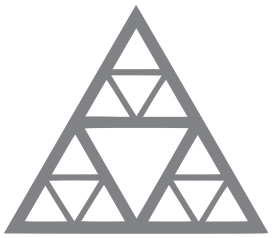


École des Ponts ParisTech

2021-2022

End-of-studies project

Department of Mechanical Engineering and Materials

Andrey Latyshev

Double degree engineering student

Finite-element implementation of standard and softening 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 : Civilité Prénom Nom

Project director : Civilité Prénom Nom

Study advisor : Civilité Prénom Nom

Todo list

Acknowledgements

Abstract

This internship aims at exploring a finite-element formulation of plasticity in the next generation FEniCS problem solving environment. The main goal is to propose an efficient and generic implementation which can tackle both classical and softening plasticity models. The intern will first familiarize himself with the DOLFINx computational environment and adapt existing implementations of legacy FEniCS code. The implementation will be then extended to the resolution of the local plasticity problem using convex optimization solvers and assess its efficiency compared to standard return mapping algorithms. Finally, softening plasticity will be considered and regularization strategies in order to prevent mesh dependency will be explored.

Résumé

Synthèse du mémoire en français

Table of contents

List of tables

List of figures

Introduction

1 Context and laboratory presentation

2 Literature review

sdf (COUSSY, 2004) sdf (Bruno et al., 2020) sdf (Lam et al., 2015) sd (Diamond & Boyd, 2016) s (Domahidi et al., 2013)

test

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.

To solve this problem, it is necessary to restore the stress field. To do this, 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

$$u_z = 0, \tag{1}$$

$$\varepsilon_{xz} = \varepsilon_{yz} = \varepsilon_{zz} = 0, \tag{2}$$

$$\sigma_{xz} = \sigma_{yz} = 0. \tag{3}$$

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

3.1.1 Plasticity

One of the simplest models describing the nonlinear behavior of solids is an elastic-plastic one. The idea of it is that at a certain moment of loading, irreversible deformations occur. 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. In this paper, we focused on models defined by von Mises, Drucker-Prager and Rankine yield criteria.

Considering plastic models, a number of hypotheses about the behavior of the material based on observations from experiments are traditionally introduced. One of such hypotheses is the additive decomposition of total strains $\underline{\underline{\varepsilon}}$

$$\underline{\underline{\varepsilon}} = \underline{\underline{\varepsilon}}^e + \underline{\underline{\varepsilon}}^p, \tag{4}$$

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

Isotropic linear elastic behavior is expressed as the following dependency

$$\underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}^e = \left(3k\underline{\underline{\underline{\underline{J}}}} + 2\mu\underline{\underline{\underline{\underline{K}}}}\right) : \underline{\underline{\varepsilon}}^e, \tag{5}$$

where $\underline{\underline{\underline{\underline{C}}}}$ is the fourth-order stiffness tensor, $k = (3\lambda + 2\mu)$ 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 forth-order tensors associated with the deviatoric operator and a unit tensor respectively.

To obtain plastic deformations, the stress point must not only be on the yield contour defining by yield function f as an explicite expression of a yield criterion. Plastic straining will take place if and only if the yield function f vanishes :

$$f(\underline{\underline{\sigma}}) = 0 \tag{6}$$

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

$$f(\underline{\underline{\sigma}}, p) = 0, \tag{7}$$

where the 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

$$\dot{\underline{\underline{\varepsilon}}}^p = \dot{\lambda}\frac{\partial f(\underline{\underline{\sigma}}, p)}{\partial \underline{\underline{\sigma}}}, \tag{8}$$

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

The associative hardening law has the following form

$$\dot{p} = -\dot{\lambda}\frac{\partial f(\underline{\underline{\sigma}}, p)}{\partial p}. \tag{9}$$

We introduce here the hardening force Θ. In our particular case of linear isotropic hardening the hardening law looks as follows

$$\Theta = Hp, \tag{10}$$

where H is hardening modulus.

Loading/unloading complementary conditions

Write full discription of KKT conditions, Drucker-Prager Theorem, etc

$$\dot{\lambda} \geq 0, \quad f(\underline{\underline{\sigma}}, p) \leq 0, \quad \dot{\lambda} f(\underline{\underline{\sigma}}, p) = 0 \tag{11}$$

Thus, taking into account the described above assumptions 4–11 we can formulate the elasto-plastic model. Let us consider the following system

$$\Omega : \operatorname{div} \underline{\underline{\sigma}} = 0, \tag{12}$$

$$\Omega : \underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}^e, \tag{13}$$

$$\partial\Omega_{\mathrm{N}} : \underline{\underline{\sigma}} \cdot \underline{n} = q \cdot \underline{n}, \tag{14}$$

$$\partial\Omega_{\mathrm{D}} : \underline{u} = \underline{u}_D, \tag{15}$$

where the equilibrium equation 12 is defined in the domain Ω as well as the constitutive equation 13 of linear elasticity, $\underline{n}$ is a surface normal to the boundary $\partial\Omega_{\mathrm{N}}$, where we apply Neuman boundary conditions 14. Dirichlet boundary conditions are defined by the displacements equality 15 on the boundary $\partial\Omega_{\mathrm{D}}$.

We call the described above system of partial differential equations 12–15 and assumptions 4–11 elastoplastic constitutive equations. In the next part, we will talk in detail about the numerical solutions of these equations.

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. On every loading step we solve the system 4–15 using the finite element and Newton methods.

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

$$V = \{\underline{u} = (u_x, u_y) \in H^1(\Omega) \,|\, \underline{u} = \underline{u}_D \text{ on } \partial\Omega_D\}, \tag{16}$$

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

$$\text{Find } \underline{u} \in V \text{ such that,} \tag{17}$$

$$R(\underline{u}) = \int_\Omega \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \,\mathrm{d}x - F_{\text{ext}} = 0, \quad \forall \underline{v} \in V, \tag{18}$$

where F_{ext} is an external force acting on the $\partial\Omega_N$.

The variational equation 18 is a nonlinear, that can be solved through successive linearizations using the Newton algorithm. The linearized problem looks like this one

$$\text{Find } \Delta\underline{u} \in V \text{ such that,} \tag{19}$$

$$\int_\Omega \left(\frac{\partial \underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}(\Delta\underline{u}))}{\partial \underline{\underline{\varepsilon}}} : \underline{\underline{\varepsilon}}(\underline{u}) \right) : \underline{\underline{\varepsilon}}(\underline{v}) \,\mathrm{d}x = -\int_\Omega \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \,\mathrm{d}x + F_{\text{ext}}, \tag{20}$$

At each loading step $n+1$, we solve the problem 17–18 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 19–20. Now it is necessary to describe the nonlinear behavior of the stress tensor $\underline{\underline{\sigma}}(\underline{u})$ and its derivative.

Here we introduce a quite common algorithm solving the elasto-plastic problems : the return-mapping procedure. It consists in finding a new stress $\underline{\underline{\sigma}}_{n+1}$ and plastic strain p_{n+1} verifying the current plasticity condition from a previous stress $\underline{\underline{\sigma}}_n$ and plastic strain p_n and an increment of total deformation $\Delta\underline{\underline{\varepsilon}}$. The hole procedure checks, if an elastic trial stress $\underline{\underline{\sigma}}^{\text{elas}}_{n+1}$ goes beyond the boundary of the yield surface, in other words if $f(\sigma^{\text{elas,eq}}_{n+1}, p_n) > 0$, then the stress tensor and the plastic strain should be corrected by projecting them on the surface $f(\underline{\underline{\sigma}}, p) = 0$.

