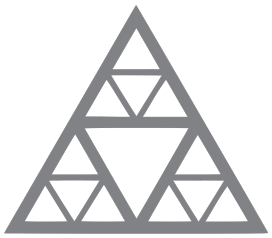


École des Ponts ParisTech

2021-2022

End-of-studies project

Department of Mechanical Engineering and Materials

Andrey Latyshev

Double degree engineering student

Finite-element implementation of plasticity using a convex optimization approach

Project carried out within Laboratoire Navier, ENPC

6 et 8 avenue Blaise Pascal, Champs-sur-Marne, 77455

21/03/2022 - 09/09/2022

Tutor : Jeremy Bleyer

Composition of jury

President : Dr. Karam Sab

Project director : Dr. Jeremy Bleyer

Study advisor : Dr. Matthieu Vandamme

Acknowledgements

I thank my scientific directors, Jeremy Bleyer and Corrado Maurini, for their wise advice, competent guidance during the internship, as well as the freedom of action provided in the research. Special thanks to Jeremy for quickly finding funding for this project, and I was able to get to work quickly.

I would also like to thank Jack Hale for his outside feedback on our work.

In addition, I would like to thank Mathieu Vandamme for his mentoring during the internship and support in finding a PhD position.

Abstract

The internship aims at exploring a finite-element formulation of plasticity in the next generation FEniCSx problem-solving environment. The main goal is to propose an efficient and generic implementation which can tackle hardening plasticity models taking into account non-smooth yield criteria. The latter is achieved through convex optimization in the context of plasticity theory.

During the internship, a framework in Python is developed, which aims to solve plasticity problems in the FEniCSx environment. The implementation finds the solution of the local plasticity problem using convex optimization solvers. In this framework, a common approach to modelling plasticity through the return-mapping algorithm and conic optimization can be found. The latter works for a wide range of yield criteria, while the classical approach is well-suited for smooth yield surfaces such as the von Mises and Drucker-Prager surfaces, but it's more challenging to manage for non-smooth ones such as the Rankine yield surface. The research was performed under the assumptions of plane strain, an associative plasticity law and linear isotropic hardening. Several numerical tests were carried out for different conic solvers where the effect of the size of the vectorized conic optimization problem was analyzed. In addition, the idea of the custom assembly was developed to change the assembly process of the FEniCSx library easily. It is based on the concept of just-in-time compilation, which allows us to improve the time performance of some parts of the framework.

Keywords : plasticity, isotropic hardening, return-mapping algorithm, convex optimization, conic solver, FEniCS Project, cvxpy, Python, just-in-time compilation.

Résumé

Le stage vise à explorer une formulation par éléments finis de la plasticité dans l'environnement de résolution de problèmes FEniCSx de nouvelle génération. L'objectif principal est de proposer une implémentation efficace et générique capable de s'attaquer aux modèles de plasticité avec écrouissage en tenant compte de critères de plasticité non lisses. Ce dernier est obtenu grâce à l'utilisation de l'optimisation convexe dans le contexte de la théorie de la plasticité.

Pendant le stage, un framework en Python est développé, qui vise à résoudre des problèmes de plasticité dans l'environnement FEniCSx. L'implémentation trouve la solution du problème de plasticité locale à l'aide de solveurs d'optimisation convexes. Dans ce cadre, on peut trouver à la fois une approche commune de la modélisation de la plasticité à travers l'algorithme de retour radial et l'utilisation de l'optimisation conique. Cette dernière fonctionne pour un large spectre de critères de plasticité, tandis que l'approche classique convient bien aux surfaces de plasticité lisses telles que les surfaces de von Mises et de Drucker-Prager, mais il est plus difficile à gérer pour les surfaces non lisses telles que, par exemple, la surface de plasticité de Rankine. La recherche a été réalisée sous les hypothèses de déformations planes, d'une loi de plasticité associative et d'un écrouissage isotrope linéaire. Plusieurs tests numériques ont été effectués pour différents solveurs coniques où l'effet de la taille du problème d'optimisation conique vectorisé a été analysé. De plus, l'idée d'un assemblage custom a été développée pour modifier facilement le processus d'assemblage de la bibliothèque FEniCSx. Il est basé sur le concept de compilation just-in-time, ce qui nous permet d'améliorer les performances temporelles de certaines parties du framework.

Mots-clés : plasticité, écrouissage isotrope, algorithme de retour radial, optimisation convexe, solveur conique, FEniCS Project, cvxpy, Python, compilation just-in-time.

Synthèse du mémoire en français

Tout modèle de plasticité est déterminé par un critère. Il s'agit d'une fonction dépendant d'un état de contrainte-déformation actuel des caractéristiques de résistance du solide et du matériau. Dans ce travail, les critères de von Mises, Drucker-Prager et Rankine sont pris en compte. De plus, on prend en compte le écrouissage isotrope linéaire pour traiter des modèles plastiques complexes.

Lors de la résolution de problèmes élastiques-plastiques, l'algorithme de retour radial est généralement utilisé. Cette méthode est largement répandue dans la littérature. Par exemple, Bonnet et al. (2014) et Borst et al. (2012) décrivent la théorie des problèmes de plasticité et ses solutions analytiques et numériques. Bien que l'algorithme soit très populaire, on rencontre des difficultés à l'implémenter dans les cas où le critère de plasticité n'est pas assez lisse. Une solution possible à cet obstacle peut être de reformuler l'algorithme en termes de problème d'optimisation, plus particulièrement en minimisant l'énergie interne du solide, où les contraintes représentent le critère de plasticité.

Parmi la grande classe de problèmes d'optimisation convexe, il existe une sous-classe de programmation conique, où les contraintes représentent un cone convexe. Il existe des méthodes efficaces et rapides pour résoudre de tels problèmes. Comme de nombreux critères de plasticité sont représentés sous la forme de fonctions coniques, cela rend l'application de l'optimisation conique très prometteuse dans le contexte de la théorie de la plasticité.

Grâce à la modélisation numérique, on peux simuler une variété de processus physiques et extrapoler les résultats. La théorie de la plasticité ne fait pas exception. Parmi les méthodes numériques les plus efficaces, la méthode des éléments finis est reconnue, qui est souvent utilisée pour modéliser le comportement des solides. L'idée de la méthode est d'intégrer des systèmes d'équations aux dérivées partielles sur un domaine discret en résolvant des systèmes d'équations algébriques linéaires.

Il existe un grand nombre de bibliothèques d'éléments finis de complexité très différente. Parmi l'ensemble des projets open-source, FEniCSx nous convient le mieux en raison de sa simplicité et de sa commodité. Il permet de modéliser en écrivant du code en Python. Cela accélère non seulement le développement, mais donne également accès à un grand nombre d'autres bibliothèques écrites pour ce langage de programmation. Le projet FEniCSx (Alnaes et al., 2015)–(Logg, Mardal et al., 2012) est une combinaison de plusieurs bibliothèques, chacune ayant son propre objectif.

Le cœur de la bibliothèque FEniCSx est écrit en C\C++, mais il existe également une interface pour y accéder le langage de script Python de haut niveau. Les programmes écrits en Python pur perdent leurs performances au profit de leurs analogues du langage C\C++. De plus, le cœur de chaque bibliothèque d'éléments finis moderne contient généralement de plusieurs milles de lignes de code, ce qui ralentit le processus de développement de nouvelles fonctionnalités. Heureusement, il existe aujourd'hui un concept de compilation just-in-time (JIT), selon lequel on peut écrire un bloc de code compilé dans un programme écrit en Python. Ensuite, on peut l'exécuter et ce code sera plus performant. Ainsi, on peut remplacer les parties critiques d'un programme par des fonctions compilées en JIT. Cela accélère considérablement le processus de développement et augmente les performances de temps des programmes. Il supprime également l'obligation pour l'utilisateur d'une bibliothèque Python particulière d'étudier en détail tout son code source afin d'ajouter une nouvelle fonctionnalité nécessaire à la recherche. Dans ce travail, on utilise la bibliothèque numba (Lam et al., 2015) à ces fins.

Dans le cadre de ce travail, la modification du processus d'assemblage devient la clé d'un code écrit efficacement. Par conséquent, on propose également notre propre concept de travail avec le projet FEniCSx, où le processus d'assemblage est modifié à l'aide de la compilation JIT implémentée avec l'aide de la bibliothèque numba. Cela nous permet d'étendre le potentiel de la bibliothèque FEniCSx et d'améliorer les performances du code scientifique.

Dans la première partie de ce rapport, on présente les bases de la théorie de la plasticité et de l'algorithme de retour radial, ainsi que la formulation du problème d'optimisation convexe. Après cela, on parle d'un problème particulier de modélisation de la plasticité de von Mises. Sur cet exemple on teste des méthodes numériques. Ensuite, on décrit deux approches pour résoudre les problèmes de plasticité : celle classique, où on utilise l'algorithme de retour radial le plus courant et celle on applique la théorie de la programmation convexe. On écrit les lignes de code Python nécessaires pour implémenter ces méthodes en utilisant la bibliothèque FEniCSx. Après cela, on parle de performances et proposons nos propres fonctionnalités qui permet d'effectuer le processus d'assemblage sans modifier le code source de la bibliothèque FEniCSx. Ensuite, on démontre les résultats, les discute et parle des perspectives du travail effectué, en comparant les méthodes proposées de modélisation de la plasticité. À la fin, les conclusions sur le travail sont présentées.

En somme, un framework a été développé. Il permet de modéliser le comportement des solides en tenant compte des effets élastiques-plastiques sur l'exemple du problème de dilatation de cylindre. Dans ce cadre, on peut trouver à la fois une approche commune de la modélisation de la plasticité à travers l'algorithme de retour radial et l'utilisation de l'optimisation conique.

Cette dernière fonctionne pour un large spectre de critères de plasticité, tandis que l'approche classique convient bien aux surfaces de plasticité lisses telles que les surfaces de von Mises et de Drucker-Prager, mais il est plus difficile à gérer pour les surfaces non lisses telles que, par exemple, la surface de plasticité de Rankine. Le travail a été effectué pour un cas particulier de déformation plane, mais ses résultats peuvent être généralisés à des problèmes de plasticité tridimensionnelle en tenant compte de la loi de écrouissage isotrope complète, ainsi que d'autres critères de plasticité.

Les résultats de simulation ont été réalisés pour différents solveurs coniques, et l'effet de la taille du problème d'optimisation conique vectorisé a été analysé. Le sujet de la résolution de problème de plasticité à l'aide de solveurs coniques ne se limite pas à ces résultats. Le travail peut être complété par des recherches supplémentaires.

Dans le cadre de ce travail, le concept d'assemblage custom a été développé pour modifier le processus d'assemblage de la bibliothèque FEniCSx. Un prototype a été écrit pour y travailler en utilisant des exemples de problèmes élastique et plastique. Grâce à cette approche, la recherche a un grand potentiel pour le continuer, notamment dans le contexte de l'amélioration la performance temporelle.

Le code est accessible au public : (Latyshev, 2022).

Table of contents

List of tables

List of figures

Introduction

Plasticity effects are one of the most important mechanical phenomena of solid behaviour. These effects manifest themselves in a solid body in the form of irreversible plastic deformations, which inevitably affect the general behaviour of the body and its properties during loading. Taking into account these effects is essential for predicting the behaviour of real physical objects under various loading conditions. Therefore accurate estimation of plastic deformations, stresses and other quantities is required.

Any plasticity model is determined by some yield criterion. It depends on the current stress-strain state of the solid and material strength characteristics. In this paper von Mises, Drucker-Prager and Rankine criteria are considered. In addition, we take into account linear isotropic hardening to treat complex plastic models.

When solving elastic-plastic problems, the return mapping algorithm is usually used. This method is widely spread in the literature. For example, Bonnet et al. (2014) and Borst et al. (2012) describe the theory of plasticity problems and its analytical and numerical solutions. Although the return mapping algorithm is quite popular, we face difficulties implementing it in cases where the yield criterion is not smooth enough. A possible solution to this obstacle can be the reformulation of the algorithm in terms of the optimization problem, particularly in minimizing the solid internal energy, where constraints represent the yield criterion.

This work aims to write a working program that allows us to simulate the elastic-plastic behaviour of solids taking into account linear isotropic hardening using convex optimization. Thus, this work, on the one hand, offers a specific implementation of this approach, on the other hand, analyzes its nuances.

Among the large class of convex optimization problems, there is a subclass of conic programming where constraints represent convex cones. There are practical and fast methods to solve such problems. Since many yield criteria are represented in the form of conic functions, this makes the application of conic optimization very promising in the context of plasticity theory.

Numerical modelling allows us to simulate various physical processes and extrapolate results. The theory of plasticity is not an exception. Among the most effective numerical methods, the finite element method is recognized and often used to model the behaviour of solids. The idea of the method is to integrate systems of partial differential equations over a discrete domain by solving systems of linear algebraic equations.

There is a large number of finite element libraries of very different complexity. FEniCSx is best suited to us among open-source projects due to its simplicity and convenience. It allows modelling by writing code in Python. This not only speeds up development, but also gives access to countless other libraries written for this programming language. The FEniCSx Project (Alnaes et al., 2015) –(Logg, Mardal et al., 2012) is a combination of several libraries, each of which has its own purpose. Let us list some of them. DOLFINx (Logg & Wells, 2010) –(Logg, Wells & Mardal, 2012) is the computational environment of FEniCSx and implements the FEniCS Problem Solving Environment in C++ and Python. Unified Form Language (UFL) (Alnæs et al., 2014) is a domain-specific language for tje declaration of finite element discretizations of variational forms. Basix (Scroggs et al., 2022) is a finite element definition and tabulation runtime library. Each of them is an integral part of the scientific code development process.

The core of the FEniCSx library is written in C\C++, but there is an interface for accessing this functionality in the high-level Python scripting language. Programs written in pure Python lose performance to their analogues from the C\C++ language. In addition, the core of any modern finite element library usually contains thousands of lines of code, which slows down the process of developing new features. Fortunately, today there is a concept of just-in-time (JIT) compilation. According to it, we can write some compiled blocks of code in a program written in Python. Then we can compile them, and this code will be more efficient. Thus, we can replace critical parts of a program with JIT compiled functions. This significantly speeds up the development process and increases the time performance. In this work, we use the numba library (Lam et al., 2015) for these purposes.

In the context of this work, changing the assembly process becomes the key to efficiently written code that fulfils its purposes. Therefore, we also offer our own concept of working with the FEniCSx project, where the assembly process is changed using JIT compilation implemented through the numba library. This allows us to expand the the FEniCSx library's potential and improve the scientific code's performance.

The result of the research is a framework written in Python. It finds a numerical solution to the problem without binding the algorithm to a specific type of yield function. The written code is publicly available at (Latyshev, 2022).

In the first part of this manuscript, we present the basics of plasticity theory (3.1.1) and the return-mapping algorithm (3.1.2), as well as formulation of the convex optimization problem (3.1.3). After that, we talk about a specific problem of modelling von Mises plasticity (3.1.5), on the example of which we are going to test numerical methods. Then we describe two approaches

to solving plasticity problems : the classical one using the most common return-mapping algorithm (3.2.4), and then its application in the context of convex programming (3.2.5). We write out lines of Python code necessary to implement these methods using the FEniCSx library. After that, we talk about performance and offer our own functionality that allows us to affect the assembly process without changing the source code of the FEniCSx library (3.3). Then we demonstrate the results (4), discuss them and talk about the prospects of the work done (5), comparing the proposed methods of plasticity modelling. In the end, the conclusions about the work are presented (5).

