and plastic $\boldsymbol{\varepsilon}^p$ parts: $\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}^e + \boldsymbol{\varepsilon}^p$. Then, by introducing the loading parameter t, the quasi-static evolution of Hooke's law for stress and strain has the following form

$$\dot{\boldsymbol{\sigma}} = \boldsymbol{C} \cdot \dot{\boldsymbol{\varepsilon}}^e = \boldsymbol{C} \cdot (\dot{\boldsymbol{\varepsilon}} - \dot{\boldsymbol{\varepsilon}}^p), \tag{4}$$

where $\boldsymbol{C}$ is the stiffness matrix and the dot above a symbol denotes a derivative with respect to the loading parameter t.

With f being the yield function and p representing an ensemble of internal variables, the plastic flow is governed by the yield criterion

$$f(\boldsymbol{\sigma}, p) \leqslant 0, \tag{5}$$

and the flow rule, connecting the plastic strain rate with the gradient of the plastic potential g

$$\dot{\boldsymbol{\varepsilon}}^p = \dot{\lambda} \frac{\partial g(\boldsymbol{\sigma}, p)}{\partial \boldsymbol{\sigma}}, \tag{6}$$

where $\dot{\lambda} > 0$ is the plastic multiplier (Simo and Hughes 1998). If $g \neq f$ in Equation (6), then we consider a general case of the non-associative plastic flow rule, otherwise, we refer to this equation as the associative one.

In this work, we limit ourselves to the case of isotropic hardening with a scalar internal variable $\dot{p} := (\frac{2}{3}\dot{\boldsymbol{e}}^p \cdot \dot{\boldsymbol{e}}^p)^{1/2}$ representing cumulative plastic strain. Thus, the hardening law has the following form

$$\dot{p} = -\dot{\lambda} q(\boldsymbol{\sigma}, p), \tag{7}$$

where q is a hardening function.

Throughout the loading of solid, the elastic deformations occur at those points, where the stress field strictly satisfies the Yield criterion (5), whereas the plastic flow starts only on the yield surface

$$f(\boldsymbol{\sigma}, p) = 0. \tag{8}$$

In short, the loading/unloading conditions can be written as follows (Simo and Hughes 1998)

$$\dot{\lambda} \geqslant 0, \quad f(\boldsymbol{\sigma}, p) \leqslant 0, \quad \dot{\lambda} \cdot f(\boldsymbol{\sigma}, p) = 0. \tag{9}$$

By applying the backward Euler scheme to Equations (4), (6) and (7) for the plastic regime at a certain point of the solid, we obtain the following constitutive equations

$$\boldsymbol{r}_{\boldsymbol{\sigma}}(\boldsymbol{\sigma}_{n+1}, \Delta\lambda) := \boldsymbol{\sigma}_{n+1} - \boldsymbol{\sigma}_n - \boldsymbol{C} \cdot \left(\Delta\boldsymbol{\varepsilon} - \Delta\lambda \frac{\partial g}{\partial \sigma}(\boldsymbol{\sigma}_{n+1}, p_{n+1})\right) = \boldsymbol{0}, \tag{10}$$

$$r_f(\boldsymbol{\sigma}_{n+1}) := f(\boldsymbol{\sigma}_{n+1}, p_{n+1}) = 0, \tag{11}$$

where Δ is associated with increments of a quantity between the next loading step $n+1$ and the current loading step n. This notation is applied to all quantities of interest throughout the text, wherever it is necessary. We note that due to the case of isotropic hardening, the internal variable p_{n+1} is excluded as it can be expressed through $\Delta\lambda$ from Equation (7).

For the elastic regime, we consider a trivial system of equations

$$\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_n + \boldsymbol{C} \cdot \Delta\boldsymbol{\varepsilon}, \tag{12}$$

$$\Delta\lambda = 0. \tag{13}$$

By introducing the residual vector $\boldsymbol{r} := [\boldsymbol{r}_{\boldsymbol{\sigma}}^\top, r_f]^\top$ and the vector of unknowns $\boldsymbol{y} := [\boldsymbol{\sigma}_{n+1}^\top, \Delta\lambda]^\top$, the Constitutive equations (10) and Equation (11) can be re-written as a nonlinear problem

$$\boldsymbol{r}(\boldsymbol{y}) = \boldsymbol{0}, \tag{14}$$

which is solved at those points of the solid, where the Yield surface (8) is reached. For a given spatial point of the solid, this nonlinear equation does not depend on the stress states at other points and so can be solved *locally*. From now on, we reference to Equation (14) as the *local* problem for a given point of the solid.

In order to solve Local problem (14), the Newton method is commonly used, which requires a consistent linearization of the residual $\boldsymbol{r}$

$$\boldsymbol{j}(\boldsymbol{y}) = \frac{\mathrm{d}\boldsymbol{r}(\boldsymbol{y})}{\mathrm{d}\boldsymbol{y}}, \tag{15}$$

where we call the matrix $\boldsymbol{j}$ the *local* Jacobian and the associated Newton method the *local* Newton method.

At this point, we have all the necessary ingredients to formulate the *global* problem. Let Ω be a domain representing a solid body, which is loaded by an external force on a part of its boundary $\partial\Omega_N$ with outward facing normal $\boldsymbol{n}$, and V be a space of admissible displacements. The equilibrium state of the solid body is described by the following variational problem: find the displacement field $\boldsymbol{u} \in V$ such that the following weak residual equation is satisfied

$$F(\boldsymbol{u}; \boldsymbol{v}) = \int_\Omega \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u})) \cdot \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}\boldsymbol{x} - F_{\text{ext}}(\boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V, \tag{16}$$

together with the Yield criterion (5). The linear functional F_{ext} in Equation (16) represents the external loading from e.g. a Neumann boundary condition or gravitational body force (see Section 4). By using the semicolon symbol among arguments of the semi-linear form $F = F(\boldsymbol{u}; \boldsymbol{v})$, we separate the argument $\boldsymbol{u}$, with respect to which the form may be nonlinear, from the argument $\boldsymbol{v}$, with respect to which F is linear.

The stress-strain relation $\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$ is a nonlinear function, which can implicitly depend on the history of loading and a set of internal variable(s). In this regard, we apply another Newton method to solve nonlinear Equation (16), which, similar to the local problem, requires the linearization of the residual. For this matter, we introduce the tangent operator $\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$ or tangent moduli, the derivative of the stress tensor

$$\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u})) = \frac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}}, \tag{17}$$

which is used to define the Gâteaux derivative of the form F in the direction $\hat{\boldsymbol{u}} \in V$

$$J(\boldsymbol{u}; \hat{\boldsymbol{u}}, \boldsymbol{v}) = \int_\Omega \left(\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u})) \cdot \boldsymbol{\varepsilon}(\hat{\boldsymbol{u}})\right) \cdot \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}\boldsymbol{x}. \tag{18}$$

We reference to this Gâteaux derivative J as the *global* Jacobian and to the Newton method, where J is used in the role of a consistent linearization of the residual F, as a *global* Newton method.

To satisfy Yield criterion (5) the *return-mapping procedure* is commonly used (Simo and Hughes 1998). The procedure consists in iterating over stress-strain states through a predictor-corrector scheme until the equilibrium of Constitutive equations (14) is reached, which is achieved by applying the local Newton method. The result of the return-mapping procedure is the evaluated values of the stress $\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$ at a given point satisfying Equation (5).

Summarizing, the model problem consists of Global problem (16) with Constraint (5) and Local problem (14). On each loading step, we solve Equation (16) by applying the global Newton method. At each iteration of the global Newton method, we satisfy Equation (5) by applying the return-mapping procedure, which involves solving Equation (14) through the local Newton method. Thus, the implementation of the model problem includes two nested Newton methods, which oblige us to compute the Local Jacobian (15) and Global Jacobian (18).

Before continuing we make some specific remarks about implementing general plasticity models in the current version of DOLFINx. To be able to evaluate the values of the operator $\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$ and assemble the form F from Global problem (16) in DOLFINx, we have to solve Local problem (14) which requires an implementation of the return-mapping procedure. The return-mapping procedure is an iterative algorithm that involves a sequence of computations where each step depends on the previous one. It is most natural to express this type of recursion within either a procedural (common) or functional (less common) programming paradigm, rather by a closed-form mathematical formula in UFL. The same argument applies to the tangent operator $\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$. This leads us to the conclusion that the stress operator $\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$ is best implemented 'externally' to UFL. However, because of this externally defined behaviour, UFL cannot be aware of the external operators derivative, and so we cannot apply UFL's SD tools to the form F to derive the Jacobian J containing $\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$. As we will see in the next section, the proposed framework elegantly deals with these difficulties, by allowing the predictor-corrector algorithm to govern the behaviour of the stress operator, while also enabling the stress operator to be naturally expressed within the variational setting and differentiated using UFL.

3 Extension of the external operator concept to DOLFINx

This section discusses the new data-centric design of DOLFINx (Baratta et al. 2023) and the automatic code generation feature of FFCx (Kirby and Logg 2006; Logg et al. 2012). We then describe how our framework exploits these concepts as well as external operators to extend the functionality of DOLFINx to cover a wider range of constitutive modelling possibilities.

Data-centric design of DOLFINx One of the new developments of the DOLFINx library is the data-centric design (Baratta et al. 2023), where data such as values of a finite element function at finite element degrees of freedom, or quadrature points, are directly available in the form of array-like data structures (e.g. the `ndarray` object of the NumPy package). This data-centric design gives the external operators straightforward access to solver data for manipulation via external programming languages that support the same array-like data structures (e.g. Numba, JAX, PyTorch, MFront, etc).

Automatic code generation of FFCx Another important development used in this work is the automatic code generation feature of the FEniCSx Form Compiler (FFCx) (Kirby and Logg 2006; Logg et al. 2012) for UFL Expressions. For more details, see (Baratta et al. 2023). In the context of this study, FFCx is used to generate code that allows DOLFINx to evaluate the values of the strain tensor $\boldsymbol{\varepsilon}(\boldsymbol{u})$ at the quadrature points of the mesh. This can then be passed as data to user-defined Python callables that define the action of the external operator as discussed in the previous paragraph.

Leveraging the aforementioned features of DOLFINx and FFCx as well as the concept of external operators (Bouziani and Ham 2021), we implement a framework that extends DOLFINx allowing for the expression of a large class of constitutive models. In our implementation of the external operator concept there is no direct interaction between the DOLFINx finite element assemblers and external operators - in contrast with the Firedrake implementation (Ham et al.

2023). Instead, we replace symbolic UFL `ExternalOperator` objects prior to assembly of the finite element form with assembled DOLFINx `Function` objects containing the evaluation of the external operators. As the DOLFINx assembler already works with forms containing the `Function` objects this approach required no changes upstream in the DOLFINx project and is therefore also compatible with third-party custom assemblers. Thus, according to our contribution, an external operator combines properties of both the symbolic UFL's `ExternalOperator` and its data-oriented counterpart the DOLFINx's `Function`, which results in the introduction of the new class `FEMExternalOperator` inheriting `ExternalOperator`.

`FEMExternalOperator` has two principal features. The first is that the `FEMExternalOperator` inherits the symbolic functionality of the UFL `ExternalOperator` so that we can also apply the UFL's SD capabilities to the forms containing the `FEMExternalOperator` objects. The second feature allows the evaluation of the `FEMExternalOperator` at interpolation points of a finite element space in contrast to the `ExternalOperator` object resulting in computed values being stored in a `Function` object through `ndarray`. The evaluation is defined by the user through Python callables (functions) that specify how an external operator (e.g. stress) acts on its operands (e.g. strains).

The data-centric design of DOLFINx enables the efficient transfer of data to and from the user-defined functions evaluating the external operators (e.g. $\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(u))$ & $\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(u))$), UFL expressions of their operands (e.g. $\boldsymbol{\varepsilon}(u)$), and the DOLFINx environment through standard NumPy arrays. The framework evaluates the values of operands using code automatically generated by FFCx, stores the result in an `ndarray` and passes this data to the user-defined callables. This design allows a wide range of Python and non-Python languages and frameworks to be used for implementing the external operators. In practice, almost all modern programming languages and frameworks can be passed, without copy, the contiguous array-like data underlying a Python `ndarray`, and there is increasing standardisation efforts around in-memory formats for sharing multi-dimensional strided array data such as DLPack (DMLC 2024).

