Expressing general constitutive models using algorithmic automatic differentiation in DOLFINx

A. Latyshev[1,2] J. Bleyer[3] J. S. Hale[1] C. Maurini[2]

[1]University of Luxembourg, Luxembourg

[2]Sorbonne Université, France

[3]Laboratoire Navier, École des Ponts, Université Gustave Eiffel, CNRS, France

June 13, FEniCS 2024

Expressing solid mechanics problems in UFL

Weak form of an equilibrium problem in solid mechanics:

$$F(\boldsymbol{u}; \boldsymbol{v}) = \int\limits_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) \cdot \boldsymbol{\varepsilon}(\boldsymbol{v}) \mathrm{d}\boldsymbol{x} = 0, \quad \forall \boldsymbol{v} \in V, \tag{1}$$

where $\boldsymbol{\varepsilon}(\boldsymbol{v}) = \frac{1}{2}(\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T)$.

Expressing the form via UFL:

```
F = ufl.inner(sigma(u), epsilon(v)) * ufl.dx
```

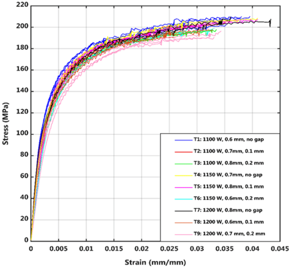
UFL limitations

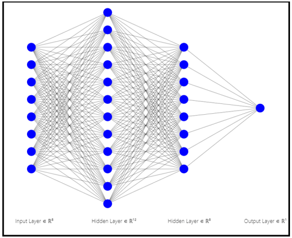
What if $\boldsymbol{\sigma}(\boldsymbol{u})$ is not expressible via analytical formulas, i.e. in UFL?

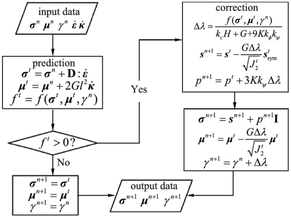
Issue: UFL is limited to express any constitutive model

Ways to express constitutive models $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\boldsymbol{u})$ in UFL:

- ✅ $\boldsymbol{\sigma}$ = analytical formula
- ❌ $\boldsymbol{\sigma}$ = experimental data
- ❌ $\boldsymbol{\sigma}$ = neural network
- ❌ $\boldsymbol{\sigma}$ = algorithm
- ❌ and so on...







$$
\begin{cases}
\dfrac{\partial f_{\varepsilon^{\text{el}}}}{\partial \Delta \varepsilon^{\text{el}}} = \underline{\underline{\mathbf{I}}} + \Delta\lambda \dfrac{\partial \underline{n}}{\partial \Delta \underline{\epsilon}_{el}} \\
\dfrac{\partial f_{\varepsilon^{\text{el}}}}{\partial \Delta p} = \underline{n} \\
\dfrac{\partial f_p}{\partial \Delta \varepsilon^{\text{el}}} = E^{-1} \underline{n}_F : \underline{\underline{\mathbf{C}}} \\
\dfrac{\partial f_p}{\partial \Delta p} = 0
\end{cases}
$$

Solution: expressing any constitutive model

We propose a framework that:

1. extends FEniCSx/DOLFINx and allows to use **any** 3rd-party library to define constitutive models $\boldsymbol{\sigma}(\boldsymbol{u})$ as a part of weak problems;
2. uses **NumPy arrays** to pass data between DOLFINx and external libraries;
3. is based on two concepts: **external operator**[1] and **automatic differentiation**;

[1] Nacime Bouziani and David A. Ham. *Escaping the abstraction: a foreign function interface for the Unified Form Language [UFL]*. 2021. arXiv: 2111.00945 [cs.MS].

What is an external operator?

An **external operator**[2] $N(\cdot)$ is a *symbolic* UFL object that

- is defined by a *computer program*, not by an analytical formula,
- *differentiable* and it's derivative $\partial N(\cdot)$ is another **external operator**.

