Acta Polytechnica **65**(6):640–654, 2025
https://doi.org/10.14311/AP.2025.65.0640

Published by the Czech Technical University in Prague

ROBUST IMPLEMENTATION OF ELASTOPLASTIC CONSTITUTIVE MODELS USING AUTOMATIC DIFFERENTIATION IN PYTORCH

Tomáš Janda[a,*], Michal Šejnoha[a], Alena Zemanová[b], Tereza Žalská[b]

[a] *Czech Technical University in Prague, Faculty of Civil Engineering, Department of Mechanics, Thákurova 7, 166 29 Prague, Czech Republic*

[b] *Czech Technical University in Prague, Faculty of Civil Engineering, Department of Geotechnics, Thákurova 7, 166 29 Prague, Czech Republic*

[*] corresponding author: tomas.janda@cvut.cz

Abstract. This paper explores the use of automatic differentiation (AD) for implementing complex elastoplastic constitutive models in finite element analysis. Traditional approaches require manually deriving and coding the derivatives of residual functions governing implicit stress return mapping, a process that becomes cumbersome for advanced models. We demonstrate how AD can simplify the implementation of consistent material stiffness operators and improve code maintainability by using PyTorch and its autograd functionality. A drained triaxial shear test is used to compare AD with manually coded and finite difference derivatives, highlighting the efficiency of the proposed approach. The example shows that AD simplifies code development and reduces the required source code by over 50 %. These findings support the use of AD as a practical approach for implementing and testing constitutive models, especially in the early development stages.

Keywords: Elastoplastic constitutive models, automatic differentiation, finite element analysis, implicit stress return mapping, Python programming language, PyTorch, Hardening soil model.

1. Introduction

Although constitutive material models based on the theory of plasticity continue to evolve to better capture the material behaviour, the strategy for incorporating these models into finite element programs remains largely unchanged. For complex models, stress return mapping cannot typically be formulated in a closed-form. In such a case, the standard approach is to express the stress return mapping implicitly and use Newton's method to update stress and hardening parameters. Implicit stress return mapping is usually accompanied by the formulation of the consistent elastoplastic operator, which significantly improves the convergence rate of the iterative solver at the structural level. Thus, the consistent tangent operator form the foundation of nonlinear finite element analysis, as first emphasised by Simo and Taylor [1].

Solving the implicit stress return mapping via Newton's method and formulating the consistent elastoplastic operator both require calculating the derivatives of the residual functions governing the stress update. The analytical derivation of operators is a lengthy process that is prone to errors. While manually deriving and implementing these derivatives is manageable for basic constitutive models, these steps become challenging for models with multiple yield criteria and diverse hardening rules.

For example, in geotechnical analysis, material models for soils often integrate several loosely coupled experimentally observed phenomena. These include stress-dependent stiffness, stress-dependent shear strength, shear-induced dilatancy influenced by the actual void ratio, and many others. Elastoplastic constitutive models incorporating such phenomena tend to have relatively complex structures, making the maintenance of partial derivatives of residual functions error-prone. This issue is further exacerbated when multiple versions of a material model are developed, implemented, and compared to each other.

Various strategies exist to simplify the derivation and implementation of derivatives. One approach is to define expressions in a computer algebra system, such as Maxima, SymPy, or Mathematica, and to obtain derivatives through automatic symbolic manipulation. The resulting expressions can often be exported as pseudo-code and directly integrated into the implementation. However, this approach is often limited to scalar expressions.

Derivatives can also be computed numerically using finite difference methods applied to existing expressions. Although this approach might be a competitive alternative to analytical derivatives for certain material models as discussed in [2], it relies on carefully estimating the truncation error, and therefore bringing yet another level of complexity to the implementation. Another application of numerical differentiation for large-strain problems is presented in [3].

This paper advocates an alternative approach based on automatic differentiation (AD) – a numerical technique that computes derivatives directly from the source code of a function by applying the chain rule,

step by step. It delivers exact derivatives without symbolic manipulations or finite differencing errors. Two modes of AD are commonly distinguished: in the *forward mode*, each operation of the function is accompanied by the evaluation of its derivative, so that both function values and derivatives are obtained together in a single pass; in the *reverse mode*, the function is evaluated first and the derivatives are then propagated backwards through the computational graph from the outputs to the inputs. It is worth noting that the efficiency of forward versus reverse AD depends on the ratio of inputs to outputs. In our Hardening Soil implementation, the residual system has essentially the same number of inputs and outputs, meaning that the cost of forward and reverse modes is comparable. However, in more complex constitutive settings, such as multiscale or viscoplastic models with many internal variables but only a few stress components as outputs, reverse mode is clearly advantageous, as it provides gradients with respect to all inputs in a single backward pass.

Early work by Lejeune et al. [4] applied the forward mode of AD combined with Taylor series expansions to elastoplasticity. Their approach was developed in the context of the Asymptotic Numerical Method, which requires high-order derivatives of the residual. Because plasticity laws can exhibit discontinuous tangents at elastic-plastic transitions, the method relied on a regularisation strategy to ensure sufficient smoothness, making the use of forward AD feasible for both small- and large-strain plasticity problems.

Chen et al. [5] used the forward mode of AD to compute consistent tangent operators in both small- and large-deformation inelasticity problems. Their numerical examples included an elasto-plastic cap model tested under plane strain and plane stress loading paths, as well as a hyperelasto-plastic hydride model at large strains. The results demonstrated that AD provides numerically exact sensitivities, and the use of AD-generated tangents recovers the expected quadratic convergence of Newton iterations, thereby removing the need for manually derived tangent operators.

Rothe and Hartmann [6] provided one of the first systematic comparisons between analytical, numerical (finite difference), and AD-generated tangents for three representative classes of constitutive models: finite-strain hyperelasticity, finite-strain elasto-viscoplasticity, and small-strain thermo-viscoplasticity. Using the OpenAD tool, they demonstrated that AD provides machine-accurate derivatives and restores the robustness of Newton's method convergence, while introducing only moderate additional computational overhead.

Vigliotti and Auricchio [7] further demonstrated the use of AD for residual and tangent computation in nonlinear solid mechanics, implementing forward AD via dual numbers in Julia. Their study focused on hyperelastic constitutive models, showing that dual-number AD can provide exact derivatives of residual forces and stiffness matrices with minimal implementation efforts.

Zwicke et al. [8] used the Tapenade AD tool to generate Jacobians for material models in large legacy Fortran finite element codes. Their work examined the automatic creation of element-level Jacobians via the forward mode of AD, avoiding direct handling of global sparsity by reusing the FEM assembly procedure. As demonstrated on viscoelastic flow problems with the Oldroyd-B constitutive model, the AD-generated tangents reproduced the quadratic convergence of Newton iterations, validating their correctness. While the performance was inferior to hand-coded derivatives, particularly due to inefficiencies in the differentiated code, their study highlighted the feasibility of incorporating AD into production FEM environments, with optimisations offering partial improvements.

Latyshev et al. [9] extended this line of research by introducing an external-operator framework in FEniCSx/DOLFINx, which allows constitutive models to be expressed in general-purpose languages (e.g. Numba, JAX). Their approach leverages algorithmic AD for tangent computations while avoiding the need to rewrite entire solvers, making the integration of complex constitutive models into automated FEM environments more practical.

In parallel, Korelc and Wriggers developed the AceGen framework [10], which integrates symbolic manipulation and algorithmic differentiation within Mathematica to automatically generate optimised finite element subroutines. AceGen has been widely adopted to produce user material routines for commercial FE codes such as Abaqus, ensuring consistent tangent operators without the need for manual derivations and providing a robust path for large-scale industrial applications.

Hillgärtner et al. [11] further demonstrated AD in commercial workflows by implementing hyperelastic and inelastic models (including the Mullins effect) in Abaqus UMATs using Tapenade-based forward AD. Their results confirmed that AD-generated tangents achieved Newton convergence comparable to manually derived operators, underlining the practical usability of AD in production FE software.

Seidl and Granzow [12] developed an AD-based framework for the calibration of elastoplastic constitutive models from full-field experimental data. They used the forward mode of AD to compute exact sensitivities of the residuals with respect to material parameters, which were then used in a Gauss-Newton optimisation scheme. As demonstrated on a J_2 plasticity model with isotropic hardening, the approach enabled robust identification of parameters directly from displacement fields obtained via digital image correlation. This study highlights another strength of AD, namely its ability to deliver accurate derivatives for parameter estimation and inverse problems in experimental mechanics.

Beyond constitutive modelling, AD has also been applied to multiphysics and structural coupling problems. Aulisa and Capodaglio [13] used AD for monolithic coupling of the material point method with FEM, while Shishir and Tabarrei [14] employed AD in the topology optimisation of coupled thermo-mechanical problems.