Workflow After describing every component of our framework, let us summarize the workflow in Figure 1 and Algorithm 1. Figure 1 schematically visualizes the process of the main steps of the workflow. Algorithm 1 provides a "minimal" code example of the framework application to a solid mechanics problem. However, it covers only the main steps of the workflow and gives only a general idea of how the framework can be used – more detailed examples of the framework application to specific plasticity problems can be found in Section 4 and supplementary material (Latyshev and Hale 2024).

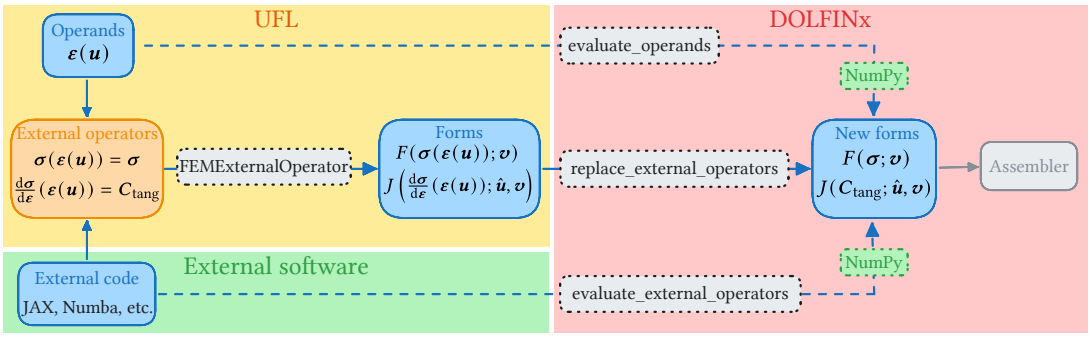


Figure 1 The diagram summarizes the workflow of implementing a solid mechanics constitutive model within the framework. The stress field $\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$ and its derivative $\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u}))$ are wrapped with the `FEMExternalOperator` objects, which depend on the UFL-expression of the strain field $\boldsymbol{\varepsilon}(\boldsymbol{u})$ and some external code. Once the forms containing `FEMExternalOperator` objects are defined, they can be replaced with their representatives from the DOLFINx environment, the `Function` objects $\boldsymbol{\sigma}$ and $\boldsymbol{C}_{\text{tang}}$. The values of the UFL-expression of $\boldsymbol{\varepsilon}(\boldsymbol{u})$ are computed at Gauss points via the `evaluate_operands` function using FFCx generated code and stored in an `ndarray`. The values of the external operators and their derivatives are then evaluated at the Gauss points via the `evaluate_external_operators` function and stored in `ndarray`. In the final step, the standard DOLFINx assembler is used to assemble the forms with the `Function` objects containing $\boldsymbol{\sigma}$ and $\boldsymbol{C}_{\text{tang}}$.

```python
def sigma_external(
    derivatives: Tuple[int, ...]
) -> Callable[[np.ndarray], np.ndarray]:
    if derivatives == (0,):
        return sigma_impl # user-defined function (external code is inside)
    elif derivatives == (1,):
        return C_tang_impl # user-defined function (external code is inside)
    else:
        raise NotImplementedError

# Define the output function space of the external operator
S = fem.functionspace(mesh, quadrature_element)

sigma = FEMExternalOperator(
    epsilon(u), # operand: UFL Expression of strains
    function_space=S, # Output function space
    external_function=sigma_external # Python callable
)

# Define the form `F` and its Jacobian `J`
F = ufl.inner(sigma, epsilon(v)) * dx - F_ext(v)
# UFL's SD creates a new `FEMExternalOperator` wrapping the
# derivative of `sigma` aka `C_tang`
J = ufl.derivative(F, u, u_hat)
J_expanded = ufl.algorithms.expand_derivatives(J)

# Create new forms with `FEMExternalOperator` replaced with `Function` objects
# appropriately sized to hold the result of evaluating the external operator
F_replaced, F_external_operators = replace_external_operators(F)
J_replaced, J_external_operators = replace_external_operators(J_expanded)

# Define final forms to assemble
F_form = fem.form(F_replaced)
J_form = fem.form(J_replaced)

# Loop implementing iterative solution algorithm.
# e.g. Newton method, fixed-point iteration etc.
for _ in range(0, 10):
    # Evaluate values of `epsilon(u)`
    evaluated_operands = evaluate_operands(F_external_operators)
    # Evaluate and update values of `sigma` via 'sigma_impl'
    evaluate_external_operators(J_external_operators, evaluated_operands)
    # Evaluate and update values of `C_tang` via 'C_tang_impl'
    evaluate_external_operators(F_external_operators, evaluated_operands)

    # Assemble Jacobian into matrix
    A_matrix = fem.assemble_matrix(J_form)
    # Assemble residual into vector
    b_vector = fem.assemble_vector(F_form)
```

Algorithm 1 Minimal and abbreviated code example of the framework applied to a non-specific solid mechanics problem. It shows how the main features of the framework (class `FEMExternalOperator` and functions `replace_external_operators`, `evaluate_operands`, `evaluate_external_operators`) are used to define the problem. Note how external operator allows for the concise and unified expression of models involving variational and non-variational terms in UFL.

Remark [Memory] The `FEMExternalOperator` class efficiently manages memory allocation and data transfer between FEniCSx and user-defined external operators by pre-allocating memory and minimizing unnecessary data duplication and copying. To store the values of an external operator in its `Function` representative prior to assembly, the `FEMExternalOperator` class allocates the memory for both the `Function` object and the function space, to which the former belongs. This memory allocation happens each time a `FEMExternalOperator` object is created, e.g. when the UFL's AD tools propagate through a form containing the `FEMExternalOperator` objects. On

the other hand, the data transfer between FEniCSx and the user's function does not involve extra memory allocation. The transfer is performed by copying the values of one NumPy array, which is a result of the user's program call, directly into another NumPy array attached to the `Function` representative. As the `Function` object is allocated in advance, the framework just updates the values of `Function`. Thus, multiple evaluations of the external operators throughout the modelling do not lead to extra allocation of resources.

Remark [Performance] For non-trivial constitutive models, the runtime of the user's implementation of the external operator usually dominates the runtime of the other aspects of evaluating an external operator, in particular, the data transfer between DOLFINx and users implemented external operators. As discussed previously, this data transfer is performed by copying the values from one `ndarray` to another. Time spent on such a copy is only a small fraction with respect to the time taken to execute the user's implementation of the operator. Notwithstanding this argument, to reach the highest level of performance we recommend users implemented external operators using just-in-time (JIT) compilation features available in libraries like Numba and JAX, or in a compiled language.

Remark [Parallelism and scaling] It is important to note that even in more sophisticated scenarios where the constitutive model is defined by dozens of internal state variables, e.g. in crystal plasticity (Méric and Cailletaud 1991), the evaluation of external operators takes only a small portion of the time consumed by the entire finite element algorithm. Although the constitutive update involves solving a system of hundreds of nonlinear equations, this process can usually be performed locally and independently of constitutive updates at other Gauss points. Consequently, the evaluation of external operators at Gauss points, which encapsulates the constitutive update, is an embarrassingly parallel task, unlike, for example, the solution of the resulting finite element linear systems. We also show programming models (JAX, Numba) where there is the automatic potential to exploit CPU instruction level parallelism, e.g. single instruction multiple data (SIMD), via Numba's use of LLVM and JAX's use of Accelerated Linear Algebra (XLA) and LLVM. Our approach works already with the process-level MPI parallelism in DOLFINx and in Appendix A.1 we show experimentally that the limit for strong scaling of the constitutive update occurs later than for the linear solve. However, we have not yet explored performance on wider hardware architectures, e.g. GPUs, or using intermediate levels of parallelism e.g. threads.

4 Application of the framework to plasticity problems

In order to show how our framework can be used to define constitutive models via external packages, we solve two plasticity problems. The first one is based on a von Mises model with isotropic hardening, which is defined via the package Numba. The second one is the plasticity problem with the non-associative plastic law of the Mohr-Coulomb yield criterion with apex smoothing, where the JAX package is applied. The detailed implementation of these problems can be found in the tutorials provided as supplementary material (Latyshev and Hale 2024).

4.1 Von Mises plasticity

The simplicity of von Mises plasticity mode with linear isotropic hardening in the case of isotropic elasticity makes it popular within solid mechanics community as we can derive all the quantities of interest analytically, which happens rarely in real applications. Thus, the von Mises plasticity is an obvious choice of a "Hello, world" example to demonstrate the main aspects of our framework.

Here we apply the Numba package to define all the quantities of interest of the von Mises model following the previous implementations of the authors within FEniCSx in the standard setting (Bleyer 2024b, plasticity.py) and in conic optimization one (Latyshev and Bleyer 2022, convex_plasticity.ipynb). The results will be compared against the implementation based on a pure UFL formulation of the same problem (Latyshev and Hale 2024, von_mises_ufl.py), which is possible as the von Mises stress vector $\boldsymbol{\sigma}$ and the tangent moduli $\boldsymbol{C}_{\text{tang}}$ may be expressed explicitly via UFL.

Problem formulation

In the first example, we consider a cylinder expansion problem in the plane strain case. The domain Ω is represented by the first quarter of the hollow cylinder with inner R_i and outer R_o radii, where symmetry conditions are set on the left and bottom sides and pressure is applied to the inner boundary $\partial\Omega_{\text{inner}}$. The behaviour of cylinder material is defined by the von Mises yield criterion with linear isotropic hardening law for the associative plastic flow

$$f(\boldsymbol{\sigma}, p) = \sigma_{\text{eq}}(\boldsymbol{\sigma}) - \sigma_0 - Hp \leqslant 0, \tag{19}$$

where $\sigma_{\text{eq}}(\boldsymbol{\sigma}) = \sqrt{\frac{3}{2}\boldsymbol{s}\cdot\boldsymbol{s}}$ is an equivalent stress, σ_0 is a uniaxial strength and $H = \frac{EE_t}{E-E_t}$ is an isotropic hardening modulus, which depends on the Young E and the tangent elastic moduli E_t. Thus, we solve the Weak formulation (16) of the cylinder expansion problem with the Constraint (19) and the linear functional $F_{\text{ext}}(\boldsymbol{v})$ representing the external force defined as

$$F_{\text{ext}}(\boldsymbol{v}) = q\int_{\partial\Omega_{\text{inner}}} \boldsymbol{n}\cdot\boldsymbol{v}\,\mathrm{d}\boldsymbol{x}, \tag{20}$$

where the parameter q is progressively increased up to a value slightly larger than the analytical collapse load of perfect plasticity

$$q_{\text{lim}} = \frac{2}{\sqrt{3}}\sigma_0 \log\frac{R_o}{R_i}. \tag{21}$$

Implementation

We treat the stress vector $\boldsymbol{\sigma}$ as an external operator acting on the strain vector $\boldsymbol{\varepsilon}(\boldsymbol{u})$ and represent it through a `FEMExternalOperator` object. By implementation of this external operator, we mean the implementation of the return-mapping procedure analytically. We evaluate the values of the stress vector $\boldsymbol{\sigma}$ and its derivative $\boldsymbol{C}_{\text{tang}}$ via the Numba package, which typically produces highly optimized machine code with runtime performance at the level of traditional compiled languages.

In Algorithm 2 we show the implementation of the `return_mapping` function that returns the values of the consistent tangent moduli $\boldsymbol{C}_{\text{tang}}^{n+1}$, the stress tensor $\boldsymbol{\sigma}^{n+1}$, and the increment of cumulative plastic strain Δp in the `ndarray`-format. On the input, it receives `ndarray` objects representing the values of the current increment of the strain tensor $\boldsymbol{\varepsilon}(\Delta\boldsymbol{u})$ and values of the history variables: the stress state $\boldsymbol{\sigma}^n$ and the internal variable p^n from the previous loading step. In the scope of the `return_mapping` function, we use only standard Python operations and NumPy functions compatible with Numba. Thus, the definition of our external operator does not depend on the FEniCSx environment and consequently is not limited by its capabilities.

