Proceedings of the Twelfth International Conference on
Engineering Computational Technology
Edited by: P. Iványi, J. Kruis and B.H.V. Topping
Civil-Comp Conferences, Volume 8, Paper 9.3
Civil-Comp Press, Edinburgh, United Kingdom, 2024
ISSN: 2753-3239, doi: 10.4203/ccc.8.9.3
©Civil-Comp Ltd, Edinburgh, UK, 2024

Automatic Differentiation in PyTorch as a Tool for Robust Implementation of Elasto-Plastic Constitutive Model

T. Janda[1], M. Šejnoha[1], A. Zemanová[2] and T. Žalská[2]

[1] Department of Mechanics, Faculty of Civil Engineering, Czech Technical University in Prague, Prague, Czechia
[2] Department of Geotechnics, Faculty of Civil Engineering, Czech Technical University in Prague, Prague, Czechia

Abstract

The paper presents a simple and robust approach to an implementation of the hardening soil model into finite element calculations. The implementation of the return stress mapping exploits the automatic differentiation of tensor variables provided by the PyTorch framework. The automatic differentiation allows for a succinct implementation despite the relatively complex structure of the nonlinear equations in the stress return algorithm. The presented approach is not limited to the hardening soil model. It can be utilised in the development and verification of other elasto-plastic constitutive models where expressing and maintaining the Jacobian matrix over different versions of a material model is time-consuming and error-prone.

Keywords: elasto-plasticity, hardening soil model, stress return mapping, automatic differentiation, PyTorch, finite element method.

1 Introduction

The development and testing of any non-trivial elasto-plastic constitutive model is often an iterative process. The results from the model's verification and validation typically involve changes in both the formulation and implementation. Reducing the amount of computer code that needs to be written and automating the tasks that can be automated makes the whole process more focused and efficient. This paper shows how the *automatic differentiation* (AD) implemented as part of the PyTorch framework can be utilised to automate the calculation of the Jacobian matrix for the residual functions within the standard implicit return stress mapping. The structure of the hardening soil model is presented to show the possible complexity of residual functions. Then the principles of the implementation are illustrated in a simple composite expression. Finally, a comparison to closed-form derivatives and numerical derivatives in terms of performance is provided.

2 Methods

This section briefly introduces the structure of the hardening soil model and the concept of automatic differentiation.

2.1 Hardening soil model

The hardening soil model is a constitutive model for soils developed in [6] and widely used in geotechnical finite element software. The model features nonlinear stress-dependent stiffness and two yield surfaces. The first cone-shaped yield surface resembles the Matsuoka-Nakai yield criterion. Its shape is controlled by the hardening parameter α_s, which depends on equivalent deviatoric plastic strain. This yield surface is referred to as the *shear* yield surface. The second yield surface has an elliptical shape and resembles the yield criterion of the modified Cam clay model. The size of this closed yield surface is controlled by preconsolidation pressure p_c. This yield surface is referred to as the *cap* surface. Both hardening parameters represent the *state* of the soil.

The model formulation brings together a number of experimentally verified theories and modelling approaches such as the modified hyperbolic stress-strain law for triaxial compression [1], the power law for elastic modulus [2] or Rowe's dilatancy theory [5] to name a few. The combination of the relatively independent features built into a single elasto-plastic model makes its structure rather convoluted when compared to other common elasto-plasticity models for soils such as the modified Cam clay model. The complexity of its internal structure makes the hardening soil model an illustrative example for the implementation choices suggested in this paper.

2.2 Stress return algorithm

The stress update procedure defines how the stress and state of a material at the n-th pseudo-time instant changes when it is loaded by a small increment of strain $\Delta\varepsilon_{kl}$. The stress $\sigma_{n,ij}$ and the hardening parameters $\alpha_{s,n}$ and $p_{c,n}$ at the beginning of a stress update are known. The stress update procedure then searches for unknown stress σ_{ij} and the hardening parameters α_s and p_c and plastic multipliers λ_s and λ_c, i.e. the *primary unknowns* at the $(n+1)$-th time instant $n+1$, so that the residua

$$R_{1,ij}(\boldsymbol{\sigma}, \boldsymbol{D}, \Delta\boldsymbol{\varepsilon}, \Delta\boldsymbol{\varepsilon}^{ps}, \Delta\boldsymbol{\varepsilon}^{pc}) = \sigma_{ij} - \sigma_{n,ij} - D_{ijkl}(\Delta\varepsilon_{kl} - \Delta\varepsilon^{ps}_{kl} - \Delta\varepsilon^{pc}_{kl}) \quad (1)$$