1 Context and laboratory presentation

The Navier Laboratory is a joint research unit of the Ecole des Ponts et Chaussées (ENPC), the Gustave Eiffel University (UGE) and the National Center for Scientific Research (CNRS), situated in the Descartes city of Marne-la-Vallée. The laboratory's staff (nearly 170 people) conduct research on the mechanics and physics of materials, structures and geomaterials, and on their applications to geotechnics, civil engineering, transport, geophysics and energy. The societal issues concern sustainable construction, natural risks, the environment and energy. In the development of the mechanical and physical laws relating to these themes, the studies undertaken are both experimental and theoretical. They rely on a variety of equipment, some of which are unique in their kind. While ensuring the balance between academic excellence and economic support, the Navier Laboratory is strongly invested in partnership research, whether public or private. In particular, he is involved as a leader or partner in seven teaching and/or research chairs with public and private industrial partners. Another major asset of the laboratory is its strong involvement in teaching, in particular at the Ecole des Ponts ParisTech. The laboratory has also developed its relations with other components of IFSTTAR in order to explore a wide field of upstream research resulting from concrete applications with strong social utility.

FIGURE 1 – Logo of the Navier laboratory

This internship was carried out at the Materials and Architectural Structures team under the direction of Jeremy Bleyer, a researcher at the Navier laboratory and professor at the Ecole des Ponts ParisTech, and under the co-direction of Corrado Maurini, professor at Sorbonne University.

Doctor Bleyer is interested in modelling various aspects of mechanical behaviour : fracture of brittle materials, limit analysis/yield design for civil engineering structures, multi-scale approaches for heterogeneous materials and structures (fibre materials, composite plates), viscoplastic fluid flows, etc. From a fundamental point of view, he uses different numerical methods : FEM, phase-

field approaches, convex optimization etc. He implements mechanical models in digital code using modern and powerful modelling libraries such as FEniCSx, cvxpy, etc.

Doctor Maurini also specializes in computational mechanics. His current research interests include the stability of structures, plates and shells, large deformations and instabilities in soft solids, variational phase-field models and others. He also develops code for the simulation of those mechanical phenomena using such libraries as FEniCSx.

2 Literature review

This research is mainly based on the work of Bruno et al. (2020), where the authors describe in detail the relationship between conic optimization problems and return-mapping algorithms for associative isotropic hardening plasticity. The article contains mathematical formulations of conic optimization problems for special cases of plasticity models, where yield criteria are considered as constraints divided into two groups. The first one contains positive semidefinite yield criteria : Rankine and Mohr-Coulomb. The second group includes second-order cone yield criteria : von Mises and Drucker-Prager. As a result, optimization problems are reduced to a canonical form and algorithms for their solution are proposed. In this work, we use the external cvxpy (Diamond & Boyd, 2016) –(Agrawal et al., 2018) library, which is sufficient to formulate an objective function and constraints using library tools to solve the optimization problem. This will allow us to write a single code that solves the original problem without necessity adapting it to each plasticity criterion.

There are a large number of solvers for conic optimization problems. As part of this work, we have selected only three : SCS, ECOS and MOSEK. SCS (O'Donoghue et al., 2016) is well suited for solving large-scale quadratic convex cone problems. ECOS (Domahidi et al., 2013) is designed for a second-order cone programming and specifically for embedded applications. It works faster than many conic solvers for small problems. And finally, MOSEK (ApS, 2019) is a commercial one. It works with different types of large-scale tasks. We used an academic license to access it.

Also, this work is a logical extension of the example of Bleyer (2018), where a numerical solution of von Mises plasticity for the last version of Fenics 2019 is described. The current work uses the same approach, but it's more efficient due to the new features of the latest version of FEniCSx. In addition, this work has been significantly expanded through convex optimization.

Since we are dealing with nonlinear problems, Newton's method is a popular approach to solve them, so many libraries implementing it. In our work, we use a conventional Newton method and a quasi-Newton one from the petsc library (Balay et al., 2022). This library is well-known for effectively implementing many vital routines for scientific programming.

3 Methodology

3.1 Theory

In this section, we will describe the mathematical formulation for solving the elastic-plastic body equilibrium problem, the return mapping algorithm and the convex optimization problem in the context of plasticity theory. To simplify further discussions, we will introduce some notation here.

It is necessary to solve a system of differential equations describing the equilibrium of a solid body under the assumption of small deformations. We will use the notation for the stress tensor $\underline{\underline{\sigma}}$ with components σ_{ij} in the Cartesian coordinate system $\underline{x} = (x, y, z)$. By denote the linear strain tensor $\varepsilon_{ij} = (\partial u_i/\partial x_j + \partial u_j/\partial x_i)/2$, where $\underline{u} = (u_x, u_y, u_z)$ is a displacement vector.

In this paper, the problems are considered in the plane strain case. This means that the following equalities hold

$$u_z = 0,$$
$$\varepsilon_{xz} = \varepsilon_{yz} = \varepsilon_{zz} = 0.$$

We note here that the results of this work can be easily generalized to the three-dimensional case without a plane strain assumption.

3.1.1 Plasticity

One of the simplest models describing the nonlinear behaviour of solids is an elastic-plastic one. The idea is that irreversible deformations occur at a certain moment of loading. They do not disappear when the external load is removed, as it happens for an elastic body model. Such deformations are called plastic. They arise at the moment when, with an increase in the external load, the values of the stress tensor reach critical ones. These limit values are determined by yield criterion, the explicit form of which defines various elastic-plastic models. This paper focused on models defined by von Mises, Drucker-Prager and Rankine yield criteria.

We assume the hypothesis of small deformations, where the total strains $\underline{\underline{\varepsilon}}$ is additively decomposed as

$$\underline{\underline{\varepsilon}} = \underline{\underline{\varepsilon}}^e + \underline{\underline{\varepsilon}}^p, \tag{1}$$

where $\underline{\underline{\varepsilon}}^e$ and $\underline{\underline{\varepsilon}}^p$ are elastic and plastic strains respectively.

Isotropic linear elastic behaviour is expressed as the following dependency

$$\underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}^e = \left(3k\underline{\underline{\underline{\underline{J}}}} + 2\mu\underline{\underline{\underline{\underline{K}}}}\right) : \underline{\underline{\varepsilon}}^e, \tag{2}$$

where $\underline{\underline{\underline{\underline{C}}}}$ is the fourth-order stiffness tensor, $k = (\lambda + \frac{2}{3}\mu)$ is a bulk modulus, λ and μ are the first and the second Lamé parameters, $\underline{\underline{\underline{\underline{K}}}}$ and $\underline{\underline{\underline{\underline{J}}}}$ are the fourth-order tensors associated with the deviatoric operator and a unit tensor respectively.

Plasticity is characterized by the existence of a yield surface (usually convex) described by the following yield condition : Plastic straining will take place only if the yield function f vanishes :

$$f(\underline{\underline{\sigma}}) \leq 0 \tag{3}$$

In models with hardening, yield function depends on the history of the body load. Here we consider linear isotropic hardening. Thus, the condition for the occurrence of plastic deformation is the following equality

$$f(\underline{\underline{\sigma}}, p) = 0, \tag{4}$$

where the internal variable $p = \sqrt{\frac{2}{3}\underline{\underline{\varepsilon}}^p : \underline{\underline{\varepsilon}}^p}$ is accumulated plastic strain.

We also assume that the plastic strain rate is proportional to the gradient of the yield function, which is expressed as a formula of the associative flow rule

$$\dot{\underline{\underline{\varepsilon}}}^p = \dot{\lambda}\frac{\partial f(\underline{\underline{\sigma}}, p)}{\partial \underline{\underline{\sigma}}}, \tag{5}$$

where the positive scalar $\dot{\lambda}$ determines the magnitude of the plastic flow.

The associative hardening law has the following form

$$\dot{p} = -\dot{\lambda}\frac{\partial f(\underline{\underline{\sigma}}, p)}{\partial p}. \tag{6}$$

We write loading/unloading conditions as follows

$$\dot{\lambda} \geq 0, \quad f(\underline{\underline{\sigma}}, p) \leq 0, \quad \dot{\lambda} f(\underline{\underline{\sigma}}, p) = 0. \tag{7}$$

Thus, taking into account the described above assumptions (1)–(7) we can formulate the elasto-plastic model. Let us consider the following system

$$\Omega : \operatorname{div} \underline{\underline{\sigma}} = 0, \tag{8}$$

$$\Omega : \underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}^e, \tag{9}$$

$$\partial\Omega_{\mathrm{N}} : \underline{\underline{\sigma}} \cdot \underline{n} = q \cdot \underline{n}, \tag{10}$$

$$\partial\Omega_{\mathrm{D}} : \underline{u} = \underline{u}_D, \tag{11}$$

where the equilibrium equation (8) is defined in the domain Ω as well as the constitutive equation (9) of linear elasticity, $\underline{n}$ is a surface normal to the boundary $\partial\Omega_{\mathrm{N}}$, where we apply Neuman boundary conditions (10). Dirichlet boundary conditions are defined by the displacements equality (11) on the boundary $\partial\Omega_{\mathrm{D}}$.

We call the described above system of partial differential equations (8)–(11) and assumptions (1)–(7) elastoplastic constitutive equations. In the next part, we will discuss in detail the numerical solutions of such an equilibrium problem.

3.1.2 Numerical solution of elastoplastic constitutive equations

The processes considered here are quasi-static, so the solution of this system is carried out in stages. In the process of solving elastoplastic constitutive equations, we load the body, gradually increasing the loading at each step. We solve the system (1)–(11) using the finite element and Newton methods on every loading step.

First of all, we need to define a weak formulation of our problem to use the finite element method. So we introduce here the space of admissible displacements V

$$V = \{\underline{u} = (u_x, u_y) \in H^1(\Omega) \,|\, \underline{u} = \underline{u}_D \text{ on } \partial\Omega_D\}, \tag{12}$$

where $H^1(\Omega)$ is the first-order Sobolev space. The variational problem will look like this

$$\text{Find } \underline{u} \in V \text{ such that,} \tag{13}$$

$$R(\underline{u}) = \int_\Omega \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \,\mathrm{d}x - F_{\mathrm{ext}}(\underline{v}) = 0, \quad \forall \underline{v} \in V, \tag{14}$$

where $F_{\mathrm{ext}}(\underline{v})$ is a linear form corresponding to the applied external forces and $\underline{\underline{\sigma}}(\underline{u})$ is the second-

rank stress tensor, which nonlinearly depends on the displacement vector $\underline{u}$.

The variational equation (14) is nonlinear that can be solved through successive linearizations using the Newton algorithm. The linearized problem looks like this one

$$\text{Find } \Delta\underline{u} \in V \text{ such that,} \tag{15}$$

$$\int_\Omega \left(\frac{\partial \underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}(\Delta\underline{u}))}{\partial \Delta\underline{\underline{\varepsilon}}} : \underline{\underline{\varepsilon}}(\underline{u}) \right) : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}x = -\int_\Omega \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}x + F_{\text{ext}}(\underline{v}), \tag{16}$$

On every loading step $n+1$, we solve the problem (13)–(14) and get the displacement $\underline{u}_{n+1}$ using the Newton method, where at each iteration, the numerical value of the displacements increment $\Delta\underline{u}$ is calculated using the finite element method for the variational problem (15)–(16). Once the Newton method has converged, we can update the displacement vector $\underline{u}_{n+1} = \underline{u}_n + \Delta\underline{u}$ and move on to the next step.

Let us write down the introduced above problems in their discrete forms, but firstly we show the discretized versions of the equations (1)–(7), where we use a one-point Euler backward integration rule for "time" derivatives :

$$\underline{\underline{\sigma}}^{\text{elas}}_{n+1} = \underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : \Delta\underline{\underline{\varepsilon}}, \tag{17}$$

$$p^{\text{elas}}_{n+1} = p_n, \tag{18}$$

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}^{\text{elas}}_{n+1} - \Delta\underline{\underline{\sigma}}, \tag{19}$$

$$p_{n+1} = p^{\text{elas}}_{n+1} + \Delta p, \tag{20}$$

$$\Delta\underline{\underline{\varepsilon}}^p = \Delta\lambda \frac{\partial f}{\partial \underline{\underline{\sigma}}}(\underline{\underline{\sigma}}_{n+1}, p_{n+1}), \tag{21}$$

$$\Delta p = -\Delta\lambda \frac{\partial f}{\partial p}(\underline{\underline{\sigma}}_{n+1}, p_{n+1}), \tag{22}$$

$$\Delta\lambda \geq 0, \quad f(\underline{\underline{\sigma}}_{n+1}, p_{n+1}) \leq 0, \quad \Delta\lambda \cdot f(\underline{\underline{\sigma}}_{n+1}, p_{n+1}) = 0, \tag{23}$$

where the increment $\Delta\underline{\underline{\varepsilon}} = \underline{\underline{\varepsilon}}(\Delta\underline{u})$ depends on the displacement $\Delta\underline{u}$ calculated on a current loading step, $\underline{\underline{\sigma}}^{\text{elas}}_{n+1}$ and p^{elas}_{n+1} are elastic predictors, $\Delta\underline{\underline{\sigma}}$ and Δp are the stress and plastic strain corrections respectively, $\Delta\underline{\underline{\varepsilon}}^p$ is an increment of plastic strain. Then we write out a discrete form of the

variational problem (13)–(14) :

$$\text{Find } \underline{u}_{n+1} \in V \text{ such that,} \tag{24}$$

$$R(\underline{u}_{n+1}) = \int_\Omega \underline{\underline{\sigma}}(\underline{u}_{n+1}) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x - F^{\text{ext}}_{n+1}(\underline{v}) = 0, \quad \forall \underline{v} \in V, \tag{25}$$

and its linearized version (15)–(16) by denoting the derivate $\frac{\partial \underline{\underline{\sigma}}}{\partial \underline{\underline{\Delta\varepsilon}}}$ by $\underline{\underline{\underline{\underline{C}}}}_{\text{tang}}$:

$$\text{Find } \Delta\underline{u} \in V \text{ such that,} \tag{26}$$

$$\int_\Omega \left(\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1} : \underline{\underline{\varepsilon}}(\Delta\underline{u}) \right) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x = -\int_\Omega \underline{\underline{\sigma}}_{n+1} : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x + F^{\text{ext}}_{n+1}(\underline{v}), \tag{27}$$

where $\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1}$ is a the forth-rank tangent stiffness tensor on the $n+1$ loading step.

Now we have all ingredients to introduce a quite common algorithm for solving the elasto-plastic problems : the return-mapping procedure. It consists in finding new stress $\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}(\underline{u}_{n+1})$ and cumulative plastic strain p_{n+1} verifying the current plasticity condition from previous stress $\underline{\underline{\sigma}}_n$, cumulative plastic strain p_n and an increment of total deformation $\Delta\underline{\underline{\varepsilon}}$. The whole procedure checks on every Newton iteration if the trial stress $\underline{\underline{\sigma}}^{\text{elas}}_{n+1}$ and the trial variable p^{elas}_{n+1} go beyond the boundary of the yield surface. In other words, if $f(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p^{\text{elas}}_{n+1}) > 0$, then the stress tensor and the cumulative plastic strain should be corrected by projecting them on the surface $f(\underline{\underline{\sigma}}_{n+1}, p_{n+1}) = 0$. It means that we need to solve the nonlinear system (17)–(23). This can be done using an additional Newton method, the internal one.