We write down the equations 4–11 in its discrete form, where we use a one-point Euler forward integration rule for "time" derivatives :

$$\underline{\underline{\sigma}}^{\text{elas}}_{n+1} = \underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : \Delta\underline{\underline{\varepsilon}} \tag{21}$$

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}^{\text{elas}}_{n+1} - \Delta\underline{\underline{\sigma}}, \tag{22}$$

$$p_{n+1} = p_n - \Delta p \tag{23}$$

$$\Delta\lambda \geq 0, \quad f(\underline{\underline{\sigma}}_{n+1}, p_{n+1}) \leq 0, \quad \Delta\lambda \cdot f(\underline{\underline{\sigma}}_{n+1}, p_{n+1}) = 0, \tag{24}$$

where the increment $\Delta\underline{\underline{\varepsilon}}$ depends on the displacement $\Delta\underline{u}$ calculated on a current loading step, $\Delta\underline{\underline{\sigma}}$ and Δp are the stress and plastic strain corrections respectively. Increments $\Delta\underline{\underline{\sigma}}, \Delta p$ and $\Delta\lambda$ can be found using the second Newton method, the local one.

We write out a discret form of the variational problem 17–18 :

$$\text{Find } \underline{u}_{n+1} \in V \text{ such that,} \tag{25}$$

$$R(\underline{u}_{n+1}) = \int_\Omega \underline{\underline{\sigma}}(\underline{u}_{n+1}) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x - F^{\text{ext}}_{n+1} = 0, \quad \forall \underline{v} \in V, \tag{26}$$

and its linearized version 19–20 by denoting the derivate $\frac{\partial\underline{\underline{\sigma}}}{\partial\underline{\underline{\varepsilon}}}$ by $\underline{\underline{\underline{\underline{C}}}}_{\text{tang}}$:

$$\text{Find } \Delta\underline{u} \in V \text{ such that,} \tag{27}$$

$$\int_\Omega \left(\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1} : \underline{\underline{\varepsilon}}(\Delta\underline{u})\right) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x = -\int_\Omega \underline{\underline{\sigma}}_n : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}x + F^{\text{ext}}_{n+1}, \tag{28}$$

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

In total we follow the next steps to solve elastoplastic constitutive equations 4–15 of the original problem

1. to divide the loading on steps (an external loop)
2. to solve the variational problem 17–18 and to find the current displacement $\underline{u}_{n+1}$ using the Newton method on every loading step (an internal loop)
3. to solve the linearized problem 27–28 and to find the displacement increment $\Delta\underline{u}$ using finite element method on every Newton iteration
4. to carry out the return-mapping procedure and to find $\underline{\underline{\sigma}}_{n+1}$ and p_{n+1} in Gauss nodes (a middle loop)

5. to find increments $\Delta\underline{\underline{\sigma}}, \Delta p$ and $\Delta\lambda$ using the second Newton method (a bottom loop)

As can be seen from the described algorithm, modeling of plastic behavior 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 last step of the algorithm 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. The solution may be to reformulate the original problem in terms of the convex optimization problem, which is the subject of the next chapter.

3.1.3 Plasticity using convex optimization

When we talk about optimization problems (or mathematical programming), we often have in mind the following minimization problem

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

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

The particular interest to us is a subfield of convex optimization, when constraints represents a convex cone, in other words conic optimization. For this type of problems, there are really efficient and fast solvers. 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 elasto-plastic constitutive equations via the MP approach. This stems from the fact that many of the 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 conic constraint.

We just need to reformulate the original return-mapping algorithm **??**–24 in terms of conic programming. Let us consider the following conic optimization problem

$$\begin{cases} \min\limits_{\underline{\underline{\sigma}},p} F(\underline{\underline{\sigma}},p), \\ f(\underline{\underline{\sigma}},p) \leq 0, \end{cases} \tag{30}$$

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

$$F(\underline{\underline{\sigma}},p) = \frac{1}{2}(\underline{\underline{\sigma}}^{\text{elas}}_{n+1} - \underline{\underline{\sigma}}) : \underline{\underline{\underline{\underline{S}}}} : (\underline{\underline{\sigma}}^{\text{elas}}_{n+1} - \underline{\underline{\sigma}}) + \frac{1}{2}H(p^{\text{elas}}_{n+1} - p)^2. \tag{31}$$

Thus, the solution $(\underline{\underline{\sigma}}^*, p^*)$ of the problem 30 is the closest projection of $(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p^{\text{elas}}_{n+1})$ on the yield surface.

3.1.4 Yield criteria

Describe every yield criterion

$$f(\underline{\underline{\sigma}}, p, \sigma_C, \sigma_T, \dots) \leq 0 \tag{32}$$

Von Mises criterion

$$\sigma_{\text{eq}} \leq R(\sigma_0, p) \tag{33}$$

where $\sigma_{\text{eq}} = \sqrt{\frac{3}{2}\underline{\underline{s}} : \underline{\underline{s}}}$.

Drucker-Prager criterion

$$\sigma_{\text{eq}} + \alpha \text{tr}\underline{\underline{\sigma}} \leq R(\sigma_0, p) \tag{34}$$

Rankine criterion

$$\sigma_{\mathrm{I}} \leq R(\sigma_T, p) \tag{35}$$

$$-\sigma_{\mathrm{III}} \leq R(\sigma_C, p) \tag{36}$$

where $\sigma_{\mathrm{I}} \geq \sigma_{\mathrm{II}} \geq \sigma_{\mathrm{III}}$ are , σ_T is σ_C.

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 1 shows the domain, where symmetry conditions are set on the left and bottom sides and pressure is set on the inner wall

$$\Omega : \operatorname{div} \underline{\underline{\sigma}} = 0, \tag{37}$$

$$\Omega : \underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}^e, \tag{38}$$

$$\partial\Omega_{\text{internal}} : \underline{\underline{\sigma}} \cdot \underline{n} = q \cdot \underline{n}, \tag{39}$$

$$\partial\Omega_{\text{left}} : u_x = 0, \tag{40}$$

$$\partial\Omega_{\text{bottom}} : u_y = 0, \tag{41}$$

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

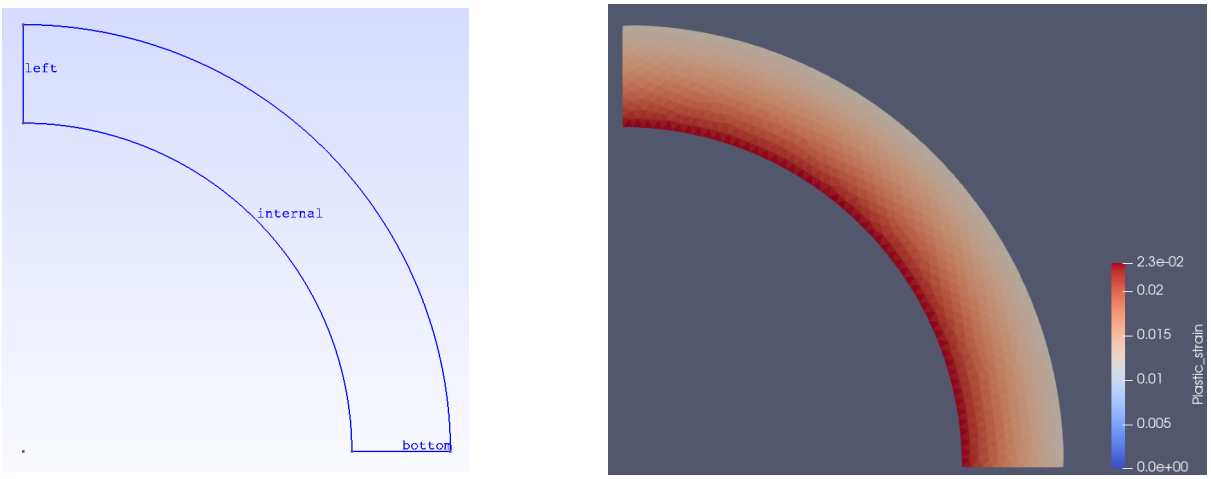
$$f(\underline{\underline{\sigma}}) = \sqrt{\frac{3}{2}\underline{\underline{s}} : \underline{\underline{s}}} - \sigma_0 - Hp \leq 0 \tag{42}$$

