CSMA 2024
16ème Colloque National en Calcul des Structures
13-17 Mai 2024, Presqu'île de Giens (Var)

A framework for expressing general constitutive models in FEniCSx

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

[1] *Institute of Computational Engineering, Department of Engineering, Université du Luxembourg, Luxembourg, {jack.hale, andrey.latyshev}@uni.lu*
[2] *CNRS, UMR 7190, Institut Jean Le Rond d'Alembert, Sorbonne Université, France, corrado.maurini@sorbonne-universite.fr*
[3] *CNRS, Laboratoire Navier, Ecole des Ponts, Université Gustave Eiffel, France, jeremy.bleyer@enpc.fr*

Résumé — Many important problems in solid mechanics involve non-trivial constitutive models that are difficult to express in variational form. It is therefore challenging to express these problems in domain-specific languages that work at the variational form level. We introduce a framework for FEniCSx / DOLFINx that allows for the straightforward implementation of a wide range of constitutive models. The application of the framework is demonstrated by implementing a von Mises elastoplastic material model with hardening using JAX and Numba software.
Mots clés — constitutive models, external operator, automated finite element solvers.

1 Introduction

The finite element method (FEM) is a robust and widely used numerical approach for solving partial differential equations (PDEs) that describe many problems in solid mechanics. In the past decade, automated PDE solvers such as FEniCS [1, 8], FreeFEM++ [11] and Firedrake [9] have introduced high-level domain-specific languages (DSL), e.g. the Unified Form Language (UFL) [2] for writing variational forms of PDEs. From the DSL these solvers can automatically generate high-performance parallel finite element code through a sequence of transformation and compilation steps. However, there are a large number of problems in solid mechanics that are not naturally expressed or implemented using the algebraic primitives provided in the UFL, such as complex plasticity models, multiscale and neural-network-based constitutive [17] models. In addition, specialised programming tools for specifying or solving non-standard constitutive models such as MFront [12] and CVXPY [6] cannot straightforwardly be incorporated into finite element solvers with automatic code generation capabilities. Consequently, the adoption of automated tools for solving PDEs in the solid mechanics community has been held back by the inherent limitations imposed by the available abstractions.

The objective of this work is to design a general framework that extends DOLFINx [3], the new finite element solver of the FEniCS Project, in such a way that arbitrary constitutive models (e.g. plasticity, neural network, multiscale/homogenisation) can be straightforwardly implemented via a wide variety of flexible programming tools (e.g. pure Python, JAX [7], Numba [15] or other external packages). Our approach is the synthesis of three recent developments :

1. The recently introduced `ExternalOperator` extension to UFL as described in [10]. In essence `ExternalOperator` allows the user to write a symbolic representation of an arbitrary unspecified operator between UFL operands (e.g. strain) and coefficients (e.g. stress). When UFL forms involving `ExternalOperator` are differentiated using UFL's symbolic differentiation tools, new `ExternalOperator` objects of the appropriate shape and rank are automatically generated. The authors of [10] provide an implementation of the `ExternalOperator` in the Firedrake [9] solver, but do not explore its use in solid mechanics constitutive modelling setting or its use in DOLFINx.
2. The data-centric design of the new DOLFINx library, where data such as finite element function coefficients are directly available in `ndarray`-like data structures. This data-centric design makes it straightforward to write external operators in packages that support `ndarray`-like data structures such as Numba, JAX, TensorFlow and PyTorch, or external libraries like MFront or CVXPY.
3. The addition of automatic code generation features [8] to DOLFINx for evaluating UFL operands (e.g. strain) at a set of pre-defined points on the reference finite element cell. The data computed

by these routines is then passed as `ndarrays`-like objects to the user definition of the external operator, as discussed in step 2. above.

We demonstrate the effectiveness of this approach by implementing the widely-known von Mises elastoplastic behaviour using Numba and JAX, with the latter example leveraging JAX's powerful automatic differentiation capabilities to avoid explicit by-hand derivation of the tangent stiffness. As a point of comparison, we implement an interpolation-based approach originally proposed in [4] and then improved in FEniCSx [16]. The three implementations provide substantially similar runtime performance in addition to being transparent and easy to follow.

Our contribution to automated FEM solvers capable of including complex constitutive models in the variational form is not the first one. One of the earliest examples is the commercial finite element environment AceFEM [14] equipped with the automatic code generation package AceGEN [13]. The latter can symbolically evaluate finite element quantities as well as generate automatic code in C and FORTRAN using the commercial Mathematica language. Besides, other projects tried to extend the functionality of FEniCS for problems with complex constitutive models, such as a special interface [12] between MFront [12] and the legacy version of FEniCS, the fenics-solid-mechanics project [18] designed specifically for plasticity problems within C++ interface and the application of the 3rd-party package CVXPY [6] for solving plasticity problems in the convex optimization setting within the FEniCSx environment [16]. Our framework is an open-source extension of the modern FEniCSx that combines previous ideas in a more general way via a compact Python interface.