In total, we follow the next algorithm to solve elastoplastic constitutive equations (1)–(11) of the original problem

1. Divide the external loading F_{ext} on N steps
2. For each loading step $n+1 \in 1 \dots N$:
   (a) Solve the variational problem (24)–(25) using the Newton method, where for each Newton iteration :
      i. Solve the linearized problem (26)–(27) using finite element method
      ii. Calculate elastic predictors $\underline{\underline{\sigma}}^{\text{elas}}_{n+1} = \underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : \Delta\underline{\underline{\varepsilon}}$ and $p^{\text{elas}}_{n+1} = p_n$
      iii. Carry out the return-mapping procedure using the internal Newton method and solve the system (17)–(23), if the yield condition $f(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p^{\text{elas}}_{n+1}) > 0$ is true
      iv. Update $\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}^{\text{elas}}_{n+1} - \Delta\underline{\underline{\sigma}}$, $p_{n+1} = p^{\text{elas}}_{n+1} + \Delta p$ and $\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1} = \frac{\partial \underline{\underline{\sigma}}_{n+1}}{\partial \underline{\underline{\Delta\varepsilon}}}$

(b) Update the total displacement vector $\underline{u}_{n+1} = \underline{u}_n + \Delta\underline{u}$

As seen from the described algorithm, modelling the plastic behaviour of solids requires significant numerical resources. Fortunately, for some special cases, depending on the type of function f, it is possible to write explicit expressions for $\Delta\lambda, \Delta\underline{\underline{\sigma}}, \Delta p,$ and $\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1}$, which will be shown in this work later on using the example of one problem. Thus the internal Newton method can be skipped.

This algorithm works correctly when the function f is smooth, i.e. its gradient exists. Therefore, the question arises : what should we do in cases where the function f is not smooth enough to apply the return-mapping procedure described above ? For example, the Rankine criterion requires eigenvalues of $\underline{\underline{\sigma}}_{n+1}$, explicit expressions of which cannot be easily obtained in the three-dimensional case. A solution may be to reformulate the original problem in terms of the convex optimization problem, which is the next chapter's subject.

3.1.3 Plasticity using convex optimization

When we talk about optimization problems (or mathematical programming, MP), we often have in mind a minimization problem such as

$$\begin{cases} \min\limits_{\underline{x}} F(\underline{x}), \\ f(\underline{x}) \leq 0, \end{cases} \tag{28}$$

where F is an objective function and f is a constraint. If these functions are convex, we deal here with a convex optimization problem (or convex programming, CP).

The particular interest to us is a subfield of convex optimization, when constraints represent a convex cone, in other words, conic optimization. For this type of problems, the solution is unique, and there are efficient and fast numerical solvers to find it. The authors of this work do not consider it necessary to delve in detail into the specifics of solving optimization problems and their algorithms. We will refer only to articles by other authors whose solvers were used in the current research.

Indeed, conic programming has been recognized as a suitable method for solving elastoplastic constitutive equations via the MP approach (Bruno et al., 2020). This stems from the fact that many yield criteria reported in the literature can be expressed as second-order and semidefinite cone constraints. In particular, the von Mises and Drucker-Prager criteria can be expressed as a

second-order conic constraint and the Rankine criterion as a semidefinite one.

We just need to reformulate the original return-mapping algorithm (17)–(23) in terms of conic programming. Let us consider the following conic optimization problem

$$\begin{cases} \min\limits_{\underline{\underline{\sigma}},p} F(\underline{\underline{\sigma}},p), \\ f(\underline{\underline{\sigma}},p) \leq 0, \end{cases} \tag{29}$$

where the function $f(\underline{\underline{\sigma}},p)$ is a yield criterion and the free energy F of an elastoplastic material is expressed as follows

$$F(\underline{\underline{\sigma}},p) = \frac{1}{2}(\underline{\underline{\sigma}}^{\text{elas}}_{n+1} - \underline{\underline{\sigma}}) : \underline{\underline{\underline{\underline{S}}}} : (\underline{\underline{\sigma}}^{\text{elas}}_{n+1} - \underline{\underline{\sigma}}) + \frac{1}{2}H(p^{\text{elas}}_{n+1} - p)^2, \tag{30}$$

where H is an isotropic hardening modulus and $\underline{\underline{\sigma}}^{\text{elas}}_{n+1} = \underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : \Delta\underline{\underline{\varepsilon}}$ and $p^{\text{elas}}_{n+1} = p_n$ are elastic predictors.

Thus, the solution $(\underline{\underline{\sigma}}^*, p^*)$ of the problem (29) is the closest projection of $(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p^{\text{elas}}_{n+1})$ on the yield surface with respect to the distance induced by $\underline{\underline{\underline{\underline{S}}}}$ and H, i.e. $(\underline{\underline{\sigma}}_{n+1}, p_{n+1})$.

We note here that the Newton method requires the calculation of the derivative $J = R'(\underline{u}_{n+1})$. In the following sections, we will talk about three ways to do that : analytically, using the quasi-Newton method and using the `derivative` function of the cvxpy library. In the case of the convex optimisation approach we use the quasi-Newton method, which approximates the derivative numerically. The fact is that this approach is applicable to a wide range of plasticity models, so it is not possible to calculate the derivative J analytically.

By analogy with the previous section 3.1.2, we describe an algorithm for solving elastoplastic constitutive equations (1)–(11) using convex optimization :

1. Divide the external loading F_{ext} on N steps
2. For each loading step $n+1 \in 1 \dots N$:
   (a) Solve the variational problem (24)–(25) using the quasi-Newton method, where for each Newton iteration :
      i. Find a numerical approximation of the derivative $J = R'(\underline{u}_{n+1})$, i.e. of the linearized problem (26)– (27)
      ii. Solve the latter using the finite element method
      iii. Calculate $\underline{\underline{\sigma}}^{\text{elas}}_{n+1} = \underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : \Delta\underline{\underline{\varepsilon}}$ and $p^{\text{elas}}_{n+1} = p_n$

iv. Carry out the return-mapping procedure by solving the conic optimization problem (29), if the yield condition $f(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p^{\text{elas}}_{n+1}) > 0$ is true

v. Update $\underline{\underline{\sigma}}_{n+1}$ and p_{n+1}

(b) Update the total displacement vector $\underline{u}_{n+1} = \underline{u}_n + \Delta\underline{u}$

3.1.4 Yield criteria

Many yield criteria can be represented as a function f, depending on the stress-strain properties of the material and various strength parameters. The following inequality defines all possible internal states of a solid in the $(\underline{\underline{\sigma}}, p)$ space :

$$f(\underline{\underline{\sigma}}, p, \sigma_C, \sigma_T, \dots) \leq 0, \tag{31}$$

where σ_T and σ_C are the tensile and compressive strengths of the material, respectively. In the inequality (31), the strict equality sign holds for stress states on the yield contour and where the inequality sign is valid whenever stresses are inside the yield contour and cause only elastic deformations.

There are various yield criteria. Some of them are based on real experiments. Within the framework of this work, we consider only 3 of them.

For the **von Mises criterion** we have

$$\sigma_{\text{eq}} \leq r(\sigma_0, p),$$

where $\sigma_{\text{eq}} = \sqrt{\frac{3}{2}\underline{\underline{s}} : \underline{\underline{s}}}$. The function $r(\sigma_0, p)$ is a relation between the uniaxial yield strength σ_0 and the internal variable p associated with plastic strain. In our case of linear isotropic hardening $r(\sigma_0, p) = \sigma_0 + Hp$, where H is an isotropic hardening modulus.

The yield condition of **Drucker-Prager** is a smooth approximation of the Mohr-Coulomb yield surface, which is a conical one in the principal stress space. The yield function looks as follows :

$$\sigma_{\text{eq}} + \alpha \text{tr}\underline{\underline{\sigma}} \leq r(\sigma_0, p),$$

where α is a material constant.

A correspondent MP problem to both criteria looks like this one

$$\begin{cases} \min\limits_{\underline{\underline{\sigma}},p} F(\underline{\underline{\sigma}}, p), \\ \sigma_{\text{eq}} + \alpha \text{tr}\underline{\underline{\sigma}} - r(\sigma_0, p) \leq 0, \end{cases}$$

where we get the von Mises criterion by setting the parameter α to zero.

The **Rankine criterion** limits tensile and compression stresses :

$$\begin{aligned} \sigma_{\text{I}} &\leq r(\sigma_T, p), \\ -\sigma_{\text{III}} &\leq r(\sigma_C, p), \end{aligned}$$

where $\sigma_{\text{I}} \geq \sigma_{\text{II}} \geq \sigma_{\text{III}}$ are principal stresses. By this example, we can consider a case of the plasticity model, where the yield function is not a simple inequality. It contains several constraints. So it is not easy to calculate its analytical formula explicitly. The correspondant convex optimization problem :

$$\begin{cases} \min\limits_{\underline{\underline{\sigma}},p} F(\underline{\underline{\sigma}}, p), \\ \sigma_{\text{I}} - r(\sigma_T, p) \leq 0, \\ -\sigma_{\text{III}} - r(\sigma_C, p) \leq 0. \end{cases}$$

3.1.5 Problem formulation

In order to demonstrate our ideas on a concrete example, we chose a cylinder expansion problem in the two-dimensional case in a symmetric formulation. The image 2 shows the domain, where symmetry conditions are set on the left and bottom sides and pressure is set on the inner wall

$$\Omega : \text{div}\, \underline{\underline{\sigma}} = 0, \tag{32}$$

$$\Omega : \underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}^e, \tag{33}$$

$$\partial\Omega_{\text{internal}} : \underline{\underline{\sigma}} \cdot \underline{n} = q \cdot \underline{n}, \tag{34}$$

$$\partial\Omega_{\text{left}} : u_x = 0, \tag{35}$$

$$\partial\Omega_{\text{bottom}} : u_y = 0, \tag{36}$$

The von Mises criterion was chosen as the yield criterion, taking into account linear isotropic

hardening :

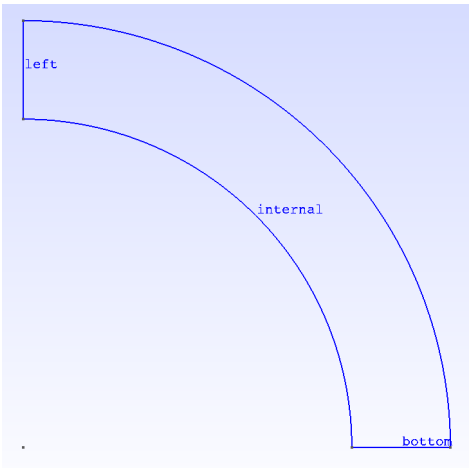
$$f(\underline{\underline{\sigma}}) = \sqrt{\frac{3}{2}\underline{\underline{s}} : \underline{\underline{s}}} - \sigma_0 - Hp \leq 0. \tag{37}$$

The weak formulation of the problem (32)–(36) looks like this

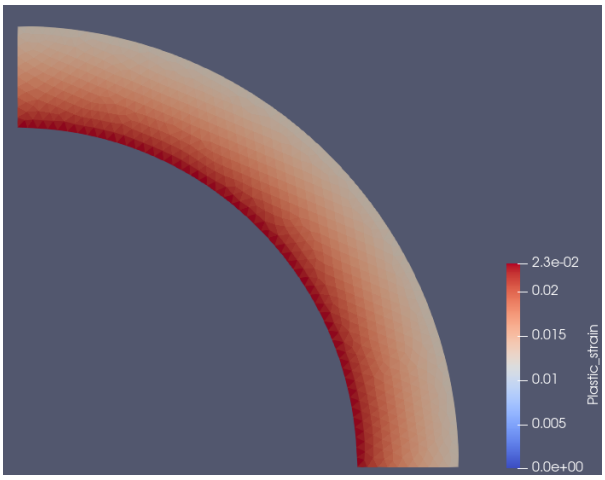
$$\text{Find } \underline{u} \text{ such that} \tag{38}$$

$$\int_\Omega \underline{\underline{\sigma}}_{n+1}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x - q_{n+1} \int_{\partial\Omega_{\text{internal}}} \underline{n} \cdot \underline{v}\,\mathrm{d}x = 0, \quad \forall \underline{v} \in V. \tag{39}$$

It will be progressively increased from 0 to $q_{\text{lim}} = \frac{2}{\sqrt{3}}\sigma_0 \log\frac{R_e}{R_i}$ which is the analytical collapse load for a perfectly-plastic material, where R_e and R_i are the external and internal radii of the cylinder respectively.



Geometry



Plastic strain on final load step

FIGURE 2 – Cylinder expansion problem

In this particular case, the return-mapping procedure can be performed analytically. A detailed conclusion of an elastic-plastic model can be found in (BONNET et al., 2014), and the implementation in the form of program code for Fenics 2019 is located on (BLEYER, 2018). Here we will limit ourselves to the relations already deduced

$$\Delta p = \begin{cases} \frac{1}{3\mu+H}, & \text{if } f(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p_{n+1}) \geq 0, \\ 0, & \text{otherwise,} \end{cases} \tag{40}$$

$$\Delta\underline{\underline{\sigma}} = \beta_{n+1}\underline{\underline{s}}_{n+1}, \tag{41}$$

where $\beta_{n+1} = \frac{3\mu}{\sigma^{\text{elas,eq}}_{n+1}}\Delta p$.

The stress derivative is written with the following formula

$$\frac{\mathrm{d}\underline{\underline{\sigma}}_{n+1}}{\mathrm{d}\Delta\underline{\underline{\varepsilon}}} = \underline{\underline{\underline{\underline{C}}}}^{\mathrm{tang}}_{n+1} = \underline{\underline{\underline{\underline{C}}}} - 3\mu\left(\frac{3\mu}{3\mu+H} - \beta_{n+1}\right)\underline{\underline{N}}_{n+1} \otimes \underline{\underline{N}}_{n+1} - 2\mu\beta_{n+1}\underline{\underline{\underline{\underline{J}}}} \tag{42}$$

where $\underline{\underline{N}}_{n+1} = \frac{\underline{\underline{s}}^{\mathrm{elas}}_{n+1}}{\sigma^{\mathrm{elas,eq}}_{n+1}}$.

So we can explicitly write out the linearization of the variational problem (38)–(39)

$$\text{Find } \Delta\underline{u} \in V \text{ such that,} \tag{43}$$

$$\int_\Omega \left(\underline{\underline{\underline{\underline{C}}}}^{\mathrm{tang}}_{n+1} : \underline{\underline{\varepsilon}}(\Delta\underline{u})\right) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x = -\int_\Omega \underline{\underline{\sigma}}_{n+1} : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x + q_{n+1}\int_{\partial\Omega_{\mathrm{internal}}} \underline{n}\cdot\underline{v}\,\mathrm{d}x, \quad \forall \underline{v} \in V. \tag{44}$$

Explicit expressions (40)–(42) are possible thanks to the form of von Mises criterion (37). Similar expressions can be obtained for other criteria. As part of this work, a code has been written that allows working with the plasticity of Drucker-Prager. In this case a detailed derivation of similar expressions can be found in the appendix section A.