The weak formulation of the problem 37–41 looks like this

$$\text{Find } \underline{u} \text{ such that} \tag{43}$$

$$\int_\Omega \underline{\underline{\sigma}}_{n+1}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \,\mathrm{d}x - q_{n+1} \int_{\partial\Omega_{\text{internal}}} \underline{n} \cdot \underline{v} \,\mathrm{d}x = 0, \quad \forall \underline{v} \in V. \tag{44}$$

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 1 – Cylinder expansion problem

In this particular case, the solution of the equilibrium problem of an elastic-plastic body can be found analytically. A detailed conclusion of a 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

$$\Delta p = \begin{cases} \frac{1}{3\mu+H}, & \text{if } f_{\text{elas}} \geq 0, \\ 0, & \text{otherwise}, \end{cases} \tag{45}$$

$$\Delta \underline{\underline{\sigma}} = \beta \underline{\underline{s}}_{n+1}, \tag{46}$$

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

The stress derivative is written with the following formula

$$\frac{\mathrm{d}\underline{\underline{\sigma}}_{n+1}}{\mathrm{d}\underline{\underline{\Delta\varepsilon}}} = \underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1} = \underline{\underline{\underline{\underline{C}}}} - 3\mu \left(\frac{3\mu}{3\mu + H} - \beta \right) \underline{\underline{N}} \otimes \underline{\underline{N}} - 2\mu\beta \underline{\underline{\underline{\underline{J}}}} \tag{47}$$

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

So we can explicitly write out the linearization of the variational problem 43–44

$$\text{Find } \Delta \underline{u} \in V \text{ such that,} \tag{48}$$

$$\int_\Omega \left(\underline{\underline{\underline{\underline{C}}}}^{\text{tang}}_{n+1} : \underline{\underline{\varepsilon}}(\Delta \underline{u}) \right) : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}x = - \int_\Omega \underline{\underline{\sigma}}_n : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}x + q_{n+1} \int_{\partial\Omega_{\text{internal}}} \underline{n} \cdot \underline{v} \, \mathrm{d}x, \quad \forall \underline{v} \in V. \tag{49}$$

Explicit expressions 45–47 are possible thanks to the form of von Mises criterion 42. Similar expressions can be obtained for other criteria. As part of this work, a code has been written that allows to work with the plasticity of Drucker-Prager. A detailed derivation of similar expressions in this case can be found in the appendix section A.

Now we have at our disposal everything necessary for the successful modeling 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 modeling 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 modeling an elastic-plastic material using the FEniCSx library, as well as how to combine this modeling 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 of 2D plan in the Voigt notation.

According to this notation the stress tensor representation is a vector

$$\boldsymbol{\sigma} = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy})^T, \tag{50}$$

the strain one is a vector

$$\boldsymbol{\varepsilon} = (\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \varepsilon_{xy})^T, \tag{51}$$

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

$$\boldsymbol{C} = \begin{pmatrix} \lambda + 2\mu & \lambda & \lambda & 0 \\ \lambda & \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & 0 & 2\mu \end{pmatrix}. \tag{52}$$

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

Thus we rewrite described above problems in the Voigt notation.

$$\int_{\Omega} \boldsymbol{\sigma}_{n+1}(\Delta \boldsymbol{u})\boldsymbol{\varepsilon}(\boldsymbol{v})\,\mathrm{d}x - q_{n+1} \int_{\partial\Omega_{\text{internal}}} \boldsymbol{n} \cdot \boldsymbol{v}\,\mathrm{d}x = 0, \quad \forall \boldsymbol{v} \in V, \tag{53}$$

where

$$\int_{\Omega} (\boldsymbol{C}_{\text{tang}}\,\boldsymbol{\varepsilon}(\Delta \boldsymbol{u})) \cdot \boldsymbol{\varepsilon}(\boldsymbol{v})\,\mathrm{d}x - q_{n+1} \int_{\partial\Omega_{\text{internal}}} \boldsymbol{n} \cdot \boldsymbol{v}\,\mathrm{d}x = 0, \quad \forall \boldsymbol{v} \in V. \tag{54}$$

The convex optimization problem 30 in Voigt notation is

$$\begin{cases} \min\limits_{\boldsymbol{\sigma},p} F(\boldsymbol{\sigma}, p), \\ f(\boldsymbol{\sigma}, p) \leq 0, \end{cases} \tag{55}$$

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

$$F(\boldsymbol{\sigma}, p) = \frac{1}{2}(\boldsymbol{\sigma}_{n+1}^{\text{elas}} - \boldsymbol{\sigma})^T \boldsymbol{S}(\boldsymbol{\sigma}_{n+1}^{\text{elas}} - \boldsymbol{\sigma}) + \frac{1}{2} H (p_{n+1}^{\text{elas}} - p)^2, \tag{56}$$

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 points 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 UFLx package. It can contain a global coefficients `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. There is an example below 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.

To efficiently calculate stresses using FEniCSx tools, it is required to use the `eval` method of the entity `fem.Expression`, which interpolates the mathematical expression of stresses in Gaussian nodes. For ours purposes 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_elas_, 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 17–18, 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

$$E^* = \frac{E}{E} = 1, \quad \sigma_0^* = \frac{\sigma_0}{E} \tag{57}$$

where E^* and σ_0^* are dimensionless Young modulus and uniaxial strength respectively.

3.2.4 Classical approach

Complete this section

```
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

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)
```

```
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)
```

```
fs.interpolate_quadrature(sig_, sig)
fs.interpolate_quadrature(n_elas_, n_elas)
fs.interpolate_quadrature(beta_, beta)
```

```
W0e = ufl.FiniteElement("Quadrature", mesh.ufl_cell(), degree=deg_stress, quad_scheme='default')
W0 = fem.FunctionSpace(mesh, W0e)
n_elas = fem.Function(W)
beta = fem.Function(W0)
p = fem.Function(W0, name="Cumulative_plastic_strain")
dp = fem.Function(W0)
```

$$R = \int_\Omega \underline{\underline{\sigma}}_{n+1} : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}\Omega - \underline{F}_{\text{ext}} = \int_\Omega \left(\underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : (\Delta\underline{\underline{\varepsilon}}_n - \Delta\underline{\underline{\varepsilon}}^p_n)\right) : \underline{\underline{\varepsilon}}(\underline{v})\,\mathrm{d}\Omega - \underline{F}_{\text{ext}} = 0 \tag{58}$$

where $\Delta\underline{\underline{\varepsilon}}_n = \underline{\underline{\varepsilon}}(\Delta\underline{u}_n)$ and $\Delta\underline{\underline{\varepsilon}}_n = \underline{\underline{\varepsilon}}^p(\Delta\underline{u}_n)$.

$$J(\underline{u}) = R'(\underline{u}) = -R(\underline{u}) \tag{59}$$

$$J(\underline{u}) = \frac{\partial}{\partial \underline{u}} \left(\int_\Omega \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}x \right) = - \int_\Omega \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}x + F_{\text{ext}} \tag{60}$$

$$\underline{\underline{\varepsilon}}(\Delta \underline{u}_n) = \frac{1}{2} \left(\nabla \Delta \underline{u} + \nabla \Delta \underline{u}^T \right) \tag{61}$$

$$\underline{\underline{\varepsilon}}^p(\Delta \underline{u}_n) = \begin{cases} \Delta p_n \left(\frac{3}{2} \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} + \alpha \underline{\underline{1}} \right), & \text{if } f(\underline{\underline{\sigma}}^{\text{elas}}_{n+1}, p_n) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{62}$$

where $\Delta p_n = p_{n+1} - p_n$

```
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
```

```
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)
```

3.2.5 Convex plasticity

We call the convex plasticity or the convex optimization approach the one 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

this problem simultaneously for several Gaussian nodes. In this case, the order is not important. Therefore, we need to reformulate the problem 55 in vector form.