2 Materials and methods

The proposed extension of FEniCSx is based on the external operator functionality recently proposed in [10]. `ExternalOperator` represents a symbolic object wrapping an external operator that is not expressible through standard UFL expressions. This object can be used directly in UFL forms. Moreover, `ExternalOperator` is equipped with the ability to be automatically differentiated. UFL automatically propagates derivatives of the external operator in a variational form according to the chain rule.

In our DOLFINx implementation of the external operator concept, we diverge from the Firedrake implementation in the sense there is no direct interaction between the form assembly and the user's implementation of the external operator. Instead, we focus on the direct passing of data and their derivatives represented by external operators as `ndarray`-like objects. This is achieved by replacing the symbolic `ExternalOperator` objects with DOLFINx `Function` objects prior to assembly of the finite element form. These user's external operators are expressed as Python callables. They represent an algorithm describing how an external operator acts on their operands (e.g. strains). Our framework is able to assemble these coefficients using the external operator functions before the coefficients are passed to the standard DOLFINx finite element assembly routines.

Whereas the external operators can be considered as constitutive models, their operands represent the main variables of these models or based on them UFL expressions. According to our framework, all operands must be calculated at interpolation points of a finite element space. The FEniCSx Form Compiler (FFCx) [8] is equipped with automatic code generation for any expression written in UFL. FFCx just-in-time compiles a C code of the UFL expression and then DOLFINx can use this compiled code to compute the values of the expression at a set of points in the reference cell.

Data transfer between the user-defined functions for computing external operators of their operands is carried out through standard NumPy arrays, allowing a wide range of Python and non-Python-based tools to be used for implementing external operators. The design does not require the user to implement or extend Python objects. To reach the highest level of performance modern code generation tools for Python such as Numba and JAX can be applied, or indeed any other tool which accepts `ndarray` or C-array-like objects.

Summing up, the algorithm consists of the following steps :

1. For each external operator define its and its derivatives's explicit implementations in the form of Python callables.
2. Create matching `ExternalOperator` objects.
3. Using UFL write a linear form containing one or more `ExternalOperator` objects.

4. Automatically derive a bilinear form through derivation of the linear form defined in the previous step.
5. Create two new UFL forms where `ExternalOperator` is replaced with DOLFINx `Function` objects to hold the result of the evaluated external operator and its derivatives.
6. Evaluate the operands of the external operators at appropriate interpolation points.
7. Evaluate the external operators.
8. Assemble the evaluated external operator into the corresponding `Function`.

3 Results and discussion

In order to show how our framework can be used in multiple ways, we chose a simple example of an elastoplastic problem of cylinder expansion. The problem is solved in the two-dimensional case in a symmetric formulation. The full description of the problem can be found here [4]. We limit ourselves to the weak formulation 1 of the cylinder expansion problem, where we find $\underline{u} \in V$ such that

$$R(\underline{u}) = \int\limits_{\Omega} \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) \,\mathrm{d}x - F_{\text{ext}}(\underline{v}) = 0, \quad \forall \underline{v} \in V, \tag{1}$$

where $\underline{v}$ is a test function in a suitable finite element function space V and $\underline{\underline{\varepsilon}}$ is the usual infinitesimal strain tensor. The relation between stress tensor $\underline{\underline{\sigma}}$ and the displacement field $\underline{u}$ is defined according to the von Mises elastoplastic constitutive model with isotropic hardening law. F_{ext} represents the external force acting on the cylinder's inner surface. The force F_{ext} progressively increases up to the analytical collapse load for perfectly plastic material.

The most common way of solving elastoplastic problems numerically is to perform the return-mapping procedure. In this particular example, explicit expressions of stress and strain decrements can be derived analytically, in addition to the derivative of the stress tensor with respect to the strain tensor, commonly called the tangent stiffness matrix [5].

The possibility of performing the return-mapping procedure analytically makes this simple example ideal for demonstrating the core idea of this framework. On the one hand, as a point of comparison, it allows a UFL-based implementation and calculation of all quantities of interest, on the other hand, it is possible to solve the problem using more general approaches involving the application of our framework in a number of different ways.