Blühdorn et al. [15] introduced the *AutoMat* framework, which combines automatic differentiation with GPU acceleration to evaluate generalised standard materials. In this approach, constitutive models are defined entirely by two potential functions (a Helmholtz free energy and a dissipation potential). In their implementation, however, only the free energy was used explicitly, while AD was applied to derive stresses, internal variable evolution laws, and consistent tangents.The authors demonstrated the integration of AutoMat into an FFT-based homogenisation scheme, using an elasto-viscoplastic composite as a benchmark problem. Their study emphasised not only the flexibility of defining new material models with minimal coding effort, but also the performance benefits of GPU parallelisation, showing that AD-generated tangents can achieve accuracy comparable to hand-derived ones while closing the runtime gap to conventional implementations.

FFT-based homogenisation is a widely used technique to compute the effective properties of heterogeneous materials by solving field equations on a regular grid in the Fourier domain. Pundir and Kammer [16] simplified this framework by employing automatic differentiation (using JAX) to obtain stresses and the consistent tangent operator directly from the constitutive definitions. Their approach removes the need for manual coding of tangents and was demonstrated for nonlinear elasticity and finite-strain plasticity in heterogeneous microstructures. This highlights how AD can greatly streamline the implementation of FFT-based methods in solid mechanics, while maintaining accuracy and computational efficiency.

The majority of the reviewed works may be considered classical implementations, where AD is used within standard FEM workflows to generate stresses and consistent tangents. By contrast, a more recent trend leverages machine learning frameworks, such as PyTorch or JAX, which move beyond classical implementations towards differentiable programming environments.

Beyond classical implementations, the use of machine learning frameworks for constitutive modelling has recently attracted attention. Masi et al. [17] proposed thermodynamics-based artificial neural networks (TANNs), where automatic differentiation ensures that stresses and consistent tangents are derived in full agreement with the underlying energy potentials. In our earlier work [18], we demonstrated that PyTorch's `autograd` engine can be directly employed to implement a complex elastoplastic constitutive model (the Hardening Soil model) and to obtain its consistent tangent without manual derivations. Shortly afterwards, Bleyer et al. [19] presented *Differentiable Constitutive Modeling with FEniCSx and JAX*, outlining a broader research agenda for differentiable constitutive modelling within ML-compatible environments, such as PyTorch or JAX, indicating the growing relevance of this direction.

Against this backdrop, this paper leverages the PyTorch package for Python. PyTorch is a well-maintained, state-of-the-art machine learning library offering a wide range of functions for multi-dimensional data manipulation. Its `torch.autograd` functionality automatically tracks derivatives of arbitrary chained expressions. Although primarily designed for implementation and training of neural networks [20–22], PyTorch's tensor algebra functions are general enough to make handling tensor quantities in constitutive models effortless.

Compared to the cited works, our focus is on demonstrating that a mainstream machine learning library can be used to implement multi-surface elastoplastic models with minimal coding effort. Unlike AceGen or other AD tools, which are highly efficient but often require specialised domain-specific languages, our approach leverages a widely adopted open-source machine learning framework, providing immediate accessibility for prototyping and teaching. Our work distinguishes itself in several ways. First, we implement the Hardening Soil model, a widely used multi-surface elastoplastic law in geomechanics, representing one of the most complex plasticity models treated with AD to date. Second, our use of PyTorch shows that such models can be implemented in a mainstream ML library without the need for domain-specific symbolic tools or AD compilers, thus lowering the barrier for adoption. Finally, our contribution is positioned as a lightweight research and teaching framework: it facilitates rapid prototyping of new constitutive models, verification of hand-derived tangents, and education in computational mechanics.

The rest of this paper is organised as follows: First, an implicit stress return mapping is formulated for the Hardening soil model along with the derivation of the consistent stiffness matrix. This section highlights the role of residual functions, primary variables, and the complex expression hierarchy within the model.

Next, the principles of automatic differentiation are introduced, and the gradient of an example composite expression is computed using PyTorch functions. The example demonstrates the efficient implementation of hierarchical tensor expressions and the automatic computation of its derivatives.

Finally, a drained triaxial shear test is simulated using the Hardening soil model. Three differentiation approaches are compared in this test: derivatives computed automatically using `PyTorch.autograd`, closed-form hard-coded derivatives, and numerical derivatives computed using the finite difference method.

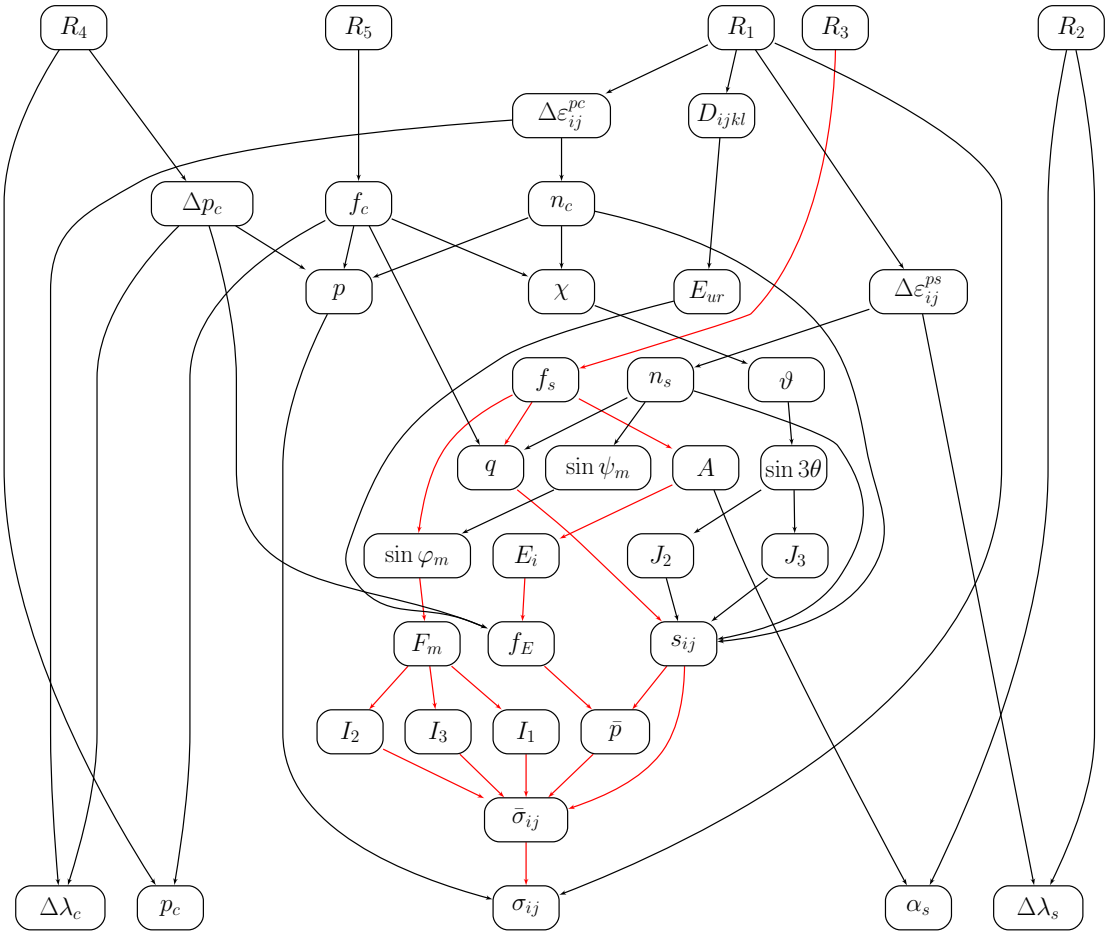

FIGURE 1. Graph showing the dependencies of the residual expression on the primary unknowns. Variables with indices $\bullet_{ij}$ and $\bullet_{ijkl}$ denote second- and fourth-order tensors, respectively.

2. METHODOLOGY

The methodology used in this paper is straightforward. First, the implicit stress return mapping and the corresponding consistent material stiffness matrix of the Hardening soil model are formulated. Then, the residual functions are implemented in Python using `PyTorch.Tensor` multidimensional arrays as the data type for tensor and scalar quantities. The Jacobian matrix for the residual functions and the consistent stiffness matrix are implemented using automatic differentiation in PyTorch, as well as hand-calculated derivatives.

A drained triaxial shear test is then simulated to evaluate the performance.

Notably, due to its complexity, the Hardening soil model particularly benefits from automatic differentiation. Nevertheless, the detailed formulation of the model is not essential for the discussion. Readers familiar with the principles of the implicit stress update for elastoplastic models can skip ahead to Section 2.1.4, where the internal structure of the model is summarised in Figure 1.

2.1. HARDENING SOIL MODEL

The stress return mapping and consistent stiffness matrix of the hardening soil model [23] are formulated in this section. Since its first formulation, the model underwent several modifications. In particular, the experimentally proven dependency of the shear strength on intermediate principal stress inspired by the Matsuoka-Nakai [24] yield criterion was incorporated in [25]. Furthermore, the yield function was refactored into a form that prevents singularities [26]. The formulation presented here builds on these variants of the model and assumes the elastic modulus dependent on the mean stress instead of the minor principal stress. This makes the response of the model slightly more consistent with the experimentally observed results of the drained triaxial shear test, especially in the unloading branch.