Now we have everything necessary for successful modelling of the von Mises elasto-plasticity. In the next part, we will talk in detail about ways to solve this problem using various approaches via the FEniCSx and cvxpy libraries.

3.2 Development

The development of a program for modelling the cylinder expansion problem was carried out with the support of the finite element library FEniCSx. As mentioned earlier, the initial problem has different formulations, for each of which a separate program is written. The open source code is available on the Github repository Latyshev, 2022.

In this section, we will talk about some nuances of modelling an elastic-plastic material using the FEniCSx library, as well as how to combine this modelling with other Python libraries that allow us to solve convex optimization problems.

3.2.1 Voigt notation

It is worth noting that the problems described above are formulated in terms of tensors of the second and fourth ranks, which complicates the development stage of the work. Taking into account the symmetry of the stress, strain and stiffness tensors, we propose to reformulate these problems in the Voigt notation.

According to this notation the stress tensor representation is a vector

$$\boldsymbol{\sigma} = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy})^T,$$

the strain one is a vector

$$\boldsymbol{\varepsilon} = (\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \varepsilon_{xy})^T,$$

where the shear components xy of vectors $\boldsymbol{\sigma}$ and $\boldsymbol{\varepsilon}$ must be multiplied by $\sqrt{2}$ for some tasks where the correct calculation of such expressions as $\underline{\underline{\sigma}} : \underline{\underline{\varepsilon}}$, $\underline{\underline{s}} : \underline{\underline{s}}$, etc is required. Other tensors, for example, deviatoric parts $\boldsymbol{s}$ and $\boldsymbol{e}$ follow the same notation logic. The stiffness matrix of linear elasticity is

$$\boldsymbol{C} = \begin{pmatrix} \lambda + 2\mu & \lambda & \lambda & 0 \\ \lambda & \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & 0 & 2\mu \end{pmatrix}.$$

For simplicity, we will also highlight the vectors in bold.

Thus we rewrite in the Voigt notation the weak formulation of the cylinder expansion problem (38)–(39) :

$$\text{Find } \boldsymbol{u}_{n+1} \in V \text{ such that,} \tag{45}$$

$$R(\boldsymbol{u}_{n+1}) = \int_\Omega \boldsymbol{\sigma}_{n+1} \cdot \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}x - q_{n+1} \int_{\partial\Omega_{\text{internal}}} \boldsymbol{n} \cdot \boldsymbol{v} \, \mathrm{d}x = 0, \quad \forall \boldsymbol{v} \in V, \tag{46}$$

where $\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}(\boldsymbol{u}_{n+1})$ implicitly depends on $\boldsymbol{u}_{n+1}$, and its linearized version (43)–(44) :

$$\text{Find } \Delta\boldsymbol{u} \in V \text{ such that,} \tag{47}$$

$$\int_\Omega \left(\boldsymbol{C}_{n+1}^{\text{tang}} \, \Delta\boldsymbol{\varepsilon}\right) \cdot \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}x = -\int_\Omega \boldsymbol{\sigma}_{n+1} \cdot \boldsymbol{\varepsilon}(\boldsymbol{v}) \, \mathrm{d}x + q_{n+1} \int_{\partial\Omega_{\text{internal}}} \boldsymbol{n} \cdot \boldsymbol{v} \, \mathrm{d}x, \tag{48}$$

where $\boldsymbol{C}_{n+1}^{\text{tang}}$ is a the tangent stiffness matrix and $\Delta\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}(\Delta\boldsymbol{u})$.

The convex optimization problem (29) in Voigt notation is

$$\begin{cases} \min\limits_{\boldsymbol{\sigma},p} F(\boldsymbol{\sigma}, p), \\ f(\boldsymbol{\sigma}, p) \leq 0, \end{cases} \tag{49}$$

where the free energy F of an elastoplastic material is expressed as follows

$$F(\boldsymbol{\sigma}, p) = \frac{1}{2}(\boldsymbol{\sigma}_{n+1}^{\text{elas}} - \boldsymbol{\sigma})^T \boldsymbol{S}(\boldsymbol{\sigma}_{n+1}^{\text{elas}} - \boldsymbol{\sigma}) + \frac{1}{2}H(p_{n+1}^{\text{elas}} - p)^2, \tag{50}$$

where $\boldsymbol{S}$ is an inverted stiffness matrix.

3.2.2 Stress tensor values

Solving the equilibrium problem of an elastic-plastic solid, we find the values of the displacement vector at each node of the finite element mesh. To find the stress field, we need to calculate the derivative of the displacements. The displacements are defined in a second-order continuous Galerkin space, which does not guarantee the continuity of the first derivative at the mesh nodes. A common strategy for calculating stresses is an approach in which the values of the first derivatives of displacements are calculated at the Gauss nodes of each element. FEniCSx has the necessary functionality to implement this strategy.

There is an entity `fem.Expression`, which represents a mathematical expression based on the UFL package. It can contain a global coefficient `fem.Function`, another essential brick of FEniCSx. The last one represents global finite vectors, such as the displacement one. Such coefficients are based on a finite element space : continuous Galerkin, discontinuous Galerkin, quadrature and others. An example below is where finite element spaces and variables associated with them are initiated.

```
deg_u = 2
deg_stress = 2
V = fem.VectorFunctionSpace(mesh, ("CG", deg_u))
We = ufl.VectorElement("Quadrature", mesh.ufl_cell(), degree=deg_stress, dim=4, quad_scheme='default')
W = fem.FunctionSpace(mesh, We)

u = fem.Function(V, name="Total_displacement")
du = fem.Function(V, name="Iteration_correction")
Du = fem.Function(V, name="Current_increment")
sig = fem.Function(W, name="Current_stress")
```

Note that the stress vector `sig` is defined on the quadrature space, i.e. the values of the stress vector are stored in Gaussian nodes.

It is required to use the `eval` method of the entity `fem.Expression`, which interpolates the mathematical expression of stresses in Gaussian nodes. For our purpose, we wrote the function `interpolate_quadrature`, which implements this idea.The code below is an example of calculating the stress vector for the whole domain.

```
deps = eps(Du)
sig_, N_, beta_, dp_ = proj_sig(deps, sig_old, p)
fs.interpolate_quadrature(sig_, sig)
```

where `Du` is a solution on a current loading step, the function `eps` defines the mathematical expression of the strain tensor, and the function `proj_sig` performs the return-mapping algorithm.

3.2.3 Newton solvers

To solve a nonlinear equation in the form (13)–(14), the Newton method is traditionally used, where the calculation of the objective function derivative is required. In this paper, three types of implementation of this method are used.

For those cases when we can find the stress derivative analytically, we use our own "naive" Newton method, which is implemented through the simplest loop, as well as using the SNES solver from the petsc library.

In this paper, we are dealing with plasticity problems for which calculating the stress derivative analytically is problematic. That is why we also use the quasi-Newton method, which numerically approximates the derivative of the objective function, which means we do not need to know its explicit expression. This study uses an implementation of this algorithm in the form of SNESQN solver from the petsc library.

We would like to note here that the exact calculation of the derivative significantly affects the convergence of Newton methods, especially at the stages, where plastic deformations occur, therefore, to speed up the calculations, the original problem is solved in a dimensionless form. In

this case, it is enough to change the input parameters, for example, as follows

$$E^* = \frac{E}{E} = 1, \quad \sigma_0^* = \frac{\sigma_0}{E},$$
$$R_i^* = \frac{R_i}{R_i} = 1, \quad R_e^* = \frac{R_e}{R_i},$$

where E^*, σ_0^*, R_i^* and R_e^* are dimensionless Young modulus, uniaxial strength, internal and external radii of the cylinder, respectively.

3.2.4 Classical approach

This section describes the main components of elastic-plastic modelling for solving the (38)–(39) problem using FEniCSx. Only essential parts of the program code will be given here. The full software implementation is made publicly available by (Latyshev, 2022). We recall that this implementation is based on the one of Bleyer (2018), where the von Mises plasticity analysis used the previous version of FEniCS 2019. The current version of FEniCSx 0.3.1.0 contains more features that allow us to improve the previous implementation. We call this approach *classical* to pay tribute to the original implementation : the approach described here uses the same idea of determining the stress field $\underline{\underline{\sigma}}_{n+1}$, the internal variable p_{n+1} and other auxiliary variables in quadrature functional spaces. In this sense, the classical approach does not bring anything new but optimizes the source code using the interpolation feature integrated into the `eval` method of `fem.Expression` entity.

To initialize target variables and function spaces, we should write the following lines of code :

```
deg_u = 2
deg_stress = 2
V = fem.VectorFunctionSpace(mesh, ("CG", deg_u))
We = ufl.VectorElement("Quadrature", mesh.ufl_cell(), degree=deg_stress, dim=4, quad_scheme='default')
W = fem.FunctionSpace(mesh, We)
W0e = ufl.FiniteElement("Quadrature", mesh.ufl_cell(), degree=deg_stress, quad_scheme='default')
W0 = fem.FunctionSpace(mesh, W0e)

u = fem.Function(V, name="Total_displacement")
du = fem.Function(V, name="Iteration_correction")
Du = fem.Function(V, name="Current_increment")
sig = fem.Function(W, name="Current_stress")
N = fem.Function(W)
beta = fem.Function(W0)
p = fem.Function(W0, name="Cumulative_plastic_strain")
dp = fem.Function(W0)
```

It can be seen from this code that we store the values of $\underline{\underline{\sigma}}_{n+1}$ in a vector `sig` ($\boldsymbol{\sigma}_{n+1}$ in Voigt notation). In the future, depending on convenience, we will work with $\underline{\underline{\sigma}}_{n+1}$ in both tensor and Voigt notations.

In this approach, we use our own custom Newton method. It means that we need to define a linearized variational problem (43)–(44) using FEniCSx tools. So we have to represent $\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1}$ as `fem.Expression` using UFL. Although FEniCSx allows us to work with the fourth-rank tensors, in this case, it is more convenient to work with the expression $\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1} : \underline{\underline{\varepsilon}}(\Delta \underline{u})$ and not to store the values of the entire tensor $\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1}$ globally, so we use the function `sigma_tang`

```
def sigma_tang(e):
N_elas = as_3D_tensor(n_elas)
return sigma(e) - 3*mu*(3*mu/(3*mu+H)-beta)*ufl.inner(N_elas, e)*N_elas - 2*mu*beta*ufl.dev(e)
```

to define a weak formulation as follows

```
u_ = ufl.TrialFunction(V)
v_ = ufl.TestFunction(V)
a_Newton = ufl.inner(sigma_tang(eps(v_)), eps(u_))*dx
res = -ufl.inner(eps(v_), as_3D_tensor(sig))*dx + F_ext(v_)
my_problem = pf.LinearProblem(a_Newton, res, Du, bcs)
```

The return-mapping procedure is performed using the `proj_sig` function

```
def proj_sig(deps, old_sig, old_p):
    sig_n = as_3D_tensor(old_sig)
    sig_elas = sig_n + sigma(deps)
    s = ufl.dev(sig_elas)
    sig_eq = ufl.sqrt(3/2.*ufl.inner(s, s))
    f_elas = sig_eq - sig0 - H*old_p
    dp = ppos(f_elas)/(3*mu+H)
    n_elas = s/sig_eq*ppos(f_elas)/f_elas
    beta = 3*mu*dp/sig_eq
    new_sig = sig_elas-beta*s
    return ufl.as_vector([new_sig[0, 0], new_sig[1, 1], new_sig[2, 2], new_sig[0, 1]]), \
        ufl.as_vector([n_elas[0, 0], n_elas[1, 1], n_elas[2, 2], n_elas[0, 1]]), \
        beta, dp
```

It accepts a `fem.Expression` object `deps` as input, as well as global vectors `old_sig` and `old_p` (stress and plastic strain values) from the previous loading step. It returns 4 `fem.Expression` objects, each of which is a combination of mathematical formulas based on the UFL functional and the `du` coefficient. The latter is a global vector of displacements increment on a current Newton iteration.

Using the `interpolate_quadrature` function, we calculate the values of these expressions at each Gaussian node and update the `sig`, `N` and `beta` variables. We have to save the last two globally since they are necessary for the `sigma_tang` function. Thus, we update the variational formulation also.

```
deps = eps(Du)
sig_, N_, beta_, dp_ = proj_sig(deps, sig_old, p)
fs.interpolate_quadrature(sig_, sig)
fs.interpolate_quadrature(N_, N)
fs.interpolate_quadrature(beta_, beta)
```

Before moving on to the convex plasticity approach, we would like to discuss another way to implement the classical one. The fact is that we can differentiate UFL expressions using the `ufl.derivative` function. Suppose we can write down a weak formulation (38)–(39). In that case, we could use external Newton methods (for example, SNES from the `petsc` library) and pass them the derivative $R'(\underline{u})$ instead of calculating it manually by finding the $\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1}$ expression. To do this, we have to rewrite the original variational problem in the following form

Find $\underline{u}_{n+1}$ such that (51)

$$R = \int\limits_\Omega \underline{\underline{\sigma}}_{n+1} : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}\Omega - \underline{F}^{\text{ext}}_{n+1}(\underline{v}) = \int\limits_\Omega \left(\underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : (\Delta\underline{\underline{\varepsilon}} - \Delta\underline{\underline{\varepsilon}}^p)\right) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}\Omega - \underline{F}^{\text{ext}}_{n+1}(\underline{v}) = 0, \tag{52}$$

where $\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}(\underline{u}_{n+1})$, $\underline{u}_{n+1} = \underline{u}_n + \Delta\underline{u}$, $\Delta\underline{\underline{\varepsilon}} = \underline{\underline{\varepsilon}}(\Delta\underline{u})$ and the plastic deformation increment $\Delta\underline{\underline{\varepsilon}}^p = \underline{\underline{\varepsilon}}^p(\Delta\underline{u})$ actually contains a return-mapping algorithm inside. Explicitly it can be written out as follows

$$\Delta\underline{\underline{\varepsilon}}^p = \begin{cases} \Delta p \left(\frac{3}{2}\frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}}\right), & \text{if } f(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p_{n+1}) > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{53}$$

In terms of FEniCSx code should be written as follows :

```
def deps_p(deps, old_sig, old_p):
    sig_n = as_3D_tensor(old_sig)
    sig_elas = sig_n + sigma(deps)
    s = ufl.dev(sig_elas)
    sig_eq = ufl.sqrt(3/2.*ufl.inner(s, s))
    f_elas = sig_eq - sig0 - H*old_p
    dp_sig_eq = ufl.conditional(f_elas > 0, f_elas/(3*mu+H)/sig_eq, 0)
    return 3./2. * dp_sig_eq * s
```

Then we can define a new variational problem and its solver :

```
residual = ufl.inner(as_3D_tensor(sig) + sigma(eps(Du) - deps_p(eps(Du), sig, p)), eps(u_))*dx - F_ext(u_)
J = ufl.derivative(ufl.inner(sigma(eps(Du) - deps_p(eps(Du), sig, p)), eps(u_))*dx, Du, v)
my_problem = pf.SNESProblem(residual, Du, J, bcs, petsc_options=petsc_options_SNES)
```

where `pf.SNESProblem` is a support class to work with the SNES solver.