At first, an approach based on UFL and FEniCSx functionality will be covered (the interpolation approach) and then we will describe the approaches based on the extended UFL using the external operator concept and modern Python-based libraries (the Numba approach and the JAX approach). All descriptions are supported with minimal code snippets.

The simplicity of the cylinder expansion problem makes it possible to solve the nonlinear elastoplastic problem without recourse to packages outside the FEniCSx environment. According to the interpolation approach we exploit the knowledge about mathematical expressions and interpolate some variables of the return-mapping procedure written through UFL's `Expression` objects over quadratures finite space. In the listing 1 you may find the main expressions of the problem written via UFL, which will be interpolated on each iteration of the Newton solver.

```python
def proj_sig(deps, sigma_old, p_old):
    """Performs the predictor-corrector return-mapping algorithm."""
    sig_n = as_3D_tensor(sigma_old)
    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*p_old
    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

sig_, n_elas_, beta_, dp_ = proj_sig(deps, sigma_old, p)
```

Listing 1: Definition of some analytical expressions of the return-mapping procedure via UFL.

Now we demonstrate how to solve the cylinder expansion problem using our framework and implementing the external operators using Numba, but firstly let us define the weak problem 1 using UFL and `ExternalOperator` in the following snippet :

```python
def sigma_ext(derivatives):
    if derivatives == (0,):
        return func_numba_sigma
    elif derivatives == (1,):
        return func_numba_C_tang
    else:
        return NotImplementedError
sigma_ex_operator = ufl.ExternalOperator(epsilon, function_space=W,
        local_operands=(epsilon,), f=sigma_ext)
R = ufl.inner(eps(v_), as_3D_tensor(sigma_ex_operator))*dx - F_ext(v_)
J = ufl.derivative(R, u, u_)
```

Listing 2: Definition of the weak problem in terms of UFL and `ExternalOperator`, where `epsilon` is an operand according to which the derivative of the operator is taken. `sigma_ext` is a callable Python function that contains definitions of how the external operator and its derivative are computed.

In the following listing 3 all mathematical expressions are written through Numba just-in-time compiled functions whose arguments are `ndarray` arrays. Numba typically produces highly optimised machine code with runtime performance on the level of traditional compiled languages.

```python
@numba.njit
def func_numba_sigma(deps, sigma, sigma_old, p_old, dp):
    return sigma

@numba.njit
def func_numba_C_tang(deps, sigma, sigma_old, p_old, dp):
    out = np.zeros((num_cells, num_gauss_points, 4, 4), dtype=PETSc.ScalarType)
    sigma_global = sigma.reshape((num_cells, num_gauss_points, 4))
    deps_global = deps.reshape((num_cells, num_gauss_points, 4))
    sigma_old_global = sigma_old.reshape((num_cells, num_gauss_points, 4))
    p_old_global = p_old.reshape((num_cells, num_gauss_points))
    dp_global = dp.reshape((num_cells, num_gauss_points))

    for i in range(0, num_cells):
        deps_local = deps_global[i]
        sigma_local = sigma_global[i]
        sigma_old_local = sigma_old_global[i]
        p_old_local = p_old_global[i]
        dp_local = dp_global[i]
        for q in range(num_gauss_points):
            sig_elas = sigma_old_local[q] + C_elas @ deps_local[q]
            s = DEV_Voigt @ sig_elas
            sig_eq = np.sqrt(3./2. * np.dot(s, s))

            f_elas = sig_eq - sig0 - H*p_old_local[q]
            f_elas_plus = ppos(f_elas)

            dp_local[q] = f_elas_plus/(3*mu_+H)

            n_elas = s/sig_eq*f_elas_plus/f_elas
            beta = 3*mu_ * dp_local[q] / sig_eq

            new_sigma = sig_elas - beta*s
            sigma_local[q][:] = new_sigma

            C_tang = get_C_tang(beta, n_elas)

            out[i][q][:,:] = C_tang

    return out.reshape(-1)
```

Listing 3: Definition of the external operator and its derivative via Numba. The function `func_numba_sigma` just returns values of the stress tensor, whereas `func_numba_C_tang` updates its values and computes the tangent stiffness tensor `C_tang` according to the analytical expressions of return-mapping procedure on each cell.