The governing equations specified in the following sections include the nonlinear stress-dependent Hooke's law, the yield functions, and the corresponding hardening laws. The model defines a cone-shaped yield surface mostly responsible for shear plastic strains and cap-shaped yield surface responsible for volumetric plastic strains.

The tension-positive sign convention for stress and strain is used in the paper. Einstein index notation and summation rule is used for tensor expressions. Tensor quantities in the role of function arguments are denoted with bold letters.

2.1.1. Elastic stress-strain relationship

When a strain increment is applied to the material, its stress evolves from the initial stress σ_{ij}^n to a new stress σ_{ij} according to the generalised Hooke's law written in the form:

$$\sigma_{ij} = D_{ijkl}\Delta\varepsilon_{kl}^e + \sigma_{ij}^n, \tag{1}$$

where $\Delta\varepsilon_{kl}^e$ is the elastic part of the strain increment and D_{ijkl} is the stress-dependent elastic stiffness tensor provided by:

$$D_{ijkl}(E_{ur}) = E_{ur}\bigg(\frac{\nu_{ur}}{(1+\nu_{ur})(1-2\nu_{ur})}\delta_{ij}\delta_{kl} + \frac{1}{2(1+\nu_{ur})}(\delta_{ik}\delta_{jl}+\delta_{il}\delta_{jk})\bigg). \tag{2}$$

The current Young's modulus for unloading-reloading E_{ur} is given as the reference Young's modulus multiplied by a stiffness factor:

$$E_{ur}(f_E) = E_{ur}^{\mathrm{ref}} f_E. \tag{3}$$

The stiffness factor depends on the offset mean stress $\bar{p}$ according to:

$$f_E(\bar{p}) = \left(\frac{\bar{p}}{\bar{p}^{\mathrm{ref}}}\right)^{m_p}, \tag{4}$$

where $\bar{p}^{\mathrm{ref}}$ and m_p are material parameters. The offset mean stress is calculated from the offset stress tensor:

$$\bar{p} = \frac{\delta_{ij}\bar{\sigma}_{ij}}{3} = \frac{\bar{\sigma}_{ii}}{3}, \tag{5}$$

which accounts for the cohesion c and the angle of internal friction φ according to:

$$\bar{\sigma}_{ij} = \sigma_{ij} - \frac{c}{\tan(\varphi)}\delta_{ij}. \tag{6}$$

The elastic strain increment is the difference between the prescribed total strain increment and the two plastic strain increments resulting from plastic yielding given the two yield surfaces:

$$\Delta\varepsilon_{kl}^e = \Delta\varepsilon_{kl} - \Delta\varepsilon_{kl}^{ps} - \Delta\varepsilon_{kl}^{pc}, \tag{7}$$

where the plastic strain increments $\Delta\varepsilon_{kl}^{ps}$ and $\Delta\varepsilon_{kl}^{pc}$ are defined in Sections 2.1.2 and 2.1.3, respectively.

2.1.2. Shear yield surface

Generally, the increment of the plastic strain attributed to the shear yield surface is given by the flow rule:

$$\Delta\varepsilon_{ij}^{ps}(\Delta\lambda, \boldsymbol{n}_s) = \Delta\lambda n_{s,ij}, \tag{8}$$

where $n_{s,ij}$ are the normals to the corresponding plastic potential. The plastic potential itself is defined as:

$$g_s(q, \bar{p}, \sin\psi_m) = q - \frac{6\sin\psi_m}{3-\sin\psi_m}\bar{p}, \tag{9}$$

where q is the equivalent deviatoric stress, see Equation (20) and ψ_m is the mobilised dilatation angle, both defined below. Note that the plastic potential is assumed to be of the Drucker-Prager type, i.e. it is independent of the Lode angle θ (or the third stress invariant I_3) and the direction of the stress return intersects with the hydrostatic axis. This implies that the derivatives of the plastic potential have to be taken with respect to $\bar{p}$ and q only and not with respect to ψ_m. This is because ψ_m ultimately depends on I_3, but the normals should be I_3-independent. Therefore, the components of the normal to the plastic potential become:

$$n_{s,ij} = \frac{\partial g_s}{\partial\bar{p}}\frac{\partial\bar{p}}{\partial\bar{\sigma}_{ij}} + \frac{\partial g_s}{\partial q}\frac{\partial q}{\partial s_{kl}}\left(\frac{\partial s_{kl}}{\partial\bar{\sigma}_{ij}} + \frac{\partial s_{kl}}{\partial\bar{p}}\frac{\partial\bar{p}}{\partial\bar{\sigma}_{ij}}\right). \tag{10}$$

The partial derivatives that appear in the above chain rule can be expressed as follows:

$$\frac{\partial g_s}{\partial\bar{p}} = -\frac{6\sin\psi_m}{3-\sin\psi_m}, \tag{11}$$

$$\frac{\partial\bar{p}}{\partial\bar{\sigma}_{ij}} = \frac{1}{3}\delta_{ij}, \tag{12}$$

$$\frac{\partial g_s}{\partial q} = 1, \tag{13}$$

$$\frac{\partial q}{\partial s_{kl}} = \frac{3}{2}\frac{s_{kl}}{q}, \tag{14}$$

$$\frac{\partial s_{kl}}{\partial\bar{\sigma}_{ij}} = \delta_{ik}\delta_{jl}, \tag{15}$$

$$\frac{\partial s_{kl}}{\partial\bar{p}} = -\delta_{kl}, \tag{16}$$

and therefore the normals are written as:

$$n_{s,ij}(\boldsymbol{s}, q, \sin\psi_m) = -\frac{2\sin\psi_m}{3-\sin\psi_m}\delta_{ij} + \frac{3}{2}\frac{s_{kl}}{q}\delta_{ik}\delta_{jl} - \frac{1}{2q}s_{kl}\delta_{kl}\delta_{ij} \tag{17}$$

$$= -\frac{2\sin\psi_m}{3-\sin\psi_m}\delta_{ij} + \frac{3}{2}\frac{s_{ij}}{q} - \frac{1}{2q}s_{kk}\delta_{ij} \tag{18}$$

$$= -\frac{2\sin\psi_m}{3-\sin\psi_m}\delta_{ij} + \frac{3}{2}\frac{s_{ij}}{q}, \tag{19}$$

since the trace of the deviatoric stress tensor $s_{kk} = 0$. This also agrees with the fact that q is independent of p, and therefore $\frac{dq}{dp} = \frac{dq}{ds_{ij}}\frac{ds_{ij}}{dp} = 0$.

The equivalent deviatoric stress reads:

$$q(\boldsymbol{s}) = \sqrt{\frac{3}{2}s_{ij}s_{ij}}, \tag{20}$$

with the deviatoric part of the stress tensor being written in terms of offset stress as:

$$s_{ij}(\bar{\boldsymbol{\sigma}}, \bar{p}) = \bar{\sigma}_{ij} - \bar{p}\delta_{ij}, \tag{21}$$

or in terms of the standard stress tensor and mean stress as:

$$s_{ij}(\boldsymbol{\sigma}, p) = \sigma_{ij} - p\delta_{ij}. \tag{22}$$

The mobilised dilatation angle is defined by:

$$\sin\psi_m(\sin\varphi_m) = \frac{\sin\varphi_m - \sin\varphi_{cs}}{1-\sin\varphi_m\sin\varphi_{cs}}, \tag{23}$$

where the friction angle at critical state φ_{cs} is constant and depends only on the material parameters:

$$\sin\varphi_{cs} = \frac{\sin\varphi - \sin\psi}{1 - \sin\varphi\sin\psi}. \tag{24}$$

The mobilised angle of internal friction depends on the so-called Matsuoka-Nakai stress factor according to:

$$\sin\varphi_m(F_m) = \sqrt{F_m}. \tag{25}$$

The stress factor on the stress invariants I_1, I_2, and I_3 and is given by:

$$F_m(I_1, I_2, I_3) = \frac{9I_3 - I_1 I_2}{I_3 - I_1 I_2}, \tag{26}$$

with the stress invariants defined as:

$$I_1(\bar{\boldsymbol{\sigma}}) = \bar{\sigma}_{ii}, \tag{27}$$

$$I_2(\bar{\boldsymbol{\sigma}}) = \frac{1}{2}(\bar{\sigma}_{ii}\bar{\sigma}_{jj} - \bar{\sigma}_{ij}\bar{\sigma}_{ij}), \tag{28}$$

$$\begin{aligned} I_3(\bar{\boldsymbol{\sigma}}) &= \frac{1}{6}(\bar{\sigma}_{ii}\bar{\sigma}_{jj}\bar{\sigma}_{kk} + 2\bar{\sigma}_{ij}\bar{\sigma}_{jk}\bar{\sigma}_{ki} - 3\bar{\sigma}_{ij}\bar{\sigma}_{ji}\bar{\sigma}_{kk}) \\ &= \det(\bar{\boldsymbol{\sigma}}). \end{aligned} \tag{29}$$

The original cone-shaped yield function is refactored into the form:

$$\begin{aligned} &f_s(q, \sin\varphi_m, A) \\ &\quad = q - \left(1 - R_f \frac{1 - \sin\varphi}{1 - \sin\varphi_m}\frac{\sin\varphi_m}{\sin\varphi}\right) \\ &\qquad \times \left(\frac{E_i^{\mathrm{ref}}}{E_{ur}^{\mathrm{ref}}} q + A\right), \end{aligned} \tag{30}$$

where R_f is the model parameter and the stress-like hardening variable A relates to the strain-like hardening variable α_s via the relation:

$$A(E_i, \alpha_s) = E_i \alpha_s, \tag{31}$$

with the α_s being one of the primary unknowns of the implicit stress return procedure. The stress-dependent modulus for the loading branch E_i depends on the same stiffness factor f_E as the unloading modulus E_{ur} and takes the form:

$$E_i(f_E) = E_i^{\mathrm{ref}} f_E. \tag{32}$$

Finally, the evolution of α_s is given by:

$$\alpha_s = \alpha_{s,n} + \Delta\lambda_s H_s. \tag{33}$$

As shown in Appendix A, the relation of the strain-like hardening parameter α_s to the plastic strain increment is made such that $H_s = 1$.