We note here that, as in the classical approach, this method is possible only in exceptional cases when it is possible to calculate $\Delta\underline{\underline{\varepsilon}}^p$ analytically.

3.2.5 Convex plasticity

We call the *convex plasticity*, or the convex optimization approach, where the return-mapping algorithm is implemented by solving the optimization problem. It is solved at some point in the domain where we know the values of the stress components and cumulative plastic strain. We noted above that the stresses are calculated in Gaussian nodes, so the problem must be solved in each individual node. This approach is not numerically efficient. It is more convenient to solve it simultaneously for several Gaussian nodes. In this case, the order is not important. Therefore, we need to reformulate the problem (49) in vector form.

For this purpose, we will assemble several Gauss nodes into groups or patches. If there are N_q Gauss nodes in total in a functional space of quadratures, then there will be $1 \leq N_{\text{patch}} \leq N_q$ points in one patch. Using this notation, we will rewrite the conic optimization problem in vector (49) form :

$$\begin{cases} \min\limits_{\boldsymbol{\Sigma}, \boldsymbol{P}} F(\boldsymbol{\Sigma}, \boldsymbol{P}), \\ \boldsymbol{f}(\boldsymbol{\Sigma}, \boldsymbol{P}) \leq 0, \end{cases} \tag{54}$$

where the vector $\boldsymbol{\Sigma}$ with the size $4 * N_{\text{patch}}$ contains sequentially 4 components of the stress vector

$\boldsymbol{\sigma}$ for N_{patch} Gauss nodes. By analogy, vectors $\boldsymbol{P}$ and $\boldsymbol{f}$ contain cumulative plastic strain p and yield criterion f values, respectively, for N_{patch} Gauss nodes. Then, keeping the previous notation, the free energy F will be written in the following form

$$F(\boldsymbol{\Sigma}, \boldsymbol{P}) = \frac{1}{2}(\boldsymbol{\Sigma}^{\text{elas}}_{n+1} - \boldsymbol{\Sigma})^T \mathbb{S} (\boldsymbol{\Sigma}^{\text{elas}}_{n+1} - \boldsymbol{\Sigma}) + \frac{1}{2} H (\boldsymbol{P}^{\text{elas}}_{n+1} - \boldsymbol{P})^T (\boldsymbol{P}^{\text{elas}}_{n+1} - \boldsymbol{P}), \tag{55}$$

where the matrix $\mathbb{S}$ with the size $4 * N_{\text{patch}} \times 4 * N_{\text{patch}}$ is the block-diagonal matrix with the inverted stiffness matrix $\boldsymbol{S}$ on its diagonal.

In order to solve this problem numerically, we use the `cvxpy` library. It allows using various third-party conic solvers. For instance, SCS, MOSEK and ECOS were used in this work. For convenience, the `ReturnMapping` class is written, which formulates the problem using the `cvxpy` library and finds its numerical solution.

Note that the free energy F does not depend on the plasticity criterion. In order to change the plasticity model, it is enough to change the constraint of the optimization problem (54). This makes the convex plasticity approach more general compared to the classical one. In this regard, it is impossible to know in advance the stress derivative, which is necessary for the Newton method, so we use the quasi-Newton one here. As a result, the algorithm of the convex plasticity approach consists of the following steps :

— We consider the problem in the weak formulation (38)–(39).
— We use the SNESQN `petsc` to solve it.
— During each quasi-Newton iteration, we carry out the projection of $\boldsymbol{\sigma}^{\text{elas}}_{n+1}$ on the yield surface, solving the vectorized optimization problem (54).

Note that the optimization problem is solved $\lfloor N_q / N_{\text{patch}} \rfloor$ times if this division contains a nonzero remainder, then the conic problem (54) is solved additionally for the remaining $N_q \bmod N_{\text{patch}}$ Gauss nodes.

3.3 Performance

The second part of this paper discusses strategies to improve the performance of the described approaches to solving plasticity problems. Before describing them in detail, we will first talk about the obstacles, that we encountered during the development of the methods presented in the first part of this research.

Let us return to a general nonlinear variational problem (13)–(14). Obviously, the case of modelling an elastic-plastic material defined by the von Mises yield criterion is just an example

where UFL features are sufficient to solve the problem. In other models, where the stress tensor $\underline{\underline{\sigma}}$ is difficult to represent explicitly. Let us show you some other examples

- $\underline{\underline{\sigma}}(\underline{u}) = f(\varepsilon_{\mathrm{I}}, \varepsilon_{\mathrm{II}}, \varepsilon_{\mathrm{III}})$,
- $\underline{\underline{\sigma}}(\underline{u}) = \underset{\alpha}{\operatorname{argmin}}\, g(\underline{\underline{\varepsilon}}(\underline{u}), \alpha)$,

where $\varepsilon_{\mathrm{I}}, \varepsilon_{\mathrm{II}}, \varepsilon_{\mathrm{III}}$ are eigenvalues of $\underline{\underline{\varepsilon}}$ and g is some scalar function.

As seen in the second example, $\underline{\underline{\sigma}}(\underline{u})$ can also implicitly depend on the value of other scalars, vectors or even tensorial quantities (α here). The latter does not necessarily need to be represented in a finite-element function space. They shall just be computed during the assembling procedure when evaluating the expression $\underline{\underline{\sigma}}(\underline{u})$ pointwise.

Although we use the quasi-Newton method to find the solution of (38)–(39), which allows us to bypass the exact calculation of the derivative of $\underline{\underline{\sigma}}$ with respect to $\underline{u}$, we would like to have a more reliable way to solve the problem in the case when we know the expression of the derivative (as, for example, it is so for von Mises and Drucker-Prager plasticity models). However, it is difficult to express it using UFL tools. Otherwise, we have a method that is able to approximate the derivative more accurately (as, for example, the `derivative()` method of the cvxpy package does). Thus, it is necessary to have a functionality that will allow to do a simultaneous calculation of $\underline{\underline{\sigma}}$ and its derivative during the assembling procedure regardless of the calculation method.

In addition to the inconveniences described above, we are faced with the problem of excessive memory allocation. For example, in the classic approach, we are forced to save global variables β_{n+1} and $\boldsymbol{N}$. Suppose there is a mesh containing N_{elements} elements. In that case, we have to allocate memory for a vector with the size $N_{\text{elements}} * 3$ for the scalar field β_{n+1} from `W0` finite space and another one with the size $N_{\text{elements}} * 3 * 4$ for the vector field $\boldsymbol{N}$ from `W` finite space. At the same time, they are intermediate variables necessary only for calculating the stress tensor and its derivative. So we would like to avoid such a waste of memory.

As the result, we require the following features to increase the capabilities of the code implemented in the first part of this project :

1. To define nonlinear expressions of variational problems in a more complex way than via UFL
2. To let an "oracle" provide values of such an expression and its derivative(s)
3. To call this oracle on-the-fly during the assembly to avoid unnecessary loops, precomputations and memory allocations

The following sections describe our own view of these features' implementation.

3.3.1 Custom assembling

Following the concept of the custom assembler (DOLFINx, 2022), which uses the power of `numba` and `cffi` Python libraries, we implemented our own version of custom assembler, where we can change the main loop of the assembling procedure.

There are several essential elements of the algorithm to be mentioned. First of all, we would like to introduce a concept of `CustomExpression`, which is essential for our study. Let us consider the next simple variational problem

$$\int_\Omega g\underline{u} \cdot \underline{v} dx, \quad \forall \underline{v} \in V,$$

where $\underline{u}$ is a trial function, $\underline{v}$ is a test function and the function g is a mathematical expression. For this moment, we must use `fem.Function` class to implement this variational form. Knowing the exact UFL expression of g we can calculate its values on every element using the interpolation procedure of `fem.Expression` class. So we save all values of g in one global vector. The goal is to have the possibility to calculate the expression g, no matter how difficult it is, in every element node (for instance, in every Gauss node if we define g on a quadrature element) during the assembling procedure. So, we introduce a new entity named as `CustomExpression`. It

1. inherits `fem.Function`
2. contains a method `eval`, which will be called inside of the assemble loop and calculates the function local values

Besides `CustomExpression`, we need another entity. Every `fem.Function` object stores its values globally, but we would like to avoid such a waste of memory updating the function value during the assembling procedure. Let us consider the previous variational form, where g contains its local-element values. If there is one local value of g (only 1 Gauss node), the use of the `fem.Constant` entity would be enough, but it is a particular case. We generally deal with quadrature functional spaces of arbitrary degree (for example, `W0` or Q2 space from the first part). We need to store different local values of g in every Gauss node. So we introduce a concept of `DummyFunction` (or 'ElementaryFunction' ?), which

1. inherits `fem.Function`
2. allocates the memory for local values only
3. can be updated during the assembling procedure

3.3.2 Examples

We implemented an elasticity problem to explain our ideas with a simple example. Let us consider a beam stretching with the left side fixed. On the other side, we apply displacements.

$$\text{Find } \underline{u} \in V \text{ s.t.} \tag{56}$$

$$\int_\Omega \underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}(\underline{u})) : \underline{\underline{\varepsilon}}(\underline{v}) d\underline{x} = 0 \quad \forall \underline{v} \in V, \tag{57}$$

with the following boundary conditions

$$(0,0) : u_y = 0, \tag{58}$$

$$\partial\Omega_{\text{left}} : u_x = 0, \tag{59}$$

$$\partial\Omega_{\text{right}} : u_x = t \cdot u_{\text{bc}}, \tag{60}$$

where u_{bc} is a maximal displacement on the right side of the beam, t is a parameter varying from 0 to 1, and where $\underline{\underline{\sigma}}(\underline{\underline{\varepsilon}})$ is our user-defined "oracle". Here we use a simple elastic behaviour :

$$\underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}) = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}, \tag{61}$$

and for which the derivative is :

$$\frac{d\underline{\underline{\sigma}}}{d\underline{\underline{\varepsilon}}} = \underline{\underline{\underline{\underline{C}}}}. \tag{62}$$

Let us focus on the key points. In this "naive" example, the derivative is constant, but in general nonlinear models, its value will directly depend on the local value of $\underline{\underline{\varepsilon}}$. We want to change this value at every assembly step and avoid additional memory allocation. As it is the stress tensor derivative, it is defined on quadrature elements. Thus, in our terms, it would be a `DummyFunction`. Obviously, $\underline{\underline{\sigma}}$ is the `CustomExpression`, which depends on $\underline{\underline{\varepsilon}}$.

The code below initiates variables associated with $\underline{\underline{\underline{\underline{C}}}}$ and $\underline{\underline{\sigma}}$ respectively :

```
q_dsigma = ca.DummyFunction(VQT, name='stiffness') # tensor C
q_sigma = ca.CustomExpression(VQV, eps(u), [q_dsigma], get_eval)  # sigma_{n+1}
```

In the `CustomExpression` constructor we observe three arguments. The first one is the UFL-expression of its variable ($\underline{\underline{\varepsilon}}$ here). It will be compiled via `ffcx` and will be sent as a "tabulated"

expression to a numba (compilable) function, which performs the calculation of `q_sigma`. The second argument is a list of `q_sigma` coefficients (`fem.Function` or `DummyFunction`), which takes part in calculations of `q_sigma`. The third argument contains the function `get_eval` generating a `CustomExpression` method `eval`, which will be called during the assembling. It describes every step of the local calculation of $\underline{\underline{\sigma}}$. In this linear elasticity example, it simply multiplies the stiffness tensor $\boldsymbol{C}$ and the strain vector $\boldsymbol{\varepsilon}$:

```
def get_eval(self:ca.CustomFunction):
    tabulated_eps = self.tabulated_input_expression
    n_gauss_points = len(self.input_expression.X)
    local_shape = self.local_shape
    C_shape = self.stiffness.shape

    @numba.njit(fastmath=True)
    def eval(sigma_current_local, coeffs_values, constants_values, coordinates, local_index, orientation):
        epsilon_local = np.zeros(n_gauss_points*3, dtype=PETSc.ScalarType)

        C_local = np.zeros((n_gauss_points, *C_shape), dtype=PETSc.ScalarType)

        sigma_local = sigma_current_local.reshape((n_gauss_points, *local_shape))

        tabulated_eps(ca.ffi.from_buffer(epsilon_local),
                    ca.ffi.from_buffer(coeffs_values),
                    ca.ffi.from_buffer(constants_values),
                    ca.ffi.from_buffer(coordinates), ca.ffi.from_buffer(local_index), ca.ffi.from_buffer(orientation))

        epsilon_local = epsilon_local.reshape((n_gauss_points, -1))

        for q in range(n_gauss_points):
            C_local[q][:] = get_C() #change DummyFunction here
            sigma_local[q][:] = np.dot(C_local[q], epsilon_local[q])

        sigma_current_local[:] = sigma_local.flatten()


        return [C_local.flatten()]
    return eval
```

Besides the local implementation of new entities, we need to change the assembling procedure loop to explicitly describe the interaction between different coefficients of linear and bilinear forms. It allows us to write a general custom assembler, which will work for any nonlinear problem. Thus we have to define two additional numba functions to calculate local values of forms kernels coefficients (see the code below).

```python
@numba.njit(fastmath=True)
def local_assembling_b(cell, coeffs_values_global_b, coeffs_coeff_values_b, coeffs_dummy_values_b, coeffs_eval_b,
                       u_local, coeffs_constants_b, geometry, entity_local_index, perm):
    sigma_local = coeffs_values_global_b[0][cell]

    output_values = coeffs_eval_b[0](sigma_local,
                                     u_local,
                                     coeffs_constants_b[0],
                                     geometry, entity_local_index, perm)

    coeffs_b = sigma_local

    for i in range(len(coeffs_dummy_values_b)):
        coeffs_dummy_values_b[i][:] = output_values[i] #C update

    return coeffs_b

@numba.njit(fastmath=True)
def local_assembling_A(coeffs_dummy_values_b):
    coeffs_A = coeffs_dummy_values_b[0]
    return coeffs_A
```

With this "naive" example, we demonstrated the possibility of intervention in the assembly process with ready-made FEniCSx tools and with the help of compiled functions of the numba library. The implementation of the proposed functionality is effective enough not to change the source code of the FEniCSx library itself.

The following example optimizes the classical approach to solving the von Mises plasticity problem : (38)–(39). We can conclude from this, that the fields $\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}_{n+1}(\Delta\underline{\underline{\varepsilon}}, \beta_{n+1}, \underline{\underline{N}}_{n+1}, dp, p_n, \underline{\underline{\sigma}}_n)$ and $\mathbf{C}^{\text{tang}}_{n+1} = \mathbf{C}^{\text{tang}}_{n+1}(\beta_{n+1}, \underline{\underline{N}}_{n+1})$ depend on the common variables β_{n+1} and $\underline{\underline{N}}_{n+1}$. With the previous implementation, it was necessary to allocate additional space for them and calculate $\underline{\underline{\sigma}}_{n+1}$ and $\mathbf{C}^{\text{tang}}_{n+1}$ separately, but now we can combine their local evaluations.