Example:

$$F = F(u, N(u); v), \quad u, v \in V, \tag{2}$$

The Gâteaux derivative of F with respect to u in the direction $\hat{u} \in V$:

$$J = \mathfrak{D}_u F(\hat{u}) = \partial_u F(\hat{u}) + \partial_N F(\partial_u N(\hat{u})), \quad u, v, \hat{u} \in V, \tag{3}$$

where $\mathfrak{D}_u\{\cdot\}(\hat{u})$ and $\partial_u\{\cdot\}(\hat{u})$ are respectively total and partial Gâteaux derivatives with respect to operand u in the direction $\hat{u}$.

[2] Nacime Bouziani and David A. Ham. *Escaping the abstraction: a foreign function interface for the Unified Form Language [UFL]*. 2021. arXiv: 2111.00945 [cs.MS].

External operators in solid mechanics

The constitutive model $\boldsymbol{\sigma}(\boldsymbol{u}) := \boldsymbol{\sigma}(\varepsilon(\boldsymbol{u}))$ is an **external operator** acting on the strain tensor $\varepsilon(\boldsymbol{u})$.

To create an external operator in the FEniCSx environment, you need to use the class `FEMExternalOperator` of our framework:

```
sigma = FEMExternalOperator(
    epsilon(u), # operand
    function_space=S, # quadrature space
    external_function=sigma_external # a Python function
)
```

The derivative $\frac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}} = \boldsymbol{C}_{\mathrm{tang}}(\varepsilon(\boldsymbol{u}))$, the *consistent tangent stiffness matrix*, is **another external operator** and created via the UFL's automatic differentiation.

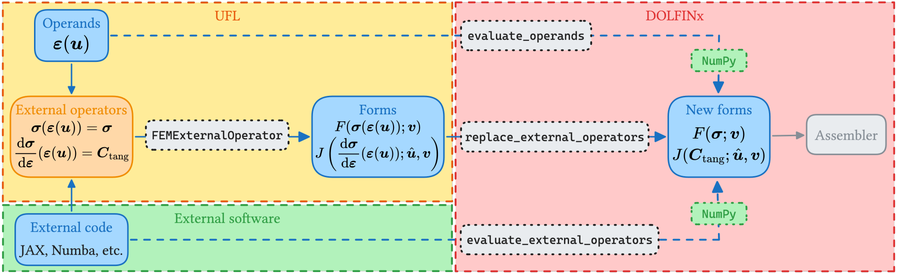
How to define external operators

```python
def sigma_impl(deps: np.ndarray) -> np.ndarray:
    ...
    return sigma_values

def C_tang_impl(deps: np.ndarray) -> np.ndarray:
    ...
    return C_tang_values

def sigma_external(
    derivatives: Tuple[int, ...]
) -> Callable[[np.ndarray], np.ndarray]:

    if derivatives == (0,):
        return sigma_impl
    if derivatives == (1,):
        return C_tang_impl
```

Escaping FEniCSx environment

Functions above may contain **any** 3rd-party code working with NumPy arrays.

Framework workflow



Plasticity of Mohr-Coulomb

Let's consider the following weak problem for $\boldsymbol{u} \in V$:

$$F(\boldsymbol{u};\boldsymbol{v}) = \int\limits_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) \cdot \varepsilon(\boldsymbol{v})\mathrm{d}\boldsymbol{x} - F_{\text{ext}}(\boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V, \tag{4}$$

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

Non-associated *Mohr-Coulomb* plasticity with apex smoothing:

$$h(\boldsymbol{\sigma}, \alpha) = \frac{I_1(\boldsymbol{\sigma})}{3}\sin\alpha + \sqrt{J_2(\boldsymbol{\sigma})K^2(\alpha) + a^2(\alpha)\sin^2\alpha} - c\cos\alpha, \tag{6}$$

$$f(\boldsymbol{\sigma}) = h(\boldsymbol{\sigma}, \phi), \text{ - yield surface} \tag{7}$$

$$g(\boldsymbol{\sigma}) = h(\boldsymbol{\sigma}, \psi), \text{ - plastic potential} \tag{8}$$

where $J_2(\boldsymbol{\sigma}) = \frac{1}{2}\boldsymbol{s} \cdot \boldsymbol{s}$ is the second invariant of the deviatoric part $\boldsymbol{s}$ of the stress tensor and F_{ext} represents external forces.

How to solve 4-5: apply a *return-mapping* procedure, a **numerical algorithm**.