2.1.3. Cap yield surface

The flow rule associated with the cap yield surface reads:

$$\Delta\varepsilon_{ij}^{pc} = \Delta\lambda_c n_{c,ij}. \tag{34}$$

The plastic potential is assumed in the form:

$$g_c(p, q, p_c, \chi) = \frac{q^2}{(M\chi)^2} + p^2 - p_c^2, \tag{35}$$

where M is the model parameter, the function χ accounts for the dependency on Lode's angle, see Equation (45). The preconsolidation pressure p_c plays the role of a hardening parameter. Because the flow direction in the deviatoric plane is assumed to be radial, the plastic potential is differentiated with respect to p and q only and not with respect to χ to obtain the flow direction $n_{c,ij}$. The subscript c denotes the flow direction on the cap yield surface. It takes the form:

$$n_{c,ij} = \frac{\partial g_c}{\partial p}\frac{\partial p}{\partial \sigma_{ij}} + \frac{\partial g_c}{\partial q}\frac{\partial q}{\partial s_{kl}}\left(\frac{\partial s_{kl}}{\partial \sigma_{ij}} + \frac{\partial s_{kl}}{\partial p}\frac{\partial p}{\partial \sigma_{ij}}\right). \tag{36}$$

The partial derivatives appearing in the above chain rule are expressed as follows:

$$\frac{\partial g_c}{\partial p} = 2p, \tag{37}$$

$$\frac{\partial p}{\partial \sigma_{ij}} = \frac{1}{3}\delta_{ij}, \tag{38}$$

$$\frac{\partial g_c}{\partial q} = \frac{2q}{(M\chi)^2}, \tag{39}$$

$$\frac{\partial q}{\partial s_{kl}} = \frac{3}{2}\frac{s_{kl}}{q}, \tag{40}$$

$$\frac{\partial s_{kl}}{\partial \sigma_{ij}} = \delta_{ik}\delta_{jl}, \tag{41}$$

$$\frac{\partial s_{kl}}{\partial \bar{p}} = -\delta_{kl}. \tag{42}$$

Therefore, the flow direction is written as:

$$\begin{aligned} n_{c,ij}(p, \boldsymbol{s}, \chi) &= \frac{2}{3}p\delta_{ij} + \frac{3s_{ij}}{(M\chi)^2} - \frac{s_{kl}}{(M\chi)^2}\delta_{kl}\delta_{ij} \\ &= \frac{2}{3}p\delta_{ij} + \frac{3s_{ij}}{(M\chi)^2}. \end{aligned} \tag{43}$$

Note that the last term was omitted from the above expression since $s_{ij}\delta_{ij} = 0$. Also note that the flow direction does not depend on p_c.

The mean stress calculated from the stress tensor σ_{ij} is as follows:

$$p = \frac{\delta_{ij}\sigma_{ij}}{3} = \frac{\sigma_{ii}}{3}. \tag{44}$$

The function χ, which complies with the projection of the Matsuoka-Nakai yield surface into the deviatoric plane, takes the form, see e.g. [25]:

$$\chi(\vartheta) = \frac{\sqrt{3}\beta}{2\sqrt{\beta^2 - \beta + 1}}\frac{1}{\cos\vartheta}, \tag{45}$$

where parameter β depends only on the angle of internal friction according to:

$$\beta = \frac{3 - \sin\varphi}{3 + \sin\varphi}. \tag{46}$$

The value of β is found between $\beta(\varphi = 0) = 1$ and $\beta(\varphi = \frac{\pi}{2}) = \frac{1}{2}$. The auxiliary parameter ϑ is defined as:

$$\vartheta(\sin 3\theta) = \begin{cases} \frac{1}{6}\arccos\left(-1 + c_\beta \sin^2 3\theta\right) & \text{for } \theta \leq 0, \\ \frac{\pi}{3} - \frac{1}{6}\arccos\left(-1 + c_\beta \sin^2 3\theta\right) & \text{for } \theta > 0, \end{cases} \tag{47}$$

where the constant c_β is defined, for convenience, as:

$$c_\beta = \frac{27\beta^2(1-\beta)^2}{2(\beta^2 - \beta + 1)^3}, \tag{48}$$

and ranges from $c_\beta(\beta(\varphi = 0)) = 0$ to $c_\beta(\beta(\varphi = \frac{\pi}{2}))$=2. The Lode angle θ is not directly evaluated. Instead, the expression $\sin 3\theta$ is assumed to be a function of the second and third invariants of the deviatoric part of the stress tensor:

$$\sin 3\theta = -\frac{3\sqrt{3}J_3}{2J_2^{\frac{3}{2}}}. \tag{49}$$

The invariants of the deviatoric part of the stress tensor are:

$$J_2 = \frac{1}{2}s_{ij}s_{ij}, \tag{50}$$

$$J_3 = \frac{1}{3}s_{ij}s_{jk}s_{ki}. \tag{51}$$

The cap yield function comes in the following form, and for a comparison, see [25]:

$$f_c(p, q, \chi, p_c) = \frac{q^2}{(M\chi)^2} + p^2 - p_c^2. \tag{52}$$

Finally, the evolution of the preconsolidation pressure p_c, which plays the role of the hardening variable for the cap yield surface, follows from:

$$\Delta p_c(p, f_E, \Delta\lambda_c) = -2pHf_E\Delta\lambda_c, \tag{53}$$

where H is the model parameter.

2.1.4. Residuals

The implicit stress return mapping is formulated in terms of residuals. The residual functions are composed of a hierarchy of the expressions defined in the previous sections and ultimately depend on primary unknowns σ_{ij}, α_s, $\Delta\lambda_s$, p_c, $\Delta\lambda_c$. The residual functions correspond to Hooke's law (1), the hardening law (33) and the yield function (30) for the shear yield surface and the hardening law (53), and the yield function (52) for the cap-yield surface, written as:

$$\begin{aligned} R_{1,ij}(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) &= \sigma_{ij} - D_{ijkl}\Delta\varepsilon^e_{kl} \\ &\quad - \sigma^n_{ij}, \\ &= 0, \end{aligned} \tag{54}$$

$$\begin{aligned} R_2(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) &= \alpha_s - \alpha_{s,n} - \Delta\lambda_s H_s \\ &= 0, \end{aligned} \tag{55}$$

$$R_3(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) = f_s = 0, \tag{56}$$

$$\begin{aligned} R_4(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) &= p_c - p_{c,n} - \Delta p_c \\ &= 0, \end{aligned} \tag{57}$$

$$R_5(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) = f_c = 0. \tag{58}$$

Figure 1 shows the graph representing the dependencies of individual residual expressions on intermediate expressions and primary unknowns.

The roots of the residual expressions are traditionally found using the Newton method, and therefore the partial derivatives of the residual function with respect to all primary unknowns are needed.

As shown in the following section, the partial derivatives of the residual function with respect to the primary unknowns can be calculated automatically using the `torch.autograd` package. Nevertheless, for comparison purposes, the partial derivatives were also derived and implemented manually as part of the function evaluating the residual expressions. To illustrate this, an example of the chain rule for the partial derivative of R_3 with respect to σ_{ij} is expressed as follows:

$$\begin{aligned} \frac{\mathrm{d}R_3}{\mathrm{d}\sigma_{ij}} = \frac{\partial R_3}{\partial f_s}\Bigg(&\frac{\partial f_s}{\partial \sin\varphi_m}\frac{\partial \sin\varphi_m}{\partial F_m}\left(\frac{\partial F_m}{\partial I_1}\frac{\partial I_1}{\partial \bar{\sigma}_{kl}}\right. \\ &\left. + \frac{\partial F_m}{\partial I_2}\frac{\partial I_2}{\partial \bar{\sigma}_{kl}} + \frac{\partial F_m}{\partial I_3}\frac{\partial I_3}{\partial \bar{\sigma}_{kl}}\right) \\ &+ \frac{\partial f_s}{\partial q}\frac{\partial q}{\partial s_{mn}}\left(\frac{\partial s_{mn}}{\partial \bar{\sigma}_{kl}} + \frac{\partial s_{mn}}{\partial \bar{p}}\frac{\partial \bar{p}}{\partial \bar{\sigma}_{kl}}\right) \\ &+ \frac{\partial f_s}{\partial A}\frac{\partial A}{\partial E_i}\frac{\partial E_i}{\partial f_E}\frac{\partial f_E}{\partial \bar{p}}\frac{\partial \bar{p}}{\partial \bar{\sigma}_{kl}}\Bigg)\frac{\partial \bar{\sigma}_{kl}}{\partial \sigma_{ij}}. \end{aligned} \tag{59}$$

Note that each partial derivative corresponds to one of the arrows in Figure 1 highlighted in red.