Compared to the elasticity case, the `CustomExpression sig` has more dependent fields. Look at the code below.

```python
C_tang = ca.DummyFunction(QT, name='tangent') # tensor C_tang
sig = ca.CustomExpression(W, eps(Du), [C_tang, p, dp, sig_old], get_eval) # sigma_n
```

As it was expected, the local evaluation of the `CustomExpression` becomes more complex

```
def get_eval(self:ca.CustomExpression):
    tabulated_eps = self.tabulated_input_expression
    n_gauss_points = len(self.input_expression.X)
    local_shape = self.local_shape
    C_tang_shape = self.tangent.shape

    @numba.njit(fastmath=True)
    def eval(sigma_current_local, sigma_old_local, p_old_local, dp_local,
             coeffs_values, constants_values, coordinates, local_index, orientation):
        deps_local = np.zeros(n_gauss_points*3*3, dtype=PETSc.ScalarType)

        C_tang_local = np.zeros((n_gauss_points, *C_tang_shape), dtype=PETSc.ScalarType)

        sigma_old = sigma_old_local.reshape((n_gauss_points, *local_shape))
        sigma_new = sigma_current_local.reshape((n_gauss_points, *local_shape))

        tabulated_eps(ca.ffi.from_buffer(deps_local),
                      ca.ffi.from_buffer(coeffs_values),
                      ca.ffi.from_buffer(constants_values),
                      ca.ffi.from_buffer(coordinates),
                      ca.ffi.from_buffer(local_index),
                      ca.ffi.from_buffer(orientation))

        deps_local = deps_local.reshape((n_gauss_points, 3, 3))

        n_elas = np.zeros((3, 3), dtype=PETSc.ScalarType)
        beta = np.zeros(1, dtype=PETSc.ScalarType)
        dp = np.zeros(1, dtype=PETSc.ScalarType)

        for q in range(n_gauss_points):
            sig_n = as_3D_array(sigma_old[q])
            sig_elas = sig_n + sigma(deps_local[q])
            s = sig_elas - np.trace(sig_elas)*I/3.
            sig_eq = np.sqrt(3./2. * inner(s, s))
            f_elas = sig_eq - sig0 - H*p_old_local[q]
            f_elas_plus = ppos(f_elas)
            dp[:] = f_elas_plus/(3*mu_+H)

            sig_eq += TPV # for the case when sig_eq is equal to 0.0
            n_elas[:,:] = s/sig_eq*f_elas_plus/f_elas
            beta[:] = 3*mu_*dp/sig_eq

            new_sig = sig_elas - beta*s
            sigma_new[q][:] = np.asarray([new_sig[0, 0], new_sig[1, 1], new_sig[2, 2], new_sig[0, 1]])
            dp_local[q] = dp[0]

            C_tang_local[q][:] = get_C_tang(beta, n_elas)

        return [C_tang_local.flatten()]
    return eval
```

Thus it can be seen more clearly the dependence of the tensor $\mathbf{C}_{n+1}^{\text{tang}}$ on the calculation of the tensor $\underline{\underline{\sigma}}_{n+1}$. The complete source code of these examples can be found in the `assembling_strategies` folder of the repository (Latyshev, 2022).

Finally, we developed our own custom assembler using two new entities. This allows us to save memory, avoid unnecessary global *a priori* evaluations and do instead on-the-fly evaluations during the assembly. More importantly, this allows to deal with more complex mathematical expressions, which can be implicitly defined, where the UFL functionality is quite limited. Thanks to ‘numba‘ and ‘cffi‘ Python libraries and some FEniCSx features, we can implement our ideas through efficient code.

4 Results

4.1 Qualitative yield criteria comparison

Here we compare various elastic-plastic models, plotting the displacements of the inner surface of the cylinder ($u_x(R_i, 0)$) at the end of the loading process. The final values of the movements are shown on the image 3. These results are obtained thanks to the convex plasticity approach, based on solving the optimization problem and using the FEniCSx and cvxpy libraries.

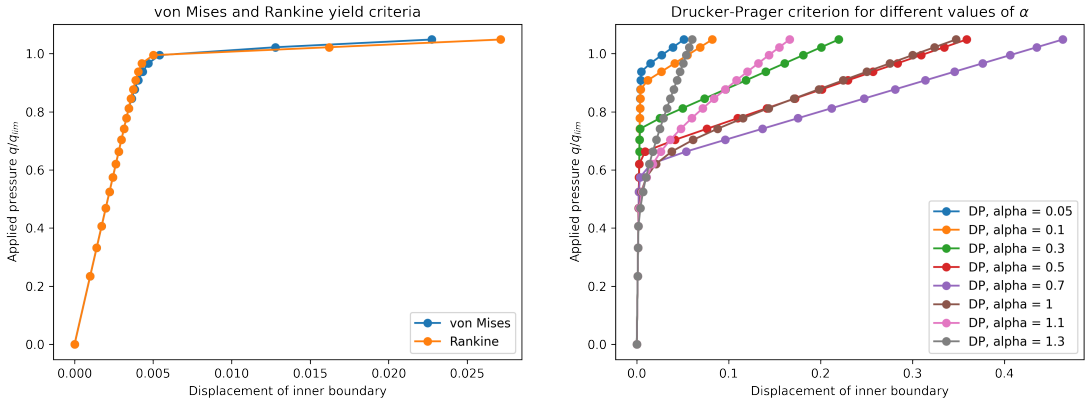


FIGURE 3 – Displacements of the inner boundary for different yield criteria using convex plasticity approach

As we can see, the proposed approach gives correct results for all the criteria considered in this paper and various values of their parameters (for the Drucker-Prager criterion, we vary the parameter α). This approach makes it possible to simulate the elastic-plastic behaviour of the material, including that determined by the Rankine criterion. We recall that it is difficult to implement the classical approach in this case, especially in the three-dimensional one.

4.2 Patch size effect on vectorized convex optimization problem

This work aims to test the implementation of the convex plasticity approach in the form of ready-made program code and its effectiveness. If this implementation solves the task significantly slower than other approaches, it will have insufficient benefits, especially if we are discussing complex three-dimensional problems. Therefore, we need to analyze the impact of solving the optimization problem using the cvxpy library on the overall simulation results.

For numerical tests, the initial problem with the von Mises criterion was chosen as the target one, as well as three finite element meshes of different densities, the sizes of which are presented in the table below :

Mesh	Nodes	Elements	Gauss nodes of Q2 space
Coarse	50	69	207
Medium	407	714	2142
Dense	811	1478	4434

TABLE 1 – Mesh data of time performance numerical tests for patch size effect

where Q2 space means the size of the finite element functional space of the second degree based on quadrature elements.

The images 4–6 demonstrate a comparison of the total program time for three meshes of different densities, using three different conic solvers depending on the patch size N_{patch}. The compilation time of the optimization problem was also measured. The fact is that before solving a problem formulated in the form of (54) cvxpy converts it into a canonical form, with which most conic solvers work. For convenience, the plots from these images are duplicated on a logarithmic scale.

As seen from the plots, the larger the patch size, the longer it takes to compile using cvxpy. Starting from some moment, the compilation time is comparable to the total one of solving the conic problem. At the same time, with large values of N_{patch}, the compilation process consumes significantly RAM. That is why, for denser meshes, it is not possible to solve the problem for patch sizes equal to the total number of Gauss nodes in quadrature space defined on these meshes (components of the stress vector $\boldsymbol{\sigma}$ and cumulative plastic strain p are defined in such spaces). Despite this, we can state that the time to solve the optimization problem decreases significantly with an increase in the patch size. Starting from some N_{patch}, this decrease does not considerably affect the modelling time.

In addition, comparing conic solvers, we see that the MOSEK solver works better than others with high-dimensional problems. However, at the same time, it is much worse with small-dimensional ones. SCS is comparable in time to MOSEK. The ECOS solver is slower to solve high-dimensional problems.

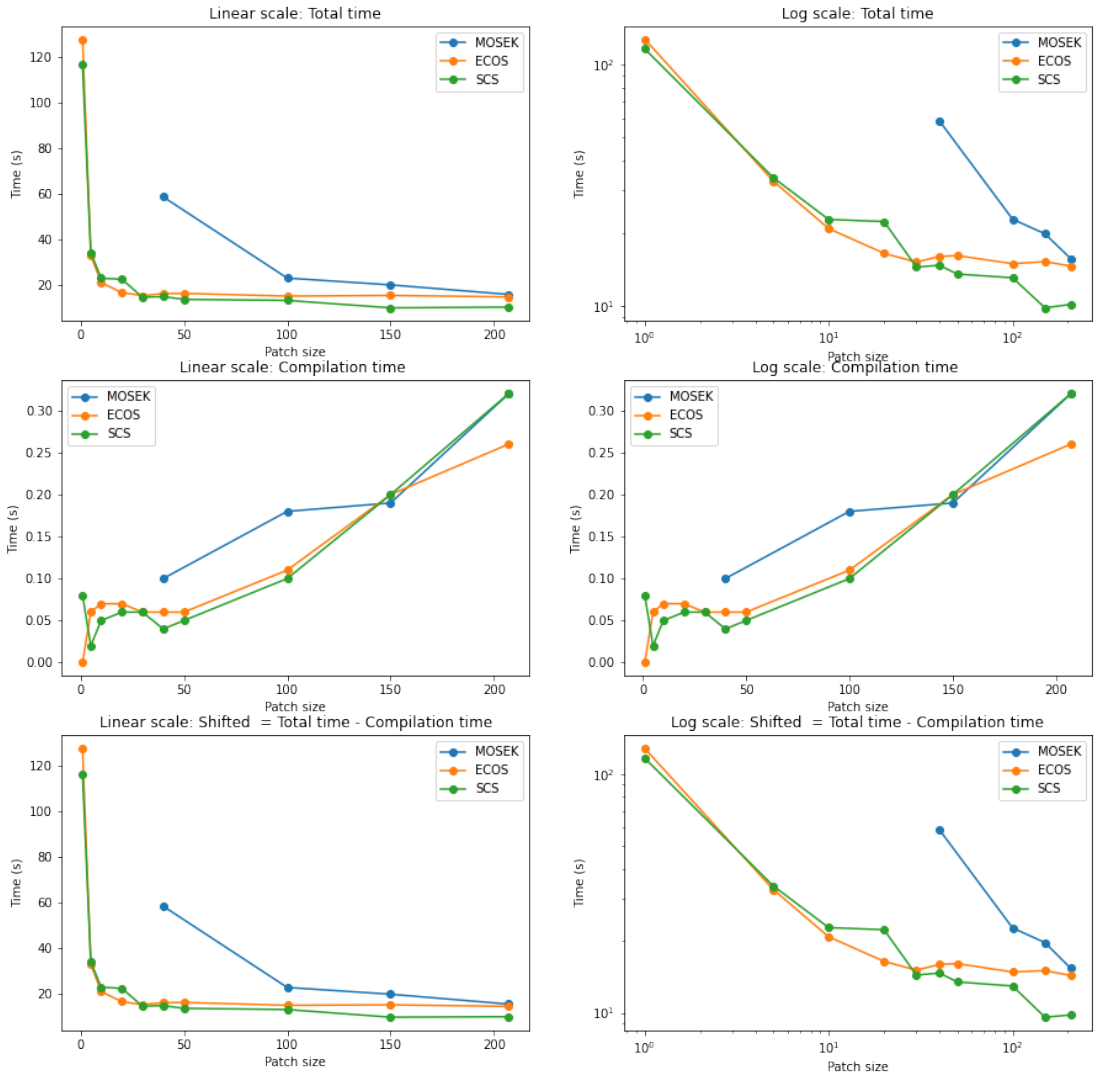


FIGURE 4 – Patch size effect on the time performance of solving the conic problem : coarse mesh case

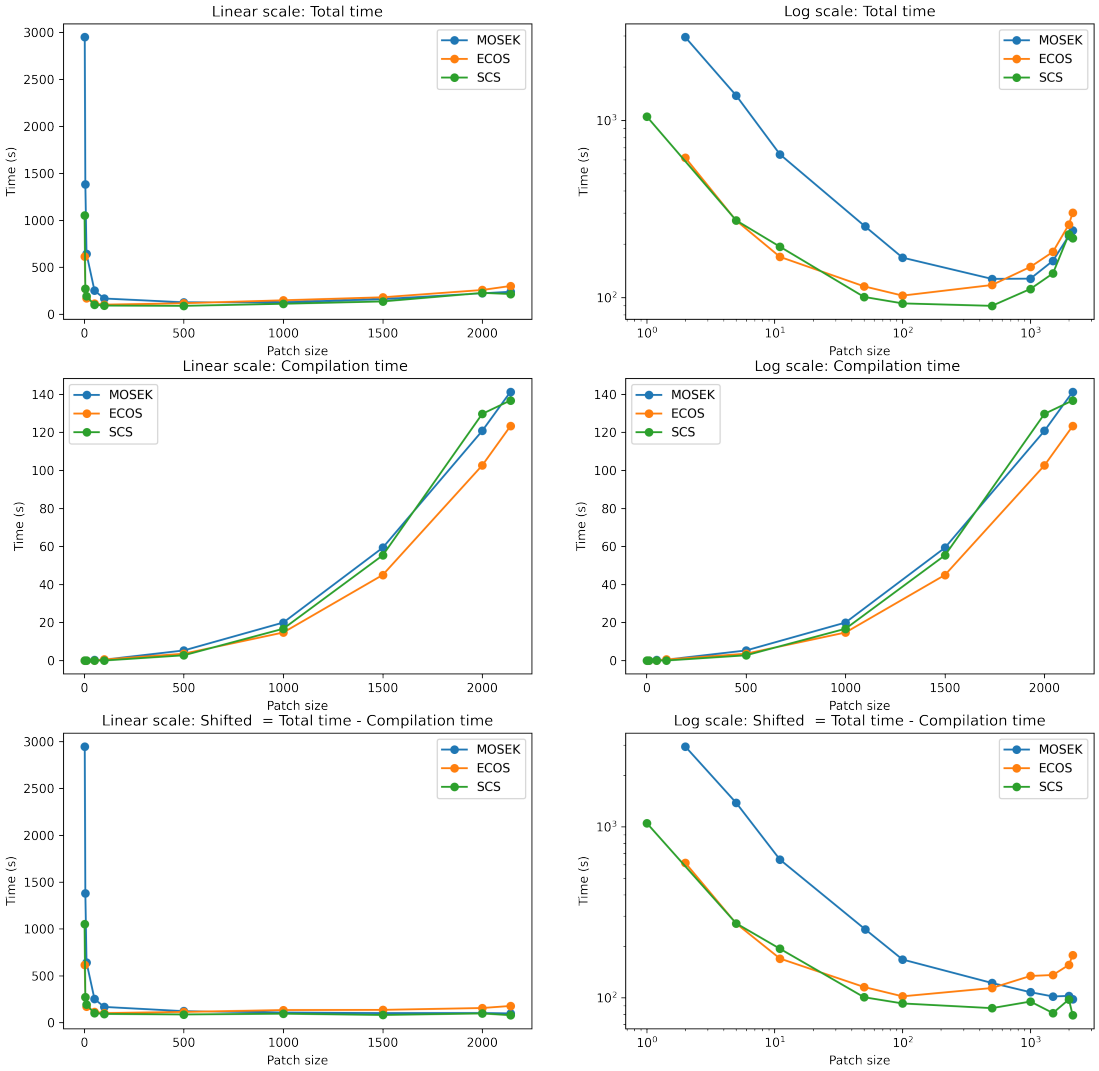


FIGURE 5 – Patch size effect on the time performance of solving the conic problem : medium mesh case