Both of the previous approaches are based on the knowledge of the explicit expression of the tangent stiffness matrix. It is possible to avoid such a derivation by using automatic differentiation tools of the JAX library. In the following listing 4, we define the functions computing the values of stress tensor according to the return-mapping procedure using JAX.

```python
@jax.jit
def deps_p(deps_local, sigma_old_local, p_old_local, dp_local):
    sig_elas = sigma_old_local + C_elas @ deps_local
    s = DEV_Voigt @ sig_elas
    sig_eq = jnp.sqrt(3./2. * jnp.vdot(s, s))

    f_elas = sig_eq - sig0 - H*p_old_local
    f_elas_plus = jax_ppos(f_elas)

    dp_local = f_elas_plus/(3*mu_+H)
    out = 3./2. * dp_local * s/sig_eq
    return out, dp_local

@jax.jit
def sig_jax(deps_local, sigma_old_local, p_old_local, dp_local):
    deps_p_local, dp_local_new = deps_p(deps_local, sigma_old_local, p_old_local, dp_local)
    sigma_local = sigma_old_local + C_elas @ (deps_local - deps_p_local)
    return sigma_local, dp_local_new
```

Listing 4: Definition of the plastic strain decrement and corrected stress tensor according to the analytical expressions of the return-mapping procedure via JAX.

Similar to the Numba approach, the `@jax.jit` decorator just-in-time compiles to native machine code. The functions in the code below 5 take the derivative of the stress tensor and compute its values in each quadrature point.

```python
dsigma_d_deps = jax.jit(jax.jacrev(sig_jax, argnums=(0), has_aux=True))
sigma_vectorized = jax.jit(jax.vmap(sig_jax, in_axes=(0, 0, 0, 0)))
dsigma_d_deps_vectorized = jax.jit(jax.vmap(dsigma_d_deps, in_axes=(0, 0, 0, 0)))

def func_jax_sigma(deps, sigma_old, p_old, dp):
    deps_global = deps.reshape((num_cells*num_gauss_points, 4))
    sigma_old_global = sigma_old.reshape((num_cells*num_gauss_points, 4))
    out, dp_new = sigma_vectorized(deps_global, sigma_old_global, p_old, dp)
    np.copyto(dp, dp_new)
    return out.reshape(-1)

@jax.jit
def func_jax_C_tang(deps, sigma_old, p_old, dp):
    deps_global = deps.reshape((num_cells*num_gauss_points, 4))
    sigma_old_global = sigma_old.reshape((num_cells*num_gauss_points, 4))
    out, _ = dsigma_d_deps_vectorized(deps_global, sigma_old_global, p_old, dp)
    return out.reshape(-1)
```

Listing 5: Definition of the external operator and its derivative via JAX. The function `func_jax_sigma` computes and returns values of the stress tensor, where `func_jax_C_tang` computes the tangent stiffness tensor `C_tang`. The latter is computed with the help of the JAX automatic differentiation tools.

The three approaches successfully solve the cylinder expansion problem 1. Now we compare the three approaches by measuring the time that each spends on compilation of JIT-ed functions, so-called the time of compilation overhead, and total running time excluding compilation overhead. The results are shown in the Fig. 1.

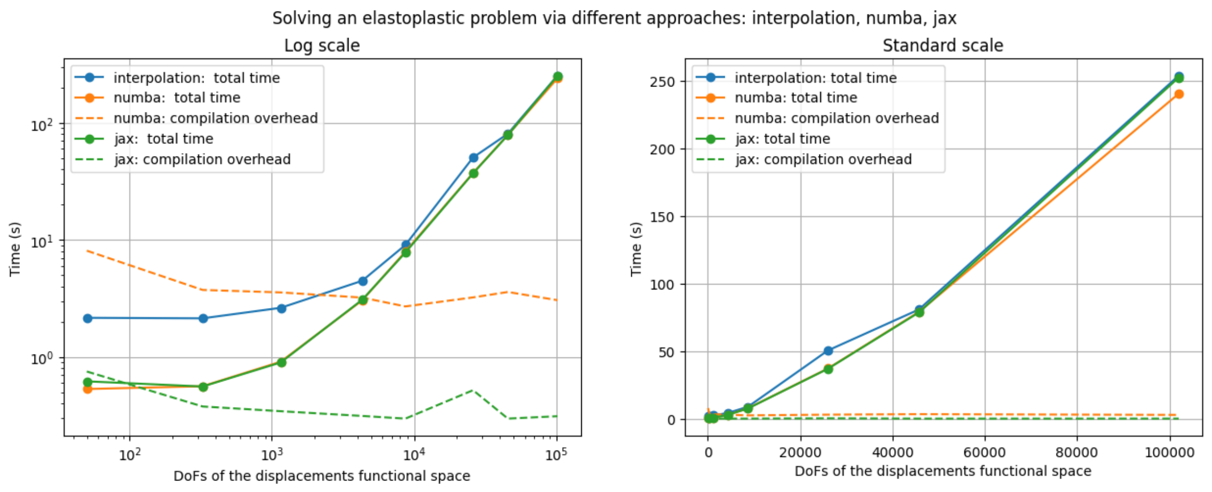

FIGURE 1 – Elapsed wall time (s) of different approaches for solving the elastoplastic problem of cylinder expansion against the dimension of finite element space. The same data is plotted on log-log (left) and standard (right) axes. The time spent on JIT compilation is denoted by "compilation overhead" and "run time" is the total time required for an approach to solve the problem excluding the compilation overhead.