2.1.5. Consistent elastoplastic operator

The tangent elastoplastic operator consistent with the implicit return mapping outlined above is formulated in this section. The approach builds on [1] and follows the procedure outlined in [27, Section 7.4].

The implicit stress return algorithm, i.e. the process of solving Equations (54) to (58) iteratively by Newton's method, is conceptually seen as a function accepting the strain increment $\Delta\varepsilon_{ij}$ as its argument and returning the updated stress σ_{ij} and the other primary unknowns. Since the increment of the strain appears only in the first residual $R_{1,ij}$, the residuals are formally written as:

$$R_{1,ij}(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c, \Delta\boldsymbol{\varepsilon}) = 0, \tag{60}$$

$$R_2(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) = 0, \tag{61}$$

$$R_3(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) = 0, \tag{62}$$

$$R_4(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) = 0, \tag{63}$$

$$R_5(\boldsymbol{\sigma}, \alpha_s, \Delta\lambda_s, p_c, \Delta\lambda_c) = 0. \tag{64}$$

These residuals are zero at the end of the stress update for any strain increment $\Delta\boldsymbol{\varepsilon}$. Therefore, the total differential of the residuals is also zero. Expressing the total differential in the matrix form and moving the derivative of the first residual with respect to strain increment to the right-hand side of the first

equation gives:

$$[\boldsymbol{J}] \begin{Bmatrix} \mathrm{d}\boldsymbol{\sigma} \\ \mathrm{d}\alpha_s \\ \mathrm{d}\Delta\lambda_s \\ \mathrm{d}p_c \\ \mathrm{d}\Delta\lambda_c \end{Bmatrix} = \begin{Bmatrix} -\frac{\partial R_{1,ij}}{\partial \Delta\varepsilon_{kl}} \mathrm{d}\Delta\varepsilon_{kl} \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}, \tag{65}$$

where:

$$[\boldsymbol{J}] = \begin{bmatrix} \frac{\partial R_{1,ij}}{\partial \sigma_{kl}} & \frac{\partial R_{1,ij}}{\partial \alpha_s} & \frac{\partial R_{1,ij}}{\partial \Delta\lambda_s} & \frac{\partial R_{1,ij}}{\partial p_c} & \frac{\partial R_{1,ij}}{\partial \Delta\lambda_c} \\ \frac{\partial R_2}{\partial \sigma_{kl}} & \frac{\partial R_2}{\partial \alpha_s} & \frac{\partial R_2}{\partial \Delta\lambda_s} & \frac{\partial R_2}{\partial p_c} & \frac{\partial R_2}{\partial \Delta\lambda_c} \\ \frac{\partial R_3}{\partial \sigma_{kl}} & \frac{\partial R_3}{\partial \alpha_s} & \frac{\partial R_3}{\partial \Delta\lambda_s} & \frac{\partial R_3}{\partial p_c} & \frac{\partial R_3}{\partial \Delta\lambda_c} \\ \frac{\partial R_4}{\partial \sigma_{kl}} & \frac{\partial R_4}{\partial \alpha_s} & \frac{\partial R_4}{\partial \Delta\lambda_s} & \frac{\partial R_4}{\partial p_c} & \frac{\partial R_4}{\partial \Delta\lambda_c} \\ \frac{\partial R_5}{\partial \sigma_{kl}} & \frac{\partial R_5}{\partial \alpha_s} & \frac{\partial R_5}{\partial \Delta\lambda_s} & \frac{\partial R_5}{\partial p_c} & \frac{\partial R_5}{\partial \Delta\lambda_c} \end{bmatrix} \tag{66}$$

is the Jacobian matrix used in Newton's method within the stress update procedure. Note that the elements in the first row and the first column are tensor quantities. These are, in fact, appropriately reshaped to form blocks within the actual 13×13 Jacobian matrix. Multiplying both sides of Equation (65) by $[\boldsymbol{J}]^{-1}$ gives:

$$\begin{Bmatrix} \mathrm{d}\boldsymbol{\sigma} \\ \mathrm{d}\alpha_s \\ \mathrm{d}\Delta\lambda_s \\ \mathrm{d}p_c \\ \mathrm{d}\Delta\lambda_c \end{Bmatrix} = [\boldsymbol{J}]^{-1} \begin{Bmatrix} -\frac{\partial R_{1,ij}}{\partial \Delta\varepsilon_{kl}} \mathrm{d}\Delta\varepsilon_{kl} \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}. \tag{67}$$

The first row of the above system of equations is written as:

$$\mathrm{d}\sigma_{ij} = -A_{ijmn} \frac{\partial R_{1,mn}}{\partial \Delta\varepsilon_{kl}} \mathrm{d}\Delta\varepsilon_{kl}, \tag{68}$$

where A_{ijmn} is the fourth-order tensor reshaped back from the 9×9 upper left block of the $\boldsymbol{J}^{-1}$ matrix.

2.2. Automatic differentiation example

This section provides an example of how the derivatives are computed automatically using the `torch.autograd` [28] package of PyTorch framework [29]. PyTorch was chosen since it satisfies the following requirements that make the use of automatic differentiation extremely easy:

- It works with hierarchical expressions where the subexpressions are implemented in separate functions. This makes the code much more reusable.
- Multidimensional functions of multidimensional arguments are supported.
- The framework supports higher level operations on multidimensional data types almost identical to NumPy [30] which is the de-facto standard numerical library in the Python ecosystem. For example, PyTorch supports expressions using Einstein index notation, eigenvalue decomposition, and the calculation of the determinants.

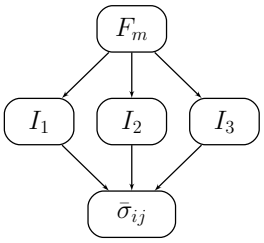

Figure 2. A subgraph of the composite expression $F_m(\bar{\boldsymbol{\sigma}})$.

The presented implementation relies on the `torch.autograd` package. This package dynamically builds a computational graph of operations performed on scalar or tensors variables. In principle, this graph is a much more detailed version of the graph shown in Figure 1, in which the nodes represent all intermediate values and the arrows represent elementary operations or function applications. Each intermediate value keeps track of how it was created, and `torch.autograd` then uses the chain rule of calculus to express gradients by backpropagating through this graph. Once gradients are computed, they are stored in an attribute of these values. For interested reader, the principles of implementing AD in PyTorch are described in [28], while an accessible introduction to the general principles of AD is provided in [31].

Here, the application of AD on composite expression involving tensor variables is illustrated here on a simple example. The following is a minimal example that still highlights the convenient properties of AD in PyTorch:

(1.) Expressions are expressed in the form of simple functions and composed together into more convoluted expressions,

(2.) high-level operations on tensors are possible, such as the use of Einstein summation notation,

(3.) gradient is calculated automatically for the final composite expression.

Consider only the small part of the computation graph in Figure 1, namely the expression of the Matsuoka-Nakai stress factor F_m given in Equation (26) and how it depends on the offset stress $\bar{\sigma}_{ij}$. The subgraph is shown in Figure 2. The factor can be viewed as a function of three stress invariants I_1, I_2, and I_3. The stress invariants I_1, I_2, and I_3 are defined in Equations (27), (28), and (29), respectively. And these can be seen as functions of the stress tensor $\bar{\sigma}_{ij}$. These expressions are only rewritten using the convenient PyTorch functions `einsum()` and `det()`:

```
from torch import Tensor, einsum, det
def Fm(I1: Tensor, I2: Tensor, I3: Tensor):
    """Matsuoka-Nakai factor."""
    return (9.0 * I3 - I1 * I2) \
        / (I3 - I1 * I2)

def I1(sigma: Tensor):
    """First invariant of stress tensor."""
    return einsum("ii", sigma)
```

```
def I2(sigma: Tensor):
    """Second invariant of stress tensor.


    a = einsum("ii,jj", sigma, sigma)
    b = einsum("ij,ij", sigma, sigma)
    result = (a - b) / 2.0
    return result

def I3(sigma: Tensor):
    """Third invariant of stress tensor."""
    return det(sigma)
```

The expressions are written as individual functions, making them easier to test and reuse. To represent the Matsuoka–Nakai factor F_m directly as a function of the stress tensor $\bar{\sigma}_{ij}$, these functions are combined in the following way:

```
def Fm_from_sigma(sigma: Tensor):
    """Fm as a function of stress tensor.


    _I1 = I1(sigma)
    _I2 = I2(sigma)
    _I3 = I3(sigma)
    return Fm(_I1, _I2, _I3)
```

With the composite function $F_m(\sigma)$ available, it becomes straightforward to define a function that returns its derivatives $\frac{\partial F_m}{\partial \sigma_{ij}}$. The implementation using the `jacobian()` function is simple as, follows:

```
from torch.autograd.functional \
  import jacobian
def dFm_dsigma_ad(sigma: Tensor):
    """Derivative of M-N factor
    w.r.t. stress tensor."""
    return jacobian(Fm_from_sigma, sigma)
```

In the context of the stress update algorithm that uses Newton's method, this idea is extended to a vector-valued residual function with a vector input. That is, a function that receives all primary unknowns and outputs the full set of residuals. Note that the possibility of formulating simple tensor-valued functions of tensor arguments, composing them into more complex expressions, and directly differentiating the resulting function is the key advantage of using the state-of-the-art AD framework such as `pytorch.autograd`. Naturally, the application of the suggested workflow is not limited to the Hardening soil model, it can also be applied to a variety of even more complex models including those with anisotropic stiffness and direction-dependent yielding.