In terms of performance, we use Numba's JIT compilation decorated `@njit` to compile the function `return_mapping` at run-time.

Validation

As can be seen from the Figure 2, our results agree with the pure UFL implementation (Latyshev and Hale 2024, von_mises_ufl.py) of the von Mises model.

4.2 Mohr—Coulomb plasticity

We implement the non-associative plasticity model of Mohr-Coulomb with apex-smoothing and solve a soil slope stability problem. We use the JAX package to define constitutive relations including the differentiation of certain terms. This example demonstrates how AD techniques may be used to define constitutive models that require differentiation of expressions without significant differentiation by hand.

The slope stability problem is based on the limit analysis within a semi-definite programming framework (Bleyer 2022, limit_analysis_3D_SDP.ipynb), where the plasticity model was replaced by the one defined through the Mohr-Coulomb yield surface with apex smoothing (Abbo and Sloan 1995).

```python
@numba.njit
def return_mapping(deps_: np.ndarray, sigma_n_: np.ndarray, p_: np.ndarray):
    """Performs the return-mapping procedure."""
    num_cells = deps_.shape[0]

    C_tang_ = np.empty((num_cells, num_quadrature_points, 4, 4),
    dtype=sigma_n_.dtype)
    sigma_ = np.empty_like(sigma_n_)
    dp_ = np.empty_like(p_)

    def _kernel(deps_local, sigmxa_n_local, p_local):
        """Performs the return-mapping procedure locally at a Gauss point."""
        sigma_elastic = sigma_n_local + C_elas @ deps_local
        s = deviatoric @ sigma_elastic
        sigma_eq = np.sqrt(3.0 / 2.0 * np.dot(s, s))

        f_elastic = sigma_eq - sigma_0 - H * p_local
        f_elastic_plus = (f_elastic + np.sqrt(f_elastic**2)) / 2.0

        dp = f_elastic_plus / (3 * mu + H)

        n_elas = s / sigma_eq * f_elastic_plus / f_elastic
        beta = 3 * mu * dp / sigma_eq

        sigma = sigma_elastic - beta * s

        n_elas_matrix = np.outer(n_elas, n_elas)

        C_tang = C_elas - 3 * mu * (3 * mu / (3 * mu + H) - beta) *
        n_elas_matrix - 2 * mu * beta * deviatoric

        return C_tang, sigma, dp

    for i in range(0, num_cells):
        for j in range(0, num_quadrature_points):
            C_tang_[i,j], sigma_[i,j], dp_[i,j] = _kernel(deps_[i,j],
            sigma_n_[i,j], p_[i,j])

    return C_tang_, sigma_, dp_
```

Algorithm 2 Implementation of the return-mapping procedure of the von Mises plasticity using Numba. The function `return_mapping` receives NumPy arrays of values of the strain tensor $\boldsymbol{\varepsilon}(\Delta \boldsymbol{u})$ and such variables conserving the previous loading history as the stress tensor $\boldsymbol{\sigma}^n$, and the cumulative plastic strain p^n from the previous loading step, evaluated in all Gauss points. The return-mapping procedure itself is implemented for one Gauss point in the function `_kernel`, which then will be called in the loops through Gauss points and cells. The function `return_mapping` returns the values of the consistent tangent moduli C_{tang}^{n+1}, the stress tensor $\boldsymbol{\sigma}^{n+1}$, and the increment of cumulative plastic strain Δp as a global flatten `ndarray`-s. These values will be used to update external operators and the variables of the loading history. Note that `num_quadrature_points` is statically defined outside the scope of `return_mapping` - this gives Numba/LLVM the opportunity to unroll the loop over quadrature points.

Problem formulation

We solve a soil slope stability problem of a domain Ω represented by a rectangle $[0, L] \times [0, H]$ under plane strain assumptions. For this problem the homogeneous Dirichlet boundary conditions for the displacement field $\boldsymbol{u} = \boldsymbol{0}$ on the right side $x = L$ and the bottom $y = 0$. The loading consists of a gravitational body force $\boldsymbol{q} = [0, -\gamma]^\top$ with γ being the soil self-weight

$$F_{\text{ext}}(\boldsymbol{v}) = \int_\Omega \boldsymbol{q} \cdot \boldsymbol{v} \, \mathrm{d}\boldsymbol{x}. \tag{22}$$

We progressively increase the soil self-weight γ until a plateau on the loading-displacement curve is reached.

The constitutive model of the soil is described by a non-associative plasticity law without

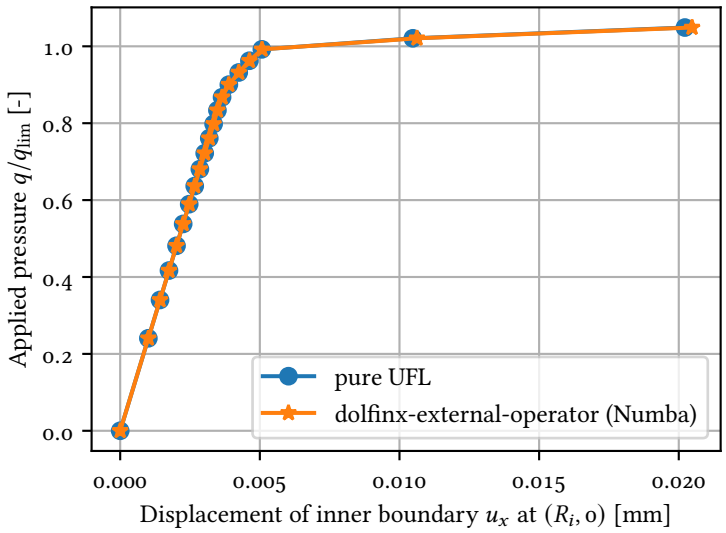


Figure 2 Displacement of the inner boundary of the cylinder $u_x(R_i, 0)$ with respect to the applied pressure in the von Mises model with isotropic hardening implemented via two methods. The plastic deformations are reached at the pressure q_{lim} equal to the analytical collapse load for perfect plasticity.

hardening that is defined by the Mohr-Coulomb yield surface f and the plastic potential g. Both quantities may be expressed through the following function h

$$h(\boldsymbol{\sigma}, \alpha) = \frac{I_1(\boldsymbol{\sigma})}{3} \sin\alpha + \sqrt{J_2(\boldsymbol{\sigma}) K^2(\theta, \alpha) + a^2(\alpha) \sin^2\alpha} - c\cos\alpha, \tag{23}$$

$$f(\boldsymbol{\sigma}) = h(\boldsymbol{\sigma}, \phi), \tag{24}$$

$$g(\boldsymbol{\sigma}) = h(\boldsymbol{\sigma}, \psi), \tag{25}$$

where c is a cohesion, ϕ, ψ and θ are friction, dilatancy and Lode angles respectively, $I_1(\boldsymbol{\sigma}) = \text{tr}\boldsymbol{\sigma}$ is the first invariant of the stress field and $J_2(\boldsymbol{\sigma}) = \frac{1}{2}\boldsymbol{s}\cdot\boldsymbol{s}$ is the second invariant of the deviatoric part of the stress. The explicit expressions of other terms of the function h together with the exact values of model parameters can be found in Appendix A.3. In summary, we solve Problem (16), with Constraint (24) and Linear functional (22).

Implementation

Similarly to the von Mises example we define the external operator $\boldsymbol{\sigma}$ and its derivative $\boldsymbol{C}_{\text{tang}}$ through the implementation of the return-mapping procedure. However in the case of Mohr-Coulomb, this procedure is not trivial, compared to the von Mises case, and must be implemented numerically. In practice, we must solve System of constitutive equations (14) through the consistent linearisation of Jacobian (15) for the local Newton method.

For the correct implementation of the return-mapping procedure in this example, we need to take derivatives of certain terms. We distinguish three levels of the differentiation: the level of definition of the local residual $\boldsymbol{r}$ from Constitutive equations (14), the local level and the global level. The first level is linked to the derivative of Plastic potential (25), $\frac{\mathrm{d}g(\boldsymbol{\sigma})}{\mathrm{d}\boldsymbol{\sigma}}$, which defines one of the Constitutive equations (14). At the local level, we need to compute the local Jacobian $\boldsymbol{j}(\boldsymbol{y}) = \frac{\mathrm{d}\boldsymbol{r}(\boldsymbol{y})}{\mathrm{d}\boldsymbol{y}}$, the derivative of the local residual $\boldsymbol{r}$ with respect to its argument. Finally, at the global level, we pass a derivative through the entire return-mapping procedure with respect to the strain tensor $\boldsymbol{\varepsilon}$ or, in other words, we compute the tangent matrix $\boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u})) = \frac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}}$, the derivative of the external operator $\boldsymbol{\sigma}$, which is needed for the global jacobian J of Variational form (16). The correct computation of all the three derivatives: $\frac{\mathrm{d}g(\boldsymbol{\sigma})}{\mathrm{d}\boldsymbol{\sigma}}$, $\frac{\mathrm{d}\boldsymbol{r}(\boldsymbol{y})}{\mathrm{d}\boldsymbol{y}}$ and $\frac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}}$, is crucial for the correct implementation of the Mohr-Coulomb model. In our implementation we apply AD techniques from the JAX package to compute the derivatives *exactly* at each level at a certain point (e.g. at a Gauss point).

At the level of the residual $\boldsymbol{r}$, the derivation of the plastic potential $g(\boldsymbol{\sigma})$ is straightforward. Once we define a function evaluating the values of g for a given stress field, it is sufficient to

call the JAX's AD `jacfwd` function which creates a new function `dgdsigma` that computes the value of the plastic potential derivative locally at a given Gauss point:

```
def g(sigma_local):
    return h(sigma_local, psi)

dgdsigma = jax.jacfwd(g)
```

where `psi` is the constant dilatancy angle.

At the local level, deriving the jacobian $\boldsymbol{j}(\boldsymbol{y}) = \frac{\mathrm{d}\boldsymbol{r}(\boldsymbol{y})}{\mathrm{d}\boldsymbol{y}}$ is more complicated than the differentiation of the plastic potential due to the presence of several mathematical equations and conditionals. Despite this, by implementing a function computing the residual $\boldsymbol{r}$, the application of the JAX's AD is still straightforward as can be seen in the following code-snippet:

```
def r(y_local, deps_local, sigma_n_local):
    sigma_local = y_local[:stress_dim]
    dlambda_local = y_local[-1]

    res_g = r_g(sigma_local, dlambda_local, deps_local, sigma_n_local)
    res_f = r_f(sigma_local, dlambda_local, deps_local, sigma_n_local)

    res = jnp.c_["0,1,-1", res_g, res_f]  # concatenates an array and a scalar
    return res

drdy = jax.jacfwd(r)
```

Subsequently, the obtained function `drdy` will be used in the local Newton method to compute the local Jacobian, see Algorithm 3 .

At the global level, the computation of the derivative $\frac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}}$ does not look trivial. Indeed, in contrast to the plastic potential g and the local residual $\boldsymbol{r}$, the stress operator $\boldsymbol{\sigma}$ is defined via the iterative solver, on which the return-mapping procedure is based. This implies passing the derivative through the entire solver in order to get the value of $\frac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}}$. Regardless of this difficulty, the automatic differentiation techniques are able to compute the derivative of such a numerical algorithm. For instance, JAX can calculate the derivative of the function `return_mapping` (see Algorithm 3) automatically in spite of the presence of the `while_loop`. This results in a new program `dsigma_ddeps` (see Algorithm 3) that computes the values of the tangent moduli $\boldsymbol{C}_{\text{tang}}$ *exactly* at a Gauss point, so there is no need for a supplementary computation of the stress tensor.

Although we obtained functions to compute all three derivatives required to define the Mohr-Coulomb model, these functions are still constructed to be called for a single Gauss point, whereas we need to evaluate them on the entire quadrature space. In this regard, JAX's vectorization transformation `vmap` serves as a convenient tool to extrapolate them to the level of the entire function space. In particular, we vectorize the function `dsigma_ddeps` to get `dsigma_ddeps_vec` that computes the values of the tangent moduli $\boldsymbol{C}_{\text{tang}}$ and the stress field $\boldsymbol{\sigma}$ at all Gauss points of the functional space simultaneously:

```
dsigma_ddeps_vec = jax.jit(jax.vmap(dsigma_ddeps, in_axes=(0, 0)))
```

where similar to the Numba package, we compile the final vectorized function by using the JAX's function `jit`.