Analysing Fig. 1 we conclude that the compilation overhead of the Numba and JAX approaches is constant on mesh refinement, which is the expected behaviour. Additionally, for this problem, we remark that JAX's JIT compilation is faster than the Numba's one.

In terms of overall timing for realistically sized problems ($> 10^4$ degrees of freedom) the three methods provide substantially similar run-time performance. We emphasise that the two `ExternalOperator`-based approaches match the performance of the previous state-of-the-art implementation proposed in [16]. Consequently, our approach provides a solid basis for future experimentation with more advanced constitutive models.

4 Conclusion

We have implemented a software framework in FEniCSx/DOLFINx that eases the implementation of finite element solvers for problems in solid mechanics involving constitutive models that cannot be straightforwardly expressed in variational form.

We showed a simple elastoplastic example implemented using three approaches, one an interpolation method originally proposed in [4] and then improved in [16] using modern FEniCSx features, and two new approaches using the `ExternalOperator` concept with operators implemented via JAX and Numba. The performance of all three implementations is substantially similar, suggesting that our approach is competitive with existing approaches, but with substantially greater flexibility in terms of the types of models that can be implemented.

In upcoming work, we plan to demonstrate the full potential of the framework for more complex problems including neural network constitutive models [17] and interfacing with external libraries such as CVXPY [6]. The framework will be released under an open-source license in due course.

5 Acknowledgment

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant reference PRIDE/21/16747448/MATHCODA. For the purpose of open access, and in fulfilment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

Références

[1] M.S. Alnaes et al. *The FEniCS Project Version 1.5*, Archive of Numerical Software, 2015.

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

[3] I.A. Baratta, J.P. Dean, J.S. Dokken, M.Habera, J.S. Hale, C.N. Richardson, M.E. Rognes, M.W. Scroggs, N. Sime, and G.N. Wells. *DOLFINx : The next generation FEniCS problem solving environment*, Zenodo, 2023.

[4] J. Bleyer, *Numerical Tours of Computational Mechanics with FEniCS*, Zenodo, 2018.

[5] M. Bonnet, A. Frangi, C. Rey. *The finite element method in solid mechanics*, Mc-Graw Hill Education, 2014.

[6] S. Diamond, S. Boyd. *CVXPY : A Python-embedded modeling language for convex optimization*, Journal of Machine Learning Research, 1-5, 2016.

[7] R. Frostig, J.J. Matthew, C. Leary. *Compiling machine learning programs via high-level tracing*, SySML, 2018.

[8] M. Habera et al. *FEniCSX : A sustainable future for the FEniCS Project*, SIAM Parallel Processing, 2020.

[9] D.A. Ham et al. *Firedrake User Manual*. Imperial College London and University of Oxford and Baylor University and University of Washington, 2023.

[10] D.A. Ham, N. Bouziani. *Escaping the abstraction : a foreign function interface for the Unified Form Language [UFL]*. First Workshop on Differentiable Programming (NeurIPS 2021), 2021.

[11] F. Hecht. *New development in FreeFem++*, Journal of numerical mathematics, 251-266, 2012.

[12] T. Helfer et al. *Introducing the open-source mfront code generator : Application to mechanical behaviours and material knowledge management within the PLEIADES fuel element modelling platform*, Computers & Mathematics with Applications, 2015.

[13] J. Korelc. *AceGen manual. Version 7.0*, University of Ljubljana, 2020.

[14] J. Korelc. *AceFEM manual. Version 7.0*, University of Ljubljana, 2020.

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

[16] A. Latyshev, J. Bleyer. *Master's thesis*, Finite-element implementation of plasticity using a convex optimization approach, Ecole des Ponts ParisTech, 2022.

[17] F. Masi, I. Stefanou, P. Vannucci, V. Maffi-Berthier. *Thermodynamics-based artificial neural networks for constitutive modeling*, Journal of the Mechanics and Physics of Solids, 2021.

[18] K.B. Ølgaard, G. N. Wells. *Software*, fenics-solid-mechanics, URL[Accessed : 2023-10-07] : https ://bitbucket.org/fenics-apps/fenics-solid-mechanics/src/master/.