2.2.1. Closed-form derivatives

For illustration, the above derivatives obtained via AD are compared to the closed-form derivatives calculated using the chain rule. This is the code that does not have to be derived and written when the implementation relies on AD only. The partial derivatives of F_m are:

```
def dF_m_dI1(I1: Tensor,
             I2: Tensor,
             I3: Tensor):
    return -I2 / (-I1 * I2 + I3) \
           + I2 * (-I1 * I2 + 9 * I3) \
           / (-I1 * I2 + I3) ** 2

def dF_m_dI2(I1: Tensor,
             I2: Tensor,
             I3: Tensor):
    return -I1 / (-I1 * I2 + I3) \
           + I1 * (-I1 * I2 + 9 * I3) \
           / (-I1 * I2 + I3) ** 2

def dF_m_dI3(I1: Tensor,
             I2: Tensor,
             I3: Tensor):
    return 9 / (-I1 * I2 + I3) \
           - (-I1 * I2 + 9 * I3) \
           / (-I1 * I2 + I3) ** 2
```

and the partial derivatives of stress invariants are:

```
from torch import eye
delta = eye(3) # Kronecker delta
def dI1_dsig():
    return delta

def dI2_dsig(sigma: Tensor):
    _I1 = I1(sigma)
    return _I1 * delta - sigma

def dI3_dsig(sigma: Tensor):
    term1 = einsum("ik,kj->ij", sigma,
                                                    sigma
                                                    )
    term2 = -I1(sigma) * sigma
    term3 = I2(sigma) * delta
    return term1 + term2 + term3
```

The chain rule yields the partial derivatives in the composite form as follows:

```
def dFm_dsigma_cf(sigma):
    _I1 = I1(sigma)
    _I2 = I2(sigma)
    _I3 = I3(sigma)
    return dF_m_dI1(_I1, _I2, _I3) \
        * dI1_dsig() \
        + dF_m_dI2(_I1, _I2, _I3) \
        * dI2_dsig(sigma) \
        + dF_m_dI3(_I1, _I2, _I3) \
        * dI3_dsig(sigma)
```

Finally, the derivatives calculated by the means of AD are compared to those obtained manually via the chain rule:

```
from torch import rand, transpose, allclose
randmat = rand((3,3)) # random 3x3 matrix
sigma = -100.0 * delta \
      - 10.0 * (randmat \
      + transpose(randmat, 0, 1))
dFm_ad = dFm_dsigma_ad(sigma)
dFm_cf = dFm_dsigma_cf(sigma)
assert(allclose(dFm_ad, dFm_cf))
```

3. Results

A simulation of a drained triaxial shear test was used to compare the AD-based implementation to the implementation with closed-form derivatives. The drained

Parameter	Value	Unit
c	10	kPa
φ	30	°
E_i^{ref}	18 182	kPa
E_{ur}^{ref}	30 000	kPa
p_{ref}	−100	kPa
m_p	0.5	–
ν_{ur}	0.2	–
ψ	4	°
M	1.04	–
R_f	0.9	–
H	25 836	kPa

TABLE 1. The parameters of the Hardening soil model used in the simulation.

triaxial shear test begins in a state of isotropic compression with a mean stress of −100 kPa. For this initial isotropic stress state, the preconsolidation pressure of $p_c = -100$ kPa was prescribed, and the strain-like hardening parameter was set to $\alpha_s = 0$. For the sake of simplicity, the strain is set to zero in this reference initial state. Then the axial stress changes to −320 kPa within 100 loading steps. The soil sample responds with axial compression and radial dilatation.

Two implicitly defined functions are nested in this type of simulation, each using Newton's method for their evaluation. At the upper level the strain increment is being searched that minimises the difference between the prescribed and actually obtained stresses for a given load step. At a lower level, the stress return mapping is performed in which new values of stress and hardening parameters are computed for a given strain increment. The parameters of the Hardening soil model used in the analysis are listed in Table 1. The simulated evolution of the stress and strain is shown in Figure 3 where ε_e is the axial strain.

3.1. Reduced source code

The expressions that compose the residual functions were implemented in a dedicated source file and separated from the general algorithm of the stress update procedure. The hand-derived partial derivatives of these expressions were also implemented in a separate file. This organisation allowed us to compare the amount of source code that the implementation of the residual function versus the derivatives required.

In particular, when using AD, only the file containing the expression for all nodes in the graph in Figure 1, having 447 lines of code, was needed. On the other hand, the implementation using manually written derivatives required also the second file containing 635 additional lines of code. This suggests that 58 % of the total amount of the source code linked directly to the material model can be omitted when AD is applied.

3.2. Performance

The performance of the AD-based implementation was compared with two alternatives: manually coded closed-form derivatives and numerical differentiation. First, the assessment was carried out on the single scalar expression $F_m(\sigma_{ij})$ with tensor argument, then on the full system of residuals defined in Section 2.1.4, and finally on the entire single-element test.

For the scalar function of the tensor argument $F_m(\sigma_{ij})$, the AD-based implementation of $\frac{\partial F_m}{\partial \sigma_{ij}}$ was 35.8 % faster than the closed-form implementation. By contrast, numerical differentiation – requiring an additional function evaluation for each component of the tensor σ_{ij} – was about twice as slow as the closed-form implementation.

When applied to the entire system of residuals, however, the AD-based implementation was 24.6 % slower than the closed-form version. The numerical differentiation approach was, again, the least efficient, taking over 18 times longer than the closed-form implementation. For comparison purposes, the closed-form derivatives were also implemented in NumPy. Their evaluation was approximately eight times faster in NumPy than in PyTorch, indicating that the PyTorch framework introduces a significant overhead, as discussed in the following section.

Finally, the performance of full single-element simulations of a triaxial shear test was examined. In this case, both the residual functions and their derivatives with respect to the primary unknowns are evaluated repeatedly. Again, all three versions of Jacobian implementations in PyTorch were tested: AD approach, closed-form derivatives, and numerical differentiation. The AD-based implementation was about 13.2 % slower than the closed-form version, while numerical differentiation remained the slowest, running more than ten times longer.

The underlying reasons for these performance differences are discussed in the following section.

4. Discussion

The performance comparison between the implementation of a simple expression $\frac{\partial F_m}{\partial \sigma_{ij}}$ based on AD and the implementation with manually coded derivatives yielded the relatively unexpected finding that the AD implementation was approximately 36 % faster than the manually derived and implemented derivatives. On the other hand, the performance comparison between the full system of residuals and the entire single-element simulation showed that the AD-based implementation was slightly faster with manually coded closed-form derivatives. This discrepancy might be explained by two competing properties of AD. On the one hand, the computation graph created by the AD framework is optimised internally and the intermediate results within the chain rule are reused when possible, while the manually coded derivatives might not be optimal in this regard. On the other hand,

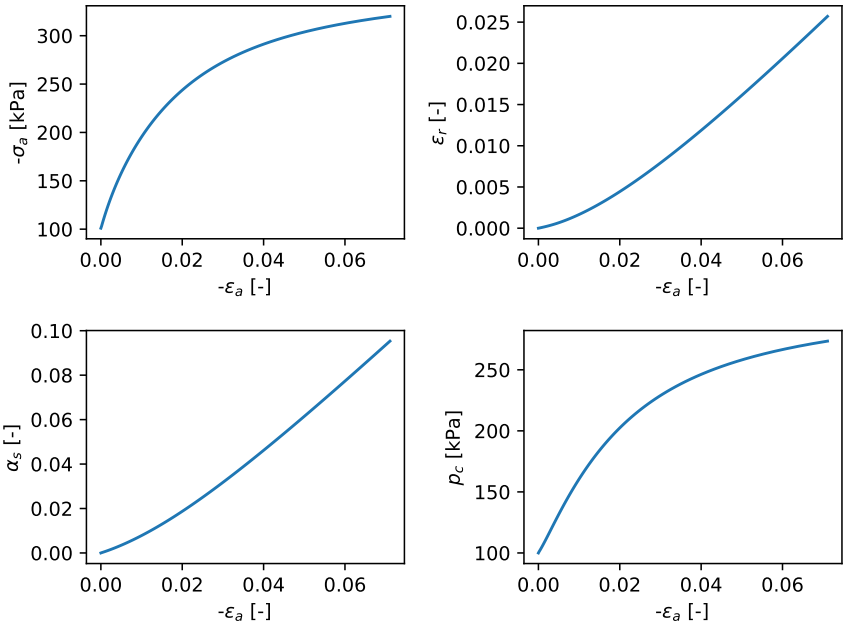

Figure 3. Results of the simulation of drained triaxial shear test. Evolution of axial stress σ_a, radial strain ε_r, cumulative equivalent deviatoric plastic strain α_s, and preconsolidation pressure p_c against axial strain ε_a.

the AD framework introduces some overhead related to the backpropagating analysis of the computation graph. Therefore the comparison does not lead to a clear conclusion which approach is faster in general.