Reutrn-mappping procedure

Constitutive equations in plasticity:

$$\begin{cases} \boldsymbol{r}_g(\boldsymbol{\sigma}_{n+1}, \Delta\lambda) = \boldsymbol{\sigma}_{n+1} - \boldsymbol{\sigma}_n - \boldsymbol{C} \cdot (\Delta\boldsymbol{\varepsilon} - \Delta\lambda \frac{\mathrm{d}g}{\mathrm{d}\boldsymbol{\sigma}}(\boldsymbol{\sigma}_{\boldsymbol{n+1}})) = \boldsymbol{0}, \\ r_f(\boldsymbol{\sigma}_{n+1}) = f(\boldsymbol{\sigma}_{n+1}) = 0, \end{cases} \tag{9}$$

Return-mapping procedure is a numerical algorithm solving the system 9 by following a predictor-corrector scheme for the stress tensor $\boldsymbol{\sigma}$.

In the general case, e.g. the *Mohr-Coulomb* case, the return-mapping requires solving the nonlinear system 9 **numerically**.

Plasticity problems in FEniCSx via JAX

Our solution: implementation of the *return-mapping* procedure using external operators via package `JAX`

`JAX` is a high-level library for automatic differentiation (AD) and numerical computing, which also supports the just-in-time compilation (JIT) feature.

Automatic differentiation (AD) in solid mechanics via JAX

Consistent tangent stiffness matrix

$$\frac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}} = \boldsymbol{C}_{\text{tang}}(\boldsymbol{\varepsilon}(\boldsymbol{u})) \tag{10}$$

Implementation via JAX:

```python
def sigma_impl(deps: np.ndarray) -> np.ndarray:
    ...
    "<return-mapping algorithm>"
    ...
    return sigma_

C_tang_impl = jax.jacfwd(sigma_impl)
```

The function `C_tang_impl` evaluate the consistent tangent stiffness matrix $\boldsymbol{C}_{\text{tang}} = \frac{\mathrm{d}\boldsymbol{\sigma}}{\mathrm{d}\boldsymbol{\varepsilon}}$ **exactly** at Gauss points.

Mohr-Coulomb plasticity via JAX

Constitutive equations in plasticity:

$$\begin{cases} \boldsymbol{r}_g(\boldsymbol{\sigma}_{n+1}, \Delta\lambda) = \boldsymbol{\sigma}_{n+1} - \boldsymbol{\sigma}_n - \boldsymbol{C} \cdot (\Delta\boldsymbol{\varepsilon} - \Delta\lambda \frac{\mathrm{d}g}{\mathrm{d}\boldsymbol{\sigma}}(\boldsymbol{\sigma}_{\boldsymbol{n+1}})) = \boldsymbol{0}, \\ r_f(\boldsymbol{\sigma}_{n+1}) = f(\boldsymbol{\sigma}_{n+1}) = 0. \end{cases} \tag{11}$$

Inner Newton loop:

```
def return_mapping(
    deps_local: np.ndarray,
    sigma_n_local: np.ndarray
) -> ... :
    ...
    ... = jax.lax.while_loop(...)
    ...
    return sigma_local, (sigma_local, niter_total, yielding, norm_res,
    dlambda)
```

The program that evaluates stress $\boldsymbol{\sigma}$ AND $\boldsymbol{C}_{\text{tang}} = \frac{\mathrm{d}\boldsymbol{\sigma}}{\mathrm{d}\boldsymbol{\varepsilon}}$ **exactly** at Gauss points:

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

Mohr-Coulomb plasticity via JAX: vectorization

Defining the external operator and its derivative through the **vectorization** over quadrature points (via `jax.vmap`):

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

Globally:

```
def C_tang_impl(deps: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    deps_ = deps.reshape((-1, 6))
    sigma_n_ = sigma_n.x.array.reshape((-1, 6))

    (C_tang_global, state) = dsigma_ddeps_vec(deps_, sigma_n_)
    sigma_global, ... = state
    ...
    return C_tang_global.reshape(-1), sigma_global.reshape(-1)
```