$$R_2(\alpha_s, \Delta\lambda_s) = \alpha_s - \alpha_{s,n} - \Delta\lambda_s H_s \quad (2)$$

$$R_3(f_s) = f_s \quad (3)$$

$$R_4(p_c, \Delta p_c) = p_c - p_{c,n} - \Delta p_c \quad (4)$$

$$R_5(f_c) = f_c \quad (5)$$

are equal to zero. The residuum $R_{1,ij}$ enforces Hooke's law in incremental form. It features the elastic tensor D_{ijkl} and the plastic strain increments $\Delta\varepsilon^{ps}_{kl}$ and $\Delta\varepsilon^{pc}_{kl}$ obtained separately from the flow rules attributed to the shear and the cap yield surfaces, respectively. The second residuum corresponds to the hardening law for the *shear* yield surface, where $\Delta\lambda_s$ is the plastic multiplier and H_s is the plastic modulus. It can be shown that increment of the hardening parameter $\Delta\alpha = \Delta\lambda_s$ and therefore the hardening modulus $H_s = 1$. The third residuum is the shear yield function f_s. The fourth residuum corresponds to the hardening law for the *cap* yield surface, where Δp_c is the increment of the hardening parameter p_c. The fifth residuum is the yield function defining the cap yield surface. The paper now proceeds by defining all the symbols that appear in the above expressions.

The stress-dependent elastic fourth-order tensor is defined as

$$D_{ijkl}(E_{ur}) = E_{ur}\left(\frac{\nu_{ur}}{(1+\nu_{ur})(1-2\nu_{ur})}\delta_{ij}\delta_{kl} + \frac{1}{2(1+\nu_{ur})}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})\right) \quad (6)$$

where the current Young's modulus for unloading-reloading is given as the reference Young's modulus multiplied by a stiffness factor

$$E_{ur}(f_E) = E^{ref}_{ur} f_E \quad (7)$$

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

$$f_E(\bar{p}) = \left(\frac{\bar{p}}{\bar{p}^{ref}}\right)^{m_p} \quad (8)$$

The offset mean stress is therefore calculated from the offset stress tensor as

$$\bar{p}(\bar{\boldsymbol{\sigma}}) = \frac{\bar{\sigma}_{ii}}{3} \quad (9)$$

where the offset stress tensor is calculated as

$$\bar{\sigma}_{ij}(\boldsymbol{\sigma}) = \sigma_{ij} - c\cot(\varphi)\delta_{ij} \tag{10}$$

and ensures that $\bar{\sigma}_{ij} = 0$ at the apex of the shear yield surface.

The flow rule attributed to the shear yield surface reads

$$\Delta\varepsilon^{ps}_{ij}(\Delta\lambda_s, \boldsymbol{n}_s) = \Delta\lambda_s n_{s,ij} \tag{11}$$

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

$$n_{s,ij}(\boldsymbol{s}, q, \sin\psi_m) = -\frac{2\sin\psi_m}{3-\sin\psi_m}\delta_{ij} + \frac{3}{2}\frac{s_{ij}}{q} \tag{12}$$

where the equivalent deviatoric stress is

$$q(\boldsymbol{s}) = \sqrt{\frac{3}{2}s_{ij}s_{ij}} \tag{13}$$

and the deviatoric part of the stress tensor is

$$s_{ij}(\bar{\boldsymbol{\sigma}}, \bar{p}) = \bar{\sigma}_{ij} - \bar{p}\delta_{ij} \tag{14}$$

The mobilised angle of internal friction

$$\sin\varphi_m(F_m) = \sqrt{F_m} \tag{15}$$

depends on the auxiliary variable F_m denoted as the Matsuoka-Nakai factor, which in turn depends on all three stress invariants I_1, I_2, I_3 as

$$F_m(I_1, I_2, I_3) = \frac{9I_3 - I_1 I_2}{I_3 - I_1 I_2} \tag{16}$$

with the stress invariants provided by

$$I_1(\bar{\boldsymbol{\sigma}}) = \bar{\sigma}_{ii} \tag{17}$$

$$I_2(\bar{\boldsymbol{\sigma}}) = \frac{1}{2}(\bar{\sigma}_{ii}\bar{\sigma}_{jj} - \bar{\sigma}_{ij}\bar{\sigma}_{ij}) \tag{18}$$

$$I_3(\bar{\boldsymbol{\sigma}}) = \frac{1}{6}(\bar{\sigma}_{ii}\bar{\sigma}_{jj}\bar{\sigma}_{kk} + 2\bar{\sigma}_{ij}\bar{\sigma}_{jk}\bar{\sigma}_{ki} - 3\bar{\sigma}_{ij}\bar{\sigma}_{ji}\bar{\sigma}_{kk}) = \det(\bar{\boldsymbol{\sigma}}) \tag{19}$$