The comparison between the closed-form derivatives showed a clear advantage of the plain NumPy over analogous PyTorch implementation. This finding relates to a significantly smaller overhead of NumPy implementation which completely avoids the construction and analysis of the computation graph. Moreover, NumPy operations are optimised for multidimensional arrays of any size, while PyTorch – being primarily a machine learning framework – is optimised mainly for larger arrays. From this perspective, the main advantage of an AD-based implementation of a material model is that it reduces the amount of work related to code maintenance during the prototyping and testing phase, when frequent changes to the model formulation are expected. Such a robust implementation might also serve to verify the code optimised for production.

The performance of the numerical differentiation is discussed below. Recall that the performance was assessed for three types of expressions:

(1.) A simple scalar function $F_m(\bar{\boldsymbol{\sigma}})$ of tensor argument,

(2.) the full system of residuals given by Equations (54) to (58),

(3.) the entire single-element simulation.

The central finite difference scheme was used to approximate the partial derivatives in all cases. The performance of the numerical differentiation relative to either closed-form differentiation or AD depends primarily on two effects. One is the number of additional function evaluations required to approximate the derivatives of a multivariate expression using the central difference formula. The other is the relative complexity of the original expression to its analytically derived Jacobian.

In the case of the simple scalar function $F_m(\bar{\boldsymbol{\sigma}})$, the number of additional function evaluations is relatively small, while the complexity of the differentiated expression is comparable to that of the original expression. Therefore, the performance of numerical differentiation is still comparable to that of a closed-form differentiation or AD. In contrast, when differentiating the full system of residuals or running the entire simulation, the number of additional function evaluations increases due to the higher number of primary variables, while the complexity of the differentiated expression becomes relatively smaller compared to the original expression. This gives an advantage to the closed-form and AD approaches, leading to the observed performance decrease of numerical differentiation in these cases.

4.1. Usage in production software

In cases where the performance is not critical, the PyTorch-based AD implementation can be used directly within FEM software that already allows for user-defined material models written in Python. One notable example of such versatile finite element code is OOFEM [32, 33].

4.2. AD in other environments

The advantages of using automatic differentiation (AD) are illustrated here in a Python environment

with the PyTorch library. This combination was chosen because Python is a widely adopted language for rapid prototyping in academia and industry, and PyTorch is a well-established framework that supports AD and provides functions that operate on multidimensional arrays in a manner similar to NumPy. However, when performance and scalability are critical considerations for implementing material models, AD can also be used in alternative infrastructures. Notably, the Julia programming language, combined with packages, such as TensorOperations.jl [34] and Zygote [35], provides a user-friendly environment for expressing deeply nested tensor operations and computing their gradients via AD. In the C++ ecosystem, the xtensor library [36] enables high-level operations on n-dimensional arrays, which can be seamlessly combined with the autodiff [37] package to perform automatic differentiation on functions that accept and return multidimensional arrays.

5. Conclusion

The implicit stress return mapping procedure and the consistent material stiffness operator for the Hardening soil model were defined in this paper. The implementation relies on the PyTorch package, in particular on its homogeneous multidimensional array data type `torch.Tensor` and linear algebra operations provided by the library. The automatic differentiation provided by `torch.autograd` package was used to express the partial derivatives of the residual functions making it possible to completely avoid the tedious work of deriving and implementing the partial derivatives of tensor expressions manually. A drained triaxial shear test was simulated in order to compare the performance of implementations based on automatic differentiation and manually coded derivatives. The following conclusions were drawn from the study:

- The higher-level functions provided by PyTorch make the implementation of the residual functions exceptionally straightforward. The usage of `torch.Tensor` data type is almost identical to `numpy.ndarray`, making the potential transition from NumPy effortless. When Einstein index notation is used in the model formulation, the `torch.einsum()` function allows for nearly one-to-one correspondence between the model formulation and its implementation.
- Obtaining the derivatives of the residual functions "free of charge" greatly simplifies not only the implementation, but also the process of testing and evaluating the different versions of the model because only the residual functions need to be specified. Even if the final production implementation is optimised and does not use automatic differentiation, the availability of the derivatives makes verifying of the final implementation much easier.
- The amount of the source code for the partial derivatives exceeds the source code of the residual functions. More than half of the source code is therefore reduced when the automatic differentiation is used.
- Attention has to be paid so that all intermediate and auxiliary expressions are of `torch.Tensor` type so that the AD algorithm can track the gradient across the entire graph of the composite expressions. For example, a problem arises when values are extracted from `torch.Tensor` to `numpy.ndarray` and used to create new `torch.Tensor` within subsequent expressions. This breaks the chain of the gradient computations despite the fact that the actual expression evaluates correctly.
- Although the performance of the AD-based implementaiton surpases the implementation with directly coded closed-form derivatives for simple expressions, the latter approach is generally slightly faster for real simulations of laboratory tests. This encourages using the AD-based implementation primarily as a prototyping and testing strategy when the consistency and coding effort are more critical than the performance.
- When the performance of the material model is in question, such as when the material model is being implemented into production software, there is room for improvement. In particular, the stress-strain relationship of the Hardening soil model is isotropic and thus can be formulated in terms of mean stress and equivalent deviatoric stress. In such a formulation the tensor-valued residual function $R_{1,ij}$ is replaced with two scalar-valued residual functions.

In terms of future prospects, the generality of the stress return mapping framework and PyTorch's flexible infrastructure makes it possible to extend this AD-based implementation approach to anisotropic elastoplastic material models. Additionally, since `torch.autograd` package also expresses higher-order derivatives, this methodology could be applied to implementing standard generalised material models formulated purely in terms of scalar potential functions.

This positions our work as a lightweight and general research tool, complementary to established AD workflows, providing a clear basis for migration toward high-performance environments when needed.

Beyond prototyping, however, the presented approach can already be used within research-grade FEM frameworks that natively expose Python APIs. For instance, in FEniCS/FEniCSx and Firedrake, residuals and Jacobians are formulated in Python, which allows seamless use of autograd-computed tangent operators within the variational problem. Similarly, in OOFEM, which provides an open-source platform for user-defined material models, a coupling with PyTorch can be established through Python bindings to incorporate AD-based tangents in full FE simulations. These examples demonstrate that the proposed approach is not restricted only to conceptual prototyping, but can already support small- to medium-scale FEM

analyses. For large-scale industrial applications, the validated constitutive kernels can then be migrated to high-performance environments such as C++ AD libraries (Sacado, Adept, CppAD, CoDiPack) or industrial DSL/code-generation tools (MFront, AceGen). In this way, our study provides a transparent research framework that bridges prototyping in Python with scalable production implementations.

Acknowledgements

The financial support provided by the Czech Grant Agency, project No. 22-12178S is gratefully acknowledged.

References

[1] J. C. Simo, R. L. Taylor. Consistent tangent operators for rate-independent elastoplasticity. *Computer Methods in Applied Mechanics and Engineering* **48**(1):101–118, 1985. https://doi.org/10.1016/0045-7825(85)90070-2

[2] A. Pérez-Foguet, A. Rodríguez-Ferran, A. Huerta. Numerical differentiation for local and global tangent operators in computational plasticity. *Computer Methods in Applied Mechanics and Engineering* **189**(1):277–296, 2000. https://doi.org/10.1016/S0045-7825(99)00296-0

[3] C. Miehe. Numerical computation of algorithmic (consistent) tangent moduli in large-strain computational inelasticity. *Computer Methods in Applied Mechanics and Engineering* **134**(3–4):223–240, 1996. https://doi.org/10.1016/0045-7825(96)01019-5

[4] A. Lejeune, H. Boudaoud, N. Mathieu, M. Potier-Ferry. Automatic solver for computational plasticity based on Taylor series and automatic differentiation. In *XII International Conference on Computational Plasticity. Fundamentals and Applications (COMPLAS XII)*, pp. 850–867. Barcelone, Spain, 2013.