Finally, we define the external operator $\boldsymbol{\sigma}$ and its derivative $\boldsymbol{C}_{\text{tang}}$ via a single function `C_tang_impl`:

```
def C_tang_impl(deps: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    deps_ = deps.reshape((-1, 4))
    sigma_n_ = sigma_n.x.array.reshape((-1, 4))

    (C_tang_global, state) = dsigma_ddeps_vec(deps_, sigma_n_)

    return C_tang_global.reshape(-1), sigma_global.reshape(-1)
```

```python
def return_mapping(deps_local: np.ndarray, sigma_n_local: np.ndarray):
    niter = 0

    dlambda = ZERO_SCALAR
    sigma_local = sigma_n_local
    y_local = jnp.concatenate([sigma_local, dlambda])

    res = r(y_local, deps_local, sigma_n_local)
    norm_res0 = jnp.linalg.norm(res)

    def cond_fun(state):
        norm_res, niter, _ = state
        return jnp.logical_and(norm_res / norm_res0 > tol, niter < Nitermax)

    def body_fun(state):
        norm_res, niter, history = state

        y_local, deps_local, sigma_n_local, res = history

        j = drdy(y_local, deps_local, sigma_n_local)
        j_inv_vp = jnp.linalg.solve(j, -res)
        y_local = y_local + j_inv_vp

        res = r(y_local, deps_local, sigma_n_local)
        norm_res = jnp.linalg.norm(res)
        history = y_local, deps_local, sigma_n_local, res

        niter += 1

        return (norm_res, niter, history)

    history = (y_local, deps_local, sigma_n_local, res)

    norm_res, niter_total, y_local = jax.lax.while_loop(cond_fun, body_fun,
    (norm_res0, niter, history))

    sigma_local = y_local[0][:stress_dim]
    dlambda = y_local[0][-1]
    sigma_elas_local = C_elas @ deps_local
    yielding = f(sigma_n_local + sigma_elas_local)

    return sigma_local, (sigma_local, niter_total, yielding, norm_res, dlambda)

dsigma_ddeps = jax.jacfwd(return_mapping, has_aux=True)
```

Algorithm 3 Implementation of the return-mapping procedure of the Mohr-Coulomb plasticity with apex smoothing via JAX package. The function `return_mapping` receives NumPy arrays of values of the strains $\boldsymbol{\varepsilon}(\Delta \boldsymbol{u})$ and the stresses $\boldsymbol{\sigma}^n$ evaluated at a given Gauss point. The return-mapping procedure is implemented via the Newton method wrapped by the JAX's `while_loop`. It returns a tuple of the new stress field values at a given Gauss point and another tuple containing the same values of the stress field plus some auxiliary data related to the Newton method. Then we apply the JAX's AD `jacfwd` to compute the derivative of the function `return_mapping` with respect to its first input. This results in the creation of a new function `dsigma_ddeps` that returns both the stress field and its derivative at a given Gauss point.

which can then be attached to the definition of the UFL external operator.

Remark [Mohr-Coulomb model without AD] The aforementioned derivatives are commonly obtained analytically in numerous applications of not only the Mohr-Coulomb model but other plasticity models as well. In practice, the analytical expressions of these derivatives are involved and so their translation into programming code can create multiple lines of code (Helfer et al. 2024), which is often a source of human error leading to additional effort spent on debugging. As we demonstrated in the application of JAX, AD can simplify the differentiation by hand and significantly reduce the amount of code needed to implement complex analytical formulas.

Verification

Within the Mohr-Coulomb example, we perform three verification tests. The first checks the correct tracing of Yield surface (24). The second is the Taylor remainder test verifying that the Jacobian (18) is a consistent first-order approximation of Residual (16). The third checks that our solution of the slope stability problem defined by Equations (16), (22) and (23) matches an existing result found in the literature.

Yield surface tracing In this part, we verify that the Mohr-Coulomb model is implemented correctly by visually tracing its yield surface. We generate several stress paths and check whether they remain within the yield surface after passing through the `return_mapping` function, which depends on the derivatives $\frac{\mathrm{d}g(\boldsymbol{\sigma})}{\mathrm{d}\boldsymbol{\sigma}}$ and $\frac{\mathrm{d}\boldsymbol{r}(\boldsymbol{y})}{\mathrm{d}\boldsymbol{y}}$. The yield surface with the stress paths projected onto the deviatoric plane is shown in Figure 3, where we observe that the yield surface of Mohr-Coulomb with apex smoothing is reached along different stress paths (colored lines). Moreover, the obtained yield surface lies along the standard Mohr-Coulomb one without smoothing (black contour line). These results justify the correct implementation of the plasticity model and the derivatives `dgdsigma` and `drdx` obtained by JAX's AD.

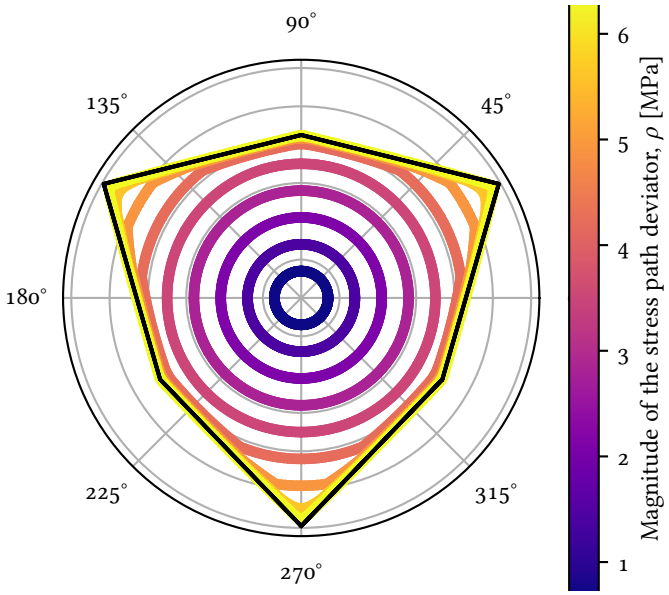


Figure 3 Tracing of the Mohr-Coulomb with apex smoothing yield surface. It is obtained by passing several stress paths projected onto the deviatoric plane (ρ, θ), where $\rho = \sqrt{2J_2(\boldsymbol{\sigma})}$ and θ is the Lode angle (see Appendix A.3). Each colour represents one loading step along the stress paths. The circles are associated with the loading during the elastic phase. Once the loading reaches the elastic limit, the circles start outlining the yield surface, which in the limit lay along the standard Mohr-Coulomb one without smoothing (black contour).

Taylor remainder test We perform a Taylor remainder test to verify that the assembled residual F and its Jacobian J are consistent zeroth- and first-order approximations of the residual F, respectively. To implement the Taylor remainder test and to ensure that the test results are *mesh-independent* we apply the Taylor remainder theorem on Banach spaces (Blanchard and Brüning 2015, p. 524) to the operator $\mathcal{F} : V \to V'$ which is linked with the form F in the following way

$$\langle \mathcal{F}(\boldsymbol{u}), \boldsymbol{v} \rangle := F(\boldsymbol{u}; \boldsymbol{v}), \tag{26}$$

where V' is a dual space of V, $\langle \cdot, \cdot \rangle$ is the $V' \times V$ duality pairing. Precise details and discussion on how we implemented the Taylor remainder test using the dual norm can be found in Appendix A.4. By applying the Taylor remainder theorem on Banach spaces, which perturbs the operator $\mathcal{F}$ in the direction $k\,\boldsymbol{\delta u} \in V$ for $k > 0$, we obtain that the norm of the zeroth- and first-order Taylor remainders $\|r_k^0\|_{V'}$ and $\|r_k^1\|_{V'}$ converge at first- and second-order convergence rates in k,

respectively

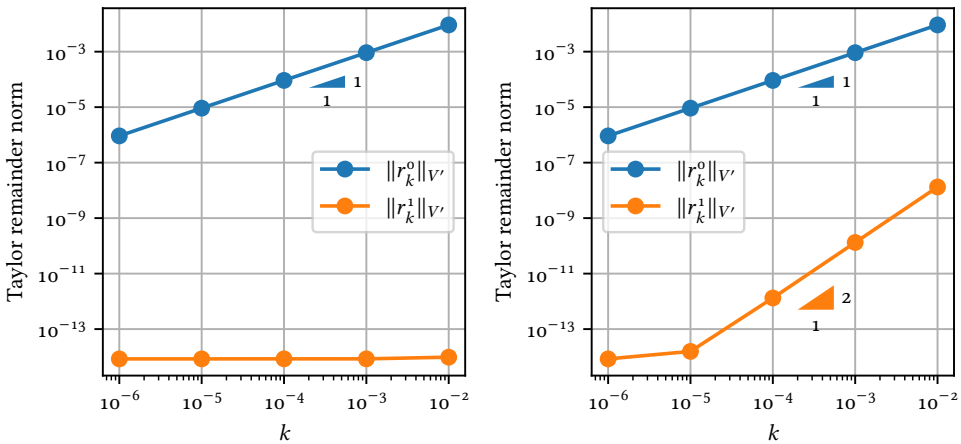
$$\|r_k^0\|_{V'} := \|\mathcal{F}(\boldsymbol{u} + k\,\boldsymbol{\delta u}) - \mathcal{F}(\boldsymbol{u})\|_{V'} \longrightarrow 0 \text{ at } O(k), \tag{27}$$

$$\|r_k^1\|_{V'} := \|\mathcal{F}(\boldsymbol{u} + k\,\boldsymbol{\delta u}) - \mathcal{F}(\boldsymbol{u}) - (\mathcal{J}(\boldsymbol{u}))(k\boldsymbol{\delta u})\|_{V'} \longrightarrow 0 \text{ at } O(k^2), \tag{28}$$

with the operator $\mathcal{J} : V \to \mathcal{L}(V, V')$ representing the Jacobian J of the form F

$$\langle (\mathcal{J}(\boldsymbol{u}))(k\boldsymbol{\delta u}), \boldsymbol{v}\rangle := J(\boldsymbol{u}; k\boldsymbol{\delta u}, \boldsymbol{v}), \tag{29}$$

where $\mathcal{L}(V, V')$ is a space of bounded linear operators from V to V'. The plots in Figure 4 show



(a) Elastic phase (b) Plastic phase

Figure 4 Taylor remainder test for the form F around (a) a solution $\boldsymbol{u}$ in the purely elastic phase and (b) a solution $\boldsymbol{u}$ containing a portion of the domain in the plastic phase. For the purely elastic phase (a) the norm of the zeroth-order Taylor remainder r_k^0 achieves the first-order convergence rate, whereas the norm first-order remainder r_k^1 is computed to the level of machine precision, as the purely elastic problem can be expressed as a quadratic functional (constant Jacobian). For the plastic phase (b), the norm of the zeroth-order Taylor remainder r_k^0 reaches the first-order convergence rate, whereas the norm of the first-order remainder r_k^1 achieves the second-order convergence rate. These results imply that the automatically derived Jacobian is a consistent (correct) first-order approximation of the residual.

that the computed first-order and second-order convergence rates of the Taylor remainders are in agreement with the expectations defined by the Taylor remainder theory. From this, we can conclude that, together, FEniCSx (UFL, FFCx and DOLFINx) and JAX produce a consistent (correct) approximation of the assembled finite element residual and Jacobian for the Mohr-Coulomb model.