The sine of the mobilised dilatation angle is defined as

$$\sin\psi_m(\sin\varphi_m) = \frac{\sin\varphi_m - \sin\varphi_{cs}}{1 - \sin\varphi_m \sin\varphi_{cs}} \tag{20}$$

The sine of the friction angle at critical state depends only on the material parameters and is given by

$$\sin\varphi_{cs} = \frac{\sin\varphi - \sin\psi}{1 - \sin\varphi\sin\psi} \tag{21}$$

so it does not evolve during the stress update. The shear yield function is written as

$$f_s(q, \sin\varphi_m, A) = q - \left(1 - R_f \frac{1-\sin\varphi}{1-\sin\varphi_m}\frac{\sin\varphi_m}{\sin\varphi}\right)\left(\frac{E_i^{ref}}{E_{ur}^{ref}}q + A\right) \tag{22}$$

where A is the stress-like hardening variable defined as

$$A(E_i, \alpha_s) = E_i \alpha_s \tag{23}$$

and the modulus E_i is calculated analogously to the unloading-reloading modulus E_i from a reference value and the stiffness factor according to equation

$$E_i(f_E) = E_i^{ref} f_E \tag{24}$$

The smoothed version of the cap yield function has an elliptical shape in the$p \times q$ plane and is written as

$$f_c(p, q, \chi, p_c) = \frac{q^2}{(M\chi)^2} + p^2 - p_c^2 \tag{25}$$

where p_c is the preconsolidation pressure acting as a hardening variable, M is a model parameter and the shape factor χ is a function of the Lode angle determining the shape of the yield surface in the deviatoric plane. The factor χ takes the form

$$\chi(\vartheta) = \frac{\sqrt{3}\beta}{2\sqrt{\beta^2 - \beta + 1}}\frac{1}{\cos\vartheta} \tag{26}$$

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

$$\beta = \frac{3 - \sin\varphi}{3 + \sin\varphi} \tag{27}$$

and therefore is constant during the stress update. The auxiliary parameter ϑ is defined as

$$\vartheta(\bar{\vartheta}) = \frac{1}{6}\arccos\left(-1 + c_\beta \sin^2 3\theta\right) \qquad \text{for } \theta \leq 0 \tag{28}$$

$$\vartheta(\bar{\vartheta}) = \frac{\pi}{3} - \frac{1}{6}\arccos\left(-1 + c_\beta \sin^2 3\theta\right) \qquad \text{for } \theta > 0 \tag{29}$$

where the constant c_β reads

$$c_\beta = \frac{27\beta^2(1-\beta)^2}{2(\beta^2 - \beta + 1)^3} \tag{30}$$

The Lode angle θ is defined as

$$\sin 3\theta(J_2, J_3) = -\frac{3\sqrt{3}J_3}{2J_2^{\frac{3}{2}}} \tag{31}$$

and the invariants of the deviatoric part of the stress tensor are

$$J_2(\boldsymbol{s}) = \frac{1}{2} s_{ij} s_{ij} \tag{32}$$

$$J_3(\boldsymbol{s}) = \frac{1}{3} s_{ij} s_{jk} s_{ki} \tag{33}$$

The increment of the plastic strain associated with the cap surface is traditionally given by the flow rule in the form

$$\Delta\varepsilon_{kl}^{pc}(\Delta\lambda_c, \boldsymbol{n}_c) = \Delta\lambda_c n_{c,ij} \tag{34}$$

where the normals to the cap plastic potential are written as

$$n_{c,ij}(p, \boldsymbol{s}, \chi) = \frac{2}{3} p \delta_{ij} + \frac{3 s_{ij}}{(M\chi)^2} \tag{35}$$

Note that the flow direction does not depend on p_c. The increment of the preconsolidation pressure follows

$$\Delta p_c(f_E, p, \Delta\lambda_c) = -2 H f_E p \Delta\lambda_c \tag{36}$$

and finally, the (non-offset) mean stress is calculated directly from the stress tensor as

$$p(\boldsymbol{\sigma}) = \frac{\sigma_{ii}}{3} \tag{37}$$

The dependencies between the residuals and primary unknowns are displayed in Figure 1. The figure contains only the variables that change during the stress update algorithm. These are:

- Residuals $R_{1,ij}$, R_2, R_3, R_4, R_5
- Auxiliary intermediate variables D_{ijkl}, $\Delta\varepsilon_{ij}^{ps}$, $\Delta\varepsilon_{ij}^{pc}$, Δp_c, f_c, E_{ur}, f_E, $\bar{p}$, $\bar{\sigma}_{ij}$, $n_{s,ij}$, q, s_{ij}, $\sin\varphi_m$, F_m, I_1, I_2, I_3, $\sin\psi_m$, f_s, A, E_i, f_c, χ, ϑ, $\sin 3\theta$, J_2, J_3, $n_{c,ij}$
- Primary unknowns σ_{ij}, α_s, p_c, λ_s, λ_c

On the other hand, the following values are kept constant within a single step of the stress update:

- Model parameters E_i^{ref}, E_{ur}^{ref}, ν_{ur}, m_p, $\bar{p}^{ref}$, c, φ, ψ, R_f, M, H
- Values directly calculated from the model parameters φ_{cs}, β, c_β
- Values that generally depend on primary unknowns but happen to be constant in this model formulation H_s
- Values at the beginning of the stress update step $\sigma_{n,ij}$, $\alpha_{s,n}$, $p_{c,n}$
- Prescribed strain increment $\Delta\varepsilon_{ij}$