[5] Q. Chen, J. T. Ostien, G. Hansen. Automatic differentiation for numerically exact computation of tangent operators in small- and large-deformation computational inelasticity. In *TMS 2014: 143rd Annual Meeting & Exhibition*, pp. 289–296. Springer International Publishing, Cham, 2016. https://doi.org/10.1007/978-3-319-48237-8_38

[6] S. Rothe, S. Hartmann. Automatic differentiation for stress and consistent tangent computation. *Archive of Applied Mechanics* **85**(8):1103–1125, 2015. https://doi.org/10.1007/s00419-014-0939-6

[7] A. Vigliotti, F. Auricchio. Automatic differentiation for solid mechanics. *Archives of Computational Methods in Engineering* **28**(3):875–895, 2021. https://doi.org/10.1007/s11831-019-09396-y

[8] F. Zwicke, P. Knechtges, M. Behr, S. Elgeti. Automatic implementation of material laws: Jacobian calculation in a finite element code with TAPENADE. *Computers & Mathematics with Applications* **72**(11):2808–2822, 2016. https://doi.org/10.1016/j.camwa.2016.10.010

[9] A. Latyshev, J. Bleyer, C. Maurini, J. S. Hale. Expressing general constitutive models in FEniCSx using external operators and algorithmic automatic differentiation, 2024.

[10] J. Korelc, P. Wriggers. *Automation of finite element methods*. Springer, Cham, 2016. https://doi.org/10.1007/978-3-319-39005-5

[11] M. Hillgärtner, T. Guo, M. Itskov. Automatic differentiation of strain-energy functions in the context of user-defined materials for the FEM. *PAMM* **20**(1):e202000050, 2021. https://doi.org/10.1002/pamm.202000050

[12] D. T. Seidl, B. N. Granzow. 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, 2022. https://doi.org/10.1002/nme.6843

[13] E. Aulisa, G. Capodaglio. Monolithic coupling of the implicit material point method with the finite element method. *Computers & Structures* **219**:1–15, 2019. https://doi.org/10.1016/j.compstruc.2019.04.006

[14] I. R. Shishir, A. Tabarraei. Multi-materials topology optimization using deep neural network for coupled thermo-mechanical problems. *Computers & Structures* **291**:107218, 2024. https://doi.org/10.1016/j.compstruc.2023.107218

[15] J. Blühdorn, N. R. Gauger, M. Kabel. AutoMat: Automatic differentiation for generalized standard materials on GPUs. *Computational Mechanics* **69**(2):589–613, 2022. https://doi.org/10.1007/s00466-021-02105-2

[16] M. Pundir, D. S. Kammer. Simplifying FFT-based methods for solid mechanics with automatic differentiation. *Computer Methods in Applied Mechanics and Engineering* **435**:117572, 2025. https://doi.org/10.1016/j.cma.2024.117572

[17] F. Masi, I. Stefanou, P. Vannucci, V. Maffi-Berthier. Thermodynamics-based artificial neural networks for constitutive modeling. *Journal of the Mechanics and Physics of Solids* **147**:104277, 2021. https://doi.org/10.1016/j.jmps.2020.104277

[18] T. Janda, M. Šejnoha, A. Zemanová, T. Žalská. Automatic differentiation in PyTorch as a tool for robust implementation of elasto-plastic constitutive model. In *Proceedings of the Twelfth International Conference on Engineering Computational Technology*, p. 9.3. 2024. https://doi.org/10.4203/ccc.8.9.3

[19] J. Bleyer, A. Latyshev, C. Maurini, J. S. Hale. Differentiable constitutive modeling with FEniCSx and JAX. In *FEniCS 2025*, pp. 1–18. Groningen, Netherlands, 2025.

[20] A. Favilli, F. Laccone, P. Cignoni, et al. Geometric deep learning for statics-aware grid shells. *Computers & Structures* **292**:107238, 2024. https://doi.org/10.1016/j.compstruc.2023.107238

[21] Y. Liao, R. Lin, R. Zhang, G. Wu. Attention-based LSTM (AttLSTM) neural network for seismic response modeling of bridges. *Computers & Structures* **275**:106915, 2023. https://doi.org/10.1016/j.compstruc.2022.106915

[22] W. Ouyang, L. Chen, A.-R. Liang, S.-W. Liu. Neural networks-based line element method for large deflection frame analysis. *Computers & Structures* **300**:107425, 2024. https://doi.org/10.1016/j.compstruc.2024.107425

[23] T. Schanz, P. A. Vermeer, P. G. Bonnier. The hardening soil model: Formulation and verification. In R. B. J. Brinkgreve (ed.), *Beyond 2000 in Computational Geotechnics*, pp. 281–296. Routledge, 1st edn., 1999. https://doi.org/10.1201/9781315138206-27

[24] H. Matsuoka, T. Nakai. Stress-deformation and strength characteristics of soil under three different principal stresses. *Proceedings of the Japan Society of Civil Engineers* **1974**(232):59–70, 1974. https://doi.org/10.2208/jscej1969.1974.232_59

[25] T. Benz. *Small-strain stiffness of soils and its numerical consequences*. No. 55 in Mitteilung. Inst. für Geotechnik, Stuttgart, 2007.

[26] L. Cocco, M. Ruiz. Numerical implementation of hardening soil model. In A. S. Cardoso, J. L. Borges, P. A. Costa, et al. (eds.), *Numerical Methods in Geotechnical Engineering IX*, pp. 195–203. CRC Press, 1st edn., 2018. https://doi.org/10.1201/9781351003629-25

[27] E. A. de Souza Neto, D. Perić, D. R. J. Owen. *Computational Methods for Plasticity: Theory and Applications*. Wiley, Chichester, West Sussex, UK, 1st edn., 2008. https://doi.org/10.1002/9780470694626

[28] A. Paszke, S. Gross, S. Chintala, et al. Automatic differentiation in PyTorch. In *31st Conference on Neural Information Processing Systems (NIPS 2017)*, pp. 1–4. 2017.

[29] A. Paszke, S. Gross, F. Massa, et al. PyTorch: An imperative style, high-performance deep learning library. *arXiv* p. 1912.01703, 2019. https://doi.org/10.48550/arXiv.1912.01703

[30] C. R. Harris, K. J. Millman, S. J. van der Walt, et al. Array programming with NumPy. *Nature* **585**(7825):357–362, 2020. https://doi.org/10.1038/s41586-020-2649-2

[31] A. Griewank, A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, chap. 1. Introduction, pp. 1–12. Society for Industrial and Applied Mathematics, 2nd edn., 2008. https://doi.org/10.1137/1.9780898717761.ch1

[32] B. Patzák, Z. Bittnar. Design of object oriented finite element code. *Advances in Engineering Software* **32**(10–11):759–767, 2001. https://doi.org/10.1016/S0965-9978(01)00027-8

[33] B. Patzák, D. Rypl. Object-oriented, parallel finite element framework with dynamic load balancing. *Advances in Engineering Software* **47**(1):35–50, 2012. https://doi.org/10.1016/j.advengsoft.2011.12.008

[34] L. Devos, M. Van Damme, et al. Tensoroperations.jl, 2023. https://doi.org/10.5281/zenodo.3245496

[35] M. Innes, A. Edelman, K. Fischer, et al. A differentiable programming system to bridge machine learning and scientific computing. *arXiv* p. 1907.07587, 2019. https://doi.org/10.48550/arXiv.1907.07587

[36] Xtensor-stack. Xtensor: Multi-dimensional arrays with broadcasting and lazy computing, 2025. [2025-09-17]. https://github.com/xtensor-stack/xtensor

[37] A. Leal. Autodiff, a modern, fast and expressive C++ library for automatic differentiation, 2018. [2025-09-17]. https://autodiff.github.io

Appendix A.

The increment of the strain-like hardening parameter $\Delta\alpha_s$ is chosen to be proportional to the increment of the plastic shear strain $\Delta\gamma_s^{ps}$ following the form:

$$\Delta\alpha_s = \frac{2}{3}\Delta\gamma_s^{ps}, \tag{69}$$

where $\Delta\gamma_s^{ps}$ reads:

$$\Delta\gamma_s^{ps} = \sqrt{\frac{3}{2}\left(\Delta\varepsilon_{ij}^p - \frac{1}{3}\delta_{ij}\Delta\varepsilon_{kk}^p\right)\left(\Delta\varepsilon_{ij}^p - \frac{1}{3}\delta_{ij}\Delta\varepsilon_{kk}^p\right)}. \tag{70}$$

By substituting Equations (8) and (19), and realising that the two terms in Equation (19) correspond to the isotropic and deviatoric parts, respectively, we arrive to:

$$\begin{aligned}
\Delta\alpha_s &= \frac{2}{3}\Delta\gamma_s^{ps} \\
&= \frac{2}{3}\Delta\lambda_s\sqrt{\frac{3}{2}\left(n_{ij} - \frac{1}{3}\delta_{ij}n_{kk}\right)\left(n_{ij} - \frac{1}{3}\delta_{ij}n_{kk}\right)} && (71)\\
&= \frac{2}{3}\Delta\lambda_s\sqrt{\frac{3}{2}\frac{3}{2}\frac{s_{ij}}{q}\frac{3}{2}\frac{s_{ij}}{q}} && (72)\\
&= \Delta\lambda_s && (73)
\end{aligned}$$

and therefore the generalised hardening modulus introduced in Equation (33) follows $H_s = 1$.