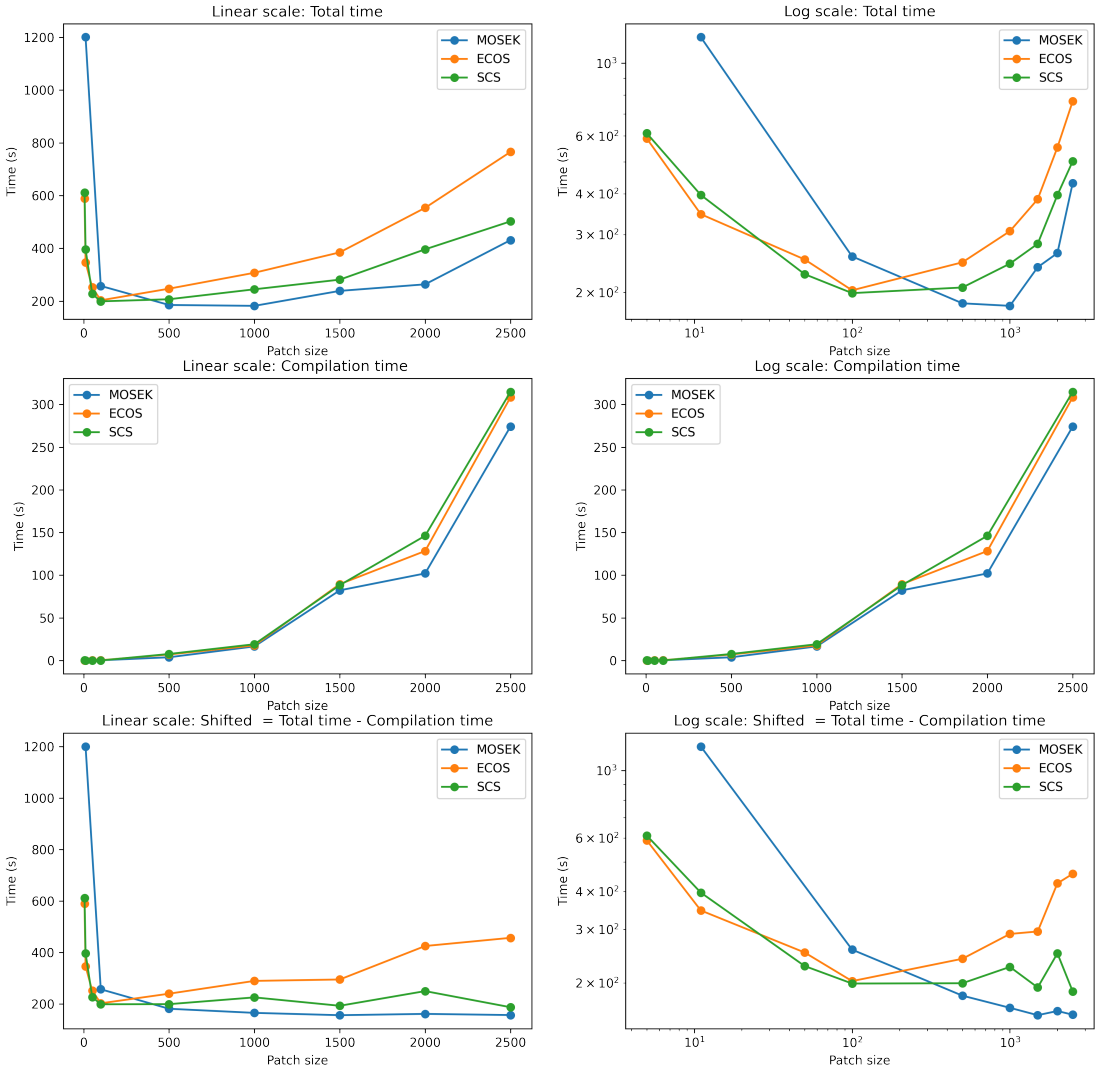


FIGURE 6 – Patch size effect on the time performance of solving the conic problem : dense mesh case

Summarizing the numerical tests described above, we can conclude the following :

— The larger the patch size, i.e. the larger the dimension of the vectorized conic optimization problem, the faster the program runs.

— Starting with a certain patch size, the program is not significantly accelerated.

— For large patch sizes, cvxpy expends substantial computer resources to compile the

optimization problem.
— For very dense meshes, it will not be possible to speed up the program by increasing the patch size.
— MOSEK works faster with problems of high dimension but slower with small ones.
— ECOS is less suitable for high-dimensional problems.
— SCS is an optimal conic solver for problems of any dimension.

4.3 Custom assembling performance

In this section, we analyze the effectiveness of the implementation of the custom assembly approach regarding the classical one depending on the mesh density. Using the von Mises criterion, a series of numerical tests was performed based on the original cylinder expansion problem. Each test corresponds to a finite element mesh. The input data is presented in the table 2.

Mesh	Nodes	Elements	Gauss nodes of Q2 space
1	50	69	207
2	811	1478	4434
3	3706	7095	67707
4	11567	22569	67707
5	31666	62392	187176

TABLE 2 – Mesh data of time performance numerical tests of the custom assembling approach

The figure 7 shows the dependence of the total running time of the program on different meshes (as the mesh number increases, its density increases as well). Using 'JIT overhead', we indicated the compilation time of Python code written using the numba package.

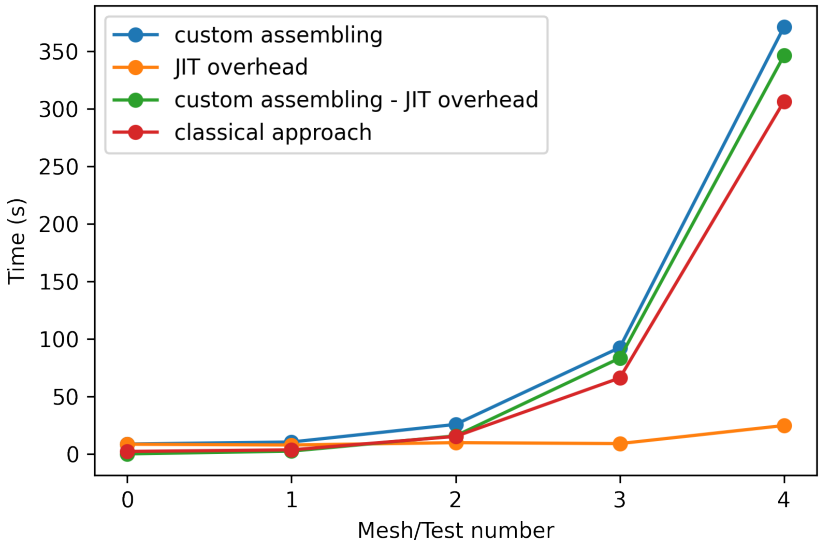


FIGURE 7 – Time performance of the custom assembling approach in comparison with the classical one

The results of the numerical experiment show that the compilation time of JIT overhead is an essential part of the total running time of the program for low-density meshes. At the same time, as the mesh density increases, the ratio of the compilation time to the total running time tends to zero. This is possible because the compilation of numba code occurs once during the first iteration of the Newton method. After that, the program uses the already compiled code throughout the entire modelling process.

From this, the compilation time is negligible compared with the total running time for complex nonlinear problems, as well as those where the use of a mesh with a large number of elements is required. At the same time, this approach is only slightly inferior in time to the classical one.

4.4 Newton and quasi-Newton methods performance

Solving the problem in the formulation (51)–(52), we can compare the performance of Newton (SNES) and quasi-Newton (SNESQN) methods. In this section, the results of numerical experiments are presented. The number of iterations required for convergence of both methods, depending on mesh density, and the total running time were recorded. This number corresponds to the last loading step when plastic deformations are maximal. The data of finite element meshes

are identical to those taken from section 4.3 and presented in the table 2.

The image 8 shows plots of the dependence of the iterations number required for convergence of two methods on a test number. In addition, plots of the total running time for each method are shown, depending on the mesh.

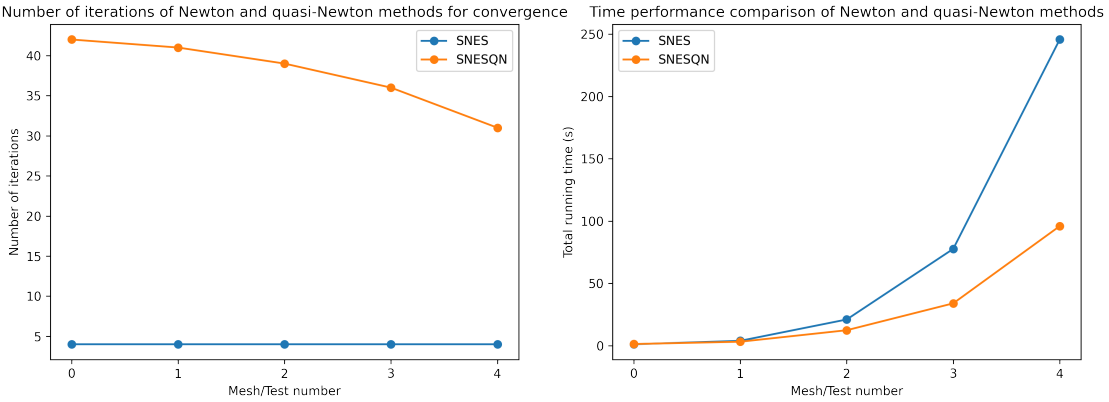


FIGURE 8 – Performance comparison of the standard Newton method (SNES) and the quasi-Newton one (SNESQN)

We observe interesting results. On the one hand, as expected, the quasi-Newton method requires more iterations for convergence. At the same time, this number decreases with increasing mesh density. This can be explained by the fact that the denser the mesh, the more accurate the values of such quantities as stresses we get. Consequently, the stress derivative is calculated more accurately, on which the convergence of the method depends. At the same time, as expected, the convergence of the standard Newton method does not depend on the mesh density, and it requires a much smaller number of iterations for convergence.

On the other hand, we note that a program based on the quasi-Newton method works faster than its SNES counterpart, despite SNESQN spending more iterations for convergence. We remind that in the formulation used here, the `ufl.derivative` function is used to take the derivative through UFL. Therefore as the number of mesh elements increases, the `ufl.derivative` function, which SNES uses, works slower than the derivative approximation through SNESQN.

5 Discussion

We wrote the first working version of the program in Python, solving plasticity problems using convex optimization for the FEniCSx Project. The work results demonstrate that this approach applies to different plasticity models. Moreover, we used modern Python libraries to write just-in-time compilation code, simplifying the development process while maintaining a performance comparable to the source code written in C\C++. Thus, we show that it is possible to solve plasticity problems efficiently, quickly and for different yield criteria, even for non-smooth ones, using modern open-source libraries.

At the same time, the work contains ready-made elements indicating the prospects for improving time performance. Custom assembly demonstrates the potential to influence the assembly process locally, on each element. Convex plasticity approach uses an external cvxpy library to implement the return-mapping procedure, which we can also perform locally. Recall that we are limited by basic objects (such as arrays of the NumPy library) to write with JIT compilable numba functions. Suppose we could wrap the required functionality of the cvxpy library and call it from inside compiled functions. In that case, we could offer a more efficient implementation of the Convex plasticity approach. This is possible. Not so long ago, the cvxpygen (Schaller et al., 2022) library appeared, which wraps written Python code into C language code automatically for various optimization tasks. In theory, this is enough to implement the idea described above. The only question is how much the resulting version of the program will be more effective compared to its original version. We note here that in place of the cvxpy library, any other functionality may be necessary to implement a new feature as a part of the FEniCSx code. Thus, the custom assembly can significantly expand the capabilities of the FEniCSx Python code.

Custom assembling can also be helpful when we have to calculate derivatives. As noted several times in this work, we actively use the Newton method, where the exact calculation of the derivative of the objective function is a sensitive place. The cvxpy library allows us to calculate the derivative of target variables (for example, from $\underline{\sigma}$) using the `derivate()` function. This functionality would allow us to abandon the less stable quasi-Newton method in favour of the ordinary one. Unfortunately, it takes much time. In this situation, wrapping the `derivate()` function in a JIT-compiled function could significantly speed up calculations. By doing this, we show that if there is any external method for calculating the derivative, custom assembling could accelerate Newton's method.

Also, from the work results, it can be concluded that using the convex plasticity approach

can become problematic for finite element meshes with a large number of elements. As it was shown, compilation of vectorized optimization problems through cvxpy can take much time, so much so that it will be comparable to the running time of the entire program. In addition, compilation consumes much RAM in this case. This is directly affected by the type of problem formulated and the compilation method. For example, going back to the cvxpygen library, the optimization problem can be compiled in C, potentially reducing compilation time and the amount of memory consumed. As a result, this obstacle requires independent research.

Ultimately, this work is distributed in the public domain. This allows other researchers to use the ideas proposed here about custom assembling and the convex plasticity approach in the finite element environment FEniCSx. We hope our work will inspire others to develop new, more accurate and faster numerical methods that solve plasticity problems (and more).

Conclusion

As a result of the work done, the framework was developed in Python, using the libraries FEniCSx, cvxpy, numba and others. It allows modelling the behaviour of solids taking into account elastic-plastic effects on the example of the cylinder expansion problem. In this framework, a common approach to modelling plasticity through the return-mapping algorithm and conic optimization can be found. The latter works on a wide range of yield criteria when the classical approach is adapted to the criteria of von Mises and Drucker-Prager. In addition to the last two, the Rankine criterion is also considered in the work. The work was performed for a particular case of plane strain. However, its results can be generalized to three-dimensional plasticity problems taking into account full-isotropic hardening law, as well as other plasticity criteria.

The simulation results were carried out for different conic solvers, and the effect of the size of the vectorized conic optimization problem was analyzed. The topic of solving plasticity using conic solvers is not limited to these results. The work can be supplemented with additional research.

In the framework of this work, the concept of custom assembling was developed to change the assembly process of the FEniCSx library. A prototype was written to work with it using examples of elastic and plastic problems. Thanks to this approach, the research has great potential to continue, especially in the context of improving time performance. For example, wrapping the code responsible for solving the conic optimization problem into a JIT-compiled function could potentially reduce the running time.

The framework code is publicly available : (Latyshev, 2022).

Bibliography

Agrawal, A., Verschueren, R., Diamond, S. & Boyd, S. (2018). A rewriting system for convex optimization problems. *Journal of Control and Decision*, *5*(1), 42-60.

Alnæs, M. S., Logg, A., Ølgaard, K. B., Rognes, M. E. & Wells, G. N. (2014). Unified Form Language : A Domain-Specific Language for Weak Formulations of Partial Differential Equations. *ACM Trans. Math. Softw.*, *40*(2).

Alnaes, M. S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M. E. & Wells, G. N. (2015). The FEniCS Project Version 1.5. *Archive of Numerical Software*, *3*.

ApS, M. (2019). *MOSEK Fusion API for Python 9.3.21.* https://docs.mosek.com/9.3/pythonfusion/index.html

Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., ... Zhang, J. (2022). *PETSc/TAO Users Manual* (rapp. tech. ANL-21/39 - Revision 3.18). Argonne National Laboratory.