For this purpose, we will assemble several Gauss nodes into groups or patches. If there are N_q Gauss points 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 55 form :

$$\begin{cases} \min\limits_{\boldsymbol{\Sigma},\boldsymbol{P}} F(\boldsymbol{\Sigma},\boldsymbol{P}), \\ \boldsymbol{f}(\boldsymbol{\Sigma},\boldsymbol{P}) \leq 0, \end{cases} \tag{63}$$

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

$$F(\boldsymbol{\Sigma},\boldsymbol{P}) = \frac{1}{2}(\boldsymbol{\Sigma}^{\text{elas}}_{n+1} - \boldsymbol{\Sigma})^T \mathbb{S} (\boldsymbol{\Sigma}^{\text{elas}}_{n+1} - \boldsymbol{\Sigma}) + \frac{1}{2} H (\boldsymbol{P}^{\text{elas}}_{n+1} - \boldsymbol{P})^T (\boldsymbol{P}^{\text{elas}}_{n+1} - \boldsymbol{P}), \tag{64}$$

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 to use various third-party conve solvers, for instance, SCS, MOSEK and ECOS were used in this work. For convenience the Return Mapping class is written, which formulates the problem using the cvxpy library and finds its numerical solution.

Note that the free energy F doesn't depend on the plasticity criterion. In order to change the plasticity model, it is enough to change the constraint of the optimization problem 63. 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 :

TOCITE — We solve the problem in a weak formulation .

TOCITE — On each loading step, using the quasi-Newton method, we solve a nonlinear problem .

— During each quasi-Newton iteration we carry out the projection of $\boldsymbol{\sigma}^{n+1}_{\text{elas}}$ on the yield surface, solving the vectorized optimization problem 63.

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 63 is solved additionally for the remaining N_q mod N_{patch} Gauss points.

3.3 Performance

The second part of this paper discusses strategies to improve the performance of the described above approaches solving plasticity problems. Before describing these ones 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.

In the particular case of modeling an elastic-plastic material defined by the von Mises yield TOCITE criterion, we explicitly wrote down the variational formulation of . Obviously, the dependence is nonlinear in nature, but this is just one example where UFLx features are sufficient to solve the problem. There are other models where the stress tensor $\underline{\underline{\sigma}}$ is difficult to represent explicitly. Let us show you some 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 scalar, vector or even tensorial quantities (α here). The latter do 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.

TOCITE Although we use the quasi-Newton method to find the solution of , 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 for von Mises and Drucker-Prager plasticity models), but it is difficult to express it using UFLx 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 which will allow to do a simultaneous calculation of $\underline{\underline{\sigma}}$ and its derivative during assembling procedure regardless of the calculation method.

In addition to the inconveniences described above, we are faced with the problem of excessive TOCITE memory allocation. For example, in the classic approach , we are forced to save global variables β and $\underline{\underline{N}}$. If there is a mesh containing N_{elements} elements, we have to allocate memory for a vector with the size $N_{\text{elements}} * 3$ for the scalar field β from `W0` finite space and another one with the size $N_{\text{elements}} * 3 * 4$ for the vector field $\underline{\underline{N}}$ (in Voigt notation) from `W` finite space. At the same time they are intermediate variables, necessary only for the calculation of the stress tensor and its derivative.

So we would like to avoid such a waste of memory.

Write about global Ctang ?

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

1. To define nonlinear expressions of variational problems in more complex way than it's allowed by UFLx
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

Next 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 UFLx 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 a possibility to calculate g expression, no matter how difficult it is, in every element node (for instance, in every gauss point, 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 an other 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 now. If there is one local value of g (only 1 gauss point), the use of the `fem.Constant` entity would be enough, but it is a particular case. In general, we are dealing 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 point. 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 assembling procedure

3.3.2 Examples

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

$$\text{Find } \underline{u} \in V \text{ s.t.} \tag{65}$$

$$\int_\Omega \underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}(\underline{u})) : \underline{\underline{\varepsilon}}(\underline{v}) d\underline{x} = 0 \quad \forall \underline{v} \in V, \tag{66}$$

with following boundary conditions

$$(0,0) : u_y = 0, \tag{67}$$

$$\partial\Omega_{\text{left}} : u_x = 0, \tag{68}$$

$$\partial\Omega_{\text{right}} : u_x = t \cdot u_{\text{bc}}, \tag{69}$$

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 :

$$\underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}) = \underline{\underline{\underline{\underline{C}}}} : \underline{\underline{\varepsilon}}, \tag{70}$$

and for which the derivative is :

$$\frac{d\underline{\underline{\sigma}}}{d\underline{\underline{\varepsilon}}} = \underline{\underline{\underline{\underline{C}}}}. \tag{71}$$

Let's focus on the key points. In this "naive" example the derivative is constant, but in

general non-linear models, its value will directly depend on the local value of $\underline{\underline{\varepsilon}}$. We would like to change this value at every assembling step and to avoid additional memory allocation. As it's the stress tensor derivative, it's 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 "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 a 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 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 describe explicitly the interaction between different coefficients of linear and bilinear forms. It allows us to write a quite general custom assembler, which will work for any kind non-linear problem. Thus we have to define two additional numba functions to calculate local values of forms kernels coefficients (see the code below).

```
@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.

TOCITE

The following example optimizes the classical approach solving the von Mises plasticity problem . We can conclude from this, that the fields $\underline{\underline{\sigma_{n+1}}} = \underline{\underline{\sigma_{n+1}}}(\underline{\underline{\Delta\varepsilon}}, \beta, \underline{\underline{N}}, dp, p_n, \underline{\underline{\sigma_n}})$ and $\mathbf{C}^{\text{tang}} = \mathbf{C}^{\text{tang}}(\beta, \underline{\underline{N}})$ depend on the common variables β and $\underline{\underline{N}}$. With the previous implementation, it was

necessary to allocate additional space for them and calculate $\underline{\underline{\sigma_{n+1}}}$ and $\mathbf{C}^{\text{tang}}$ separately, but now we can combine their local evaluations.

In comparison with the elasticity case the `CustomExpression sig` has more dependent fields. Look at the code below

```
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 been seen more clearly the dependance of the tensor $\mathbf{C}^{\text{tang}}$ on the calculation of the tensor $\underline{\underline{\sigma}}_{n+1}$. The full source code of this examples can be found in the `assembling_strategies` folder of the repository .

TOCITE

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

4 Results

4.1 Qualitative yield criterions 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 2. These results are obtained thanks to the convex plasticity approach, based on solving the optimization problem and using the FEniCSx and cvxpy libraries.