Solution of slope stability problem We now demonstrate that our numerical solution of the slope stability problem matches the results found in the literature. We progressively increase the second component of the gravitational body force $\boldsymbol{q} = [0, -\gamma]^\top$ from Equation (22), the soil self-weight γ, up to the critical value $\gamma_{\text{lim}}^{\text{num}}$, when the perfect plasticity plateau is reached on the loading-displacement curve at the top left corner $(0, H)$. Then we compare $\gamma_{\text{lim}}^{\text{num}}$ with analytical γ_{lim} found through the formula of the slope stability factor l_{lim}

$$l_{\text{lim}} = \gamma_{\text{lim}} H / c. \tag{30}$$

Equation (30) is derived in the case of the standard Mohr-Coulomb model without smoothing under plane strain assumptions for associative plastic flow (W. F. Chen and Liu 1990). In the case of the rectangular slope with friction angle ϕ and dilatancy angle ψ both equal to 30°, $l_{\text{lim}} = 6.69$ (W. F. Chen and Liu 1990, p. 368). The values of model parameters used in this numerical test can be found in Table A.2. The orange loading-displacement curve shown in Figure 6 confirms that the numerically estimated yield strength limit reached for $\gamma_{\text{lim}}^{\text{num}}$ is close to γ_{lim}. Additionally, we demonstrate the magnitude of the displacement field at the last loading step in Figure 5, where the slip of the rectangular slope can be observed on the left side $x = 0$.

We remark that our implementation also supports non-associative plastic flow where $f \neq g$ in Equations (24) and (25). We perform a second simulation setting $\psi = 10°$ and $\phi = 30°$ with

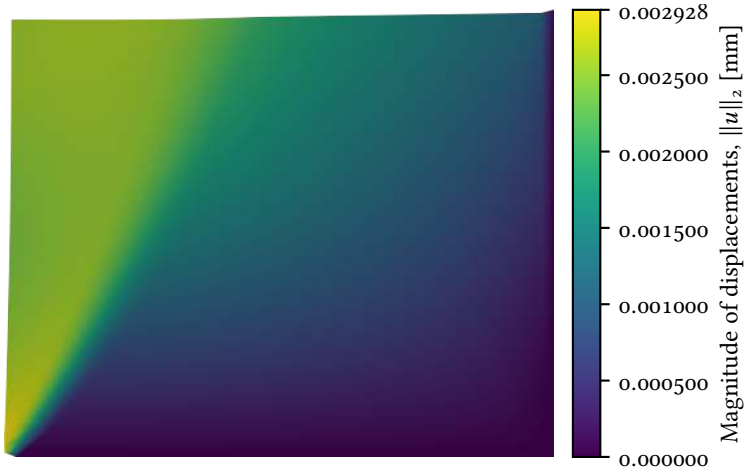


Figure 5 Slip of the slope for the Mohr-Coulomb problem. The domain is warped by the displacement field (magnified). The magnitude of the displacement field is shown by the colour and reaches its maximum at the bottom left corner.

all other parameters as in Table A.2. This creates the green load displacement curve shown in Figure 6. In this scenario, the material law exhibits less volume expansion and makes the plastic potential surface flatter, which results in a lower yield strength limit than in the associative case.

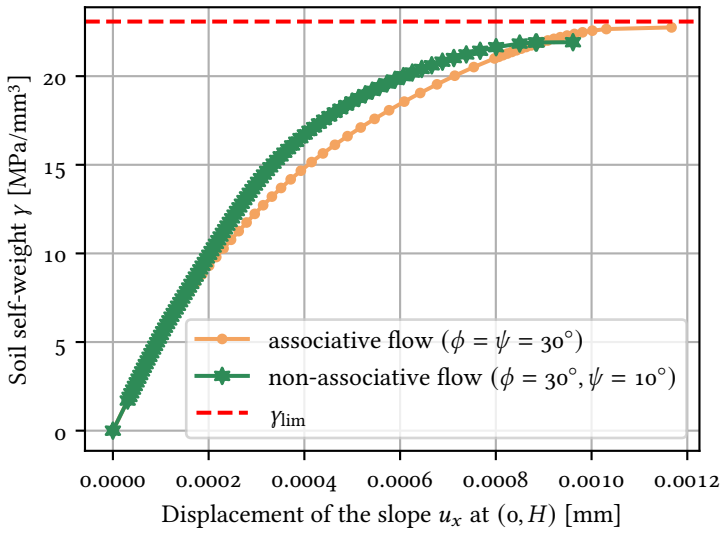


Figure 6 Displacement of the slope $u_x(0, H)$ with respect to the soil self-weight γ in the Mohr-Coulomb model with apex smoothing for the associative plastic flow ($\phi = \psi = 30°$) and the non-associative one ($\phi = 30°, \psi = 10°$). Reaching the yield strength limit $\gamma_{\text{lim}}^{\text{num}}$ (the plateau) is associated with losing the stability by the slope. The γ_{lim} for the associative flow is obtained via an analytical solution using the standard Mohr-Coulomb model without smoothing (W. F. Chen and Liu 1990, p. 368).

5 Conclusion

In this article, we described a methodology to define a range of constitutive models in FEniCSx / DOLFINx and a supporting software framework. Our framework provides the user with an interface that wraps a constitutive model via an external operator (Bouziani and Ham 2021). Data is passed between the external operator and DOLFINx using standard array-like data structures, allowing a wide range of different programming languages and environments to be used. Together these developments enable the definition of a general class of constitutive models in FEniCSx. In particular, the application of the framework is demonstrated with two plasticity problems: the von Mises model with isotropic hardening (using the Numba package) and the Mohr-Coulomb model with apex smoothing (using the JAX package). In the example of the

Mohr-Coulomb model, we demonstrated that AD significantly reduces the amount of manual differentiation required to express a general constitutive model.

This paper has focused on using modern algorithmic automatic differentiation tools, exemplified by JAX, to implement constitutive models, and we are particularly excited about the possibilities that this type of approach can open in mechanics problems involving challenging constitutive models. Although we focused on one constitutive model type (small strain elastoplasticity) implemented using two programming environments (Numba and JAX), the framework serves as a general interface to expand the capabilities of FEniCSx to different types of constitutive models implemented in numerous ways. For example, CVXPY (Diamond and Boyd 2016) could be used to implement and solve convex plasticity models (Latyshev and Bleyer 2022), PyTorch and TensorFlow to build data-centric neural network-based constitutive models or an external solver could be called to implement a multi-scale model. Beyond solid mechanics, the framework could be applied to fluid mechanics problems with general non-Newtonian constitutive behaviour or various multi-physics problems such as magnetorheological elastomers (Mukherjee et al. 2021).

Aside from these newer approaches to implementing constitutive models, Rosenbusch et al. 2024 notes that a great number of useful constitutive models have already been implemented in traditional programming languages, e.g. C and Fortran, via *de facto* standard interfaces, e.g. UMAT (Lucarini and Martínez-Pañeda 2024) or generated using dedicated tools, e.g. ZMAT (Z-Set Software 2023) and MFront (Helfer et al. 2020). In future work it would be valuable to include interfaces and examples inclusive to these approaches.

Although we mentioned the possibility of executing constitutive models on GPUs, we have not explored this direction here; although it is now well established that high-order matrix-free finite element methods (vs low-order matrix assembly finite element methods we tackled here) are optimal for the performance profile of GPUs, the overall picture of how matrix-free methods should be applied to solid mechanics problems with general constitutive models remains an active topic of research (Brown et al. 2021; Brown et al. 2022; Lewandowski et al. 2023).

On a final note, we hope that the software framework proves useful to the wider community for integrating constitutive models, both old and new, into FEniCSx, and for exploring the possibilities that novel hardware architectures can bring to problems in solid mechanics.

A Appendices

A.1 Strong scaling of the Mohr-Coulomb model

As a complement to the remark on the parallelism at the end of Section 3, we show a strong parallel scaling test with the Mohr-Coulomb model described in Section 4.2. These results do not constitute a rigorous performance analysis but do indicate reasonable performance and scaling.

In the strong parallel scaling test the number of displacement degrees-of-freedom is kept fixed at 321 602 and the number of MPI processes is doubled in a geometric sequence from 1 through to 64. For the first ten pseudo-timesteps of the slope stability problem we measured the maximum wall time across the MPI processes of the external operator and operand evaluations, Jacobian assemblies and inner linear solves of the outer Newton method. We additionally record the total time for all ten pseudo-timesteps. The results are shown in Figures A.1 and A.2.

The main takeaway is that while evaluations of the operands, external operators and Jacobian assemblies scale perfectly up to 64 MPI processes, the direct linear solver MUMPS reaches its strong scaling limit around 16 MPI processes. This result supports our statement that the constitutive update is an "embarrassingly parallel task" in contrast with other time consuming parts of the full solution process.

Remark [Setup] The numerical simulations were performed on a machine with 2 sockets, each containing an AMD EPYC Rome 7742 64-core processor and 2 TB DDR4 RAM. The software environment was Debian 11, using Podman 5.3.1 container runtime. Instructions for building the environment used in the strong scaling test, as well as the Docker file, are available in (Latyshev 2024).

Remark [Differences] The implementation of the problem differs from the one described in the main text. Instead of passing the entire `return_mapping` function through the JAX AD transformations, here we

apply the implicit function theorem to define the tangent moduli C_{tang} directly. This alternative implementation provides better performance (results not shown) and can be found in the supplementary material (Latyshev 2024, demo_plasticity_mohr_coulomb_mpi.py).

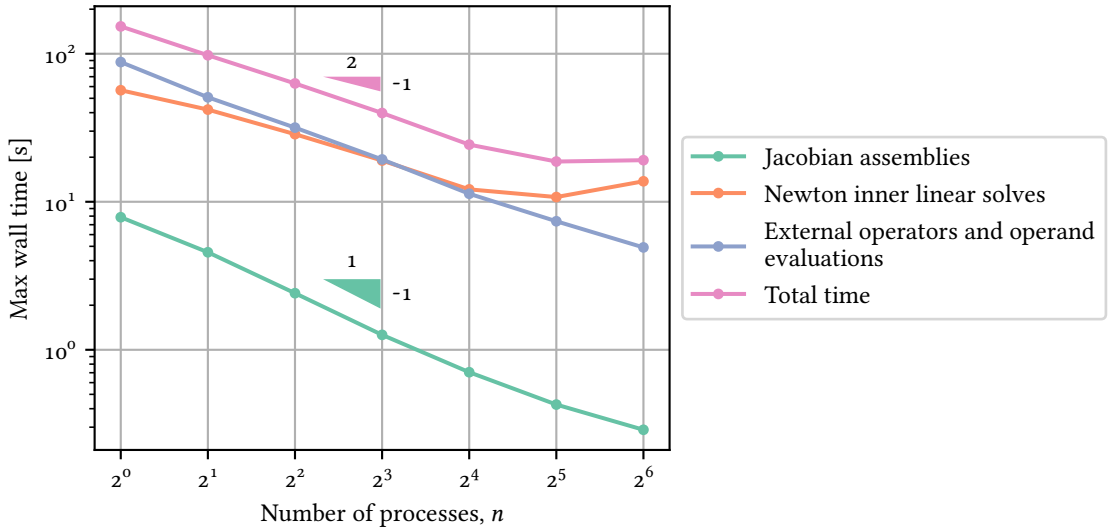


Figure A.1 Strong scaling test of the first ten pseudo-timesteps of the Mohr-Coulomb model. The plot indicates that the direct linear solve reaches its scaling limit at around 16 processes while evaluations of the operands, external operator and Jacobian assemblies continue to scale strongly up to 64 processes. We also remark that the time for the operand and external operator evaluation is on the same order as the linear solver. .

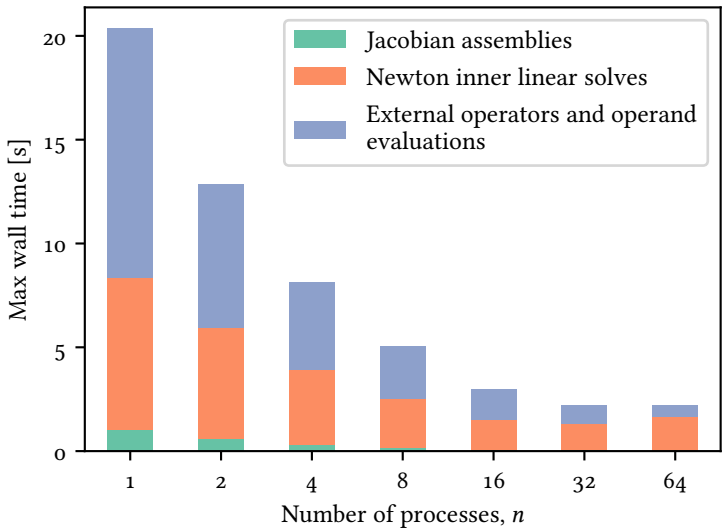


Figure A.2 Breakdown of timings for the tenth pseudo-timestep of the strong scaling Mohr-Coulomb test. .