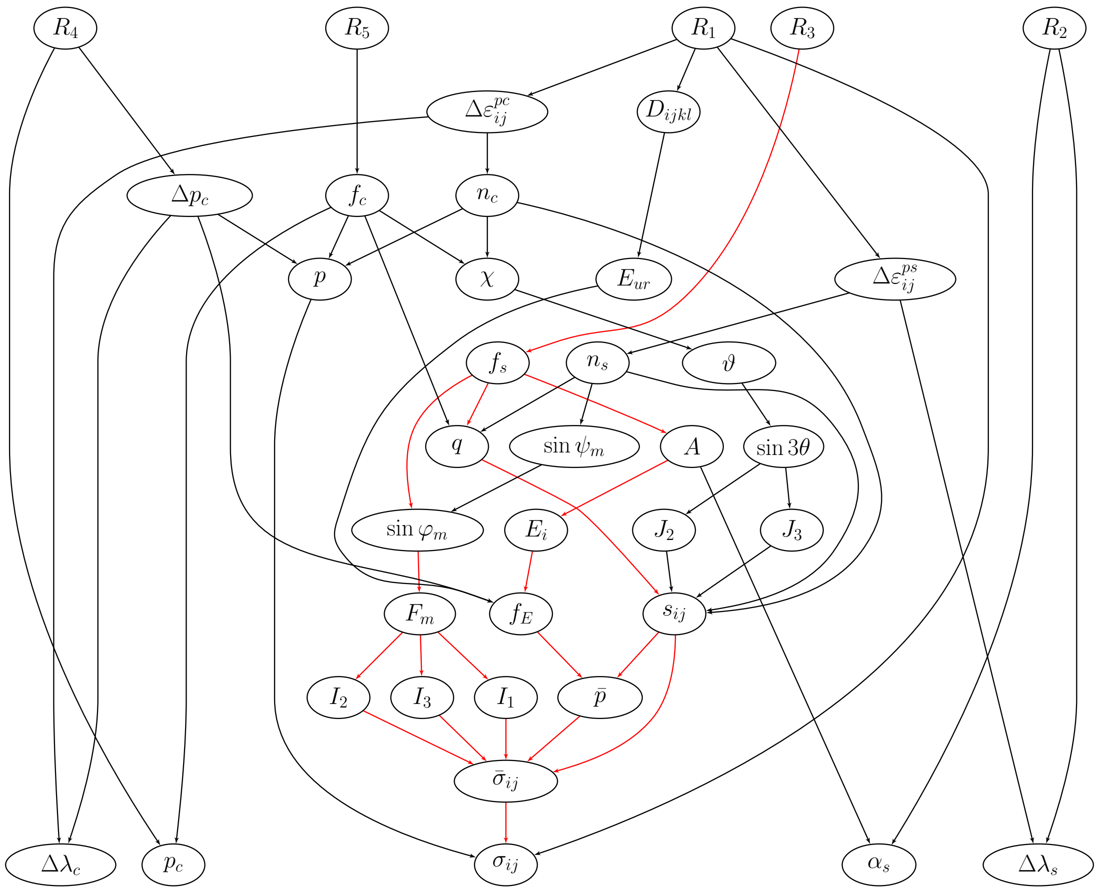

Figure 1: Directed acyclic graph representing the dependencies of residuals R_i on the primary unknowns σ_{ij}, α_s, p_c, λ_s and λ_c. Note that each variable depends on all variables to which its arrows point. Therefore each arrow also corresponds to a (partial) derivative. These are subsequently composed via chain rule. For illustration, the subgraph highlighted in red shows the dependency of R_3 on σ_{ij}. Each red arrow therefore represents one partial derivative in equation (38)

2.3 Newton's method

The standard way to find the values of the primary unknowns, i.e. the roots of the residual functions, utilises Newton's method. Its application typically requires us to express the residuals in the form of a vector-valued function whose argument is a 1D vector containing the primary unknowns. The derivatives of this residual function with respect to the primary unknowns is then known as the Jacobian matrix. In the standard scenario, the Jacobian matrix is calculated by first expressing the partial derivatives of expressions (2) to (37) and subsequently composing them according to chain rules.

Expressing all elements of the Jacobian matrix for the residual functions defined in the previous section is beyond the extent of this paper. In fact, the main goal of this paper is to outline a way how the calculation of the Jacobian matrix can be automated without directly expressing them. Nevertheless, the derivative of the third residuum R_3 with respect to the current stress tensor σ_{ij} is shown here as an example of which partial derivatives need to be calculated and how they are composed. The expression reads

$$\begin{aligned}\frac{\mathrm{d}R_3}{\mathrm{d}\sigma_{ij}} = \frac{\partial R_3}{\partial f_s}\Bigg(&\frac{\partial f_s}{\partial \sin\varphi_m}\frac{\partial \sin\varphi_m}{\partial F_m}\left(\frac{\partial F_m}{\partial I_1}\frac{\partial I_1}{\partial \bar{\sigma}_{kl}} + \frac{\partial F_m}{\partial I_1}\frac{\partial I_1}{\partial \bar{\sigma}_{kl}} + \frac{\partial F_m}{\partial I_1}\frac{\partial I_1}{\partial \bar{\sigma}_{kl}}\right)\\ &+ \frac{\partial f_s}{\partial q}\frac{\partial q}{\partial s_{mn}}\left(\frac{\partial s_{mn}}{\partial \bar{\sigma}_{kl}} + \frac{\partial s_{mn}}{\partial \bar{p}}\frac{\partial \bar{p}}{\partial \bar{\sigma}_{kl}}\right) + \frac{\partial f_s}{\partial A}\frac{\partial A}{\partial E_i}\frac{\partial E_i}{\partial f_E}\frac{\partial f_E}{\partial \bar{p}}\frac{\partial \bar{p}}{\partial \bar{\sigma}_{kl}}\Bigg)\frac{\partial \bar{\sigma}_{kl}}{\partial \sigma_{ij}} \qquad (38)\end{aligned}$$

We observe that a) many of the partial derivatives are second or fourth-order tensor expressions, b) the chain rules quickly become complex when the structure of the dependencies grows and c) visualising the dependencies in the form of a directed graph helps us to decide which derivatives to express and how to compose them according to the chain rule.

2.4 Automatic differentiation

Automatic differentiation (AD) is a technique that provides partial derivatives of a mathematical expression implemented as a function in a computer program. The principle of AD requires that a derivative is defined for every elementary operation that the AD framework offers. In other words, the AD framework knows what the derivative of addition, subtraction, multiplication, application of elementary function, etc., is. Because each mathematical expression is a composition of these elementary operations, the AD framework keeps track of the expression structure and builds an expression tree similar in principle to the one shown in Figure 1. This expression tree can then be traversed and the derivatives automatically composed out of the known derivatives of elementary operations.

Note that AD differs from symbolic differentiation achieved automatically through computer algebra systems such as Maxima or SymPy. It also differs from numerical differentiation based on finite differences.

3 Results

Automatic differentiation of tensor expressions is first illustrated on the composite expressions for F_m viewed as a function of σ_{ij} as defined in (16) to (19). Then the results are compared to closed-form derivatives and numerical derivatives in terms of execution times.