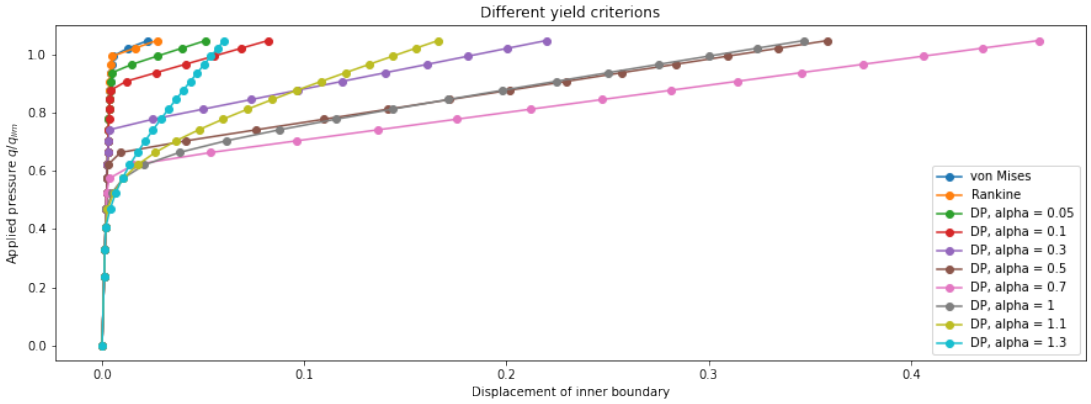


FIGURE 2 – Displacements of the inner boundary for different yield criteria using convex plasticity approach

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 behavior 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

The purpose of this work is not only the implementation of the convex plasticity approach in the form of ready-made program code, but also its effectiveness. If this implementation solves the task significantly slower than other approaches, then there will be insufficient benefits in it, especially if we are talking about 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 3 finite element meshes of different densities, the sizes of which are presented in the table below :

Mesh	Nodes	Elements	Gauss points 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 3–5 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 TOCITE a problem formulated in the form of cvxpy converts it into a canonical form, which most conic solvers work with. For convenience, the plots from these images are duplicated on a logarithmic scale.

As can be seen from the plots, the larger the patch size, the longer it takes to compile using cvxpy. Starting from some point, this time is comparable to the total time 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 points 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 with an increase in the size of the patch, the time to solve the optimization problem decreases significantly. Starting from some N_{patch}, this decrease does not considerably affect the time of the program.

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

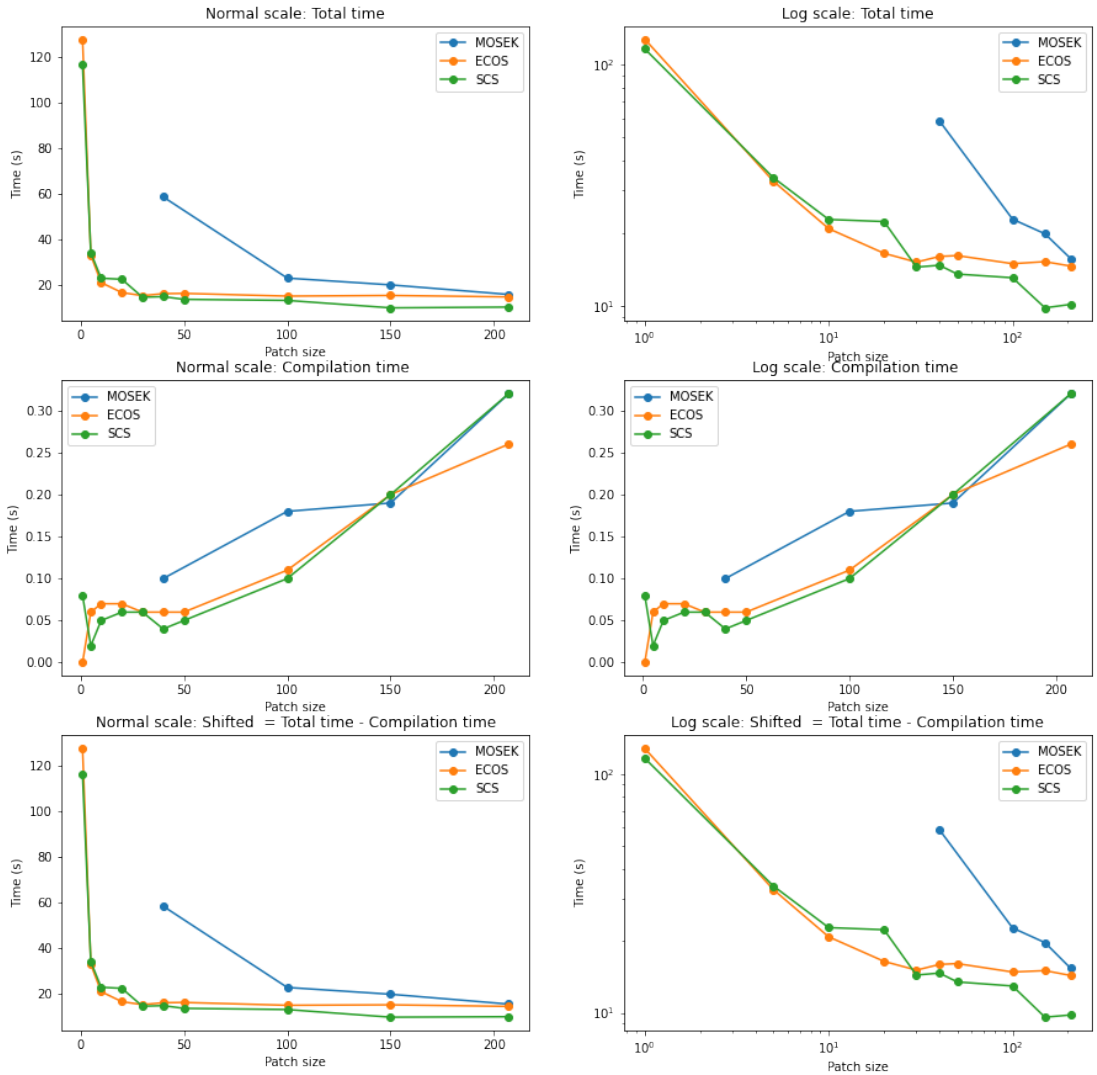


FIGURE 3 – Patch size effect on the time performance of solving the conic problem : coarse mesh case