Bleyer, J. (2018). *Numerical Tours of Computational Mechanics with FEniCS*. Zenodo. https://doi.org/10.5281/zenodo.1287832

Bonnet, M., Frangi, A. & Rey, C. (2014). *The finite element method in solid mechanics*. McGraw Hill Education.

Borst, R., Crisfield, M., Remmers, J. & Verhoosel, C. (2012). *Non-Linear Finite Element Analysis of Solids and Structures : Second Edition*. John Wiley & Sons, Ltd.

Bruno, H., Barros, G., Menezes, I. F. & Martha, L. F. (2020). Return-mapping algorithms for associative isotropic hardening plasticity using conic optimization. *Applied Mathematical Modelling*, *78*, 724-748.

Diamond, S. & Boyd, S. (2016). CVXPY : A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, *17*(83), 1-5.

DOLFINx. (2022). *Custom assembler test* [Accessed : 2022-10-19]. https://github.com/FEniCS/dolfinx/blob/main/python/test/unit/fem/test_custom_assembler.py

Domahidi, A., Chu, E. & Boyd, S. (2013). ECOS : An SOCP solver for embedded systems, In *European Control Conference (ECC)*.

Lam, S. K., Pitrou, A. & Seibert, S. (2015). Numba : A LLVM-Based Python JIT Compiler, In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, Austin, Texas, Association for Computing Machinery.

Latyshev, A. (2022). *Convex-plasticity* [Accessed : 2022-10-19]. https://github.com/andrash/convex-plasticity

Logg, A., Mardal, K.-A. & Wells, G. N. (2012). *Automated Solution of Differential Equations by the Finite Element Method*. Springer.

Logg, A. & Wells, G. N. (2010). DOLFIN : Automated Finite Element Computing. *ACM Transactions on Mathematical Software*, *37*.

Logg, A., Wells, G. N. & Mardal, K. (2012). DOLFIN : a C++/Python Finite Element Library. In *Automated Solution of Differential Equations by the Finite Element Method*. Springer.

O'Donoghue, B., Chu, E., Parikh, N. & Boyd, S. (2016). Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications*, *169*(3), 1042-1068.

Schaller, M., Banjac, G., Diamond, S., Agrawal, A., Stellato, B. & Boyd, S. (2022). Embedded Code Generation With CVXPY. *IEEE Control Systems Letters*, *6*, 2653-2658.

Scroggs, M. W., Baratta, I. A., Richardson, C. N. & Wells, G. N. (2022). Basix : a runtime finite element basis evaluation library. *Journal of Open Source Software*, *7*(73), 3982.

Appendices

A Drucker-Prager material

In this part, we analytically derive such expressions as $\underline{\underline{\sigma}}_{n+1}, \Delta p, \Delta\underline{\underline{\sigma}}, \underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1}$, etc. for the Drucker-Prager criterion, which simplify a numerical solution of plasticity problems. The conclusion is similar to the corresponding formulas from section **??** for the von Mises criterion, repeating the reasoning from (Bonnet et al., 2014).

We duplicate the Drucker-Prager yield criterion below

$$f(\underline{\underline{\sigma}}, p) = \sigma_{\text{eq}} + \alpha \text{tr}\underline{\underline{\sigma}} - R(p) \leq 0, \tag{63}$$

where the expression $R(p) = \sigma_0 + hp$ represents a linear isotropic hardening.

We recall the dependence of stresses on deformations, simultaneously introducing elastic trail stress $\underline{\underline{\sigma}}_{\text{elas}}$.

$$\underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}^e = (3k\underline{\underline{\underline{\underline{J}}}} + 2\mu\underline{\underline{\underline{\underline{K}}}}) : (\underline{\underline{\varepsilon}} - \underline{\underline{\varepsilon}}^p) = \underline{\underline{\sigma}}_{\text{elas}} - (3k\underline{\underline{\underline{\underline{J}}}} + 2\mu\underline{\underline{\underline{\underline{K}}}}) : \underline{\underline{\varepsilon}}^p, , \tag{64}$$

$$\underline{\underline{\sigma}}_{\text{elas}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}} = (3k\underline{\underline{\underline{\underline{J}}}} + 2\mu\underline{\underline{\underline{\underline{K}}}}) : \underline{\underline{\varepsilon}}. \tag{65}$$

For further implications, it will be convenient to work in terms of deviatory parts of deformations and stresses, respectively :

$$\underline{\underline{e}} = \underline{\underline{\varepsilon}} - \frac{1}{3}\text{tr}\underline{\underline{\varepsilon}}\,\underline{\underline{1}},$$
$$\underline{\underline{s}} = \underline{\underline{\sigma}} - \frac{1}{3}\text{tr}\underline{\underline{\sigma}}\,\underline{\underline{1}}.$$

In this particular case, the associative plasticity law looks as follows

$$\dot{\underline{\underline{\varepsilon}}}^p = \dot{\lambda}\frac{\partial f}{\partial \underline{\underline{\sigma}}} = \dot{\lambda}\left(\frac{3}{2}\frac{\underline{\underline{s}}}{\sigma_{\text{eq}}} + \alpha\underline{\underline{1}}\right). \tag{66}$$

From this, we can conclude that

$$\dot{p} = \sqrt{\frac{2}{3}\dot{\underline{\underline{e}}}^p : \dot{\underline{\underline{e}}}^p} = \dot{\lambda}, \tag{67}$$

At each $n+1$ loading step, we look for the true values of the stress tensor, cumulative plastic strain and other quantities. At the same time, we know its values from the previous step n, this allows us to represent each quantity as the sum of the value from the last step and some increment. Everywhere below, we adhere to the following notation

$$\begin{aligned}
\underline{\underline{\sigma}}_{n+1} &= \underline{\underline{\sigma}}_{n} + \Delta\underline{\underline{\sigma}}, \\
\underline{\underline{s}}_{n+1} &= \underline{\underline{s}}_{n} + \Delta\underline{\underline{s}}, \\
\underline{\underline{\varepsilon}}^{p}_{n+1} &= \underline{\underline{\varepsilon}}^{p}_{n} + \Delta\underline{\underline{\varepsilon}}^{p}, \\
\underline{\underline{e}}^{p}_{n+1} &= \underline{\underline{e}}^{p}_{n} + \Delta\underline{\underline{e}}^{p}, \\
p_{n+1} &= p_n + \Delta p.
\end{aligned}$$

Now let us rewrite the associative plasticity law (66) in its discrete form using the expression (67)

$$\Delta\underline{\underline{\varepsilon}}^{p} = \Delta p \left(\frac{3}{2} \frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}} + \alpha \underline{\underline{1}} \right) \tag{68}$$

$$\Delta\underline{\underline{e}}^{p} = \Delta p \frac{3}{2} \frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}} \tag{69}$$

Differentiating the expression (64) by time, substituting the equalities (68)–(69) and applying the backward Euler scheme we get

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}^{\text{elas}}_{n+1} - 2\mu\Delta\underline{\underline{e}}^{p} - k\text{tr}(\Delta\underline{\underline{\varepsilon}}^{p})\underline{\underline{1}}, \tag{70}$$

$$\underline{\underline{s}}_{n+1} = \underline{\underline{s}}^{\text{elas}}_{n+1} - 2\mu\Delta\underline{\underline{e}}^{p}, \tag{71}$$

$$\underline{\underline{s}}^{\text{elas}}_{n+1} = \underline{\underline{s}}_{n+1} + 3\mu\Delta p \frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}} = \underline{\underline{s}}_{n+1}(1 + 3\mu\Delta p \frac{1}{\sigma^{\text{eq}}_{n+1}}), \tag{72}$$

$$\sigma^{\text{elas,eq}}_{n+1} = \sigma^{\text{eq}}_{n+1}(1 + 3\mu\Delta p \frac{1}{\sigma^{\text{eq}}_{n+1}}), \tag{73}$$

from which it follows that

$$\frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} = \frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}} \tag{74}$$

and

$$\sigma^{\text{eq}}_{n+1} = \sigma^{\text{elas,eq}}_{n+1} - 3\mu\Delta p. \tag{75}$$

Let us write down useful expressions

$$\mathrm{tr}\underline{\underline{\sigma}}_{n+1} = \mathrm{tr}\underline{\underline{\sigma}}^{\mathrm{elas}}_{n+1} - 3\kappa \mathrm{tr}\Delta\underline{\underline{\varepsilon}}^{p}, \tag{76}$$

$$\mathrm{tr}\Delta\underline{\underline{\varepsilon}}^{p} = 3\alpha\Delta p. \tag{77}$$

Since we are implementing a return-mapping procedure for the $n+1$ loading step, we are looking for $\underline{\underline{\sigma}}_{n+1}$ and p_{n+1} which are situated on the Drucker-Prager yield surface, i.e. they satisfy the equality

$$\sigma^{\mathrm{eq}}_{n+1} + \alpha \mathrm{tr}\underline{\underline{\sigma}}_{n+1} - R(p_{n+1}) = 0, \tag{78}$$

substituting expressions (75)–(77) into equality (78), we get an expression for the cumulative plastic strain increment Δp

$$\Delta p = \frac{\sigma^{\mathrm{elas,eq}}_{n+1} + \alpha \mathrm{tr}\underline{\underline{\sigma}}^{\mathrm{elas}}_{n+1} - \sigma_0 - hp_n}{\gamma}, \tag{79}$$

where $\gamma = 3\mu + 9\alpha^2\kappa + h$.

Combining the equations (68)–(70) and (79) together we can derive the final expression of $\underline{\underline{\sigma}}_{n+1}$:

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}^{\mathrm{elas}}_{n+1} - \left(\beta_{n+1}\underline{\underline{s}}^{\mathrm{elas}}_{n+1} + 3k\alpha\Delta p\underline{\underline{1}}\right), \tag{80}$$

where $\beta_{n+1} = 3\mu\frac{\Delta p}{\sigma^{\mathrm{elas,eq}}_{n+1}}$.

As the text repeatedly pointed out, an important element of the numerical solution of plasticity problems is the stress-strain derivative. In the case of the Drucker-Prager criterion, we can calculate it analytically. This means that we need to find an explicit expression for

$$\underline{\underline{\underline{\underline{C}}}}^{\mathrm{tang}}_{n+1} = \frac{\partial\underline{\underline{\sigma}}_{n+1}}{\partial\Delta\underline{\underline{\varepsilon}}} = \underline{\underline{\underline{\underline{C}}}} - \frac{\partial(2\mu\Delta\underline{\underline{e}}^{p} + k\mathrm{tr}\Delta\underline{\underline{\varepsilon}}^{p}\underline{\underline{1}})}{\partial\Delta\underline{\underline{\varepsilon}}} = \underline{\underline{\underline{\underline{C}}}} - \underline{\underline{\underline{\underline{D}}}},$$

where $\underline{\underline{\underline{\underline{C}}}}^{\mathrm{tang}}_{n+1}$, $\underline{\underline{\underline{\underline{C}}}}$ and $\underline{\underline{\underline{\underline{D}}}}$ are the forth order tensors and $\underline{\underline{\underline{\underline{D}}}}$ represents a plastic correction of the derivative.

First of all, let us write following derivatives explicitly using the equalities (65), (75)–

(76), (79) :

$$\frac{\partial \underline{\underline{s}}^{\text{elas}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}} = 2\mu \underline{\underline{\underline{\underline{K}}}}, \tag{81}$$

$$\frac{\partial \sigma^{\text{elas,eq}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}} = \frac{3\mu}{\sigma^{\text{elas,eq}}_{n+1}} \underline{\underline{s}}^{\text{elas}}_{n+1}, \tag{82}$$

$$\frac{\partial \text{tr}\sigma^{\text{eq}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}} = 3k\underline{\underline{1}}, \tag{83}$$

$$\frac{\partial \Delta p}{\partial \Delta \underline{\underline{\varepsilon}}} = \frac{1}{\gamma} \frac{\partial (\sigma^{\text{elas,eq}}_{n+1} + \alpha \text{tr}\underline{\underline{\sigma}}^{\text{elas}}_{n+1})}{\partial \Delta \underline{\underline{\varepsilon}}} = \frac{1}{\gamma} \left(\frac{3\mu}{\sigma^{\text{elas,eq}}_{n+1}} \underline{\underline{s}}^{\text{elas}}_{n+1} + 3k\alpha \underline{\underline{1}} \right) = \frac{1}{\gamma} (3\mu \underline{\underline{N}}_{n+1} + 3k\underline{\underline{1}}), \tag{84}$$

where $\underline{\underline{N}}_{n+1} = \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} = \frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}}$. Then we use these expressions to get $\underline{\underline{\underline{\underline{D}}}}$:

$$\underline{\underline{\underline{\underline{D}}}} = \frac{\partial (2\mu \Delta \underline{\underline{\varepsilon}}^p + k\text{tr}\Delta \underline{\underline{\varepsilon}}^p \underline{\underline{1}})}{\partial \Delta \underline{\underline{\varepsilon}}} = \frac{\partial}{\partial \Delta \underline{\underline{\varepsilon}}} \left(\Delta p \left(3\mu \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} + 3k\alpha \underline{\underline{1}} \right) \right) =$$

$$= \left(\frac{\partial \Delta p}{\partial \Delta \underline{\underline{\varepsilon}}} \otimes \left(3\mu \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} + 3k\alpha \underline{\underline{1}} \right) + \Delta p \left(3\mu \frac{\partial \underline{\underline{s}}^{\text{elas}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}} \frac{1}{\sigma^{\text{elas,eq}}_{n+1}} \right) - \Delta p 3\mu \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{(\sigma^{\text{elas,eq}}_{n+1})^2} \otimes \frac{\partial \sigma^{\text{elas,eq}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}} \right) =$$

$$= \left(\left(3\mu \underline{\underline{N}}_{n+1} + 3\alpha k \underline{\underline{1}} \right) \frac{1}{\gamma} \otimes \left(3\mu \underline{\underline{N}}_{n+1} + 3\alpha k \underline{\underline{1}} \right) + 2\mu \beta_{n+1} \underline{\underline{\underline{\underline{K}}}} - 3\mu \beta_{n+1} \underline{\underline{N}}_{n+1} \otimes \underline{\underline{N}}_{n+1} \right),$$

$$\underline{\underline{\underline{\underline{D}}}} = \left(3\mu \left(\frac{3\mu}{\gamma} - \beta_{n+1} \right) \underline{\underline{N}}_{n+1} \otimes \underline{\underline{N}}_{n+1} + \frac{9\alpha\mu k}{\gamma} (\underline{\underline{1}} \otimes \underline{\underline{N}}_{n+1} + \underline{\underline{N}}_{n+1} \otimes \underline{\underline{1}}) + \frac{9\alpha^2 k^2}{\gamma} \underline{\underline{1}} \otimes \underline{\underline{1}} + 2\mu \beta_{n+1} \underline{\underline{\underline{\underline{K}}}} \right)$$

Thus, we have derived the analytical formulas necessary for the return-mapping procedure in the case of the Drucker-Prager yield criterion. Substituting zero instead of the alpha parameter in the formulas above, we get exactly the analytical expressions for the von Mises criterion presented in section **??**.