Plasticity before AD and JAX

Constitutive equations in plasticity:

$$\begin{cases} \boldsymbol{r}_g(\boldsymbol{\sigma}_{n+1}, \Delta\lambda) = \boldsymbol{\sigma}_{n+1} - \boldsymbol{\sigma}_n - \boldsymbol{C} \cdot (\Delta\boldsymbol{\varepsilon} - \Delta\lambda \frac{\mathrm{d}g}{\mathrm{d}\boldsymbol{\sigma}}(\boldsymbol{\sigma}_{\boldsymbol{n+1}})) = \boldsymbol{0}, \\ r_f(\boldsymbol{\sigma}_{n+1}) = f(\boldsymbol{\sigma}_{n+1}) = 0, \end{cases} \tag{12}$$

where $g(\boldsymbol{\sigma}) = h(\boldsymbol{\sigma}, \psi)$ is a plastic potential.

Instead of

```
n = dgdsigma = jax.jacfwd(g)
```

we had to manually derive[3]:

$$\mathbf{n} = \frac{\partial G_F}{\partial I_1}\mathbf{I} + \left(\frac{\partial G_F}{\partial J_2} + \frac{\partial G_F}{\partial \theta}\frac{\partial \theta}{\partial J_2}\right)\sigma^{\mathbf{D}} + \frac{\partial G_F}{\partial \theta}\frac{\partial \theta}{\partial J_3} J_3(\sigma^{\mathbf{D}})^{-1} : \mathbf{P}^{\mathbf{D}},$$

$$\ldots$$

[3] Thomas Helfer et al. *Invariant-based implementation of the Mohr-Coulomb elasto-plastic model in OpenGeoSys using MFront.* TFEL/MFront. URL: https://thelfer.github.io/tfel/web/MohrCoulomb.html (visited on 05/10/2024).

Plasticity before AD and JAX

Instead of

```
drdx = jax.jacfwd(r)
j = drdx(x_local, deps_local, sigma_n_local)
...
dsigma_ddeps = jax.jacfwd(sigma_return_mapping, has_aux=True)
```

we had to manually derive:[4]

$$\begin{cases} \dfrac{\partial r_{\varepsilon^{el}}}{\partial \Delta \varepsilon^{el}} = \mathbf{I} + \Delta\lambda \dfrac{\partial \mathbf{n}}{\partial \Delta \varepsilon^{el}} \\ \dfrac{\partial r_{\varepsilon^{el}}}{\partial \Delta p} = \mathbf{n} \\ \dfrac{\partial r_p}{\partial \Delta \varepsilon^{el}} = \mathbf{E}^{-1}\mathbf{n}_F : \mathbf{C} \\ \dfrac{\partial r_p}{\partial \Delta p} = 0 \end{cases} -> \boldsymbol{j} = \begin{pmatrix} \frac{\partial r_{\varepsilon_{el}}}{\partial \Delta \varepsilon_{el}} & \frac{\partial r_{\varepsilon_{el}}}{\partial \Delta p} \\ \frac{\partial r_p}{\partial \Delta \varepsilon_{el}} & \frac{\partial r_p}{\partial \Delta p} \end{pmatrix} -> \begin{matrix} \dfrac{\mathrm{d}\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))}{\mathrm{d}\boldsymbol{\varepsilon}} = \\ = [(\boldsymbol{j}^{-1})_{i,j}]\boldsymbol{C}_{\text{elas}}, \\ i,j = 1, ..., k, \\ \boldsymbol{C}_{\text{elas}} \in M_{k\times k} \end{matrix} \tag{13}$$

[4] Thomas Helfer et al. *Invariant-based implementation of the Mohr-Coulomb elasto-plastic model in OpenGeoSys using MFront.* TFEL/MFront. URL: https://thelfer.github.io/tfel/web/MohrCoulomb.html (visited on 05/10/2024).

Verification of return-mapping procedure

Yield surface of Mohr-Coulomb with apex smoothing:

$$f(\boldsymbol{\sigma}, \phi) = \frac{I_1(\boldsymbol{\sigma})}{3} \sin \phi + \sqrt{J_2(\boldsymbol{\sigma}) K^2(\phi) + a^2(\phi) \sin^2 \phi} - c \cos \phi, \qquad (14)$$

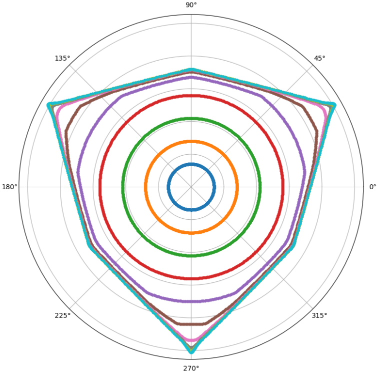


Figure: The yield surface tracing through passing several stress paths within the deviatoric plane (ρ, θ), where $\rho = \sqrt{2J_2}$ and θ is a Lode angle.