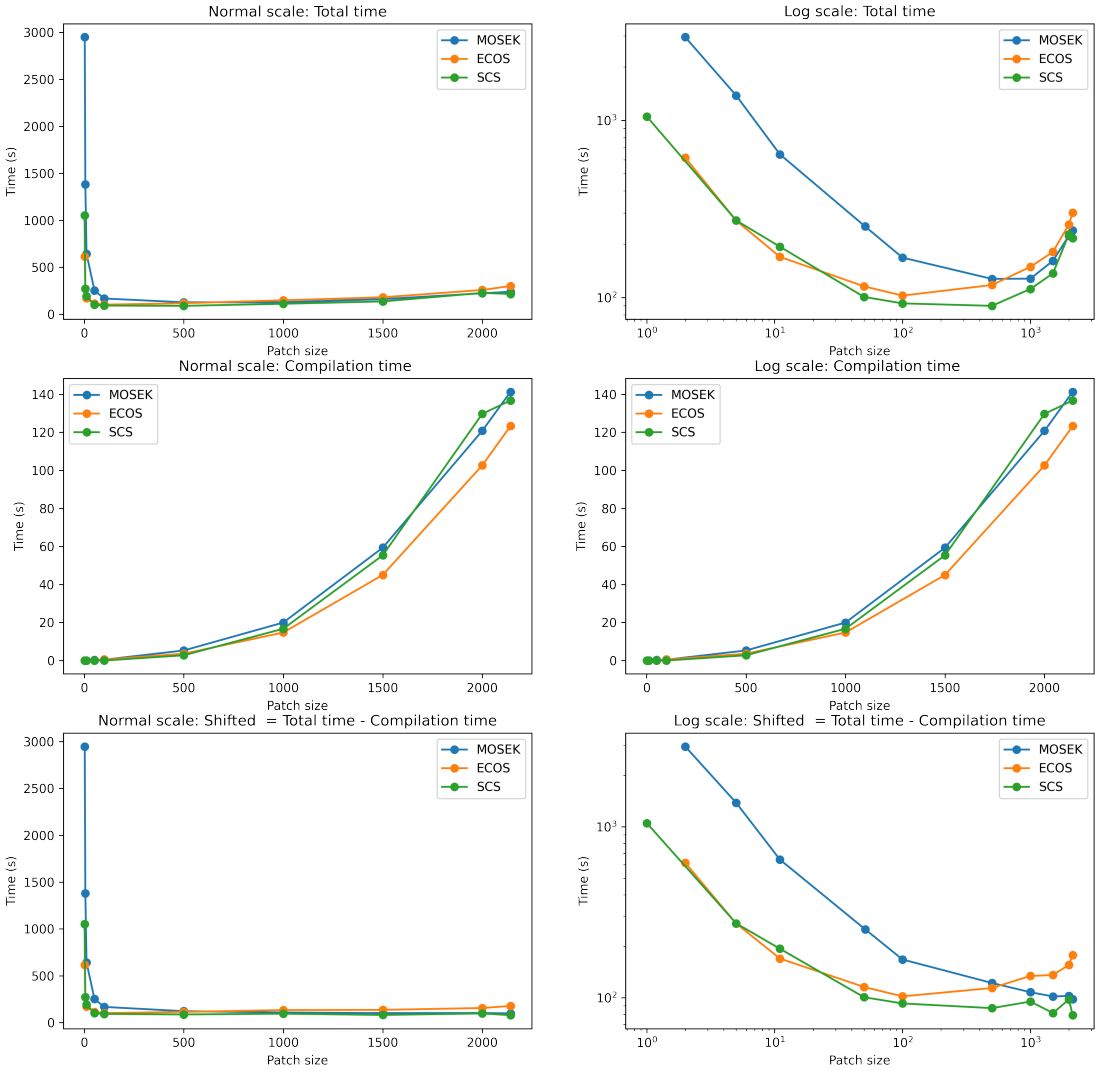


FIGURE 4 – Patch size effect on the time performance of solving the conic problem : medium mesh case

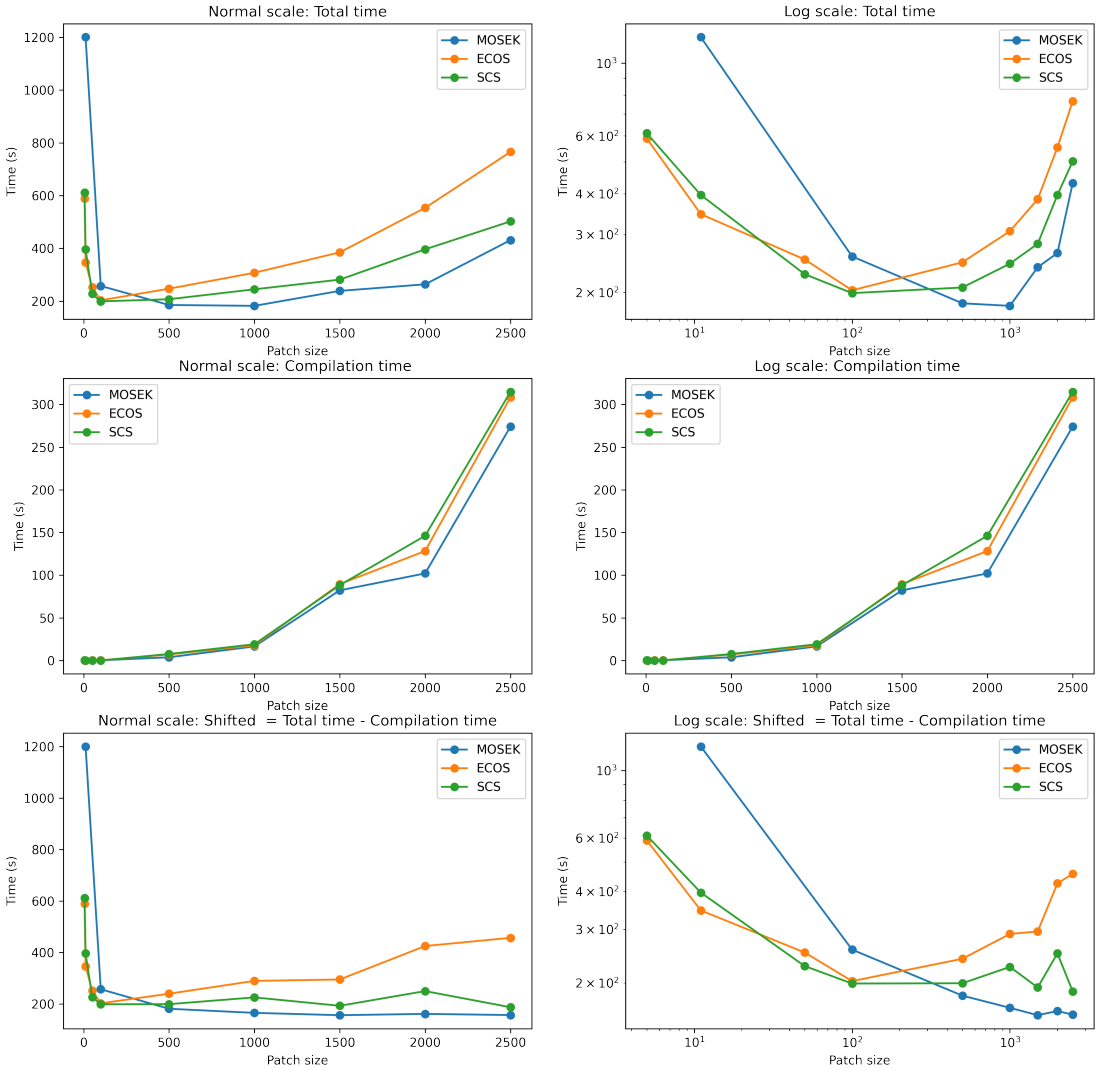


FIGURE 5 – Patch size effect on the time performance of solving the conic problem : dense mesh case

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 and depending on the mesh density. A series of numerical tests was performed based on the original cylinder expansion problem using the von Mises criterion. Each test corresponds to a finite element mesh, data of which are presented in the table 2.

Mesh	Nodes	Elements	Gauss points 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 6 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.

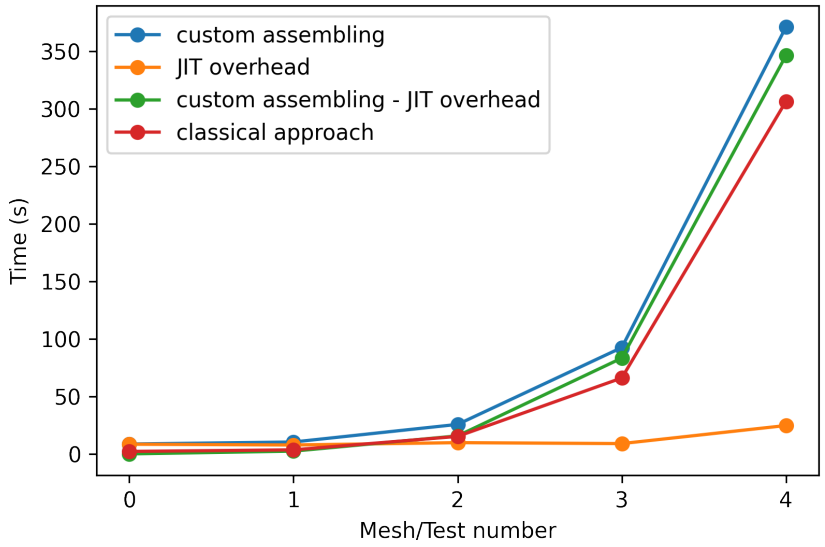


FIGURE 6 – Time performance of the custom assembling approach in comparison with the classical one

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 of the program tends to zero. This is possible due to the fact that 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 modeling process.