3.1 Implementation in PyTorch

PyTorch [4] is a machine learning framework for Python that is built around a Tensor data type. The tensor is a homogeneous multidimensional array similar to ndarray used in Numpy. Although PyTorch is designed mainly for the development of deep learning models, its palette of convenient operations on tensors together with automatic differentiation [3] makes it a convenient tool for other engineering problems such as fast and straightforward prototyping and verification of elasto-plastic material models. From this point of view its main useful features are the ability to specify the operations on tensors directly in the Einstein summation notation and the automatic calculation of derivatives of any composite scalar or tensor expression.

For illustration, consider the expressions for F_m, I_1, I_2, I_3 defined in (16) to (19). Their implementation is

```
>>> 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 implemented in separate functions which makes it easier to test them and reuse them. To express the Matsuoka–Nakai factor F_m directly as a function of stress σ_{ij} the above functions are composed as follows

```
>>> 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 $F_m(\sigma)$ at hand, it is now easy to create a function returning its derivatives $\frac{\partial F_m}{\partial \sigma_{ij}}$. The implementation using PyTorch's `jacobian` function is as straightforward as

```
>>> from torch.autograd.functional import jacobian
>>> def dFm_dsigma_ad(sigma: Tensor):
```

```
...     """Derivative of Matsuoka-Nakai factor wrt stress tensor."""
...     return jacobian(Fm_from_sigma, sigma)
```

In context of the stress update algorithm based on Newton's method, the above concept is applied on a vector valued residual function of vector argument, i.e., the function that accepts all primary unknowns and returns the values of all residua.

3.2 Closed-form derivatives

For illustration purposes the above derivatives obtained via AD are compared compared with closed-form derivatives calculated through chain rule. 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 composes the partial derivatives 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, we compare the derivatives calculated through AD to derivatives calculated the standard way via 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)
>>> print(allclose(dFm_ad, dFm_cf))
True
```

Regarding the performance, the AD implementation of $\frac{\partial F_m}{\partial \sigma_{ij}}$ showed 35.8% speedup compared to the closed-form implementation. The execution time was also measured for implementation based on numerical differentiation. In particular, the implementation based on the central difference formula, i.e. an implementation that required two function calls for each element of the function's tensorial argument σ_{ij}, was approximately 3 times slower than the closed-form implementation.

4 Discussion

The above example shows that PyTorch operations on tensor variables, e.g., the functions `einsum()` or `det()` together with automatic differentiation make the implementation of the elasto-plastic material model as straightforward as writing the expressions for the residuals. The derivatives needed for both Newton's method within the stress update and calculation of the algorithmic stiffness matrix are calculated automatically by the AD framework.

The performance test on a very simple composite expression for the Matsuoka-Nakai factor F_m slightly favours the AD implementation. Nevertheless, such a test is far from representative and there certainly are cases where hard-coded closed-form gradient outperforms the AD-based implementation. When performance is in question, there are other approaches to make the stress update more efficient such as formulating the stress update in terms of stress invariants instead of full stress tensor, or at least exploiting the symmetry of stress and strain tensors.

5 Conclusions

The paper outlined an approach to implementation of an elasto-plastic constitutive model which does not require hand-calculating the derivatives of the residual functions needed by Newton's method in the stress update procedure and in the formulation of the algorithmic stiffness operator. The implementation builds on automatic differentiation implemented in the PyTorch package for Python.

Acknowledgements

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

References

[1] J. M. Duncan and C. Y. Chang. “Nonlinear Analysis of Stress and Strain in Soils”. In: *Journal of Soil Mechanics & Foundations Div* (Sept. 1970).

[2] N. Janbu. “Soil compressibility as determined by odometer and triaxial tests”. In: *Proc. 3rd ECSMFE* 1 (1963), pp. 19–25.

[3] A. Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS 2017 Autodiff Workshop* (2017).

[4] A. Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. DOI: 10.48550/arXiv.1912.01703. arXiv: 1912.01703[cs,stat].

[5] P. W. Rowe and Geoffrey Ingram Taylor. “The stress-dilatancy relation for static equilibrium of an assembly of particles in contact”. In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 269.1339 (Jan. 1997). Publisher: Royal Society, pp. 500–527. DOI: 10.1098/rspa.1962.0193.

[6] T. Schanz, P.A. Vermeer, and P.G. Bonnier. “The hardening soil model: Formulation and verification”. In: *Beyond 2000 in Computational Geotechnics*. Ed. by Ronald B. J. Brinkgreve. 1st ed. Routledge, 1999, pp. 281–296. ISBN: 978-1-315-13820-6. DOI: 10.1201/9781315138206-27.