A.2 von Mises plasticity

Table A.1 contains the material and geometry parameters of the von Mises plasticity example.

A.3 Mohr-Coulomb with apex smoothing

Here we cover the plasticity model with a non-associated flow rule based on the Mohr-Coulomb yield criterion with apex smoothing (Abbo and Sloan 1995) from Section 4.2. The model is defined

Symbol	Value	Units	Meaning
E	$70 \cdot 10^3$	MPa	Young modulus
E_t	$E/100$	MPa	Tangent modulus
ν	0.3	-	Poisson ratio
σ_0	250	MPa	Yield strength
R_e	1.3	mm	External radius of the cylinder
R_i	1.0	mm	Internal radius of the cylinder

Table A.1 Material and geometry parameters used in the von Mises plasticity example in Section 4.1.

by the yield function f and the plastic potential g expressed as follows:

$$h(\boldsymbol{\sigma}, \alpha) = \frac{I_1(\boldsymbol{\sigma})}{3} \sin\alpha + \sqrt{J_2(\boldsymbol{\sigma}) K^2(\theta, \alpha) + a^2(\alpha) \sin^2\alpha} - c\cos\alpha, \tag{A.1}$$

$$f(\boldsymbol{\sigma}) = h(\boldsymbol{\sigma}, \phi), \tag{A.2}$$

$$g(\boldsymbol{\sigma}) = h(\boldsymbol{\sigma}, \psi), \tag{A.3}$$

where ϕ and ψ are the friction and dilatancy angles respectively, c is the cohesion parameter, $I_1(\boldsymbol{\sigma}) = \mathrm{tr}\boldsymbol{\sigma}$ is the first invariant of the stress field, $J_2(\boldsymbol{\sigma}) = \frac{1}{2}\boldsymbol{s} \cdot \boldsymbol{s}$ is the second invariant of the deviatoric part of the stress field. In the quantity $a(\alpha) := a\tan\phi/\tan\alpha$, the tension cut-off parameter a defines how close the hyperbolic approximation is to the Mohr-Coulomb yield surface without smoothing.

In the Equation (A.1) the term $K(\theta, \alpha)$ depends on either friction or dilatancy angle through the parameter α and the Lode angle $\theta = \theta(\boldsymbol{\sigma})$:

$$\theta(\boldsymbol{\sigma}) = \frac{1}{3}\arcsin\left(\frac{-3\sqrt{3}J_3(\boldsymbol{\sigma})}{2\sqrt{J_2^3(\boldsymbol{\sigma})}}\right) \in [-\frac{\pi}{6}, \frac{\pi}{6}], \tag{A.4}$$

where $J_3 = \det \boldsymbol{s}$ is the third invariant of the deviatoric part of the stress field. The term $K(\theta, \alpha)$ is defined as follows:

$$K(\theta, \alpha) = \begin{cases} \cos\theta - \frac{1}{\sqrt{3}}\sin\alpha\sin\theta, & |\theta| < \theta_T, \\ A(\alpha) + B(\alpha)\sin 3\theta + C(\alpha)\sin^2 3\theta, & |\theta| \geqslant \theta_T, \end{cases} \tag{A.5}$$

where θ_T is a transition angle, which "in practice, should not be too near 30° to avoid ill-conditioning ... and the typical value is 25°" (Abbo and Sloan 1995, p. 429) and the terms A, B and C are defined as follows:

$$A(\alpha) = -\frac{1}{\sqrt{3}}\sin\alpha\,\mathrm{sign}\,\theta\sin\theta_T - B(\alpha)\,\mathrm{sign}\,\theta\sin 3\theta_T - C(\alpha)\sin^2 3\theta_T + \cos\theta_T, \tag{A.6}$$

$$\begin{aligned} B(\alpha) = {} & \mathrm{sign}\,\theta\sin 6\theta_T \left(\cos\theta_T - \frac{1}{\sqrt{3}}\sin\alpha\,\mathrm{sign}\,\theta\sin\theta_T\right)/18\cos^3 3\theta_T \\ & - 6\cos 6\theta_T \left(\mathrm{sign}\,\theta\sin\theta_T + \frac{1}{\sqrt{3}}\sin\alpha\cos\theta_T\right)/18\cos^3 3\theta_T, \end{aligned} \tag{A.7}$$

$$\begin{aligned} C(\alpha) = {} & -\cos 3\theta_T \left(\cos\theta_T - \frac{1}{\sqrt{3}}\sin\alpha\,\mathrm{sign}\,\theta\sin\theta_T\right)/18\cos^3 3\theta_T \\ & - 3\,\mathrm{sign}\,\theta\sin 3\theta_T \left(\mathrm{sign}\,\theta\sin\theta_T + \frac{1}{\sqrt{3}}\sin\alpha\cos\theta_T\right)/18\cos^3 3\theta_T, \end{aligned} \tag{A.8}$$

with

$$\mathrm{sign}\,\theta = \begin{cases} +1, & \theta \geqslant 0°, \\ -1, & \theta < 0°. \end{cases} \tag{A.9}$$

The set of all parameters together with their values used for the numerical simulation (cf. Section 4.2) is given in the Table A.2. The values are based on the limit analysis within semi-definite programming framework (Bleyer 2022, limit_analysis_3D_SDP.ipynb) and the implementation of the Mohr-Coulomb model with apex smoothing in MFront (Helfer et al. 2024).

Symbol	Value	Units	Meaning
E	6778	MPa	Young modulus
ν	0.25	-	Poisson ratio
c	3.45	MPa	Cohesion
ϕ	30	∘, degree	Friction angle
ψ	$\{10, 30\}$	°, degree	Dilatancy angle
θ_T	26	°, degree	Transition angle
a	$0.26c/\tan\phi$	MPa	Tension cut-off parameter
L	1.2	mm	Length of the slope
H	1.0	mm	Height of the slope

Table A.2 Material and geometry parameters used in the Mohr-Coulomb with apex smoothing plasticity model, see Section 4.2.

A.4 Taylor remainder test in the dual space

In Section 4.2, we formulated the Taylor remainder test by establishing the convergence of the norm of the Taylor remainders r_k^0 and r_k^1, see Equations (27) and (28), of the operator $\mathcal{F}$, which links to the form F through Equation (26). In the next sections, for practical reasons, we reformulate the Taylor remainder test for the finite subset $V_h \subset V$ and explain how we implement it in more detail. Our implementation leads to *mesh-independent* norms for the Taylor remainders, which is a direct consequence of explicitly considering the *dual space*. This choice is motivated by the work of Kirby (2010) and will be discussed in the final part of this appendix.

Notation

For simplicity we work with scalar-valued functions $u : \Omega \to \mathbb{R}$ – the arguments here extend trivially to the vector-valued case $\boldsymbol{u} : \Omega \to \mathbb{R}^2$ used in the main text. For the vector-valued case in the main text we use the inverse of the finite element discretisation of the vector-valued Laplacian for the Riesz map L^{-1}, see Equation (A.20) .

We suppose that the space V_h is spanned by a finite set of basis functions $\{\varphi_i\}_{i=1}^n$ with $n = \dim V_h$. Then $u_h \in V_h$ can be represented as a linear combination of the basis functions

$$u_h = \sum_{i=1}^{n} u_i \varphi_i, \tag{A.10}$$

where the coefficients u_i forms the Euclidean vector $\mathsf{u} = [u_1, \dots, u_n]^\top \in \mathbb{R}^n$.

Following the notation of Kirby (2010), we introduce an interpolation operator $\mathcal{I}_h : \mathbb{R}^n \to V_h$ that maps the vector of coefficients u into the function $u_h \in V_h$

$$\mathcal{I}_h \mathsf{u} = u_h. \tag{A.11}$$

Similarly, the operator $\mathcal{I}_h' : \mathbb{R}^n \to V_h'$ maps the Euclidean vector $\mathsf{f} \in \mathbb{R}^n$ to the linear functional $f \in V_h'$ where V_h' is the dual space to V_h

$$\mathcal{I}_h' \mathsf{f} = f. \tag{A.12}$$

Implementation of the Taylor remainder test

The objective of the Taylor remainder test is to check that the computer implementation of the form $F : V_h \times V_h \to \mathbb{R}$ and its Jacobian $J : V_h \times V_h \times V_h \to \mathbb{R}$ in the direction $k\delta u_h \in V_h$ with $k > 0$ are consistent zeroth- and first-order approximations of the form F respectively. To this end, we introduce the operators $\mathcal{F} : V_h \to V_h'$ and $\mathcal{J} : V_h \to \mathcal{L}(V_h, V_h')$, where $\mathcal{L}(V_h, V_h')$ is a space of bounded linear operators from V_h to V_h'. These operators are linked with the forms F and J in the following way respectively

$$\langle \mathcal{F}(u_h), v_h \rangle := F(u_h; v_h), \tag{A.13}$$

$$\langle (\mathcal{J}(u_h))(k\delta u_h), v_h \rangle := J(u_h; k\delta u_h, v_h), \tag{A.14}$$

where, as in the main text, $\langle \cdot, \cdot \rangle$ is the $V_h' \times V_h$ duality pairing and the semicolon is used to emphasize that the forms are nonlinear in the first argument and linear in the remaining ones.

We assume the functional $\mathcal{F}$ is once Fréchet-differentiable allowing for the application of the Taylor (remainder) theorem on Banach spaces (Blanchard and Brüning 2015, p. 524). This allows us to establish the first-order convergence rate in k of the dual norm of the zeroth-order Taylor remainder r_k^0

$$\|r_k^0\|_{V_h'} := \|\mathcal{F}(u_h + k\,\delta u_h) - \mathcal{F}(u_h)\|_{V_h'} \underset{k\to 0}{\longrightarrow} 0 \text{ at } O(k), \tag{A.15}$$

and the second-order convergence rate in k of the dual norm of the first-order Taylor remainder r_k^1

$$\|r_k^1\|_{V_h'} := \|\mathcal{F}(u_h + k\,\delta u_h) - \mathcal{F}(u_h) - (\mathcal{J}(u_h))(k\delta u_h)\|_{V_h'} \underset{k\to 0}{\longrightarrow} 0 \text{ at } O(k^2). \tag{A.16}$$

In practice, to compute the norm in the dual space V_h' we apply the Riesz representation theorem (Riesz 1907; Axler 2020). The theorem states that there is a linear isometric isomorphism $\mathcal{R} : V' \to V$, which associates a linear functional $f \in V'$ with a unique element $\mathcal{R}f = u \in V$ such that

$$\langle f, v \rangle = (u, v), \quad \forall v \in V, \tag{A.17}$$

where $(\cdot, \cdot)$ is a standard inner product in V. Moreover, the norms satisfy the equality

$$\|f\|_{V'} = \|u\|_V. \tag{A.18}$$

If $V = \{v \in H^1(\Omega) : v|_\Gamma = 0\}$, where the subset $\Gamma \subset \partial\Omega$ of the boundary of the domain Ω has positive measure $\mu(\Gamma) > 0$ then the Riesz map can be defined through the following weak form of the Laplace operator (Kirby 2010, p. 273)

$$\langle f, v \rangle = (\mathcal{R}f, v) = \int_\Omega \nabla(\mathcal{R}f) \cdot \nabla v \,\mathrm{d}x, \quad \forall v \in V, \tag{A.19}$$

from where, for $V_h \subset V$, we obtain that the Riesz map is the matrix L^{-1} representing the inverse Laplace operator, which is defined as

$$\mathsf{L}_{ij} = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}x, \quad i, j = 1, \ldots, n. \tag{A.20}$$

Thus, if $f \in V_h'$ and $\mathsf{f} := (\mathcal{I}_h')^{-1} f \in \mathbb{R}^n$ then the Riesz representer $\mathcal{R}f = \mathcal{I}_h(\mathsf{L}^{-1}\mathsf{f})$.