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

5 Discussion

general talk

about benefits of custom assembling approach

about benefits of plasticity + conic optimization

oprimization problem formulation

perspectives : custom assembling + SNESQN + cvxpygen

perspectives : derivate() from cvxpy to C

Conclusion

Bibliography

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. https://hal.archives-ouvertes.fr/hal-01083772

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. https://doi.org/https://doi.org/10.1016/j.apm.2019.10.006

COUSSY, O. (2004). *Poromechanics.* England, John Wiley & Sons.

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*. Récupérée 8 octobre 2022, à partir de 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. https://doi.org/10.1145/2833157.2833162

Latyshev, A. (2022). Convex-plasticity. GitHub.

Appendices

A Drucker-Prager material

$$f(\underline{\underline{\sigma}}, p) = \sigma_{\text{eq}} + \alpha \text{tr}\underline{\underline{\sigma}} - R(p) \leq 0 \tag{72}$$

$$\dot{\underline{\underline{\varepsilon}}}^p = \dot{\lambda}\frac{\partial f}{\partial \underline{\underline{\sigma}}} = \dot{\lambda}\left(\frac{3}{2}\frac{\underline{\underline{s}}}{\sigma_{\text{eq}}} + \alpha\underline{\underline{1}}\right) \tag{73}$$

$$\dot{p} = \sqrt{\frac{2}{3}\dot{\underline{\underline{\varepsilon}}}^p : \dot{\underline{\underline{\varepsilon}}}^p} = \dot{\lambda}\sqrt{1+2\alpha^2} \tag{74}$$

$$\Delta\underline{\underline{\varepsilon}}^p_{n} = \frac{\Delta p_n}{\sqrt{1+2\alpha^2}}\left(\frac{3}{2}\frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}} + \alpha\underline{\underline{1}}\right) \tag{75}$$

$$\Delta\underline{\underline{e}}^p_{n} = \frac{\Delta p_n}{\sqrt{1+2\alpha^2}}\frac{3}{2}\frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}} \tag{76}$$

$$\underline{\underline{\sigma}} = (3k\underline{\underline{\underline{\underline{J}}}} + 2\mu\underline{\underline{\underline{\underline{K}}}}) : (\underline{\underline{\varepsilon}} - \underline{\underline{\varepsilon}}^p) = \underline{\underline{\sigma}}^{\text{elas}}_{n+1} - (3k\underline{\underline{\underline{\underline{J}}}} + 2\mu\underline{\underline{\underline{\underline{K}}}}) : \underline{\underline{\varepsilon}}^p \tag{77}$$

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}^{\text{elas}}_{n+1} - 2\mu\Delta\underline{\underline{e}}^p_n - k\text{tr}(\Delta\underline{\underline{\varepsilon}}^p_n)\underline{\underline{1}} \tag{78}$$

$$\underline{\underline{s}}_{n+1} = \underline{\underline{s}}^{\text{elas}}_{n+1} - 2\mu\Delta\underline{\underline{e}}^p_n \tag{79}$$

$$\underline{\underline{s}}^{\text{elas}}_{n+1} = \underline{\underline{s}}_{n+1} + 3\mu\frac{\Delta p_n}{\sqrt{1+2\alpha^2}}\frac{\underline{\underline{s}}_{n+1}}{\sigma^{\text{eq}}_{n+1}} = \underline{\underline{s}}_{n+1}(1 + 3\mu\frac{\Delta p_n}{\sqrt{1+2\alpha^2}}\frac{1}{\sigma^{\text{eq}}_{n+1}}) \tag{80}$$

$$\sigma^{\text{elas,eq}}_{n+1} = \sigma^{\text{eq}}_{n+1}(1 + 3\mu\frac{\Delta p_n}{\sqrt{1+2\alpha^2}}\frac{1}{\sigma^{\text{eq}}_{n+1}}) \tag{81}$$

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

$$\sigma_{n+1}^{\text{eq}} = \sigma_{n+1}^{\text{elas,eq}} - \frac{3\mu}{\sqrt{1+2\alpha^2}}\Delta p_n \tag{83}$$

$$\text{tr}\underline{\underline{\sigma}}_{n+1} = \text{tr}\underline{\underline{\sigma}}_{n+1}^{\text{elas}} - 3\kappa \text{tr}\Delta\underline{\underline{\varepsilon}}_{n}^{p} \tag{84}$$

$$\text{tr}\Delta\underline{\underline{\varepsilon}}_{n}^{p} = \frac{3\alpha}{\sqrt{1+2\alpha^2}}\Delta p_n \tag{85}$$

$$\sigma_{n+1}^{\text{eq}} + \alpha \text{tr}\underline{\underline{\sigma}}_{n+1} - R(p_n + \Delta p_n) = 0 \tag{86}$$

$$R(p) = \sigma_0 + hp \tag{87}$$

$$\sigma_{n+1}^{\text{elas,eq}} - \frac{3\mu}{\sqrt{1+2\alpha^2}}\Delta p_n + \alpha \text{tr}\underline{\underline{\sigma}}_{n+1}^{\text{elas}} - 3\kappa\frac{3\alpha^2}{\sqrt{1+2\alpha^2}}\Delta p_n - \sigma_0 - hp_n - h\Delta p_n = 0 \tag{88}$$

$$\Delta p_n = \frac{\sigma_{n+1}^{\text{elas,eq}} + \alpha \text{tr}\underline{\underline{\sigma}}_{n+1}^{\text{elas}} - \sigma_0 - hp_n}{\frac{3\mu+9\alpha^2\kappa}{\sqrt{1+2\alpha^2}} + h} = \frac{\sigma_{n+1}^{\text{elas,eq}} + \alpha \text{tr}\underline{\underline{\sigma}}_{n+1}^{\text{elas}} - \sigma_0 - hp_n}{\gamma} \tag{89}$$

$$\gamma = \frac{3\mu + 9\alpha^2\kappa}{\sqrt{1+2\alpha^2}} + h \tag{90}$$

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}_{n+1}^{\text{elas}} - 2\mu\Delta\underline{\underline{e}}_{n}^{p} - \kappa \text{tr}(\Delta\underline{\underline{\varepsilon}}_{n}^{p})\underline{\underline{1}} \tag{91}$$