Verification of derivatives via Taylor test

$$R_0 = |F(\boldsymbol{u} + h\,\boldsymbol{\delta u}; \boldsymbol{v}) - F(\boldsymbol{u}; \boldsymbol{v})| \longrightarrow 0 \text{ at } O(h), \tag{15}$$

$$R_1 = |F(\boldsymbol{u} + h\,\boldsymbol{\delta u}; \boldsymbol{v}) - F(\boldsymbol{u}; \boldsymbol{v}) - J(\boldsymbol{u}; h\boldsymbol{\delta u}, \boldsymbol{v})| \longrightarrow 0 \text{ at } O(h^2), \tag{16}$$

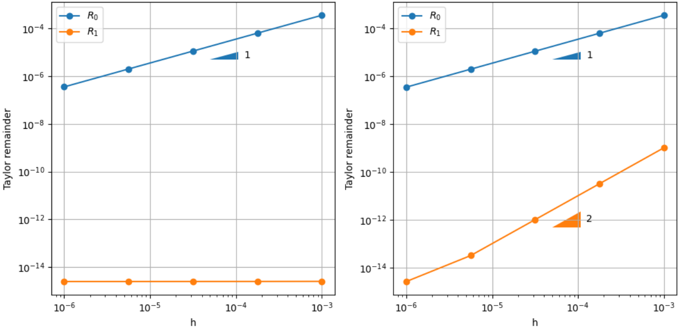


Figure: Taylor test for the form F. There are the zeroth-order R_0 and the first-order R_1 Taylor remainders for the elastic phase (on the left) and the plastic phase (on the right).

What other problems to solve?

Some examples of constitutive models where the framework can be useful:

- $\boldsymbol{\sigma} = \text{return\_mapping}(\boldsymbol{u})$
- $\boldsymbol{\sigma} = \text{another\_numerical\_algorithm}(\boldsymbol{u})$
- $\boldsymbol{\sigma} = \text{FE}^2(\boldsymbol{u})$
- $\boldsymbol{\sigma} = \text{neural\_network}(\boldsymbol{u})$
- $\boldsymbol{\sigma} = \text{experimental\_data}(\boldsymbol{u})$
- $\boldsymbol{\sigma} = \text{call\_to\_existing\_material\_library}(\boldsymbol{u})$
- $\boldsymbol{\sigma} = \text{convex\_solver}(\boldsymbol{u})$
- $\boldsymbol{\sigma} = \text{surrogate\_model}(\boldsymbol{u})$
- ...

Extending DOLFINx

The framework goes beyond the solid mechanics:
It enables the support of automatic differentiation in DOFLINx via JAX!

The framework can help to integrate in DOLFINx other interesting packages and techniques:

❤️ = DOLFINx + NumPy +
- ✅ Numba (JIT compilation, vectorization),
- ✅ JAX (AD, JIT compilation, vectorization),
- 🟨 PyTorch (neural networks),
- ❓ what else?,
- ...

Conclusion

1. We implemented a **framework** extending DOLFINx and providing a special **interface** to FEniCSx users.
2. This **interface** allows to use of **any** 3rd-party library to define constitutive models as a part of weak problems.
3. In particular, it enables support of general **automatic differentiation (AD)** in FEniCSx via the **JAX** library.
4. **AD** is a very effective and robust tool to compute derivatives defined by computer programs. This is very beneficial in the context of constitutive models.
5. **Just-in-time (JIT)** compilation guarantees efficient implementation of a constitutive model within the framework.

Tutorials and how to use (save the link via QR code!):

⭐ => a-latyshev.github.io/dolfinx-external-operator/

Any questions? Contact me!

Andrey Latyshev : andrey.latyshev@uni.lu

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant reference PRIDE/21/16747448/MATHCODA.