The Riesz representation theorem leads us to the following formula expressing the norm of the linear functional $f \in V_h'$ from the finite dual space V_h' through the Riesz matrix L^{-1} (Kirby 2010, p. 281)

$$\|f\|_{V_h'}^2 = \mathsf{f}^\top \mathsf{L}^{-1} \mathsf{f}. \tag{A.21}$$

Now we introduce the vectors of coefficients $\mathsf{r}_k^i = \mathcal{I}_h^{-1} r_k^i \in \mathbb{R}^n$, $i \in \{0, 1\}$ of the Taylor remainders into Equations (A.15) and (A.16)

$$\mathsf{r}_k^0 = \mathsf{F}(\mathsf{u} + k\,\delta\mathsf{u}) - \mathsf{F}(\mathsf{u}) \in \mathbb{R}^n, \tag{A.22}$$

$$\mathsf{r}_k^1 = \mathsf{F}(\mathsf{u} + k\,\delta\mathsf{u}) - \mathsf{F}(\mathsf{u}) - \mathsf{J}(\mathsf{u}) \cdot k\delta\mathsf{u} \in \mathbb{R}^n, \tag{A.23}$$

where $\delta\mathsf{u} = \mathcal{I}_h^{-1}\delta u_h$, $\mathsf{F}(\cdot) = (\mathcal{I}_h')^{-1}\mathcal{F}(\cdot)$ and $\mathsf{J}(\cdot) = (\mathcal{I}_h')^{-1}\mathcal{J}(\cdot)\mathcal{I}_h$ (Kirby 2010, p. 280). Finally, by combining Equations (A.21) to (A.23), we can derive expressions for the norms of the remainders in the dual space V_h'

$$\|r_k^i\|_{V_h'}^2 = (\mathsf{r}_k^i)^\top \mathsf{L}^{-1} \mathsf{r}_k^i, \quad i \in \{0, 1\}. \tag{A.24}$$

Summary

The overall implementation of the mesh-independent Taylor remainder test presented in this work consists of the following steps:

1. Fix the Euclidean vectors u and $\delta\mathsf{u}$ such that $u_h = \mathcal{I}_h\mathsf{u} \in V_h$ and $\delta u_h = \mathcal{I}_h\delta\mathsf{u} \in V_h$.
2. Compute the Euclidean vector $\mathsf{F}(\mathsf{u})$ and the matrix $\mathsf{J}(\mathsf{u})$.
3. Compute the matrix L defined in Equation (A.20).
4. For each $k > 0$:
   (a) Compute the vectors r_k^0 and r_k^1 following Equation (A.22) and Equation (A.23).
   (b) Solve the linear systems $\mathsf{L}\mathsf{y} = \mathsf{r}_k^i$, $i \in \{0, 1\}$ (i.e. apply the Riesz map).
   (c) Compute the norms of the Taylor remainders as $\|r_k^i\|_{V_h'}^2 = (\mathsf{r}_k^i)^\top \cdot \mathsf{y}$, $i \in \{0, 1\}$.

Dual vs Euclidean norms

The Taylor theorem could be applied directly to the form $F : V_h \times V_h \to \mathbb{R}$ or the vector-function $\mathsf{F} : \mathbb{R}^n \to \mathbb{R}^n$ representing the operator $\mathcal{F} : V_h \to V_h'$ but such a choice would lead us to the use of mesh-dependent Euclidean norms for the correspondent Taylor remainders.

As demonstrated by Kirby (2010), the Riesz map serves as the "simplest" preconditioner for iterative solvers, providing mesh-independent convergence rates when computed through dual norms. We apply the same idea to the Taylor remainder test, where the question on the choice for the norm of Taylor remainders naturally arise. In practice, we often work with the Euclidean vector $\mathsf{F}(\mathsf{u})$, which represents an element from the dual space. Hence, applying the dual norm to this object through the Riesz matrix results in a mesh-independent object. From this perspective, considering the operator $\mathcal{F}$ within the Taylor remainder test is more convenient as it inherently encourages the use of dual norms, in contrast to the application of the Taylor theorem to the form F or the vector-function F. Thus, as the work Kirby (2010) demonstrates that when Euclidean representations encode Hilbert space objects, it is crucial to account for the functional nature of these representations to achieve mesh-independent estimates.

References

[SW] Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* 2015. SWHID: ⟨swh:1:snp:9b95c9f287721757b29324b83c69d602eeofc70a;origin=https://github.com/tensorflow/tensorflow⟩.

Abatour, M., K. Ammar, S. Forest, C. Ovalle, N. Osipov, and S. Quilici (2024). A generic formulation of anisotropic thermo-elastoviscoplasticity at finite deformations for finite element codes. *Computational Mechanics*. [DOI], [OA].

Abbo, A. and S. Sloan (1995). A Smooth Hyperbolic Approximation to the Mohr-Coulomb Yield Criterion. *Computers & Structures* 54(3):427–441. [DOI].

Alnæs, M., J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells (2015). The FEniCS Project Version 1.5. *Archive of Numerical Software* 3(100):9–23. [DOI], [OA].

Alnæs, M. S., A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells (2014). Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations. *ACM Transactions on Mathematical Software* 40(2):1–37. [DOI].

Axler, S. (2020). *Measure, Integration & Real Analysis*. Vol. 282. Graduate Texts in Mathematics. Springer. [OA].

[SW] Baratta, I. A., J. P. Dean, J. S. Dokken, M. Habera, J. S. Hale, C. N. Richardson, M. E. Rognes, M. W. Scroggs, N. Sime, and G. N. Wells, *DOLFINx: The next Generation FEniCS Problem Solving Environment* 2023. SWHID: ⟨swh:1:snp:582c883a8982fa5723ce07749638493cff1e4b34;origin=https://github.com/FEniCS/dolfinx⟩.

Blanchard, P. and E. Brüning (2015). *Mathematical Methods in Physics: Distributions, Hilbert Space*

Operators, Variational Methods, and Applications in Quantum Physics. Vol. 69. Springer. 598 pp. [DOI].

[SW] Bleyer, J., *Fenics_optim – Convex Optimization Interface in FEniCS* version 2.0.1, 2022. DOI: 10.5281/zenodo.3604085, SWHID: ⟨swh:1:snp:bdeeb512425b5937ad9b305da5d6a1844f5bafe2;origin=https://gitlab.enpc.fr/navier-fenics/fenics-optim⟩.

[SW] Bleyer, J., *Dolfinx_materials: A Python Package for Advanced Material Modelling* version 0.3.0, 2024. DOI: 10.5281/zenodo.13882183, SWHID: ⟨swh:1:snp:21b985e61521c6838d98b93cb13ffa9debb9e35b;origin=https://github.com/bleyerj/dolfinx_materials⟩.

[SW] Bleyer, J., *Numerical tours of Computational Mechanics with FEniCSx* version 0.2, 2024. DOI: 10.5281/zenodo.10470942, SWHID: ⟨swh:1:snp:0a85fabcfea2a599c4c3c3d149c5b732dc183664;origin=https://github.com/bleyerj/comet-fenicsx⟩.

Blühdorn, J., N. R. Gauger, and M. Kabel (2022). AutoMat: Automatic Differentiation for Generalized Standard Materials on GPUs. *Computational Mechanics* 69(2):589–613. [DOI], [OA].

Bonnet, M., A. Frangi, and C. Rey (2014). *The Finite Element Method in Solid Mechanics*. McGraw-Hill Education. 352 pp. ISBN: 978-8838674464.

Bouziani, N. and D. A. Ham (2021). Escaping the Abstraction: A Foreign Function Interface for the Unified Form Language [UFL]. First Workshop on Differentiable Programming (NeurIPS 2021) (Dec. 13, 2021). [OA].

Brothers, M. D., J. T. Foster, and H. R. Millwater (2014). A Comparison of Different Methods for Calculating Tangent-Stiffness Matrices in a Massively Parallel Computational Peridynamics Code. *Computer Methods in Applied Mechanics and Engineering* 279:247–267. [DOI].

Brown, J., A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, V. Dobrev, Y. Dudouit, L. Ghaffari, T. Kolev, D. Medina, W. Pazner, T. Ratnayaka, J. Thompson, and S. Tomov (2021). libCEED: Fast algebra for high-order element-based discretizations. *Journal of Open Source Software* 6(63):2945. [DOI], [OA].

Brown, J., V. Barra, N. Beams, L. Ghaffari, M. Knepley, W. Moses, R. Shakeri, K. Stengel, J. L. Thompson, and J. Zhang (2022). *Performance Portable Solid Mechanics via Matrix-Free p-Multigrid*. [OA].

Bucalem, M. L. and K.-J. Bathe (2011). *The Mechanics of Solids and Structures - Hierarchical Modeling and the Finite Element Solution*. Springer. ISBN: 9783540264002.

Buche, M. R. and M. N. Silberstein (2020). Statistical mechanical constitutive theory of polymer networks: The inextricable links between distribution, behavior, and ensemble. *Physical Review E* 102(1):012501. [DOI], [ARXIV].

Chen, Q., J. T. Ostien, and G. Hansen (2014). Automatic Differentiation for Numerically Exact Computation of Tangent Operators in Small-and Large-Deformation Computational Inelasticity. 143rd Annual Meeting & Exhibition (San Diego, United States, Feb. 16–20, 2014). Springer, pp 289–296. [DOI].

Chen, W. F. and X. L. Liu (1990). *Limit Analysis in Soil Mechanics*. Vol. 52. Developments in Geotechnical Engineering. Elsevier Science. 477 pp. ISBN: 9780444598356.

Coussy, O. (2004). *Poromechanics*. Wiley. 320 pp. [DOI].

Diamond, S. and S. Boyd (2016). CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *Journal of Machine Learning Research* 17(83):1–5. [URL].

[SW] DMLC, *DLPack* version 1.0, 2024. Distributed (Deep) Machine Learning Community (DMLC). SWHID: ⟨swh:1:snp:54ada4296b2db9a6ff73188368711ee9ff0479bf;origin=https://github.com/dmlc/dlpack⟩.

Dummer, A., M. Neuner, P. Gamnitzer, and G. Hofstetter (2024). Robust and Efficient Implementation of Finite Strain Generalized Continuum Models for Material Failure: Analytical, Numerical, and Automatic Differentiation with Hyper-Dual Numbers. *Computer Methods in Applied Mechanics and Engineering* 426:116987. [DOI], [OA].

Ferry, J. D. (1980). *Viscoelastic Properties of Polymers*. John Wiley & Sons. ISBN: 9780471048947.

Feyel, F. (2003). A Multilevel Finite Element Method (FE2) to Describe the Response of Highly Non-Linear Structures Using Generalized Continua. *Computer Methods in Applied Mechanics and Engineering* 192(28-30):3233–3244. [DOI].

Frostig, R., M. J. Johnson, and C. Leary (2018). Compiling Machine Learning Programs via High-Level Tracing. *Systems for Machine Learning*. SysML Conference 2018 (Stanford, United

States, Mar. 31–Apr. 2, 2019). [HAL].

Fuhg, J. N., G. A. Padmanabha, N. Bouklas, B. Bahmani, W. Sun, N. N. Vlassis, M. Flaschel, P. Carrara, and L. De Lorenzis (2025). A review on data-driven constitutive laws for solids. *Archives of Computational Methods in Engineering* 32:1841–1883. [DOI], [ARXIV].

Griewank, A. and A. Walther (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics. 438 pp. [DOI].

Ham, D. A., P. H. J. Kelly, L. Mitchell, C. Cotter, R. C. Kirby, K. Sagiyama, N. Bouziani, S. Vorderwuelbecke, T. Gregory, J. Betteridge, D. R. Shapero, R. Nixon-Hill, C. Ward, P. E. Farrell, P. D. Brubeck, I. Marsden, T. H. Gibson, M. Homolya, T. Sun, A. T. T. McRae, F. Luporini, A. Gregory, M. Lange, S. W. Funke, F. Rathgeber, G.-T. Bercea, and G. R. Markall (2023). *Firedrake User Manual*.

Hecht, F. (2012). New Development in Freefem++. *Journal of Numerical Mathematics* 20(3-4):251–266. [DOI], [HAL].