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma}}_{n+1}^{\text{elas}} - \frac{1}{\sqrt{1+2\alpha^2}}\left(\beta\underline{\underline{s}}_{n+1}^{\text{elas}} + 3k\alpha\Delta p_n\underline{\underline{1}}\right) \tag{92}$$

$$\beta = 3\mu\frac{\Delta p_n}{\sigma_{n+1}^{\text{elas,eq}}} \tag{93}$$

$$\frac{\partial \underline{\underline{s}}^{\text{elas}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} = 2\mu \underline{\underline{\underline{\underline{K}}}} \tag{94}$$

$$\frac{\partial \sigma^{\text{elas,eq}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} = \frac{3\mu}{\sigma^{\text{elas,eq}}_{n+1}} \underline{\underline{s}}^{\text{elas}}_{n+1} \tag{95}$$

$$\frac{\partial \text{tr}\sigma^{\text{elas,eq}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} = 3k\underline{\underline{1}} \tag{96}$$

$$2\mu \Delta \underline{\underline{e}}^{p}_{n} + k\text{tr}\Delta \underline{\underline{\varepsilon}}^{p}_{n} \underline{\underline{1}} = \frac{\Delta p_n}{\sqrt{1+2\alpha^2}} \left(3\mu \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} + 3k\alpha \underline{\underline{1}} \right) \tag{97}$$

$$\frac{\partial \Delta p_n}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} = \frac{1}{\gamma} \frac{\partial (\sigma^{\text{elas,eq}}_{n+1} + \alpha \text{tr}\underline{\underline{\sigma}}^{\text{elas}}_{n+1})}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} = \frac{1}{\gamma} \left(\frac{3\mu}{\sigma^{\text{elas,eq}}_{n+1}} \underline{\underline{s}}^{\text{elas}}_{n+1} + 3k\alpha \underline{\underline{1}} \right) = \frac{1}{\gamma} (3\mu \underline{\underline{n}}_{\text{elas}} + 3k\underline{\underline{1}}) \tag{98}$$

$$\frac{\partial \underline{\underline{\sigma}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} = \underline{\underline{\underline{\underline{C}}}}_{-} \frac{\partial (2\mu \Delta \underline{\underline{e}}^{p}_{n} + k\text{tr}\Delta \underline{\underline{\varepsilon}}^{p}_{n} \underline{\underline{1}})}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} = \underline{\underline{\underline{\underline{C}}}}_{-} \underline{\underline{\underline{\underline{D}}}} \tag{99}$$

$$\underline{\underline{\underline{\underline{D}}}} = \frac{1}{\sqrt{1+2\alpha^2}} \frac{\partial}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} \left(\Delta p_n \left(3\mu \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} + 3k\alpha \underline{\underline{1}} \right) \right) = \tag{100}$$

$$= \frac{1}{\sqrt{1+2\alpha^2}} \left(\frac{\partial \Delta p_n}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} \otimes \left(3\mu \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} + 3k\alpha \underline{\underline{1}} \right) + \Delta p_n \left(3\mu \frac{\partial \underline{\underline{s}}^{\text{elas}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} \frac{1}{\sigma^{\text{elas,eq}}_{n+1}} \right) - \Delta p_n 3\mu \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{(\sigma^{\text{elas,eq}}_{n+1})^2} \otimes \frac{\partial \sigma^{\text{elas,eq}}_{n+1}}{\partial \Delta \underline{\underline{\varepsilon}}_{n}} \right) = \tag{101}$$

$$= \frac{1}{\sqrt{1+2\alpha^2}} \left(\left(3\mu \underline{\underline{n}}_{\text{elas}} + 3\alpha k \underline{\underline{1}} \right) \frac{1}{\gamma} \otimes \left(3\mu \underline{\underline{n}}_{\text{elas}} + 3\alpha k \underline{\underline{1}} \right) + 2\mu\beta \underline{\underline{\underline{\underline{K}}}} - 3\mu\beta \underline{\underline{n}}_{\text{elas}} \otimes \underline{\underline{n}}_{\text{elas}} \right) \tag{102}$$

$$\underline{\underline{\underline{\underline{D}}}} = \frac{1}{\sqrt{1+2\alpha^2}} \left(3\mu \left(\frac{3\mu}{\gamma} - \beta \right) \underline{\underline{n}}_{\text{elas}} \otimes \underline{\underline{n}}_{\text{elas}} + \frac{9\alpha\mu k}{\gamma} (\underline{\underline{1}} \otimes \underline{\underline{n}}_{\text{elas}} + \underline{\underline{n}}_{\text{elas}} \otimes \underline{\underline{1}}) + \frac{9\alpha^2 k^2}{\gamma} \underline{\underline{1}} \otimes \underline{\underline{1}} + 2\mu\beta \underline{\underline{\underline{\underline{K}}}} \right) \tag{103}$$

$$\underline{\underline{\underline{\underline{D}}}} : \Delta \underline{\underline{\varepsilon}}_{n} = \tag{104}$$

$$= \frac{1}{\sqrt{1+2\alpha^2}} \left(\underline{\underline{n}}_{\text{elas}} : \Delta \underline{\underline{\varepsilon}}_{n} 3\mu \left(\frac{3\mu}{\gamma} - \beta \right) \underline{\underline{n}}_{\text{elas}} + \frac{9\alpha\mu k}{\gamma} (\underline{\underline{n}}_{\text{elas}} : \Delta \underline{\underline{\varepsilon}}_{n} \underline{\underline{1}} + \text{tr}\Delta \underline{\underline{\varepsilon}}_{n} \underline{\underline{n}}_{\text{elas}}) + \frac{9\alpha^2 k^2}{\gamma} \text{tr}\Delta \underline{\underline{\varepsilon}}_{n} \underline{\underline{1}} + 2\mu\beta \Delta \underline{\underline{e}}^{p}_{n} \right) \tag{105}$$

$$\tag{106}$$

$$F = \int_\Omega \underline{\underline{\sigma}}_{n+1} : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}\Omega - \underline{F}_{\text{ext}} = \tag{107}$$

$$= \int_\Omega \left(\underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : (\Delta \underline{\underline{\varepsilon}}_n - \Delta \underline{\underline{\varepsilon}}_n^p) \right) : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}\Omega - \underline{F}_{\text{ext}} \tag{108}$$

where $\underline{F}_{\text{ext}} = q \int_{\partial\Omega_{\text{inside}}} \underline{n} \cdot \underline{v} \, \mathrm{d}s$, $\Delta \underline{\underline{\varepsilon}}_n = \underline{\underline{\varepsilon}}(\Delta \underline{u}_n)$ and $\Delta \underline{\underline{\varepsilon}}_n = \underline{\underline{\varepsilon}}^p(\Delta \underline{u}_n)$

$$\underline{\underline{\varepsilon}}(\Delta \underline{u}_n) = \frac{1}{2} \left(\nabla \Delta \underline{u} + \nabla \Delta \underline{u}^T \right) \tag{109}$$

$$\underline{\underline{\varepsilon}}^p(\Delta \underline{u}_n) = \begin{cases} \Delta p_n \left(\frac{3}{2} \frac{\underline{\underline{s}}^{\text{elas}}_{n+1}}{\sigma^{\text{elas,eq}}_{n+1}} + \alpha \underline{\underline{1}} \right), & \text{if } f(\underline{\underline{\sigma}}^{\text{elas}}, p_n) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{110}$$

where $\Delta p_n = p_{n+1} - p_n$

$$\underline{\underline{\varepsilon}}^p(\Delta \underline{u}) = \underline{\underline{\varepsilon}}^p(\Delta \underline{u}, p_n, p_{n+1}, \underline{\underline{\sigma}}_n) \tag{111}$$

$$F(\Delta \underline{u}, \underline{v}) = \int_\Omega \left(\underline{\underline{\sigma}}_n + \underline{\underline{\underline{\underline{C}}}} : (\underline{\underline{\varepsilon}}(\Delta \underline{u}) - \underline{\underline{\varepsilon}}^p(\Delta \underline{u})) \right) : \underline{\underline{\varepsilon}}(\underline{v}) \, \mathrm{d}\Omega - \underline{F}_{\text{ext}} \tag{112}$$

$$J(\underline{u}, \underline{v}) = \frac{\partial F(\Delta \underline{u}, \underline{v})}{\partial \Delta \underline{u}}(\underline{u}) \tag{113}$$

```python
    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_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
        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]
            dp = ppos(f_elas)/(3*mu_+H)

            if f_elas >= 0:
                n_elas[:,:] = s/sig_eq*ppos(f_elas)/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

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

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