Helfer, T., J. Bleyer, T. Frondelius, I. Yashchuk, T. Nagel, and D. Naumov (2020). The 'MFront-GenericInterfaceSupport' Project. *Journal of Open Source Software* 5(48):2003. [DOI], [OA].

Helfer, T., B. Michel, J.-M. Proix, M. Salvo, J. Sercombe, and M. Casella (2015). Introducing the Open-Source Mfront Code Generator: Application to Mechanical Behaviours and Material Knowledge Management within the PLEIADES Fuel Element Modelling Platform. *Computers & Mathematics with Applications* 70(5):994–1023. [DOI], [OA].

[SW] Helfer, T., B. Michel, J.-M. Proix, J. Sercombe, M. Casella, and M. Salvo, *Invariant-based implementation of the Mohr-Coulomb elasto-plastic model in OpenGeoSys using MFront* version 4.2.1, 2024. VCS: https://thelfer.github.io/tfel/web/MohrCoulomb.html.

Homolya, M., L. Mitchell, F. Luporini, and D. A. Ham (2018). TSFC: A Structure-Preserving Form Compiler. *SIAM Journal on Scientific Computing* 40(3):C401–C428. [DOI], [OA].

Kirby, R. C. (2010). From Functional Analysis to Iterative Methods. *SIAM Review* 52(2):269–293. [DOI].

Kirby, R. C. and A. Logg (2006). A Compiler for Variational Forms. *ACM Transactions on Mathematical Software* 32(3):417–444. [DOI], [ARXIV].

Korelc, J. (1997). Automatic Generation of Finite-Element Code by Simultaneous Optimization of Expressions. *Theoretical Computer Science* 187(1-2):231–248. [DOI], [OA].

Korelc, J. and P. Wriggers (2016). *Automation of Finite Element Methods*. Springer. 346 pp. [DOI].

Lam, S. K., A. Pitrou, and S. Seibert (2015). Numba: A LLVM-based Python JIT Compiler. Second Workshop on the LLVM Compiler Infrastructure in HPC (Austin, USA, Nov. 15–20, 2015). ACM, pp 1–6. [DOI].

Latyshev, A. (2024). *Accompanying dataset for the paper "Expressing general constitutive models in FEniCSx using external operators and algorithmic automatic differentiation"*. Dataset. [DOI].

[SW] Latyshev, A. and J. Bleyer, *Convex-Plasticity: Efficient Implementation of Plasticity Problems Resolution Using Convex Optimization Solvers Incorporated in a Finite Element Code* 2022. SWHID: ⟨swh:1:dir:577c47880e6dcd53d4b3180387062773ba117d35;origin=https://github.com/a-latyshev/convex-plasticity;visit=swh:1:snp:5dcee27135e59813050e3ab8814dad6ffb9ba285;anchor=swh:1:rev:8d69fae3b57871a0245ac678cf85f0d864e2577a⟩.

Latyshev, A., J. Bleyer, J. Hale, and C. Maurini (2024). A Framework for Expressing General Constitutive Models in FEniCSx. 16e Colloque National en Calcul de Structures (Giens, France, May 13–17, 2024). [HAL].

[SW] Latyshev, A. and J. S. Hale, *dolfinx-external-operator* 2024. SWHID: ⟨swh:1:dir:31f070937a900400249d32131cbefbb2d9c899f0;origin=https://github.com/a-latyshev/dolfinx-external-operator;visit=swh:1:snp:dffd7eccaof066de947eac3fc61ffd78f6aa24b7;anchor=swh:1:rev:53756daea149feca3bab36855dbf8cdd66a6c0c1⟩.

Lewandowski, K., D. Barbera, P. Blackwell, A. H. Roohi, I. Athanasiadis, A. McBride, P. Steinmann, C. Pearce, and Ł. Kaczmarczyk (2023). Multifield finite strain plasticity: Theory and numerics. *Computer Methods in Applied Mechanics and Engineering* 414:116101. [DOI], [OA].

Lindsay, A., R. Stogner, D. Gaston, D. Schwen, C. Matthews, W. Jiang, L. K. Aagesen, R. Carlsen, F. Kong, A. Slaughter, C. Permann, and R. Martineau (2021). Automatic Differentiation in MetaPhysicL and Its Applications in MOOSE. *Nuclear Technology* 207(7):905–922. [DOI], [OA].

Linka, K. and E. Kuhl (2023). A new family of Constitutive Artificial Neural Networks towards

automated model discovery. *Computer Methods in Applied Mechanics and Engineering* 403:115731. [DOI], [OA].

Logg, A., K. B. Ølgaard, M. E. Rognes, and G. N. Wells (2012). FFC: The FEniCS Form Compiler. *Automated Solution of Differential Equations by the Finite Element Method*. Vol. 84. Springer. Chap. 11, pp 227–238. [DOI].

Lucarini, S. and E. Martínez-Pañeda (2024). UMAT4COMSOL: An Abaqus user material (UMAT) subroutine wrapper for COMSOL. *Advances in Engineering Software* 190:103610. [DOI], [OA].

Lyness, J. N. (1968). Differentiation Formulas for Analytic Functions. *Mathematics of Computation* 22(102):352–362. [DOI].

Lyness, J. N. and C. B. Moler (1967). Numerical Differentiation of Analytic Functions. *SIAM Journal on Numerical Analysis* 4(2):202–210. [DOI].

Mandel, J. (1965). Généralisation de la théorie de plasticité de W. T. Koiter. *International Journal of Solids and Structures* 1(3):273–295. [DOI].

Masi, F., I. Stefanou, P. Vannucci, and V. Maffi-Berthier (2021). Thermodynamics-Based Artificial Neural Networks for Constitutive Modeling. *Journal of the Mechanics and Physics of Solids* 147:104277. [DOI], [OA].

Méric, L. and G. Cailletaud (1991). Single Crystal Modeling for Structural Calculations: Part 2—Finite Element Implementation. *Journal of Engineering Materials and Technology* 113(1):171–182. [DOI], [HAL].

Mukherjee, D., M. Rambausek, and K. Danas (2021). An Explicit Dissipative Model for Isotropic Hard Magnetorheological Elastomers. *Journal of the Mechanics and Physics of Solids* 151:104361. [DOI], [OA].

Ogden, R. W. (1997). *Non-linear Elastic Deformations*. Courier Corporation. ISBN: 9780486696485.

[SW] Ølgaard, K. B. and N. W. Garth, *FEniCS Solid Mechanics* 2017. SWHID: ⟨swh:1:snp:7a008f58ab9a65c8d3133e713f2afff31bc06b21;origin=https://bitbucket.org/fenics-apps/fenics-solid-mechanics⟩.

Rajagopal, K. and A. Srinivasa (2000). A thermodynamic frame work for rate type fluid models. *Journal of Non-Newtonian Fluid Mechanics* 88(3):207–227. [DOI].

Riesz, F. (1907). Sur une espèce de géométrie analytique des systèmes de fonctions sommables. FR. *Comptes rendus de l'Académie des Sciences* 144:1409–1411.

Rosenbusch, S. M., P. Diercks, V. Kindrachuk, and J. F. Unger (2024). Integrating custom constitutive models into FEniCSx: A versatile approach and case studies. *Advances in Engineering Software* 206:103922. [DOI], [OA].

Rothe, S. and S. Hartmann (2015). Automatic Differentiation for Stress and Consistent Tangent Computation. *Archive of Applied Mechanics* 85(8):1103–1125. [DOI].

Saether, E., V. Yamakov, and E. H. Glaessgen (2009). An embedded statistical method for coupling molecular dynamics and finite element analyses. *International Journal for Numerical Methods in Engineering* 78(11):1292–1319. [DOI].

Seidl, D. T. and B. N. Granzow (2022). Calibration of Elastoplastic Constitutive Model Parameters from Full-Field Data with Automatic Differentiation-Based Sensitivities. *International Journal for Numerical Methods in Engineering* 123(1):69–100. [DOI], [ARXIV].

Simo, J. C. and T. J. R. Hughes (1998). *Computational Inelasticity*. Springer-Verlag. 392 pp. [DOI].

Stainier, L., A. Leygue, and M. Ortiz (2019). Model-Free Data-Driven Methods in Mechanics: Material Data Identification and Solvers. *Computational Mechanics* 64(2):381–393. [DOI], [ARXIV].

Tanaka, M., D. Balzani, and J. Schröder (2016). Implementation of Incremental Variational Formulations Based on the Numerical Calculation of Derivatives Using Hyper Dual Numbers. *Computer Methods in Applied Mechanics and Engineering* 301:216–241. [DOI].

Thakolkaran, P., A. Joshi, Y. Zheng, M. Flaschel, L. De Lorenzis, and S. Kumar (2022). NN-EUCLID: Deep-learning Hyperelasticity without Stress Data. *Journal of the Mechanics and Physics of Solids* 169:105076. [DOI], [OA].

Tschoegl, N. W. (2012). Linear Viscoelastic Response. *The Phenomenological Theory of Linear Viscoelastic Behavior*. Springer. Chap. 2, pp 35–68. [DOI].

Ulloa, J., L. Stainier, M. Ortiz, and J. E. Andrade (2024). Data-Driven Micromorphic Mechanics for Materials with Strain Localization. *Computer Methods in Applied Mechanics and Engineering*

429:117180. [DOI], [ARXIV].

Vigliotti, A. and F. Auricchio (2021). Automatic Differentiation for Solid Mechanics. *Archives of Computational Methods in Engineering* 28(3):875–895. [DOI], [ARXIV].

Wang, X. and W. Hong (2012). A visco-poroelastic theory for polymeric gels. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 468(2148):3824–3841. [DOI], [OA].

Xue, T., S. Liao, Z. Gan, C. Park, X. Xie, W. K. Liu, and J. Cao (2023). JAX-FEM: A Differentiable GPU-accelerated 3D Finite Element Solver for Automatic Inverse Design and Mechanistic Data Science. *Computer Physics Communications* 291(108802):108802. [DOI], [OA].

[SW] Z-Set Software, *ZMAT* 2023.

Zhang, W., D. S. Li, T. Bui-Thanh, and M. S. Sacks (2022). Simulation of the 3D hyperelastic behavior of ventricular myocardium using a finite-element based neural-network approach. *Computer Methods in Applied Mechanics and Engineering* 394:114871. [DOI], [OA].

Zlatić, M., F. Rocha, L. Stainier, and M. Čanađija (2024). Data-Driven Methods for Computational Mechanics: A Fair Comparison between Neural Networks Based and Model-Free Approaches. *Computer Methods in Applied Mechanics and Engineering* 431:117289. [DOI], [ARXIV].

Authors' contributions AL: Conceptualisation, Formal analysis, Investigation, Methodology, Software, Validation, Visualisation, Writing - original draft, Writing - review and editing. JB: Conceptualisation, Methodology, Supervision (Masters thesis of AL), Validation, Writing - review and editing. CM: Conceptualisation, Supervision (Masters and PhD thesis of AL), Project administration, Writing - review and editing. JSH: Conceptualisation, Formal analysis, Funding acquisition, Methodology, Project administration, Software, Supervision (PhD thesis of AL), Writing - review and editing.

Supplementary Material The software framework presented in this article is available at the permalink doi:10.5281/zenodo.10907417. The repository includes von Mises and Mohr-Coulomb plasticity examples covered in the text as well as further fully documented examples. The Python scripts plotting the figures in this article from their data are available in (Latyshev 2024). The software solutions used in this work are also archived at the permalink swh:1:dir:31f070937a900400249d32131cbefbb2d9c899f0.

Acknowledgements The authors would like to thank Patrick E. Farrell for his valuable remarks on computing the dual norm in the Taylor remainder test and Jørgen S. Dokken for his contribution extending the software framework to codimension one mesh entities. We would like to thank the anonymous reviewers for their questions on incorporating traditional approaches for implementing constitutive models and execution on GPUs that led us to improve the discussion section.

Funding This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant reference PRIDE/21/16747448/MATHCODA. For the purpose of open access, and in fulfilment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

Competing interests The authors declare that they have no competing interests.

Journal's Note JTCAM remains neutral with regard to the content of the publication and institutional affiliations.